Adobe® Flex™ 2

Building and Deploying Flex 2 Applications

© 2006 Adobe Systems Incorporated. All rights reserved.

Building and Deploying Flex™ 2 Applications

If this guide is distributed with software that includes an end-user agreement, this guide, as well as the software described in it, is furnished under license and may be used or copied only in accordance with the terms of such license. Except as permitted by any such license, no part of this guide may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, recording, or otherwise, without the prior written permission of Adobe Systems Incorporated. Please note that the content in this guide is protected under copyright law even if it is not distributed with software that includes an end-user license agreement.

The content of this guide is furnished for informational use only, is subject to change without notice, and should not be construed as a commitment by Adobe Systems Incorporated. Adobe Systems Incorporated assumes no responsibility or liability for any errors or inaccuracies that may appear in the informational content contained in this guide.

Please remember that existing artwork or images that you may want to include in your project may be protected under copyright law. The unauthorized incorporation of such material into your new work could be a violation of the rights of the copyright owner. Please be sure to obtain any permission required from the copyright owner.

Any references to company names in sample templates are for demonstration purposes only and are not intended to refer to any actual organization.

Adobe, the Adobe logo, Flex, Flex Builder and Flash Player are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States and/or other countries. ActiveX and Windows are either registered trademarks or trademarks of Microsoft Corporation in the United States and other countries. Linux is a registered trademark of Linus Torvalds. Macintosh is a trademark of Apple Computer, Inc., registered in the United States and other countries. Solaris is a registered trademark or trademark of Sun Microsystems, Inc. in the United States and other countries. UNIX is a registered trademark of The Open Group. All other trademarks are the property of their respective owners.

This product includes software developed by the Apache Software Foundation (http://www.apache.org/). Macromedia Flash 8 video is powered by On2 TrueMotion video technology. © 1992-2005 On2 Technologies, Inc. All Rights Reserved. http://www.on2.com. This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/). Portions licensed from Nellymoser (www.nellymoser.com). Portions utilize Microsoft Windows Media Technologies. Copyright (c) 1999-2002 Microsoft Corporation. All Rights Reserved. Includes DVD creation technology used under license from Sonic Solutions. Copyright 1996-2005 Sonic Solutions. All Rights Reserved. This Product includes code licensed from RSA Data Security. Portions copyright Right Hemisphere, Inc. This product includes software developed by the OpenSymphony Group (http://www.opensymphony.com/).

Sorenson Spark

Sorenson™ Spark™ video compression and decompression technology licensed from Sorenson Media, Inc.

Adobe Systems Incorporated, 345 Park Avenue, San Jose, California 95110, USA

Notice to U.S. government end users. The software and documentation are “Commercial Items,” as that term is defined at 48 C.F.R. §2.101, consisting of “Commercial Computer Software” and “Commercial Computer Software Documentation,” as such terms are used in 48 C.F.R. §12.212 or 48 C.F.R. §227.7202, as applicable. Consistent with 48 C.F.R. §12.212 or 48 C.F.R. §§227.7202-1 through 227.7202-4, as applicable, the Commercial Computer Software and Commercial Computer Software Documentation are being licensed to U.S. Government end users (a) only as Commercial items and (b) with only those rights as are granted to all other end users pursuant to the terms and conditions herein. Unpublished-rights reserved under the copyright laws of the United States. Adobe Systems Incorporated, 345 Park Avenue, San Jose, CA 95110-2704, USA. For U.S. Government End Users, Adobe agrees to comply with all applicable equal opportunity laws including, if appropriate, the provisions of Executive Order 11246, as amended, Section 402 of the Vietnam Era Veterans Readjustment Assistance Act of 1974 (38 USC 4212), and Section 503 of the Rehabilitation Act of 1973, as amended, and the regulations at 41 CFR Parts 60-1 through 60-60, 60-250 ,and 60-741. The affirmative action clause and regulations contained in the preceding sentence shall be incorporated by reference.

Part Number: 90069415 (12/06)

Contents

Chapter 1: About Flex Documentation 7

PART 1: BUILDING AND DEPLOYING OVERVIEW

Chapter 2: Flex Application Development 13

About building and deploying applications 13
About building applications for Flex 2 SDK 23
About building applications for Flex Data Services 26

Chapter 3: Flex Application Structure 31

Installation directory structure 31
Development directory structure 34
Compiling an application. 46
Deployment directory structure. 48

Chapter 4: Applying Flex Security 51

Introduction 51
Loading assets 64
Using J2EE authentication. 67
Using RPC services 70
Using data services 72
Making other connections 73
Using SSL 74
Writing secure Flex applications. 76
Configuring client security settings. 82
Other resources 85

Chapter 5: Optimizing Flex Applications .87

About performance 88
Improving client-side performance. 88
Improving server-side performance 123
Improving Flex Charting component performance 127

Chapter 6: Improving Startup Performance 131

About startup performance 131
About startup order 132
Using deferred creation 134
Creating deferred components. 138
Using ordered creation 143
Using the callLater() method. 150

Chapter 7: Building Overview 153

About the Flex development tools 153
About application files 156

PART 2: BUILDING FLEX APPLICATIONS

Chapter 8: Flex 2 SDK and Flex Data Services Configuration . 161

About configuration files 161
Flex 2 SDK configuration 165
Flex Data Services configuration. 168
Flash Player configuration 178

Chapter 9: Using the Flex Compilers 179

About the Flex compilers 179
About the command-line compilers 187
About configuration files 190
About option precedence 194
Using the application compiler 195
Using the component compiler. 215
Viewing errors and warnings. 227
About SWC files 229
About manifest files. 231

Chapter 10: Using Runtime Shared Libraries 233

About RSLs 233
Creating libraries 238
Using RSLs. 240
RSL example 241

Chapter 11: Logging 245

About logging 245
Using the debugger version of Flash Player 247

Client-side logging and debugging 251
Compiler logging 264
Web-tier logging 265

Chapter 12: Using the Command-Line Debugger 269

About debugging 269
Invoking the command-line debugger 272
Configuring the command-line debugger 276
Using the command-line debugger commands 277

Chapter 13: Using ASDoc 289

About the ASDoc tool 289
Creating ASDoc comments 290
Documenting ActionScript elements 295
Documenting MXML files 301
ASDoc tags 301
Running the ASDoc tool 307

Chapter 14: Creating Applications for Testing 311

Tasks and techniques for testable applications overview 311
Compiling applications for testing 312
Creating testable applications 317
Writing the wrapper 320
Understanding the automation framework 321
Instrumenting events 324
Instrumenting custom components 328
Instrumenting composite components 337
Example: Instrumenting the RandomWalk custom component . . . 339

PART 3: DEPLOYING FLEX APPLICATIONS

Chapter 15: Deploying Flex Applications 351

About deploying an application 351
Deployment options 352
Compiling for deployment 358
Deployment checklist 360

Chapter 16: Creating a Wrapper 367

About the wrapper 367
Creating a wrapper 370

Adding features to the wrapper 375
About the <object> and <embed> tags 378
Requesting an MXML file without the wrapper. 389

Chapter 17: Using Express Install 391

About Express Install 391
Editing your wrapper. 392
Configuring Express Install on Flex Data Services 396
Alternatives to Express Install. 397

PART 4: CONFIGURING JRUN

Chapter 18: Configuring JRun. 401

About JRun application servers 401
Starting and stopping JRun servers 404
Adding and removing servers 405
Configuring JRun servers 406
Using the sniffer 411

Index 415

CHAPTER 1

About Flex Documentation

1

Building and Deploying Flex 2 Applications describes the process of building and deploying Adobe® Flex™ 2 applications. If you are using Adobe® Flex™ Data Services, this manual also includes a section that describes how to configure the integrated Adobe® JRun™ Application Server.

Contents

Using this manual 7
Accessing the Flex documentation 8

Using this manual

This manual can help anyone who is developing Flex applications. However, this book is most useful if you have basic experience using Flex, or have read *Getting Started with Flex 2*. *Getting Started with Flex 2* provides an introduction to Adobe® Flex™ 2 and helps you develop the basic knowledge that makes using this manual easier.

Building and Deploying Flex 2 Applications is divided into the following parts:

Part	Description
Part 1, "Building and Deploying Overview"	Describes the basic process for building and deploying Flex applications.
Part 2, "Building Flex Applications"	Describes how to compile, debug, and text Flex applications.
Part 3, "Deploying Flex Applications"	Describes how to deploy Flex applications.
Part 4, "Configuring JRun"	Describes how to configure the integrated JRun Application Server for Flex Data Services.

Accessing the Flex documentation

The Flex documentation is designed to provide support for the complete spectrum of participants.

Documentation set

The Flex documentation set includes the following titles:

Book	Description
Getting Started with Flex 2	Contains an overview of Flex features and application development procedures.
Flex 2 Developer's Guide	Describes how to develop your dynamic web applications.
Building and Deploying Flex 2 Applications	Describes how to build and deploy Flex applications.
Creating and Extending Flex 2 Components	Describes how to create and extend Flex components.
Migrating Applications to Flex 2	Provides an overview of the migration process, as well as detailed descriptions of changes in Flex and ActionScript.
Using Flex Builder 2	Contains comprehensive information about all Adobe® Flex™ Builder™ 2 features, for every level of Flex Builder users.
Adobe Flex 2 Language Reference	Provides descriptions, syntax, usage, and code examples for the Flex API.

Viewing online documentation

All Flex documentation is available online in HTML and Adobe® Portable Document Format (PDF) files from the Adobe website. It is also available from the Adobe® Flex™ Builder™ Help menu.

Beta Beta Beta Beta Beta Beta Beta Beta Beta Beta

Typographical conventions

The following typographical conventions are used in this book:

- *Italic font* indicates a value that should be replaced (for example, in a folder path).
- `Code font` indicates code.
- *`Code font italic`* indicates a parameter.
- **Boldface font** indicates a verbatim entry.

Beta Beta Beta Beta Beta Beta Beta Beta Beta Beta

PART 1

Building and Deploying Overview

1

This part describes the basic process for building and deploying Flex applications.

The following topics are included:

Chapter 2: Flex Application Development 13
Chapter 3: Flex Application Structure 31
Chapter 4: Applying Flex Security 51
Chapter 5: Optimizing Flex Applications. 87
Chapter 6: Improving Startup Performance 131

CHAPTER 2

Flex Application Development

2

You can divide the process of creating a Flex application into five phases: design, configure, build, deploy, and secure. This topic contains an overview of these phases, and describes these phases for both Adobe Flex 2 SDK and for Flex Data Services.

Contents

About building and deploying applications 13
About building applications for Flex 2 SDK 23
About building applications for Flex Data Services 26

About building and deploying applications

It is difficult to define the exact process that all Flex developers use to build and deploy applications. However, the process typically involves five distinct phases:

1. Design
2. Configure
3. Build
4. Deploy
5. Secure

The following sections contain an overview of these general phases of development, and describe how each phase applies to the development of an application for Flex 2 SDK and for Flex Data Services.

Design phase

In the design phase, you make basic decisions about how to write code for reusability, how your application interacts with its environment, how your application accesses application resources, and many other decisions. In the design phase, also define your development and deployment environments, including the directory structure of your application.

Although these design decisions specify how your application interacts with its environment, you also have architectural issues to decide. For example, you might choose to develop your application based on a particular design pattern, such as Model-View-Controller (MVC).

About design patterns

One common starting point of the design phase is to identify one or more design patterns relevant for your application. A design pattern describes a solution to a common programming problem or scenario. Although the design pattern might give you insight into how to approach an application design, it does not necessarily define how to write code for that solution.

Many types of design patterns have been catalogued and documented. For example, the Functional design pattern specifies that each module of your application performs a single action, with little or no side effects for the other modules in your application. The design pattern does not specify what a module is, commonly though it corresponds to a class or method.

About MVC

The goal of the Model-View-Controller (MVC) architecture is that by creating components with a well-defined and limited scope in your application, you increase the reusability of the components and improve the maintainability of the overall system. Using the MVC architecture, you can partition your system into three categories of components:

Model components Encapsulates data and behaviors related to the data processed by the application. The model might represent an address, the contents of a shopping cart, or the description of a product.

View components Defines your application's user interface, and the user's view of application data. The view might contain a form for entering an address, a DataGrid control for showing the contents of a shopping cart, or an image of a product.

Controller components Handles data interconnectivity in your application. The Controller provides application management and the business logic of the application. The Controller does not necessarily have any knowledge of the View or the Model.

For example, with the MVC design, you could implement a data-entry form that has three distinct pieces:

- The model consists of XML data files or the remote data service calls to hold the form data.
- The view is the presentation of any data and display of all user interface elements.
- The controller contains logic that manipulates the model and sends the model to the view.

The promise of the MVC architecture is that by creating components with a well-defined and limited scope, you increase the reusability of these components and improve the maintainability of the overall system. In addition, you can modify components without affecting the entire system.

Although you can consider a Flex application as part of the View in a distributed MVC architecture, you can use Flex to implement the entire MVC architecture on the client. A Flex application has its own view components that define the user interface, model components that represent data, and controller components that communicate with back-end systems.

About Struts

Struts is an open-source framework that facilitates the development of web applications based on Java servlets and other related technologies. Because it provides a solution to many of the common problems that developers face when building these applications, Struts has been widely adopted in a large variety of development efforts, from small projects to large-scale enterprise applications.

Struts is based on a Model-View-Controller (MVC) architecture, with a focus on the controller part of the MVC architecture. In addition, it provides JSP tag libraries to help you create the view in a traditional JSP/HTML environment.

Configure phase

Before you write your first line of application code, or before you deploy an application, you must ensure that you configure your environment correctly. Configuration is a broad term and encompasses several different tasks.

For example, you must configure your development and deployment environments to ensure that your application can access the required resources and data services. If your application requires access to a web service, ensure that your application has the correct access rights to the web service. If you application runs outside a firewall, ensure that it can access resources inside the firewall.

Before you can begin to develop an application for Flex Data Services, you must first deploy the Flex Data Services web application on your J2EE application server or servlet container. As part of this configuration, you define how the application references the services provides by Flex Data Services.

The following sections contain an overview of configuration tasks.

About run-time configuration

Most run-time configuration has to do with configuring access to remote data services, such as web services. For example, during application development, you run your application behind a firewall, where the application has access to all necessary resources and data services. However, when you deploy the application, you must ensure that an executing application can still access the necessary resources when the application runs outside of the firewall.

One configuration issue for Flex 2 SDK applications is the placement of a crossdomain.xml file. For security, by default Flash Player does not allow an application to access a remote data service from a domain other than the domain from which the application was served. Therefore, a server that hosts a data service must be in the same domain as the server hosting your application, or the remote server must define a crossdomain.xml file. A crossdomain.xml file is an XML file that provides a way for a server to indicate that its data and documents are available to SWF files served from specific domains, or from all domains. By default, place the crossdomain.xml at the root directory of the server that is serving the data.

Flex 2 SDK does not include a server-side proxy for handling data service requests. Therefore, you must ensure that you configure data services for direct access by your application, or make data service requests through your own proxy server. Note that Flex Data Services does include an integrated server-side proxy.

About Flex Data Services configuration

When using Flex Data Services, you must perform additional configuration beyond what is required for a Flex 2 SDK application. Flex Data Services contains many services that you configure in preparation for building and deploying a Flex Data Services application. For example, you might have to configure the Flex Proxy Service, Remoting Service, Message Service, or Data Management Service. The primary mechanism for configuring Flex Data Services are configuration files under the WEB-INF/flex directory of the Flex Data Services web application.

Build phase

Building your application is an iterative process that includes three main tasks:

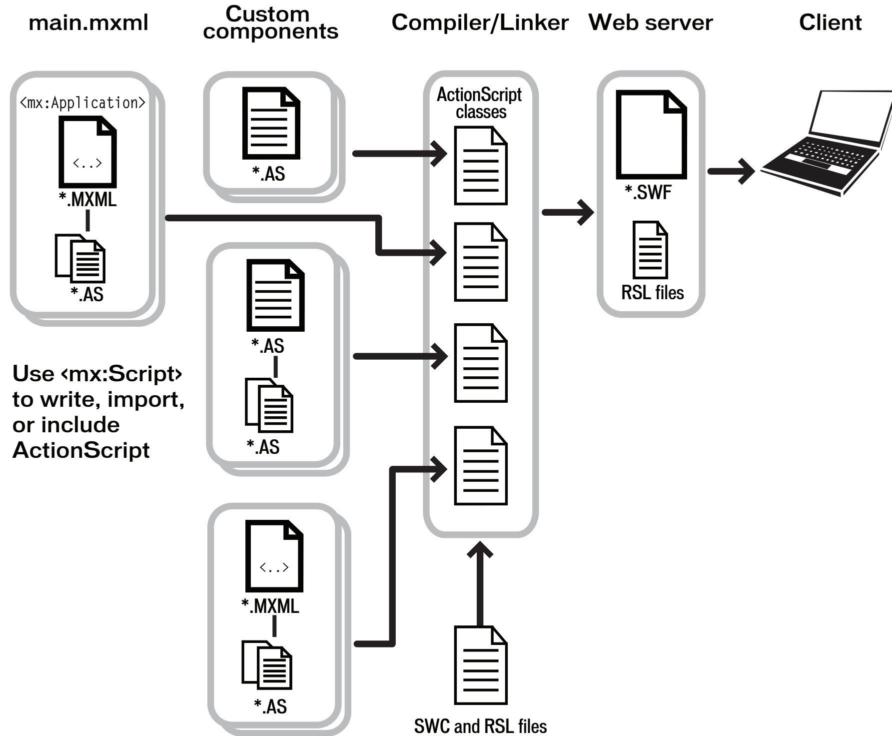
1. Compile
2. Debug
3. Test

This section contains an overview of these three tasks.

About compiling

Compiling your application converts your application files and assets into a single SWF file. During compilation, you set compiler options to enable accessibility, enable debug information in the output, set library paths, and set other options. You can configure the compiler as part of configuring your project in Flex Builder, by using command-line arguments to the compiler, or by setting options in a configuration file.

When you compile your application, the Flex compiler creates a single SWF file from all of the application files (Adobe® MXML™, AS, RSL, SWC, and asset files), as the following example shows:



Flex provides two compilers: mxmlc and compc. You can use the compc and mxmlc compilers from within Flex Builder or from a command line. If you have Flex Data Services, you can open the mxmlc compiler in response to an HTTP request to an MXML file.

You use mxmlc to compile MXML, ActionScript, SWC, and RSL files into a single SWF file. After your application is compiled and deployed on your web or application server, a user can make an HTTP request to download and play the SWF file on their computer.

You use compc to create resources that you use to create the application. For example, you can compile components, classes, and other files into SWC files or into RSLs, and then statically or dynamically link these libraries to your application.

For more information, see Chapter 9, “Using the Flex Compilers,” on page 179.

About debugging an application

Flex provides several tools that you use to debug your application, including the following:

You can run Flex applications in two different versions of Adobe® Flash® Player 9: the standard version, which the general public uses, and the debugger version, which application developers use to debug their applications during the development process.

Flash Player You can run Flex applications in two different versions of Flash Player: the standard version, which the general public uses, and the debugger version, which application developers use to debug their applications during the development process.

Flex Builder visual debugger The Flex Builder debugger allows you to run and debug applications. You can use the debugger to set and manage breakpoints; control application execution by suspending, resuming, and terminating the application; step into and over the code; watch variables; evaluate expressions; and so on.

Flex Command-line debugger A command line version of the debugger that you can use outside of Flex Builder.

For more information, see Chapter 12, "Using the Command-Line Debugger," on page 269.

About testing an application

Due to the size, complexity, and large amounts of data handled by applications, maintaining the quality of a large software application can be difficult. To help with this task, you can use automated testing tools that test and validate application behavior without human intervention.

Deploy phase

When you deploy your application, you make it available to customers. Typically, you deploy the application as a SWF file on a web server so that users can access it by using an HTTP request to the SWF file.

When you deploy the application's SWF file, you must also deploy all of the assets required by the application. For example, if the application requires access to video or image files, or to XML data files, you must make sure to deploy those assets as well. If the application uses an RSL, you must also deploy the RSL.

Deploying assets may not necessarily be as simple as copying the assets to a location on your web server. Flash Player has built-in security features that controls the access of application assets at run time.

This section contains an overview of the deployment phase. For more information, see Chapter 15, "Deploying Flex Applications," on page 351.

What happens during a request to a SWF file

When a customer requests the SWF file, the web server or application server returns the SWF file to the client computer. The SWF file then runs locally on the client.

In some cases, a request to a Flex SWF file can cause multiple requests to multiple SWF files. For example, if your application uses Runtime Shared Libraries (RSLs), the web server or application server returns an RSL as a SWC file to the client along with the application SWF file. If your application uses the Flex history manager, a request to the application returns the application SWF file and the history manager's SWF file.

Server-side caching

Your web server or application server typically caches the SWF file on the first request, and then serves the cached file on subsequent requests. You configure server-side caching by using the options available in your web server or application server.

Client-side caching

The SWF file returned to the client is typically cached by the customer's browser on first request. Depending on the browser configuration, the SWF file typically remains in the cache until the browser closes. When the browser reopens, the next request to the SWF file must reload it from the server.

Integrating Flex applications with your web application

To incorporate a Flex application into a website, you typically embed the SWF file in an HTML, JSP, Adobe® ColdFusion®, or other type of web page. The page that embeds the SWF file is known as the *wrapper*.

A wrapper consists of an `<object>` tag and an `<embed>` tag that format the SWF file on the page, define data object locations, and pass run-time variables to the SWF file. In addition, the wrapper can include support for history management and Flash Player version detection and deployment.

When you compile an application with Flex Builder, it automatically creates a wrapper file for you in the bin directory associated with the Flex Builder project. You can copy the contents of the wrapper file into your HTML pages to reference the SWF file. The Flex Data Services compiler also generates the wrapper for you.

You can edit the wrapper to manipulate how Flex appears in the browser. You can also add JavaScript or other logic in the page to communicate with Flex or generate customized pages.

When using the mxmlc command-line compiler, you must write the wrapper yourself. For more information, see Chapter 16, "Creating a Wrapper," on page 367.

Secure phase

Security is not necessarily a phase of the application development process, but is an issue that you should take into consideration during the entire development process. That is, you do not configure, build, test, and deploy an application, and then define the security issues. Rather, you take security into consideration during all phases.

Building security into your application often takes two main efforts:

- Using the security features built into Flash Player
- Building security into your application
- Securing Flex Data Services

Flash Player has several security features built into it, including sandbox security, that you can take advantage of because you are building applications for Flash Player.

But, Flash Player security is not enough for many application requirements. For example, your application may require the user to log in, or perform authentication in some other way, before accessing data services. When you must handle security issues beyond those built into Flash Player, design them into your application from the initial design phase, test them during the compile phase, and verify them during the deploy phase.

For more information on security, see Chapter 4, “Applying Flex Security,” on page 51.

About the security model

The Flex security model protects both the client and the server. Consider the following general aspects of security when you deploy Flex applications:

- Flash Player operating in a sandbox on the client
- Authorizing and authenticating users who access a server’s resources

Flash Player runs inside a security sandbox that prevents the client from being hijacked by malicious application code. This sandbox prevents a user from running a Flex application that can access system files and perform other tasks.

Flex Data Services is a J2EE application, so it supports working with the web application security of any J2EE application server. You use J2EE authentication and authorization techniques to prevent unauthorized users from accessing your applications. The Flex Data Services application includes several built-in security mechanisms that let you control access to web services, HTTP services, and Java classes such as EJBs.

Flash Player security

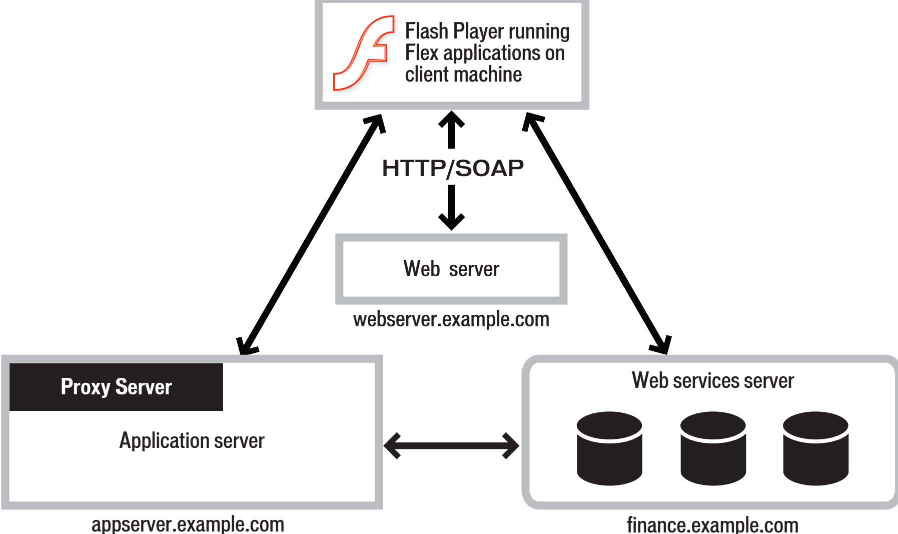
Flash Player has an extensive list of features that ensure Flash content is secure, including the following:

- Uses the encryption capabilities of SSL in the browser to encrypt all communications between a Flash application and the server
- Includes an extensive sandbox security system that limits transfer of information that might pose a risk to security or privacy
- Does not allow applications to read data from the local drive, except for SharedObjects that were created by that domain
- Does not allow writing any data to the disk except for data that is encapsulated in SharedObjects
- Does not allow web content to read any data from a server that is not from the same domain, unless that server explicitly allows access
- Enables the user to disable the storage of information for any domain
- Does not allow data to be sent from a camera or microphone unless the user gives permission

About building applications for Flex 2 SDK

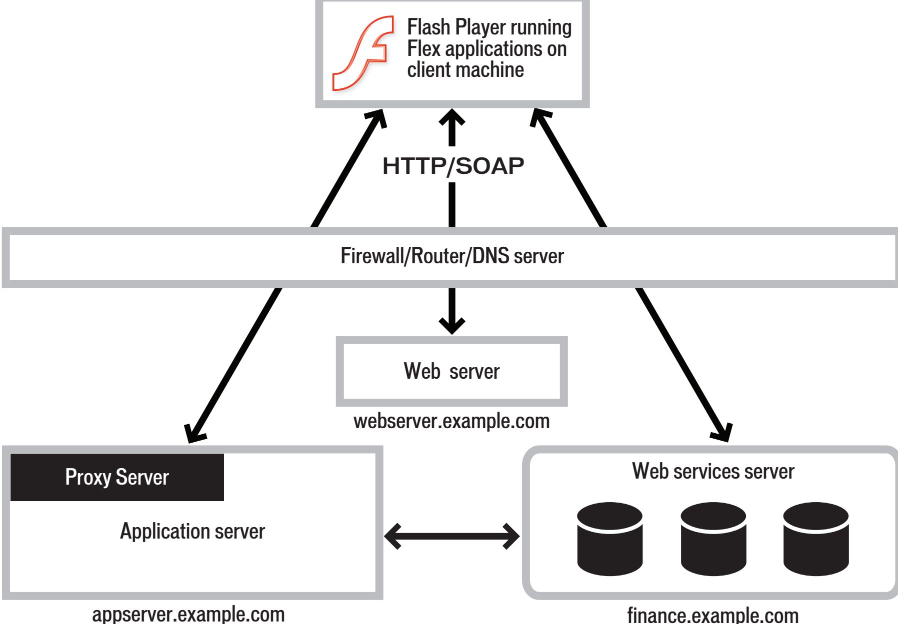
The section “About building and deploying applications” on page 13 described the five basic phases that you use to develop a Flex application: design, configure, build, deploy, and secure. This section describes the five phases of development specifically for Flex 2 SDK.

The following example shows a typical development environment for a Flex 2 SDK application:



In this example, application development happens in an environment that is behind a firewall, and you deploy your application SWF file on webserver.example.com. To run the application, you make a request to it from a computer that is also within the firewall. The executing SWF file can access resources on any other server as necessary. In the development environment, the SWF file can directly access web services, or it can access them through a proxy server.

The following example shows a typical deployment environment for a Flex 2 SDK application:



In this example, the customer requests the application SWF file from webserver.example.com, the server returns the SWF file to the customer, and the SWF file plays. The executing SWF file must be able to access the necessary resources from outside the firewall.

Design phase

With Flex 2 SDK, one of your first design decisions might be to choose a design pattern that fits your application requirements. That design pattern might have implications on how you structure your development environment, determine the external data services that your application must access, and define how you integrate your Flex application into a larger web application.

Configure phase

For run-time configuration, you ensure that your executing SWF file can access the necessary resources including asset files (such as image files) and external data services. If you access a resource on a domain other than the domain from which the SWF file is served, you must define a crossdomain.xml file on the target server, or make the request through a proxy server.

Build phase

To build an application for Flex 2 SDK, you define a directory structure on your development system for application files, and define the location of application assets. You then compile, debug, and test your application.

The compile-time configuration for a Flex 2 SDK application is primarily a process of setting compiler options to define the location of SWC and RSLs, to create a SWF file with debug information, or to set additional compiler options. When compiling applications, you compile your application into a single SWF file, and then deploy the SWF file to a web server or application server for testing.

Deploy phase

With Flex 2 SDK, you deploy your application SWF file on your web server or application server. Users then access the deployed SWF file by making an HTTP request in the form:

`http://`*`hostname`*`/`*`path`*`/filename.swf`

If you embed your SWF file in an HTML or other type of web page using a wrapper, users request the wrapper page. The request to the wrapper page causes the web server or application server to return the SWF file along with the wrapper page.

Secure phase

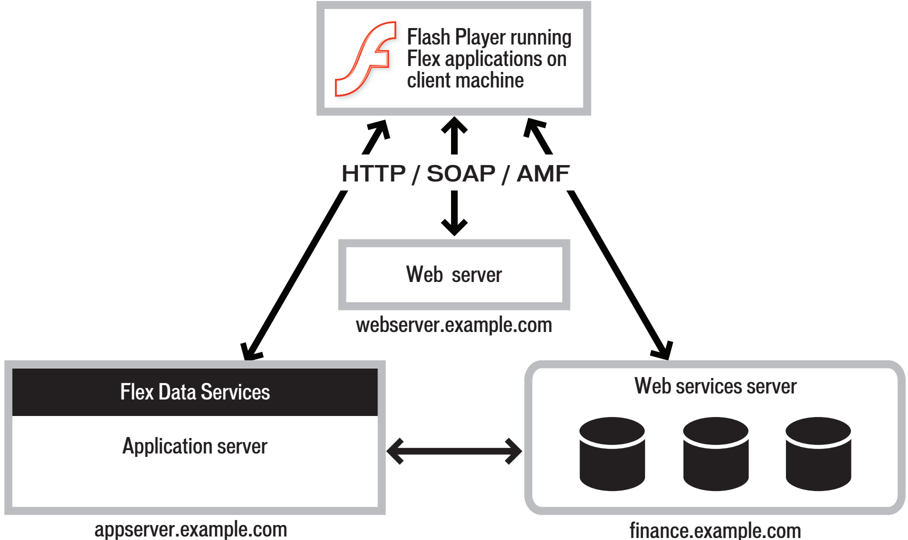
Security issues for Flex 2 SDK applications often have to do with how the application accesses external resources. For example, you might require a user to log in to access resources, or you might want the application to be able to access external data services that implement some other form of access control.

About building applications for Flex Data Services

Flex Data Services includes all of the functionality of Flex 2 SDK, and adds the Flex Message Service, the RPC services, and the Flex Data Management Service. When you develop applications using Flex Data Services, you perform many of the same actions that you do when building applications for Flex 2 SDK. For more information on Flex 2 SDK, see "About building applications for Flex 2 SDK" on page 23.

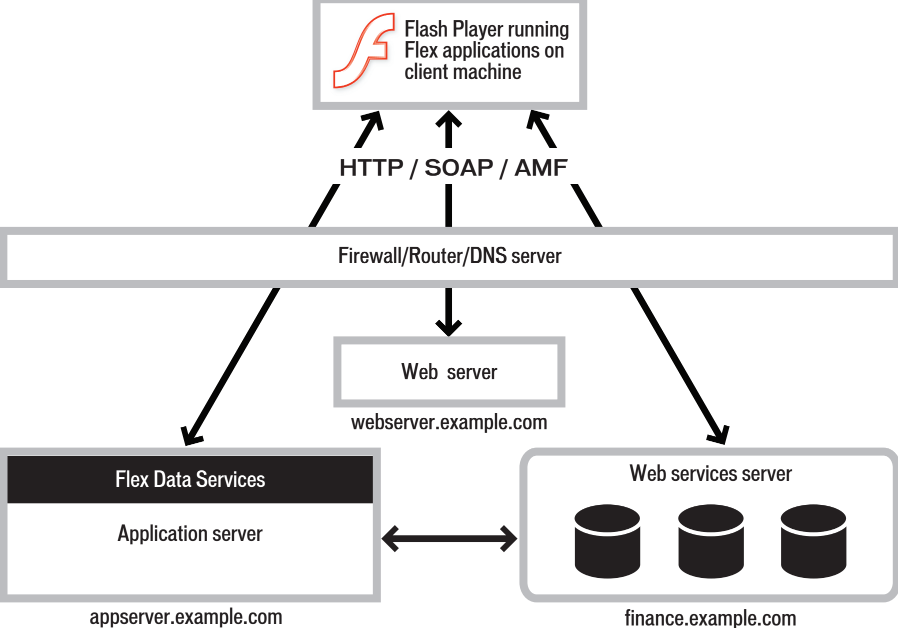
Because of the added functionality of Flex Data Services, you also perform additional tasks as part of developing an application. This section describes the five phases of development specifically for Flex Data Services.

The following example shows a typical development environment of a Flex Data Services application:



Notice that this example shows the Flex Data Services application making a request to appserver.example.com, the J2EE server of servlet container hosting Flex Data Services. Since Flex Data Services contains a proxy server so you can use it to proxy requests for external resources. This example also shows Flex Data Services application using the AMF protocol, which is not available with the Flex 2 SDK.

The following example shows a typical deployment environment of a Flex Data Services application:



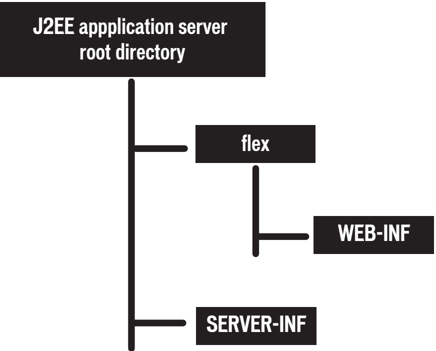
Before using Flex Data Services

Before you can begin developing an application for Flex Data Services, you must deploy the Flex Data Services web application on your development server.

When you install Flex Data Services, you are given the option of installing the integrated JRun 4 J2EE application server to host Flex Data Services. Alternatively, you can install Flex Data Services without installing JRun4. In this case, you deploy the Flex Data Services web application on your J2EE application server or servlet container before you begin to develop your application.

NOTE Flex Data Services includes JRun's web server as its default web server. This web server is not meant to be used as a production server. It is for development purposes only.

The following example shows the directory structure of the deployed web application for Flex Data Services:



To access the services provided by Flex Data Services, you only need to know the URL and port number associated with the context root directory of the deployed web application. For example, if you install Flex Data Services with the integrated JRun4 application server, the context root has the following URL:

```
http://localhost:8700/flex/
```

Design phase

The design phase for a Flex Data Services application can be similar to the process for a Flex 2 SDK application. However, Flex Data Services applications are typically larger and more complex than applications developed using Flex 2 SDK, and require a more complex design.

Much of the design complexity of a Flex Data Services application has to do with using its remote data services. For example, one characteristic of a Flex Data Services application is its ability to use the Flex Messaging service to enable participation in Java Message Service (JMS) messaging. Or, your application might be required to send messages to a ColdFusion Component (CFC) page using the ColdFusion Event Gateway Adapter.

Because of the complexities of Flex Data Services applications, the design phase of application development can take longer and be more involved than for a Flex 2 SDK application.

Configure phase

The configure phase for a Flex Data Services application includes the same tasks as for Flex 2 SDK, plus the additional configuration steps for the services provided by Flex Data Services. After deploying the Flex Data Services web application, you must configure the data services, and other services, for your development and deployment environments.

Build phase

When developing applications with Flex Data Services, you can compile them in the same way that you compile them using Flex 2 SDK. That means you can use the compiler built into Flex Builder, or the command-line compiler supplied with Flex 2 SDK. In this scenario, your source code and other application assets are typically stored in a directory structure outside of your J2EE application server or servlet container. To run the application, you copy the compiled SWF file to your J2EE application server or servlet container.

You can also decide to develop your application in the directory structure of the Flex Data Services web application deployed on your J2EE server or servlet container, or in a separate web application directory structure. In this scenario, you use the integrated web compiler built into Flex Data Services to compile your application.

The web compiler compiles your application in response to an HTTP request to the main MXML file in your application, meaning the MXML file that contains the `<mx:Application>` tag. To do so, you deploy your application as MXML, ActionScript, and SWC files under the web root directory of your application server. When a user requests the main MXML file for the first time, the request triggers the compilation. After the initial compilation, the resultant SWF file is cached by Flex Data Services and returned in response to future requests.

Deploy phase

You have two options for how to deploy your Flex Data Services application: you can compile and deploy a single SWF file, or you can deploy your application as a collection of MXML and ActionScript files. In the second scenario, you use the integrated web compiler built into Flex Data Services to compile your application upon an HTTP request. In most cases, you deploy your application as a SWF file so that you are not putting source code on your deployment server.

After you decide how to deploy your application, you can package your application as a J2EE web application, meaning as a WAR file, and deploy the WAR fie on your J2EE application server or servlet container.

Secure phase

The increased design complexity of Flex Data Services applications over Flex 2 SDK applications means that you also have increased security issues. Much of the functionality of Flex Data Services is concerned with accessing remote data services, and you have to consider security whenever you allow an application to access remote data.

For example, Flex Data Services transports messages to and from service destinations over message channels that are part of the Flex messaging system. You can restrict access to a privileged group of users by applying a security constraint in a destination definition in the Flex services configuration file. A security constraint ensures that a user is authenticated, by using custom or basic authentication, before accessing the destination.

For more information on security, see Chapter 4, "Applying Flex Security," on page 51.

CHAPTER 3

Flex Application Structure

3

One of your first tasks when developing a Flex application is to set up your development directory structure. As part of setting up this directory structure, you must decide how to organize your application assets, how to share assets across applications, and how to configure the compiler to create your application SWF file.

This topic describes the Flex installation, development, and deployment directory structures, and how to use options to the Flex compiler to create your application SWF file based on the directory structure.

Contents

Installation directory structure 31
Development directory structure 34
Compiling an application. 46
Deployment directory structure. 48

Installation directory structure

Before you can begin to set up your application development environment, be familiar with the Flex installation directory structure. This section describes the installation directory structure for the following:

- Flex 2 SDK
- Flex Builder
- Flex Data Services
- Flex Charting components

Flex 2 SDK installation directory structure

When you install Flex 2 SDK, the installer creates the following directory structure under the installation directory:

Directory	Description
/bin	Contains the executable files, such as the mxmlc and compc compilers.
/frameworks	Contains configuration files, such as flex-config.xml and default.css.
/frameworks/libs	Contains the library SWC files. You use the files to compile your application.
/frameworks/locale	Contains the localization resource files.
/frameworks/source	Contains the Flex framework source code.
/frameworks/themes	Contains the theme files that define the basic look and feel of all Flex components.
/lib	Contains JAR files.
/player	Contains the different versions of Flash Player—the standard version and the debugger version.
/resources	Contains template HTML wrapper files.
/samples	Contains sample applications.

Flex Builder installation directory structure

Wen you install Flex Builder, you install Flex 2 SDK plus Flex Builder. The installer creates the following directory structure:

Directory	Description
Flex Builder 2	The top-level directory for Flex Builder.
/configuration	A standard Eclipse folder that contains the config.ini file and error logs.
/features	A standard Eclipse folder that contains the plug-ins corresponding to features of Flex Builder.
/Flex SDK 2	Contains the Flex 2 SDK, except for the Player directory. For a directory description, see "Flex 2 SDK installation directory structure" on page 32.

Directory	Description
/jre	Contains the Java Runtime Environment installed with Flex Builder used by default when you run the stand-alone version of Flex Builder.
/Player	Contains the different versions of Flash Player–the standard version and the debugger version.
/plugins	Contains the Eclipse plugins used by Flex Builder.

Flex Data Services installation directory structure

When you install Flex Data Services, the installer creates the following directory structure:

Directory	Description
fds2	The top-level directory for Flex Data Services. This directory contains the WAR files that you use to deploy Flex Data Services on your J2EE server.
/flex_sdk_2/bin	Contains the executable files, such as the mxmlc and compc compilers.
/flex_sdk_2/frameworks	Contains configuration files, such as flex-config.xml and default.css.
/flex_sdk_2/frameworks/libs	Contains the library SWC files. You use the files to compile your application.
/flex_sdk_2/frameworks/locale	Contains Flex SDK localization resource files.
/flex_sdk_2/frameworks/locale-fds	Contains Flex Data Services localization resource files.
/flex_sdk_2/frameworks/source	Contains the Flex framework source code.
/flex_sdk_2/frameworks/themes	Contains the Flex theme files that define the basic look and feel of all Flex components.
/flex_sdk_2/lib	Contains the Flex Data Services JAR files.
/jrun4	If you choose to install JRun 4 with Flex Data Services, the directory that contains the JRun 4 application files.
/resources	Subdirectories contain the different versions of Flash Player (the standard version and the debugger version), templates for creating wrappers, configuration files, and security resources and assets.

Flex Charting Components installation directory structure

Flex Charting components are a set of components that you add to an existing installation of the Flex 2 SDK, Flex Builder, or Flex Data Services.

When you install Flex Charting components, the installer automatically copies the necessary files, including charts.swc, to your Flex installation. If you choose not to copy the necessary files to your Flex installation during installation, you must perform that copy yourself. For more information, see the installation instructions.

Development directory structure

This section describes how to set up the directory structure of your development environment. As part of this process, you define the directory location for application-specific assets, assets shared across applications, and the location of other application files and assets.

Flex file types

A Flex application consists of many different file types. The following sections describes the options that you must consider when deciding where to place each type of file.

The following table describes the different Flex file types:

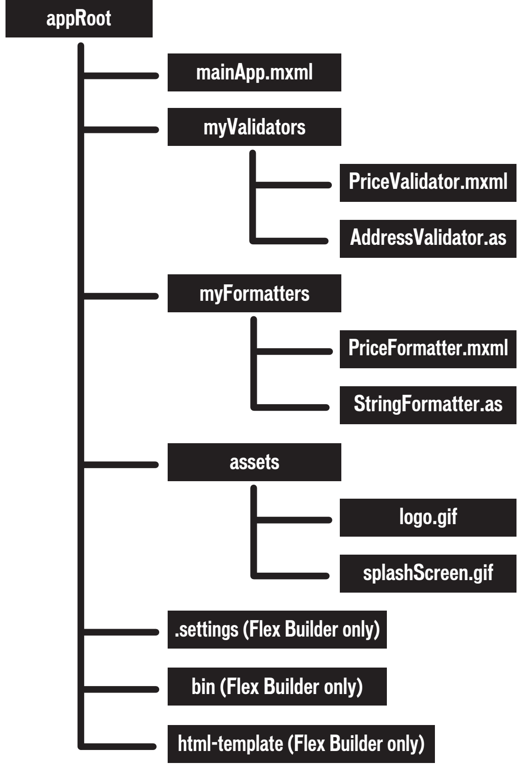
File format	Extension	Description
MXML	.mxml	Your application typically has one main application MXML file that contains the `<mx:Application>` tag, and one or more MXML files that implement your custom MXML components.
ActionScript	.as	A utility class, Flex custom component class, or other logic implemented as an ActionScript file.
SWC	.swc	A custom library file, or a custom component implemented as an MXML or ActionScript file, then packaged as a SWC file. A SWC file contains components that you package and reuse among multiple applications. The SWC file is then statically linked into your application at compile time when you create the application's SWF file.

File format	Extension	Description
RSL	.swc	A custom library implemented as an MXML or ActionScript file, and then deployed as a Runtime Shared Library (RSL). An RSL is a stand-alone SWC file that is downloaded separately from your application's SWF file, cached on the client computer for use with multiple application SWF files, and dynamically linked to your application.
CSS file	.css	A text file template for creating a Cascading Style Sheets file.
Assets	.flv, .mp3, .jpg, .gif, .swf, .png, .svg, .xml, other	The assets required by your application, including image, skin, sound, and video files.

Flex 2 SDK directory structure

A typical Flex application consists of a main MXML file, the file that contains the `<mx:Application>` tag, one or more MXML files that implement custom MXML components, one or more ActionScript files that contains custom components and custom logic, and asset files.

The following example shows an example of the directory structure of a simple Flex application:



This application consists of a root application directory and directories for different types of files. Everything required to compile and run the application is contained in the directory structure of the application.

Flex Builder adds additional directories to the application that are not present for Flex 2 SDK applications:

.settings Contains the preference settings for your Flex Builder project

bin Contains the generated SWF file and wrapper file, and the generated debug SWF and debug wrapper files, if generated

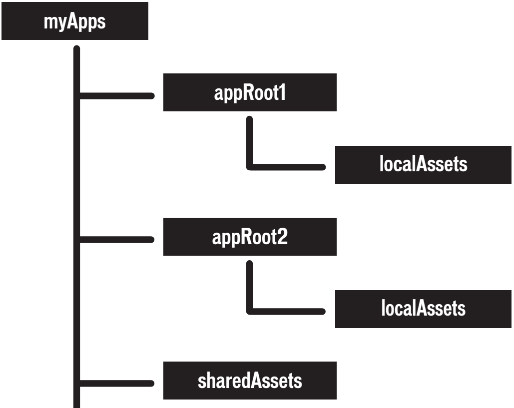
html-template Contains additional files used by specific Flex features, such as the history manager and Flash Player detection files, used by Flex Builder to generate a wrapper for your SWF file.

There are no inherent restrictions in Flex for the location of the root directory of your application, so you can put it almost anywhere in the file system of your computer. If you are using Flex Builder, the default location of the application root directory in Microsoft Windows is My Documents\Flex Builder 2*project_name* (for example, C:\Documents and Settings*userName*\My Documents\Flex Builder 2\myFlexApp).

Sharing assets among applications

Typically, you do not develop a single application in isolation from all other applications. Your application shares files and assets with other applications.

The following example shows two Flex applications, appRoot1 and appRoot2. Each application has a directory for local assets, and can access shared assets from a directory outside of the application's directory structure:



The location of the shared assets does not have to be at the same level as the root directories of the Flex applications. It only needs to be somewhere accessible by the applications at compile time.

In the following example, you use the Image control in an MXML file in the appRoot1 directory to access an asset from the shared assets directory:

```
<?xml version="1.0"?>
<!-- apparch/EmbedExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Image id="loader1" source="@Embed(source='../assets/bird.gif')"/>
</mx:Application>
```

Consideration for accessing application assets

One of the decisions that you must make when you create a Flex application is whether to load your assets at run time, or to embed the assets within the application's SWF file.

When you embed an asset, you compile it into your application's SWF file. The advantage to embedding an asset is that it is included in the SWF file, and can be accessed faster than having to load it from a remote location at run time. The disadvantage of embedding is that your SWF file is larger than if you load the asset at run time.

If you decide to access an asset at run time, you can load it from the local file system of the computer on which the SWF file runs, or you can access a remote asset, typically though an HTTP request over a network.

A SWF file can access one type of external asset: local or over a network; the SWF file cannot access both types. You determine the type of access allowed by the SWF file by using the `use-network` flag when you compile your application. When you set the `use-network` flag to `false`, you can access assets in the local file system, but not over the network. The default value is `true`, which lets you access assets over the network, but not in the local file system.

For more information on the `use-network` flag, see Chapter 9, "Using the Flex Compilers," on page 179. For more information on embedding application assets, see Chapter 30, "Embedding Assets," in *Flex 2 Developer's Guide*.

Sharing MXML and ActionScript files among applications

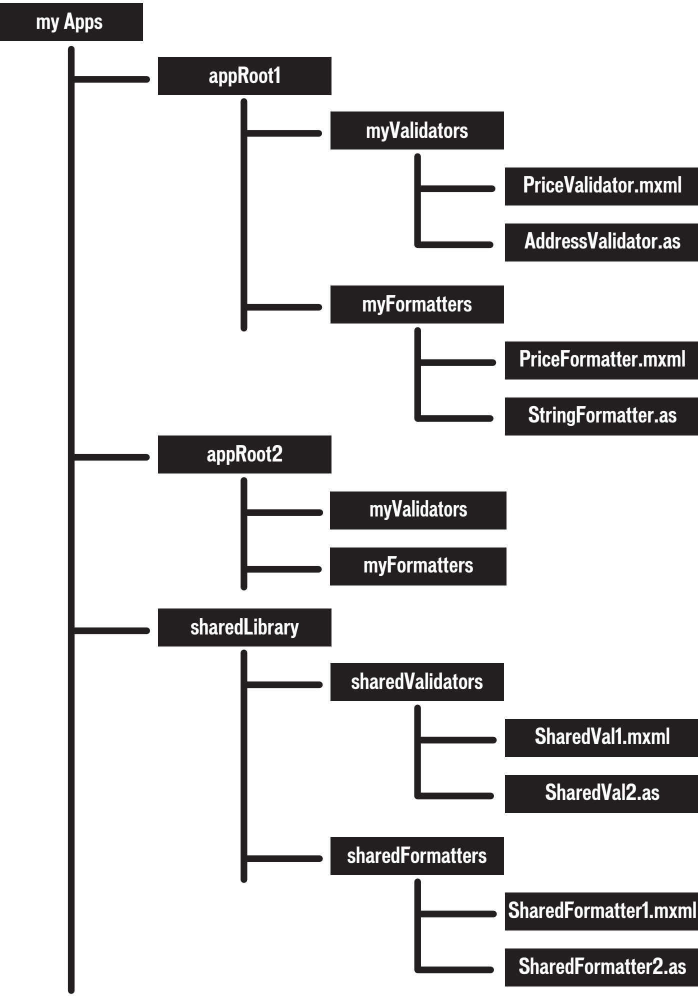
You can build an entire Flex application in a single MXML file that contains both your MXML code and any supporting ActionScript code. As your application gets larger, your single file also grows in size and complexity. This type of application would soon become difficult to understand and debug, and very difficult for multiple developers to work on simultaneously.

Flex supports a component-based development model. You use the predefined components included with Flex to build your applications, and create components for your specific application requirements. You can create custom components using MXML or ActionScript.

Beta Beta Beta Beta Beta Beta Beta Beta Beta Beta

Defining your own components has several benefits. One advantage is that components let you divide your applications into modules that you can develop and maintain separately. By implementing commonly used logic within custom components, you can also build a suite of reusable components that you can share among multiple Flex applications.

The following example shows two Flex applications, appRoot1 and appRoot2. Each application has a subdirectory for local MXML and ActionScript components, and can also reference a library of shared components:



The Flex compiler uses the source path to determine the directories where it searches for MXML and ActionScript files. By default, the root directory of the application is included in the source path; therefore, a Flex application can access any MXML and ActionScript files in its main directory, or in a subdirectory.

For shared MXML and ActionScript files that are outside of the application's directory structure, you modify the source path to include the directories that the compiler searches for MXML and ActionScript files. The component search order in the source path is based on the order of the directories listed in the source path.

You can set the source path as part of configuring your project in Flex Builder, in the flex-config.xml file, or set it when you open the command-line compiler. In this example, you set the source path to:

```
C:\myApps\sharedLibrary
```

To access a component in an MXML file, you specify a namespace definition that defines the directory location of the component relative to the source path. In the following example, an MXML file in the appRoot1 directory accesses an MXML component in the local directory structure, and in the directory containing the shared library of components:

```
<?xml version="1.0"?>
<!-- apparch/ComponentNamespaces.mxml -->
<mx:Application
  xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:MyLocalComps="myFormatters.*"
  xmlns:MySharedComps="sharedFormatters.*"
>

  <MyLocalComps:PriceFormatter/>

  <MySharedComps:SharedFormatter2/>

</mx:Application>
```

The MXML tag name for a custom component is composed of two parts: the namespace prefix, in this example MyLocalComps and MySharedComps, and the tag name. The namespace prefix tells Flex the directory in the source path that contains the file that implements the custom component. The tag name corresponds to the filename of the component, in this example PriceFormatter.mxml and SharedFormatter2.mxml.

Using a SWC file in a Flex 2 SDK application

A SWC file is a Flex library file that contains one or more components implemented in MXML or ActionScript. All Flex library files are shipped as SWC files in the frameworks/libs directory. This includes the following SWC files:

- fds.swc
- framework.swc
- playerglobal.swc
- rpc.swc

You can also create SWC files that you package and reuse among multiple applications. You typically use static linking with SWC files, which means the compiler includes all components, classes, and their dependencies in the application SWF file when you compile the application. For more information on static linking, see Chapter 10, “Using Runtime Shared Libraries,” on page 233.

By default, the Flex compiler includes all SWC files in the frameworks/libs directory when it compiles your application. For your custom SWC files, you use the `lib-path` option to the mxmlc compiler, or set the Library path in Flex Builder, to specify the location of the SWC file.

Using an RSL in a Flex 2 SDK application

One way to reduce the size of your application’s SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but must be transferred to the client only once. These shared files are known as Runtime Shared Libraries or RSLs.

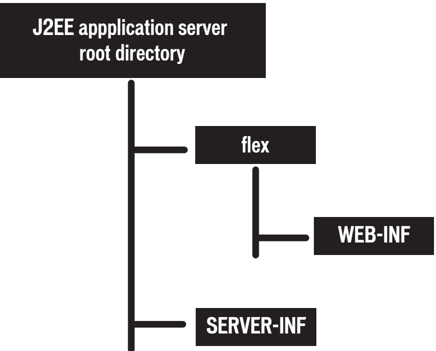
An RSL is a stand-alone file that the client downloads separately from your application’s SWF file, and caches on the client computer for use with multiple application SWF files. Using an RSL reduces the resulting file size for your applications. The benefits increase as the number of applications that use the RSL increases. If you only have one application, putting components into RSLs does not reduce the aggregate download size, and may increase it.

You create an RSL as a SWC file that you package and reuse among multiple applications. To reference an RSL, you use the `runtime-shared-libraries` option for the command-line compiler, or Flex Builder. You typically use dynamic linking with RSLs, which means some classes used by an application are left in an external file that is loaded at run time. For more information on RSLs and dynamic linking, see Chapter 10, “Using Runtime Shared Libraries,” on page 233.

Flex Data Services application directory structure

Before you can develop an application for Flex Data Services, you must deploy the Flex Data Services web application. You can deploy the web application on a J2EE application server or servlet container.

The following example shows the directory structure of the Flex Data Services web application:



This image does not show the complete directory structure of the WEB-INF directory. The following table describes the complete directory structure in more detail:

Directory	Description
/flex	Contains the WEB-INF directory. This is the root directory for the Flex web application (named flex). If you develop your application in the flex web application, this directory also includes all files that must be accessible by the user's web browser, such as MXML files, JSPs, HTML pages, Cascading Style Sheets, images, and JavaScript files. You can place these files directly in the web application root directory or in arbitrary subdirectories that do not use the reserved name WEB-INF.
/WEB-INF	Contains the standard web application deployment descriptor (web.xml) that configures Flex. This directory might also contain a vendor-specific web application deployment descriptor.
/WEB-INF/classes	Contains Java class files and configuration files.
/WEB-INF/flex	Contains Flex configuration files.
/WEB-INF/flex/jars	Contains the JAR files.

Directory	Description
/WEB-INF/flex/libs	Contains the SWC component file framework.swc, that contains the Flex application framework files, and other SWC files.
/WEB-INF/flex/locale	Contains localization resource files.
/WEB-INF/flex/themes	Contains the Flex theme files that define the basic look and feel of all Flex components.
/WEB-INF/flex/user_classes	Contains custom ActionScript classes, MXML components, and SWC files.
/WEB-INF/lib	Contains Flex server code in JAR files.

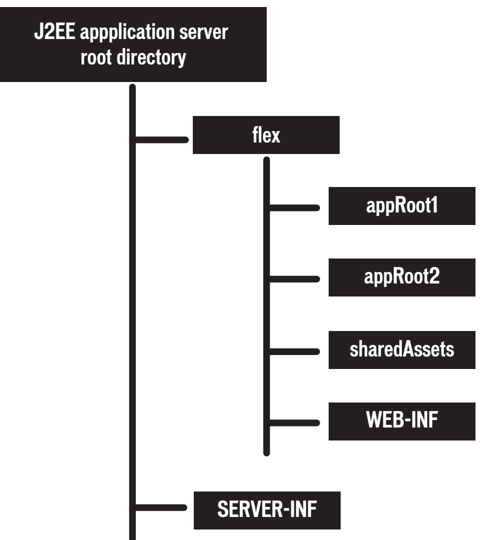
Options for developing a Flex Data Services application

When you develop applications for Flex Data Services, you have two choices for how you arrange the directory structure of your application:

- Use the same directory layout as you do for Flex 2 SDK. In this case, you define the directory structure on your computer, compile the application into a SWF file, and then deploy it on the server that hosts Flex Data Services. For more information, see "Flex 2 SDK directory structure" on page 36.
- Define a directory structure on your J2EE server or servlet container, either a directory structure in its own web application or in the directory structure of the Flex Data Services web application.

Beta Beta Beta Beta Beta Beta Beta Beta Beta Beta

The following example shows an example of the directory structure of a Flex Data Services application within the Flex web application:

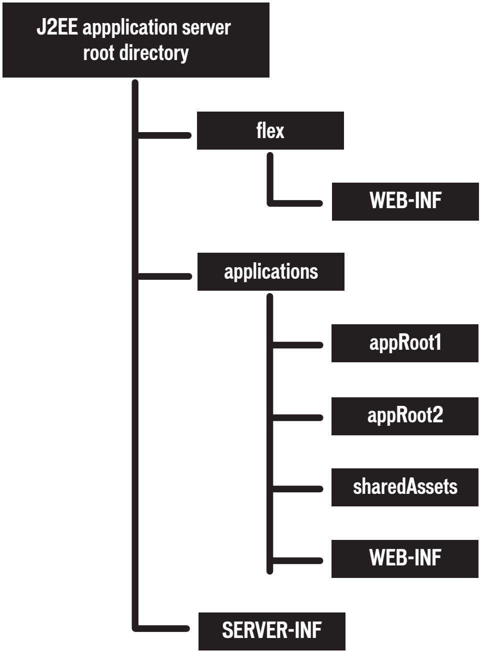


In this example, you define your application in the directory structure of the Flex Data Services web application.

You can define each application in its own directory structure, with the local assets for the application below the application's root directory. For assets shared across applications, such as image files, you can define a directory that is accessible by all applications.

You can place shared components such as MXML, ActionScript, and SWC files in the /WEB-INF/flex/user_classes directory. This directory is included in the default source path for all applications. To use a different directory, ensure that you include it in the application's source path. For more information on setting the source path, see Chapter 9, "Using the Flex Compilers," on page 179.

The following example shows a more common approach to web application design. In this example, you develop your application in its own web application, outside of the Flex Data Services web application:



This configuration has the advantage of isolating your application files from the Flex Data Services web application so that you can package and distribute them separately.

Using a SWC file in a Flex Data Services application

When you use SWC files in your Flex Data Services application, you have two choices for how you arrange the directory structure of your application:

- Use the same directory layout as you do for Flex 2 SDK. In this case, you define the directory structure on your computer, and place your SWC files in a directory that you specify using the `lib-path` option to the compiler, or set the Library path in Flex Builder. For more information, see "Using a SWC file in a Flex 2 SDK application" on page 41.
- If you define a directory structure on your J2EE server or servlet container, you can copy your SWC file to the *flex_app_root*/WEB-INF/flex/user_classes directory. You can also copy SWC files to a directory specified by the `<library-path>` child tag in the flex-config.xml file. SWC files must be stored at the top level of the user_classes directory or the directory specified by the `<library-path>` element.

Using an RSL in Flex Data Services application

For Flex Data Services, you specify the location of an RSL in the same way as you do for Flex 2 SDK. For more information, see "Using an RSL in a Flex 2 SDK application" on page 41.

Compiling an application

Compiling your application converts your application files and assets into a single SWF file. During compilation, you set compiler options to enable accessibility, enable debug information in the output, set library paths, and set other options. You can configure the compiler as part of configuring your project in Flex Builder, by using command-line arguments to the compiler, or by setting options in a configuration file.

FOr more information on compiling applications, see Chapter 9, "Using the Flex Compilers," on page 179.

About case sensitivity during a compile

The Flex compilers use a case-sensitive file lookup on all file systems. On case-insensitive file systems, such as the Macintosh and Windows file systems, the Flex compiler generates a case-mismatch error when you use a component with the incorrect case. On case-sensitive file systems, such as the UNIX file system, the Flex compiler generates a component-not-found error when you use a component with the incorrect case.

Compiling a Flex 2 SDK application

Flex 2 SDK includes two compilers, mxmlc and compc. You use mxmlc to compile MXML files, ActionScript files, SWC files, and RSLs into a single SWF file. After your application is compiled and deployed on your web or application server, a user can make an HTTP request to download and play the SWF file on their computer. You use the compc compiler to compile components, classes, and other files into SWC files or RSLs.

To compile an application with Flex 2 SDK, you use the mxmlc compiler in the bin directory of your Flex 2 SDK directory. The most basic mxmlc example is one in which the MXML file for your application has no external dependencies (such as components in a SWC file or ActionScript classes). In this case, you open mxmlc from the command line and point it to your MXML file, as the following example shows:

```
$ mxmlc c:/myFiles/app.mxml
```

Beta Beta Beta Beta Beta Beta Beta Beta Beta Beta

The mxmlc compiler has many options that you can specify on the command line, or that you can set in the flex-config.xml file. For example, to disable warning messages, you set the `warnings` options to `false`, as the following example shows:

```
$ mxmlc -warnings=false c:/myFiles/app.mxml
```

You only specify the main application file, the file that contains the `<mx:Application>` tag, to the compiler. The compiler searches the default source path for any MXML and ActionScript files that your application references. If your application references MXML and ActionScript files in directories that are not included in the default source path, you can use the `source-path` option to add a directory to the source path, as the following example shows:

```
$ mxmlc -source-path path1 path2 path3 -- c:/myFiles/app.mxml
```

In this example, you specify a list of directories, separated by spaces, and terminate that list with `--`.

Compiling an application that uses SWC files

Often, you use SWC files when compiling MXML files. You specify the SWC files in the compiler by using the `library-path` option.

The following example adds two SWC files to the `library-path` when it compiles your application:

```
$ ./mxmlc -library-path+=c:/myLibraries/MyRotateEffect.swc
  c:/myLibraries/MyButtonSwc.swc -- c:/myFiles/app.mxml
```

Compiling an application that uses RSL

To use an RSL in your application, use the `runtime-shared-libraries` compiler option. The following example compiles an application with an RSL at the command line:

```
$ mxmlc -runtime-shared-libraries+=c:/myRSLs/myCompRSL.swc
  c:/myFiles/app.mxml
```

Compiling a Flex Builder application

When you compile a project with Flex Builder, you open the Flex compilers from within Flex Builder itself, not from the command line. You can build your projects manually or let Flex Builder automatically compile them for you. In either case, the Flex Builder compiler creates the SWF application files, generates a wrapper, places the output files in the proper location, and alerts you to any errors encountered during compilation. You then run and debug your applications as needed.

If you must modify the default build settings, you have several options for controlling how your projects are built into applications. For example, you can set build preferences on individual projects or on all the projects in your workspace, modify the build output path, change the build order, and so on. You can also create custom build instructions using third-party tools, such as Apache Ant.

When your projects are built, automatically or manually, a release and debug version of your application SWF files are placed in the project output folder along with the wrapper. The debug version of your application contains debugging information and, therefore, is used when you debug your application. The standard version does not include the additional debugging information and is smaller. A wrapper file embeds the application SWF file and is used to run or debug your application in a web browser.

Compiling a Flex Data Services application

If you have Flex Data Services, you can compile your applications in one of two ways:

- You can compile them in the same way that you compile them using Flex 2 SDK or Flex Builder. For more information, see "Compiling a Flex 2 SDK application" on page 46 and "Compiling a Flex Builder application" on page 47.
- You can deploy your application as MXML, ActionScript, SWC, RSL, and assets files on your J2EE application server. When a user requests the main MXML file, typically using an HTTP request, the request triggers the compiler that is included in a deployed Flex Data Services web application.

  To configure the web tier compiler, you use the **/WEB-INF/flex**/flex-config.xml file.

  A request to an MXML file has the following form:

  `http://`*`hostname`*`/`*`path`*`/filename.mxml`

  Upon receiving an HTTP request for an MXML file, Flex performs the following steps:

  a. Compiles the MXML file to produce a SWF file.
  b. Caches the compiled SWF file on the server.
  c. Returns the SWF file to the client within a wrapper page.
  d. Subsequent requests return the cached SWF file; they do not trigger a recompilation.

Deployment directory structure

When you deploy an application, ensure that the directory structure of the deployed application is correct. For Flex 2 SDK applications, the directory structure is typically very simple, while for Flex Data Services applications, it can be much more complicated.

Flex 2 SDK deployment directory structure

When you use Flex 2 SDK, you compile your application into a SWF file, and optionally one or more RSLs, that you deploy by copying to your web server. After you deploy the application, users can access it from the web server.

When you deploy your application, you also must be aware of how your application accesses its assets. If you embedded all of your application assets into the SWF file, you can deploy the application as a stand-alone SWF file.

However, if you decide to access assets at run time, the application requests assets during execution. You must ensure that you deploy all of the necessary assets, in the correct location, so that you can run the application correctly.

Accessing an RSL at run time

If you compiled your application using an RSL, you must ensure that the RSL is also deployed to your web server, along with your application's SWF file. The directory location of the RSL must match the directory location that you specified at compile time using the `runtime-shared-libraries` option for the compiler.

Flex Data Services deployment directory structure

For Flex Data Services, you must ensure that you deploy any necessary assets on your deployment server, much in the same way as you deploy the assets for a Flex 2 SDK application. For more information, see "Flex 2 SDK deployment directory structure" on page 49.

Before you can deploy your Flex Data Services application, ensure that you also deploy the Flex Data Services web application.

You have several options for how you deploy a Flex Data Services application:

- You can precompile your application into a SWF file and deploy it to your web server or application server, much like you do for Flex 2 SDK and Flex Builder.
- You can package your application as a J2EE web application, meaning as a WAR or EAR file, and deploy the WAR fie on your J2EE application server or servlet container.
- You can define your application within the directory structure of the Flex Data Services web application. You then deploy the web application on your J2EE application server or servlet container.

For more information, see Chapter 15, "Deploying Flex Applications," on page 351.

CHAPTER 4

Applying Flex Security

4

This topic is intended for developers (including programmers and other authors) designing and publishing Flex applications, who are concerned with the security aspects of the applications they develop.

Contents

Introduction 51
Loading assets 64
Using J2EE authentication. 67
Using RPC services 70
Using data services 72
Making other connections 73
Using SSL 74
Writing secure Flex applications. 76
Configuring client security settings. 82
Other resources 85

Introduction

Adobe Flash Player runs Flash applications (also referred to as SWF files). Flash content is delivered as a series of instructions in binary format to Flash Player over web protocols in the precisely described SWF (.swf) file format. The SWF files themselves are typically hosted on a server and then downloaded to, and displayed on, the client computer when requested. Most of the content consists of binary ActionScript instructions. ActionScript is the ECMA standards-based scripting language that Flash uses that features APIs designed to allow the creation and manipulation of client-side user interface elements and for working with data.

The Flex security model protects both client and the server. Consider the following two general aspects to security:

- Authorization and authentication of users accessing a server's resources
- Flash Player operating in a sandbox on the client

Flex supports working with the web application security of any J2EE application server. In addition, precompiled Flex applications can integrate with the authentication and authorization scheme of any underlying server technology to prevent users from accessing your applications. The Flex framework also includes several built-in security mechanisms that let you control access to web services, HTTP services, and server-based resources such as EJBs.

Flash Player runs inside a security sandbox that prevents the client from being hijacked by malicious application code.

Declarative compared to programmatic security

The two common approaches to security are declarative and programmatic. Often, declarative security is server based. Using the server's configuration, you provide protection to a resource or set of resources. You use the container's authentication and authorization schemes to protect that resource from unauthorized access.

The declarative approach to security casts a wide net. Declarative security is implemented as a separate layer from the web components that it works with. You set up a security system, such as a set of file permissions or users, groups, and roles, and then you plug your application's authentication mechanism into that layer.

With declarative security, either a user gains access to the resource or they do not. Usually the content cannot be customized based on roles. In an HTML-based application, the result is that users are denied access to certain pages. However, in a Flex environment, the typical result of declarative security is that the user is denied access to the entire application, since the application is seen as a single resource to the container.

Declarative security lets programmers who write web applications ignore the environment in which they write. Declarative security is typically set up and maintained by the deployer and not the developer of the application. Also, updates to the web application do not generally require a refactoring of the security model.

Programmatic security gives the developer of the application more control over access to the application and its resources. Programmatic security can be much more detailed than declarative security. For example, a developer using programmatic security can allow or deny a user access to a particular component inside the application.

Although programmatic security is typically configured by the developer of the application, it usually interacts with the same systems as declarative security, so the relationship between developer and deployer of the application must be cooperative when implementing programmatic security.

Declarative security is recommended over programmatic security for most applications because the design promotes code reuse, making it more maintainable. Furthermore, declarative security puts the responsibility of security into the hands of the people who specialize in its implementation; application programmers can concentrate on writing applications and people who deploy the applications in a specific environment can concentrate on enforcing security policies and take advantage of that context.

Client security overview

When considering security issues, you cannot think of Flex applications as traditional web applications. Flex applications typically consist of a single monolithic SWF file that is loaded by the client once. Web applications, on the other hand, usually consist of many individual pages that are loaded one at a time.

Most web applications access resources such as web services that are outside of the client. When a Flex application accesses an external resource, two factors apply:

- Is the user authorized to access this resource?
- Can the client load the resource, or is it prevented from loading the resource, because of its sandbox limitations?

The following basic security rules always apply by default:

- Resources in the same security sandbox can always access each other.
- SWF files in a remote sandbox can never access local files and data.

This section introduces you to security issues related to the client architecture that affect Flex applications.

Flash Player security features

Much of Flash Player security is based on the domain of origin for loaded SWF files, media, and other assets. A SWF file from a specific Internet domain, such as www.example.com, can always access all data from that domain. These assets are put in the same security grouping, known as a security sandbox. For example, a SWF file can load SWF files, bitmaps, audio, text files, and any other asset from its own domain. Also, cross-scripting between two SWF files from the same domain is permitted, as long as both files are written using ActionScript 3.0. Cross-scripting is the ability of one SWF file to use ActionScript to access the properties, methods, and objects in another SWF file. Cross-scripting is not supported between SWF files written using ActionScript 3.0 and files using previous versions of ActionScript; however, these files can communicate by using the LocalConnection class.

Memory usage and disk storage protections

Flash Player includes security protections for disk data and memory usage on the client computer.

The only type of persistent storage is through the SharedObject class, which is embodied as a file in a directory whose name is related to that of the owning SWF file. A Flex application cannot typically write, modify, or delete any files on the client computer other than SharedObject data files, and it can only access SharedObject data files under the established settings per domain.

Flash Player helps limit potential denial-of-service attacks involving disk space (and system memory) through its monitoring of the usage of SharedObject classes. Disk space is conserved through limits automatically set by Flash Player (the default is 100K of disk space for each domain). The author can set the Flash application to prompt the user for more disk space, or Flash Player automatically prompts the user if an attempt is made to store data that exceeds the limit. In either case, the disk space limit is enforced by Flash Player until the user gives explicit permission for an increased allotment for that domain.

Flash Player contains memory and processor safeguards that help prevent Flash applications from taking control of excess system resources for an indefinite period of time. For example, Flash Player can detect an application that is in an infinite loop and select it for termination by prompting the user. The resources that the application uses are immediately released when the application closes.

Flash Player uses a garbage collector engine. The processing of new allocation requests always first ensures that memory is cleared so that the new usage always obtains only clean memory and cannot view any previous data.

Privacy

Privacy is an important aspect of overall security. Adobe products, including Flash Player, provide very little information that would reveal anything about a user (or their computer). Flash Player does not provide personal information about users (such as names, e-mail addresses, and phone numbers), or provide access to other sensitive information (such as credit card numbers or account information).

What Flash Player does provide is basically standardized hardware and software configuration information that authors might use to enhance the user experiences in the environment encountered. The same information is often available already from the operating system or web browser.

Information about the client environment that is available to the Flex application includes:

- User agent string, which typically identifies the embedding browser type and operating system of the client
- System capabilities such as the language or the presence of an MP3 decoder (see the Capabilities class)
- Presence of a camera and microphone
- Keyboard and mouse input

ActionScript also includes the ability to replace the contents of the client's Clipboard by using the `setClipboard()` method of the System class. This method does not have a corresponding `getClipboard()` method, so protected data that might be stored in the Clipboard already is not accessible to Flash Player.

About sandboxes

The sandbox type indicates the type of security zone in which the SWF file is operating. In Flash Player, all SWF files (and HTML files, for the purposes of SWF-to-HTML scripting) are placed into one of four types of sandbox:

remote All files from non-local URLs are placed in a remote sandbox. There are many such sandboxes, one for each Internet (or intranet) domain from which files are loaded.

local-with-filesystem The default sandbox for local files. SWF files in this sandbox may not contact the Internet (or any servers) in any way—they may not access network endpoints with addresses such as HTTP URLs.

local-with-networking SWF file in this sandbox may communicate over the network but may not read from local file systems.

local-trusted This sandbox is not restricted. Any local file can be placed in this sandbox if given authorization by the end user. This authorization can come in two forms: interactively through the Settings Manager or noninteractively through an executable installer that creates Flash Player configuration files on the user's computer.

You can determine the current sandbox type by using the `sandboxType` property of the Security class, as the following example shows:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- security/DetectCurrentSandbox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script><![CDATA[
        [Bindable]
        private var l_sandboxType:String;

        private function initApp():void {
            l_sandboxType = String(Security.sandboxType);
        }
    ]]></mx:Script>

    <mx:Form>
        <mx:FormItem id="fi1" label="Security.sandboxType">
            <mx:Label id="l1" text="{l_sandboxType}"/>
        </mx:FormItem>
    </mx:Form>

</mx:Application>
```

When you compile a Flex application, you have some control over which sandbox the application is in. This determination is a combination of the value of the `use-network` compiler option (the default is `true`) and whether the SWF file was loaded by the client over a network connection or as a local file.

The following table shows how the sandbox type is determined:

use-network	Loaded	Sandbox type
true or false	locally	local-trusted
false	locally	local-with-filesystem
true	locally	local-with-network
true	network	remote

Browser security

Flash Player clients can be one of the following four types:

- Embedded Flash Player

- Debugger version of embedded Flash Player
- Stand-alone Flash Player
- Debugger version of stand-alone Flash Player

The stand-alone Flash Player runs on the desktop. It is typically used by people who are running applications that are installed and maintained by an IT department that has access to the desktop on which the application runs.

The embedded Flash Player is run within a browser. Anyone with Internet access can run applications from anywhere with this player. For Internet Explorer, the embedded player is loaded as an ActiveX control inside the browser. For Netscape-based browsers (including Firefox), it is loaded as a plug-in inside the browser. Using an embedded player lets the developer use browser-based technologies such as FORM and BASIC authentication as well as SSL.

Browser APIs

Applications hosting the Flash Player ActiveX control or Flash Player plug-in can use the EnforceLocalSecurity and DisableLocalSecurity API calls to control security settings. If DisableLocalSecurity is opened, the application does not benefit from the local-with-networking and local-with-file-system sandboxes. All files loaded from the local file system are placed into the local-trusted sandbox. The default behavior for an ActiveX control hosted in a client application is DisableLocalSecurity.

If EnforceLocalSecurity is opened, the application can use all three local sandboxes. The default behavior for the browser plug-in is EnforceLocalSecurity.

Cross-scripting

Cross-scripting is when a SWF file communicates directly with another SWF file. This communication includes calling methods and setting properties of the other SWF file.

SWF file loading and cross-scripting are always permitted between SWF files that reside in the same sandbox. For example, any local-with-filesystem SWF file can load and cross-script any other local-with-filesystem SWF file; any local-with-networking SWF file can load and cross-script any other local-with-networking SWF file; and so on. The restrictions appear when two SWF files from different sandboxes or two remote SWF files with different domains attempt to cooperate.

For SWF files in the remote sandbox, if two SWF files were loaded from the same domain, they can cross-script without any restrictions. If both SWF files were loaded from a network, but from different domains, you must provide permissions to allow them to cross-script.

To enable cross-scripting between SWF files, use the Security class's `allowDomain()` and `allowInsecureDomain()` methods.

You call these methods from the called SWF file and specify the calling SWF file's domain. For example, if SWF1 in domainA.com calls a method in SWF2 in domainB, SWF2 must call the `allowDomain()` method and specifically allow SWF files from domainA.com to cross-script the method, as the following example shows:

```
import flash.system.Security;
Security.allowDomain("domainA.com");
```

If the SWF files are in different sandboxes (for example, if one SWF file was loaded from the local file system and the other from a network) they must adhere to the following set of rules:

- Remote SWF files (those served over HTTP and other non-local protocols) can never load local SWF files.
- Local-with-networking SWF files can never load local-with-filesystem SWF files, or vice versa.
- Local-with-filesystem SWF files can never load remote SWF files.
- Local-trusted SWF files can load SWF files from any sandbox.

To facilitate SWF-to-SWF communication, you can also use the LocalConnection class. For more information, see "Using the LocalConnection class" on page 74.

ExternalInterface

You use the ExternalInterface API to let your Flex application call scripts in the wrapper and to allow the wrapper to call functions in your Flex application. The ExternalInterface API consists primarily of the `call()` and `addCallback()` methods in the flash.net package.

This communication relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` properties define. You set the values of the `allowScriptAccess` and `allowNetworking` properties in the SWF file's wrapper. For more information, see "About the <object> and <embed> tags" on page 378.

By default, the Flex application and the HTML page it is calling must be in the same domain for the `call()` method to succeed. For more information, see Chapter 34, "Communicating with the Wrapper," in *Flex 2 Developer's Guide*.

The navigateToURL() method

The `navigateToURL()` method opens or replaces a window in the Flash Player's container application. You typically use it to launch a new browser window, although you can also embed script in the method's call to perform other actions.

This usage of the `navigateToURL()` method relies on the domain-based security restrictions that the `allowScriptAccess` and `allowNetworking` parameters define. You set the values of the `allowScriptAccess` and `allowNetworking` parameters in the SWF file's wrapper. For more information, see "About the <object> and <embed> tags" on page 378.

Caching

Flex applications reside entirely on the client. If the browser loads the application, the application SWF file, plus externally loaded images and other media files, are stored locally on the client in the browser's cache. These files reside in the cache until cleared.

Storing a SWF file in the browser's cache can potentially expose the file to people who would not otherwise be able to see it.

Broswer or operating system	Cache location
Internet Explorer	C:\Documents and Settings*username*\Local Settings\Temporary Internet Files
Firefox on Windows	C:\Documents and Settings*username*\Application Data\Mozilla\Firefox\Profiles\{*user_id*}.default\Cache
UNIX	$HOME/.mozilla/firefox/{*user_id*}.default/Cache/

These files can remain in the cache even after the browser is closed.

To prevent client browsers from caching the SWF file, try setting the following HTTP headers in the wrapper that returns the Flex application's SWF file:

```
Cache-control: no-cache, no-store, must-revalidate, max-age=-1
Pragma: no-cache, no-store
Expires: -1
```

Trusted sites and directories

The browser security model includes levels of trust applied to specific websites. Flash Player interacts with this model by assigning a sandbox based on whether the browser declared the site of the SWF file's origin trusted.

If Flash Player loads a SWF file from a trusted website, the SWF file is put in the local-trusted sandbox. The SWF file can read from local data sources and communicate with the Internet.

You can also assign a SWF file the local-trusted sandbox when you load a SWF file from the local file system. To do this, you configure a directory as trusted by Flash Player (which results in the SWF file being put in the local-trusted sandbox) by adding a FlashPlayerTrust configuration file that specifies the directory to trust. This requires administrator access privileges to the client system, so it is typically used in controlled environments. Users can also define a directory as trusted by using the Flash Player User Settings Manager. For more information, see Flash Player documentation.

Server security overview

When you use Flex Data Services, you install it as a web application on a J2EE application server. In addition, you can optionally install the JRun Java application server when you install Flex Data Services or Flex Builder. This section provides an overview of security-related issues for Flex server-based products.

About J2EE security

To handle authentication and authorization of the system, J2EE security covers multiple areas, each of which is handled by different roles:

- **Role definition and programmatic security** The application developer defines the roles that apply at the web application level. The application developer can optionally implement programmatic security, as required by the application. Declarative security is recommended over programmatic security for most J2EE applications.
- **Client coding** The application developer ensures that clients pass the required credentials (typically a user ID and password) at the appropriate time. Web application clients prompt for user ID and password.
- **Role resolution and declarative security** The application assembler resolves and links programmer-defined roles with system roles. The application assembler also specifies declarative security in the web deployment descriptors.
- **Security architecture and user-store management** The administrator controls the user store, coordinates global role definition, and customizes security to match the site-specific security environment.

For information on using J2EE security methods, see your application server's documentation.

Deploying secure applications

When you deploy an application, you make the application accessible to your users. The process of deploying an application is dependent on your application, your application requirements, and your deployment environment. This section provides some strategies you can employ to ensure that the application you deploy is secure.

Deploying local SWF files versus network SWF files

Client computers can obtain individual SWF files from a number of sources, such as from an external website or a local file system. When SWF files are loaded into Flash Player, they are individually assigned to security sandboxes based on their origin.

Flash Player classifies SWF files downloaded from the network (such as from external websites) in separate sandboxes that correspond to their website origin domains. By default, these files are authorized to access additional network resources that come from the specific (exact domain name match) site. Network SWF files can be allowed to access additional data from other domains by explicit website and author permissions.

A local SWF file describes any file referenced by using the "file:\\" protocol or a UNC path, which does not include an IP address or a qualifying domain. For example, "\\test\test.swf" and "file:\\test.swf" are considered local files, while "\\test.com\test.swf" and "\\192.168.0.1\test.swf" are not considered local files.

Local SWF files from local origins, such as local file systems or UNC network paths, are placed into one of three sandboxes: local-with-networking, local-with-filesystem, and local-trusted.

When you compile the Flex application, if you set the `use-network` compiler option to `false`, local SWF files are placed in the local-with-filesystem sandbox. If you set the `use-network` compiler option to `true`, local SWF files are placed in the local-with-networking sandbox.

Local SWF files that are registered as trusted (by users or by installer programs) are placed in the local-trusted sandbox. Users can also reassign (move) a local SWF file to or from the local-trusted sandbox based on their security considerations.

Deploy checklist

Before you deploy your application, ensure that your proxy servers, firewalls, and assets are configured properly. Adobe provides a deployment checklist that you can follow. For more information, see "Deployment checklist" on page 360.

Enabling production mode

You enable production mode for Flex Data Services when your application is actively running on a public-facing server. Enabling or disabling production mode mainly affects how the web-tier compiler compiles a Flex Data Services application deployed as MXML and ActionScript files.

For more information, see "Enabling production mode" on page 359.

Clearing the server cache

When in production mode, the server running Flex Data Services caches the SWF file after it is first compiled and does not poll for file changes until you restart the server. As a result, if you make changes to your Flex application or to dependent files such as imported images, movies, or class libraries, you must restart your server before those changes take effect. Not restarting can cause users to get the wrong version of your application or other assets. Flex polls for file changes only during startup when production mode is enabled.

Remove wildcards

If your application relies on assets loaded from another domain, and that domain has a crossdomain.xml file on it, remove wildcards from that file if possible. For example, change the following:

```
<cross-domain-policy>
  <allow-access-from domain="*" to-ports="*"/>
</cross-domain-policy>
```

to this:

```
<cross-domain-policy>
  <allow-access-from domain="*.myserver.com" to-ports="80,443,8100,8080" />
</cross-domain-policy>
```

Also, set the value of the `to-ports` attribute of the `allow-access-from` tag to ensure that you are only allowing necessary ports access to the resources.

Check your application for calls to the `allowDomain()` and `allowInsecureDomain()` methods. During development, you might pass these methods a wildcard character (*), but now restrict those methods to allowing requests only from the necessary domains.

Deploy assets to WEB-INF

In some deployments, you want to make assets such as data files accessible to the application, but not accessible to anyone requesting the file. If you are using Flex Data Services or a J2EE-based server, you can deploy those files to a subdirectory within the WEB-INF directory. Based on J2EE security constraints, no J2EE server can return a resource from the WEB-INF directory to any client request. The only way to access files in this directory is with server-side code.

Precompiling source code

Precompile MXML files, JSP files, and class files. After you precompile your MXML and JSP files, remove the source files from the public-facing server.

To precompile MXML files, use the mxmlc compiler utility in the bin directory. For more information on using the utility, see "About the command-line compilers" on page 187.

To precompile JSP files on JRun, for example, you use the jspc precompiler utility located in the JRun bin directory. For information on precompiling JSP files on your application server, see your application server documentation.

Preventing access to certain file types

If you deploy the Flex Data Services web application, you can ensure that file types such as *.JSP, *.AS, and *.MXML cannot be sent in their raw format to the client. The FlexForbiddenServlet intercepts requests to the specified file types and prevents clients from accessing the source files. By default, it intercepts direct client requests for *.as and *.swc files.

You can intercept other file type requests with the FlexForbiddenServlet by adding additional servlet mappings in the web.xml file in the /flex/WEB-INF directory. For example, to prevent users from accessing your GIF files directly, add the following mapping to the web.xml file:

```
<servlet-mapping>
   <servlet-name>FlexForbiddenServlet</servlet-name>
   <url-pattern>*.gif</url-pattern>
</servlet-mapping>
```

The value of the `url-pattern` is not case sensitive, so adding *.gif prevents users from accessing files with the GIF, gif, and Gif extensions.

Securing JRun

When you install Flex Data Services, you optionally install the JRun J2EE web application server. You can use this server to run the Flex web application and enable messaging, named destinations, and other server-based Flex features.

This integrated version of JRun is not full featured and should not be used in a production environment. However, you can use some techniques to make it as secure as possible if necessary. These techniques include:

- Disable directory browsing. By default, directory browsing is enabled. Disable it if you are going to make your application publicly accessible by using the JRun server. For more information, see "Enabling and disabling directory browsing" on page 409.
- Disable the JRun Web Service (JWS). If you use the web server connector to connect to another web server, disable the JWS. Set the `deactivated` attribute of the WebService to `true` in the jrun.xml file.

- Disable hot-deploy. Set the `HotDeploy` attribute of the DeployerService to `false` in the jrun.xml file.
- Disable web services. If your application does not use web services, disable the AxisServlet mappings in the SERVER-INF/default-web.xml file.
- Prevent access to particular file types. You can use URL patterns to define file types (such as *.as or *.swc) that should not be returned by a request. For more information, see "Preventing access to certain file types" on page 63.

For more information on using the integrated JRun server, see Chapter 18, "Configuring JRun," on page 401 and the JRun documentation.

Loading assets

The most common task that developers will perform that requires an understanding of security is loading external assets. This section describes topics related to loading assets.

Data compared to content

The Flash Player security model makes a distinction between loading content and accessing or loading data. Content is defined as media: visual media that Flash Player can display, such as audio, video, or a SWF file that includes displayed media. Data is defined as something that you can manipulate only with ActionScript code.

You can load data in one of two ways: by extracting data from loaded media content, or by directly loading data from an external file (such as an XML file) or socket connection. You can extract data from loaded media by using the `BitmapData.draw()` method, the `Sound.id3` property, or the `SoundMixer.computeSpectrum()` method. You can load data by using classes such as the SWFLoader, URLStream, URLLoader, Socket, and XMLSocket classes.

The Flash Player security model defines different rules for loading content and accessing data. Loading content has fewer restrictions than accessing data. In general, content such as SWF files, bitmaps, MP3 files, and videos can be loaded from anywhere, but if the content is from a domain other than that of the loading SWF file, it will be partitioned in a separate security sandbox.

Loading remote assets

Loading remote or network assets relies on three factors:

- Type of asset. If the target asset is a content asset, such as an image file, you do not need any specific permissions from the target domain to load its assets into your Flex application. If the target asset is a data asset, such as an XML file, you must have the target domain's permission to access this asset. For more information on the types of assets, see "Data compared to content" on page 64.
- Target domain. If you are loading data assets from a different domain, the target domain must provide a crossdomain.xml policy file. This file contains a list of URLs and URL patterns that it allows access from. The calling domain must match one of the URLs or URL patterns in that list. For more information about the crossdomain.xml file, see "Using cross-domain policy files" on page 65. If the target asset is a SWF file, you can also provide permissions by calling the `loadPolicyFile()` method and loading an alternative policy file inside that target SWF file. For more information, see "Using cross-domain policy files" on page 65.
- Loading SWF file's sandbox. To load an asset from a network address, you must ensure that your SWF file is in either the remote or local-with-networking sandbox. To ensure that a SWF file can load assets over the network, you must set the `use-network` compiler option to `true` when you compile the Flex application. This is the default. If the application was loaded from the local file system with `use-network` set to `false`, the application is put in the local-with-filesystem sandbox and it cannot load remote SWF files.

Loading assets from a remote location that you do not control can potentially expose your users to risks. For example, the remote website B contains a SWF file that is loaded by your website A. This SWF file normally displays an advertisement. However, if website B is compromised and its SWF file is replaced with one that asks for a username and password, some users might disclose their login information. To prevent data submission, the loader has a property called `allowNetworking` with a default value of `never`.

Using cross-domain policy files

To make data available to SWF files in different domains, use a *cross-domain policy file*. A cross-domain policy file is an XML file that provides a way for the server to indicate that its data and documents are available to SWF files served from other domains. Any SWF file that is served from a domain that the server's policy file specifies is permitted to access data or assets from that server.

When a Flash document attempts to access data from another domain, Flash Player attempts to load a policy file from that domain. If the domain of the Flash document that is attempting to access the data is included in the policy file, the data is automatically accessible.

The default policy file is named crossdomain.xml and resides at the root directory of the server that is serving the data. The following example policy file permits access to Flash documents that originate from foo.com, friendOfFoo.com, *.foo.com, and 105.216.0.40:

```
<?xml version="1.0"?>
<!-- http://www.foo.com/crossdomain.xml -->
<cross-domain-policy>
  <allow-access-from domain="www.friendOfFoo.com"/>
  <allow-access-from domain="*.foo.com"/>
  <allow-access-from domain="105.216.0.40"/>
</cross-domain-policy>
```

You can also configure ports in the crossdomain.xml file. For more information about crossdomain.xml policy files, see *Programming ActionScript 3.0*.

You can use the `loadPolicyFile()` method to access a nondefault policy file.

Loading local assets

In some cases, your SWF file might load assets that reside on the client's local file system. This typically happens when the Flex application is embedded on the client device and loaded from a network. If the application is allowed to access local assets, it cannot access network assets.

To ensure that a Flex application can access assets in the local sandbox, the application must be in the local-with-filesystem or local-trusted sandbox. To ensure this, you set the `use-network` compiler option to `false` when you compile the application. The default value of this option is `true`.

When you load another SWF file that is in the local file system into your application with a class such as SWFLoader, and you want to call methods or access properties of that SWF file, you do not need to explicitly enable cross-scripting.

If the SWF files are in different sandboxes (for example, you loaded the main SWF file into the local-with-network sandbox, but loaded the asset SWF file from the network), you cannot cross-script because they are in different sandboxes. Remote SWF files cannot load local SWF files, and vice versa.

Using J2EE authentication

To effectively implement secure web applications, you should understand the following concepts:

Authentication The process of gathering user credentials (user name and password) and validating them in the system. This requires checking the credentials against a user repository such as a database, flat file, or LDAP implementation, and authenticating that the user is who they say they are.

Authorization The process of making sure that the authenticated user is allowed to view or access a given resource. If a user is not authorized to view a resource, the container does not allow access.

Using container-based authentication

J2EE uses the Java Authentication and Authorization Service (JAAS), Java security manager, and policy files to enforce access controls on users and ties this enforcement to web server roles. The authenticating mechanism is role based. That is, all users who access a web application are assigned to one or more roles. Example roles are manager, developer, and customer.

Application developers can assign usage roles to a web application, or to individual resources that make up the application. Before a user is granted access to a web application resource, the container ensures that the user is identified (logged in) and that the user is assigned to a role that has access to the resource. Any unauthorized access of a web application results in an HTTP 401 (Unauthorized) status code.

Authentication requires a website to store information about users. This information includes the role or roles assigned to each user. In addition, websites that authenticate user access typically implement a login mechanism that forces verification of each user's identity by using a password. After the website validates the user, the website can then determine the user's roles.

This logic is typically implemented in one of the following forms:

- JDBC Login Module
- LDAP Login Module
- Windows Login Module
- Custom JAAS Login Module

Authentication occurs on a per-request basis. The container typically checks every request to a web application and authenticates it.

Authentication requires that the roles that the application developer defines for a web application be enforced by the server that hosts the application. This section describes how to set the roles, and other authentication information, for the application during application development.

As part of developing and deploying an application, you must configure the following application authentication settings:

- Access roles to applications
- Resource protection
- Application server validation method

The web application's deployment descriptor, web.xml, contains the settings for controlling application authentication. This file is stored in the web application's WEB-INF directory.

Using authentication to control access to Flex applications

To use authentication to prevent unauthorized access to your Flex application, you typically use the container to set up constraints on resources. You then challenge the user who then submits credentials. These credentials determine the success or failure of the user's login attempt, as the container's authentication logic determines.

For example, you can protect the page that the Flex application is returned with, or protect the SWF file itself. You do this in the web.xml file by defining specific URL patterns, as the following example shows:

```
<web-app>
  <security-constraint>
    <web-resource-collection>
      <web-resource-name>Payroll Application</web-resource-name>
        <url-pattern>/payroll/*</url-pattern>
        <http-method>GET</http-method>
        <http-method>POST</http-method>
      </web-resource-collection>
    <auth-constraint>
      <role-name>manager</role-name>
    </auth-constraint>
  </security-constraint>
</web-app>
```

When the browser tries to load a resource that is secured by constraints in the web.xml file, the browser either challenges the user (if you are using BASIC authentication) or forwards the user to a login page (with FORM authentication).

With BASIC authentication, the user enters a username and password in a popup box that the browser creates. To specify that an application uses BASIC authentication, you use the `login-config` element and its `auth-method` subelement in the web application's web.xml file, as the following example shows:

```
<web-app>
  <login-config>
    <auth-method>BASIC</auth-method>
    <realm-name>Managers</realm-name>
  </login-config>
  ...
</web-app>
```

With FORM authentication, you must code the page that accepts the username and password, and submit them as FORM variables named `j_username` and `j_password`. This form can be implemented in HTML or as a Flex application or anything that can submit a form.

When you configure FORM authentication, you can specify both a login form and an error form in the web.xml file, as the following example shows:

```
<web-app>
  <login-config>
    <auth-method>FORM</auth-method>
    <form-login-config>
      <form-login-page>/login.htm</form-login-page>
      <form-error-page>/loginerror.htm</form-error-page>
    </form-login-config>
  </login-config>
</web-app>
```

You submit the results of the form validation to the `j_security_check` action. The server executing the application recognizes this action and processes the form.

A simple HTML-based form might appears as follows:

```
<form method="POST" action="j_security_check">
  <table>
    <tr><td>User</td><td><input type=text name="j_username"></tr>
    <tr><td>Password</td><td><input type=password name="j_password"></tr>
  </table>
  <input type=submit>
</form>
```

The results are submitted to the container's JAAS system with base-64 encoding, which means they can be read by anyone that can view the TCP/IP traffic. Use encryption to prevent these so-called "man-in-the-middle" attacks. In both BASIC and FORM authentication, if the user accessed the resource through SSL, the username and password submission are encrypted, as is all traffic during that exchange.

After it is complete, the container populates the browser's security context and provides or denies access to the resource. Flash Player inherits the security context of the underlying browser. As a result, when you make a data service call, the established credentials are used.

When a user fails an authentication attempt with invalid credentials, be sure not to return information about which item was incorrect. Instead, use a generic message such as "Your login information was invalid."

For application-server specific information about using custom authentication with Flex Data Services, you can use the examples in *flex_install_dir*/resources/security/examples.

Using RPC services

You can use the RPC services classes—RemoteObject, HTTPService, and WebService—not only to control access to the data that goes into an MXML page, but also to control the data and actions that flow out of it. You can also use service authentication to allow only certain users to perform certain actions. For example, if you have an application that allows employee data to be modified through a RemoteObject call, use RemoteObject authentication to make sure that only managers can change the employee data.

A service-based architecture makes it easy to implement several different security models for your Flex application. You can use programmatic security to limit access to services, or you can apply declarative security constraints to entire services.

Accessing RPC services with Flex tags such as the `<mx:WebService>` and `<mx:HTTPService>` tags is possible with and without Flex Data Services. However, if you are not running Flex Data Services, your Flex application's SWF file must connect to the service directly, which means that it can encounter security-based limitations. This section describes the security restraints applied to Flex applications when they access RPC services with and without Flex Data Services.

Connecting to RPC services with Flex Data Services

When you run Flex Data Services, you can use the proxy and defined destinations to connect to RPC services such as a web service. Destinations are configured in the Flex services configuration files. You do not have to store information about the destinations in the Flex application itself.

Typically, you set the `use-proxy` compiler option to `true` when you are accessing the HTTPService and WebService services with Flex Data Services. These are the only services that use the Proxy Service. When you do this, you can use the default destinations that are configured for you, and you no longer have to provide a crossdomain.xml file on the target domain because the proxy is making the connection for you.

Connecting to RPC services without Flex Data Services

When you are not running Flex Data Services, you cannot use the proxy functionality or the named destination functionality. In this case, destinations must be configured entirely in the Flex application (and not in configuration files); the component must communicate directly with the RPC service.

In addition, you must set the `use-proxy` compiler option to `false` when you compile the application.

When `use-proxy` is `false`, one of the following must be `true`:

- The RPC is in the same domain as the Flex application that calls it.
- The RPC's host system has a crossdomain.xml file that explicitly allows access from the Flex application's domain.

Using secured services

Secured services are services that are protected by resource constraints. The service itself behaves as a resource that needs authentication and the container defines its URL pattern as requiring authorization.

You might have a protected Flex application that calls a protected resource. In this case, with BASIC authentication and a proxied destination, the user's credentials are passed through to the service. The user only has to log on once when they first start the Flex application, and not when the application attempts to access the service.

Without a proxy, the user is challenged to enter their credentials a second time when the application attempts to access the service.

When you use secured services, keep the following in mind:

- If possible, use HTTPS for your services when you use authentication. In BASIC and custom authentication, user names and passwords are sent in a base-64 encoding. Using base-64 encoding hides the data only from plain view; HTTPS actually encrypts the data. You can use HTTPS in these cases by making sure HTTPS is set up on your server and by adding a protocol attribute with the value `https` on the service, and by adding a crossdomain.xml file.
- To ensure that the WebService and HTTPService endpoints are secure, use a browser window to access the URL you are trying to secure. This should always bring up a BASIC authentication prompt.
- If the BASIC or custom login box appears but you can't log in, make sure that the users and roles were added correctly to your application server. This is often an error-prone task that is overlooked as the source of the problem.

Using data services

The Flex Data Management Service provides data synchronization between application tiers, real-time data updates, data replication, occasionally connected application services, and integration with data sources through adapters. This feature lets you create applications that work with distributed data, and lets you manage large collections of data and nested data relationships, such as one-to-one and one-to-many relationships.

The Flex messaging service provides messaging services for collaborative and real-time applications. This feature lets you create applications that can send messages to and receive messages from other applications, including Flex applications and Java Message Service (JMS) applications.

For information about configuring security for the Flex data services, see the *Flex 2 Developer's Guide*.

Making other connections

Flash Player can connect to servers, services, and load data from sources other than RPC services. This section describes some of these sources and the security issues regarding them.

Using RTMP

Flash Player uses the Real-Time Messaging Protocol (RTMP) for client-server communication. This is a TCP/IP protocol designed for high-performance transmission of audio, video, and data messages. RTMP sends unencrypted data, including authentication information (such as a name and a password).

Although RTMP in and of itself does not offer security features, Flash communications applications can perform secure transactions and secure authentication through an SSL-enabled web server.

Flash Player also provides support for versions of RTMP that are tunneled through HTTP and HTTPS. RTMP refers to RTMP transmitted within an HTTP wrapper, and RTMPS is RTMP transmitted within an HTTPS wrapper.

Using sockets

Sockets let you read and write raw binary or XML data with a connected server. Sockets transmit over TCP. Because of this, Flash Player cannot take advantage of the built-in encryption capabilities of the browser. However, you can use encryption algorithms written in ActionScript to protect the data that is being communicated.

Cross-domain access to socket and XML socket connections is disabled by default. Access to socket connections in the same domain of the SWF file on ports lower than 1024 is also disabled by default. You can permit access to these connections by serving a cross-domain policy file from any of the following locations:

- The same port as the main socket connection
- A different port
- The HTTP server on port 80 in the same domain as the socket server

For more information, see the Socket and XMLSocket classes in *Flash ActionScript Language Reference*.

Using the LocalConnection class

The LocalConnection class lets you develop SWF files that can send instructions to each other. LocalConnection objects can communicate only among SWF files that are running on the same client computer, but they can be running in different applications—for example, a SWF file running in a browser and a SWF file running in a projector. (A projector is a SWF file saved in a format that can run as a stand-alone application—that is, the projector doesn't require Flash Player to be installed since it is embedded inside the executable file.)

For every LocalConnection communication, there is a sender SWF file and a listener SWF file. The simplest way to use a LocalConnection object is to allow communication only between LocalConnection objects located in the same domain because you won't have security issues.

Applications served from different domains that need to be able to make LocalConnection calls to each other must be granted cross-domain LocalConnection permissions. To do this, the listener must allow the sender permission by using the `LocalConnection.allowDomain()` or `LocalConnection.allowInsecureDomain()` methods.

Adobe does not recommend using the `LocalConnection.allowInsecureDomain()` method because allowing non-HTTPS documents to access HTTPS documents compromises the security offered by HTTPS. It is best that all Flash SWF files that make LocalConnection calls to HTTPS SWF files are served over HTTPS.

For more information about using the LocalConnection class, see *Programming ActionScript 3.0*.

To facilitate SWF-to-SWF communication, you can also use cross-scripting. For more information, see "Cross-scripting" on page 57.

Using SSL

A SWF file playing in a browser has many of the same security concerns as an HTML page being displayed in a browser. This includes the security of the SWF file while it is being loaded into the browser, as well as the security of communication between Flash and the server after the SWF file has loaded and is playing in the browser. In particular, data communication between the browser and the server is susceptible to being intercepted by third parties. The solution to this issue in HTML is to encrypt the communication between the client and server to make any data captured by third parties undecipherable and thus unusable. This encryption is done by using an SSL-enabled browser and server.

Because a SWF file running within a browser uses the browser for almost all of its communication with the server, it can take advantage of the browser's built-in SSL support. This lets communication between the SWF file and the server be encrypted. Furthermore, the actual bytes of the SWF file are encrypted while they are being loaded into the browser. Thus, by playing a SWF file within an SSL-enabled browser through an HTTPS connection with the server, you can ensure that the communication between Flash Player and the server is encrypted and secure.

The one exception to this security is the way Flash Player uses persistent sockets (through the ActionScript XMLSocket object), which does not use the browser to communicate with the server. Because of this, SWF files that use sockets cannot take advantage of the built-in encryption capabilities of the browser. However, you can use one-way encryption algorithms written in ActionScript to encrypt the data being communicated.

MD5 is a one-way encryption algorithm described in RFC 1321. This algorithm has been ported to ActionScript, which enables developers to secure one-way data by using the MD5 algorithm before it is sent from the SWF file to the server. For more information about RFC 1321, see www.faqs.org/rfcs/rfc1321.html or www.rsasecurity.com/rsalabs/faq/3-6-6.html.

Using secure endpoints with Flex Data Services

When you use Flex Data Services, you can use the predefined secure channels with proxied destinations. To use encryption with an RTMP channel (the channel that you use to connect to an RTMP endpoint), use the Secure RTMP channel instead of the standard RTMP channel. This channel supports real-time messaging and server-pushed broadcasts. This channel requires a digital certificate, and contains child elements for specifying a keystore filename and password.

To use HTTPS with an AMF channel, use the Secure AMF channel. You do this by specifying the flex.messaging.endpoints.SecureAMFEndpoint class in your channel definition. To use HTTPS with an HTTP channel, use the Secure HTTPS message channel. You do this by specifying the flex.messaging.endpoints.SecureHTTPEndpoint class in your channel definition.

For more information, see the *Flex 2 Developer's Guide*.

Using secure endpoints without Flex Data Services

To access HTTP services or web services through HTTPS without named destinations, you can specify the protocols using "https" in the `wsdl` or `url` properties; for example:

```
<mx:WebService url="https://myservice.com" .../>
<mx:HTTPService wsdl="https://myservice.com" .../>
```

By default, a SWF file served over an unsecure protocol, such as HTTP, cannot access other documents served over the secure HTTPS protocol, even when those documents come from the same domain. As a result, if you loaded the SWF file over HTTP but want to connect to the service through HTTPS, you must add `secure="false"` in the crossdomain.xml file on the services's server, as the following example shows:

```
<cross-domain-policy>
  <allow-access-from domain="*.mydomain.com" secure="false"/>
</cross-domain-policy>
```

If you loaded the SWF file over HTTPS, you do not have to make any changes.

Writing secure Flex applications

When you code a Flex application, keep the topics in this section in mind to ensure that the application you write is as secure as possible.

MXML tags with security restrictions

Some MXML tags trigger operations that require security settings. Operations that trigger security checks include:

- Referencing a URL that is outside the exact domain of the application that makes a request.
- Referencing an HTTPS URL when the application that makes the request is not served over HTTPS.
- Referencing a resource that is in a different sandbox.

In these cases, access rights must be granted through one of the permission-granting mechanisms such as the `allowDomain()` method or a crossdomain.xml file.

MXML tags that can trigger security checks include:

- Any class that extends the Channel class.
- RPC-related tags that use channels such as `<mx:WebService>`, `<mx:RemoteObject>`, and `<mx:HTTPService>`.
- Messaging tags such as `<mx:Producer>` and `<mx:Consumer>`.
- The `<mx:DataService>` tag.
- Tags that load SWF files such as `<mx:SWFLoader>`.

In addition to these tags and their underlying classes, many Flash classes trigger security checks including ExternalInterface, Loader, NetStream, SoundMixer, URLLoader, and URLRequest.

Disabling viewSourceURL

If you enabled the view source feature by setting the value of the `viewSourceURL` property on the `<mx:Application>` tag, you must be sure to remove it before you put your application into production.

This functionality applies only to Flex Builder users.

Remove sensitive information from SWF files

Flash applications share many of the same concerns and issues as web pages when it comes to protecting the security of data. Because the SWF file format is an open format, you can extract data and algorithms contained within a SWF file. This is similar to how HTML and JavaScript code can be easily viewed by users. However, SWF files make viewing the code more difficult. A SWF file is compiled and is not human-readable like HTML or JavaScript.

But security is not obtained through obscurity. A number of third-party tools can extract data from compiled SWF files. As a result, do not consider that any data, variables, or ActionScript code compiled into a Flash application are secure. You can use a number of techniques to secure sensitive information and still make it available for use in your SWF files.

To help ensure a secure environment, use the following general guidelines:

- Do not include sensitive information, such as user names, passwords, or SQL statements in SWF files.
- Do not use client-side username and password checks for authentication.
- Remove debug code, unused code, and comments from code before compiling to minimize the amount of information about your application that is available to someone with a decompiler or a debugger version of Flash Player.
- If your SWF file needs access to sensitive information, load the information into the SWF file from the server at run time. The data will not be part of the compiled SWF file and thus cannot be extracted by decompiling the SWF file. Use a secure transfer mechanism, such as SSL, when you load the data.
- Implement sensitive algorithms on the server instead of in ActionScript.
- Use SSL whenever possible.
- Only deploy your web applications from a trusted server. Otherwise, the server-side aspect of your application could be compromised.

Input validation

Input validation means ensuring that input is what it says it is or is what it is supposed to be. If your application is expecting name and address information, but it gets SQL commands, have a validation mechanism in your application that checks for and filters out SQL-specific characters and strings before passing the data to the execute method.

In many cases, you want users to provide input in TextInput, TextArea, and other controls that accept user input. If you use the input from these controls in operations inside the application, make sure that the input is free of possible malicious characters or code.

One approach to enforcing input validation is to use the Flex validator classes by using the `<mx:Validator>` tag or the tag for the appropriate validator type. Validators ensure that the input conforms to a predetermined pattern. For example, the NumberValidator class ensures that a string represents a valid number. This validator can ensure that the input falls within a given range (specified by the `minValue` and `maxValue` properties), is an integer (specified by the `domain` property), is non-negative (specified by the `allowNegative` property), and does not exceed the specified precision.

In typical client-server environments, data validation occurs on the server after data is submitted to it from the client. One advantage of using Flex validators is that they execute on the client, which lets you validate input data before transmitting it to the server. By using Flex validators, you eliminate the need to transmit data to and receive error messages back from the server, which improves the overall responsiveness of your application.

You can also write your own ActionScript filters that remove potentially harmful code from input. Common approaches include stripping out dollar sign ($), quotation mark ("), semi-colon (;) and apostrophe (') characters because they have special meaning in most programming languages. Because Flex also renders HTML in some controls, also filter out characters that can be used to inject script into HTML, such as the left and right angle brackets (“<” and “>”), by converting these characters to their HTML entities “<” and “>”. Also filter out the left and right parentheses (“(”and “)”) by translating them to “(” and “)”, and the pound sign (“#”) and ampersand (“&”) by translating them to “#” (#) and “&” (&).

Another approach to enforcing input validation is to use strongly-typed, parameterized queries in your SQL code. This way, if someone tries to inject malicious SQL code into text that is used in a query, the SQL server will reject the query.

For more information on potentially harmful characters and conversion processes, see http://www.cert.org/tech_tips/malicious_code_mitigation.html.

For more information about validators, see Chapter 40, “Validating Data,” in *Flex 2 Developer’s Guide*.

ActionScript

This section introduces some ways to try to make your use of ActionScript more secure.

Handling errors

The SecurityError exception is thrown when some type of security violation takes place. Security errors include:

- An unauthorized property access or method call was made across a security sandbox boundary.
- An attempt was made to access a URL not permitted by the security sandbox.
- A socket connection was attempted to an unauthorized port number, for example, a port below 1024, without a policy file present.
- An attempt was made to access the user's camera or microphone, and the request to access the device was denied by the user.

Flash Player dispatches SecurityErrorEvent objects to report the occurrence of a security error. Security error events are the final events dispatched for any target object. This means that any other events, including generic error events, are not dispatched for a target object that experiences a security error.

Your event listener can access the SecurityErrorEvent object's `text` property to determine what operation was attempted and any URLs that were involved, as the following example shows:

```
<?xml version="1.0"?>
<!-- security/SecurityErrorExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
    <mx:Script><![CDATA[
        import flash.net.URLLoader;
        import flash.net.URLRequest;
        import flash.events.SecurityErrorEvent;
        import mx.controls.Alert;

        private var loader:URLLoader = new URLLoader();

        private function initApp():void {
            loader.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
securityErrorHandler);
        }

        private function triggerSecurityError():void {
            // This URL is purposefully broken so that it will trigger a
            // security error.
            var request:URLRequest = new URLRequest("http://
www.[yourDomain].com");

            // Triggers a security error.
            loader.load(request);
        }

       private function securityErrorHandler(event:SecurityErrorEvent):void
{
            Alert.show("A security error occurred! Check trace logs for
details.");
            trace("securityErrorHandler: " + event.text);
        }
    ]]></mx:Script>

    <mx:Button id="b1" label="Click Me To Trigger Security Error"
click="triggerSecurityError()"/>

</mx:Application>
```

If no event listeners are present, the debugger version of Flash Player automatically displays an error message that contains the contents of the text property.

In general, try to wrap methods that might trigger a security error in a try/catch block. This prevents users from seeing information about destinations or other properties that you might not want to be visible.

Suppressing debug output

Flash Player writes debug output from a `trace()` method or the Logging API to a log file on the client. Any client can be running the debugger version of Flash Player. As a result, remove calls to the `trace()` method and Logging API calls that produce debugging output so that clients cannot view your logged information.

If you use the Logging API in your custom components and classes, set the value of the LogEventLevel to `NONE` before compilation, as the following example shows:

```
myTraceTarget.level = LogEventLevel.NONE;
```

For more information about the Logging API, see "Using the Logging API" on page 252.

Using host-based authentication

IP addresses and HTTP headers are sometimes used to perform host-based authentication. For example, you might check the Referer header or the client IP address to ensure that a request comes from a trusted source.

However, request headers such as Referer can be spoofed easily. This means that clients can pretend to be something they are not by settings headers or faking IP addresses. The solution to the problem of client spoofing is to not use HTTP header data as an authentication mechanism.

Using passwords

Using passwords in your Flex application is a common way to protect resources from unauthorized access. Test the validity of the password on the server rather than the client, because the client has access to all the logic in the local SWF file.

Never store passwords locally. For example, do not store username and password combinations in local SharedObjects. These are stored in plain-text and unencrypted, just as cookie files are. Anyone with access to the user's computer can access the information inside a SharedObject.

To ensure that passwords are transmitted from the client to the server safely, enforce the use of SSL or some other secure transport-level protocol.

When you ask for a password in a TextArea or TextInput control, set the `displayAsPassword` property to `true`. This displays the password as asterisks as it is typed.

Storing persistent data with the SharedObject class

Flash Player supports persistent shared objects through the SharedObject class. The SharedObject class stores data on users' computers. This data is usually local, meaning that it was obtained with the `SharedObject.getLocal()` method. You can also create persistent remote data with the SharedObject class; this requires Flash Media Server (formerly Flash Communication Server).

Each remote sandbox has an associated store of persistent SharedObject directory on the client. For example, when any SWF from domain1.com reads or writes data with the SharedObject class, Flash Player reads or writes that object in the domain1.com object store. Likewise for a SWF from domain2.com, Flash Player uses the domain2.com store. To avoid name collisions, the directory path defaults to the full path in the URL of the creating SWF file. This process can be shortened by using the `localPath` parameter of the `SharedObject.getLocal()` method, which allows other SWF files from the same domain to access a shared object after it is created.

Every domain has a maximum amount of data that a SharedObject class can save in the object store. This is an allocation of the user's disk space in which applications from that domain can store persistent data. Users can change the quota for a domain at any time by choosing Settings from the Flash Player context menu. When an application tries to store data with a SharedObject class that causes Flash Player to exceed its domain's quota, a dialog box appears, asking the user whether to increase the domain quota.

Configuring client security settings

Some security control features in Flash Player target user choices, and some target the modern corporate and enterprise environments, such as when the IT department would like to install Flash Player across the enterprise but has concerns about IT security and privacy. To help address these types of requirements, Flash Player provides various installation-time configuration choices. For example, some corporations do not want Flash Player to have access to the computer's audio and video hardware; other environments do not want Flash Player to have any read or write access to the local file system.

Three groups can make security choices: the application author (using developer controls), the administrative user (using administrator controls), and the local user (with user controls).

This section describes the ways to configure Flash Player's security settings.

About the mm.cfg file

You configure the debugger version of Flash Player by using the settings in the mm.cfg text file. You must create this file when you first configure the debugger version of Flash Player.

The settings in this file let you enable or disable `trace()` logging, set the location of the `trace()` file's output, and configure client-side error and warning logging.

For more information, see "Configuring the debugger version of Flash Player" on page 248.

About the mms.cfg file

The primary purpose for the Macromedia® Security Configuration file (mms.cfg) is to support the corporate and enterprise environments where the IT department wants to install Flash Player across the enterprise, while enforcing some common global security and privacy settings (supported with installation-time configuration choices).

On operating systems that support the concept of user security levels, the file is flagged as requiring system administrator (or root) permissions to modify or delete it.

- On Mac OS X systems using mms.cfg, the security configuration file is located at /Library/Application Support/Macromedia/mms.cfg.
- On Microsoft Windows, the file is located in the Macromedia Flash Player folder within the system directory (for example, C:\winnt\system32\macromed\flash\mms.cfg on a default Windows XP installation).

You can use this file to configure security settings that deal with data loading, privacy, and local file access. The settings include:

- `FileDownloadDisable`
- `FileUploadDisable`
- `LocalStorageLimit`
- `AVHardwareDisable`

For a complete list of options and their descriptions, see http://www.adobe.com/devnet/flashplayer/articles/flash_player_8_security.pdf.

About FlashPlayerTrust files

Flash Player provides a way for administrative users to register certain local files so that they are always loaded into the local-trusted sandbox. Often an installer for a native application or an application that includes many SWF files will do this. Depending on whether Flash Player will be embedded in a nonbrowser application, one of two strategies can be appropriate: register SWF files and HTML files to be trusted, or register applications to be trusted. Only applications that embed the browser plug-ins can be trusted—the stand-alone players and standard browsers do not check to see if they were trusted.

The installer creates files in a directory called FlashPlayerTrust. These files list paths of trusted files. This directory, known as the Global Flash Player Trust directory, is alongside the mms.cfg file, in the following location, which requires administrator access:

- Windows: system\Macromed\Flash\FlashPlayerTrust (for example, C:\winnt\system32\Macromed\Flash\FlashPlayerTrust)
- OS X: app support/Macromedia/FlashPlayerTrust (for example, /Library/Application Support/Macromedia/FlashPlayerTrust)

These settings affect all users of the computer. If an installer is installing an application for all users, the installer can register its SWF files as trusted for all users.

For more information about FlashPlayerTrust files, see http://www.adobe.com/devnet/flashplayer/articles/flash_player_8_security.pdf.

About the Settings Manager

The Settings Manager allows the individual user to specify various security, privacy, and resource usage settings for Flash applications executing on their client computer. For example, the user can control application access to select facilities (such as their camera and microphone), or control the amount of disk space allotted to a SWF file's domain. The settings it manages are persistent and controlled by the user.

The user can indicate their personal choices for their Flash Player settings in a number of areas, either globally (for Flash Player itself and all Flash applications) or specifically (applying to specific domains only). To designate choices, the user can select from the six tab categories along the top of the Settings Manager dialog box:

- Global Privacy Settings
- Global Storage Settings
- Global Security Settings
- Flash Player Update Settings
- Privacy Settings for Individual Websites

- Storage Settings for Individual Websites

To access the Settings Manager for your Flash Player:

1. Open an application in Flash Player.
2. Right-click and select Settings.

   The Adobe Flash Player Settings dialog box appears.
3. Select the Privacy tab (on the far left).
4. Click the Advanced button.

   Flash Player launches a new browser window and loads the Settings Manager help page.

Other resources

The following table lists resources that are useful in understanding the Flash Player security model and implementing security in your Flex applications:

Resource name	Location
Security Topic Center	http://www.adobe.com/devnet/security
Security Advisories	http://www.adobe.com/support/security
Flash Player Security & Privacy	http://www.adobe.com/products/flashplayer/security
Security Resource Center	http://www.adobe.com/resources/security
Flash Player 9 Security white paper	http://www.adobe.com/go/fp9_0_security
"Flash Player Security" chapter in *Programming ActionScript 3.0*	http://www.adobe.com/go/progAS3_security
"Networking and Communications" chapter in *Programming ActionScript 3.0*.	http://www.adobe.com/go/AS3_networking_and_communications
Security Changes in Flash Player 8	http://www.adobe.com/devnet/flash/articles/fplayer8_security.html
Security Changes in Flash Player 7	http://www.adobe.com/devnet/flash/articles/fplayer_security.html
Understanding Service Authentication	http://www.adobe.com/devnet/flex/articles/security_framework_print.html
Settings Manager	http://www.macromedia.com/support/documentation/en/flashplayer/help/settings_manager.html

CHAPTER 5

Optimizing Flex Applications

5

After you have a working application, you can explore ways to make that application download faster and perform better. This topic describes some techniques that you can use to improve your application's performance.

Contents

About performance 88

Improving client-side performance. 88

Improving server-side performance 123

Improving Flex Charting component performance 127

About performance

The two distinct aspects of Adobe Flex performance that you must consider are client-side performance and server-side performance.

Client-side performance The responsiveness of the user interface when it runs on Adobe Flash Player 9 on a user's computer. This responsiveness depends on how you wrote the application, the application's complexity, and the client's operating environment (network, operating system, and hardware). For details about improving client-side performance, see "Improving client-side performance" on page 88.

Server-side performance The responsiveness and scalability attributes of servers running the Flex application in production. The server-side performance of Flex depends on the operating environment, caching and compilation settings, the way you wrote the application, the number of clients accessing the server, and the size and complexity of the data being transported between the client and the server. Server-side performance can also be attributed to data sources that the server accesses to provide data to a Flex application. For details about improving server-side performance, see "Improving server-side performance" on page 123.

Improving client-side performance

Tuning software to achieve maximum performance is not an easy task. You must commit to producing efficient implementations and monitor software performance continuously during the software development process.

This section describes some general guidelines for testing applications for performance. It also describes some techniques you can use to do the actual testing, such as using the `getTimer()` method and checking initialization time.

Before you begin actual testing, you should understand some of the influences that client settings can have on performance testing. For more information, see "Configuring the client environment" on page 94.

General guidelines

You can use the following general guidelines when you improve your application and the environment in which it runs:

- Set performance targets early in the software design stage. If possible, try to estimate an acceptable performance target early in the application development cycle. Certain usage scenarios dictate the performance requirements. It would be disappointing to fully implement a product feature and then find out that it is too slow to be useful.

- Understand performance characteristics of the application framework. Some components and operations in the Flex framework are more efficient than others. This topic introduces you to some strategies for using the optimal methods of using the Flex framework.
- Understand performance characteristics of the application code. In medium-sized or large-sized projects, it is common for a product feature to use codes or components written by other developers or by third-party vendors. Knowing what is slow and what is fast in dependent components and code is essential in getting the design right.
- Do not attempt to test a large application's performance all at once. Rather, test small pieces of the application so that you can focus on the relevant results instead of being overwhelmed by data.
- Test the performance of your application early and often. It is always best to identify problem areas early and resolve them in an iterative manner, rather then trying to shove performance enhancements into existing, poorly performing code at the end of your application development cycle.
- Avoid optimizing code too early. Even though early testing can highlight performance hot spots, refrain from fixing them while you are still developing those areas of the application; doing so might unexpectedly delay the implementation schedule. Instead, document the issues and prioritize all the performance issues as soon as your team finishes the feature implementation.

Testing applications for performance

This section provides some techniques that you can use to test start-up and run-time performance of your Flex applications. You can use the techniques described in this section to monitor memory consumption, time application initialization, and time events.

Calculating application initialization time

One approach to performance profiling is to use code to gauge the start-up time of your application. This can help identify bottlenecks in the initialization process, and reveal deficiencies in your application design, such as too many components or too much reliance on nested containers.

The `getTimer()` method in flash.utils returns the number of milliseconds that have elapsed since Flash Player was initialized. This indicates the amount of time since the application began playing. The Timer class provides a set of methods and properties that you can use to determine how long it takes to execute an operation.

Before each update of the screen, Flash Player calls the set of functions that are scheduled for the update. Sometimes, a function should be called in the next update to allow the rest of the code scheduled for the current update to execute. You can instruct Flash Player to call a function in the next update by using the callLater() method. This method accepts a function pointer as an argument. The method then puts the function pointer on a queue, so that the function is called the next time the player dispatches either a render event or an enterFrame event.

The following example records the time it takes the Application object to create, measure, lay out, and draw all of its children. This example does not include the time to download the SWF file to the client, or to perform any of the server-side processing, such as checking the Flash Player version, checking the SWF file cache, and so on.

```
<?xml version="1.0"?>
<!-- optimize/ShowInitializationTime.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="callLater(showInitTime)">
    <mx:Script><![CDATA[
        import flash.utils.Timer;

        [Bindable]
        public var t:String;
        private function showInitTime():void {
            // Record the number of ms since the player was initialized.
            t = "App startup: " + getTimer() + " ms";
        }
    ]]></mx:Script>
    <mx:Label id="l1" text="{t}"/>
</mx:Application>
```

This example uses the callLater() method to delay the recording of the startup time until after the application finishes and the first screen updates. The reason that the showInitTime function pointer is passed to the callLater() method is to make sure that the application finishes initializing itself before calling the getTimer() method.

For more information on using the callLater() method, see "Using the callLater() method" on page 150.

Calculating elapsed time

Some operations take longer than others. Whether these operations are related to data loading, instantiation, effects, or some other factor, it's important for you to know how long each aspect of your application takes.

You can calculate elapsed time by using the `getTimer()` method. The following example calculates the instantiation time for all the form elements:

```
<?xml version="1.0" encoding="iso-8859-1"?>
<!-- optimize/ShowElapsedTime.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="init()">
    <mx:Script><![CDATA[
        [Bindable]
        public var dp:Array = [
            {food:"apple", type:"fruit", color:"red"},
            {food:"potato", type:"vegetable", color:"brown"},
            {food:"pear", type:"fruit", color:"green"},
            {food:"orange", type:"fruit", color:"orange"},
            {food:"spinach", type:"vegetable", color:"green"},
            {food:"beet", type:"vegetable", color:"red"}
        ];
        [Bindable]
        public var t:String;

        public var sTime:Number;

        private function init():void {
            f1.addEventListener("preinitialize", startTime, true);
            f1.addEventListener("creationComplete", endTime, true);
        }

        private function startTime(e:Event):void {
            // Get the time when the preinitialize event is dispatched.
            sTime = getTimer();
        }

        private function endTime(e:Event):void {
            // Get the time when the creationComplete event is dispatched.
            var eTime:Number = getTimer();

            // Calculate initialization time by subtracting the number of
            // milliseconds between the preinitialize and creationComplete
events.
            var totalTime:Number = eTime - sTime;

           // Use target rather than currentTarget because these events are
          // triggered by each child of the Form control during the capture
            // phase.
            t = "Time to create " + e.target + ": " + totalTime + "ms";
        }

    ]]></mx:Script>

    <mx:Label id="l1" text="{t}"/>
```

```
    <mx:Form id="f1">
        <mx:FormHeading label="Sample Form" id="fh1"/>
        <mx:FormItem label="List Control" id="fi1">
            <mx:List dataProvider="{dp}" labelField="food" id="list1"/>
        </mx:FormItem>
        <mx:FormItem label="DataGrid control" id="fi2">
            <mx:DataGrid width="200" dataProvider="{dp}" id="dg1"/>
        </mx:FormItem>
        <mx:FormItem label="Date controls" id="fi3">
            <mx:DateChooser id="dc"/>
            <mx:DateField id="df"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Calculating memory usage

You use the `totalMemory` property in the System class to find out how much memory has been allocated to Flash Player on the client. The `totalMemory` property represents all the memory allocated to Flash Player, not necessarily the memory being used by objects. Depending on the operating system, Flash Player will be allocated more or less resources and will allocate memory with what is provided.

You can record the value of `totalMemory` over time by using a Timer class to set up a recurring interval for the timer event, and then listening for that event.

The following example displays the total amount of memory allocated (totmem) to Flash Player at 1-second intervals. This value will increase and decrease. In addition, this example shows the maximum amount of memory that had been allocated (maxmem) since the application started. This value will only increase.

```
<?xml version="1.0"?>
<!-- optimize/ShowTotalMemory.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initTimer()">
    <mx:Script><![CDATA[
        import flash.utils.Timer;
        import flash.events.TimerEvent;

        [Bindable]
        public var time:Number = 0;
        [Bindable]
        public var totmem:Number = 0;
        [Bindable]
        public var maxmem:Number = 0;



        public function initTimer():void {
            // The first parameter is the interval (in milliseconds). The
           // second parameter is number of times to run (0 means infinity).
            var myTimer:Timer = new Timer(1000, 0);
            myTimer.addEventListener("timer", timerHandler);
            myTimer.start();
        }

        public function timerHandler(event:TimerEvent):void {
            time = getTimer()
            totmem = flash.system.System.totalMemory;
            maxmem = Math.max(maxmem, totmem);
        }
    ]]></mx:Script>
    <mx:Form>
        <mx:FormItem label="Time:">
            <mx:Label text="{time} ms"/>
        </mx:FormItem>
        <mx:FormItem label="totalMemory:">
            <mx:Label text="{totmem} bytes"/>
        </mx:FormItem>
        <mx:FormItem label="Max. Memory:">
            <mx:Label text="{maxmem} bytes"/>
        </mx:FormItem>
    </mx:Form>
</mx:Application>
```

Configuring the client environment

When testing applications for performance, it is important to configure the client properly. This section describes some issues you might encounter when configuring the client.

Choosing the version of Flash Player

When you test your applications for performance, ensure that you use the standard version of Flash Player rather than the debugger version of Flash Player, if possible. The debugger version of Player provides support for the `trace()` method and the Logging API. Using logging or the `trace()` method can significantly slow player performance, because the player must write log entries to disk while running the application.

If you do use the debugger version of Flash Player, you can disable logging and the `trace()` method by setting the `TraceOutputFileEnable` property to 0 in your mm.cfg file. You can keep `trace()` logging working, but disable the Logging API that you might be using in your application, by setting the logging level of the TraceTarget logging target to `NONE`, as the following example shows:

```
myLogger.log(LogEventLevel.NONE, s);
```

For performance testing, consider writing run-time test results to text components in the application rather than calling the `trace()` method so that you can use the standard version of Flash Player and not the debugger version of Flash Player.

For more information about configuring `trace()` method output and logging, see Chapter 11, “Logging,” on page 245.

Disabling SpeedStep

If you are running performance tests on a Windows laptop computer, disable Intel SpeedStep functionality. SpeedStep toggles the speed of the CPU to maximize battery life. SpeedStep can toggle the CPU at unpredictable times, which makes the results of a performance test less accurate than they would otherwise be.

To disable SpeedStep:

1. Select Start > Settings > Control Panel.
2. Double-click the Power Settings icon.

   The Power Options Properties dialog box displays.
3. Select the Power Schemes tab.
4. Select High System Performance from the Power Schemes drop-down box.
5. Click OK.

Changing timeout length

When you test your application, be aware of the `scriptTimeLimit` property. If an application takes too long to initialize, Flash Player warns users that a script is causing Flash Player to run slowly and prompts the user to abort the application. If this is the situation, you can set the `scriptTimeLimit` property of the `<mx:Application>` tag to a longer time so that the Flex application has enough time to initialize.

However, the default value of the `scriptTimeLimit` property is 60 seconds, which is also the maximum, so you can only increase the value if you have previously set it to a lower value. You rarely need to change this value.

The following example sets the `scriptTimeLimit` property to 30:

```
<?xml version="1.0"?>
<!-- optimize/ChangeScriptTimeLimit.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
scriptTimeLimit="30">
    <!-- Empty application -->
</mx:Application>
```

Preventing client-side caching

When you test performance, ensure that you are not serving files from the local cache to Flash Player. Otherwise, this can give false results about download times. Also, during development and testing, you might want to change aspects of the application such as embedded images, but the browser continues to use the old images from your cache.

If the date and time in the If-Modified-Since request header matches the data and time in the Last-Modified response header, the browser loads the SWF file from its cache. Then the server returns the 304 Not Modified message. If the Last-Modified header is more recent, the server returns the SWF file.

You can use the following techniques to disable client-side caching:

- Delete the Flex files from the browser’s cache after each interaction with your application. Browsers typically store the SWF file and other remote assets in their cache. On Microsoft Internet Explorer, for example, you can delete all the files in c:/Documents and Settings/username/Local Settings/Temporary Internet Files to force a refresh of the files on the next request.

- Set the HTTP headers for the SWF file request to prevent caching of the SWF file on the client. The following example shows how to set headers that prevent caching in JSP:

```
// Set Cache-Control to no-cache.
response.setHeader("Cache-Control", "no-cache");
// Prevent proxy caching.
response.setHeader("Pragma", "no-cache");
// Set expiration date to a date in the past.
response.setDateHeader("Expires", 946080000000L); //Approx Jan 1, 2000
// Force always modified.
response.header("Last-Modified", new Date());
```

- Invalidate all items in the client cache by making a change to the flex-config.xml file and saving it. This file stores all the options that the web-tier compiler uses. When the file changes, Adobe Flex Data Services recompiles the application SWF file the next time the MXML file is requested.
- Delete the cache.dep file in the /WEB-INF/flex/ directory. This forces the server to invalidate all items in its cache.
- Use the Flex Data Services server's caching functionality to control client-side caching. You can set the number of files to cache to 0, which means that Flex Data Services does not cache any files. Every request results in a recompilation. Because the SWF file has a newer data-time stamp, the server sends a new SWF file to the requesting client and the file is never read from the browser's cache. To do this, you set the value of the `content-size` property in the flex-webtier-config.xml file to 0 as the following example shows:

```
<cache>
  <content-size>0</content-size>
  <http-maximum-age>1</http-maximum-age>
  <file-watcher-interval>1</file-watcher-interval>
</cache>
```

- Add the `recompile=true` query string parameter to your request to force a recompile of the application. For more information, see "Compiler configuration" on page 168.

Using the JRun sniffer

The sniffer can help you see the transfer times and what files are being transmitted during the request and response of the Flex application. The sniffer is included with the integrated JRun server. You optionally install this server when you install Flex Data Services. To use the sniffer, launch the sniffer utility in the *flex_install_dir*/jrun4/bin directory.

For more information, see "Using the sniffer" on page 411.

Reducing SWF file sizes

You can improve initial user experience by reducing the time it takes to start an application. Part of this time is determined by the download process, where the SWF file is returned from the server to the client. The smaller the SWF file, the shorter the download wait. In addition, reducing the size of the SWF file also results in a shorter application initialization time. Larger SWF files take longer to unpack in Flash Player.

The mxmlc compiler includes several options that can help reduce SWF file size.

Using the bytecode optimizer

The bytecode optimizer can reduce the size of the Flex application's SWF file by using bytecode merging and peephole optimization. Peephole optimization removes redundant instructions from the bytecode.

If you are using Flex Data Services, you can set the `optimize` option in the flex-config.xml file. If you are using Flex Builder or the mxlmc command-line compiler, you can set the `optimize` compiler option to `true`, as the following example shows:

```
mxmlc -optimize=true MyApp.mxml
```

The default value of the `optimize` option is `true`.

Disabling debugging

Disabling debugging can make your SWF files smaller. When debugging is enabled, the Flex compilers include line numbers and other navigational information in the SWF file that are only used in a debugging environment. Disabling debugging reduces functionality of the fdb command-line debugger and the debugger built into Flex Builder.

To disable debugging, set the `debug` compiler option to `false`. The default value for the mxmlc compiler is `false`. The default value for the compc compiler is `true`.

For more information about debugging, see Chapter 12, "Using the Command-Line Debugger," on page 269.

Using strict mode

When you set the `strict` compiler option to `true`, the compiler verifies that definitions and package names in `import` statements are used in the application. If the imported classes are not used, the compiler reports an error.

The following example shows some examples of when strict mode throws a compiler error:

```
package {
  import flash.utils.Timer; // Error. This class is not used.
  import flash.printing.* // Error. This class is not used.
  import mx.controls.Button; // Error. This class is not used.
  import mx.core.Application; // No error. This class is used.

  public class Foo extends Application {
  }

}
```

The `strict` option also performs compile-time type checking, which provides a small optimization increase in the application at run time.

The default value of the `strict` compiler option is `true`.

Examining linker dependencies

To find ways to reduce SWF file sizes, you can look at the list of ActionScript classes that are linked into your SWF file.

You can generate a report of linker dependencies by setting the `link-report` compiler option to `true`. The output of this compiler option is a report that shows linker dependencies in an XML format.

The following example shows the dependencies for the ProgrammaticSkin script as it appears in the linker report:

```
<script name="C:\flex2sdk\frameworks\libs\framework.swc(
    mx/skins/ProgrammaticSkin)" mod="1141055632000" size="5807">
  <def id="mx.skins:ProgrammaticSkin"/>
  <pre id="mx.core:IFlexDisplayObject"/>
  <pre id="mx.styles:IStyleable"/>
  <pre id="mx.managers:ILayoutClient"/>
  <pre id="flash.display:Shape"/>
  <dep id="String"/>
  <dep id="flash.geom:Matrix"/>
  <dep id="mx.core:mx_internal"/>
  <dep id="uint"/>
  <dep id="mx.core:UIComponent"/>
  <dep id="int"/>
  <dep id="Math"/>
  <dep id="Object"/>
  <dep id="Array"/>
  <dep id="mx.core:IStyleClient"/>
  <dep id="Boolean"/>
  <dep id="Number"/>
  <dep id="flash.display:Graphics"/>
</script>
```

The following table describes the tags used in this file:

Tag	Description
`<script>`	Indicates the name of a compilation unit used in the creation of the application SWF file. Compilation units must contain at least one public definition, such as a class, function, or namespace. The `name` attribute shows the origin of the script, either from a source file or from a SWC file (for example, frameworks.swc). If you set `keep-generated=true` on the command line, all classes in the generated folder are listed as scripts in this file. The `size` attribute shows the class' size, in bytes. The `mod` attribute shows the time stamp when the script was created.
`<def>`	Indicates the name of a definition. A definition, like a script, can be a class, function, or namespace.
`<pre>`	Indicates a definition that must be linked in to the SWF file before the current definition is linked in. This tag means *prerequisite*. For class definitions, this tag shows the direct parent class (for example, flash.events:Event), plus all implemented interfaces (for example, mx.core:IFlexDisplayObject and mx.managers:ILayoutClient) of the class.
`<dep>`	Indicates other definitions that this definition depends on (for example, String, _ScrollBarStyle, and mx.core:IChildList). This is a reference to a definition that the current script requires. Some script definitions have no dependencies, so the `<script>` tag might have no `<dep>` child tags.
`<ext>`	Indicates a dependency to an asset that was not linked in. These dependencies show up in the linker report when you use the `external-library-path`, `externs`, or `load-externs` compiler options to add assets to the SWF file.

You can examine the list of prerequisites and dependencies for your application definition. You do this by searching for your application's root MXML file by its name; for example, MyApp.mxml. You might discover that you are linking in some classes inadvertently. When writing code, it is common to make a reference to a class but not actually require that class in your application. That reference causes the referenced class to be linked in, and it also links in all the classes on which the referenced class depends.

If you look through the linker report, you might find that you are linking in a class that is not needed. If you do find an unneeded class, try to identify the linker dependency that is causing the class to be linked in, and try to find a way to rewrite the code to eliminate that dependency.

Avoiding initializing unused classes

Some common ways to avoid unnecessary references include avoiding initializing classes you do not use and performing type-checking with the `getQualifiedClassName()` method.

The following example checks if the class is a Button control. This example forces the compiler to include a Button in the SWF file, even if the child is not a Button control and the entire application has no Button controls.

```
<?xml version="1.0"?>
<!-- optimize/UnusedClasses.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="checkChildType()">
    <mx:Script><![CDATA[
    import mx.controls.Button;

    public function checkChildType():void {
        var child:DisplayObject = getChildAt(0);
        var childIsButton:Boolean = child is mx.controls.Button;
        trace("child is mx.controls.Button: " + childIsButton); // False.
    }
    ]]></mx:Script>

    <!-- This control is here so that the getChildAt() method succeeds. -->
    <mx:DataGrid/>

</mx:Application>
```

You can use the `getQualifiedClassName()` method to accomplish the same task as the previous example. This method returns a String that you can compare to the name of a class without causing that class to be linked into the SWF.

The following example does not create a linker dependency on the Button control:

```
<?xml version="1.0"?>
<!-- optimize/GetQualifiedClassNameExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        creationComplete="checkChildType()">
    <mx:Script><![CDATA[
    public function checkChildType():void {
        var child:DisplayObject = getChildAt(0);
        var childClassName:String = getQualifiedClassName(child);
        var childIsButton:Boolean = childClassName == "mx.controls::Button"
        trace("child class name = Button " + childIsButton);
    }
    ]]></mx:Script>

    <!-- This control is here so that the getChildAt() method succeeds. -->
    <mx:DataGrid/>

</mx:Application>
```

Externalizing assets

One method of reducing the SWF file size is to externalize assets; that is, to load the assets at run time rather than embed them at compile time. You can do this with assets such as images, SWF files, and sound files.

Embedded assets load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded assets also require you to recompile your applications whenever your asset changes.

The following example embeds the shapes.swf file into the Flex application at compile time:

```
<?xml version="1.0"?>
<!-- optimize/EmbedAtCompileTime.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Image source="@Embed(source='../assets/bird.gif')"/>
</mx:Application>
```

The following example loads the shapes.swf file into the Flex application at run time:

```
<?xml version="1.0"?>
<!-- optimize/EmbedAtRunTime.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Image source="../assets/bird.gif"/>
</mx:Application>
```

The only supported image type that you cannot load at run time is SVG. Flash Player requires that the compiler transcodes that file type at compile time. The player cannot transcode that file type at run time.

When you load SWF files from domains that are not the same as the loading SWF file, you must use a crossdomain.xml file or other mechanism to enable the proper permissions. For more information on using the crossdomain.xml file, see "Using cross-domain policy files" on page 65.

An alternative to reducing SWF file sizes by externalizing assets is to increase the SWF file size by embedding assets. By embedding assets such as images, sounds, and SWF files, you can reduce the network bandwidth and connections. The SWF file size increases, but the application requires fewer network connections to the server.

For information on loading assets, see Chapter 30, "Embedding Assets," in *Flex 2 Developer's Guide*.

Using character ranges for embedded fonts

By specifying a range of symbols that compose the face of an embedded font, you reduce the size of an embedded font. Each character in a font must be described; if you remove some of these characters, it reduces the overall size of the description information that Flex must include for each embedded font.

You can set the range of glyphs in the flex-config.xml file or in the `font-face` declaration in each MXML file. You specify individual characters or ranges of characters using the Unicode values for the characters, and you can set multiple ranges for each font declaration.

In CSS, you can set the Unicode range with the `unicodeRange` property, as the following example shows:

```
@font-face {
   src:url("../assets/MyriadWebPro.ttf");
   fontFamily: myFontFamily;
   unicodeRange:
      U+0041-U+005A, /* Upper-Case [A..Z] */
      U+0061-U+007A, /* Lower-Case a-z */
      U+0030-U+0039, /* Numbers [0..9] */
      U+002E-U+002E; /* Period [.] */
}
```

In the flex-config.xml file, you can set the Unicode range with the `<language-range>` block, as the following example shows:

```
<language-range>
   <lang>Latin I</lang>
   <range>U+0020,U+00A1-U+00FF,U+2000-U+206F,U+20A0-U+20CF,U+2100-U+2183</
   range>
</language-range>
```

For more information, see Chapter 19, "Using Fonts," in *Flex 2 Developer's Guide*.

Using multiple SWF files

One way to reduce the size of an application's file is to break the application up into logical parts that can be sent to the client and loaded over a series of requests rather than all at once. By breaking a monolithic application into smaller applications, users can interact with your application more quickly, but possibly experience some delays while the application is running.

One approach is to use the SWFLoader control. This technique can work with SWF files that add graphics or animations to an application, or SWF files that act as stand-alone applications inside the main application. If you import SWF files that require a large amount of user interaction, however, consider building them as custom components. SWF files produced with earlier versions of Flex or ActionScript may not work properly when loaded with the SWFLoader control.

Rather than loading SWF files into the main application with the SWFLoader control, consider having the SWF files communicate with each other as separate applications. You can do this with local SharedObjects, LocalConnection objects, or with the ExternalInterface API.

Another approach to loading multiple small SWF files rather than one large one is to use the HTML wrapper to provide a framework for loading the SWF files. The Samples Explorer, which is included with Flex 2 SDK, uses HTML frames to load many smaller applications, one at a time. >

Comparing dynamic and static linking

Most large applications use libraries of ActionScript classes and components. You must decide whether to use static or dynamic linking when using these libraries in your Flex applications.

When you use *static linking*, the compiler includes all components, classes, and their dependencies in the application SWF file when you compile the application. The result is a larger SWF file that takes longer to download but loads and runs quickly because all the code is in the SWF file. To compile your application that uses libraries and to statically link those definitions into your application, you use the `library-path` and `include-libraries` options to specify the locations of SWC files.

Dynamic linking is when some classes used by an application are left in an external file that is loaded at run time. The result is a smaller SWF file size for the main application, but the application relies on external files that are loaded during run time.

To dynamically link classes and components, you compile a library. You then instruct the compiler to exclude that library's contents from the application SWF file. You must still provide link-checking at compile time even though the classes are not going to be included in the final SWF file.

You use dynamic linking by creating component libraries and compiling them with your application by using the `external-library-path`, `externs`, or `load-externs` compiler options. These options instruct the compiler to exclude resources defined by their arguments from inclusion in the application, but to check links against them and prepare to load them at run time. The `external-library-path` option specifies SWC files or directories for dynamic linking. The `externs` option specifies individual classes or symbols for dynamic linking. The `load-externs` option specifies an XML file that describes which classes to use for dynamic linking. This XML file has the same syntax as the file produced by the `link-report` compiler option.

For more information about linking, see "About linking" on page 234. For more information about compiler options, see Chapter 9, "Using the Flex Compilers," on page 179.

Using RSLs to reduce SWF file size

One way to reduce the size of your application's SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time, but must be transferred only once to the client. These shared files are known as *Runtime Shared Libraries* (RSLs).

If you have multiple applications but those applications share a core set of components or classes, your users will be required to download those assets only once as an RSL. The applications that share the assets in the RSL use the same cached RSL as the source for the libraries as long as they are in the same domain. The resulting file size for your applications can be reduced. The benefits increase as the number of applications that use the RSL increases.

For more information, see Chapter 10, "Using Runtime Shared Libraries," on page 233.

Application coding

The MXML language provides a rich set of controls and classes that you can use to create interactive applications. This richness sometimes can reduce efficient performance. This section describes some techniques that a Flex developer can use to improve the run-time performance of the Flex application.

Object creation and destruction

Object creation is the task of instantiating all the objects in your application. These objects include controls, components, and objects that contain data and other dynamic information. Optimizing the process of object creation and destruction can result in significant performance gains.

No single task during application initialization takes up the most time. The best way to improve performance is to create fewer objects. You can do this by deferring the instantiation of objects, or changing the order in which they are created to improve perceived performance.

Using ordered creation

You can improve *perceived* startup time of your Flex application by ordering the creation of containers in the initial view. The default behavior of Flex is to create all containers and their children in the initial view, and then display everything at once. The user cannot interact with the application or see meaningful data until all the containers and their children are created.

In some cases, you can improve the user's initial experience by displaying the components in one container before creating the components in the next container. This process is called ordered creation.

To use ordered creation, you set the `creationPolicy` property of a container to `queued`, as the following example shows:

```
<?xml version="1.0"?>
<!-- optimize/QueuedPanels.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Panel id="panel1" creationPolicy="queued" width="100%"
height="33%">
        <mx:Button id="button1a"/>
        <mx:Button id="button1b"/>
    </mx:Panel>

    <mx:Panel id="panel2" creationPolicy="queued" width="100%"
height="33%">
        <mx:Button id="button2a"/>
        <mx:Button id="button2b"/>
    </mx:Panel>

    <mx:Panel id="panel3" creationPolicy="queued" width="100%"
height="33%">
        <mx:Button id="button3a"/>
        <mx:Button id="button3b"/>
    </mx:Panel>
</mx:Application>
```

This adds the container's children to a queue. Flash Player instantiates and displays all of the children within the first container in the queue before instantiating the children in the next container in the queue.

For more information on ordered creation, see "Using ordered creation" on page 143.

Using deferred creation

To improve the start-up time of your application, minimize the number of objects that are created when the application is first loaded. If a user-interface component is not initially visible at start up, create that component only when you need it. This is called deferred creation. Containers that have multiple views, such as an Accordion, provide built-in support for this behavior. You can use ActionScript to customize the creation order of multiple-view containers or defer the creation of other containers and controls.

To use deferred creation, you set the value of a component's `creationPolicy` property to `all`, `auto`, or `none`. If you set it to `none`, Flex does not instantiate a control's children immediately, but waits until you instruct Flex to do so. In the following example, the children of the VBox container are not be instantiated when the application is first loaded, but only after the user clicks the button:

```
<?xml version="1.0"?>
<!-- optimize/CreationPolicyNone.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[

        private function createButtons(e:Event):void {
            myVBox.createComponentsFromDescriptors();
        }

    ]]></mx:Script>

    <mx:Panel title="VBox with Repeater">
       <mx:VBox id="myVBox" height="100" width="125" creationPolicy="none">
            <mx:Button id="b1" label="Hurley"/>
            <mx:Button id="b2" label="Jack"/>
            <mx:Button id="b3" label="Sawyer"/>
        </mx:VBox>
    </mx:Panel>

    <mx:Button id="myButton" click="createButtons(event)" label="Create
Buttons"/>

</mx:Application>
```

You call methods such as `createComponentFromDescriptor()` and `createComponentsFromDescriptor()` on the container to instantiate its children at run time. For more information on using deferred instantiation, see "Using deferred creation" on page 134.

Destroying unused objects

Flash Player provides built-in garbage collection that frees up memory by destroying objects that are no longer used. To ensure that the garbage collector destroys your unused objects, remove all references to that object, including the parent's reference to the child.

On containers, you can call the `removeChild()` or `removeChildAt()` method to remove references to child controls that are no longer needed. The following example removes references to button instances from the myVBox control:

```
<?xml version="1.0"?>
<!-- optimize/DestroyObjects.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        private function destroyButtons(e:Event):void {
            myVBox.removeChild(b1);
            myVBox.removeChild(b2);
            myVBox.removeChild(b3);
        }
    ]]></mx:Script>

    <mx:Panel title="VBox with Repeater">
        <mx:VBox id="myVBox" height="100" width="125">
            <mx:Button id="b1" label="Hurley"/>
            <mx:Button id="b2" label="Jack"/>
            <mx:Button id="b3" label="Sawyer"/>
        </mx:VBox>
    </mx:Panel>

    <mx:Button id="myButton2" click="destroyButtons(event)" label="Destroy
Buttons"/>

</mx:Application>
```

You can clear references to unused variables by setting them to `null` in your ActionScript; for example:

```
myDataProvider = null
```

To ensure that destroyed objects are garbage collected, you must also remove event listeners on them by using the `removeEventListener()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- optimize/RemoveListeners.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp(event)">
    <mx:Script><![CDATA[
        private function initApp(e:Event):void {
            b1.addEventListener("click",myClickHandler);
            b2.addEventListener("click",myClickHandler);
            b3.addEventListener("click",myClickHandler);
        }

        private function destroyButtons(e:Event):void {
            b1.removeEventListener("click",myClickHandler);
            b2.removeEventListener("click",myClickHandler);
            b3.removeEventListener("click",myClickHandler);

            myVBox.removeChild(b1);
            myVBox.removeChild(b2);
            myVBox.removeChild(b3);
        }

        private function myClickHandler(e:Event):void {
            // Do something here.
        }
    ]]></mx:Script>

    <mx:Panel title="VBox with Repeater">
        <mx:VBox id="myVBox" height="100" width="125">
            <mx:Button id="b1" label="Hurley"/>
            <mx:Button id="b2" label="Jack"/>
            <mx:Button id="b3" label="Sawyer"/>
        </mx:VBox>
    </mx:Panel>

    <mx:Button id="myButton" click="destroyButtons(event)" label="Destroy
Buttons"/>

</mx:Application>
```

You cannot call the `removeEventListener()` method on an event handler that you added inline. In the following example, you cannot call `removeEventListener()` on b1's `click` event handler, but you can call it on b2's and b3's event handlers:

```
<?xml version="1.0"?>
<!-- optimize/RemoveSomeListeners.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp(event)">
    <mx:Script><![CDATA[
        private function initApp(e:Event):void {
            b2.addEventListener("click",myClickHandler);
            b3.addEventListener("click",myClickHandler);
        }

        private function destroyButtons(e:Event):void {
            b2.removeEventListener("click",myClickHandler);
            b3.removeEventListener("click",myClickHandler);

            myVBox.removeChild(b1);
            myVBox.removeChild(b2);
            myVBox.removeChild(b3);
        }

        private function myClickHandler(e:Event):void {
            // Do something here.
        }
    ]]></mx:Script>

    <mx:Panel title="VBox with Repeater">
        <mx:VBox id="myVBox" height="100" width="125">
           <mx:Button id="b1" label="Hurley" click="myClickHandler(event)"/
>
            <mx:Button id="b2" label="Jack"/>
            <mx:Button id="b3" label="Sawyer"/>
        </mx:VBox>
    </mx:Panel>

    <mx:Button id="myButton" click="destroyButtons(event)" label="Destroy
Buttons"/>

</mx:Application>
```

The *`weakRef`* parameter to the `addEventListener()` method provides you with some control over memory resources for listeners. A strong reference (when `weakRef` is `false`) prevents the listener from being garbage collected. A weak reference (when `weakRef` is `true`) does not. The default is `false`.

For more information about the `removeEventListener()` method, see Chapter 5, "Using Events," in the *Flex 2 Developer's Guide*.

Using styles

You use styles to define the look and feel of your Flex applications. You can use them to change the appearance of a single component, or apply them globally. Be aware that some methods of applying styles are more expensive than others. This section describes some of the ways you can increase your application's performance by changing the way you apply styles.

For more information about using styles, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*.

Reducing calls to the setStyle() method

Run-time cascading styles are very powerful, but use them sparingly and in the correct context. Calling the `setStyle()` method can be an expensive operation because the call requires notifying all the children of the newly-styled object. The resulting tree of children that must be notified can be quite large.

A common mistake that impacts performance is overusing or unnecessarily using the `setStyle()` method. In general, you only use the `setStyle()` method when you change styles on existing objects. Do not use it when you set up styles for an object for the first time. Instead, set styles in an `<mx:Style>` block, as style properties on the MXML tag, through an external CSS style sheet, or as global styles.

Some applications must call the `setStyle()` method during the application or object instantiation. If this is the case, call the `setStyle()` method early in the instantiation phase. Early in the instantiation phase means setting styles from the component or application's `preinitialize` event, instead of the `initialize` or `creationComplete` event. By setting the styles as early as possible during initialization, you avoid unnecessary style notification and lookup.

If you programmatically create a component and want to set styles on that component, call the `setStyle()` method before you attach it to the display list with a call to the `addChild()` method, as the following example shows:

```
<?xml version="1.0"?>
<!-- optimize/CreateStyledButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp(event)">
    <mx:Script><![CDATA[
        import mx.controls.Button;

        public function initApp(e:Event):void {
            var b:Button = new Button();
            b.label="Click Me";
            b.setStyle("color", 0x00CCFF);
            panel1.addChild(b);
        }
    ]]></mx:Script>

    <mx:Panel id="panel1"/>

</mx:Application>
```

Setting global styles

Changing global styles (changing a CSS ruleset that is associated with a class or type selector) at run time is an expensive operation. Any time you change a global style, Flash Player must perform the following actions:

- Traverse the entire application looking for instances of that control.
- Check all the control's children if the style is inheriting.
- Redraw that control.

The following example globally changes the Button control's color style property:

```
<?xml version="1.0"?>
<!-- optimize/ApplyGlobalStyles.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp(event)">
    <mx:Script><![CDATA[
        public function initApp(e:Event):void {
            StyleManager.getStyleDeclaration("Button").setStyle("color",
0x00CCFF);
        }
    ]]></mx:Script>

    <mx:Panel id="panel1">
        <mx:Button id="b1" label="Click Me"/>
        <mx:Button id="b2" label="Click Me"/>
        <mx:Button id="b3" label="Click Me"/>
    </mx:Panel>

</mx:Application>
```

If possible, set global styles at authoring time by using CSS. If you must set them at run time, try to set styles by using the techniques described in "Reducing calls to the setStyle() method" on page 110.

Calling the setStyleDeclaration() and loadStyleDeclarations() methods

The `setStyleDeclaration()` method is computationally expensive. You can prevent Flash Player from applying or clearing the new styles immediately by setting the `update` parameter to `false`.

The following example sets new class selectors on different targets, but does not trigger the update until the last style declaration is applied:

```
<?xml version="1.0"?>
<!-- styles/SetStyleDeclarationExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp()">
  <mx:Script><![CDATA[
        import mx.styles.StyleManager;

        private var myButtonStyle:CSSStyleDeclaration = new
CSSStyleDeclaration('myButtonStyle');
        private var myLabelStyle:CSSStyleDeclaration = new
CSSStyleDeclaration('myLabelStyle');
        private var myTextAreaStyle:CSSStyleDeclaration = new
CSSStyleDeclaration('myTextAreaStyle');

        private function initApp():void {
            myButtonStyle.setStyle('color', 'blue');
            myLabelStyle.setStyle('color', 'blue');
            myTextAreaStyle.setStyle('color', 'blue');
        }

        private function applyStyles():void {
            StyleManager.setStyleDeclaration("Button", myButtonStyle,
false);
            StyleManager.setStyleDeclaration("Label", myLabelStyle, false);
            StyleManager.setStyleDeclaration("TextArea", myTextAreaStyle,
true);
        }
  ]]></mx:Script>

    <mx:Button id="myButton" label="Click Me" click="applyStyles()"/>
    <mx:Label id="myLabel" text="This is a label"/>
    <mx:TextArea id="myTextArea" text="This is a TextArea"/>

</mx:Application>
```

When you pass `false` for the `update` parameter, Flash Player stores the selector but does not apply the style. When you pass `true` for the `update` parameter, Flash Player recomputes the styles for every visual component in the application.

The `loadStyleDeclarations()` method is similarly computationally expensive. When you load a new style sheet, this method triggers an update to the display list by default. You can prevent Flash Player from applying or clearing the new style sheets immediately by setting the `update` parameter to `false`. When you chain calls to `loadStyleDeclarations()` methods, set the update parameter to `false` for all calls except the last one.

Working with containers

Containers provide a hierarchical structure that lets you control the layout characteristics of container children. You can use containers to control child sizing and positioning, or to control navigation among multiple child containers.

When you develop your Flex application, try to minimize the number of containers that you use. This is because most containers provide relative sizing and positioning, which can be resource-intensive operations, especially when an application first starts.

One common mistake is to create a container that contains a single child. Sometimes having a single child in a container is necessary, such as when you use the container's padding to position the child. But try to identify and remove containers such as these that provide no real functionality. Also keep in mind that the root of an MXML component does not need to be a container.

Another sign of possibly too many containers is when you have a container nested inside another container, where both the parent and child containers have the same type (for example, HBoxes).

Minimizing container nesting

It is good practice to avoid deeply nested layouts when possible. For simple applications, if you have nested containers more than three levels deep, you can probably produce the same layout with fewer levels of containers. Deep nesting can lead to performance problems. For larger applications, deeper nesting might be unavoidable.

When you nest containers, each container instance runs measuring and sizing algorithms on its children (some of which are containers themselves, so this measuring procedure can be recursive). When the layout algorithms have processed, and the relative layout values have been calculated, Flash Player draws the complex collection of objects comprising the view. By eliminating unnecessary work at object creation time, you can improve the performance of your application.

Using Grid containers

A Grid container is useful for aligning multiple objects. When you use Grid containers, however, you introduce additional levels of containers with the GridItem and GridRow controls. In many cases, you can achieve the same results by using the VBox and HBox containers, and these containers use fewer levels of nesting.

Using layout containers

You can sometimes improve application start-up time by using Canvas containers, which perform absolute positioning, instead of relative layout containers, such as the Form, HBox, VBox, Grid, and Tile containers.

Canvas containers are the only containers that let you specify the location of their child controls by default. All other containers are relative containers by default, which means that they lay everything out relative to other components in the container. You can make Application and Panel containers do absolute positioning.

Canvas containers eliminate the layout logic that other containers use to perform automatic positioning of their children at startup, and replace it with explicit pixel-based positioning. When you use a Canvas container, you must remember to set the *x* and *y* positions of all of its children. If you do not set the *x* and *y* positions, the Canvas container's children lay out on top of each other at the default *x*, *y* coordinates (0,0).

The canvas container is not always more efficient than other containers, however, because it must measure itself to make sure that it is large enough to contain its children. Applications that use canvases typically contain a much flatter containment hierarchy. As a result, using canvas containers can lead to less nesting and fewer overall containers, which improves performance.

Canvas containers support constraints, which means that if the container changes size, the children inside the container move with it.

Using absolute sizing

Writing object widths and heights into the code can save time because the Flex layout containers do not have to calculate the size of the object at run time. By specifying container or control widths or heights, you lighten the relative layout container's processing load and subsequently decrease the creation time for the container or control. This technique works with any container or control.

Improving effect performance

Effects let you add animation and motion to your application in response to user or programmatic action. For example, you can use effects to cause a dialog box to bounce slightly when it receives focus, or to slowly fade in when it becomes visible.

Effects can be one of the most processor-intensive tasks performed by a Flex application. This section describes some techniques for improving the performance of effects.

For more information on the topics introduced here, see Chapter 17, "Using Behaviors," in *Flex 2 Developer's Guide*.

Increasing effect duration

Increase the duration of your effect with the `duration` property. Doing this spreads the distinct, choppy stages over a longer period of time, which lets the human eye fill in the difference for a smoother effect.

Hiding parts of the target view

Make parts of the target view invisible when the effect starts, play the effect, and then make those parts visible when the effect has completed. To do this, you add logic in the `effectStart` and `effectEnd` event handlers that controls what is visible before and after the effect.

When you apply a Resize effect to a Panel container, for example, the measurement and layout algorithm for the effect executes repeatedly over the duration of the effect. When a Panel container has many children, the animation can be jerky because Flex cannot update the screen quickly enough. Also, resizing one Panel container often causes other Panel containers in the same view to resize.

To solve this problem, you can use the Resize effect's `hideChildrenTargets` property to hide the children of Panel containers while the Resize effect is playing. The value of the `hideChildrenTargets` property is an Array of Panel containers that should include the Panel containers that resize during the animation. When the `hideChildrenTargets` property is `true`, and before the Resize effect plays, Flex iterates through the Array and hides the children of each of the specified Panel containers.

Avoiding bitmap-based backgrounds

Designers often give their views background images that are solid colors with gradients, slight patterns, and so forth. To ease what Flash Player redraws during an effect, try using a solid background color for your background image. Or, if you want a slight gradient instead of a solid color, use a background image that is a SWF or SVG file. These are easier for Flash Player to redraw than standard JPG or PNG files.

Suspending background processing

To improve the performance of effects, you can disable background processing in your application for the duration of the effect by setting the `suspendBackgroundProcessing` property of the Effect to `true`. The background processing that is blocked includes component measurement and layout, and responses to data services for the duration of the effect.

Using the cachePolicy property

An effect can use the bitmap caching feature in Flash Player to speed up animations. An effect typically uses bitmap caching when the target component's drawing does not change while the effect is playing.

The `cachePolicy` property of UIComponents controls the caching operation of a component during an effect. The `cachePolicy` property can have the following values:

CachePolicy.ON Specifies that the effect target is always cached.

CachePolicy.OFF Specifies that the effect target is never cached.

CachePolicy.AUTO Specifies that Flex determines whether the effect target should be cached. This is the default value.

The `cachePolicy` property is useful when an object is included in a redraw region but the object does not change. For more information about redraw regions, see "Understanding redraw regions" on page 118.

The `cachePolicy` property provides a wrapper for the `cacheAsBitmap` property. For more information, see "Using the cacheAsBitmap property" on page 118.

Improving rendering speed

The actual rendering of objects on the screen can take a significant amount of time. Improving the rendering times can dramatically improve your application's performance. Use the techniques in this section to help improve rendering speed. In addition, use the techniques described in the previous section, "Improving effect performance" on page 115, to improve effect rendering speed.

Setting movie quality

You can use the `quality` property of the wrapper's `<object>` and `<embed>` tags to change the rendering of your Flex application in Flash Player. Valid values for the `quality` property are `low`, `medium`, `high`, `autolow`, `autohigh`, and `best`. The default value is `best`.

The `low` setting favors playback speed over appearance and never uses anti-aliasing. The `autolow` setting emphasizes speed at first but improves appearance whenever possible. The `autohigh` setting emphasizes playback speed and appearance equally at first, but sacrifices appearance for playback speed if necessary. The `medium` setting applies some anti-aliasing and does not smooth bitmaps. The `high` setting favors appearance over playback speed and always applies anti-aliasing. The `best` setting provides the best display quality and does not consider playback speed. All output is anti-aliased and all bitmaps are smoothed.

For information on these settings, see "About the <object> and <embed> tags" on page 378.

Understanding redraw regions

A *redraw region* is the region around an object that must be redrawn when that object changes. Everything in a redraw region is redrawn during the next rendering phase after an object changes. The area that Flash Player redraws includes the object, and any objects that overlap with the redraw region, such as the background or the object's container.

You can see redraw regions at run time in the debugger version of Flash Player by selecting View > Show Redraw Regions in the player's menu. When you select this option, the debugger version of Flash Player draws red rectangles around each redraw region while the application runs.

By looking at the redraw regions, you can get a sense of what is changing and how much rendering is occurring while your application runs. Flash Player sometimes combines the redraw regions of several objects into a single region that it redraws. As a result, if your objects are spaced close enough together, they might be redrawn as part of one region, which is better than if they are redrawn separately. If the number of regions is too large, Flash Player might redraw the entire screen.

Using the cacheAsBitmap property

To improve rendering speeds, make careful use of the `cacheAsBitmap` property. You can set this property on any UIComponent.

When you set the `cacheAsBitmap` property to `true`, Flash Player stores a copy of the initial bitmap image of an object in memory. If you later need that object, and the object's properties have not changed, Flash Player uses the cached version to redraw the object. This can be faster than using the vectors that make up the object.

Setting the `cacheAsBitmap` property to `true` can be especially useful if you use animations or other effects that move objects on the screen. Instead of redrawing the object in each frame during the animation, Flash Player can use the cached bitmap.

The downside is that changing the properties of objects that are cached as bitmaps is more computationally expensive. Each time you change a property that affects the cached object's appearance, Flash Player must remove the old bitmap and store a new bitmap in the cache. As a result, only set the `cacheAsBitmap` property to `true` for objects that do not change much.

Enable bitmap caching only when you need it, such as during the duration of an animation, and only on a few objects at a time because it can be a memory-intensive operation. The best approach might be to change this property at various times during the object's life cycle, rather than setting it once.

Using filters

To improve rendering speeds, do not overuse of filters such as DropShadowFilter. The expense of the filter is proportional to the number of pixels in the object that you are applying the filter to. As a result, it is best to use filters on smaller objects.

Using device text

Mixing device text and vector graphics can slow rendering speeds. For example, a DataGrid control that contains both text and graphics inside a cell will be much slower to redraw than a DataGrid that contains just text.

Using clip masks

Using the `scrollRect` and `mask` properties of an object are expensive operations. Try to minimize the number of times you use these properties.

Using large data sets

This section describes how to minimize overhead when working with large data sets.

Paging

When you use a DataService class to get your remote data, you might have a collection that does not initially load all of its data on the client. You can prevent large amounts of data from traveling over the network and slowing down your application while that data is processed using paging. The data that you get incrementally is referred to as *paged* data, and the data that has not yet been received is *pending* data.

The paging features of the DataService class include:

- Maximum message size on the destination can be configured.
- If size exceeds the maximum value, multiple message batches are used.
- Client reassembles separate messages.
- Asynchronous data paging across the network.
- User interface elements can display portions of the collection without waiting for the entire collection to load.

For more information, see Chapter 7, "Using Data Providers and Collections," in *Flex 2 Developer's Guide*.

Disabling live scrolling

One issue when using a DataGrid control with large data sets is that it may be slow to scroll when using the scrollbar. When the DataGrid displays newly visible data, it calls the `getItemAt()` method on the data provider.

The default behavior of a DataGrid is to continuously update data when the user is scrolling through it. As a result, performance can degrade if you just simply scroll through the data on a DataGrid because the DataGrid is continuously calling the `getItemAt()` method. This can be a computationally expensive method to call.

You can disable this *live scrolling* so that the view is only updated when the scrolling stops by setting the `liveScrolling` property to false.

The default value of the `liveScrolling` property is true. All subclasses of ScrollControlBase, including TextArea, HorizontalList, TileList, and DataGrid, have this property.

Dynamically repeating components

This section describes the relative benefits of using List-based controls rather than the Repeater control to dynamically repeat components. If you must use the Repeater, however, it also describes some techniques for improving the performance of that control.

Comparing List-based controls to the Repeater control

To dynamically repeat components, you can choose between the Repeater or List-based controls, such as HorizontalList, TileList, or List. To achieve better performance, you can often replace layouts you created with a Repeater with the combination of a HorizontalList or TileList and an item renderer.

The Repeater object is useful for repeating a small set of simple user interface components, such as RadioButton controls and other controls typically used in Form containers. You can use the HorizontalList, TileList, or List control when you display more than a few repeated objects.

The HorizontalList control displays data horizontally, similar to the HBox container. The HorizontalList control always displays items from left to right. The TileList control displays data in a tile layout, similar to the Tile container. The TileList control provides a direction property that determines if the next item is down or to the right. The List control displays data in a single vertical column.

Unlike the Repeater object, which instantiates all objects that are repeated, the HorizontalList, TileList, and List controls only instantiate what is visible in the list. The Repeater control takes a data provider (typically an Array) that creates a new copy of its children for each entry in the Array. If you put the Repeater control's children inside a container that does not use deferred instantiation, your Repeater control might create many objects that are not initially visible.

For example, a VBox container creates all objects within itself when it is first created. In the following example, the Repeater control creates all the objects whether or not they are initially visible:

```
<?xml version="1.0"?>
<!-- optimize/VBoxRepeater.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">

    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        [Bindable]
        public var imgList:ArrayCollection = new ArrayCollection([
            {img:"bird.gif"},
            {img:"bird-gray.gif"},
            {img:"bird-silly.gif"}
        ]);

    ]]></mx:Script>

    <mx:Panel title="VBox with Repeater">
        <mx:VBox height="150" width="250">
            <mx:Repeater id="r" dataProvider="{imgList}">
                <mx:Image source="../assets/{r.currentItem.img}"/>
            </mx:Repeater>
        </mx:VBox>
    </mx:Panel>

</mx:Application>
```

If you use a List-based control, however, Flex only creates those controls in the list that are initially visible. The following example uses the List control to create only the image needed for rendering:

```
<?xml version="1.0"?>
<!-- optimize/ListItems.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
   <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;

        private static var birdList:Array = ["../assets/bird.gif","../
assets/bird-gray.gif","../assets/bird-silly.gif"];
        [Bindable]
        private var birdListAC:ArrayCollection = new
ArrayCollection(birdList);

        private function initCatalog():void {
            birdlist.dataProvider = birdListAC;
        }

    ]]></mx:Script>

    <mx:Panel title="List">
        <mx:List id="birdlist" rowHeight="150" width="250" rowCount="1"
itemRenderer="mx.controls.Image" creationComplete="initCatalog()">
        </mx:List>
    </mx:Panel>
</mx:Application>
```

Using the Repeater control

When using a Repeater control, keep the following techniques in mind:

- Avoid repeating objects that have clip masks because using clip masks is a resource-intensive process.
- Ensure that the containers used as the children of the Repeater control do not have unnecessary container nesting and are as small as possible. If a single instance of the repeated view takes a noticeable amount of time to instantiate, repeating makes it worse. For example, multiple Grid containers in a Repeater object do not perform well because Grid containers themselves are resource-intensive containers to instantiate.
- Set the `recycleChildren` property to `true`. The `recycleChildren` property is a Boolean value that, when set to `true`, binds new data items into existing Repeater children, incrementally creates children if there are more data items, and destroys extra children that are no longer required.

The default value of the `recycleChildren` property is `false` to ensure that you do not leave stale state information in a repeated instance. For example, suppose you use a Repeater object to display photo images and each Image control has an associated NumericStepper control for how many prints you want to order. Some of the state information, such as the image, comes from the `dataProvider` property. Other state information, such as the print count, is set by user interaction. If you set the `recycleChildren` property to `true` and page through the photos by incrementing the Repeater object's `startingIndex` value, the Image controls bind to the new images, but the NumericStepper control maintains the old information. Use `recycleChildren="false"` only if it is too cumbersome to reset the state information manually, or if you are confident that modifying your `dataProvider` property should not trigger a recreation of the Repeater object's children.

Keep in mind that the `recycleChildren` property has no effect on a Repeater object's speed when the Repeater object loads the first time. The `recycleChildren` property improves performance only for subsequent changes to the Repeater control's data provider. If you know that your Repeater object creates children only once, you do not have to use the `recycleChildren` property or worry about the stale state situation.

Improving server-side performance

The Flex web application is deployed as a standard J2EE web application and is supported on a number of popular J2EE application servers.

Flex relies on the Flash Player client for the user interface, and on the server solely for transferring the client application and then sending requested data. When a user requests an application by using a URL, the SWF file is transferred to the client where it begins to execute. Requests for data are typically handled by the server running Flex and are subsequently transferred to the client. This separation of processing lets the client handle simple tasks, such as field validation, data formatting, sorting, and filtering, which frees up valuable server CPU cycles.

This section describes some techniques to use to improve server-side performance, both for when you are using the web-tier compiler and the features of the Flex Data Services server, and when you deploy Flex as a web application on another platform.

Some of the techniques described in this section apply to improving the performance of the Flex Builder and command-line compilers.

Precompiling

MXML compilation is CPU intensive. In most situations, this is not a factor because an MXML page is compiled once by the server the first time it is requested. On subsequent requests, the MXML page is served from a cache.

In general, precompile your MXML pages with the mxmlc command-line compiler or the built-in Flex Builder compiler, and create your own wrapper to load the SWF files into the browser. This option enables you to deploy bytecode, but is somewhat more involved than letting Flex build your HTML wrappers for you on the fly. Some features, such as Flash Player detection and history management are not generated automatically. You must add the code for these features in your application's wrapper.

Using incremental compilation

When you use Flex Data Services, you can use incremental compilation to help decrease the time it takes to compile with the web-tier compiler. When incremental compilation is enabled, the compiler inspects changes to the bytecode between revisions of an application and only recompiles the sections of compiled code that changed.

To enable incremental compilation for the web-tier compiler, you set the value of the `incremental-compile` option in the flex-webtier-config.xml file to `true`. The default value is `true`. The `incremental` option in the flex-config.xml file has no effect when you use the web-tier compiler.

You can use the `recompile` query string parameter to override the `incremental-compile` option, as the following example shows:

```
http://www.mydomain.com/MyApp.mxml?recompile=true
```

This option forces a complete recompile of the application.

Incremental compilation requires more memory than standard compilation. Therefore, you might find it more advantageous to disable incremental compilation when you use the web-tier compiler in a production environment. The Flex Data Services server generally caches the first-compiled SWF file, and returns that file on subsequent requests. Unless your applications change frequently, you only benefit from incremental compilation for the first request.

If production mode is enabled, disable incremental compilation.

For more information on incremental compilation, see "About incremental compilation" on page 214.

Disabling Express Install and player detection

You can disable the Express Install and player detection features if they are not required. Doing so reduces the size of the wrapper and prevents the server from having to transmit Express Install and player detection files. This can reduce network overhead.

To disable Express Install, remove the Express Install code from your wrapper. You can do this in the wrapper generated by Flex Builder by deselecting the Detect Flash Version and Use Express Install options in the Flex Compiler properties dialog box in Flex Builder.

If you are using Flex Data Services, you can disable Express Install and player detection by setting the values of the `use-player-detection` and `use-express-install` options to `false` in the flex-webtier-config.xml file.

For more information about Express Install, see Chapter 17, "Using Express Install," on page 391.

Disabling history management

You can disable the Flex history management feature if it is not required. Disabling this feature prevents the server from transmitting the history management files, such as the history.js, history.htm, and history.swf files, which can reduce network overhead.

To disable history management, remove the history management code from your wrapper. You can do this in Flex Builder by deselecting the Enable History Management option in the Flex Compiler properties dialog box.

If you are using Flex Data Services, you can disable history management by setting the value of the `use-history-management` option to `false` in the flex-webtier-config.xml file.

For more information about history management, see Chapter 32, "Using the History Manager," in *Flex 2 Developer's Guide*.

Tuning JVM heap sizes

Flex Data Services is an application running on a Java application server. Therefore, the performance of your Flex applications can be affected by the underlying JVM of the application server. The most common way to potentially increase the performance and reliability of your JVM is to tune the JVM's heap size. Configuring the JVM can lead you to faster and more efficient compilations.

In addition, the Flex command-line compilers use the Java JRE. You can improve the performance of your command-line compilers by adjusting the JVM's heap size.

The most common JVM configuration is to set the size of the Java heap. The Java heap is the amount of memory reserved for the JVM. The actual size of the heap varies as classes are loaded and unloaded. If the heap requires more memory than the maximum allocated, performance suffers as the JVM performs garbage collection to maintain enough free memory for the applications to run.

For more information, see "Changing the JVM heap size" on page 167.

Caching

The Flex Data Services server returns SWF files based on client requests. The caching mechanism used by Flex Data Services reduces delays by storing compiled SWF files in a content cache and responding to requests with the cached files when possible. In addition, the Flex Data Services server caches resources including components (SWC, MXML, and ActionScript files), CSS style sheets, images, RSLs, and ActionScript files included with the `<mx:Script>` tag.

You can configure settings such as the watcher interval and number of files cached. Increasing the value of the watcher interval, for example, can result in slightly increased performance because the server does not check the cache as often.

For more information, see "Cache settings" on page 173. You can also configure font caching. For more information, see "Caching fonts and glyphs" on page 174.

In addition to caching fonts, you can also restrict the number of characters in a embedded font definition to reduce the overall size of the font definition included in the SWF file. You do this by using the `language-range` compiler option. For more information, see Chapter 19, "Using Fonts," in *Flex 2 Developer's Guide*.

Using headless servers

A *headless server* is one that is running on UNIX or Linux and often does not have a monitor, keyboard, mouse, or even a graphics card. Headless servers are most commonly encountered in ISPs and ISVs, where available space is at a premium and servers are often mounted in racks. Enabling headless mode reduces the graphics requirements of the underlying system and can make for a more efficient use of memory.

To enable headless mode of the Flex application, define the value of the `<headless-server>` tag in the flex-config.xml file. Setting this property to `true` is required to support fonts and SVG images in a nongraphical environment. The `<headless-server>` tag is a child tag of `<compiler>`. The following example sets `<headless-server>` to `true`:

```
<headless-server>true</headless-server>
```

The default value is commented out.

Setting the value of `<headless-server>` to `true` in flex-config.xml sets the system property `java.awt.headless` to `true`.

Enabling production mode

Production mode disables settings such as warnings, debugging, and caching. You enable production mode in the flex-webtier-config.xml file, as the following example shows:

```
<production-mode>true</production-mode>
```

The default value of the `production-mode` option is `false`.

Improving Flex Charting component performance

This section describes some techniques you can use to improve the performance of charting controls. Charting controls are included in Flex Charting components. For more information about charting controls, see Part 7, “Charting Components,” in *Flex 2 Developer’s Guide*.

Avoiding filtering series data

When possible, set the `filterData` property to `false`. In the transformation from data to screen coordinates, the various series types filter the incoming data to remove any missing values that are outside the range of the chart; missing values would render incorrectly if drawn to the screen. For example, a chart that represents vacation time for each week in 2003 might not have a value for the July fourth weekend because the company was closed. If you know your data model will not have any missing values at run time, or values that fall outside the chart’s data range, you can instruct a series to explicitly skip the filtering step by setting its `filterData` property to `false`.

Coding the LinearAxis object

If possible, do not let a LinearAxis object autocalculate its range. A LinearAxis control calculating its numeric range can be a resource-intensive calculation. If you know reasonable minimum and maximum values for the range of your LinearAxis, specify them to help your charts render quicker.

In addition to specifying the range for a LinearAxis, specify an interval (the numeric distance between label values along the axis) value. Otherwise, the chart control must calculate this value.

Coding the CategoryAxis object

Modifying a CategoryAxis object's data provider is more resource intensive than modifying a Series object's data provider. If the data bound to your chart is going to change, but the categories in your chart will stay static, have the CategoryAxis' data provider and Series' data provider refer to different objects. This prevents the CategoryAxis from reevaluating its data provider, which is a resource-intensive computation.

Styling AxisRenderer objects

Improve the rendering time of your AxisRenderers objects by setting particular styles. The AxisRenderers perform many calculations to ensure that they render correctly in all situations. The more help you can give them in restricting their options, the faster they render. Setting the `labelRotation` and `canStagger` styles on the AxisRenderer improve performance. You can set these styles within the tag or in CSS.

Specifying gutter styles

Specify gutter styles when possible. The gutter area of a Cartesian chart is the area between the margins and the actual axis lines. With default values, the chart adjusts the gutter values to accommodate axis decorations. Calculating these gutter values can be resource intensive. By explicitly setting the values of the `gutterLeft`, `gutterRight`, `gutterTop`, and `gutterBottom` style properties, your charts draw quicker and more efficiently.

Using drop shadows

To improve performance, do not use drop-shadows on your series items unless they are necessary. You can selectively add shadows to individual chart series by using renderers such as the ShadowBoxItemRenderer and ShadowLineRenderer classes.

Shadows are implemented as filters in charting controls. As a result, you must remove these shadows by setting the chart control's `seriesFilters` property to an empty Array. The following example removes the shadows from all series, but then changes the renderer for the third series to be a shadow renderer:

```
<?xml version="1.0"?>
<!-- optimize/RemoveShadowsColumnChart.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        [Bindable]
        public var expenses:ArrayCollection = new ArrayCollection([
            {Month: "Jan", Income: 2000, Expenses: 1500, Profit: 500},
            {Month: "Feb", Income: 1000, Expenses: 200, Profit: 800},
            {Month: "Mar", Income: 1500, Expenses: 500, Profit: 1000}
        ]);
    ]]></mx:Script>
    <mx:Panel title="Column Chart">
        <mx:ColumnChart id="myChart" dataProvider="{expenses}">
            <mx:seriesFilters>
                <mx:Array/>
            </mx:seriesFilters>
            <mx:horizontalAxis>
                <mx:CategoryAxis dataProvider="{expenses}"
categoryField="Month"/>
            </mx:horizontalAxis>
            <mx:series>
                <mx:ColumnSeries xField="Month" yField="Income"
displayName="Income"/>
                <mx:ColumnSeries xField="Month" yField="Expenses"
displayName="Expenses"/>
                <mx:ColumnSeries xField="Month" yField="Profit"
displayName="Profit"
itemRenderer="mx.charts.renderers.ShadowBoxItemRenderer"/>
            </mx:series>
        </mx:ColumnChart>
        <mx:Legend dataProvider="{myChart}"/>
    </mx:Panel>
</mx:Application>
```

CHAPTER 6

Improving Startup Performance

6

Adobe Flex helps you improve the actual and perceived startup times of your applications. You can do this by deferring the creation of certain controls until a later time, or customize the order in which containers are created and displayed in your applications. This topic describes these techniques, which can improve the startup performance of your Flex applications.

Contents

About startup performance 131
About startup order 132
Using deferred creation. 134
Creating deferred components 138
Using ordered creation 143
Using the callLater() method 150

About startup performance

You could increase the startup time and decrease performance of your applications if you create too many objects or put too many objects into a single view. To improve startup time, minimize the number of objects that are created when the application is first loaded. If a user-interface component is not initially visible at startup, avoid creating that component until you need it. This is called deferred creation. Containers that have multiple views, such as an Accordion container, provide built-in support for this behavior. You can use ActionScript to customize the creation order of multiple-view containers or defer the creation of other containers and controls.

After you improve the *actual* startup time of your application as much as possible, you can improve *perceived* startup time by ordering the creation of containers in the initial view. The default behavior of Flex is to create all containers and their children in the initial view, and then display everything at one time. The user will not be able to interact with the application or see meaningful data until all the containers and their children are created. In some cases, you can improve the user's initial experience by displaying the components in one container before creating the components in the next container. This process is called ordered creation.

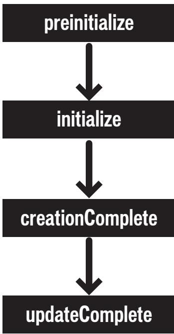
The remaining sections of this topic describe how to use deferred creation to reduce overall application startup time and ordered creation to make the initial startup time appear as short as possible to the user. But before you can fully understand ordered creation and deferred creation, you must also understand the differences between single-view and multiple-view containers, the order of events in a component's startup life cycle, and how to manually instantiate controls from their child descriptors.

About startup order

All Flex components trigger a number of events during their startup procedure. These events indicate when the component is first created, plotted internally, and drawn on the screen. The events also indicate when the component is finished being created and, in the case of containers, when its children are created.

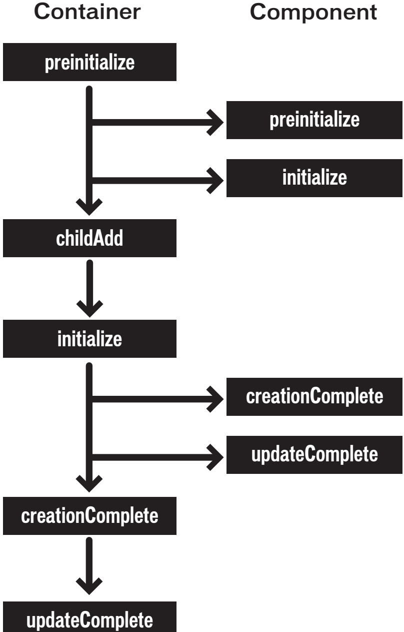
Components are instantiated, added or linked to a parent, and then sized and laid out inside their container. The component creation order is as follows:

The following example shows the major events that are dispatched during a component's creation life cycle:



The creation order is different for containers and components because containers can be the parent of other components or containers. Components within the containers must also go through the creation order. If a container is the parent of another container, the inner container's children must also go through the creation order.

The following example shows the major events that are dispatched during a container's creation life cycle:



After all components are created and drawn, the Application object dispatches an `applicationComplete` event. This is the last event dispatched during an application startup.

The creation order of multiview containers (navigators) is different from standard containers. By default, all top-level views of the navigator are instantiated. However, Flex creates only the children of the initially visible view. When the user navigates to the other views of the navigator, Flex creates those views' children. For more information on the deferred creation of children of multiview containers, see "Using deferred creation" on page 134.

For a detailed description of the component creation life cycle, see Chapter 10, "Creating Advanced Visual Components in ActionScript," in *Creating and Extending Flex 2 Components*.

Using deferred creation

By default, containers create only the controls that initially appear to the user. Flex creates the container's other descendants if the user navigates to them. Containers with a single view, such as Box, Form, and Grid containers, create all of their descendants during the container's instantiation because these containers display all of their descendants immediately.

Containers with multiple views, called navigator containers, only create and display the descendants that are visible at any given time. These containers are the ViewStack, Accordion, and TabNavigator containers.

When navigator containers are created, they do not immediately create all of their descendants, but only those descendants that are initially visible. Flex defers the creation of descendants that are not initially visible until the user navigates to a view that contains them.

The result of this deferred creation is that an MXML application with navigator containers loads more quickly, but the user experiences brief pauses when he or she moves from one view to another when interacting with the application.

You can instruct each container to create their children or defer the creation of their children at application startup by using the container's `creationPolicy` property. This can improve the user experience after the application loads. For more information, see "About the creationPolicy property" on page 134.

You can also create individual components whose instantiation is deferred by using the `createComponentsFromDescriptors()` method. For more information, see "Creating deferred components" on page 138.

About the creationPolicy property

To defer the creation of any component, container, or child of a container, you use the `creationPolicy` property. Every container has a `creationPolicy` property that determines how the container decides whether to create its descendants when the container is created. You can change the policy of a container using MXML or ActionScript.

The valid values for the `creationPolicy` property are `auto`, `all`, `none`, and `queued`. The meaning of these settings depends on whether the container is a navigator container (multiple-view container) or a single-view container. For information on the meaning of these values see "Single-view containers" on page 135 and "Multiple-view containers" on page 136.

The creationPolicy property is not inheritable. This means that if you set the value of the creationPolicy property to none on an outer container, all containers within that container have the default value of the creationPolicy property, unless otherwise set. They do not inherit the value of none for their creationPolicy. Also, if you have two containers at the same level (of the same type) and you set the creationPolicy of one of them, the other container has the default value of the creationPolicy property unless you explicitly set it.

Single-view containers

Single-view containers by default create all their children when the application first starts. You can use the creationPolicy property to change this behavior. The following table describes the values of the creationPolicy property when you use it with single-view containers:

Value	Description
all, auto	Creates all controls inside the single-view container. The default value is auto, but all results in the same behavior.
none	Instructs Flex to not instantiate any component within the container until you manually instantiate the controls. When the value of the creationPolicy property is none, explicitly set a width and height for that container. Normally, Flex scales the container to fit the children that are inside it, but because no children are created, proper scaling is not possible. If you do not explicitly resize the container, it grows to accommodate the children when they are created. To manually instantiate controls, you use the createComponentsFromDescriptors() method. For more information, see "Creating deferred components" on page 138.
queued	Has no effect on deferred creation. For more information on using the queued value of the creationPolicy property, see "Using ordered creation" on page 143.

The following example sets the value of a VBox container's creationPolicy property to auto, the default value:

```
<?xml version="1.0"?>
<!-- layoutperformance/AutoCreationPolicy.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:VBox id="myVBox" creationPolicy="auto">
        <mx:Button id="b1" label="Get Weather"/>
    </mx:VBox>
</mx:Application>
```

The default behavior of all single-view containers is that they and their children are entirely instantiated when the application starts. If you set the `creationPolicy` property to `none`, however, you can selectively instantiate controls within the containers by using the techniques described in "Creating deferred components" on page 138.

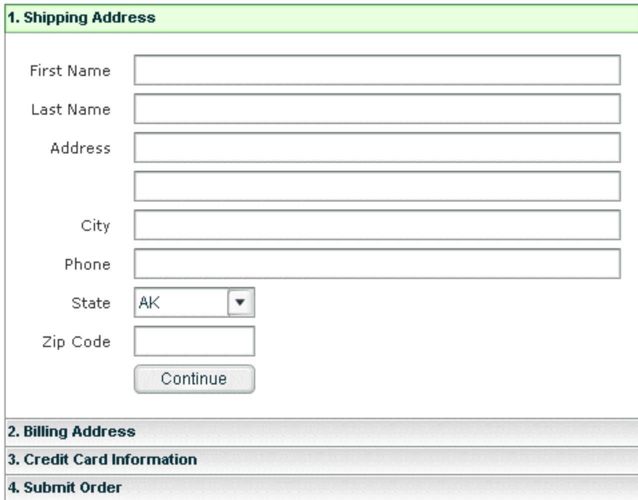
Multiple-view containers

Containers with multiple views, such as the ViewStack and Accordion, do not immediately create all of their descendants, but only those descendants that are visible in the initial view. Flex defers the instantiation of descendants that are not initially visible until the user navigates to a view that contains them. The following containers have multiple views and, so, are defined as navigator containers:

- ViewStack
- TabNavigator
- Accordion

When you instantiate a navigator container, Flex creates all of the top-level children. For example, creating an Accordion container triggers the creation of each of its views, but not the controls within those views. The `creationPolicy` property determines the creation of the child controls inside each view.

When you set the `creationPolicy` property to `auto` (the default value), navigator containers instantiate only the controls and their children that appear in the initial view. The first view of the Accordion container is the initial pane, as the following example shows:



When the user navigates to another panel in the Accordion container, the navigator container creates the next set of controls, and recursively creates the new view's controls and their descendants. You can use the Accordion container's `creationPolicy` property to modify this behavior. The following table describes the values of the `creationPolicy` property when you use it with navigator containers:

Value	Description
`all`	Creates all controls in all views of the navigator container. This setting causes a delay in application startup time, but results in quicker response time for user navigation.
`auto`	Creates all controls only in the initial view of the navigator container. This setting causes a faster startup time for the application, but results in slower response time for user navigation. This setting is the default for multiple-view containers.
`none`	Instructs Flex to not instantiate any component within the navigator container or any of the navigator container's panels until you manually instantiate the controls. To manually instantiate controls, you use the `createComponentsFromDescriptors()` method. For more information, see "Creating deferred components" on page 138.
`queued`	This property has no effect on deferred creation. For more information on using the `queued` value of the `creationPolicy` property, see "Using ordered creation" on page 143.

The following example sets the `creationPolicy` property of an Accordion container to `all`, which instructs the container to instantiate all controls for every panel in the navigator container when the application starts:

```
<?xml version="1.0"?>
<!-- layoutperformance/AllCreationPolicy.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Panel title="Accordion">
        <mx:Accordion id="myAccordion" creationPolicy="all">
            <mx:VBox label="Accordion Button for Panel 1">
                <mx:Label text="Accordion container panel 1"/>
                <mx:Button label="Click Me"/>
            </mx:VBox>
            <mx:VBox label="Accordion Button for Panel 2">
                <mx:Label text="Accordion container panel 2"/>
                <mx:Button label="Click Me"/>
            </mx:VBox>
            <mx:VBox label="Accordion Button for Panel 3">
                <mx:Label text="Accordion container panel 3"/>
                <mx:Button label="Click Me"/>
            </mx:VBox>
        </mx:Accordion>
    </mx:Panel>
</mx:Application>
```

Creating deferred components

When you set a container's `creationPolicy` property to `none`, components declared as MXML tags inside that container are not created. Instead, objects that describe those components are added to an Array. These objects are called descriptors. You can use the `createComponentsFromDescriptors()` method to manually instantiate those components. This method is defined on the Container base class.

Using the createComponentsFromDescriptors() method

You use the `createComponentsFromDescriptors()` method of a container to create all the children of a container at one time.

The `createComponentsFromDescriptors()` method has the following signature:

```
container.createComponentsFromDescriptors(recurse:Boolean):Boolean
```

The *`recurse`* argument determines whether Flex should recursively instantiate children of the components. Set the parameter to `true` to instantiate children of the components, or `false` to not instantiate the children. The default value is `false`.

On a single-view container, calling the `createComponentsFromDescriptors()` method instantiates all controls in that container, regardless of the value of the `creationPolicy` property.

In navigator containers, if you set the `creationPolicy` property to `all`, you do not have to call the `createComponentsFromDescriptors()` method, because the container creates all controls in all views of the container. If you set the `creationPolicy` property to `none` or `auto`, calling the `createComponentsFromDescriptors()` method creates only the current view's controls and their descendents.

Another common usage is to set the navigator container's `creationPolicy` property to `auto`. You can then call *`navigator`*`.getChildAt(n).createComponentsFromDescriptors()` to explicitly create the children of the *n*-th view.

The following example does not instantiate any of the buttons in the HBox container when the application starts up, but does when the user changes the value of the `creationPolicy` property. The user initiates this change by selecting *all* from the drop-down list.

```
<?xml version="1.0"?>
<!-- layoutPerformance/ChangePolicy.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="appInit()">
    <mx:Script><![CDATA[
        [Bindable]
        public var p:String;

        private function appInit():void {
            p = policy.selectedItem.toString();
        }

        private function changePolicy():void {
            var polType:String = policy.value.toString();
            hb.creationPolicy = polType;
            if (polType == "none") {
                // do nothing
            } else if (polType == "all") {
                hb.createComponentsFromDescriptors();
            }
        }
    ]]></mx:Script>
    <mx:ComboBox id="policy"
close="p=String(policy.selectedItem);changePolicy();">
        <mx:dataProvider>
            <mx:Array>
                <mx:String>none</mx:String>
                <mx:String>all</mx:String>
            </mx:Array>
        </mx:dataProvider>
    </mx:ComboBox>

    <mx:Panel title="Creation Policy" id="hb" creationPolicy="none">
        <mx:Button label="1" width="50" y="0" x="0"/>
        <mx:Button label="2" width="50" y="0" x="75"/>
        <mx:Button label="3" width="50" y="0" x="150"/>
    </mx:Panel>

    <mx:Label text="{p}"/>

</mx:Application>
```

Using the childDescriptors property

When a Flex application starts, Flex creates an object of type Object that describes each MXML component. These objects contain information about the component's name, type, and properties set in the object's MXML tag. Flex adds these objects to an Array that each container maintains. For example, applications with two Canvas containers have an Array with objects that describe the Canvas containers. Those containers, in turn, have an Array with objects that describe their children.

Each object in the Array is an object of type ComponentDescriptor. You can access this Array by using a container's `childDescriptors` property, and use a zero-indexed value to identify the descriptor. All containers have a `childDescriptors` property.

Depending on the value of the `creationPolicy` property, Flex immediately begins instantiating controls inside the containers or it defers their instantiation. If instantiation is deferred, you can use the properties of this Array to access the ComponentDescriptor of each component and create that object at a specified time.

The `childDescriptors` property points to an Array of objects, so you can use Array functions, such as `length`, to iterate over the children, as the following example shows:

```
<?xml version="1.0"?>
<!-- layoutperformance/AccessChildDescriptors.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import mx.core.ComponentDescriptor;
        import flash.utils.*;

        public function iterateOverChildren():void {
            // Get the number of descriptors.
            var n:int = tile.childDescriptors.length;
            for (var i:int = 0; i < n; i++) {
                var c:ComponentDescriptor = tile.childDescriptors[i];
                var d:Object = c.properties;

                // Trace ids and types of objects in the Array.
                trace(c.id + " is of type " + c.type);

               // Trace the properties added in the MXML tag of the object.
                for (var p:String in d) {
                    trace("Property: " + p + " : " + d[p]);
                }
            }
        }

    ]]></mx:Script>

    <mx:Tile id="tile" creationComplete="iterateOverChildren();">
        <mx:TextInput id="myInput" text="Enter text here"/>
        <mx:Button id="myButton" label="OK" width="150"/>
    </mx:Tile>

</mx:Application>
```

Properties of the ComponentDescriptor include `id`, `type`, and `properties`. The `properties` property points to an object that contains the properties that were explicitly added in the MXML tag. This object does not store properties such as styles and events.

Destroying components

After you create a component, it continues to exist until the user quits the application or you detach it from its parent and the garbage collector destroys it.

To detach a component from its parent container, you can use the `removeChild()` or `removeChildAt()` methods. You can also use the `removeAllChildren()` method to remove all child controls from a container. Calling these methods does not immediately delete the objects from memory. If you do not have any other references to the child, Flash Player garbage collectsw it at some future point. But if you have stored a reference to that child on some other object, the child is not removed from memory.

For more information on using these methods, see the View class in the *Adobe Flex 2 Language Reference*.

Using ordered creation

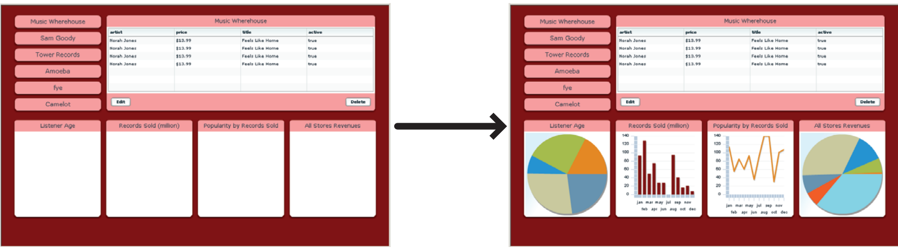
By default, Flex displays the Loading progress bar while it initializes an application. Only after all of the components in the application are initialized and laid out does Flex display any portion of the application. Using ordered creation, you can customize this experience and change the user's perception of how quickly the application starts.

During initialization, Flex first creates all the containers, and then fills in each container with its children and data. Finally, Flex displays the application in its entirety. This causes the user to wait for the entire application to load before beginning to interact with it.

However, you can instruct Flex to display the children of each container as its children are created rather than waiting for the entire application to finish loading. You do this using a technique called ordered creation.

The following example shows a complex application that contains a single container at the top, and four containers across the bottom. This application implements ordered creation so that the contents of each container become visible before the entire application is finished loading.

In this example, the image on the left shows the application after the first container is populated with its children and data, but before the remaining four containers are populated. The image on the right shows the final state of the application.



If this application did not use ordered creation, Flex would not display anything until all components in all five of the containers were created, resulting in a longer perceived start-up time.

To gradually display children of each container, you add them to an instantiation queue. The "Adding containers to the queue" on page 144 section describes how to add containers to the queue so that you can improve perceived layout performance of your Flex applications.

Adding containers to the queue

You can add any number of containers to the instantiation queue so that their contents are displayed in queue order. To add a container to the instantiation queue, you set the value of the container's `creationPolicy` property to `queued`. Unless you specify a queue order, containers are instantiated and displayed in the order in which they appear in the application source code.

In the following example, the Panel containers and their child controls are added to the instantiation queue. To the user, each Panel container appears one at a time during the application startup. The perceived startup time of the application is greatly reduced. The panels are created in the order that they appear in the source code (panel1, panel2, panel3):

```
<?xml version="1.0"?>
<!-- layoutperformance/QueuedCreationPolicy.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Panel id="panel1" creationPolicy="queued" width="100%"
height="33%">
        <mx:Button id="button1a"/>
        <mx:Button id="button1b"/>
    </mx:Panel>

    <mx:Panel id="panel2" creationPolicy="queued" width="100%"
height="33%">
        <mx:Button id="button2a"/>
        <mx:Button id="button2b"/>
    </mx:Panel>

    <mx:Panel id="panel3" creationPolicy="queued" width="100%"
height="33%">
        <mx:Button id="button3a"/>
        <mx:Button id="button3b"/>
    </mx:Panel>
</mx:Application>
```

Flex first creates all the panels. Flex then instantiates and displays all the children in the first panel in the queue before moving on to instantiate the children in the next panel in the queue.

To specify an order for the containers in the instantiation queue, you use the `creationIndex` property. For more information on using the `creationIndex` property, see "Setting queue order" on page 145.

Setting queue order

When you use ordered creation, the order in which containers appear in the instantiation queue determines in what order Flex displays the container and its children on the screen.

You can set the order in which containers are queued by using the container's `creationIndex` property, in conjunction with setting the `creationPolicy` property to `queued`. Flex creates the containers in their `creationIndex` order until all containers in the initial view are created. Flex then revisits each container and creates its children in the same order. After creating all the children in a container that is in the queue, Flex displays that container before starting to create the children in the next container.

All containers support a `creationIndex` property.

If you set a `creationIndex` property on a container, but do not set the `creationPolicy` property to `queued`, Flex ignores the `creationIndex` property and uses `auto`, the default value, for the `creationPolicy` property.

The following example sets the `creationIndex` property so that the contents of the panel3 container are shown first, then the contents of panel2, and finally the contents of panel1:

```
<?xml version="1.0"?>
<!-- layoutperformance/QueueOrder.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        private function logCreationOrder(e:Event):void {
            trace(e.currentTarget.id + " created");
        }
    ]]></mx:Script>

    <mx:Fade id="SlowFade" duration="4000"/>

   <mx:Panel id="panel1" title="Panel 1 (index:2)" creationPolicy="queued"
creationIndex="2"  creationComplete="logCreationOrder(event)"
creationCompleteEffect="{SlowFade}">
        <mx:Button id="button1a" creationComplete="logCreationOrder(event)"/
>
        <mx:Button id="button1b" creationComplete="logCreationOrder(event)"/
>
    </mx:Panel>

   <mx:Panel id="panel2" title="Panel 2 (index:1)" creationPolicy="queued"
creationIndex="1"  creationComplete="logCreationOrder(event)"
creationCompleteEffect="{SlowFade}">
        <mx:Button id="button2a" creationComplete="logCreationOrder(event)"/
>
        <mx:Button id="button2b" creationComplete="logCreationOrder(event)"/
>
    </mx:Panel>

   <mx:Panel id="panel3" title="Panel 3 (index:0)" creationPolicy="queued"
creationIndex="0" creationComplete="logCreationOrder(event)"
creationCompleteEffect="{SlowFade}">
        <mx:Button id="button3a" creationComplete="logCreationOrder(event)"/
>
        <mx:Button id="button3b" creationComplete="logCreationOrder(event)"/
>
    </mx:Panel>
</mx:Application>
```

This example uses the inherited `creationCompleteEffect` effect to play an effect just as the container finishes creation. The result is that the panels fade in slow enough that you can see the order of creation. After a container is initialized and displayed, Flex waits for all effects to finish playing before initializing the next container.

If you set the same value of the `creationIndex` property for two queued containers, Flex creates and displays their contents in the order in which they are defined in the application's source code. The value of the `creationIndex` property can be any valid Number, including 0 and negative values.

Dynamically adding containers to the queue

You can dynamically add containers to the instantiation queue by using the Application class's `addToCreationQueue()` method. The `addToCreationQueue()` method has the following signature:

```
function addToCreationQueue(id:Object, preferredIndex:int,
  callbackFunction:Function, parent:IFlexDisplayObject):void
```

The following table describes the `addToCreationQueue()` method's arguments:

Argument	Description
`id`	Specifies the name of the container that you want to add to the queue.
`preferredIndex`	(Optional) Specifies the container's position in the queue. By default, Flex places the container at the end of the queue, but this value lets you explicitly choose the queue order for the container.p
`callbackFunction`	This parameter is currently ignored.
`parent`	This paremeter is currently ignored.

Only use the `addToCreationQueue()` method to add containers to the queue whose `creationPolicy` property is set to `none`. Containers whose `creationPolicy` property is set to `auto` or `all` will probably created their children during the application initialization. Containers whose `creationPolicy` property is set to `queued` are already in the queue.

After it is added to the queue, the container that is to be created with the `addToCreationQueue()` method then competes with containers whose `creationPolicy` is set to `queued`, depending on their position in the queue. Flex creates containers whose `preferredIndex` property is lowest.

The following example uses the `addToCreationQueue()` method to create the children of the HBox containers when the user clicks the Create Later button. The calls to the `addToCreationQueue()` method specify a `preferredIndex` for each box, which causes the boxes and their children to be created in a different order from the order in which they are defined in the application's source code (in this example, box1, box3, box2, and then box4).

```
<?xml version="1.0"?>
<!-- layoutPerformance/AddToCreationQueueExample.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        public function doCreate():void {
             addToCreationQueue('box1', 0);
             addToCreationQueue('box2', 2);
             addToCreationQueue('box3', 1);
             addToCreationQueue('box4', 3);
        }
    ]]></mx:Script>

    <mx:HBox backgroundColor="#CCCCFF" horizontalAlign="center">
        <mx:Label text="addToCreationQueue()" fontWeight="bold"/>
    </mx:HBox>

    <mx:Form>
        <mx:FormItem label="first:">
           <mx:HBox id="box1" creationPolicy="none" width="200" height="50"
                    borderStyle="solid" backgroundColor="#CCCCFF">
                <mx:Button label="Button 1"/>
                <mx:Button label="Button 2"/>
            </mx:HBox>
        </mx:FormItem>
        <mx:FormItem label="fourth:">
            <mx:HBox id="box2" creationPolicy="none" width="200" height="50"
                    borderStyle="solid" backgroundColor="#CCCCFF">
                <mx:Button label="Button 3"/>
                <mx:Button label="Button 4"/>
            </mx:HBox>
        </mx:FormItem>
        <mx:FormItem label="second:">
            <mx:HBox id="box3" creationPolicy="none" width="200" height="50"
                    borderStyle="solid" backgroundColor="#CCCCFF">
                <mx:Button label="Button 5"/>
                <mx:Button label="Button 6"/>
            </mx:HBox>
        </mx:FormItem>
        <mx:FormItem label="third:">
            <mx:HBox id="box4" creationPolicy="none" width="200" height="50"
                    borderStyle="solid" backgroundColor="#CCCCFF">
                <mx:Button label="Button 7"/>
                <mx:Button label="Button 8"/>
```

```
            </mx:HBox>
        </mx:FormItem>
    </mx:Form>
    <mx:Button label="Create Later" click="doCreate();"/>
</mx:Application>
```

In some cases, the `addToCreationQueue()` method does not act as you might expect. The reason is that if the instantiation queue is empty, the first container to be put in that queue triggers the creation process of its children. Other containers might subsequently be added to the queue, but the instantiation of the first container added has already been triggered, regardless of the value of its `preferredIndex` property. The result is that an item might not have the lowest `preferredIndex` value, but because no other containers are in the queue, Flex begins creating that container. Flex cannot stop instantiating the first container once it starts.

In the following example, although the redBox container has the highest `preferredIndex`, Flex creates its children first because the queue was empty when Flex encountered this line in the code. By the time redBox is complete, the other containers will be in the queue, and Flex proceeds with the next lowest item in the queue; in this case, the whiteBox container, followed by blueBox and, finally, greenBox.

```
function doCreate():void {
   addToCreationQueue('redBox', 4);
   addToCreationQueue('blueBox', 2);
   addToCreationQueue('whiteBox', 1);
   addToCreationQueue('greenBox', 3);
}
```

Combining containers with different creationPolicy settings

You can mix containers with different `creationPolicy` settings and change the order in which they are created and their children are displayed. Flex creates outer containers before inner containers, regardless of their `creationIndex`. This is because Flex does not create the children of a queued container until all containers at that level are created.

Setting a container's `creationPolicy` property does not override the policies of the containers within that container. For example, if you queue an outer container, but set the inner container's `creationPolicy` to `none`, Flex creates the inner container, but not any child controls of that inner container.

In the following example, the button1 control is never created because its container specifies a `creationPolicy` of `none`, even though the outer container sets the `creationPolicy` property to `all`:

```
<?xml version="1.0"?>
<!-- layoutperformance/TwoCreationPolicies.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:HBox label="HBox1" creationPolicy="all" creationIndex="0">
        <mx:HBox label="HBox1" creationPolicy="none">
            <mx:Button id="button1" label="Click Me"/>
        </mx:HBox>
    </mx:HBox>
</mx:Application>
```

Using the callLater() method

The `callLater()` method queues an operation to be performed for the next screen refresh, rather than in the current update. Without the `callLater()` method, you might try to access a property of a component that is not yet available. The `callLater()` method is most commonly used with the `creationComplete` event to ensure that a component has finished being created before Flex proceeds with a specified method call on that component.

All objects that inherit from the UIComponent class can open the `callLater()` method. It has the following signature:

```
callLater(method:Function, args:Array):void
```

The *`method`* argument is the function to call on the object. The *`args`* argument is an optional Array of arguments that you can pass to that function.

The following example defers the invokation of the `doSomething()` method, but when it is opened, Flex passes the Event object and the “Product Description” String in an Array to that function:

```
callLater("doSomething", [event, "Product Description"]);
...
function doSomething(event:Event, title:String):void {
   ...
}
```

The following example uses a call to the `callLater()` method to ensure that new data is added to a DataGrid before Flex tries to put focus on the new row. Without the `callLater()` method, Flex might try to focus on a cell that does not exist and throw an error:

```
<?xml version="1.0"?>
<!-- layoutperformance/CallLaterAddItem.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
initialize="initData()">
    <mx:Script><![CDATA[
```

```
        import mx.collections.*;
        private var DGArray:Array = [
           {Artist:'Pavement', Album:'Slanted and Enchanted', Price:11.99},
           {Artist:'Pavement', Album:'Brighten the Corners', Price:11.99}];

        [Bindable]
        public var initDG:ArrayCollection;
        //Initialize initDG ArrayCollection variable from the Array.
        public function initData():void {
            initDG=new ArrayCollection(DGArray);
        }

        public function addNewItem():void {
            var o:Object;
           o = {Artist:'Pavement', Album:'Nipped and Tucked', Price:11.99};
            initDG.addItem(o);
            callLater(focusNewRow);
        }

        public function focusNewRow():void {
            myGrid.editedItemPosition = {
                columnIndex:0,rowIndex:myGrid.dataProvider.length-1
            };
        }

    ]]></mx:Script>

    <mx:DataGrid id="myGrid" width="350" height="200"
dataProvider="{initDG}"  editable="true">
        <mx:columns>
            <mx:Array>
                <mx:DataGridColumn dataField="Album" />
                <mx:DataGridColumn dataField="Price" />
            </mx:Array>
        </mx:columns>
    </mx:DataGrid>

    <mx:Button id="b1" label="Add New Item" click="addNewItem()"/>

</mx:Application>
```

Another use of the `callLater()` method is to create a recursive method. Because the function is called only after the next screen refresh (or frame), the `callLater()` method can be used to create animations or scroll text with methods that reference themselves.

The following example scrolls ticker symbols across the screen and lets users adjust the speed with an HSlider control:

```
<?xml version="1.0"?>
<!-- layoutperformance/CallLaterTicker.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
    [Bindable]
    public var text:String =
"SLAR:95.5....TIBA:42....RTF:34.15....AST:23.42";
    [Bindable]
    public var speed:Number = 5;

    public function initTicker():void {
        theText.move( this.width+10, 0 ); // Start the text on the right
side.
        callLater(moveText);
    }

    public function moveText():void {
        var xpos:Number = theText.x;
        if( xpos-speed+theText.width < 0 ) {
            xpos = this.width+10; // Start the text on the right side.
        }
        xpos -= speed;
        theText.move(xpos,0);
        callLater(moveText);
    }

    public function changeSpeed():void {
        speed = speedSelector.value;
    }
    ]]></mx:Script>

    <mx:Panel title="Ticker Sample" width="400" height="200">
        <mx:Canvas creationComplete="initTicker()"
                horizontalScrollPolicy="off" backgroundColor="red"
color="white"
                width="100%">
            <mx:Label id="theText" text="{text}" y="0"/>
        </mx:Canvas>
        <mx:HBox>
            <mx:Label text="Speed:"/>
            <mx:HSlider minimum="1" maximum="10" value="{speed}"
                id="speedSelector" snapInterval="1" tickInterval="1"
                change="changeSpeed()"/>
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```

CHAPTER 7

Building Overview

7

This topic introduces the tools available in a typical Adobe Flex development environment. It also describes the asset types that you work with to create Flex applications.

Contents

About the Flex development tools 153
About application files 156

About the Flex development tools

This section introduces several aspects of the typical Flex development environment.

Configuration files

Be familiar with the ways to configure your development environment. Adobe Flex Software Development Kit (SDK) and Adobe Flex Data Services primarily provide XML files that you use to configure the settings.

The flex-config.xml file defines the default compiler options for the compilers. The flex-webtier-config.xml defines the configuration settings for the server running Flex Data Services.

In addition to server and compiler configuration settings, you can also modify the messaging and data management settings, the JVM heap size, Adobe Flash Player settings, and logging and caching settings.

For more information about configuring your Flex 2 SDK and Flex Data Services environment, see Chapter 8, "Flex 2 SDK and Flex Data Services Configuration," on page 161.

Compilers

Flex includes application compilers and component compilers. You use the application compilers to compile SWF files from MXML and other source files. You use the component compilers to compile SWC files from component files. You can then use SWC files as dynamic or static libraries with your Flex applications.

The application compilers take the following forms:

- Flex Builder project compiler. The Flex Builder application compiler is opened by Flex Builder for Flex Projects and ActionScript Projects.
- mxmlc command-line compiler. You open the mxlmc compiler from the command line to create a SWF file that you then deploy to a website.
- Web-tier compiler. The web-tier compiler is a set of servlets that run in a J2EE application server such as JRun. The application server passes requests for *.mxml files to the servlet container, which then compiles a SWF file and return the results to the client. In a production environment, you generally precompile Flex applications so that they are not compiled at request time.

The component compilers take the following forms:

- Flex Builder library project compiler. The Flex Builder component compiler is openedd by Flex Builder for Flex library projects.
- compc command-line compiler. You open the compc compiler from the command line to create SWC files. You can use these SWC files as static component libraries, themes, or runtime shared libraries (RSLs).

For information on using the compilers, see Chapter 9, “Using the Flex Compilers,” on page 179.

Debugger

To test your applications, you run the application SWF files in a web browser or the stand-alone Flash Player. If you encounter errors in your applications, you can use the debugging tools to set and manage breakpoints in your code; control application execution by suspending, resuming, and terminating the application; step into and over the code statements; select critical variables to watch; evaluate watch expressions while the application is running; and so on.

Flex provides the following debugging tools:

Flex Builder debugger The Flex Builder Debugging perspective provides all of the debugging tools you expect from a robust, full-featured development tool. You can set and manage breakpoints; control application execution by suspending, resuming, and terminating the application; step into and over the code; watch variables; evaluate expressions; and so on. For more information, see *Using Flex Builder 2*.

The fdb command-line debugger The fdb command-line debugger provides a command-line interface to the debugging experience. With fdb, you can step into code, add breakpoints, check variables, and perform many of the same tasks you can with the Flex Builder visual debugger. For more information, see Chapter 12, "Using the Command-Line Debugger," on page 269.

Loggers

You can log messages at several different points in a Flex application's life cycle. You can log messages when you compile the application, when you deploy it to a web application server, or when a client runs it. You can log messages on the server or on the client. These messages are useful for informational, diagnostic, and debugging activities. This section describes the various logging mechanisms that you can use when working with Flex applications.

Flex includes the following logging mechanisms:

Client-side logging When you use the debugger version of Flash Player, you can use the `trace()` global method to write out messages or configure a TraceTarget to customize log levels of applications for data services-based applications. For more information, see "Client-side logging and debugging" on page 251.

Compiler logging When compiling your Flex applications from the command line and in Flex Builder, you can view deprecation and warning messages, and sources of fatal errors. For more information, see "Compiler logging" on page 264.

Web-tier logging The Flex web application provides some control over logging messages for the FlexMxmlServlet and lets you write the web-tier compiler log messages to your application server's logs. For more information, see "Web-tier logging" on page 265.

Server-side data services logging You can perform server-side logging for data service messages. You configure server-side logging in the logging section of the Flex services configuration file. By default, the output is sent to the System.out file, but you can also configure the logging to use your application server's logging mechanism. For more information, see Chapter 43, "Configuring Data Services," in the *Flex 2 Developer's Guide*.

About application files

Flex applications can use many types of application files such as classes, component libraries, theme files, and runtime shared libraries (RSLs). This section describes the types of files you can use in your Flex applications.

Component classes

You can use any number of component classes in your Flex applications. These classes can be MXML or ActionScript files. You can use classes to extend existing components or define new ones.

Component classes can take the form of MXML, ActionScript files, or as SWC files. In MXML or ActionScript files, the components are not compiled but reside in a directory structure that is part of your compiler's source path. SWC files are described in "SWC files" on page 156.

Component libraries are not dynamically linked unless they are used in a Runtime Shared Library (RSL). Component classes are statically linked at compile time, which means that they must be in the compiler's source path. For information about creating and using custom component classes, see *Creating and Extending Flex 2 Components*.

SWC files

A *SWC* file is an archive file for Flex components and other assets. SWC files contain a SWF file and a catalog.xml file. The SWF file inside the SWC file implements the compiled component or group of components and includes embedded resources as symbols. Flex applications extract the SWF file from a SWC file, and use the SWF file's contents when the application refers to resources in that SWC file. The catalog.xml file lists of the contents of the component package and its individual components.

You compile SWC files by using the component compilers. These include the compc command-line compiler and the Flex Builder Library Project compiler. SWC files can be component libraries, RSLs, theme files, and resource bundles.

To include a SWC file in your application at compile time, it must be located in the library path. For more information about SWC files, see "About SWC files" on page 229.

Component libraries

A component library is a SWC file that contains classes and other assets that your Flex application uses. The component library's file structure defines the package system that the components are in.

Typically, component libraries are statically linked into your application, which means that the compiler compiles it into the SWF file before the user downloads that file.

To build a component library SWC file, you use the `include-classes`, `include-namespaces`, and `include-sources` component compiler options. For more information on building component libraries, see "Using the component compiler" on page 215.

Runtime Shared Libraries

You can use shared assets that can be separately downloaded and cached on the client in Flex. These shared assets are loaded by multiple applications at run time, but must be transferred only once to the client. These shared files are known as *Runtime Shared Libraries* or *RSLs*.

RSLs are the only kind of application asset that is dynamically linked into your Flex application. When you compile your application, the RSL source files must be available to the compiler so that it can perform proper link checking. The assets themselves are not included in the application SWF file, but are only referenced at run time.

To create an RSL SWC file, you add files to a library by using the `include-classes` and `include-namespaces` component compiler options. To use RSLs when compiling your application, you use the `external-library-path`, `externs`, `load-externs`, and `runtime-shared-libraries` application compiler options. The `external-library-path`, `externs`, and `load-externs` options provide the compile-time location of the libraries. The `runtime-shared-libraries` option provides the run-time location of the shared library. The compiler requires this for dynamic linking.

For more information, see Chapter 10, "Using Runtime Shared Libraries," on page 233.

Themes

A *theme* defines the look and feel of a Flex application. A theme can define something as simple as the color scheme or common font for an application, or it can be a complete reskinning of all the components used by the application.

Themes usually take the form of a SWC file. However, themes can also be composed of a CSS file and embedded graphical resources, such as symbols from a SWF file.

Theme files must be available to the compiler at compile-time. You build a theme file by using the `include-file` and `include-classes` component compiler options to add skin files and style sheets to a SWC file. You then reference the theme SWC file when you compile the main Flex application by using the `theme` application compiler option.

For more information about themes, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*.

Resource bundles

You can package libraries of localized properties files and ActionScript classes into a SWC file. The application compiler can then statically use this SWC file as a resource bundle. For more information about creating and using resource bundles, see Chapter 25, "Localizing Flex Applications," in *Flex 2 Developer's Guide*.

Other assets

Other application assets include images, fonts, movies, and sound files. You can embed these assets at compile time or access them at run time.

When you embed an asset, you compile it into your application's SWF file. The advantage of embedding an asset is that it is included in the SWF file, and can be accessed faster than when the application has to load it from a remote location at run time. The disadvantage of embedding an asset is that your SWF file is larger than if you load the resource at run time.

The alternative to embedding an asset is to load the asset at run time. You can load an asset from the local file system in which the SWF file runs, or you can access a remote asset, typically though an HTTP request over a network.

Embedded assets load immediately, because they are already part of the Flex SWF file. However, they add to the size of your application and slow down the application initialization process. Embedded assets also require you to recompile your applications whenever your asset files change.

For more information, see Chapter 30, "Embedding Assets," in *Flex 2 Developer's Guide*.

PART 2

Building Flex Applications

2

This part describes how to compile, debug, and test Flex applications.

The following topics are included:

Chapter 7: Building Overview 153
Chapter 8: Flex 2 SDK and Flex Data Services Configuration . 161
Chapter 9: Using the Flex Compilers 179
Chapter 10: Using Runtime Shared Libraries. 233
Chapter 11: Logging 245
Chapter 12: Using the Command-Line Debugger. 269
Chapter 13: Using ASDoc. 289
Chapter 14: Creating Applications for Testing 311

CHAPTER 8

Flex 2 SDK and Flex Data Services Configuration

8

This section provides an overview of how to use the configuration files included with Flex 2 SDK and Flex Data Services. You use these files to configure the compilers and other aspects of the product. For Flex Data Services, be aware that Flex is a web application running within a web application server. As a result, you make many configuration settings at the server level. For more information, consult your web application server's documentation.

This topic does not describe how to configure Flex Builder. That is described in *Using Flex Builder 2*.

Contents

About configuration files 161
Flex 2 SDK configuration 165
Flex Data Services configuration 168
Flash Player configuration 178

About configuration files

This section helps you understand how the the various compilers and servers use the configuration files.

Applying license keys

When you upgrade Flex Data Services from a trial edition to the commercial edition or add Flex Charting components, you must add the new license key to the license.properties files.

If you are using Flex 2 SDK and install charting, edit the license.properties file in the *sdk_install_dir*/frameworks directory.

If you are running Flex Data Services, you must edit the license.properties file in the following locations:

- In every deployed WAR file. The license.properties file is in the WEB-INF/flex directory. By default, a license.properties file is in the flex, flex-admin, and samples WAR files. If you create a web application that includes Flex, you must also edit the license.properties file in that WAR file.
- In the *flex_install_dir*/flex_sdk_2/frameworks directory.

Root variables

This section refers to the *flex_install_dir* variable. For Flex SDK, this is the top-level directory where you installed the SDK. Under this directory are the bin, frameworks, lib, and samples directories. For Flex Data Services, this is the top-level directory where you installed Flex Data Services. Under this directory are the flex_sdk_2, resources, and UninstallerData directories, and, optionally, the jrun4 directory.

This section refers to the *flex_app_root* directory as the top level location for many files. If you deploy Flex Data Services as a WAR file on an application server, this is the root of the WAR file. For example, if you installed JRun with the integrated JRun application server, flex_app_root refers to the *flex_install_dir*/jrun4/servers/default/flex directory.

Configuration files layout

The layout of the configuration files for Flex SDK is simple. It includes a jvm.config file, fdb command-line debugger shell script, and the mxmlc and compc command-line compiler shell scripts for configuring the JVM that the compiler uses. It also includes the flex-config.xml file that sets the compiler options, as well as executable files for fdb, mxmlc, and compc.

The layout of the configuration files for Flex SDK is as follows:

```
flex_install_dir/
    bin/jvm.config
    bin/mxmlc
    bin/mxmlc.exe
    bin/compc
    bin/compc.exe
    bin/fdb
    bin/fdb.exe
    frameworks/flex-config.xml
```

The layout of the configuration files for the Flex Data Services is relatively complex. Flex Data Services includes the Flex SDK, so it stores the shell scripts and jvm.config files in the flex_sdk_2 directory:

```
flex_install_dir/
     flex_sdk_2/bin/jvm.config
     flex_sdk_2/bin/mxmlc
     flex_sdk_2/bin/compc
     flex_sdk_2/frameworks/flex-config.xml
```

In addition to these tools, you can edit the web application and web-tier compiler settings with the following configuration files:

```
flex_app_root/
     web.xml
     WEB-INF/flex/flex-config.xml
```

Also, the Flex Data Services web application includes the data services configuration files:

```
flex_app_root/WEB-INF/flex/
     data-management-config.xml fds-
     services-config.xml
     messaging-config.xml
     proxy-config.xml
     remoting-config.xml
```

In addition to these configuration files, Flex Data Services can include a JRun application server, which has its own configuration files. Edit the following files to configure the *default* JRun server:

```
flex_install_dir/jrun4/
     bin/jvm.config
     servers/default/flex/WEB-INF/jrun-web.xml
     servers/default/SERVER-INF/jrun.xml
```

The following table describes the configuration files shown in the previous lists:

File Name	Description
fdb	Configures the JVM for the fdb command-line debugger shell script. For more information, see Chapter 12, "Using the Command-Line Debugger," on page 269.
flex-config.xml	Configures the Flex compilers. Flex Data Services includes two copies of this file: one in the *flex_install_dir*/flex_sdk_2/frameworks directory for use with the command line compilers. The other copy of this file is in the *flex_app_root*/WEB-INF/flex/ directory for use with the web-tier compiler.

File Name	Description
services-config.xml	Defines the basic settings for data services such as logging and security. This file does not usually contain service-specific destination definitions. You typically define individual service destination definitions in the fds-*service_name*.xml files, which are referenced by services-config.xml. For more information, see "Data services configuration" on page 170.
fds-*service_name*.xml	Contains service-specific configuration information. For more information, see "Data services configuration" on page 170.
jrun.xml	Defines settings for the optional, integrated JRun application server. You use this file to configure settings for JRun services such as logging, web server, and the security. For more information, see "Configuring JRun servers" on page 406.
jrun-web.xml	Contains web application elements that are specific to the JRun web application server. For more information, see "Configuring JRun servers" on page 406.
jvm.config	Configures the JVM used by your J2EE application server or compiler. You use this file to pass arguments to the JVM, including memory management and source path definitions. For more information, see "JVM configuration" on page 166.
mm.cfg	Configures client-side logging for the debugger version of Flash Player. For more information, see "Configuring the debugger version of Flash Player" on page 248.
mms.cfg	Configures Flash Player auto-update and other settings. For more information, see the Flash Player documentation.
mxmlc and compc shell scripts	Configures the JVM for the mxmlc and compc shell scripts. For more information, see "Command-line compiler configuration" on page 165.
web.xml	Configures your Flex application to run on the J2EE web application server. You use this file to define context parameters, filters, servlet mappings, JSP tag libraries, error handling, and other settings for your web application. For more information, see "Servlet configuration" on page 171.

About application and server verbiage

This section discusses two types of applications: web applications and Flex applications. *Web applications* are applications that run inside a container. The container typically provides services such as state management, fault handling, and data interoperation. Often, a web application takes the form of a WAR file that you expand into a directory on your application server. Flex Data Services is a web application that runs on top of your application server and its associated web server. It is made up of servlets that manage the data services and applications.

Flex applications are typically *.mxml files plus helper files such as images, ActionScript classes, and custom components. You write Flex applications using the MXML syntax and run those applications inside the Flex web application. A single Flex web application can contain any number of Flex applications such as a WYSIWIG editor, a shopping cart, or a stock charting application.

The JRun application server is the server on which web applications run. The default JRun server is an instance of the JRun application server. You can add any number of instances of the JRun server. Each must be started and maintained separately.

Flex 2 SDK configuration

Flex 2 SDK includes the mxmlc and compc command-line compilers. You use mxmlc to compile Flex applications from MXML, ActionScript, and other source files. You use the compc compiler to compile component libraries, Runtime Shared Libraries (RSLs), and theme files.

The compilers are located in the *sdk_install_dir*/bin directory. You configure the compiler options with the flex-config.xml file. The compilers use the Java JRE. As a result, you can also configure settings such as memory allocation and source path with the JVM arguments.

Command-line compiler configuration

The flex-config.xml file defines the default compiler options for the compc and mxmlc command-line compilers. You can use this file to set options such as debugging, SWF file metadata, and themes to apply to your application. For a complete list of compiler options, see "Using the application compiler" on page 195 and "Using the component compiler" on page 215.

The flex-config.xml file is located in the *flex_install_dir*/frameworks directory. If you change the location of this file relative to the location of the command-line compilers, you can use the `load-config` compiler option to point to its new location.

You can also use a local configuration file that overrides the compiler options of the flex-config.xml file. You give this local configuration file the same name as the MXML file, plus -config.xml" and store it in the same directory. When you compile your MXML file, the compiler looks for a local configuration file first, then the flex-config.xml file.

For more information on compiler configuration files, see "About configuration files" on page 190.

JVM configuration

The Flex compilers use the Java JRE. Configuring the JVM can result in faster and more efficient compilations. Without a JVM, you cannot use the mxmlc and compc command-line compilers. You can configure JVM settings such as the Java source path, Java library path, and memory settings.

On Windows, you use the compc.exe and mxmlc.exe executable files in the bin directory to compile Flex applications and component libraries. You use the fdb.exe executable file in the bin directory to debug applications. The executable files use the jvm.config file to set JVM arguments. The jvm.config file is in the same directory as the executable files. If you move it or the executable files to another directory, they use their default settings and not the settings defined in the jvm.config file.

The fdb, compc, and mxmlc shell scripts (for UNIX, Linux, or Windows systems running a UNIX-shell emulator such as Cygwin) do not take a configuration file. You set the JVM arguments inside the shell script file.

Locating the jvm.config file

The location of the jvm.config file depends on which Flex product you use. The following table shows the location and use of the product-specific jvm.config files:

Product	Location of jvm.config	Description
Flex 2 SDK	*sdk_install_dir*/bin	Used by the Java process opened by the mxmlc and compc command-line executable files.
Flex Data Services	*flex_install_dir*/jrun4/bin	Used by the web-tier compiler when used with the integrated JRun application server. If you deploy the flex.war file on another application server, consult that product's documentation.
Flex Data Services	*flex_install_dir*/ flex_sdk_2/bin	Used by the Java process opened by the mxmlc and compc command-line executable files.

Changing the JVM heap size

The most common JVM configuration is to set the size of the Java heap. The Java heap is the amount of memory reserved for the JVM. The actual size of the heap during run time varies as classes are loaded and unloaded. If the heap requires more memory than the maximum amount allocated, performance will suffer as the JVM performs garbage collection to maintain enough free memory for the applications to run.

You can set the initial heap size (or minimum) and the maximum heap size on most JVMs. By providing a larger heap size, you give the JVM more memory with which to defer garbage collection. However, you must not assign all of the system's memory to the Java heap so that other processes can run optimally.

To set the initial heap size on the Sun HotSpot JVM, change the value of the `Xms` property. To change the maximum heap size, change the value of the `Xmx` property. The following example sets the initial heap size to 256M and the maximum heap size to 512M:

```
java.args=-Xms256m -Xmx512m -Dsun.io.useCanonCaches=false
```

In addition to increasing your JVM's heap size, you can tune the JVM in other ways. Some JVMs provide more granular control over garbage collecting, threading, and logging. For more information, consult your JVM documentation or view the options on the command line. If you are using the Sun HotSpot JVM, for example, you can enter `java -X` or `java -D` on the command line to see a list of configuration options.

In many cases, you can also use a different JVM. Benchmark your Flex application and the application server on several different JVMs. Choose the JVM that provides you with the best performance.

Setting the `useCanonCaches` argument to `false` is required to support Windows file names.

Flex Data Services configuration

Flex Data Services runs as a web application within a web application server. You deploy the flex.war file to the server. You use the underlying application server settings to configure settings such as security and logging, but you can configure other settings in the Flex application's configuration files.

If you installed Flex Data Services with the integrated JRun Java application server, see Chapter 18, "Configuring JRun," on page 401 for additional server configuration information.

Compiler configuration

Flex Data Services uses the web-tier compiler to compile MXML files and other assets into a SWF file. You configure the web-tier compiler by editing the *flex_app_root*/WEB-INF/flex/flex-webtier-config.xml file. Options in this file include enabling production mode and changing debugging settings. The settings apply to all Flex applications found under the *flex_app_root* directory.

The flex-webtier-config.xml file also imports the contents of the flex-config.xml file for additional compiler settings such as the application's library path and source path. For more information on the flex-config.xml file, see "Command-line compiler configuration" on page 165.

If you change options in the flex-webtier-config.xml file, restart your application server.

In addition to the flex-webtier-config.xml file, you can change some settings of the web-tier compiler at run time using query string parameters. For example, you can enable accessibility using a URL request such as the following:

```
http://www.mysite.com/MyApp.mxml?accessible=true
```

The Flex Data Services server converts query string parameters to `flashVars` variables in the wrapper. The web-tier compiler interprets these variables and compiles the Flex application accordingly.

In most cases, enabling production mode prevents the query string parameters from taking effect.

The following table describes query string parameters that you can use to override compiler settings for the web-tier compiler:

Query string parameter	Description
accessible=true|false	Enables accessibility features when compiling the Flex application or SWC file. This option overrides the `accessible` compiler option in the flex-config.xml file.
debug=true|false	Adds debugging information to the application.swf file. This option overrides the `debug` compiler option in the flex-config.xml file. When production mode is enabled, setting this option has no effect.
recompile=true|false	Forces a full compilation regardless of caching, if one is required by the caching settings.
showAllWarnings=true|false	Enables or disables all warning messages. This setting overrides all other warning settings, such as `show-coach-warnings` and `show-binding-warnings`. This option has no equivalent in the flex-config.xml file. When production mode is enabled, setting this option has no effect.
showBindingWarnings=true|false	Shows a warning when Flash Player cannot detect changes to a bound property. This option overrides the `show-binding-warnings` compiler option in the flex-config.xml file. When production mode is enabled, setting this option has no effect.
verboseStackTraces=true|false	Shows additional information in the stack traces. This option overrides the `verbose-stacktraces` compiler option in the flex-config.xml file. When production mode is enabled, setting this option has no effect.

For more information about the wrapper, see Chapter 16, “Creating a Wrapper,” on page 367.

JVM configuration

Flex is an application running on an application server. As a result, the performance of your Flex applications can be affected by the underlying JVM of the application server.

Where you set your JVM's arguments depends on your application server's configuration. And depending on which JVM you are using, you might have more or less memory management options available to you. On the integrated version of JRun, for example, change the values of the `jvm.args` property in the *flex_install_dir*/jrun4/bin/jvm.config file. For more information, see "Changing the JVM heap size" on page 167.

Data services configuration

Flex Data Services includes several configuration files that let you configure the data services. These files are located in the *flex_app_root*/WEB-INF/flex directory.

The primary file is services-config.xml. This file defines the basic settings for data services such as logging and security. This file does not define service destinations. Individual service destination definitions are set in the other files referenced by services-config.xml. This configuration lets you separate the types of service destination definitions into multiple files, with each file dedicated to defining destinations.

The following table describes the default configuration files included with Flex Data Services:

Configuration File	Description
services-config.xml	This is the default location of configuration information for all data services information. All other data services configuration files are referenced by this file with the `service-include` tag. For example: `<service-include file-path="messaging-config.xml"/>` The default configuration defines services, logging, security, and system settings for data services. It also references the other configuration files that define a service.
data-management-config.xml	Defines a Data Management service for use by applications running on Flex Data Services. The Flex Data Management Service that you define in this configuration file works in conjunction with a client-side Data Service component to distribute and synchronize data among multiple client applications.
messaging-config.xml	Defines Message Service destinations for use by applications running on Flex Data Services.
proxy-config.xml	Defines proxy service destinations for Web Services and HTTP services for applications running on Flex Data Services.
remoting-config.xml	Defines RemoteObject service destinations for use by applications running on Flex Data Services.

For more information on configuring Flex data services, see *Flex 2 Developer's Guide*.

Servlet configuration

Flex Data Services uses servlets that intercept requests for *.mxml and *.swf files. These servlets open the web-tier compiler to compile and return a SWF file and wrapper.

Each Flex Data Services web application contains configuration files that you can edit to customize Flex Data Services. You can use the *flex_app_root*/WEB-INF/flex/web.xml file to change settings such as the servlet mappings, paths, and other settings.

The following table describes some of the servlets defined in the web.xml file:

Servlet	Description
FlexMxmlServlet	Defines the servlet wrapper for the web-tier compiler. This servlet is opened for all *.mxml requests. The definition of this servlet includes the location of the web-tier compiler's configuration file.
FlexSwfServlet	Returns SWF files. This servlet is opened for all *.swf requests.
MessageBrokerServlet	Manages the Messaging Service. This servlet is opened for all requests matching the /messagebroker/* pattern. The definition of this servlet includes the location of the services-config.xml configuration file.

You can view the servlet mappings that define the patterns that Flex uses to open the web-tier compiler, as the following example shows:

```
<servlet-mapping>
  <servlet-name>FlexMxmlServlet</servlet-name>
  <url-pattern>*.mxml</url-pattern>
</servlet-mapping>
```

If you are using the JRun integrated server, see Chapter 18, "Configuring JRun," on page 401 for specific configuration information.

Logging configuration

Flex Data Services provides logging for the following types of messages:

Startup messages from the FlexMxmlServlet These messages are primarily informational. For example, when the servlet starts, you will get information about your license version. For more information, see "Configuring web application logging" on page 265.

Web-tier compiler messages When the server running Flex Data Services responds to a request for an MXML file, it opens the web-tier compiler. The compiler can write error and warning messages to the wrapper and the web application logging mechanism. For more information, see "Configuring web-tier compiler logging" on page 267.

JRun application server messages If you are using the integrated JRun application server with Flex Data Services or Flex Builder, you can configure the JRun logging mechanism by using the *flex_install_dir*/jrun4/servers/default/SERVER-INF/jrun.xml file. For more information, see "Configuring JRun logging" on page 410.

Server-side caching configuration

The first time an MXML file is requested, the Flex web-tier compiler compiles the application's SWF file and caches it in memory. On subsequent requests from *different* clients, Flex returns the cached SWF file. On subsequent requests from the *same* client, Flex usually returns a `304 Not Modified` header. In this case, the client then reads the SWF file out of the browser's local cache.

The caching mechanism polls the Flex files at regular intervals to determine if new files should be returned or if a new application should be compiled. If the MXML file changes, Flex recompiles the application and replaces the existing SWF in its content cache with the new one. Flex stores caching information in the *flex_root*/WEB-INF/flex/cache.dep file.

In addition to SWF files, Flex also caches generated SWC files and other asset files in memory. On the client side, these files are all cached by the browser. Flex does not cache images or other files that are included by reference in applications. Image caching is usually handled by the browser or by the web server.

Embedded images and resources are stored inside the application (whereas included resources are sent separate from the application's SWF file). If the source file for an embedded resource changes, Flex recompiles the application on the next request. Flex then replaces the existing SWF file in the content cache with the newly compiled file.

In production mode, changes to compiled files are only recognized at server startup.

Adding most query string parameters or changing `flashVars` variables in the wrapper to a request do not cause Flex to recompile the application.

Caching is enabled by default.

For information on preventing client-side caching, see "Preventing client-side caching" on page 95.

Cache settings

The default cache settings in the flex-webtier-config.xml file are as follows:

```
<cache>
  <content-size>200</content-size>
  <http-maximum-age>1</http-maximum-age>
  <file-watcher-interval>1</file-watcher-interval>
</cache>
```

The following table describes the child tags of `<cache>`:

Child tag	Description
`<content-size>`	The maximum number of files that Flex Data Services caches. The default value is 10. When the maximum number of files reaches the size limit, Flex Data Services removes the least-recently used files from the cache. The wrapper and other generated files such as AC_OETags.js are not stored in the cache. Flex creates additional cache entries for different URL pairs (for example, page.mxml and page.mxml?accessible=true). By setting accessible=true in the URL, you create a unique SWF file that is added to the cache. Not all URL variables force a recompilation. For more information on the HTML wrapper, see Chapter 16, “Creating a Wrapper,” on page 367.
`<http-maximum-age>`	The maximum amount of time, in seconds, during which a browser can return its copy of the file without checking the server for freshness of the document. The default value is `1` second. This setting sets the value of the Cache-Control HTTP header. Flex assigns the value of this property to the max-age setting in that header.
`<file-watcher-interval>`	The amount of time Flex waits before polling for changes to MXML and dependent files (such as MXML component files, SWC files, images, included *.as files, and ActionScript class files). When Flex Data Services finds that a file has changed, Flex Data Services recompiles the SWF file and replaces the cached SWF file with the new one. If production mode is enabled, Flex only checks dependent files when the server starts up. The default value is `1` second.

Using incremental compilation

You can use incremental compilation to decrease the time it takes to compile an application or component library with the Flex application compilers. When incremental compilation is enabled, the compiler inspects changes to the bytecode between revisions and only recompiles the section of bytecode that changed.

To enable incremental compilation for the web-tier compiler, you set the value of the `incremental-compile` option in the flex-webtier-config.xml file to `true`. The default is `true`.

For more information about incremental compilation, see "About incremental compilation" on page 214.

Caching fonts and glyphs

Fonts and glyphs can be expensive to reload every time you compile them into an application. Because of this, Flex Data Services lets you cache the fonts and glyphs used by your Flex applications running on a Flex Data Services server. By caching your fonts, you can improve response time of components that use embedded fonts.

Flex Data Services caches a specified number of embedded font faces in memory. The default number of fonts is 20. You can control the maximum number of fonts and character glyph outlines for each font face to cache using the `<max-cached-fonts>` and the `<max-glyphs-per-face>` settings in the flex-config.xml file.

The following table describes the caching-related child tags of the `<fonts>` tag:

Child tag	Description
`<max-cached-fonts>`	Sets the number of embedded font faces that Flex stores in its cache before rotating out older font faces. Each cached font uses up memory related to the size of the TrueType font (TTF) file. The default value is `20`. When the number of embedded font faces exceeds `max-cached-fonts`, Flex removes the least-recently used from the cache.
`<max-glyphs-per-face>`	Defines the maximum number of character glyph outlines to cache for each font face. The default value is 1000.

To obtain further optimizations, you can reduce the size of the font by specifying its character range. For more information on embedding fonts, see Chapter 19, "Using Fonts," in *Flex 2 Developer's Guide*.

Production mode

Production mode is a state of the Flex application that you use when the application is running live on a public-facing server. Enabling production mode affects many aspects of Flex, including debugging, logging, and caching. The following sections describe the effect that enabling production mode has on these features.

The default value of `<production-mode>` is `false`. To enable production mode, change the value of the `<production-mode>` tag to `true`, as the following example shows:

```
<production-mode>true</production-mode>
```

You must restart the server running Flex in order for changes to the production mode to take effect.

For more information about production mode, see "Enabling production mode" on page 359.

Configuring mappings

In a development environment, it is generally acceptable for testers to request an application by its full name in their browser. This can lead to unattractive request strings. But in a production environment, you will want control over how the request string looks to users who request your Flex applications.

This section shows how to map file extensions to the MXML compiler and how to set your Flex application as a the default file.

The techniques described here are standard across all J2EE-compiant application servers. For more information about them, consult your J2EE server's documentation.

Using virtual directories

Virtual directories map a request path to a real path in your file system. You can keep your files in directories outside of the application directory structure, but map them to a path inside the application directory structure.

For example, you can store files in an images folder at C:/images/production and use those images in your applications by adding a virtual directory that points to the /flex/images/production folder.

In Windows systems, use forward slashes for path separators in XML configuration files.

To add a virtual directory on the JRun application server, for example, add a virtual mapping in the *jrun_root/server_name/*WEB-INF/jrun-web.xml file. The following example adds the images virtual directory:

```
<jrun-web-app>
  <virtual-mapping>
    <resource-path>/flex/images/*</resource-path>
    <system-path>c:/images/production</system-path>
  </virtual-mapping>
</jrun-web-app>
```

To add virtual directories for non-JRun web application servers, consult your application server documentation. To enable or disable directory browsing, see "Configuring directory browsing" on page 177.

Changing the default application mapping

You can map any file as the entry point for your application using the `<welcome-file-list>` property in the web.xml file. Whatever file you specify becomes the default file that the application server returns when no file is specified in a request to the web application's context root.

The default value of `<welcome-file-list>` is index.html. You can map a request to a particular MXML file, instead. The following example instructs the application server to open MyApp.mxml by default:

```
<welcome-file-list>
   <welcome-file>MyApp.mxml</welcome-file>
</welcome-file-list>
```

In this example, the client can make the following request to get the MyApp.mxml application:

```
http://www.yourdomain.com/flex/
```

Changing the context root

The context root maps requests to the Flex application. In practice, the context root defines the URL prefix that clients use to request files in your application.

The default context root is /flex. For example, the context root in the following URL is /flex:

```
http://localhost:8700/flex/myApp.mxml
```

The value is typically equivalent to a call to the `request.getContextPath()` method.

The context root for the Flex web application is set in the *flex_app_root*/WEB-INF/flex/flex-config.xml file. To change the context root, edit the `<context-root>` tag. The following example changes the context root to /:

```
<context-root>/</context-root>
```

The ColdFusion MX standalone configuration uses a context root of /.

You can also set the value of the context root using the `context-root` compiler option, as the following example shows:

```
$ mxmlc -context-root=/samples MyApp.mxml
```

To access the context root of the web application from inside your Flex application, use the `@ContextRoot()` method. The following example sets the value of the HTTPService's `url` property and then traces that property:

```
<?xml version="1.0"?>
<!-- config/ContextRootTest.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="l1.text='url='+s1.url">
    <mx:HTTPService id="s1" url="@ContextRoot()/service.jsp"/>
    <mx:Label id="l1"/>
</mx:Application>
```

If you do not specify the value of the `context-root` compiler option, the `@ContextRoot()` method returns an exception.

In the Flex configuration files, you can use the `{context.root}` token to represent the value of the `context-root` compiler option. If you do not set the value of the context-root option, the `{context.root}` token's value is null. You commonly use the `{context.root}` token in the flex-config.xml file to specify the URLs of the gateways and proxies, and to point to Flash Player detection and deployment resources.

If you are running your MXML apps inside http://localhost:8700/flex, then you typically set "/flex" as the context root. The value of `{context.root}` includes the prefix "/". As a result, you are not required to add a forward slash before the `{context.root}` token.

Configuring directory browsing

When directory browsing is enabled, you can make a general request that does not specify a filename, and you can view the entire contents of the directory rather than receiving a File Not Found error.

By default, the default JRun server enables directory browsing.

To change the directory browsing setting, you configure the initialization parameter of the FileServlet in the SERVER-INF/*application_name*-web.xml file. The following example disables directory browsing by setting the `browsDirs` parameter to `false`:

```
<servlet>
  <servlet-name>FileServlet</servlet-name>
  <servlet-class>jrun.servlet.file.FileServlet</servlet-class>
  <init-param>
    <param-name>browseDirs</param-name>
    <param-value>false</param-value>
  </init-param>
</servlet>
```

Flash Player configuration

You can use the standard version or the debugger version of Flash Player as clients for your Flex applications. The debugger version of Flash Player can log output from the `trace()` global method as well as data services messages and custom log events.

You enable and disable logging and configure the location of the output file in the mm.cfg file. This file is located in the HOMEPATH/HOMEDRIVE/ directory. For more information on locating and editing the mm.cfg file, see "Configuring the debugger version of Flash Player" on page 248.

You can configure the standard version and the debugger version of Flash Player auto-update and other settings by using the mms.cfg file. This file is in the same directory as the mm.cfg file. For more information on auto-update, see the Flash Player documentation.

CHAPTER 9

Using the Flex Compilers

9

This topic describes using the Flex 2 application and component compilers. These compilers can be opened on the command line, in Adobe Flex Builder, and at run time in the Flex web application. This topic also describes what SWC files are and how to write manifest files for use by the compilers.

Contents

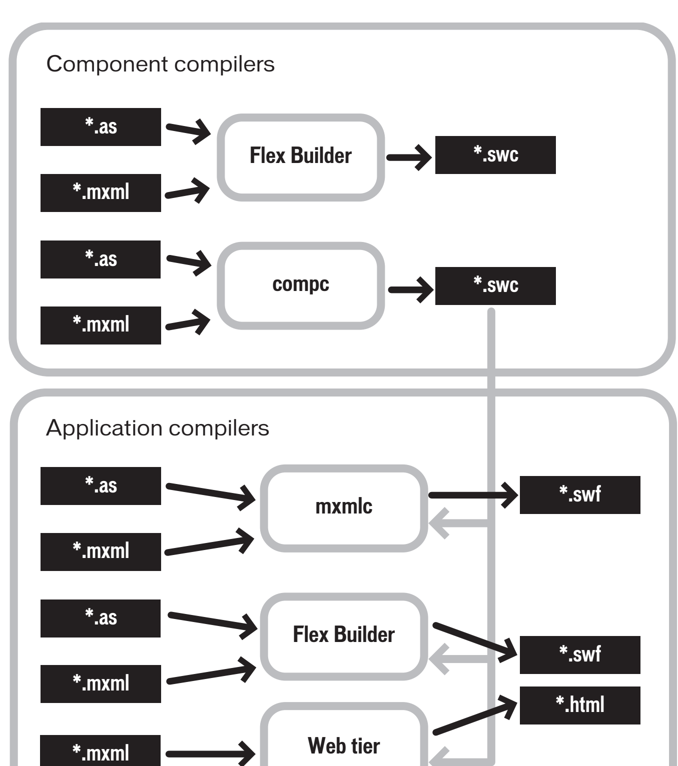
About the Flex compilers 179
About the command-line compilers 187
About configuration files 190
About option precedence 194
Using the application compiler 195
Using the component compiler 215
Viewing errors and warnings 227
About SWC files 229
About manifest files 231

About the Flex compilers

You can use Adobe Flex tools to compile applications, component libraries, themes, and run-time shared libraries (RSLs). Flex includes the following application and component compilers:

- Application compilers. The application compilers create SWF files from MXML, ActionScript, and other assets such as images, SWF files, and SWC files.
- Component compilers. The component compilers create SWC files from the same kinds of files. The application compilers then use the SWC files as component libraries, themes, or RSLs.

The following example shows the input and output of the Flex compilers:



You open the application compiler with the mxmlc command-line tool, the Flex Builder Build Project option, or with the run-time web-tier compiler. You open the component compiler with the Flex Builder Build Project option for a Library Project or with the compc command-line tool.

About the application compilers

The application compilers create SWF files that are run in an Adobe Flash Player client. The client can be a stand-alone Flash Player or a Flash Player in a browser, which takes the form of an ActiveX control for Microsoft Internet Explorer or a plug-in for Netscape-based browsers.

Flex Builder **project compiler.** The Flex Builder application compiler is opened by Flex Builder for Flex Projects and ActionScript Projects. (The component compiler is used for Library Projects.) It is similar in functionality to the mxmlc command-line compiler, although the way you set options is different. You use this compiler to compile Flex Builder projects that you will later deploy. For more information, see "Using the Flex Builder application compiler" on page 181.

The mxmlc command-line compiler. You open the mxlmc compiler from the command line to create a SWF file that you then deploy to a website. Typically, you pass the name of the application's root MXML file to the compiler. The output is a SWF file. For more information, see "Using the mxmlc application compiler" on page 182.

Web-tier compiler. The web-tier compiler is a set of servlets and servlet filters that run in a J2EE application server. The application server passes requests for *.mxml files to the servlet container, which then compiles a SWF file and return the results to the client. The Flex web application is installed with the Flex Data Services. For more information, see "Using the web-tier application compiler" on page 183.

The Flex Builder compiler, web-tier compiler, and mxmlc compiler have similar sets of options. These are described in "About the application compiler options" on page 196.

You can compile applications that are written entirely in ActionScript and contain no MXML. You can compile these "ActionScript-only" applications with the Flex Builder and mxmlc compilers. You cannot compile ActionScript-only applications with the web-tier compiler. This compiler requires that there be at least a root application MXML file.

Using the Flex Builder application compiler

You use the Flex Builder application compiler to create SWF files from MXML, ActionScript, and other source files. You use this compiler to precompile SWF files that you deploy later, or you can deploy them immediately to a server running the Flex web application.

To open the Flex Builder application compiler, you select Project > Build. The Flex Builder application compiler is opened by Flex Builder for Flex Projects and ActionScript Projects. (You use the component compiler for Library Projects.)

To edit the compiler settings, use the settings on the Project > Properties > Flex Compiler dialog box. For information on the compiler options, see "About the application compiler options" on page 196.

The Flex Builder compiler has the same options as the mxmlc compiler. Some options are implemented with GUI controls in the Flex Compiler dialog box. To set the source path and library options, select Project > Properties > Flex Build Path and use the Flex Build Path dialog box.

You can set the values of most options in the "Additional compiler arguments" field by using the same syntax as on the command line. For information about the syntax for setting options in the Flex Compiler dialog box, see "About the command-line compilers" on page 187.

By default, Flex Builder exposes the compilation options through the project properties. If you want to use a configuration file, you can create your own and pass it to the compiler by using the `load-config` option. For more information on setting compiler options with configuration files, see "About configuration files" on page 190.

In addition to generating SWF files, the Flex Builder compiler also generates an HTML wrapper that you can use when you deploy the new Flex application. The HTML wrapper includes the `<object>` and `<embed>` tags that reference the new SWF file, as well as scripts that support history management and player version detection. For more information about the HTML wrapper, see Chapter 16, "Creating a Wrapper," on page 367.

The Flex Builder application compiler uses incremental compilation by default. For more information on incremental compilation, see "About incremental compilation" on page 214.

Using the mxmlc application compiler

You use the mxmlc command-line compiler to create SWF files from MXML, AS, and other source files. You can open it as a shell script and executable file for use on Windows and UNIX systems. You use this compiler to precompile Flex applications that you deploy later.

The command-line compiler is installed with Flex 2 SDK and Flex Data Services. It is in the *flex_install_dir*/bin directory in Flex 2 SDK and the *flex_install_dir*/flex_sdk_2/bin directory in Flex Data Services. The compiler is also included in the default Flex Builder installation, in the *flex_builder_install_dir*/Flex SDK 2/bin directory.

To use the mxmlc utility, you should understand its syntax and how to use configuration files. For more information, see "About the command-line compilers" on page 187.

The basic syntax of the mxmlc utility is as follows:

```
mxmlc [options] target_file
```

The default option is the target file to compile into a SWF file, and it is required to have a value. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file; for example:

```
mxmlc -option arg1 arg2 arg3 -- target_file.mxml
```

To see a list of options for mxmlc, you can use the `help list` option, as the following example shows:

```
mxmlc -help list
```

To see a list of all options available for mxmlc, including advanced options, you use the following command:

```
mxmlc -help list advanced
```

The default output of mxmlc is *filename*.swf, where *filename* is the name of the root application file. The default output location is in the same directory as the target, unless you specify an output file and location with the `output` option.

The mxmlc command-line compiler does not generate an HTML wrapper. You must create your own wrapper to deploy a SWF file that the mxmlc compiler produced. The wrapper is used to embed the SWF object in the HTML tag. It includes the `<object>` and `<embed>` tags, as well as scripts that support Flash Player version detection and history management. For information about creating an HTML wrapper, see Chapter 16, “Creating a Wrapper,” on page 367.

The mxmlc utility uses the default compiler settings in the flex-config.xml file. This file is in the *flex_sdk_dir*/frameworks/ directory. You can change the settings in this file or use another custom configuration file. For more information on using configuration files, see “About configuration files” on page 190.

The mxmlc compiler is highly customizable with a large set of options. For information on the compiler options, see “About the application compiler options” on page 196.

You can also open the mxmlc compiler with the `java` command on the command line. For more information, see “Invoking the command-line compilers with Java” on page 189.

Using the web-tier application compiler

You use the web-tier application compiler to create SWF files from MXML, AS, and other source files at run time. The web-tier compiler is made up of servlets and servlet filters that run inside a J2EE servlet container on a Java application server.

The web-tier compiler accepts a request from an HTTP client, and compiles the result of that request into a SWF file that it returns to the client. To use the web-tier compiler, you deploy the Flex WAR file on a J2EE server. You then request the MXML file in the browser. For information on installing the Flex WAR file, see the Flex installation and deployment instructions on the Adobe website.

The web-tier compiler requires that at least one MXML file contains the root `<mx:Application>` tag. You request this MXML file in the browser. On the first request, the web-tier compiler compiles the MXML file and its assets into a SWF file. It returns an HTML wrapper that embeds the resulting SWF file to the client. The HTML wrapper includes the `<object>` and `<embed>` tags that reference the new SWF file, as well as scripts that support history management and player version detection. For more information about the HTML wrapper, see Chapter 16, "Creating a Wrapper," on page 367.

On each subsequent request from the same client, the SWF file is loaded from the browser's local cache, unless the compiled SWF file in the server's cache has a newer timestamp. On each subsequent request from other clients, the Flex web application returns the SWF file that has already been compiled. The Flex web application does not open the web-tier compiler unless one of its source files have changed.

To set options for the web-tier compiler, you edit the flex-config.xml file. This file is located in the *flex_webapp_root*/WEB-INF/flex directory. The options in this configuration file are similar to those available for the command-line and Flex Builder application compilers. For information on setting the compiler options, see "About the application compiler options" on page 196.

In addition to the compiler options you set in flex-config.xml, you can also set debugging and logging options in the flex-webtier-config.xml file, located in the same directory. You also use this file to enable production mode and to configure player version detection and caching.

You can use query string parameters as run-time arguments. Query string parameters are converted to `flashVars` variables that are passed to the SWF file. For more information on passing in request data, see Chapter 34, "Communicating with the Wrapper," in *Flex 2 Developer's Guide*.

The web-tier compiler is included for development purposes only. It is not intended to be used in a production environment. Instead, precompile Flex applications and deploy them on a production server.

About the component compiler

You use the component compiler to generate a SWC file from component source files and other asset files such as images and style sheets. A SWC file is an archive of Flex components and other assets. For more information about SWC files, see "About SWC files" on page 229.

In some ways, the component compiler is similar to the application compiler. The application compiler produces a SWF file from one or more MXML and ActionScript files; the component compiler produces a SWC file from its input files. SWC files are compressed files that contain a SWF file (library.swf), asset files, and a catalog.xml file.

You use the component compiler to create the following kinds of assets:

Component libraries. Component libraries are SWC files that contain one or more components that are used by applications. SWC files also contain the namespace information that describe the contents of the SWC file. For more information about component libraries, see "About SWC files" on page 229.

Run-time shared libraries (RSLs). RSLs are external shared assets that can be separately downloaded and cached on the client. These shared assets are loaded by any number of applications at run time. For more information on RSLs, see Chapter 10, "Using Runtime Shared Libraries," on page 233.

Themes. Themes are a combination of graphical and programmatic skins, and Cascading Style Sheets (CSS). You use themes to define the look and feel of a Flex application. For more information on themes, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*.

You open the component compiler in the following ways:

Flex Builder **Library Project compiler.** Flex Builder uses the component compiler when you create a Library Project. (The application compiler is used for Flex Projects and ActionScript Projects). For more information, see "Using the Flex Builder component compiler" on page 186.

The compc command-line compiler. You open the compc compiler from the command line to create a SWC file. For more information, see "Using the compc component compiler" on page 186.

The component compiler has many of the same options as the application compilers, as described in "About the application compiler options" on page 196. Also, the component compiler has additional options as described in "About the component compiler options" on page 215.

Like the application compiler, you can use the `load-config` option to point the component compiler to a configuration file, rather than specify command-line options or set the options in the Flex Library Compiler dialog box.

When you compile a SWC file, store the new file in a location that is not the same as the source files. If both sets of files are accessible by Flex when you compile your application, unexpected behavior can occur.

The SWC files that are produced by the component compilers do not require an HTML wrapper because they are used as themes, RSLs, or component libraries that are input to other compilers to create Flex applications. You never deploy a SWC file that users can request directly.

Using the Flex Builder component compiler

You use the component compiler in Flex Builder to create SWC files. You do this by setting up a Flex Library Project by using the New Library Project command. You add MXML and ActionScript components, style sheets, SWF files, and other assets to the project.

To open the Flex Builder component compiler, select Project > Build Project for your Flex Library Project.

You edit the compiler settings by using the settings on the Project > Properties > Flex Library Compiler dialog box. Using the "Additional compiler arguments" field in this dialog box, you can set compiler options as if you were using the command-line compiler. For information about the syntax for setting options in the Flex Library Compiler dialog box, see "About the command-line compilers" on page 187.

You can set the value of some options using the GUI controls. To set the resource options in the Flex Library Build Path dialog box, select Project > Properties > Flex Library Build Path.

Flex Builder does not expose a default configuration file to set compiler options but you can create your own and pass it to the compiler with the `load-config` option. For more information on setting compiler options with configuration files, see "About configuration files" on page 190.

Using the compc component compiler

You use the compc command-line compiler to compile SWC files from MXML, ActionScript, and other source files such as style sheets, images, and SWF files.

You can open the compc compiler as a shell script and executable file for use on Windows and UNIX systems. It is in the *flex_install_dir*/bin directory in Flex 2 SDK and the *flex_install_dir*/flex_sdk_2/bin directory in Flex Data Services.

To use the compc compiler, you should understand how to pass options and use configuration files. For more information, see "About the command-line compilers" on page 187.

The syntax of the compc compiler is as follows:

```
compc [options] -include-classes class [...]
```

The default option for compc is `include-classes`. At least one of the "include-" options is required.

To see a list of supported options for compc, you can use the `help list` option, as the following example shows:

```
compc -help list
```

The compc compiler uses the default compiler settings in the flex-config.xml file. Like the application compiler, you can change these settings or use another custom configuration file. Unlike the application compiler, however, the component compiler does not support the use of default configuration files.

You cannot use the compc compiler to create a SWC file from a FLA file or other file created in the Adobe Flash authoring environment.

You can also open the compc compiler with the `java` command on the command line. For more information, see "Invoking the command-line compilers with Java" on page 189.

About the command-line compilers

You use the mxmlc and compc command-line compilers to compile your MXML and AS files into SWF and SWC files. You can use the utilities to precompile Flex applications that you want to deploy on another server or to automate compilation in a testing environment.

To use the command-line compilers, you must have a Java run-time environment in your system path.

For Flex 2 SDK, the command-line compilers are located in the *flex_install_dir*/bin directory. For Flex Data Services, the compilers are located in the *flex_install_dir*/flex_sdk_2/bin directory. For Flex Builder, the compilers are located in the *flex_builder_install_dir*/Flex SDK 2/bin directory.

When using mxmlc and compc on the command line, you can also use a configuration file to store your options rather than list them on the command line. You can store command-line options as XML blocks in a configuration file. For more information, see "About configuration files" on page 190.

Command-line syntax

The mxmlc and compc compilers take many options. The options are listed in the help which you can view with the `help` option, as the following example shows:

```
mxmlc -help
```

This displays a menu of choices for getting help. The most common choice is to list the basic configuration options:

```
mxmlc -help list
```

To see advanced options, use the `list advanced` option, as the following example shows:

```
mxmlc -help list advanced
```

To see a list of entries whose names or descriptions include a particular String, use the following syntax:

```
mxmlc -help pattern
```

The following example returns descriptions for the `external-library-path`, `library-path`, and `runtime-shared-libraries` options:

```
mxmlc -help list library
```

For a complete description of mxmlc options, see "About the application compiler options" on page 196. For a complete description of compc options, see "About the component compiler options" on page 215.

Many command-line options, such as `show-actionscript-warnings` and `accessible`, have `true` and `false` values. You specify these values by using the following syntax:

```
mxmlc -accessible=true -show-actionscript-warnings=true
```

Some options, such as `source-path`, take a list of one or more options. You can see which options take a list by examining the help output. Square brackets (`[ ]`) that surround options indicate that the option can take a list of one or more parameters.

You can separate each entry in a list with a space or a comma. The syntax is as follows:

```
-var val1 val2
```

or

```
-var=val1, val2
```

If you do not use commas to separate entries, you terminate a list by using a double hyphen, as the following example shows:

```
-var val1 val2 -- -next_option
```

If you use commas to separate entries, you terminate a list by not using a comma after the last entry, as the following example shows:

```
-var=val1, val2 -next_option
```

You can append values to an option using the += operator. This adds the new entry to the end of the list of existing entries rather than replacing the existing entries. The following example adds the c:/myfiles directory to the `library-path` option:

```
mxmlc -library-path+=c:/myfiles
```

Using abbreviated option names

In some cases, the command-line help shows an option with dot-notation syntax; for example, `source-path` is shown as `compiler.source-path`. This notation indicates how you would set this option in a configuration file. On the command line, you can specify the option with only the final node, `source-path`, as long as that node is unique, as the following example shows:

```
mxmlc -source-path . c:/myclasses/ -- foo.mxml
```

For more information about using configuration files to store command-line options, see "About configuration files" on page 190.

Some compiler options have aliases. *Aliases* provide shortened variations of the option name to make command lines more readable and less verbose. For example, the alias for the `output` option is `o`. You can view a list of options by their aliases by using the following command:

```
mxmlc -help list aliases
```

or

```
mxmlc -help list advanced aliases
```

You can also see the aliases in the verbose help output by using the following command:

```
mxmlc -help list details
```

Invoking the command-line compilers with Java

Flex provides a simple interface to the command-line compilers. For UNIX users, there is a shell script. For Windows users, there is an executable file. These files are located in the bin directory. You can also invoke the compilers using Java. This lets you integrate the compilers into Java-based projects (such as Ant) or other utilities.

The shell scripts and executable files for the command-line compilers wrap calls to the mxmlc.jar and compc.jar JAR files. To invoke the compilers from Java, you call the JAR files directly. For Flex SDK, the JAR files are located in the *flex_install_dir*/lib/ directory. For Flex Builder, they are located in the *flex_builder_install_dir*/Flex SDK 2/lib. For Flex Data Services, the JAR files are located in the *flex_webapp_root*/WEB-INF/flex/jars directory.

To invoke a command in a JAR file, use the `java` command from the command line and specify the JAR file you want to execute with the `jar` option. You must also specify the value of the `+flexlib` option. This advanced option lets you set the root directory that is used by the compiler to locate the flex-config.xml file, among other files. You typically point it to your frameworks directory. From there, the compiler can detect the location of other configuration files.

The following example compiles MyApp.mxml into a SWF file using the JAR file to invoke the mxmlc compiler:

```
java -jar ../lib/mxmlc.jar +flexlib c:/flex_2_sdk/frameworks
  c:/flex2/MyApp.mxml
```

You pass all other options as you would when you open the command-line compilers. The following example sets the locale and source path when compiling MyApp:

```
java -jar ../lib/mxmlc.jar +flexlib c:/flex_2_sdk/frameworks
  -locale en_US -source-path locale/{locale} c:/flex2/MyApp.mxml
```

About configuration files

Configuration files can be used by the command-line utilities, Flex Builder, and the web-tier compiler.

Flex includes a default configuration file named flex-config.xml. This configuration file contains most of the default compiler settings for the application and component compilers. You can customize this file or create your own custom configuration file.

Flex Data Services include the flex-config.xml file in the *flex_webapp_root*/WEB-INF/flex directory. Flex 2 SDK includes the flex-config.xml file in the *flex_install_dir*/frameworks directory.

The Flex Builder compilers do not use a flex-config.xml file by default. The default settings are stored internally. You can, however, create a custom configuration file and pass it to the Flex Builder compilers by using the `load-config` option. Flex Builder includes a copy of the flex-config.xml file that you can use as a template for your custom configuration file. This file located in the *flex_builder_install_dir*/Flex SDK 2/frameworks directory.

You can generate a configuration file with the current settings by using the `dump-config` option, as the following example shows:

```
mxmlc -dump-config myapp-config.xml
```

Locating configuration files

You can specify the location of a configuration file by using the `load-config` option. The target configuration file can be the default flex-config.xml file, or it can be a custom configuration file. The following example loads a custom configuration file:

```
compc -load-config=myconfig.xml
```

If you specify the filename with the += operator, your loaded configuration file is used *in addition to* and not instead of the flex-config.xml file:

```
compc -load-config+=myconfig.xml
```

With the mxmlc compiler, you can also use a local configuration file. A *local configuration file* does not require you to point to it on the command line. Rather, Flex examines the same directory as the target MXML file for a configuration file with the same name (one that matches the *filename*-config.xml filename). If it finds a file, it uses it in conjunction with the flex-config.xml file. You can also specify a configuration file by using the `load-config` option with the += operator.

For example, if your application's top-level file is called MyApp.mxml, the compiler first checks for a MyApp-config.xml file for configuration settings. With this feature, you can easily compile multiple applications using different configuration options without changing your command-line options or your flex-config.xml file.

Options in the local configuration file take precedence over options set in the flex-config.xml file. Options set in a configuration file that the `load-config` option specify take precedence over the local configuration file. Command-line settings take precedence over all configuration file settings. For more information on the precedence of compiler options, see "About option precedence" on page 194.

Configuration file syntax

You store values in a configuration file in XML blocks. In general, the tags you use match the command-line options. This section describes the syntax of the XML blocks in a configuration file, and how to find the appropriate tags for a particular compiler option.

About the root tag

The root tag of the default configuration file, flex-config.xml, is `<flex-config>`. If you write a custom configuration file, it must also have this root tag. Compiler configuration files must also have an XML declaration tag, as the following example shows:

```
<?xml version="1.0"?>
<flex-config xmlns="http://www.adobe.com/2006/flex-config">
```

You must close the `<flex-config>` tag as you would any other XML tag. All compiler configuration files must be closed with the following tag:

```
</flex-config>
```

In general, the second tag in a configuration file is the `<compiler>` tag. This tag wraps most compiler options. However, not all compiler options are set in the `<compiler>` block of the configuration file.

Tags that you must wrap in the compiler block are prefixed by `compiler` in the help output (for example, `compiler.services`). If the option uses no dot-notation in the help output (for example, `include-file`), it is a tag at the root level of the configuration file, and the entry appears as follows:

```
<compiler>
...
</compiler>
<include-file>
  <name>logo.gif</name>
  <path>c:/images/logo/logo1.gif</path>
</include-file>
```

In some cases, options have multiple parent tags, as with the fonts options, such as `compiler.fonts.managers` and `compiler.fonts.languages.language`. Other options that require parent tags when added to a configuration file include the `frames.frame` option and the metadata options. The following sections describe methods for determining the syntax.

Getting the configuration file tags

Use the `help list` option of the command-line compilers to get the configuration file syntax of the compiler options; for example:

```
mxmlc -help list advanced
```

The following is the entry for the `source-path` option:

```
-compiler.source-path [path-element][...]
```

This indicates that in the configuration file, you can have one or more `<path-element>` child tags of the `<source-path>` tag, and that `<source-path>` is a child of the `<compiler>` tag. The following example shows how this should appear in the configuration file:

```
<compiler>
  <source-path>
    <path-element>.</path-element>
    <path-element>c:/myclasses/</path-element>
  </source-path>
</compiler>
```

Understanding leaf nodes

The help output uses dot-notation to separate child tags from parent tags, with the right-most entry being known as the *leaf node*. For example, `-tag1.tag2` indicates that `<tag2>` should be a child tag of `<tag1>`.

Angle brackets (`< >`) or square brackets (`[ ]`) that surround an option indicate that the option is a leaf node.

Square brackets indicate that there can be a list of one or more parameters for that option.

If the leaf node of a tag in the angle bracket is unique, you do not have to specify the parent tags in the configuration file. For example, the help usage shows the following:

```
compiler.fonts.managers [manager-class][...]
```

You can specify the value of this option in the configuration file, as the following example shows:

```
<compiler>
  <fonts>
    <managers>
      <manager-class>flash.fonts.JREFontManager</manager-class>
    </managers>
  </fonts>
</compiler>
```

However, the `<manager-class>` leaf node is unique, so you can set the value without specifying the `<fonts>` and `<managers>` parent tags, as the following example shows:

```
<compiler>
  <manager-class>flash.fonts.JREFontManager</manager-class>
</compiler>
```

If the help output shows multiple options listed in angle brackets, you set the values of these options at the same level inside the configuration file and do not make them child tags of each other. For example, the usage for default-size (`default-size <width> <height>`) indicates that the default size of the application is set in a configuration file, as the following example shows:

```
<default-size>
  <height>height_value</height>
  <width>width_value</width>
</default-size>
```

Using tokens

You can pass custom token values to the compiler using the following syntax:

```
+token_name=value
```

In the configuration file, you reference that value using the following syntax:

```
${token_name}
```

You can use the `@Context` token in your configuration files to represent the context root of the application. You can also use the `${flexlib}` token to represent the frameworks directory. This is useful if you set up your own configuration and are not using the default `library-path` settings.

The default value of the `${flexlib}` token is `application_home\frameworks`.

Appending values

In a configuration file, you can specify the `append` attribute of any tag that takes a list of arguments. Set this attribute to `true` to indicate that the values should be appended to the option rather than replace it. The default value is `false`.

Setting the `append` attribute to `true` lets you compound the values of options with multiple configuration files. The following example appends two entries to the `library-path` option:

```
<library-path append="true">
  <path-element>/mylibs</path-element>
  <path-element>/myotherlibs</path-element>
</library-path>
```

About option precedence

You can set the same options in multiple places and the Flex compilers use the value from the source that has the highest precedence.

If you do not specify an option on the command line, the compilers check for a `load-config` option and get the value from that file.

When using the mxmlc compiler, Flex checks to see if there is an *app_name*-config.xml file in the same directory as the target MXML file. This is known as the local configuration file and is described in "Locating configuration files" on page 190. The syntax and structure of local configuration files are the same as with the flex-config.xml file.

If no `load-config` option is specified, the compilers check for the flex-config.xml file. The compilers look for the directory using the `application.home` environment variable. The following location is the default:

```
{application.home}/frameworks
```

Most options have a default value that the compilers use if the option is not set in any of the other ways.

The following table shows the possible sources of options for each compiler. The table also shows the order of precedence for each option. The options set using the method described in a lower row take precedence over the options set using the methods described in a higher row.

Compiler options	Flex Builder	Web-tier	mxmlc	compc
Default settings	Yes	No	No	No
flex-config.xml	No	Yes	Yes	Yes
Local configuration file	No	No	Yes	No
Configuration file specified by load-config option	Yes	No	Yes	Yes
Command-line option	No	No	Yes	Yes
Options panel	Yes	No	No	No

You can mix and match the source of the compiler options. For example, you can specify a custom configuration file with the `load-config` option, and also set additional options on the command line.

You can also use multiple configuration files. You can chain them by using the += operator with the `load-config` option. If you specify a configuration file with this option, the compilers also look for the flex-config.xml and local (*appname*-config.xml) configuration files.

Using the application compiler

The application compiler's options let you define settings such as the library path and whether to include debug information in the resulting SWF file. Also, you can set application-specific settings such as the frame rate at which the SWF file should play and its height and width.

In Flex Builder, you open the application compiler by building a new Flex Project. Some of the options described in this section have equivalents in the Flex Builder environment. For example, you can use the tabs in the Flex Build Path dialog box to add classes and libraries to your project.

This section includes a detailed description of the application compiler options and a set of examples that use the application compiler.

About the application compiler options

The following table describes the application compiler options:

Option	Description
`accessible=true|false`	Enables accessibility features when compiling the Flex application or SWC file. The default value is `false`. For more information on using the Flex accessibility features, see Chapter 36, "Creating Accessible Applications," in *Flex 2 Developer's Guide*.
`actionscript-file-encoding` *`string`*	Sets the file encoding for ActionScript files. For more information, see "Setting the file encoding" on page 212.
`advanced`	Lists advanced help options when used with the help option, as the following example shows: `mxmlc -help advanced` This is an advanced option.
`allow-source-path-overlap=true|false`	Checks if a `source-path` entry is a subdirectory of another `source-path` entry. It helps make the package names of MXML components unambiguous. This is an advanced option.
`as3=true|false`	Use the ActionScript 3.0 class-based object model for greater performance and better error reporting. In the class-based object model, most built-in functions are implemented as fixed methods of classes. The default value is `true`. If you set this value to `false`, you must set the `es` option to `true`. This is an advanced option.
`benchmark=true|false`	Prints detailed compile times to the standard output. The default value is `true`.

Option	Description
`context-root` *`context-path`*	Sets the value of the `{context.root}` token, which is often used in channel definitions in the flex-services.xml file and other settings in the flex-config.xml file. The default value is null. For more information on using the `{context.root}` token, see "Changing the context root" on page 176.
`contributor` *`name`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.
`creator` *`name`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.
`date` *`text`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.
`debug=true|false`	Generates a debug SWF file. This file includes line numbers and filenames of all the source files. When a run-time error occurs, the stacktrace shows these line numbers and filenames. This information is also used by the command-line debugger and the Flex Builder debugger. Enabling the `debug` option generates larger SWF files. For the mxmlc compiler, the default value is `false`. For the compc compiler, the default value is `true`. For SWC files generated with the compc compiler, set this value to `true`, unless the target SWC file is an RSL. In that case, set the `debug` option to `false`. For information about the command-line debugger, see Chapter 12, "Using the Command-Line Debugger," on page 269. If you set this option to `true`, Flex also sets the `verbose-stacktraces` option to `true`
`debug-password` *`string`*	Lets you engage in remote debugging sessions with the Flash IDE. This is an advanced option.

Option	Description
`default-background-color` *int*	Sets the application's background color. You use the 0x notation to set the color, as the following example shows: `-default-background-color=0xCCCCFF` The default value is null. The default background of a Flex application is an image of a gray gradient. You must override this image for the value of the `default-background-color` option to be visible. For more information, see "Editing application settings" on page 213. This is an advanced option.
`default-frame-rate` *int*	Sets the application's frame rate. The default value is 24. This is an advanced option.
`default-script-limits` *max-recursion-depth* *max-execution-time*	Defines the application's script execution limits. The `max-recursion-depth` value specifies the maximum depth of Adobe Flash Player call stack before Flash Player stops. This is essentially the stack overflow limit. The default value is 1000. The `max-execution-time` value specifies the maximum duration, in seconds, that an ActionScript event handler can execute before Flash Player assumes that it is hung, and aborts it. The default value is 60 seconds. You cannot set this value above 60 seconds. You can override these settings in the application. This is an advanced option.
`default-size` *width height*	Defines the default application size, in pixels. This is an advanced option.
`defaults-css-url` *string*	Defines the location of the default style sheet. Setting this option overrides the implicit use of the defaults.css style sheet in the framework.swc file. For more information on the defaults.css file, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*. This is an advanced option.

Option	Description
`description` *text*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.
`dump-config` *filename*	Outputs the compiler options in the flex-config.xml file to the target path; for example: `mxmlc -dump-config myapp-config.xml` This is an advanced option.
`es=true|false`	Instructs the compiler to use the ECMAScript edition 3 prototype-based object model to allow dynamic overriding of prototype properties. In the prototype-based object model, built-in functions are implemented as dynamic properties of prototype objects. The default value is `false`. Using the ECMAScript edition 3 prototype-based object model lets you use untyped properties and functions in your application code. As a result, if you set the value of the `es` compiler option to `true`, you must set the `strict` compiler option to `false`. Otherwise, the compiler will throw errors. If you set this option to `true`, you must also set the value of the `as3` compiler option to `false`. This is an advanced option.
`externs` *symbol [...]*	Sets a list of symbols to exclude from linking when compiling a SWF file. This option provides compile-time link checking for external references that are dynamically linked. For more information about dynamic linking, see "About linking" on page 234. This is an advanced option.
`external-library-path` *path-element [...]*	Specifies a list of SWC files or directories to exclude from linking when compiling a SWF file. This option provides compile-time link checking for external components that are dynamically linked. For more information about dynamic linking, see "About linking" on page 234. You can use the += operator to append the new SWC file to the list of external libraries.

Option	Description
`fonts.flash-type=true|false`	Sets the default value that determines whether embedded fonts use the FlashType rendering engine. Setting the value of the `flashType` property in a style sheet overrides this value. The default value is `false`. For more information about using FlashType, see Chapter 19, "Using Fonts," in *Flex 2 Developer's Guide*.
`fonts.languages.language-range` *`lang range`*	Specifies the range of Unicode settings for that language. For more information, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*. This is an advanced option.
`fonts.local-fonts-snapshot` *`path_to_file`*	Sets the location of the local font snapshot file. The file contains system font data. This is an advanced option.
`fonts.managers` *`manager-class [...]`*	Defines the font manager. The default is flash.fonts.JREFontManager. You can also use the flash.fonts.BatikFontManager. For more information, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*. This is an advanced option.
`fonts.max-cached-fonts` *`string`*	Sets the maximum number of fonts to keep in the server cache. For more information, see "Caching fonts and glyphs" on page 174. This is an advanced option.
`fonts.max-glyphs-per-face` *`string`*	Sets the maximum number of character glyph-outlines to keep in the server cache for each font face. For more information, see "Caching fonts and glyphs" on page 174. This is an advanced option.

Option	Description
`frames.frame` *`label class_name`* `[...]`	Specifies a SWF file frame label with a sequence of class names that are linked onto the frame. This option lets you add asset factories that stream in after the application that then publish their interfaces with the ModuleManager class. The advantage to doing this is that the application starts faster than it would have if the assets had been included in the code, but does not require moving the assets to an external SWF file. This is an advanced option.
`generate-frame-loader=true|false`	Toggles the generation of an IFlexBootstrap-derived loader class. This is an advanced option.
`headless-server=true|false`	Enables the headless implementation of the Flex compiler. This sets the following: `System.setProperty("java.awt.headless", "true")` The headless setting (java.awt.headless=true) is required to use fonts and SVG on UNIX systems without X Windows. This is an advanced option.
`help [-list [advanced]]`	Prints usage information to the standard output. For more information, see "Command-line syntax" on page 187.
`include-libraries` *`library [...]`*	Links all classes inside a SWC file to the resulting application SWF file, regardless of whether or not they are used. Contrast this option with the `library-path` option that includes only those classes that are referenced at compile time. To link one or more classes whether or not they are used and not an entire SWC file, use the `includes` option. This option is commonly used to specify resource bundles.

Option	Description
`includes class [...]`	Links one or more classes to the resulting application SWF file, whether or not those classes are required at compile time. To link an entire SWC file rather than individual classes, use the `include-libraries` option.
`incremental=true|false`	Enables incremental compilation. For more information, see "About incremental compilation" on page 214. This option is `true` by default for the Flex Builder application compiler. For the command-line compiler, the default is `false`. The web-tier compiler does not support incremental compilation.
`keep-as3-metadata=class_name [...]`	Specifies metadata that you want to keep. By default, the compiler keeps the following metadata: • Bindable • Managed • ChangeEvent • NonCommittingChangeEvent • Transient If you want to preserve the default metadata, you should use the += operator to append your new metadata, rather than the = operator which replaces the default metadata. This is an advanced option.

Option	Description
`keep-all-type-selectors=true|false`	Instructs the compiler to keep a style sheet's type selector in a SWF file, even if that type (the class) is not used in the application. This is useful when you have a modular application that loads other applications. For example, the loading SWF file might define a type selector for a type used in the loaded (or, target) SWF file. If you set this option to `true` when compiling the loading SWF file, then the target SWF file will have access to that type selector when it is loaded. If you set this option to `false`, the compiler will not include that type selector in the loading SWF file at compile time. As a result, the styles will not be available to the target SWF file. This is an advanced option.
`keep-generated-actionscript=true|false`	Determines whether to keep the generated ActionScript class files. The generated class files include stubs and classes that are generated by the compiler and used to build the SWF file. When using the application compiler, the default location of the files is the /generated subdirectory, which is directly below the target MXML file. If the /generated directory does not exist, the compiler creates one. When using the compc component compiler, the default location of the /generated directory is relative to the output of the SWC file. When using Flex Builder, the default location of the generated files is the /bin/generated directory. The default names of the primary generated class files are *filename*-generated.as and *filename*-interface.as. The default value is `false`. This is an advanced option.
`language` *`code`*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.

Option	Description
`library-path` *path-element [...]*	Links SWC files to the resulting application SWF file. The compiler only links in those classes for the SWC file that are required. The default value of the `library-path` option includes all SWC files in the libs directory and the current locale. These are required. To point to individual classes or packages rather than entire SWC files, use the `source-path` option. If you set the value of the `library-path` as an option of the command-line compiler, you must also explicitly add the framework.swc and locale SWC files. Your new entry is not appended to the `library-path` but replaces it, unless you use the += operator. On the command line, you use the += operator to append the new argument to the list of existing SWC files. In a configuration file, you can set the `append` attribute of the `library-path` tag to `true` to indicate that the values should be appended to the library path rather than replace existing default entries.
`license` *product_name license_key*	Defines the license key to use when compiling. Valid values for *product_name* include `fds`, `charting`, and `flexbuilder`.

`link-report` *filename*	Prints linking information to the specified output file. This file is an XML file that contains `<def>`, `<pre>`, and `<ext>` symbols showing linker dependencies in the final SWF file.
	The file format output by this command can be used to write a file for input to the `load-externs` option.
	For more information on the report, see "Examining linker dependencies" on page 98.
	This is an advanced option.

Option	Description
`load-config` *filename*	Specifies the location of the configuration file that defines compiler options. If you specify a configuration file, you can override individual options by setting them on the command line. All relative paths in the configuration file are relative to the location of the configuration file itself. Use the += operator to chain this configuration file to other configuration files. For more information on using configuration files to provide options to the command-line compilers, see "About configuration files" on page 190.

`load-externs` *filename [...]*	Specifies the location of an XML file that contains `<def>`, `<pre>`, and `<ext>` symbols to omit from linking when compiling a SWF file. The XML file uses the same syntax as the one produced by the `link-report` option. For more information on the report, see "Examining linker dependencies" on page 98.
	This option provides compile-time link checking for external components that are dynamically linked.
	For more information about dynamic linking, see "About linking" on page 234.
	This is an advanced option.

`locale` *string*	Specifies the locale that should be packaged in the SWF file (for example, en_EN). You run the mxmlc compiler multiple times to create SWF files for more than one locale, with only the `locale` option changing. You must also include the parent directory of the individual locale directories, plus the token `{locale}`, in the `source-path`; for example: `mxmlc -locale en_EN -source-path locale/ {locale} MainApp.mxml` For more information, see Chapter 25, "Localizing Flex Applications,"in *Flex 2 Developer's Guide*.

Option	Description
`localized-description` *text lang*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.
`localized-title` *text lang*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.
`namespaces.namespace` *uri manifest*	Specifies a namespace for the MXML file. You must include a URI and the location of the manifest file that defines the contents of this namespace. This path is relative to the MXML file. For more information about manifest files, see "About manifest files" on page 231.
`optimize=true|false`	Enables the ActionScript optimizer. This optimizer reduces file size and increases performance by optimizing the SWF file's bytecode. The default value is `false`.
`output` *filename*	Specifies the output path and filename for the resulting file. If you omit this option, the compiler saves the SWF file to the directory where the target file is located. The default SWF filename matches the target filename, but with a SWF file extension. If you use a relative path to define the *filename*, it is always relative to the current working directory, not the target MXML application root. The compiler creates extra directories based on the specified filename if those directories are not present. When using this option with the component compiler, the output is a SWC file rather than a SWF file.
`publisher` *name*	Sets metadata in the resulting SWF file. For more information, see "Adding metadata to SWF files" on page 211.

Option	Description
`raw-metadata` *`XML_string`*	Defines the metadata for the resulting SWF file. The value of this option overrides any metadata.* compiler options (such as `contributor`, `creator`, `date`, and `description`). This is an advanced option.
`resource-bundle-list` *`filename`*	Prints a list of resource bundles to input to the compc compiler to create a resource bundle SWC file. The *`filename`* argument is the name of the file that contains the list of bundles. For more information, see Chapter 25, "Localizing Flex Applications," in *Flex 2 Developer's Guide*.
`runtime-shared-libraries` *`url [...]`*	Specifies a list of run-time shared libraries (RSLs) to use for this application. RSLs are dynamically-linked at run time. You specify the location of the SWF file relative to the deployment location of the application. For example, if you store a file named library.swf file in the *web_root*/libraries directory on the web server, and the application in the web root, you specify libraries/library.swf. For more information about RSLs, see Chapter 10, "Using Runtime Shared Libraries," on page 233.
`services` *`filename`*	Specifies the location of the services-config.xml file. This file is used by Flex Data Services.
`show-binding-warnings=true|false`	Shows a warning when Flash Player cannot detect changes to a bound property. The default value is `true`. For more information about compiler warnings, see "Using SWC files" on page 213.
`show-actionscript-warnings=true|false`	Shows warnings for ActionScript classes. The default value is `true`. For more information about viewing warnings and errors, see "Viewing warnings and errors" on page 227.

Option	Description
`show-deprecation-warnings=true|false`	Shows deprecation warnings for Flex components. To see warnings for ActionScript classes, use the `show-actionscript-warnings` option. The default value is `true`. For more information about viewing warnings and errors, see “Viewing warnings and errors” on page 227.
`show-unused-type-selector-warnings=true|false`	Shows warnings when a type selector in a style sheet or `<mx:Style>` block is not used by any components in the application.
`source-path` *`path-element [...]`*	Adds directories or files to the source path. The Flex compiler searches directories in the source path for MXML or AS source files that are used in your Flex applications and includes those that are required at compile time. You can use wildcards to include all files and subdirectories of a directory. To link an entire library SWC file and not individual classes or directories, use the `library-path` option. The source path is also used as the search path for the component compiler's `include-classes` and `include-resource-bundles` options. You can also use the += operator to append the new argument to the list of existing source path entries. This option has the following default behavior: • If `source-path` is empty, the target file's directory will be added to source-path. • If `source-path` is not empty and if the target file's directory is a subdirectory of one of the directories in `source-path`, `source-path` remains unchanged. • If `source-path` is not empty and if the target file's directory is not a subdirectory of any one of the directories in `source-path`, the target file's directory is pre-pended to `source-path`.

Option	Description
`strict=true|false`	Prints undefined property and function calls; also performs compile-time type checking on assignments and options supplied to method calls. The default value is `true`. For more information about viewing warnings and errors, see “Viewing warnings and errors” on page 227.
`theme` *`filename [...]`*	Specifies a list of theme files to use with this application. Theme files can be SWC files with CSS files inside them or CSS files. For information on compiling a SWC theme file, see Chapter 18, “Using Styles and Themes,” in *Flex 2 Developer’s Guide*.
`title` *`text`*	Sets metadata in the resulting SWF file. For more information, see “Adding metadata to SWF files” on page 211.
`use-network=true|false`	Specifies that the current application uses network services. The default value is `true`. When the `use-network` property is set to `false`, the application can access the local filesystem (for example, use the `XML.load()` method with *file:* URLs) but not network services. In most circumstances, the value of this property should be `true`. For more information about the `use-network` property, see Chapter 4, “Applying Flex Security,” on page 51.
`use-resource-bundle-metadata=true|false`	Enables resource bundles. Set to `true` to instruct the compiler to process the contents of the `[ResourceBundle]` metadata tag. The default value is `true`. For more information, see Chapter 25, “Localizing Flex Applications,” in *Flex 2 Developer’s Guide*. This is an advanced option.

Option	Description
`verbose-stacktraces=true|false`	Generates source code that includes line numbers. When a run-time error occurs, the stacktrace shows these line numbers. Enabling this option generates larger SWF files. Enabling this option does not generate a debug SWF file. To do that, you must set the `debug` option to `true`. The default value is `false`.
`version`	Returns the version number of the MXML compiler. If you are using a trial or Beta version of Flex, the version option also returns the number of days remaining in the trial period and the expiration date.
`warn-`*`warning_type`*`=true|false`	Enables specified warnings. For more information, see "Viewing warnings and errors" on page 227.
`warnings=true|false`	Enables all warnings. Set to `false` to disable all warnings. This option overrides the `warn-`*`warning_type`* options. The default value is `true`.

The following sections provide examples of using the mxmlc application compiler options on the command line. You can also use these techniques with the application compilers in the Flex Builder and web-tier environments.

Basic example

The most basic example is one in which the MXML file has no external dependencies (such as components in a SWC file or ActionScript classes) and no special options. In this case, you invoke the mxmlc compiler and point it to your MXML file as the following example shows:

```
mxmlc c:/myfiles/app.mxml
```

The default option is the target file to compile into a SWF file, and it is required to have a value. If you use a space-separated list as part of the options, you can terminate the list with a double hyphen before adding the target file; for example:

```
mxmlc -option arg1 arg2 arg3 -- target_file.mxml
```

Adding metadata to SWF files

The application compilers support adding metadata to SWF files. This metadata can be used by search engines and other utilities to gather information about the SWF file. This metadata represents a subset of the Dublin Core schema.

You can set the following values:

- `contributor`
- `creator`
- `date`
- `description`
- `language`
- `localized-description`
- `localized-title`
- `publisher`
- `title`

For the mxmlc command-line compiler and the web-tier compiler, the default metadata settings in the flex-config.xml file are as follows:

```
<metadata>
  <title>Adobe Flex 2 Application</title>
  <description>http://www.adobe.com/flex</description>
  <publisher>unknown</publisher>
  <creator>unknown</creator>
  <language>EN</language>
</metadata>
```

You can also set the metadata values as command-line options. The following example sets some of the metadata values:

```
mxmlc -language+=klingon -title "checkintest!" -localized-description "it
  r0x0rs" en-us -localized-description "c'est magnifique!" fr-fr
  -creator "Flexy Frank" -publisher "Franks Beans" flexstore.mxml
```

In this example, the following values are compiled into the resulting SWF file:

```
<rdf:RDF xmlns:rdf='http://www.w3.org/1999/02/22-rdf-syntax-ns#'>
  <rdf:Description rdf:about='' xmlns:dc='http://purl.org/dc/elements/
  1.1'>
    <dc:format>application/x-shockwave-flash</dc:format>
    <dc:title>checkintest!</dc:title>
    <dc:description>
      <rdf:Alt>
        <rdf:li xml:lang='fr-fr'>c'est magnifique!</rdf:li>
        <rdf:li xml:lang='x-default'>http://www.adobe.com/flex<
          /rdf:li>
        <rdf:li xml:lang='en-us'>it r0x0rs</rdf:li>
      </rdf:Alt>
    </dc:description>
    <dc:publisher>Franks Beans</dc:publisher>
    <dc:creator>Flexy Frank</dc:creator>
    <dc:language>EN</dc:language>
    <dc:language>klingon</dc:language>
    <dc:date>Dec 16, 2005</dc:date>
  </rdf:Description>
</rdf:RDF>
```

For information on the SWF file format, see the Flash File Format (SWF) Specification, which is available through the Player Licensing program.

Setting the file encoding

You use the `actionscript-file-encoding` option to set the file encoding so that the application compiler correctly interprets ActionScript files. This tag does not affect MXML files because they are XML files that contain an encoding specification in the `xml` tag.

You use the `actionscript-file-encoding` option when your ActionScript files do not contain a Byte Order Mark (BOM), and the files use an encoding that is different from the default encoding of your computer. If your ActionScript files contain a BOM, the compiler uses the information in the BOM to determine the file encoding.

For example, if your ActionScript files use Shift_JIS encoding, have no BOM, and your computer uses ISO-8859-1 as the default encoding, you use the `actionscript-file-encoding` option, as the following example shows:

```
actionscript-file-encoding=Shitf_JIS
```

Editing application settings

The mxmlc compiler includes options to set the application's frame rate, size, script limits, and background color. By setting them when you compile your application, you do not need to edit the HTML wrapper or the application's MXML file. You can override these settings by using properties of the `<mx:Application>` tag or properties of the `<object>` and `<embed>` tags in the HTML wrapper.

The following command-line example sets default application properties:

```
mxmlc -default-size 240 240 -default-frame-rate=24
  -default-background-color=0xCCCCFF -default-script-limits 5000 10
  -- c:/myfiles/flex2/misc/MainApp.mxml
```

To successfully apply the value of the `default-background-color` option to your Flex application, you must set the default background *image* to an empty string. Otherwise, this image of a gray gradient covers the background color. For more information, see Chapter 14, "Using the Application Container," in *Flex 2 Developer's Guide*.

For more information about the HTML wrapper, see Chapter 16, "Creating a Wrapper," on page 367.

Using SWC files

Often, you use SWC files when compiling MXML files. SWC files can provide themes, components, or other helper files. You typically specify SWC files used by the application by using the `library-path` option.

The following example compiles the RotationApplication.mxml file into the RotationApplication.swf file:

```
mxmlc -library-path+=c:/mylibraries/MyButtonSwc.swc
  c:/myfiles/comptest/testRotation.mxml
```

In a configuration file, this appears as the following example shows:

```
<compiler>
  <library-path>
    <path-element>c:/flexdeploy/frameworks/libs/framework.swc</path-
  element>
    <path-element>c:/flexdeploy/frameworks/locale/{locale}
      /framework_rb.swc</path-element>
    <path-element>c:/mylibraries/MyButtonSwc.swc</path-element>
  </library-path>
</compiler>
```

About incremental compilation

You can use incremental compilation to decrease the time it takes to compile an application or component library with the Flex application compilers. When incremental compilation is enabled, the compiler inspects changes to the bytecode between revisions and only recompiles the section of bytecode that has changed. These sections of bytecode are also referred to as *compilation units*.

You enable incremental compilation by setting the `incremental` option to `true`, as the following example shows:

```
mxmlc -incremental=true MyApp.mxml
```

Incremental compilation means that the compiler inspects your code, determines which parts of the application are affected by your changes, and only recompiles the newer classes and assets. The Flex compilers generate many compilation units that do not change between compilation cycles. It is possible that when you change one part of your application, the change might not have any effect on the bytecode of another.

As part of the incremental compilation process, the compiler generates a cache file that lists the compilation units of your application and information on your application's structure. This file is located in the same directory as the file that you are compiling. For example, if my application is called MyApp.mxml, the cache file is called MyApp_*n*.cache, where *n* represents a checksum generated by the compiler based on compiler configuration. This file helps the compiler determine which parts of your application must be recompiled. One way to force a complete recompile is to delete the cache file from the directory.

Incremental compilation can help reduce compile time on small applications, but you achieve the biggest gains on larger applications.

The default value of the `incremental` compiler option is `true` for the Flex Builder application compiler. For the mxmlc command-line compiler, the default is `false`.

To enable incremental compilation for the web-tier compiler, you set the value of the `incremental-compile` option in the flex-webtier-config.xml file to `true`. The default is `true`. The `incremental` option in the flex-config.xml file has no effect when you use the web-tier compiler. If production mode is enabled, disable incremental compilation because the web-tier compiler does not recompile pages unless the server is restarted.

You can use the `recompile` query string parameter to override the `incremental-compile` option. This query string parameter forces a full recompilation of all parts of the application.

For more information about query string compiler overrides, see "Compiler configuration" on page 168.

Using the component compiler

In Flex Builder, you open the component compiler by building a new Flex Library Project. Some of the options described in this section have equivalents in the Flex Builder environment. You use the tabs in the Flex Library Build Path dialog box to add classes, libraries, and other resources to the SWC file.

This section describes the component compiler options and includes examples that show component compiler usage.

About the component compiler options

The component compiler options let you define settings such as the classes, resources, and namespaces to include in the resulting SWC file.

The component compiler can take most of the application compiler options, and the options described in this section. For a description of the application compiler options, see "About the application compiler options" on page 196. Application compiler options that do not apply to the component compiler include the metadata options (such as `contributor`, `title`, and `date`), default application options (such as `default-background-color` and `default-frame-rate`), `locale`, `debug-password`, and `theme`.

The component compiler has compiler options that the application compilers do not have. The following table describes the component compiler options that are not used by the application compilers:

Option	Description
`directory`	Outputs the SWC file in an open directory format rather than a SWC file. You use this option with the `output` option to specify a destination directory, as the following example shows: `compc -directory -output destination_directory` You use this option when you create RSLs. For more information, see Chapter 10, "Using Runtime Shared Libraries," on page 233.
`include-classes class [...]`	Specifies classes to include in the SWC file. You provide the class name (for example, MyClass) rather than the file name (for example, MyClass.as) to the file for this option. As a result, all classes specified with this option must be in the compiler's source path. You specify this by using the `source-path` compiler option. You can use packaged and unpackaged classes. To use components in namespaces, use the `include-namespaces` option. If the components are in packages, ensure that you use dot-notation rather than slashes to separate package levels. This is the default option for the component compiler.
`include-file name path [...]`	Adds the file to the SWC file. This option does not embed files inside the library.swf file. This is useful for skinning and theming, where you want to add non-compiled files that can be referenced in a style sheet or embedded as assets in MXML files. If you use the `[Embed]` syntax to add a resource to your application, you are not required to use this option to also link it into the SWC file. For more information, see "Adding nonsource classes" on page 225.

Option	Description
`include-namespaces` *uri [...]*	Specifies namespace-style components in the SWC file. You specify a list of URIs to include in the SWC file. The *uri* argument must already be defined with the `namespace` option. To use components in packages, use the `include-classes` option.
`include-resource-bundles` *string [...]*	Specifies the resource bundles to include in this SWC file. All resource bundles specified with this option must be in the compiler's source path. You specify this using the `source-path` compiler option. For more information on using resource bundles, see Chapter 25, "Localizing Flex Applications," in *Flex 2 Developer's Guide*.
`include-sources` *path-element*	Specifies classes or directories to add to the SWC file. When specifying classes, you specify the path to the class file (for example, MyClass.as) rather than the class name itself (for example, MyClass). This lets you add classes to the SWC file that are not in the source path. In general, though, use the `include-classes` option, which lets you add classes that are in the source path. If you specify a directory, this option includes all files with an MXML or AS extension, and ignores all other files.

On the command line, you cannot point the compc utility to a single directory and have it compile all components and source files in that directory. You must specify each source file to compile.

If you have a large set of components in a namespace to include in a SWC file, you can use a manifest file to avoid having to type an unwieldy compc command. For information on creating manifest files, see "About manifest files" on page 231.

The following sections describe common scenarios where you could use the compc command-line compiler. You can apply the techniques described here to compiling SWC files in Flex Builder with the Flex Library Compiler.

Compiling stand-alone components and classes

In many cases, you have one or more components that you use in your Flex applications, but you do not have them in a package structure. You want to be able to use them in the generic namespace ("*") inside your Flex applications. In these cases, you use the `include-classes` option to add the components to your SWC file.

The following command-line example compiles two MXML components, Rotation.as and RotationInstance.as, into a single SWC file:

```
compc -source-path . -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/
  user_classes/RotationClasses.swc -include-classes
  rotationClasses.Rotation rotationClasses.RotationInstance
```

The rotationClasses directory is a subdirectory of the current directory, which is in the source path. The SWC file is output to the user_classes directory, so the new components require no additional configuration to be used in a server environment.

You use the `include-classes` option to add components to the SWC file. You use just the class name of the component and not the full filename (for example, MyComponent rather than MyComponent.as). Use dot-notation to specify the location of the component in the package structure.

You also set the `source-path` to the current directory or a directory from which the component directory can be determined.

You can also add the framework.swc and framework_rb.swc files to the `library-path` option. This addition is not always required if the compiler can determine the location of these SWC files on its own. However, if you move the compiler utility out of the default location relative to the frameworks files, you must add it to the library path.

The previous command-line example appears in a configuration file as follows:

```
<compiler>
  <source-path>
    <path-element>.</path-element>
  </source-path>
  <output>c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
    RotationClasses.swc</output>
</compiler>
<include-classes>
  <class>rotationClasses.Rotation</class>
  <class>rotationClasses.RotationInstance</class>
<include-classes>
```

To use components that are not in a package in a Flex application, you must declare a namespace that includes the directory structure of the components. The following example declares a namespace for the components compiled in the previous example:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmln:local="rotationclasses.*">
  ...
  <local:Rotation id="Rotate75" angleFrom="0" angleTo="75" duration="100"/>
  ...
</mx:Application>
```

To use the generic namespace of "*" rather than a namespace that includes a component's directory structure, you can include the directory in the `source-path` as the following command-line example shows:

```
compc -source-path . c:/flexdeploy/comps/rotationClasses
  -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
  RotationComps.swc -include-classes Rotation RotationInstance
```

Then, you can specify the namespace in your application as:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmln:local="*">
```

You are not required to use the directory name in the `include-classes` option if you add the directory to the source path.

These options appear in a configuration file, as the following example shows:

```
<compiler>
  <source-path>
    <path-element>.</path-element>
    <path-element>c:/flexdeploy/comps/rotationClasses</path-element>
  </source-path>
  <output>c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
  RotationComps.swc</output>
</compiler>
<include-classes>
  <class>Rotation</class>
  <class>RotationInstance</class>
<include-classes>
```

This example assumes that the components are not in a named package. For information about compiling packaged components, see "Compiling components in packages" on page 220.

Compiling components in packages

Some components are created inside packages or directory structures so that they can be logically grouped and separated from application code. As a result, packaged components can have a namespace declaration that includes the package name or a unique namespace identifier that references their location within a package.

You compile packaged components similarly to how you compile components that are not in packages. The only difference is that you must use the package name in the namespace declaration, regardless of how you compiled the SWC file, and that package name uses dot-notation instead of slashes. You must be sure to specify the location of the classes in the `source-path`.

In the following command-line example, the MyButton component is in the mypackage package:

```
compc -source-path . c:/flexdeploy/comps/mypackage/
  -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
  MyButtonComp.swc -include-classes mypackage.MyButton
```

These options appear in a configuration file, as the following example shows:

```
<compiler>
  <source-path>
    <path-element>.</path-element>
    <path-element>c:/flexdeploy/comps/mypackage/</path-element>
  </source-path>
  <output>c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
    MyButtonComp.swc</output>
</compiler>
<include-classes>
  <class>mypackage.MyButton</class>
<include-classes>
```

To access the MyButton class in your application, you must declare a namespace that includes its package; for example:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:mine="mypackage.*">
```

You can use the compc compiler to compile components from multiple packages into a single SWC file. In the following command-line example, the MyButton control is in the mypackage package, and the CustomComboBox control is in the acme package:

```
compc -source-path . -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/
  user_classes/CustomComps.swc -include-classes mypackage.MyButton
  acme.CustomComboBox
```

You then define each package as a separate namespace in your MXML application:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:mine="mypackage.*" xmlns:acme="acme.*">
  <mine:MyButton/>
  <acme:CustomComboBox/>
</mx:Application>
```

Compiling components using namespaces

When you have many components in one or more packages that you want to add to a SWC file and want to reference from an MXML file through a custom namespace, you can list them in a manifest file, then reference that manifest file on the command line. Also, you can specify a namespace for that component or define multiple manifest files and, therefore, specify multiple namespaces to compile into a single SWC file.

When you use manifest files to define the components in your SWC file, you specify the namespace that the components use in your Flex applications. You can compile all the components from one or more packages into a single SWC file. If you have more than one package, you can set it up so that all packages use a single namespace or so that each package has an individual namespace.

Components in a single namespace

In the manifest file, you define which components are in a namespace. The following sample manifest file defines two components to be included in the namespace:

```
<?xml version="1.0"?>
<!-- SimpleManifest.xml -->
<componentPackage>
  <component id="MyButton" class="MyButton"/>
  <component id="MyOtherButton" class="MyOtherButton"/>
</componentPackage>
```

The manifest file can contain references to any number of components in a namespace. The `class` option is the full class name (including package) of the class. The `id` property is optional, but you can use it to define the MXML tag interface that you use in your Flex applications. If the compiler cannot find one or more files listed in the manifest, it throws an error. For more information on using manifest files, see "About manifest files" on page 231.

On the command line, you define the namespace with the `namespace` option; for example:

```
-namespace http://mynamespace SimpleManifest.xml
```

Next, you target the defined namespace for inclusion in the SWC file with the `include-namespaces` option; for example:

```
-include-namespaces http://mynamespace
```

The `namespace` option matches a namespace (such as "http://www.adobe.com/2006/mxml") with a manifest file. The `include-namespaces` option instructs compc to include all the components listed in that namespace's manifest file in the SWC file.

After you define the manifest file, you can compile the SWC file. The following command-line example compiles the components into the "http://mynamespace" namespace:

```
compc -source-path . -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/
  user_classes/MyButtons.swc -namespace http://mynamespace
  SimpleManifest.xml -include-namespaces http://mynamespace
```

In a configuration file, these options appear as the following example shows:

```
<compiler>
  <source-path>
    <path-element>.</path-element>
  </source-path>
  <output>c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
    MyButtons.swc</output>
  <namespaces>
    <namespace>
      <uri>http://mynamespace</uri>
      <manifest>SimpleManifest.xml</manifest>
    </namespace>
  </namespaces>
</compiler>
<include-namespaces>
  <uri>http://mynamespace</uri>
<include-namespaces>
```

In your Flex application, you can access the components by defining the new namespace in the `<mx:Application>` tag, as the following example shows:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:a="http://
  mynamespace">
  <a:MyButton/>
  <a:MyOtherButton/>
</mx:Application>
```

Components in multiple namespaces

You can use the compc compiler to compile components that use multiple namespaces into a SWC file. Each namespace must have its own manifest file. The following command-line example compiles components defined in the AcmeManifest.xml and SimpleManifest.xml manifest files:

```
compc -source-path . -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/
  user_classes/MyButtons.swc -namespace http://acme2006
  AcmeManifest.xml -namespace http://mynamespace SimpleManifest.xml
  -include-namespaces http://acme2006 http://mynamespace
```

In this case, all components in both the http://mynamespace and http://acme2006 namespaces are targeted and included in the output SWC file.

In a configuration file, these options appear as the following example shows:

```
<compiler>
  <source-path>
    <path-element>.</path-element>
  </source-path>
  <output>c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
    MyButtons.swc</output>
  <namespaces>
    <namespace>
      <uri>http://acme2006</uri>
      <manifest>AcmeManifest.xml</manifest>
    </namespace>
    <namespace>
      <uri>http://mynamespace</uri>
      <manifest>SimpleManifest.xml</manifest>
    </namespace>
  </namespaces>
</compiler>
<include-namespaces>
  <uri>http://acme2006</uri>
  <uri>http://mynamespace</uri>
<include-namespaces>
```

In your MXML application, you define both namespaces separately:

```
<?xml version="1.0"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
  xmlns:simple="http://mynamespace" xmlns:acme="http://acme2006">
  <simple:SimpleComponent/>
  <acme:AcmeComponent/>
</mx:Application>
```

You are not required to include all namespaces that you define as target namespaces. You can define multiple namespaces, but use only one target namespace. You might do this if some components use other components that are not directly exposed as MXML tags. You cannot then directly access the components in the unused namespace, however.

The following command line example defines two namespaces, http://acme2006 and http://mynamespace, but only includes one as a namespace target:

```
compc -source-path . -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/
  user_classes/MyButtons.swc -namespace http://acme2006
  AcmeManifest.xml -namespace http://mynamespace SimpleManifest.xml
  -include-namespaces http://mynamespace
```

Adding utility classes

You can add any classes that you want to use in your Flex applications to a SWC file. These classes do not have to be components, but are often files that components use. They are classes that might be used at run time and, therefore, are not checked by the compiler. For example, your components might use a library of classes that perform mathematical functions, or use a custom logging utility. This documentation refers to these classes as *utility classes*. Utility classes are not exposed as MXML tags.

To add utility classes to a SWC file, you use the `include-sources` option. This option lets you specify a path to a class file rather than the class name, or specify an entire directory of classes.

The following command-line example adds the FV_calc.as and FV_format.as utility classes to the SWC file:

```
compc -source-path . -output c:/jrun4/servers/flex2/flex/WEB-INF/flex/
  user_classes/MySwc.swc -include-sources FV_classes/FV_format.as
  FV_classes/FV_calc.as -include-classes asbutton.MyButton
```

In a configuration file, these options appear as the following example shows:

```
<compiler>
  <source-path>
    <path-element>.</path-element>
  </source-path>
  <output>c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
    MySwc.swc</output>
</compiler>
<include-classes>
  <class>asbutton.MyButton</class>
</include-classes>
<include-sources>
  <path-element>FV_classes/FV_format.as</path-element>
  <path-element>FV_classes/FV_calc.as</path-element>
<include-sources>
```

When specifying files with the `include-sources` option, you must give the full filename (for example, FV_calc.as instead of FV_calc) because the file is not a component.

You can also provide a directory name to the `include-sources` option. In this case, the compiler includes all files with an MXML or AS extension, and ignores all other files.

Classes that you add with the `include-sources` option can be accessed from the generic namespace in your Flex applications. To use them, you need to add the following code in your Flex application tag:

```
xmlns:local="*"
```

You can then use them as tags; for example:

```
<local:FV_calc id="calc" rate=".0125" nper="12" pmt="100" pv="0" type="1"/>
```

Adding nonsource classes

You often include noncompiled (or nonsource) files with your applications. A *nonsource* file is a class or resource (such as a style sheet or graphic) that is not compiled but is included in the SWC file for other classes to use. For example, a font file that you embed or a set of images that you use as graphical skins in a component's style sheet should not be compiled but should be included in the SWC file. These are classes that you typically do not use the `[Embed]` syntax to link in to your application.

Use the `include-file` option to define nonsource files in a SWC file.

The syntax for the `include-file` option is as follows:

```
-include-file name path
```

The *`name`* argument is the name used to reference the embedded file in your Flex applications. The *`path`* argument is the current path to the file in the file system.

When you use the `include-file` option, you specify both a name and a filepath, as the following example shows:

```
compc -include-file logo.gif c:/images/logo/logo1.gif ...
```

In a configuration file, these options appear as the following example shows:

```
<compiler>
  <output>c:/jrun4/servers/flex2/flex/WEB-INF/flex/user_classes/
    Combo.swc</output>
</compiler>
<include-file>
  <name>logo.gif</name>
  <path>c:/images/logo/logo1.gif</path>
</include-file>
<include-classes>
  <class>asbutton.MyButton</class>
<include-classes>
```

Each name that you assign to a resource must be unique because the name becomes a global variable.

You cannot specify a list of files with the `include-file` option. So, you must add a separate `include-file` option for each file that you include, as the following command-line example shows:

```
compc -include-file file1.jpg ../images/file1.jpg
  -include-file file2.jpg ../images/file2.jpg -- -output MyFile.swc
```

If you want to add many resources to the SWC file, consider using a configuration file rather than listing all the resources on the command line. For an example of a configuration file that includes multiple resources in a SWC file, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*.

In general, specify a file extension for files that you include with the `include-file` option. In some cases, omitting the file extension can lead to a loss of functionality. For example, if you include a CSS file in a theme SWC file, you must set the name to be *.css. When Flex examines the SWC file, it applies all CSS files in that SWC file to the application. CSS files without the CSS extension are ignored.

Creating themes

You can use the `include-file` and `include-classes` options to add skin files and style sheets to a SWC file. The SWC file can then be used as a theme. For more information about using themes in Flex applications, see Chapter 18, "Using Styles and Themes," in *Flex 2 Developer's Guide*.

Viewing errors and warnings

You can use the compiler options to specify what level of warnings and errors to view. Also, you can set levels of logging with the compiler options. This section describes these techniques.

Viewing warnings and errors

There are several options that let you customize the level of warnings and errors that are displayed by the Flex compilers, including the following:

- `show-binding-warnings`
- `show-actionscript-warnings`
- `show-deprecation-warnings`
- `strict`
- `warnings`

To disable all warnings, set the `warnings` option to `false`.

The `show-actionscript-warnings` option displays compiler warnings for the following situations:

- Situations that are probably not what the developer intended, but are still legal; for example:

```
if (a = 10)      // Did you really want '==' instead of '='?
if (b == NaN)    // Any comparison with NaN is always false.
var b;           // Missing type declaration.
```

- Usage of deprecated or removed ActionScript 2.0 APIs.
- Situations where APIs behave differently in ActionScript 2.0 than in ActionScript 3.0.

You can customize the types of warnings displayed by using options that begin with *warn* (for example, `warn-constructor-return-values` and `warn-bad-type-cast`). A complete list of warnings are available in the advanced command-line help or in the flex-config.xml file.

The `strict` option enforces typing and reports run-time verifier errors at compile time. This option assumes that definitions are not dynamically redefined at run time, so these checks can be made at compile time. It displays errors for conditions such as undefined references, const and private violations, argument mismatches, and type checking.

The `show-deprecation-warnings` and `show-binding-warnings` options display warnings when you use deprecated APIs and when Flash Player cannot detect changes to bound properties, respectively.

About deprecation

The command-line compilers express deprecation warnings by default. In some cases, Flex functionality has been deprecated. Deprecated features and properties have the following characteristics:

- Generate compilation warnings that Flex displays in the HTML wrapper for the application.
- Continue to work in Flex 2.
- Will be removed from the product in a future major release.

You can suppress deprecation warnings by setting the `show-deprecated-warnings` option to `false`.

About logging

Errors and warnings are reported differently, depending on which compiler you are using.

The mxmlc and compc command-line compilers send error and warning messages to the standard output. You can redirect this output by using the redirector (>).

Flex Builder displays error and warning messages in the Problems tab.

The web-tier compiler displays error and warning messages in the requesting browser by default. The web-tier compiler also stores error and warning messages in a log file. You configure the location of this log file in the flex-webtier-config.xml file. You can enable or disable file logging, and set the size, log level, and location of the log file.

The following example shows the default logging settings for the web-tier compiler:

```
<logging>
  <level>info</level>
  <console>
    <enable>true</enable>
  </console>
  <file>
    <enable>true</enable>
    <file-name>/WEB-INF/flex/logs/flex.log</file-name>
    <maximum-size>200KB</maximum-size>
    <maximum-backups>3</maximum-backups>
  </file>
</logging>
```

For more information on configuring logging for the web-tier compiler, see "Web-tier logging" on page 265.

About SWC files

A *SWC* file is an archive file for Flex components and other assets. SWC files contain a SWF file and a catalog.xml file. The SWF file implements the compiled component or group of components and includes embedded resources as symbols. Flex applications extract the SWF file from a SWC file and use the SWF file's contents when the application refers to resources in that SWC file. The catalog.xml file lists of the contents of the component package and its individual components.

In most cases, the symbols defined in the SWF file in the SWC file that are referenced by the application are embedded in the Flex application at compile-time. This is known as static linking. Dynamic linking is when the SWF file is loaded at run time. To achieve dynamic linking of the SWF file, you must use the SWC file as a runtime shared library, or RSL. For more information, see Chapter 10, "Using Runtime Shared Libraries," on page 233.

SWC files make it easy to exchange components and other assets among Flex developers. You need only exchange a single file, rather than the MXML or ActionScript files and images and other resource files. The SWF file in a SWC file is compiled, which means that the code is loaded efficiently and it is hidden from casual view. Also, compiling a component as a SWC file can make namespace allocation an easier process.

You can package and expand SWC files with tools that support the PKZip archive format, such as WinZip or jar. However, do not manually change the contents of a SWC file, and do not try to run the SWF file that is in a SWC file in Flash Player.

An application running on a Flex Data Services server caches the contents of SWC files in the user_classes directory. If you change a SWC file in the user_classes directory, the Flex web-tier compiler does not automatically recompile the application. You must restart the application server so that Flex will load the new SWC file.

You can also save SWC files as open directories rather than archived files. This give you easier access to the contents of the SWC file. You create an open directory SWC with the `directory` option of the compc compiler. This is typically only used when you create an RSL.

When you use the component compiler to create a SWC file, you can include any number of components. When you use components from that SWC file in your MXML applications, the application compiler only includes those components that are used by your application, and dependent classes, in the final SWF file.

About included SWC files

Flex includes SWC files such as framework.swc, framework_rb.swc, playerglobal.swc, charts.swc, fds.swc, and rpc.swc. Flex 2 SDK includes the SWC files in the *flex_install_dir*/ frameworks/libs directory. Flex Builder includes the SWC files in the *flex_builder_install_dir*/ Flex SDK 2/frameworks/libs directory. Flex Data Services includes the SWC files in the *flex_webapp_root*/WEB-INF/flex/libs directory.

The following table describes the included SWC files:

SWC file	Description
charts.swc	Contains all the charting components. This is a separate product that you can add to your Flex installation. For more information, see Part 7, "Charting Components," in *Flex 2 Developer's Guide*.
charts_rb.swc	Contains the resource bundles for the Flex Charting components. For more information about resource bundles, see Chapter 25, "Localizing Flex Applications," in *Flex 2 Developer's Guide*.
fds_rb.swc	Contains the resource bundles for Flex Data Services classes.
fds.swc	Contains the class libraries for working with Flex Data Services. This file is only available for Flex Data Services developers. For more information, see Chapter 47, "Understanding Flex Messaging," in *Flex 2 Developer's Guide*.
flex.swc	Contains the Flex framework. Used for building ActionScript-only applications.
framework_rb.swc	Contains the resource bundles for the framework.
framework.swc	Contains all the built-in components of the Flex framework.
playerglobal.swc	Contains the Flash class libraries, such as flash.utils and flash.net, and the Flash primitives, such as String and Object.
rpc.swc	Contains the class libraries for working with RPC services, including SOAP-compliant web services, Adobe remote object services, and REST-style services. For more information, see Chapter 44, "Understanding RPC Components," in *Flex 2 Developer's Guide*.

Distributing SWC files

After you generate a SWC file, you can use it in your Flex applications. If you are running a Flex application on an application server with Flex Data Services, you copy the SWC file to the *flex_webapp_root*/WEB-INF/flex/user_classes directory.

You can also copy SWC files to a directory specified by the `library-path` compiler option. You must store SWC files at the top level of the user_classes directory or the directory specified by the `library-path`. You cannot store SWC files in subdirectories.

To use a SWC file when compiling components or applications from the command line or from within Flex Builder, you specify the location of the SWC file with the `library-path` compiler option.

NOTE: Do not store custom components or classes in the *flex_root*/WEB-INF/flex/libs directory. This directory is for Adobe classes and components.

Using components in SWC files

If a component in a SWC file does not have a namespace, you can add a generic namespace identifier in your `<mx:Application>` tag to use the component, as the following example shows:

```
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:a="*">
```

If the component has a package name as part of its namespace, you must do one of the following:

- Add the package name to the namespace declaration in the `<mx:Application>` tag; for example:

  ```
  <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
    xmlns:but="mycomponents.*">
  ```

- Create a manifest file and recompile the SWC file. You pass the manifest file to the compc compiler by using the `namespace` option. In the `<mx:Application>` tag, you specify only the unique namespace URI that you used with compc. For more information on specifying a namespace for the component, see "Compiling components using namespaces" on page 221.

About manifest files

Manifest files map a component namespace to class names. They define the package names that the components used before being compiled into a SWC file. They are not required when compiling SWC files, but they can help keep your source files organized.

Manifest files use the following syntax:

```
<?xml version="1.0"?>
<componentPackage>
  <component id="component_name" class="component_class"/>
  [...]
</componentPackage>
```

For example:

```
<?xml version="1.0"?>
<componentPackage>
  <component id="MyButton" class="package1.MyButton"/>
  <component id="MyOtherButton" class="package2.MyOtherButton"/>
</componentPackage>
```

In a manifest file, the `id` property of each `<component>` tag must be unique. It is the name you use for the tag in your Flex applications. For example, you define the `id` as `MyButton` in the manifest file:

```
<component id="MyButton" class="asbutton.MyButton"/>
```

In your Flex application, you use `MyButton` as the tag name:

```
<local:MyButton label="Click Me"/>
```

The `id` property in the manifest file entry is optional. If you omit it, you can use the class name as the tag. This is useful if you have two classes with the same name in different packages. In this case, you use the manifest to define the tags, as the following example shows:

```
<?xml version="1.0"?>
<componentPackage>
  <component id="BoringButton" class="boring.MyButton"/>
  <component id="GreatButton" class="great.MyButton"/>
</componentPackage>
```

Some SWC files consist of multiple components from different packages, so compc includes a manifest file with your SWC file in those cases to prevent compiler errors.

When compiling the SWC file, you specify the manifest file by using the `namespace` and the `include-namespaces` options. You define the namespace and its contents with the namespace option:

```
-namespace http://mynamespace mymanifest.xml
```

Then you identify that namespace's contents for inclusion in the SWC file:

```
-include-namespaces http://mynamespace
```

CHAPTER 10

Using Runtime Shared Libraries

10

Adobe Flex supports Runtime Shared Libraries (RSLs). This topic describes how to configure and use them to take advantage of their benefits.

Contents

About RSLs 233
Creating libraries 238
Using RSLs 240
RSL example 241

About RSLs

One way to reduce the size of your application's SWF file is by externalizing shared assets into stand-alone files that can be separately downloaded and cached on the client. These shared assets are loaded by multiple applications at run time, but must be transferred only once to the client. These shared files are known as *Runtime Shared Libraries* or *RSLs*.

If you have multiple applications but those applications share a core set of components or classes, the users have to download those assets only once as an RSL. The applications that share the assets in the RSL use the same cached RSL as the source for the libraries as long as they are in the same domain. The resulting file size for the applications can be reduced. The benefits increase as the number of applications that use the RSL increases. If you have only one application, RSLs do not reduce the aggregate download size, and might increase it.

Definition of RSLs

RSLs are libraries of components that are shared by applications in the same domain. You create an RSL by using either the Flex Builder's Build Project option for your Flex Library Project or the compc command-line compiler. After you compile the RSL, you can pass the library's location to the compiler when compiling your application.

You can benefit from the use of RSLs if you meet all of the following conditions:

- You host multiple applications in the same domain.
- You have custom component libraries.
- More than one application uses those custom component libraries.

Not all applications can benefit from RSLs.

The following is a list of typical applications that can benefit from RSLs:

- Large applications that load multiple smaller applications that use a common component library. The top-level application and all the subordinate applications can share components that are stored in a common RSL.
- A family of applications on a server built with a common component library. When the user accesses the first application, they download an application SWF file and the RSL. When they access the second application, they download only the application SWF file (the client has already downloaded the RSL, and the components in the RSL are shared between the two applications).
- A single monolithic application can benefit from RSLs if the application itself changes frequently, but it has a large set of components that rarely change. In this case, the components are downloaded once, while the application itself might be downloaded many times. This might be the case with the charts add-on to Flex Charting components, where you might have an application that uses the charts that you change frequently, but the charting components themselves remain fairly static.

About linking

Understanding linking can help you understand how RSLs work. The Flex compilers support static linking and dynamic linking. Static linking is the most common method of compiling a Flex application. However, dynamic linking lets you take advantage of RSLs to achieve some improvements on final SWF file size and, therefore, application download time.

When you use *static linking*, the compiler includes all referenced classes and their dependencies in the application SWF file when you compile the application. The end result is a large file that takes longer to download but loads and runs quickly because all the code is in the SWF file.

To compile your application with a library and to statically link its definitions into your application, you use the `library-path` and `include-libraries` options to specify locations of SWC files. When you use the `library-path` option, the compiler includes only those classes required at compile time in the SWF file. The `include-libraries` option includes the entire contents of the SWC file, regardless of which classes are required. You can also use the `source-path` and `includes` options to embed individual classes in your SWF file.

Dynamic linking is when some classes used by an application are left in an external file that is loaded at run time. The result is a smaller SWF file size for the main application, but the application relies on external files that are loaded during run time. RSLs use dynamic linking.

When you want to use dynamically link classes, you compile them into a library. You then instruct the compiler to exclude that library's contents from the application SWF file. You must provide link-checking at compile time even though the classes are not going to be included in the final SWF file.

To specify which files to dynamically link, you use the `external-library-path`, `externs`, or `load-externs` compiler options. These options instruct the compiler to exclude their arguments from inclusion in the application, but to check links against them and prepare to load them at run time. The `external-library-path` option specifies SWC files or directories for dynamic linking. The `externs` option specifies individual classes or symbols for dynamic linking. The `load-externs` option specifies an XML file that describes what classes to use for dynamic linking. This XML file has the same syntax as the file produced by the `link-report` compiler option. For more information about this report, see "Examining linker dependencies" on page 98.

The order in which you specify the external assets for RSLs is significant because the base classes must be loaded before the classes that use them.

You also use the `runtime-shared-libraries` option to specify the location of an RSL that contains the dynamically linked components. This RSL contains the matching set of definitions and is used at run time.

You can view the linking information for your application by using the `link-report` compiler option. For more information about the command-line compiler options, see Chapter 9, "Using the Flex Compilers," on page 179.

For more information on using these compiler options to use RSLs, see "Using RSLs" on page 240.

RSL considerations

RSLs are not necessarily beneficial for all applications. Try to test both the download time and startup time of your application with and without RSLs.

RSLs are not shared across domains. If a client runs an application in domain1 and uses an RSL, and then launches an application in domain2 that uses the same RSL, the client downloads the RSL twice.

An RSL usually increases the startup time of an application. This is because the entire library is loaded into a Flex application regardless of how much of the RSL is actually used. For this reason, make your RSLs as small as possible. This contrasts with how statically linked libraries are used. When you compile a Flex application, the compiler extracts just the components it needs from those component libraries. In general, libraries can be any size you want and only affect compilation time, not download time.

If you have several applications that share several components libraries, it might be tempting to merge the libraries into a single library that you use as an RSL. However, if the individual applications generally do not use more than one or two libraries each, the penalty for having to load the large RSL might be higher than it would be to have the applications incrementally load multiple smaller RSLs.

Test your application with both a single large RSL and multiple smaller RSLs, because the gains are largely application specific. It may be better to build one RSL that has some extra classes than to build two RSLs, if most users will load both of them anyway.

If you do have overlapping classes in multiple RSLs, be sure to synchronize the versions so that the wrong class is never loaded.

You cannot use RSLs in ActionScript-only projects if the base class is Sprite or MovieClip. RSLs require that the base class, such as Application or SimpleApplication, understand RSL loading.

About the framework.swc file

The Flex framework is a standard SWC file. By default, it is not used as an RSL. The entire framework.swc file is not linked into every application. The Flex compiler only links in the parts of the framework.swc file that the application uses. For example, if an application only uses Button, Panel, and TextArea controls, only the Button, Panel, and TextArea controls and their dependencies are linked by the compiler.

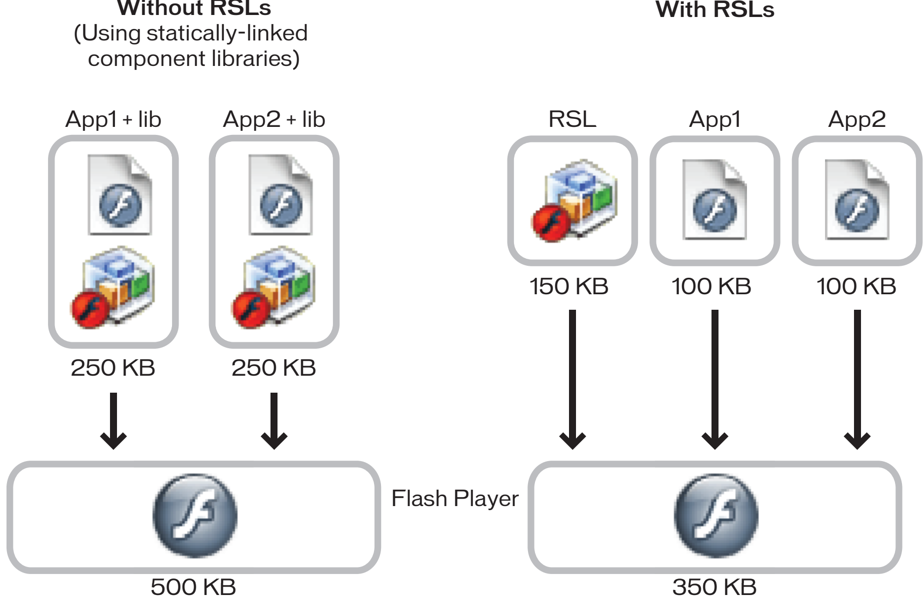
Most applications link in at least some of the contents of the framework.swc file, so you might wonder why the framework.swc file is not an RSL. The reason is that the classes in an RSL are always entirely linked in, regardless of which classes inside the RSL the application uses. If the RSL has many classes, and the application links in only a few of them, the compiler still links in all classes in the RSL into the application's SWF file.

If you used the framework.swc file as an RSL, all applications would include the entire framework. The effect of this would be to drastically increase startup time and increase the initial download size of applications. The framework.swc file is very large, and you would need many applications to offset the initial download size difference that would result from using it as an RSL.

RSL benefits

The following example shows the possible benefit of separating shared components into an RSL. In this example, the component library's size is 150 KB (kilobytes) and the compiled application's size is 100 KB. Without RSLs, you compile the component library into both applications for an aggregate download size of 500 KB. If you add a third or fourth application, the aggregate download size increases by 250 KB for each additional application.

With RSLs, the RSL needs to be downloaded once only. The result is an aggregate download size of 350 KB, or a 30% savings. If you add a third or fourth application, the aggregate download size increases by 100 KB for each additional application. The benefits of using an RSL increase with each new application in the same domain.



In this example, the applications with statically-linked libraries run only after Flash Player loads the 250 KB for each application. With dynamically linked RSLs, however, only the first application must load the entire 250 KB (the combined size of the application and the RSL). The second application runs when just 100 KB loads because the RSL is cached.

The illustrated scenario shows one possible outcome. If your applications do not use all of the components in the RSL, the size difference (and, as a result, the savings in download time) might not be as great.

Suppose that each application only uses half of the components in the RSL. If you statically link the library, the total download size is 100 KB + 75 KB for the first application and the library and 100 KB + 75 KB for the second application and the library, or an aggregate download size of 350 KB. In this second case, the combined download size when using RSLs and when not using RSLs is the same.

In general, the more Flex applications that are running on the same domain that use a common RSL, the greater the benefit.

Using RSLs

To use RSLs, you perform the following tasks:

- **Create a library** You can do this with either the Flex Builder Library Project or the compc command-line compiler. You can output the library as a SWC file or an open directory. The library includes a library.swf file and a catalog.xml file. For more information, see "Creating libraries" on page 238.
- **Define the library as an external resource** You do this when you compile the application by passing the compile-time location of the components as well as the run-time location of the library's library.swf file. For more information, see "Using RSLs" on page 240.

Creating libraries

You can create a library using either Flex Builder or the compc command-line compiler. The library is a SWC file or open directory that contains a library.swf file and a catalog.xml file. A library generally contains custom components and classes. You can use libraries as RSLs, but it is not a requirement.

In Flex Builder, you add resources to a library by using the Flex Library Build Path dialog box.

On the command line, you add files to the library by using the `include-classes` and `include-namespaces` options.

The following command-line example creates a library called CustomCellRenderer with the compc compiler:

```
compc -source-path ../mycomponents/components/local
  -include-classes CustomCellRendererComponent -directory=true -debug=false
  -output ../libraries/CustomCellRenderer
```

All included components must be statically linked in the library.swf file of the resulting SWC file. When you use the compc compiler to create the library, do not use the `include-file` option to add files to the library, because this option does not statically link files inside the library.swf file.

You can specify that the output be an open directory rather than a SWC file by using the `directory` option. If you do not specify that the output be an open directory, you must extract the library.swf file from the SWC file with a compression utility, such as PKZip, because you later refer to the location of that file. In addition, when you deploy the RSL and the application, the SWF file must be unzipped.

The options on the command line in the previous example can also be represented by a configuration file as the following example shows:

```
<?xml version="1.0">
<flex-config>
  <compiler>
    <source-path>
      <path-element>mycomponents/components/local</path-element>
    </source-path>
  </compiler>
  <output>libraries/CustomCellRenderer</output>
  <directory>true</directory>
  <debug>false</false>
  <include-classes>
    <class>CustomCellRendererComponent</class>
  </include-classes>
</flex-config>
```

The output is an open directory that contains the following files:

- catalog.xml
- library.swf

After you create the library.swf file, you can compile your application and specify that file's location for use at run time. For more information, see "Using RSLs" on page 240.

Set the `debug` option to `false` when you use the compc compiler to compile an RSL. The default value is `true` for compc, which means that the compiler, by default, includes extra information in the SWC file to make it debuggable. Avoid this for an RSL you intend to use in production so that the RSL's files are as small as possible.

For more information on using the compc compiler options, see Chapter 9, "Using the Flex Compilers," on page 179.

Using RSLs

To use RSLs when compiling your application, you use the following application compiler options:

- `runtime-shared-libraries` Provides the run-time location of the shared library.
- `external-library-path|externs|load-externs` Provides the compile-time location of the libraries. The compiler requires this for dynamic linking.

Use the `runtime-shared-libraries` option to specify the location of the SWF file that the application loads as an RSL at run time. You specify the location of the SWF file relative to the deployment location of the application. For example, if you store the library.swf file in the *web_root*/libraries directory on the web server, and the application in the web root, you specify libraries/library.swf.

You can specify one or more libraries with this option. If you specify more than one library, separate each library with a comma.

Use the `external-library-path` option to specify the location of the library's SWC file or open directory that the application references at compile time. The compiler provides compile-time link checking by using the library specified by this option. You can also use the `externs` or `load-externs` options to specify individual classes or an XML file that defines the contents of the library.

The following command-line example compiles the MyApp application that uses two libraries:

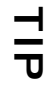
```
mxmlc -runtime-shared-libraries=
  ../libraries/CustomCellRenderer/library.swf,
  ../libraries/CustomDataGrid/library.swf
  -external-library-path=../libraries/CustomCellRenderer,
  ../libraries/CustomDataGrid MyApp.mxml
```

The order of the libraries is significant because the base classes must be loaded before the classes that use them.

You can also use a configuration file, as the following example shows:

```
<compiler>
  <external-library-path>
    <path-element>../libraries/CustomCellRenderer</path-element>
    <path-element>../libraries/CustomDataGrid</path-element>
    <path-element>../libs/playerglobal.swc</path-element>
  </external-library-path>
</compiler>
<runtime-shared-libraries>
  <url>../libraries/CustomCellRenderer/library.swf</url>
  <url>../libraries/CustomDataGrid/library.swf</url>
</runtime-shared-libraries>
```

The `runtime-shared-libraries` option is the relative location of the library.swf files when the application has been deployed. The `external-library-path` option is the location of the SWC file or open directory at compile time. Because of this, you must know the deployment locations of the libraries relative to the application when you compile it. You do not have to know the deployment structure when you create the library, because you use the compc command-line compiler to create a SWC file.



The playerglobal.swc file is the default external library that defines the base classes for Flash Player.

In the previous example, the file structure at compile time looks like this:

```
c:/appfiles/MyApp.mxml
c:/libraries/CustomCellRenderer/CustomCellRenderer.swc
c:/libraries/CustomDataGrid/CustomDataGrid.swc
```

The presence of the library.swf at compile time is not actually necessary. The Flex compiler does not verify the SWF file's existence, but does compile the location specified by the `runtime-shared-libraries` option into the application code.

The deployed files are structured like this:

```
web_root/MyApp.swf
web_root/libraries/CustomCellRenderer/library.swf
web_root/libraries/CustomDataGrid/library.swf
```

For more information about using the compilers, see Chapter 9, "Using the Flex Compilers," on page 179.

RSL example

This example walks you through the process of using an RSL with an application. It uses the command-line compilers, but you can apply the same process to creating and using RSLs with a Flex Builder project.

Keep in mind that a SWC file is a library that contains a SWF file that contains run-time definitions and additional metadata that is used by the compiler for dependency tracking, among other things. You can open SWC files with any archive tool, such as WinZip, and examine the contents.

Before you use an RSL, first learn how to statically link a SWC file. To do this, you build a SWC file and then set up your application to use that SWC file.

In this example you have an application named app.mxml that uses the ProductConfigurator.as and ProductView.as classes. The files and classes involved are:

```
project/src/app.mxml
project/libsrc/ProductConfigurator.as
project/libsrc/ProductView.as
project/lib/
project/bin/
```

To compile this application, you can link the classes in the /libsrc directory using the `source-path` option, as the following example shows:

```
cd project/src
mxmlc -o=../bin/app.swf -source-path+=../libsrc app.mxml
```

This command adds the ProductConfigurator and ProductView classes to the SWF file.

To use a library, you use the compc compiler to create the SWC file, as the following command shows:

```
cd project
compc -source-path+=libsrc -debug=false -o=lib/mylib.swc
  ProductConfigurator ProductView
```

Be sure to set the `debug` option to `false`. The result is the project/lib/mylib.swc file, which contains the implementations of the ProductConfigurator and ProductView classes.

To recompile your application with the new library, you add the library with the `library-path` option, as the following example shows:

```
cd project/src
mxmlc -o=../bin/app.swf -library-path+=../lib/mylib.swc app.mxml
```

After you create a library, you can recompile the application to use the RSL. Complete the following three steps:

1. Instruct the compiler to ***not*** link the library classes into your application.
2. Prepare the RSL so that it can be found and used at run time.
3. Instruct the compiler to generate extra metadata that loads your RSL.

The first step is to specifically exclude the classes in your library from being compiled into your application. You do this with the `external-library-path` option, as the following example shows:

```
cd project/src
mxmlc -o=../bin/app.swf -external-library-path+=../lib/mylib.swc app.mxml
```

If you tried to run app.swf now, Flash Player would throw a run-time exception because the ProductConfigurator and ProductView classes are not yet defined. The `external-library-path` configuration option instructs the compiler to compile against these libraries, but omit their definitions. You can also use the `externs` option if you want to exclude classes on a class-by-class basis, but it is generally more convenient to use the `external-library-path` option.

The next step is to prepare the RSL so that it can be found at run time. To do this, you extract the library.swf file from the SWC file with any archive tool, such as WinZip or jar.

The following example extracts the SWF file by using the unzip utility on the command line:

```
cd project/lib
unzip mylib.swc library.swf
mv library.swf ../bin/myrsl.swf
```

This example renames the library.swf file to myrsl.swf and and moves it to the same directory as the application SWF file.

The final step is to recompile the application to use the RSL. You do this with the `runtime-shared-libraries` option, as the following example shows:

```
cd project/src
mxmlc -o=../bin/app.swf -external-library-path+=../lib/mylib.swc
  -runtime-shared-libraries=myrsl.swf app.mxml
```

The new SWF file dynamically loads the RSL before running the application.

CHAPTER 11

Logging

11

You can log messages at several different points in an Adobe Flex application's life cycle. You can log messages when you compile the application, when you deploy it to a web application server, or when a client runs it. You can log messages on the server or on the client. These messages are useful for informational, diagnostic, and debugging activities. This topic describes the various logging mechanisms you can use when you work with Flex applications.

Contents

About logging 245
Using the debugger version of Flash Player 247
Client-side logging and debugging. 251
Compiler logging. 264
Web-tier logging. 265

About logging

When you encounter a problem with your application, whether during compilation or at run time, the first step is to gather diagnostic information to locate the cause of the problem. The source of a problem typically is in one of two places: the server web application, or the client application.

Flex includes several different logging and error reporting mechanisms that you can use to track down failures:

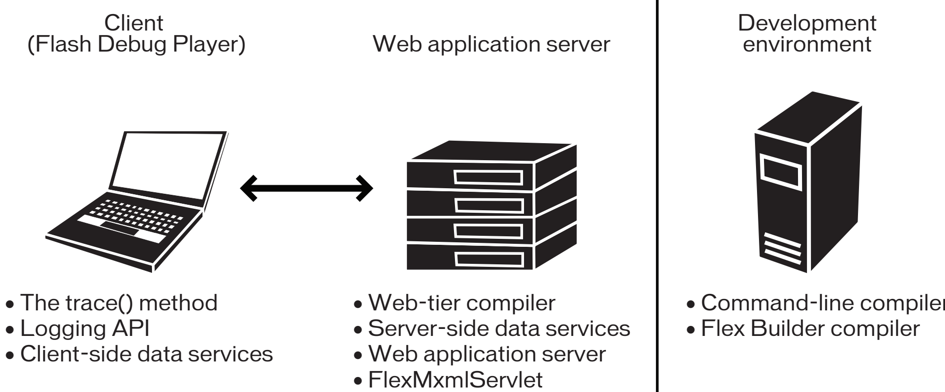
Client-side logging and debugging With the debugger version of Flash Player, you can use the global `trace()` method to write out messages or configure a TraceTarget to customize log levels of applications for data services-based applications. For more information, see "Client-side logging and debugging" on page 251.

Compiler logging When compiling your Flex applications from the command line and in Flex Builder, you can view deprecation and warning messages, and sources of fatal errors. For more information, see "Compiler logging" on page 264.

Web-tier logging The Flex web application provides some control over logging messages for the FlexMxmlServlet and lets you write the web-tier compiler log messages to your application server's logs. For more information, see "Web-tier logging" on page 265.

Server-side data services logging You can perform server-side logging for data service messages. You configure server-side logging in the logging section of the Flex services configuration file (services-config.xml). By default, output is sent to System.out, but you can also configure the logging to use your application server's logging mechanism. For more information, see Chapter 43, "Configuring Data Services," in *Flex 2 Developer's Guide*.

The following example shows the types of logging you can do in the appropriate environment:



To use client-side debugging utilities such as the `trace()` global method and client-side data services logging, you must install and configure the debugger version of Flash Player. This is described in "Using the debugger version of Flash Player" on page 247. The debugger verison of Flash Player is not required to log compiler messages, server-side data services messages, or web-tier compiler messages.

Using the debugger version of Flash Player

The debugger verison of Flash Player is a tool for development and testing. Like the standard version of Adobe Flash Player 9, it runs SWF files in a browser or on the desktop in a stand-alone player. Unlike Flash Player, the debugger version of Flash Player enables you to do the following:

- Output statements and application errors to the debugger version of the Flash Player local log file by using the `trace()` method.
- Write data services log messages to the local log file of the debugger version of Flash Player.
- View run-time errors (RTEs).
- Use the fdb command-line debugger.
- Use the Flex Builder debugging tool.

> **NOTE** Any client running the debugger version of Flash Player can view your application's `trace()` statements and other log messages unless you disable them. For more information, see "Suppressing debug output" on page 81.

The debugger version of Flash Player lets you take advantage of the client-side logging utilities such as the `trace()` method and the Logging API. You are not required to run the debugger version of Flash Player to log compiler and web-tier messages because these logging mechanisms do not require a player.

In nearly all respects, the debugger version of Flash Player appears to be the same as the standard version of Flash Player. To determine whether or not you are running the debugger version of Flash Player, use the instructions in "Determining Flash Player version in Flex" on page 250.

The debugger version of Flash Player comes in ActiveX, Plug-in, and stand-alone versions for Microsoft Internet Explorer, Netscape-based browsers, and desktop applications, respectively. You can find the debugger version of Flash Player installers in the following locations:

- Flex Builder: *install_dir*/Player/debug
- Flex SDK: *install_dir*/player/debug
- Flex Data Services: *install_dir*/resources/player/debug

Uninstall your current Flash Player before you install the debugger version of Flash Player. For information on installing the debugger version of Flash Player, see the Flex installation instructions.

You can enable or disable trace logging, change the location of the trace output, and perform other configuration tasks for the debugger version of Flash Player. For more information, see "Configuring the debugger version of Flash Player" on page 248.

Configuring the debugger version of Flash Player

You use the settings in the mm.cfg text file to configure the debugger version of Flash Player. You must create this file when you first configure the debugger version of Flash Player. The location of this file depends on your operating system. The following table shows where to create the mm.cfg file for several operating systems:

Operating system	Create file in ...
Macintosh OS X	`MacHD:Library:Application Support:macromedia:mm.cfg`
Microsoft Windows XP	`C:\Documents and Settings\`*`user_name`*`\mm.cfg`
Windows 2000	`C:\mm.cfg`
Linux	`home/`*`user_name`*`/mm.cfg`

TIP

On Windows, the location of the mm.cfg file is determined by the `HOMEDRIVE` and `HOMEPATH` environment variables. The `HOMEDRIVE` variable specifies the drive letter of the path to the home directory. On most Microsoft Windows systems, the default value is C:, the primary hard disk drive. The `HOMEPATH` variable specifies the path to the home directory, relative to `HOMEDRIVE`.
On Microsoft Windows 2000, the default location is \.
On Microsoft Windows XP, the default location is \Documents and Settings*user_name*, where *user_name* is your system user name.

The following table lists the properties that you can set in the mm.cfg file:

Property	Description
ErrorReportingEnable	Enables the logging of error messages. Set the ErrorReportingEnable property to 1 to enable the debugger version of Flash Player to write error messages to the log file. To disable logging of error messages, set the ErrorReportingEnable property to 0. The default value is 0.
MaxWarnings	Sets the number of warnings to log before stopping. The default value of the MaxWarnings property is 100. After 100 messages, the debugger version of Flash Player writes a message to the file stating that further error messages will be suppressed. Set the MaxWarnings property to override the default message limit. For example, you can set it to 500 to capture 500 error messages. Set the MaxWarnings property to 0 to remove the limit so that all error messages are recorded.
TraceOutputFileEnable	Enables trace logging. Set TraceOutputFileEnable to 1 to enable the debugger version of Flash Player to write trace messages to the log file. Disable trace logging by setting the TraceOutputFileEnable property to 0. The default value is 0.
TraceOutputFileName	**Note**: Beginning with the Flash Player 9 Update, Flash Player ignores the TraceOutputFileName property. Sets the location of the log file. By default, the debugger version of Flash Player writes error messages to a file named flashlog.txt, located in the same directory in which the mm.cfg file is located. Set TraceOutputFileName to override the default name and location of the log file by specifying a new location and name in the following form: On Macintosh OS X, you should use colons to separate directories in the TraceOutputFileName path rather than slashes. TraceOutputFileName=<*fully qualified path/filename*>

The following sample mm.cfg file enables error reporting and trace logging:

```
ErrorReportingEnable=1
TraceOutputFileEnable=1
```

Log file location

The default log file location changed between the initial Flash Player 9 release and the Flash Player 9 Update. In the initial Flash Player 9 release, the default location is the same directory as the mm.cfg file and you can update the log file location and name through the `TraceOutputFileName` property. Beginning with the Flash Player 9 Update, you cannot modify the log file location or name and the log file location has changed, as follows:

Windows C:\Documents and Settings*user_name*\Application Data\Macromedia\Flash Player\Logs

Macintosh Users/*user_name*/Library/Preferences/Macromedia/Flash Player/Logs/

Linux home/*user_name*/macromedia/Flash_Player/Logs/flashlog.txt

Determining Flash Player version in Flex

To determine which version of Flash Player you are currently using—the standard version or the debugger version—you can use the Capabilities class. This class contains information about Flash Player and the system that it is currently operating on. To determine if you are using the debugger version of Flash Player, you can use the `isDebugger` property of that class. This property returns a Boolean value: the value is `true` if the current player is the debugger version of Flash Player and `false` if it is not.

The following example uses the `playerType`, `version`, and `isDebugger` properties of the Capabilities class to display information about the Player:

```
<?xml version="1.0"?>
<!-- logging/CheckDebugger.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        import flash.system.Capabilities;

        private function reportVersion():String {
            if (Capabilities.isDebugger) {
                return "Debugger version of Flash Player";
            } else {
                return "Flash Player";
            }
        }
        private function reportType():String {
            return Capabilities.playerType + " (" + Capabilities.version +
")";
        }
    ]]></mx:Script>
    <mx:Label text="{reportVersion()}"/>
    <mx:Label text="{reportType()}"/>
</mx:Application>
```

Other properties of the Capabilities class include `hasPrinting`, `os`, and `language`.

Client-side logging and debugging

Often, you use the `trace()` method when you debug applications to write a checkpoint message on the client, which signals that your application reached a specific line of code, or to output the value of a variable.

The debugger version of Flash Player has two primary methods of writing messages that use `trace()`:

- The global `trace()` method. The global `trace()` method prints Strings to a specified output log file. For more information, see "Using the global trace() method" on page 252.
- Logging API. The Logging API provides a layer of functionality on top of the `trace()` method that you can use with your custom classes or with the data service APIs. For more information, see "Using the Logging API" on page 252.

Configuring the debugger version of Flash Player to record trace() output

To record messages on the client, you must use the debugger version of Flash Player.

The debugger version of Flash Player sends output from the `trace()` method to the flashlog.txt file. The default location of this file is the same directory as the mm.cfg file. You can customize the location of this file by using the `TraceOutputFileName` property in the mm.cfg file. You must also set `TraceOutputFileEnable` to 1 in your mm.cfg file.

For more information, see "Configuring the debugger version of Flash Player" on page 248.

Using the global trace() method

You can use the debugger version of Flash Player to capture output from the global `trace()` method and write that output to the client log file. You can use `trace()` statements in any ActionScript or MXML file in your application. Because it is a global function, you are not required to import any ActionScript classes packages to use the `trace()` method.

The following example defines a function that logs the various stages of the Button control's startup life cycle:

```
<?xml version="1.0"?>
<!-- logging/ButtonLifeCycle.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script><![CDATA[
        private function traceEvent(event:Event):void {
            trace(event.currentTarget + ":" + event.type);
        }
    ]]></mx:Script>

    <mx:Button id="b1" label="Click Me"
        preinitialize="traceEvent(event)"
        initialize="traceEvent(event)"
        creationComplete="traceEvent(event)"
        updateComplete="traceEvent(event)"
    />

</mx:Application>
```

The following example shows the output of this simple application:

```
TraceLifecycle_3.b1:Button:preinitialize
TraceLifecycle_3.b1:Button:initialize
TraceLifecycle_3.b1:Button:creationComplete
TraceLifecycle_3.b1:Button:updateComplete
TraceLifecycle_3.b1:Button:updateComplete
TraceLifecycle_3.b1:Button:updateComplete
```

Messages that you log by using the `trace()` method should be Strings. If the output is not a String, use the `String(...)` String conversion function, or use the object's `toString()` method, if one is available, before you call the `trace()` method.

To enable tracing, you must configure the debugger version of Flash Player as described in "Configuring the debugger version of Flash Player to record trace() output" on page 251.

Using the Logging API

The Logging API lets an application capture and write messages to a target's configured output. Typically the output is equivalent to the global `trace()` method, but it can be anything that an active target supports.

The Logging API consists of the following parts:

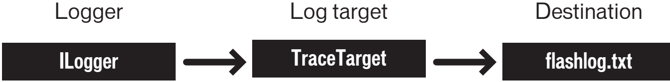
Logger The logger provides an interface for sending a message to an active target. Loggers implement the ILogger interface and call methods on the Log class. The two classes of information used to filter a message are category and level. Each logger operates under a category. A *category* is a string used to filter all messages sent from that logger. For example, a logger can be acquired with the category “orange”. Any message sent using the “orange” logger only reaches those targets that are listening for the “orange” category. In contrast to the category that is applied to all messages sent with a logger, the *level* provides additional filtering on a per-message basis. For example, to indicate that an error occurred within the “orange” subsystem, you can use the error level when logging the message. The supported levels are defined by the LogEventLevel class. The Flex framework classes that use the Logging API set the category to the fully qualified class name as a convention.

Log target The log target defines where log messages are written. Flex predefines two log targets: TraceTarget and MiniDebugTarget. The most commonly used log target is TraceTarget. This log target connects the Logging API to the trace system so that log messages are sent to the same location as the output of the `trace()` method. For more information on the `trace()` method, see “Using the global trace() method” on page 252.

You can also write your own custom log target. For more information, see “Implementing a custom logger with the Logging API” on page 259.

Destination The destination is where the log message is written. Typically, this is a file, but it can also be a console or something else, such as an in-memory object. The default destination for TraceTarget is the flashlog.txt file. You configure this destination on the client.

The following example shows a sample relationship between a logger, a log target, and a destination:



Flex Data Services uses the Logging API for diagnostic and error reporting. For more information, see “Using the Logging API with data services” on page 255.

You can also use the Logging API to send messages from custom code you write. You can do this when you create a set of custom APIs or components or when you extend the Flex framework classes and you want users to be able to customize their logging. For more information, see “Implementing a custom logger with the Logging API” on page 259.

The following packages within the Flex framework are the only ones that use the Logging API:

- mx.rpc.*
- mx.messaging.*
- mx.data.*

To configure client-side logging in MXML or ActionScript, create a TraceTarget object to log messages. The TraceTarget object logs messages to the same location as the output of the `trace()` statements. You can also use the TraceTarget to specify which classes to log messages for, and what level of messages to log.

The levels of logging messages are defined as constants of the LogLevelEvent class. The following table lists the log level constants and their numeric equivalents, and describes each message level:

Logging level constant (int)	Description
`ALL (0)`	Designates that messages of all logging levels should be logged.
`DEBUG (2)`	Logs internal Flex activities. This is most useful when debugging an application. Select the `DEBUG` logging level to include `DEBUG`, `INFO`, `WARN`, `ERROR`, and `FATAL` messages in your log files.
`INFO (4)`	Logs general information. Select the `INFO` logging level to include `INFO`, `WARN`, `ERROR`, and `FATAL` messages in your log files.
`WARN (6)`	Logs a message when the application encounters a problem. These problems do not cause the application to stop running, but could lead to further errors. Select the `WARN` logging level to include `WARN`, `ERROR`, and `FATAL` messages in your log files.
`ERROR (8)`	Logs a message when a critical service is not available or a situation has occurred that restricts the use of the application. Select the `ERROR` logging level to include `ERROR` and `FATAL` messages in your log files.
`FATAL (1000)`	Logs a message when an event occurs that results in the failure of the application. Select the `FATAL` logging level to include only `FATAL` messages in your log files.

The log level lets you restrict the amount of messages sent to any running targets. Whatever log level you specify, all "lower" levels of messages are written to the log. For example, if you set the log level to `DEBUG`, all log levels are included. If you set the log level to `WARNING`, only `WARNING`, `ERROR`, and `FATAL` messages are logged. If you set the log level to the lowest level of message, `FATAL`, only `FATAL` messages are logged.

Using the Logging API with data services

The data services classes are designed to use the Logging API to log client-side and server-side messages.

To enable the Logging API with data services:

1. Create a TraceTarget logging target and set the value of one or more filter properties to include the classes whose messages you want to log. You can filter the log messages to a specific class or package. You can use wildcards (*) when defining a filter.
2. Set the log level by using the `level` property of the log target. You can also add detail to the log file output, such as the date and time that the event occurred, by using properties of the log target.
3. When you create a target within ActionScript, call the Log class's `addTarget()` method to add the new target to the logging system. Calling the `addTarget()` method is not required when you create a target in MXML. As long as the client is using the debugger version of Flash Player and meets the requirements described in "Configuring the debugger version of Flash Player to record trace() output" on page 251, the messages are logged.

The following example configures a TraceTarget logging target in ActionScript:

```
<?xml version="1.0"?>
<!-- charts/ActionScriptTraceTarget.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initLogging();">

    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        import mx.logging.targets.*;
        import mx.logging.*;

        [Bindable]
        public var myData:ArrayCollection;

        private function initLogging():void {
            // Create a target.
            var logTarget:TraceTarget = new TraceTarget();

            // Log only messages for the classes in the mx.rpc.* and
            // mx.messaging packages.
            logTarget.filters=["mx.rpc.*","mx.messaging.*"];

            // Log all log levels.
            logTarget.level = LogEventLevel.ALL;

            // Add date, time, category, and log level to the output.
            logTarget.includeDate = true;
            logTarget.includeTime = true;
            logTarget.includeCategory = true;
            logTarget.includeLevel = true;

            // Begin logging.
            Log.addTarget(logTarget);
        }
    ]]></mx:Script>

    <!-- HTTPService is in the mx.rpc.http.* package -->
    <mx:HTTPService
        id="srv"
        url="../assets/data.xml"
        useProxy="false"
        result="myData=ArrayCollection(srv.lastResult.data.result)"
    />

    <mx:LineChart id="chart" dataProvider="{myData}" showDataTips="true">
        <mx:horizontalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
            <mx:LineSeries yField="apple" name="Apple"/>
```

```
            <mx:LineSeries yField="orange" name="Orange"/>
            <mx:LineSeries yField="banana" name="Banana"/>
        </mx:series>
    </mx:LineChart>

    <mx:Button id="b1" click="srv.send();" label="Load Data"/>

</mx:Application>
```

In the preceding example, the `filters` property is set to log messages for all classes in the mx.rpc and mx.messaging packages. In this case, it logs messages for the HTTPService class, which is in the mx.rpc.http.* package.

You can also configure a log target in MXML. When you do this, though, you must be sure to use an appropriate number (such as 2) rather than a constant (such as `DEBUG`). The following example sets the values of the filters for a TraceTarget logging target by using MXML syntax:

```
<?xml version="1.0"?>
<!-- charts/MXMLTraceTarget.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
creationComplete="initApp();">

    <mx:Script><![CDATA[
        import mx.collections.ArrayCollection;
        import mx.logging.Log;

        [Bindable]
        public var myData:ArrayCollection;

        private function initApp():void {
            Log.addTarget(logTarget);
        }

    ]]></mx:Script>

    <mx:TraceTarget id="logTarget" includeDate="true" includeTime="true"
includeCategory="true" includeLevel="true">
        <mx:filters>
            <mx:Array>
                <mx:String>mx.rpc.*</mx:String>
                <mx:String>mx.messaging.*</mx:String>
            </mx:Array>
        </mx:filters>
        <!-- 0 is represents the LogEventLevel.ALL constant. -->
        <mx:level>0</mx:level>
    </mx:TraceTarget>

    <!-- HTTPService is in the mx.rpc.http.* package -->
    <mx:HTTPService
        id="srv"
        url="../assets/data.xml"
        useProxy="false"
        result="myData=ArrayCollection(srv.lastResult.data.result)"
    />

    <mx:LineChart id="chart" dataProvider="{myData}" showDataTips="true">
        <mx:horizontalAxis>
            <mx:CategoryAxis categoryField="month"/>
        </mx:horizontalAxis>
        <mx:series>
            <mx:LineSeries yField="apple" name="Apple"/>
            <mx:LineSeries yField="orange" name="Orange"/>
            <mx:LineSeries yField="banana" name="Banana"/>
```

```
        </mx:series>
    </mx:LineChart>

    <mx:Button id="b1" click="srv.send();" label="Load Data"/>

</mx:Application>
```

Implementing a custom logger with the Logging API

If you write custom components or an ActionScript API, you can use the Logging API to access the trace system in the debugger version of Flash Player. You do this by defining your log target as a TraceTarget, and then calling methods on your logger when you log messages.

The following example extends a Button control. It writes log messages for the startup life cycle events, such as `initialize` and `creationComplete`, and the common UI events, such as `click` and `mouseOver`.

```
package { // The empty package.
    import mx.controls.Button;
    import flash.events.*;
    import mx.logging.*;
    import mx.logging.targets.*;

    public class MyCustomLogger extends Button {

        private var myLogger:ILogger;

        public function MyCustomLogger() {
            super();
            initListeners();
            initLogger();
        }
        private function initListeners():void {
            // Add event listeners life cycle events.
            addEventListener("preinitialize", logLifeCycleEvent);
            addEventListener("initialize", logLifeCycleEvent);
            addEventListener("creationComplete", logLifeCycleEvent);
            addEventListener("updateComplete", logLifeCycleEvent);

            // Add event listeners for other common events.
            addEventListener("click", logUIEvent);
            addEventListener("mouseUp", logUIEvent);
            addEventListener("mouseDown", logUIEvent);
            addEventListener("mouseOver", logUIEvent);
            addEventListener("mouseOut", logUIEvent);
        }
        private function initLogger():void {
            myLogger = Log.getLogger("MyCustomClass");
        }

        private function logLifeCycleEvent(e:Event):void {
            if (Log.isInfo()) {
                myLogger.info(" STARTUP: " + e.target + ":" + e.type);
            }
        }

        private function logUIEvent(e:MouseEvent):void {
            if (Log.isDebug()) {
                myLogger.debug(" EVENT:   " + e.target + ":" + e.type);
            }
        }
    }
}
```

Within the application that uses the MyCustomLogger class, define a TraceTarget, as the following example shows:

```
<?xml version="1.0"?>
<!-- charts/LoadCustomLogger.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns="*" >
    <mx:TraceTarget level="0" includeDate="true" includeTime="true"
includeCategory="true" includeLevel="true">
        <mx:filters>
            <mx:Array>
                <mx:String>*</mx:String>
            </mx:Array>
        </mx:filters>
    </mx:TraceTarget>
    <MyCustomLogger/>
</mx:Application>
```

After running this application, the flashlog.txt file looks similar to the following:

```
3/9/2006 18:58:05.042 [INFO] MyCustomLogger STARTUP:
  Main_3.mcc:MyCustomLogger:preinitialize
3/9/2006 18:58:05.487 [INFO] MyCustomLogger STARTUP:
  Main_3.mcc:MyCustomLogger:initialize
3/9/2006 18:58:05.557 [INFO] MyCustomLogger STARTUP:
  Main_3.mcc:MyCustomLogger:creationComplete
3/9/2006 18:58:05.567 [INFO] MyCustomLogger STARTUP:
  Main_3.mcc:MyCustomLogger:updateComplete
3/9/2006 18:58:05.577 [INFO] MyCustomLogger STARTUP:
  Main_3.mcc:MyCustomLogger:updateComplete
3/9/2006 18:58:05.577 [INFO] MyCustomLogger STARTUP:
  Main_3.mcc:MyCustomLogger:updateComplete
3/9/2006 18:58:06.849 [DEBUG] MyCustomLogger EVENT:
  Main_3.mcc:MyCustomLogger:mouseOver
3/9/2006 18:58:07.109 [DEBUG] MyCustomLogger EVENT:
  Main_3.mcc:MyCustomLogger:mouseDown
3/9/2006 18:58:07.340 [DEBUG] MyCustomLogger EVENT:
  Main_3.mcc:MyCustomLogger:mouseUp
3/9/2006 18:58:07.360 [DEBUG] MyCustomLogger EVENT:
  Main_3.mcc:MyCustomLogger:click
3/9/2006 18:58:07.610 [DEBUG] MyCustomLogger EVENT:
  Main_3.mcc:MyCustomLogger:mouseOut
```

To log a message, you call the appropriate method of the ILogger interface. The ILogger interface defines a method for each log level: `debug()`, `info()`, `warn()`, `error()`, and `fatal()`. The logger logs messages from these calls if their levels are at or under the log target's logging level. If the target's logging level is set to `all`, the logger records messages when any of these methods are called.

To improve performance, a static method corresponding to each level exists on the Log class, which indicates if any targets are listening for a specific level. Before you log a message, you can use one of these methods in an `if` statement to avoid running the code. The previous example uses the `Log.isDebug()` and `Log.isInfo()` static methods to ensure that the messages are of level `INFO` or `DEBUG` before logging them.

The previous example logs messages dispatched from any category because the TraceTarget's `filters` property is set to the wildcard character (*). The framework code sets the category of the logger to the fully qualified class name of the class in which logging is being performed. This is by convention only; any String specified when calling `Log.getLogger(x)` is the category required in a `filters` property to receive the message.

When you set the `filters` property for logging within the Flex framework, you can restrict this to a certain package or packages, or to other classes. To restrict the logging to your custom class only, add the category specified when the logger was acquired ("MyCustomLogger") to the `filters` Array, as the following example shows:

```
<mx:filters>
  <mx:Array>
    <mx:String>*</mx:String>
  </mx:Array>
</mx:filters>
```

In ActionScript, you can set the `filters` property by using the following syntax:

```
traceTarget.filters = ["p1.*", "p2.*", "otherPackage*"];
```

The wildcard character can appear only at the end of a value in the Array.

The `Log.getLogger()` method sets the category of the logger. You pass this method a String that defines the category.

The Flex packages that use the Logging API set the category to the current class name by convention, but it can be any String that falls within the filters definitions.

The value of the category must fall within the definition of at least one of the filters for the log message to be logged. For example, if you set the `filters` property to something other than "*" and you use `Log.getLogger("MyCustomLogger")`, the filter Array must include an entry that matches MyCustomLogger, such as "MyCustomLogger" or "My*".

You can include the logger's category in your log message, if you set the logger's `includeCategory` property to `true`.

You can also use the ILogger interface's `log()` method to customize the log message, and you can specify the logging level in that method. The following example logs messages that use the log level that is passed into the method:

```
package { // The empty package.
    import mx.controls.Button;
    import flash.events.*;
    import flash.events.MouseEvent;
    import mx.logging.*;
    import mx.logging.targets.*;

    public class MyCustomLogger2 extends Button {

        private var myLogger:ILogger;

        public function MyCustomLogger2() {
            super();
            initListeners();
            initLogger();
        }
        private function initListeners():void {
            // Add event listeners life cycle events.
            addEventListener("preinitialize", logLifeCycleEvent);
            addEventListener("initialize", logLifeCycleEvent);
            addEventListener("creationComplete", logLifeCycleEvent);
            addEventListener("updateComplete", logLifeCycleEvent);

            // Add event listeners for other common events.
            addEventListener("click", logUIEvent);
            addEventListener("mouseUp", logUIEvent);
            addEventListener("mouseDown", logUIEvent);
            addEventListener("mouseOver", logUIEvent);
            addEventListener("mouseOut", logUIEvent);
        }
        private function initLogger():void {
            myLogger = Log.getLogger("MyCustomClass");
        }

        private function logLifeCycleEvent(e:Event):void {
            if (Log.isInfo()) {
                dynamicLogger(LogEventLevel.INFO, e, "STARTUP");
            }
        }

        private function logUIEvent(e:MouseEvent):void {
            if (Log.isDebug()) {
                dynamicLogger(LogEventLevel.DEBUG, e, "EVENT");
            }
        }
```

```
        private function dynamicLogger(level:int, e:Event,
prefix:String):void {
            var s:String = "__" + prefix + "__" + e.currentTarget + ":" +
e.type;
            myLogger.log(level, s);
        }

    }
}
```

Compiler logging

Flex provides you with control over the output of warning and debug messages for the application and component compilers. When you compile, you can enable the message output to help you to locate and fix problems in your application. The settings that you use to control messages are defined in the flex-config.xml file or as command-line compiler options.

You have a high level of control over what compiler messages are displayed. For example, you can enable or disable messages, such as deprecation and binding-related warnings in the flex-config.xml file, by using the `show-deprecation-warnings` and `show-binding-warnings` options. The following example disables these messages in the flex-config.xml file:

```
<show-deprecation-warnings>false</show-deprecation-warnings>
<show-binding-warnings>false</show-binding-warnings>
```

You can also set these options on the command line or as an option of the Flex Builder application compiler.

If you enable compiler messages, they are written to the console window (or System.out) by default.

For more information on the compiler logging settings, see “Viewing warnings and errors” on page 227.

The web-tier compiler has an additional logging mechanism that you configure in a different configuration file. For more information, see “Web-tier logging” on page 265.

Web-tier logging

The web-tier compiler is used by the Flex web application that runs on a web application server. The Flex web application provides logging for the following types of messages:

Startup messages from the FlexMxmlServlet These messages are primarily informational in nature. For example, when the servlet starts, you will get information about your license version. For more information, see "Configuring web application logging" on page 265.

Web-tier compiler messages When the server running Flex Data Services responds to a request for an MXML file, it opens the web-tier compiler. The compiler can write error and warning messages to the wrapper and the web application logging mechanism. For more information, see "Configuring web-tier compiler logging" on page 267.

JRun application server messages If you are using the integrated JRun application server with Flex Data Services or Flex Builder, you can configure the JRun server logging mechanism by using the *flex_install_dir*/jrun4/servers/default/SERVER-INF/jrun.xml file. For more information, see "Configuring JRun logging" on page 410.

Configuring web application logging

By default, the Flex web application logs informational messages about the FlexMxmlServlet to the console and to a log file. These messages include information about the current license and file access.

When the server starts up, the FlexMxmlServlet writes a message regarding the license service. In addition, when a client makes a bad file request to the FlexMxmlServlet, the servlet writes a "File Not Found" error message to the log.

If you use the integrated JRun Java application server, you can view additional servlet initialization log entries in the *jrun_install_dir*/logs/default-event.log file. For more information, see "Configuring JRun logging" on page 410.

You configure web-tier logging by using the flex-webtier-config.xml file in the *flex_root*/WEB-INF/flex directory.

The default location of the log file is *flex_root*/WEB-INF/flex/logs/flex.log. You can set the name and location, size, and the number of backup files to keep in the flex-webtier-config.xml file.

The following example shows the default logging settings in the flex-config.xml file:

```
<logging>
  <level>info</level>
  <console>
    <enable>true</enable>
  </console>
  <file>
    <enable>true</enable>
    <file-name>/WEB-INF/flex/logs/flex.log</file-name>
    <maximum-size>200KB</maximum-size>
    <maximum-backups>3</maximum-backups>
  </file>
</logging>
```

The value of `<file-name>` must be an absolute path or it must start with a forward slash (/). If the location is invalid, Flex logs an error to the console and the application continues without file logging. On UNIX, if you start the path with a forward slash and one of the parent directories (other than root '/') exists, the path is absolute. Otherwise, the path is relative to the application.

You can also set the log level for the web application logger by using the `level` property. This setting applies only to the servlet messages and not to the compiler messages.

To disable the web application's file logging, set the value of the `enable` property in the `file` block to `false`. To disable the web application's console logging, set the value of the `<console>` property to `false`, as the following example shows:

```
<logging>
  <console>
    <enable>false</enable>
  </console>
  ...
</logging>
```

The web application logger can also send compiler error and warning messages to the web application server's logging mechanism. For more information, see "Configuring web-tier compiler logging" on page 267.

For more information about the web-tier compiler, see "Using the web-tier application compiler" on page 183.

Configuring web-tier compiler logging

A request for a *.mxml file to the web application server running Flex triggers the web-tier compiler to compile a SWF file. When this compiler runs, it can produce warning and error log messages just as the command-line compiler can. However, you can have the log messages sent to the web application server's logging mechanism by using the `log-compiler-errors` property in the flex-webtier-config.xml file.

The flex-webtier-config.xml file is located in the *flex_root*/WEB-INF/flex directory.

The web application logger does not log run-time messages, regardless of their severity. This logger only logs server-side compiler messages.

To log compiler errors to the log file or console, you must set `<production-mode>` to `false`. If production mode is enabled, Flex does not generate log entries for compiler errors.

To write web-tier compiler warning and error messages to the web application server logging mechanism, set the value of the `log-compiler-errors` property to `true`, as the following example shows:

```
<debugging>
  <log-compiler-errors>true</log-compiler-errors>
</debugging>
```

The web-tier compiler logging messages are controlled by the Flex web application logging mechanism. For example, the log file is limited to the size and number of backups specified in the `logging` block of the flex-webtier-config.xml file. For more information, see "Configuring web application logging" on page 265.

You use the `level` property in the `logging` block to set the level of log messages to write to the log file by using the `info` and `error` levels. Do not confuse these messages with the client-side error messages that share the same log-level names that are written to the trace logs. These messages apply only to the compiler's results, which are never written to logs on the client side.

When you set the `production-mode` property to `true`, the Flex application does not log any compiler messages, regardless of the value of the `log-compiler-errors` property.

For more information about the web-tier compiler, see "Using the web-tier application compiler" on page 183.

CHAPTER 12

Using the Command-Line Debugger

12

If you encounter errors in your applications, you can use the debugging tools to set and manage breakpoints in your code; control application execution by suspending, resuming, and terminating the application; step into and over the code statements; select critical variables to watch; evaluate watch expressions while the application is running; and so on. This topic describes debugging with the debugger version of Flash Player and the fdb command-line debugger.

Contents

About debugging 269

Invoking the command-line debugger 272

Configuring the command-line debugger 276

Using the command-line debugger commands. 277

About debugging

Debugging Flex applications can be as simple as enabling `trace()` statements or as complex as stepping into an application's source files and running the code, one line at a time. The Flex Builder debugger and the command-line debugger, fdb, let you step through and debug ActionScript files used by your Flex applications.

This topic describes how to use the fdb command-line debugger. To use the Flex Builder debugger, see *Using Flex Builder 2*. To use either debugger, you must install and configure the debugger version of Flash Player. To determine if you are running the debugger version or the standard version of Flash Player, open any Flex application in the player and right-click the mouse button. If you see the Show Redraw Regions option, you are running the debugger version of Flash Player. For more information about the debugger version of Flash Player, and how to detect which player you are running, see "Using the debugger version of Flash Player" on page 247.

Using the command-line debugger

The fdb command-line debugger is located in the *flex_install_dir*/bin directory. To start fdb, open a command prompt, change to that directory, and enter "fdb".

For a description of available commands, use the following tutorial command:

```
(fdb) help
```

For an overview of the fdb debugger, use the following tutorial command:

```
(fdb) tutorial
```

To debug a Flex application, you first generate a debug SWF file. Debug SWF files are similar to other application SWF files except that they contain debugging-specific information that the debugger and the debugger version of Flash Player use during debugging sessions. Debug SWF files are larger than non-debug SWF files, so generate them only when you are going to debug with them.

To generate the debug SWF file using the mxmlc command-line compiler, you set the `debug` option to `true`, either on the command line or in the flex-config.xml file. The following example sets the `debug` option to `true` on the command line:

```
mxmlc -debug=true myApp.mxml
```

To generate the debug SWF file using the web-tier compiler with Flex Data Services, you can either set the `debug` compiler option to `true` in the flex-config.xml file or append debug=true on the query string:

```
http://www.yourdomain.com/MyApp.mxml?debug=true
```

As long as production mode is not enabled, the web-tier compiler will generate a debug SWF file when you set it by using the `debug` query string parameter.

After you generate a debug SWF file, you then connect fdb to the debugger version of Flash Player. The debugger uses this connection to transfer information from the SWF file to the command line so that you can add breakpoints, inspect variables, and do other common debugging tasks. This connection is made through TCP/IP.

Command-line debugger limitations

The command-line debugger supports debugging only at the ActionScript level and does not support the Flash Timeline concept. The debugger also does not support adding breakpoints inside script snippets in MXML tags. You can set breakpoints on event handlers defined for MXML tags.

Flash Player may interact with a server. The debugger does not assist in debugging the server-side portion of the application, nor does it offer support for inspecting any of the IP transactions that take place from Flash Player to the server, and vice versa.

Command-line debugger shortcuts

You can open commands within the fdb debugger by using the fewest number of nonambiguous keystrokes. For example, to use the `print` command, you can type **p**, because no other command begins with that letter.

Using the default browser

When you debug an application in a web browser, fdb opens the player in the default browser. The *default browser* is the browser that opens when you open a web-specific file without specifying an application. You must also have the debugger version of Flash Player installed with this browser. If you do not have the correct version of the debugger version of Flash Player, Flash displays an error indicating that your Flash Player does not support all fdb commands.

Your default browser might not be the first browser that you installed on your computer. For example, if you installed another web browser *after* installing Microsoft Internet Explorer, Internet Explorer might not be your default browser.

To determine your default browser:

1. From the Windows ToolBar, select Start.
2. Select Run.
3. Enter a URL in the Run dialog box; for example:

   ```
   http://www.adobe.com
   ```
4. Click OK.

   Windows opens the default browser or displays an error message indicating that there is no application configured to handle your request.

To set Internet Explorer 6.x as your default browser:

1. Open the Internet Explorer application.
2. Select Tools > Internet Options.
3. Select the Programs tab.
4. Check the "Internet Explorer should check to see whether it is the default browser" checkbox.
5. Click OK.

The next time you start Internet Explorer, Internet Explorer prompts you to make it the default browser. If you are not prompted, Internet Explorer is already your default browser.

To set Firefox as your default browser:

1. Open the Firefox application.
2. Select Tools > Options.
3. Select the General icon to view general settings.
4. Check the "Firefox should check to see if it is the default browser when starting" checkbox.
5. Click OK.

   The next time you start FireFox, FireFox prompts you to make it the default browser. If you are not prompted, FireFox is already your default browser.

About the source files

Each application can have any number of ActionScript files. Some of the files that fdb steps into are external class files, and some are generated by the Flex compilers.

In general, Flex generates a single file that contains ActionScript statements used in `<mx:Script>` blocks in the root MXML file, and an additional file for each ActionScript class that the application uses. Flex generates many source files so that you can navigate the application from within the debugger.

To view a list of files that are used by the application you are debugging, use the `info files` command. For more information, see "Getting status" on page 286.

The generated ActionScript class files are sometimes referred to as compilation units. For more information about compilation units, see "About incremental compilation" on page 214.

Invoking the command-line debugger

This section describes how to start a debugging session with the fdb command-line debugger.

After you start a session, you typically type **continue** once before you set break points and perform other debugging tasks. This is because the first frame that suspends debugging occurs before the application has finished initialization.

For more information about which commands are available after you start a debugging session, see "Using the command-line debugger commands" on page 277.

Starting a session with the stand-alone debugger version of Flash Player

You can start a debugging session with the stand-alone debugger version of Flash Player. You do this by compiling the application into a SWF file, and then invoking the SWF file with the fdb command-line debugger. The fdb debugger opens the debugger version of the stand-alone Flash Player.

The debugger version of the stand-alone Flash Player runs as an independent application. It does not run within a web browser or other shell. The debugger version of the stand-alone Flash Player does not support any server requests, such as web services and dynamic SWF loading, so not all applications can be properly debugged inside the debugger version of the stand-alone Flash Player.

To debug with the debugger version of the stand-alone Flash Player:

1. Compile the Flex application's debug SWF file and set the `debug` option to `true`.

   The following example compiles an application with the mxmlc command-line compiler:

   ```
   mxmlc -debug=true myApp.mxml
   ```

   You can also compile an application SWF file by using the web-tier compiler or the Flex Builder compiler. For more information on Flex compilers, see "About the Flex compilers" on page 179.

2. Find the *flex_install_dir*/bin directory. You installed the Flex application files to this directory.

3. Type **fdb** from the command line. The fdb prompt appears.

   You can also open fdb with the JAR file, as the following example shows:

   ```
   java -jar ../lib/fdb.jar
   ```

4. Type **run** at the fdb prompt, followed by the path to the SWF file; for example:

   ```
   (fdb) run c:/myfiles/fonts/EmbedFlashTypeFont.swf
   ```

   The fdb debugger starts the Flex application in the debugger version of the stand-alone Flash Player, and the `(fdb)` command prompt appears. You can also start a session by typing `fdb` ***`filename`***`.swf` at the command prompt, rather than by using the `run` command.

Starting a session in a browser with Flex Data Services

You can have Flex Data Services compile the application when you want to start a debugging session. You do this by starting the Flex Data Services server and invoking the MXML file from fdb.

When you use Flex Data Services to start a debugging session, you open the debugger version of Flash Player in the default browser. For information about changing the default browser, see "Using the default browser" on page 271.

To debug a SWF file with Flex Data Services:

1. Start the application server on which the Flex web application is running. If you are using Flex Data Services with the integrated JRun server, start the default JRun server.
2. Open a separate console window.
3. Find the *flex_install_dir*/bin directory. You installed the Flex application files to this directory.
4. Type **fdb** from the command line, followed by the path to the MXML file; for example:

   ```
   fdb http://localhost:8100/flex/MyApp.mxml
   ```

   You can also open fdb with the JAR file, as the following example shows:

   ```
   java -jar ../lib/fdb.jar http://localhost:8100/flex/MyApp.mxml
   ```

   The fdb debugger starts the Flex application in your default browser and appends `?debug=true` to the query string. This generates a debug SWF file if one is not already present. The (fdb) command prompt appears in the console window, as the following example shows:

   ```
   Attempting to launch and connect to Player using URI
   http://localhost:8100/flex/wrapper/file1.mxml
   Player connected; session starting.
   Set breakpoints and then type 'continue' to resume the session.
   (fdb)
   ```

   If fdb does not connect to the player, you might not have the debugger version of Flash Player installed for your default browser. For information on installing the debugger version of Flash Player, see the installation instructions.

Starting a session in a browser without Flex Data Services

You can start a debugging session in a browser when you are not running Flex Data Services. This requires that you pre-compile the SWF file and are able to request it from a web server.

To debug a SWF file without Flex Data Services:

1. Compile the Flex application's debug SWF file and set the `debug` option to `true`. The following example compiles an application with the mxmlc command-line compiler:

   ```
   mxmlc -debug=true myApp.mxml
   ```

   You can also compile an application SWF file by using the web-tier compiler or the Flex Builder compiler. For more information on Flex compilers, see "About the Flex compilers" on page 179.

2. Create an HTML wrapper that embeds this SWF file, if you have not already done so. For more information on creating a wrapper, see Chapter 16, "Creating a Wrapper," on page 367.

3. Copy the SWF file and its wrapper files to your web server.

4. Find the *flex_install_dir*/bin directory. You installed the Flex application files to this directory.

5. Type **fdb** in the command line. The fdb prompt appears.

   You can also open fdb with the JAR file, as the following example shows:

   ```
   java -jar ../lib/fdb.jar
   ```

6. Type **run** at the fdb prompt; for example:

   ```
   (fdb) run
   ```

   This instructs fdb to wait for a player to connect to it.

7. In your browser, request a wrapper that embeds the debug SWF file.

   Do not request the SWF file directly in a browser because some browsers do not allow you to run a SWF file directly.

If you started the browser before you started fdb, the browser prompts you to supply a location for the debugger utility, as the following example shows:

In this case, select localhost and click OK.

Alternatively, you can type `run` *`filename`*`.html` at the command line, and fdb launches the browser for you.

Configuring the command-line debugger

You can configure the current session of the fdb command-line debugger using variables that exist entirely within fdb; they are not part of your application. The configuration variables are prefixed with $.

The following table describes the most common configuration variables used by fdb:

Variable	Description
`$invokegetters`	Set to 0 to prevent fdb from firing getter functions. The default value is `1` (enabled).
`$listsize`	Sets the number of lines to display with the list command. The default value is `10`.

To set the value of a configuration variable, you use the `set` command, as the following example shows:

```
(fdb) set $invokegetters = 0
```

For more information on using the `set` command, see Chapter 12, “Changing data values,” on page 283.

Using the command-line debugger commands

This section describes commands that you use to debug and navigate your Flex application by using the fdb command-line debugger.

Running the debugger

The fdb debugger provides several commands for stepping through the debugged application's files. The following table summarizes the commands that let you step through the debugged application files:

Command	Description
`continue`	Continues running the application.
`file [file]`	Specifies an application to be debugged, without starting it. This command does not cause the application to start; use the `run` command without an argument to start debugging the application.
`finish`	Continues until the function exits.
`next [N]`	Continues to the next source line in the application. The optional argument *N*, means do this *N* times or until the program stops for some other reason.
`quit`	Exits from the debug session.
`run [file]`	Starts a debugging session by running the specified file. To run the application that the file command previously specified, execute the `run` command without any options. The `run` command starts the application in a browser or stand-alone Flash Player.
`step [N]`	Steps into the application. Optional argument *N*, means do this *N* times or until the program stops for some other reason. These commands are non-blocking, which means that when they return, the client has at least begun the operation, but it has not necessarily finished it.

When you start a session, fdb stops execution before Flex renders the application on the screen. Use the `continue` command to get to the application's starting screen.

The following example shows a sample application after it starts:

```
(fdb) continue
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] ComboBase: y = undefined text_mc.bl = undefined
[trace] RadioButtonGroup.addInstance: instance =
  _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton1 data = undefined label = 2005
[trace] RadioButtonGroup.addInstance: instance =
  _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton2 data = undefined label = 2004
[trace] RadioButtonGroup.addInstance: instance =
  _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton3 data = undefined label = 2005
[trace] RadioButtonGroup.addInstance: instance =
  _level0._VBox0._Accordion0._For
m2._FormItem3._RadioButton4 data = undefined label = 2006
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 12
[trace] ComboBase: y = 0 text_mc.bl = 14
```

During the debugging session, you interact with the application in the debugger version of Flash Player. For example, if you select an item from the drop-down list, the debugger continues to output information to the command window:

```
[trace] SSL : ConfigureScrolling
[trace] SSP : 5 51 true 47
[trace] ComboBase: y = 0 text_mc.bl = 14
[trace] layoutChildren : bRowHeightChanged
[trace] >>SSL:layoutChildren
[trace] deltaRows 5
[trace] rowCount 5
[trace] <<SSL:layoutChildren
[trace] >>SSL:draw
[trace] bScrollChanged
[trace] SSL : ConfigureScrolling
[trace] SSP : 5 51 false 46
[trace] SSL Drawing Rows in UpdateControl 5
[trace] <<SSL:draw
```

You can store commonly used commands in a source file, and then load that file by using the `source` command. For more information, see "Accessing commands from a file" on page 281.

Setting breakpoints

Setting breakpoints is a critical aspect of debugging any application. You can set breakpoints on any ActionScript code in your Flex application. You can set breakpoints on statements in any external ActionScript file, on ActionScript statements in an `<mx:Script>` tag, or on MXML tags that have event handler properties. In the following MXML code, `click` is an event handler property:

```
<mx:Button click="ws.getWeather.send();"/>
```

Breakpoints are maintained from session to session. However, when you change the target file or quit fdb, breakpoints are lost.

The following table summarizes the commands for manipulating breakpoints with the ActionScript debugger:

Command	Description
`break [args]`	Sets a breakpoint at the specified line or function. The argument can be a line number or function name. With no arguments, the `break` command sets a breakpoint at the currently stopped line (not the currently listed line). If you specify a line number, fdb breaks at the start of code for that line. If you specify a function name, fdb breaks at the start of code for that function.
`clear [args]`	Clears a breakpoint at the specified line or function. The argument can be a line number or function name. If you specify a line number, fdb clears a breakpoint in that line. If you specify a function name, fdb clears a breakpoint at the beginning of that function. With no argument, fdb clears a breakpoint in the line that the selected frame is executing in. See the `delete` command, which clears breakpoints by number.
`commands [`*`breakpoint`*`]`	Sets commands to execute when the specified breakpoint is encountered. If you do not specify a breakpoint, the commands are applied to the last breakpoint.

Command	Description

`condition bnum [expression]`	Specifies a condition that must be met to stop at the given breakpoint. The fdb debugger evaluates *expression* when the *bnum* breakpoint is reached. If the value is `true` or nonzero, fdb stops at the breakpoint. Otherwise, fdb ignores the breakpoint and continues execution.
	To remove the condition from the breakpoint, do not specify an *expression*.
	You can use conditional breakpoints to stop on all events of a particular type. For example, to stop on every `initialize` event, use the following commands:
	`(fdb) break UIEvent:dispatch`
	`Breakpoint 18 at 0x16cb3: file UIEventDispatcher.as, line 190`
	`(fdb) condition 18 (eventObj.type == 'initialize')`

`delete [args]`	Deletes breakpoints. Specify one or more comma- or space-separated breakpoint numbers to delete those breakpoints. To delete all breakpoints, do not provide an argument.
`disable breakpoints [bp_num]`	Disables breakpoints. Specify one or more space-separated numbers as options to disable only those breakpoints.
`enable breakpoints [bp_num]`	Enables breakpoints that were previously disabled. Specify one or more space-separated numbers as options to enable only those breakpoints.

The following example sets a breakpoint on the `myFunc()` method, which is triggered when the user clicks a button:

```
(fdb) break myFunc
Breakpoint 1 at 0x401ef: file file1.mxml, line 5
(fdb) continue
Breakpoint 1, myFunc() at file1.mxml:5
 5              ta1.text = "Clicked";
(fdb)
```

To see all breakpoints and their numbers, use the `info breakpoints` command. This will also tell you if a breakpoint is unresolved.

You can use the `commands` command to periodically print out values of objects and variables whenever fdb encounters a particular breakpoint. The following example prints out the value of ta1.text (referred to as `$1`), executes the `where` command, and then continues when it encounters the button's click handler breakpoint:

```
(fdb) commands 1
Type commands for when breakpoint 1 is hit, one per line.
End with a line saying just 'end'.
>print ta1.text
>where
>continue
>end
(fdb) cont
Breakpoint 1, myFunc() at file1.mxml:5
 5              ta1.text = "Clicked";
$1 = ""
#0   [MovieClip 1].myFunc(event=undefined) at file1.mxml:5
#1   [MovieClip 1].handler(event=[Object 18127]) at file1.mxml:15
```

Breakpoints are not specific to a single SWF file. If you set a breakpoint in a file that is common to multiple SWF files, fdb applies the breakpoint to all SWF files.

For example, suppose you have four SWF files loaded and each of those SWF files contains the same version of an ActionScript file, view.as. To set a breakpoint in the `init()` function of the view.as file, you need to set only a single breakpoint in one of the view.as files. When fdb encounters any of the `init()` functions, it triggers the break.

Accessing commands from a file

You can use the `source` command to read fdb commands from a file and execute them. This lets you write commands such as breakpoints once and use them repeatedly when debugging the same application in different sessions or across different applications.

The `source` command has the following syntax:

```
(fdb) source file
```

The value of file can be a filename for a file in the current working directory or an absolute path to a remote file. To determine the current working directory, use the `pwd` command.

The following examples read in the mycommands.txt file from different locations:

```
(fdb) source mycommands.txt
(fdb) source mydir\mycommands.txt
(fdb) source c:\mydir\mycommands.txt
```

Examining data values

The `print` command displays values of members such as variables, objects, and properties. This command excludes functions, static variables, constants, and inaccessible member variables (such as the members of an Array).

The `print` command uses the following syntax:

```
print [variable_name | object_name[.] | property]
```

The `print` command prints the value of the specified variable, object, or property. You can specify the `name` or `name.property` to narrow the results. If fdb can determine the type of the entity, fdb displays the type.

If you specify the `print` command on an object, fdb displays a numeric identifier for the object.

To list all the properties of an object, use trailing dot-notation syntax. The following example prints all the properties of the object myButton:

```
(fdb) print myButton
```

To print the value of a single variable, use dot-notation syntax, as the following example shows:

```
(fdb) print myButton.label
```

Use the `what` command to view the context of a variable. The `what` command has the following syntax:

```
(fdb) what variable
```

Use the `display` command to add an expression to the autodisplay list. Every time debugging stops, fdb prints the list of expressions in the autodisplay list. The `display` command has the following syntax:

```
(fdb) display [expression]
```

The expression is the same as the arguments for the `print` command, as the following example shows:

```
(fdb) display myButton.color
```

To view all expressions on the autodisplay list, use the `info display` command.

To remove an expression from the autodisplay list, use the `undisplay` command. The `undisplay` command has the following syntax:

```
(fdb) undisplay [list_num]
```

Use the `undisplay` command without an argument to remove all entries on the autodisplay list. Specify one or more `list_num` options separated by spaces to remove numbered entries from the autodisplay list.

You can temporarily disable autodisplay expressions by using the `disable display` command. The `disable` command has the following syntax:

```
(fdb) disable display [display_num]
```

Specify one or more space-separated numbers as options to disable only those entries in the autodisplay list.

To re-enable the display list, use the `enable display` command, which has the same syntax as the `disable display` command.

Changing data values

You can use the `set` command to assign the value of a variable or a configuration variable. The `set` command has the following syntax:

```
set [expression]
```

Depending on the variable type, you use different syntax for the *expression*. The following example sets the variable `i` to the number 3:

```
(fdb) set i = 3
```

The following example sets the variable `employee.name` to the string `Reiner`:

```
(fdb) set employee.name = "Reiner"
```

The following example sets the convenience variable `$myVar` to the number `20`:

```
(fdb) set $myVar = 20
```

You use the `set` command to set the values of fdb configuration variables. For more information, see "Configuring the command-line debugger" on page 276.

Viewing file contents

You use the `list` command to view lines of code in the ActionScript files. The `list` command uses the following syntax:

```
list [- | line_num[,line_num] | [file_name:]line_num | file_name[:line_num]
  | [file_name:]function_name]
```

You use the `list` command to print the lines around the specified function or line of the current file. If you do not specify an argument, `list` prints 10 lines after or around the previous listing. If you specify a filename, but not a line number, `list` assumes line 1.

If you specify a single numeric argument, the `list` command lists 10 lines around that line. If you specify more than one comma-separated numeric argument, the `list` command displays lines between and including those line numbers.

To set the list location to where the execution is currently stopped, use the `home` command.

The following example lists code from line 10 to line 15:

```
(fdb) list 10, 15
```

If you specify a hyphen (-) in the previous example, the `list` command displays the 10 lines before a previous 10-line listing.

Specify a line number to list the lines around that line in the current file; for example:

```
(fdb) list 10
```

Specify a filename followed by a line number to list the lines around that line in that file; for example:

```
(fdb) list effects.mxml:10
```

Specify a function name to list the lines around the beginning of that function; for example:

```
(fdb) list myFunction
```

Specify a filename followed by a function name to list the lines around the beginning of that function. This lets you distinguish among like-named static functions; for example:

```
(fdb) list effects.mxml:myFunction
```

You can resolve ambiguous matches by extending the value of the function name or filename, as the following examples show:

Filenames:

```
(fdb) list UIOb
Ambiguous matching file names:
UIComponent.as#66
UIComponentDescriptor.as#67
UIComponentExtensions.as#68
(fdb) list UIComponent.
```

Function names:

```
(fdb) list init
Ambiguous matching function names:
init
initFromClipParameters
(fdb) list init(
```

Viewing and changing the current file

The `list` command acts on the current file by default. To change to a different file, use the `cf` command. The `cf` command has the following syntax:

```
(fdb) cf [file_name|file_number]
```

For example, to change the file to MyApp.mxml, use the following command:

```
(fdb) cf MyApp.mxml
```

If you do not specify a filename, the cf command lists the name and file number of the current file.

To view a list of all files used by the current application, use the info files command. For more information, see "Getting status" on page 286.

Viewing the current working directory

Use the pwd command to view the file system's current working directory. This is the directory from which fdb was run; for example:

```
(fdb) pwd
c:/Flex2SDK/bin/
```

Locating source files

Usually, fdb can find the source files for your application to display them with the list command. In some situations, however, you need to add a directory to the search path so that fdb can find the source files. This can be necessary, for example, when the application was compiled on a different computer than you are using to debug the application.

You use the directory command to add a directory to the search path. This command adds the specified directory or directories to the beginning of the list of directories that fdb searches for source files. The syntax for the directory command is as follows:

```
(fdb) directory path
```

For example:

```
(fdb) directory C:\MySource;C:\MyOtherSource
```

On Windows, use the semicolon character as a separator. On Macintosh and UNIX, use the colon character as a separator.

To see the current list of directories in the search path, use the show directories command.

Using truncated file and function names

The fdb debugger supports truncated file and function names. You can specify file_name and function_name arguments with partial names, as long as the names are unambiguous.

If you use truncated file and function names, fdb tries to map the argument to an unambiguous function name first, and then a filename. For example, list foo first tries to find a function unambiguously starting with *foo* in the current file. If this fails, it tries to find a file unambiguously starting with *foo*.

Printing stack traces

Use the `bt` command to display a back trace of all stack frames. The `bt` command has the following syntax:

```
(fdb) bt
```

Getting status

Use the `info` command to get general information about the application. The `info` command has the following syntax:

```
info [options] [args]
```

The `info` command displays general information about the application being debugged. The following table describes the options of the `info` command:

Option	Description
`arguments`	Displays the argument variables of the current stack frame.
`breakpoints`	Displays the status of user-settable breakpoints.
`display`	Displays the list of autodisplay expressions.

`files [arg]`	Displays the names of all files used by the target application. This includes authored files and system files, plus generated files. Also indicates the file number for each file.
	You can use wildcards and literals to select and sort the output. The `info files` command supports the following:
	`info files character` Alphabetically lists files with names that start with the specified *character*. The following example lists all files starting with the letter V:
	`info files V`
	`info files *.extension` Alphabetically lists all files with the given extension. The following example lists all files with the as extension:
	`info files *.as`
	`info files *string*` Alphabetically lists all files with names that include *string*.

`functions [arg]`	Displays all function names used in this application. The `info functions` command optionally takes an argument; for example: `info functions` Lists all functions in all files. `info functions` Lists all functions in the current file. `info functions MyApp.mxml` Lists all functions in the MyApp.mxml file.
`handle`	Displays settings for fault handling in the debugger.
`locals`	Displays the local variables of the current stack frame.
`sources`	Displays authored source files used by the target application.

Option	Description
`stack`	Displays the backtrace of the stack.
`swfs`	Displays all current SWF files.
`targets`	Displays the HTTP or file URL of the target application.
`variables`	Displays all global and static variable names.

For additional information about these options, use the `help` command, as the following example shows:

```
(fdb) help info targets
```

Handling faults

Use the `handle` command to specify how fdb reacts to Flash Player exceptions during execution. To view the current settings, use the `info` command, as the following example shows:

```
(fdb) info handle
```

The `handle` command has the following syntax:

```
(fdb) handle exception [action]
```

The *`fault_type`* is the category of fault that fdb handles. The *`action`* is what fdb does in response to that fault. The possible actions are `print`, `noprint`, `stop`, and `nostop`. The following table describes these actions:

Action	Description
`print`	Prints a message if this type of fault occurs.
`noprint`	Does not print a message if this type of fault occurs.
`stop`	Stops execution of the debugger if this type of fault occurs.
`nostop`	Does not stop execution of the debugger if this type of fault occurs.

Getting help

Use the `help` command to get information on particular topics. The `help` command has the following syntax:

```
help [topic]
```

The `help` command provides a relatively terse description of each command and its usage. The following example opens the `help` command:

```
(fdb) help
```

Type **help** followed by the command name to get the full help information, as the following example shows:

```
(fdb) help delete
```

Terminating the session

You use the `kill` and `exit` commands to end the current debugging session and exit from the fdb application. The `kill` and `exit` commands do not take any arguments. If fdb opened the default browser, you can also terminate the fdb session by closing the browser window.

To stop the current session, use the `kill` command; for example:

```
(fdb) kill
```

Using the `kill` command does not quit the fdb application. You can immediately start another session. To exit from fdb, use the `exit` command; for example:

```
(fdb) exit
```

CHAPTER 13

Using ASDoc

13

ASDoc is a command-line tool that you can use to create API language reference documentation as HTML pages from the classes in your Flex application. The Adobe Flex team uses the ASDoc tool to generate the *Adobe Flex 2 Language Reference*.

Contents

About the ASDoc tool 289
Creating ASDoc comments 290
Documenting ActionScript elements. 295
Documenting MXML files. 301
ASDoc tags 301
Running the ASDoc tool 307

About the ASDoc tool

The ASDoc tool parses one or more ActionScript class definitions and MXML files, and generates API language reference documentation for all public and protected methods and properties, and for all `[Bindable]`, `[DefaultProperty]`, `[Event]`, `[Style]`, and `[Effect]` metadata tags.

You can specify a single class, multiple classes, an entire namespace, or a combination of these inputs as inputs to the ASDoc tool.

ASDoc generates its output as a directory structure of HTML files that matches the package structure of the input class files. Also, ASDoc generates an index of all public and protected methods and properties. To view the ASDoc output, you open the index.html file in the top-level directory of the output.

Invoking the ASDoc tool

To invoke ASDoc, invoke the `asdoc` utility from the bin directory of your Flex installation. For example, from the bin directory, enter the following command to create output for the Flex Button class:

```
asdoc -source-path C:\flex\frameworks\source
  -doc-classes mx.controls.Button
  -main-title "Flex API Documentation"
  -window-title "Flex API Documentation"
  -output flex-framework-asdoc
```

In this example, the source code for the Button class is in the directory C:\flex\frameworks\source\mx\controls. The output is written to C:\flex\bin\flex-framework-asdoc directory.

To view the output, open the file C:\flex\bin\flex-framework-asdoc\index.html. For more information on running the `asdoc` command, see "Running the ASDoc tool" on page 307.

Creating ASDoc comments

A standard programing practice is to include comments in source code. The ASDoc tool recognizes a specific type of comment in your source code and copies that comment to the generated output. This section describes the formatting and parsing rules for comments recognized by the ASDoc tool.

Writing an ASDoc comment

An ASDoc comment consists of the text between the characters `/**` that mark the beginning of the ASDoc comment, and the characters `*/` that mark the end of it. The text in a comment can continue onto multiple lines.

Use the following format for an ASDoc comment:

```
/**
* Main comment text.
*
* @tag Tag text.
*/
```

As a best practice, prefix each line of an ASDoc comment with an asterisk (*) character, followed by a single white space to make the comment more readable in the ActionScript or MXML file, and to ensure correct parsing of comments. When the ASDoc tool parses a comment, the leading asterisk and white space characters on each line are discarded; blanks and tabs preceding the initial asterisk are also discarded.

The ASDoc comment in the previous example creates a single-paragraph description in the output. To add additional comment paragraphs, enclose each subsequent paragraph in HTML paragraph tags, `<p></p>`. You must close the `<p>` tag, in accordance with XHTML standards, as the following example shows:

```
/**
* First paragraph of a multiparagraph description.
*
* <p>Second paragraph of the description.</p>
*/
```

All of the classes that ship with Flex contain the ASDoc comments that appear in the *Adobe Flex 2 Language Reference*. For example, view the mx.controls.Button class for examples of ASDoc comments.

Placing ASDoc comments

Place an ASDoc comment immediately before the declaration for a class, interface, constructor, method, property, or metadata tag that you want to document, as the following example shows for the `myMethod()` method:

```
/**
* This is the typical format of a simple
* multiline (single paragraph) main description
* for the myMethod() method, which is declared in
* the ActionScript code below.
* Notice the leading asterisks and single white space
* following each asterisk.
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

The ASDoc tool ignores comments placed in the body of a method and recognizes only one comment per ActionScript statement.

A common mistake is to put an `import` statement between the ASDoc comment for a class and the `class` declaration. Because an ASDoc comment is associated with the next ActionScript statement in the file after the comment, this example associates the comment with the import statement, not the `class` declaration:

```
/**
* This is the class comment for the class MyClass.
*/
import flash.display.*;   // MISTAKE - Do not to put import statement here.
class MyClass {
}
```

Formatting ASDoc comments

The main body of an ASDoc comment begins immediately after the starting characters, `/**`, and continues until the tag section, as the following example shows:

```
/**
* Main comment text continues until the first @ tag.
*
* @tag Tag text.
*/
```

The first sentence of the main description of the ASDoc comment should contain a concise but complete description of the declared entity. The first sentence ends at the first period that is followed by a space, tab, or line terminator.

ASDoc uses the first sentence to populate the summary table at the top of the HTML page for the class. Each type of class element (method, property, event, effect, and style) has a separate summary table in the ASDoc output.

The tag section begins with the first ASDoc tag in the comment, which is defined by the first @ character that begins a line, ignoring leading asterisks, white space, and the leading separator characters, `/**`. The main description cannot continue after the tag section begins.

The text following an ASDoc tag can span multiple lines. You can have any number of tags, where some tags can be repeated, such as the `@param` and `@see` tags, while others cannot.

The following example shows an ASDoc comment that includes a main description and a tag section. Notice the use of white space and leading asterisks to make the comment more readable:

```
/**
* Typical format of a simple multiline comment.
* This text describes the myMethod() method, which is declared below.
*
* @param param1 Describe param1 here.
* @param param2 Describe param2 here.
*
* @return Describe return value here.
*
* @see someOtherMethod
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

For a complete list of the ASDoc tags, see “ASDoc tags” on page 301.

Using the @private tag

By default, the ASDoc tool generates output for all public and protected elements in an ActionScript class, even if you omit the ASDoc comment. To make ASDoc ignore an element, insert an ASDoc comment that contains the `@private` tag anywhere in the comment. The ASDoc comment can contain additional text along with the `@private` tag, which is also excluded from the output.

ASDoc also generates output for all public classes in the list of input classes. You can specify to ignore an entire class by inserting an ASDoc comment that contains the `@private` tag before the class definition. The ASDoc comment can contain additional text along with the `@private` tag, which is also excluded from the output.

Excluding an inherited element

By default, the ASDoc tool copies information and a link for all ActionScript elements inherited by a subclass from a superclass. In some cases, a subclass may not support an inherited element. You can use the `[Exclude]` metadata tag to cause ASDoc to omit the inherited element from the list of inherited elements.

The `[Exclude]` metadata tag has the following syntax:

```
[Exclude(name="elementName", kind="property|method|event|style|effect")]
```

For example, to exclude documentation on the `click` event in the MyButton subclass of the Button class, insert the following `[Exclude]` metadata tag in the MyButton.as file:

```
[Exclude(name="click", kind="event")]
```

Using HTML tags

You must write the text of an ASDoc comment in XHTML-compliant HTML. You can use selected HTML entities and HTML tags to define paragraphs, format text, create lists, and add anchors. For a list of the supported HTML tags, see "Summary of commonly used HTML elements" on page 305.

The following example comment contains HTML tags to format the output:

```
/**
* This is the typical format of a simple multiline comment
* for the myMethod() method.
*
* <p>This is the second paragraph of the main description
* of the <code>myMethod</code> method.
* Notice that you do not use the paragraph tag in the
* first paragraph of the description.</p>
*
```

```
* @param param1 Describe param1 here.
* @param param2 Describe param2 here.
*
* @return A value of <code>true</code> means this;
* <code>false</code> means that.
*
* @see someOtherMethod
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

Using special characters

The ASDoc tool might fail if your source files contain non-UTF-8 characters such as curly quotes. If it does fail, the error messages it displays should refer to a line number in the interim XML file that was created for that class. That can help you track down the location of the special character.

ASDoc passes all HTML tags and tag entities in a comment to the output. Therefore, if you want to use special characters in a comment, you must enter them using HTML code equivalents. For example, to use a less-than (<) or greater-than (>) symbols in a comment, use `<` and `>`. To use the at-sign (@) in a comment, use `&64;`. Otherwise, these characters will be interpreted as literal HTML characters in the output.

Hiding text in ASDoc comments

The ASDoc style sheet contains a class called `hide`, which you use to hide text in an ASDoc comment by setting the class attribute to `hide`. Hidden text does not appear in the ASDoc output, as the following example shows:

```
/**
*  Dispatched when the user presses the Button control.
*  If the <code>autoRepeat</code> property is <code>true</code>,
*  this event is dispatched repeatedly as long as the button stays down.
*
*  <span class="hide">This text is hidden.</span>
*  @eventType mx.events.FlexEvent.BUTTON_DOWN
*/
```

Rules for parsing ASDoc comments

The following rules summarize how ASDoc processes an ActionScript file:

- If an ASDoc comment precedes an ActionScript element, ASDoc copies the comment and code element to the output file.

- If an ActionScript element is not preceded by an ASDoc comment, ASDoc copies the code element to the output file with an empty description.
- If an ASDoc comment contains the `@private` ASDoc tag, the associated ActionScript element and the ASDoc comment are ignored.
- The comment text should always precede any @ tags, otherwise the comment text is interpreted as an argument to an @ tag. The only exception is the `@private` tag, which can appear anywhere in an ASDoc comment.
- HTML tags, such as `<p></p>`, and `<ul></ul>`, in ASDoc comments are passed through to the output.
- HTML tags must use XML style conventions, which means there must be a beginning and ending tag. For example, an `<li>` tag must always be closed by a `</li>` tag.

Documenting ActionScript elements

You can add ASDoc comments to class, property, method, and metadata elements to document ActionScript classes. For more information on documenting MXML files, see "Documenting MXML files" on page 301.

Documenting classes

The ASDoc tool automatically includes all public classes in its output. Place the ASDoc comment for a class just before the `class` declaration, as the following example shows:

```
/**
* The MyButton control is a commonly used rectangular button.
* MyButton controls look like they can be pressed.
* They can have a text label, an icon, or both on their face.
*/
public class MyButton extends UIComponent {
}
```

This comment appears at the top of the HTML page for the associated class.

To configure ASDoc to omit the class from the output, insert an @private tag anywhere in the ASDoc comment, as the following example shows:

```
/**
* @private
* The MyHiddenButton control is for internal use only.
*/
public class MyHiddenButton extends UIComponent {
}
```

Documenting properties

The ASDoc tool automatically includes all public and protected properties in its output. You can document properties that are defined as variables or defined as setter and getter methods.

Documenting properties defined as variables

Place the ASDoc comment for a public or protected property that is defined as a variable just before the `var` declaration, as the following example shows:

```
/**
*  The default label for MyButton.
*
*  @default null
*/
public var myButtonLabel:String;
```

A best practice for a property is to include the `@default` tag to specify the default value of the property. The `@default` tag has the following format:

```
@default value
```

This tag generates the following text in the output for the property:

```
The default value is value.
```

For properties that have a calculated default value, or a complex description, omit the `@default` tag and describe the default value in text.

ActionScript lets you declare multiple properties in a single statement. However, this does not allow for unique documentation for each property. Such a statement can have only one ASDoc comment, which is copied for all properties in the statement. For example, the following documentation comment does not make sense when written as a single declaration and would be better handled as two declarations:

```
/**
 * The horizontal and vertical distances of point (x,y)
 */
public var x, y;      // Avoid this
```

ASDoc generates the following documentation from the preceding code:

```
public var x
  The horizontal and vertical distances of point (x,y)

public var y
  The horizontal and vertical distances of point (x,y)
```

Documenting properties defined by setter and getter methods

Properties that are defined by setter and getter methods are handled in a special way by the ASDoc tool because these elements are used as if they were properties rather than methods. Therefore, ASDoc creates a property definition for an item that is defined by a setter or a getter method.

If you define a setter method and a getter method, insert a single ASDoc comment before the getter, and mark the setter as `@private`. Adobe recommends this practice because usually the getter comes first in the ActionScript file, as the following example shows:

```
/**
* Indicates whether or not the text field is enabled.
*/
public function get html():Boolean {};

/**
* @private
*/
public function set html(value:Boolean):void {};
```

The following rules define how ASDoc handles properties defined by setter and getter methods:

- If you precede a setter or getter method with an ASDoc comment, the comment is included in the output.
- If you define both a setter and a getter method, only a single ASDoc comment is needed – either before the setter or before the getter.
- If you define a setter method and a getter method, insert a single ASDoc comment before the getter, and mark the setter as `@private`.
- You do not have to define the setter method and getter method in any particular order, and they do not have to be consecutive in the source-code file.
- If you define just a getter method, the property is marked as read-only.
- If you define just a setter method, the property is marked as write-only.
- If you define both a public setter and public getter method in a class, and you want to hide them by using the `@private` tag, they both must be marked `@private`.
- If you have only one public setter or getter method in a class, and it is marked `@private`, ASDoc applies normal `@private` rules and omits it from the output.
- A subclass always inherits its visible superclass setter and getter method definitions.

Documenting methods

The ASDoc tool automatically includes all public and protected methods in its output. Place the ASDoc comment for a public or protected method just before the `function` declaration, as the following example shows:

```
/**
* This is the typical format of a simple multiline documentation comment
* for the myMethod() method.
*
* <p>This is the second paragraph of the main description
* of the <code>myMethod</code> method.
* Notice that you do not use the paragraph tag in the
* first paragraph of the description.</p>
*
* @param param1 Describe param1 here.
* @param param2 Describe param2 here.
*
* @return A value of <code>true</code> means this;
* <code>false</code> means that.
*
* @see someOtherMethod
*/
public function myMethod(param1:String, param2:Number):Boolean {}
```

If the method takes an argument, include an `@param` tag for each argument to describe the argument. The order of the `@param` tags in the ASDoc comment should match the order of the arguments to the method. The `@param` tag has the following syntax:

```
@param paramName description
```

Where *paramName* is the name of the argument and *description* is a description of the argument.

If the method returns a value, use the `@return` tag to describe the return value. The `@return` tag has the following syntax:

```
@return description
```

Where *description* describes the return value.

Documenting metadata

Flex uses metadata tags to define elements of a component. ASDoc recognizes these metadata tags and treats them as if there were properties or method definitions. The metadata tags recognized by ASDoc include:

- `[Bindable]`
- `[DefaultProperty]`

- [Effect]
- [Event]
- [Style]

For more information on these metadata tags, see Chapter 5, "Using Metadata Tags in Custom Components," in *Creating and Extending Flex 2 Components*.

Documenting bindable properties

A bindable property is any property that can be used as the source of a data binding expression. To mark a property as bindable, you insert the [Bindable] metadata tag before the property definition, or before the class definition to make all properties defined within the class bindable.

When a property is defined as bindable, ASDoc automatically adds the following line to the output for the property:

```
This property can be used as the source for data binding.
```

For more information on the [Bindable] metadata tag, see Chapter 5, "Using Metadata Tags in Custom Components," in *Creating and Extending Flex 2 Components*.

Documenting default properties

The [DefaultProperty] metadata tag defines the name of the default property of the component when you use the component in an MXML file.

When ASDoc encounters the [DefaultProperty] metadata tag, it automatically adds a line to the class description that specifies the default property. For example, see the List control in *Adobe Flex 2 Language Reference*.

For more information on the [DefaultProperty] metadata tag, see Chapter 5, "Using Metadata Tags in Custom Components," in *Creating and Extending Flex 2 Components*.

Documenting effects, events, and styles

You use metadata tags to add information about effects, events, and styles in a class definition. The [Effect], [Event], and [Style] metadata tags typically appear at the top of the class definition file. To document the metadata tags, insert an ASDoc comment before the metadata tag, as the following example shows:

```
/**
* Defines the name style.
*/
[Style "name"]
```

For events and effects, the metadata tag includes the name of the event class associated with the event or effect. The following example shows an event definition from the Flex mx.controls.Button class:

```
/**
* Dispatched when the user presses the Button control.
* If the <code>autoRepeat</code> property is <code>true</code>,
* this event is dispatched repeatedly as long as the button stays down.
*
* @eventType mx.events.FlexEvent.BUTTON_DOWN
*/
[Event(name="buttonDown", type="mx.events.FlexEvent")]
```

In the ASDoc comment for the `mx.events.FlexEvent.BUTTON_DOWN` constant, you insert a table that defines the values of the `bubbles`, `cancelable`, `target`, and `currentTarget` properties of the Event class, and any additional properties added by a subclass of Event. At the end of the ASDoc comment, you insert the `@eventType` tag so that ASDoc can find the comment, as the following example shows:

```
/**
* The FlexEvent.BUTTON_DOWN constant defines the value of the
* <code>type</code> property of the event object
* for a <code>buttonDown</code> event.
*
* <p>The properties of the event object have the following values:</p>
* <table class=innertable>
* <tr><th>Property</th><th>Value</th></tr>
* ...
* </table>
*
* @eventType buttonDown
*/
 public static const BUTTON_DOWN:String = "buttonDown"
```

The ASDoc tool does several things for this event:

- In the output for the mx.controls.Button class, ASDoc creates a link to the event class that is specified by the `type` argument of the `[Event]` metadata tag.
- ASDoc copies the description of the `mx.events.FlexEvent.BUTTON_DOWN` constant to the description of the `buttonDown` event in the Button class.

For a complete example, see the mx.controls.Button and mx.events.FlexEvent classes.

For more information on the `[Effect]`, `[Event]`, and `[Style]` metadata tags, see Chapter 5, “Using Metadata Tags in Custom Components,” in *Creating and Extending Flex 2 Components*.

Documenting MXML files

You can use the ASDoc tool with MXML files as well as ActionScript files. All ActionScript entities defined in an `<mx:Script>` block, such as properties and methods, appear in the output. Items defined in MXML tags do not appear in the ASDoc output.

Because the format of an ASDoc comment uses ActionScript syntax, you can only insert an ASDoc comment in an `<mx:Script>` block of an MXML file.

MXML files correspond to ActionScript classes where the superclass corresponds to the first tag in the MXML file. For an application file, that tag is the `<mx:Application>` tag and therefore an MXML application file appears in the ASDoc output as a subclass of the Application class.

ASDoc tags

The following table lists the ASDoc tags:

ASDoc tag	Description	Example
`@copy reference`	Copies an ASDoc comment from the referenced location. The main description, `@param`, and `@return` content is copied; other tags are not copied. You typically use the `@copy` tag to copy information from a source class or interface not in the inheritance list of the destination class. If the source class or interface is in the inheritance list, use the `@inheritDoc` tag instead. You can add content to the ASDoc comment before the `@copy` tag. Specify the location by using the same syntax as you do for the `@see` tag. For more information, see "Using the @see tag" on page 304.	`@copy #stop` `@copy MovieClip#stop`
`@default` *`value`*	Specifies the default value for a property, style, or effect. The ASDoc tool automatically creates a sentence in the following form when it encounters an `@default` tag: `The default value is` *`value`*`.`	`@default 0xCCCCCC`

ASDoc tag	Description	Example
@eventType *package.class.CONSTANT* @eventType *String*	Use the first form in a comment for an [Event] metadata tag. It specifies the constant that defines the value of the Event.type property of the event object associated with the event. The ASDoc tool copies the description of the event constant to the referencing class. Use the second form in the comment for the constant definition. It specifies the name of the event associated with the constant. If the tag is omitted, ASDoc cannot copy the constant's comment to a referencing class.	See "Documenting effects, events, and styles" on page 299
@example *exampleText*	Applies style properties, generates a heading, and puts the code example in the correct location. Enclose the code in <listing version="3.0"></listing> tags. Whitespace formatting is preserved and the code is displayed in a gray, horizontally scrolling box.	@example The following code sets the volume level for your sound: <listing version="3.0" > var mySound:Sound = new Sound(); mySound.setVolume(VOL_HIGH); </listing>
@exampleText *string*	Use this tag in an ASDoc comment in an external example file that is referenced by the @example tag. The ASDoc comment must precede the first line of the example, or follow the last line of the example. External example files support one comment before and one comment after example code.	/** * This text does not appear * in the output. * @exampleText But this does. */

ASDoc tag	Description	Example
`@inheritDoc`	Use this tag in the comment of an overridden method or property. It copies the comment from the superclass into the subclass, or from an interface implemented by the subclass. The main ASDoc comment, `@param`, and `@return` content are copied; other tags are not. You can add content to the comment before the `@inheritDoc` tag. When you include this tag, ASdoc uses the following search order: 1. Interfaces implemented by the current class (in no particular order) and all of their base-interfaces. 2. Immediate superclass of current class. 3. Interfaces of immediate superclass and all of their base-interfaces. 4. Repeat steps 2 and 3 until the Object class is reached. You can also use the `@copy` tag, but the `@copy` tag is for copying information from a source class or interface that is not in the inheritance chain of the subclass.	`@inheritDoc`
`@internal` *`text`*	Hides the text attached to the tag in the generated output. The hidden text can be used for internal comments.	`@internal Please do not publicize the undocumented use of the third parameter in this method.`
`@param` *`paramName description`*	Adds a descriptive comment to a method parameter. The *`paramName`* argument must match a parameter definition in the method signature.	`@param fileName The name of the file to load.`

ASDoc tag	Description	Example
`@private`	Exclude the element from the generated output. To omit an entire class, put the `@private` tag in the ASDoc comment for the class; to omit a single class element, put the `@private` tag in the ASDoc comment for the element.	`@private`
`@return` *`description`*	Adds a Returns section to a method description with the specified text. ASDoc automatically determines the data type of the return value.	`@return The translated message.`
`@see` *`reference [displayText]`*	Adds a See Also heading with a link to a class element. For more information, see "Using the @see tag" on page 304.	`@see flash.display.MovieClip`
`@throws` *`package.class.className description`*	Documents an error that a method can throw.	`@throws SecurityError Local untrusted SWFs may not communicate with the Internet.`

Using the @see tag

The `@see` tag lets you create cross-references to elements within a class; to elements in other classes in the same package; and to other packages. You can also cross-reference URLs outside of ASDoc. The `@see` tag has the following syntax:

```
@see referenceOrLink [displayText]
```

where *referenceOrLink* specifies the destination of the link, and *displayText* optionally specifies the link text. The location of the destination of the `@see` tag is determined by the prefix to the *referenceOrLink* attribute:

- *`#`* ASDoc looks in the same class for the link destination.
- *`ClassName`* ASDoc looks in a class in the same package for the link destionation.
- *`PackageName`* ASDoc looks in a different package for the link destionation.
- `global` ASDoc looks in the Top Level package for the link destionation.

The following table shows several examples of the @see tag:

Example	Result
@see "Just a label"	Text string
@see http://www.cnn.com	External website
@see package-detail.html	Local HTML file
@see Array	Top-level class
@see AccessibilityProperties	Class in same package
@see flash.display.TextField	Class in different package
@see Array#length	Property in top level class
@see flash.ui.ContextMenu#customItems	Property in class in different package
@see #updateProperties()	Method in same class as @see tag
@see Array#pop()	Method in top-level class
@see flash.ui.ContextMenu#clone()	Method in class in different package
@see global#Boolean()	Package method in Top Level (global)
@see flash.util.#clearInterval()	Package method in flash.util

Summary of commonly used HTML elements

The following table lists commonly used HTML tags and character codes within ASDoc comments:

Tag or Code	Description
<p>	Starts a new paragraph. You must close <p> tags. Do not use <p> for the first paragraph of a doc comment (the paragraph after the opening /**) or the first paragraph associated with a tag. Use the <p> tag for subsequent; for example: /** * The first sentence of a main description. * * <p>This line starts a new paragraph.</p> * * <p>This line starts a third paragraph.</p> */ ASDoc ignores white space in comments; to add white space for readability in the AS file, do not use the <p> tag but just add blank lines.
class="hide"	Hides text. Use this tag if you want to add documentation to the source file but do not want it to appear in the output.

Tag or Code	Description
`<listing>`	Indicates a program listing (sample code). Use this tag to enclose code snippets that you format as separate paragraphs, in monospace font, and in a gray background to distinguish the code from surrounding text. You must close `<listing>` tags.
`<pre>`	Formats text in monospace font, such as a description of an algorithm or a formula. Do not use ` ` tags at end of line. Use `<listing>` tag for code snippets.
` `	Adds a line break. You must close this tag. Comments for most tags are automatically formatted; you do not generally have to add line breaks. To create additional white space, add a new paragraph instead. This tag may not be supported in the future, so use it only if necessary.
`<ul>, <li>`	Creates a list. You must close these tags.

`<table>`
`<th>`
`<tr>`
`<td>`
	Creates a table. For basic tables that conform to ASDoc style, set the class attribute to `innertable`. Avoid setting any special attributes. Avoid nesting structural items, such as lists, within tables.
	ASDoc uses a standard CSS stylesheet that has definitions for the `<table>`, `<th>`, `<tr>` and `<td>` tags. You must close these tags.
	Use `<th>` for header cells instead of `<td>`, so the headers get formatted correctly.

`<img>`	Inserts an image. To create the correct amount of space around an image, enclose the image reference in `<p></p>` tags. Captions are optional; if you use a caption, make it boldface. You must close the `<img>` tag by ending it with `/>`, as the following example shows: `<img src = "../../images/matrix.jpg" />`
`<code>`	Applies monospace formatting. You must close this tag.
`<strong>`	Applies bold text formatting. You must close this tag.
`<em>`	Applies italic formatting. You must close this tag.
`<`	Less-than operator (<) . Ensure that you include the final semicolon (;).
`>`	Greater-than operator (>). Ensure that you include the final semicolon (;).
`&`	Ampersand (&). Ensure that you include the final semicolon (;).
`~~`	Asterisk (*). Because asterisks are used to delimit comments, ASDoc does not support asterisks within a comment, so use the double tilde (~~) to represent the mathematical operator.
`—`	Em dash.

Tag or Code	Description
`™`	Trademark symbol (™) that is not registered. This character is superscript by default, so do not enclose it in `<sup>` tags.
` `	Nonbreaking space.
`®`	Registered trademark symbol (®). Enclose this character in `<sup>` tags to make it superscript.
`°`	Degree symbol.
`@`	Do not use an @ sign in an ASDoc comment; instead, insert the HTML character code: @.

Running the ASDoc tool

You use the following options to specify the list of classes processed by the `asdoc` command: `doc-classes`, `doc-sources`, `doc-namespaces`. The `doc-classes` and `doc-namespaces` options require you to specify the `source-path` option to specify the root directory of your files.

The most basic example is to specify a class or list of classes using the `doc-classes` option, as the following example shows:

```
asdoc -source-path . -doc-classes comps.GraphingWidget
  comps.GraphingWidgetTwo
```

In this example, the classes must be located at comps\GraphingWidget.as and comps\GraphingWidgetTwo.as, where comps is a subdirectory of the directory from which you run the `asdoc` command. The arguments of the `doc-classes` option use dot notation that corresponds to the package name of the class.

If the classes are not in the current directory, use the `source-path` option to specify that directory. For example, if the two input classes are in the directory C:\flex\class_dir\comps, then use the following command-line to invoke `asdoc`:

```
asdoc -source-path C:\flex\class_dir -doc-classes comps.GraphingWidget
  comps.GraphingWidgetTwo
```

You can also specify the source classes by using the `doc-sources` option. This option causes `asdoc` to recursively search directories. The following command line generates output for all classes in the current directory and its subdirectories:

```
asdoc -source-path . -doc-sources .
```

You can specify a namespace as the input by using the `doc-namespaces` option. The following command line documents all the classes in the core framework:

```
asdoc -source-path frameworks
  -namespace http://framework frameworks/core-framework-manifest.xml
  -doc-namespaces http://framework
```

Excluding classes

All of the classes specified by the `doc-classes`, `doc-sources`, and `doc-namespaces` options are documented, with the following exceptions:

- If you specified the class by using the `exclude-classes` option, the class is not documented.
- If the ASDoc comment for the class contains the `@private` tag, the class is not documented.
- If the class is found in a SWC, the class is not documented.

In the following example, you generate output for all classes in the current directory and its subdirectories, except for the two classes comps\PageWidget and comps\ScreenWidget.as:

```
asdoc -source-path . -doc-sources . -exclude-classes comps.PageWidget
  comps.ScreenWidget
```

Note that the excluded classes are still compiled along with all of the other input classes; only their content in the output is suppressed.

If you set the `exclude-dependencies` option to `true`, dependent classes found when compiling classes are not documented. The default value is `false`, which means any classes that would normally be compiled along with the specified classes are documented.

For example, you specify class A by using the `doc-classes` option. If class A imports class B, both class A and class B are documented.

Options to the asdoc command

The options to the `asdoc` command work the same way that mxmlc and compc options work. For more information on mxmlc and comp, see Chapter 9, “Using the Flex Compilers,” on page 179.

The following table lists the options to the asdoc command:

Option	Description
-doc-classes *path-element [...]*	A list of classes to document. These classes must be in the source path. This is the default option. This option works the same way as does the -include-classes option for the compc component compiler. For more information, see Chapter 9, "Using the component compiler," on page 215.
-doc-namespaces *uri manifest*	A list of URIs whose classes should be documented. The classes must be in the source path. You must include a URI and the location of the manifest file that defines the contents of this namespace. This option works the same way as does the -include-namespaces option for the compc component compiler. For more information, see Chapter 9, "Using the component compiler," on page 215.
-doc-sources *path-element [...]*	A list of files that should be documented. If a directory name is in the list, it is recursively searched. This option works the same way as does the -include-sources option for the compc component compiler. For more information, see Chapter 9, "Using the component compiler," on page 215.
-exclude-classes *string*	A list of classes that should not be documented. You must specify individual class names. Alternatively, if the ASDoc comment for the class contains the @private tag, is not documented.
-exclude-dependencies true|false	Whether all dependencies found by the compiler are documented. If true, the dependencies of the input classes are not documented. The default value is false.
-footer *string*	The text that appears at the bottom of the HTML pages in the output documentation.
-left-frameset-width *int*	An integer that changes the width of the left frameset of the documentation. You can change this size to accommodate the length of your package names. The default value is 210 pixels.
-main-title *"string"*	The text that appears at the top of the HTML pages in the output documentation. The default value is "API Documentation".

Option	Description
`-output string`	The output directory for the generated documentation. The default value is "`asdoc-output`".

`-package name "description"`	The descriptions to use when describing a package in the documentation. You can specify more than one `package` option.
	The following example adds two package descriptions to the output:
	`asdoc -doc-sources my_dir`
	`-output myDoc`
	`-package com.my.business "Contains business classes and interfaces"`
	`-package com.my.commands "Contains command base classes and interfaces"`

`-templates-path string`	The path to the ASDoc template directory. The default is the asdoc/templates directory in the ASDoc installation directory. This directory contains all the HTML, CSS, XSL, and image files used for generating the output.
`-window-title "string"`	The text that appears in the browser window in the output documentation. The default value is "`API Documentation`".

The `asdoc` command also recognizes the following options from the compc component compiler:

- `-source-path`
- `-library-path`
- `-namespace`
- `-load-config`
- `-actionscript-file-encoding`
- `-help`
- `-advanced`
- `-benchmark`
- `-strict`
- `-warning`

For more information, see "Using the application compiler" on page 195. All other application compiler options are accepted but ignored so that you can use the same command-lines and configuration files for the ASDoc tool that you can use for mxmlc and compc.

CHAPTER 14

Creating Applications for Testing

14

You can create applications and components that can be tested with automated testing tools such as Mercury QuickTest Professional (QTP). This topic includes information intended for Flex developers who write applications that are then tested by Quality Control (QC) professionals who use these testing tools. For information on installing and running the Flex plug-in with QTP, QC professionals should see *Testing Flex Applications with Mercury QuickTest Professional*.

Contents

Tasks and techniques for testable applications overview 311
Compiling applications for testing 312
Creating testable applications 317
Writing the wrapper 320
Understanding the automation framework 321
Instrumenting events 324
Instrumenting custom components 328
Instrumenting composite components 337
Example: Instrumenting the RandomWalk custom component 339

Tasks and techniques for testable applications overview

Flex developers should review the information about tasks and techniques for creating testable applications, and then update their Flex applications accordingly. QC testing professionals who use QTP should use the documentation provided in the separate book, *Testing Flex Applications with Mercury QuickTest Professional*. That document is available for download with the Flex plug-in for QTP.

Use the following general steps to create a testable application:

1. Review the guidelines for creating testable applications. For more information, see "Creating testable applications" on page 317.
2. Build a testable application or prepare the application to use automation at run time.
   - To build a testable application, you include automation libraries at compile time. Compile the application's SWF file with the automation.swc, automation_agent.swc, and qtp.swc files specified in the compiler's `include-libraries` option. If your application uses charts, you must also add the automation_charts.swc file. For information on the compilation process, see "Compiling applications for testing" on page 312.
   - To use automation at run time, you create a wrapper SWF file that is compiled with the automation libraries. In this wrapper SWF file, you use the SWFLoader to load the SWF file that you plan to test only at run time. For more information, see "Using run-time loading" on page 314.
3. Create an HTML wrapper with proper object naming. For more information, see "Writing the wrapper" on page 320.
4. Prepare customized components for testing. If you have custom components that extend UIComponent, make them testable. For more information, see "Instrumenting custom components" on page 328.
5. Deploy the application's assets to a web server. Assets can include the SWF file; HTML wrapper; external assets such as theme files, graphics, and video files; module SWF files; and run-time shared libraries (RSLs). The QC professional must be able to access the main application. For more information about QTP, see *Testing Flex Applications with Mercury QuickTest Professional*.

Compiling applications for testing

You must precompile applications that you plan to test. The functional testing classes are embedded in the application at compile time, and the application has no external dependencies for automated testing at run time.

When you embed functional testing classes in your application SWF file at compile time, you increase the size of the SWF file. If the size of the SWF file is not important, you can use the same SWF file for functional testing and deployment. If the size of the SWF file is important, you typically generate two SWF files: one with functional testing classes embedded and one without.

When you precompile the Flex application for testing, you must reference the automation.swc, automation_agent.swc, and qtp.swc files in your `include-libraries` compiler option. If your application uses charts, you must also add the automation_charts.swc file. When you create the final release version of your Flex application, you recompile the application without the references to these SWC files. For more information about using the automation SWC files, see the Flex Automation Release Notes.

If you do not deploy your application to a server, but instead request it by using the file protocol or run it from within Adobe Flex Builder, you must put the SWF file into the local-trusted sandbox. This requires configuration information that is separate from the SWF file and the wrapper. For more information that is specific to QTP, see *Testing Flex Applications with Mercury QuickTest Professional*.

To include the automation.swc, automation_agent.swc, and qtp.swc libraries, you can add them to the compiler's configuration file or as a command-line option. For the Adobe Flex 2 SDK, the configuration file is located at *flex_install_dir*/frameworks/flex-config.xml. For Adobe Flex Data Services, this file is located at *flex_install_dir*/flex/WEB-INF/flex/flex-config.xml file. Add the following code to the configuration file:

```
<include-libraries>
  ...
  <library>/libs/automation.swc</library>
  <library>/libs/automation_agent.swc</library>
  <library>/libs/qtp.swc</library>
</include-libraries>
```

You can also specify the location of the automation.swc and qtp.swc files when you use the command-line compiler with the `include-libraries` compiler option. The following example adds automation.swc and qtp.swc files to the application:

```
mxmlc -include-libraries+=../frameworks/libs/automation.swc;../frameworks/
  libs/automation_agent.swc../frameworks/libs/qtp.swc MyApp.mxml
```

If your application uses charts, you must also add the automation_charts.swc file to the `include-libraries` compiler option.

Explicitly setting the `include-libraries` option on the command line overwrites, rather than appends, the existing libraries. As a result, if you add the automation.swc, automation_agent.swc, and qtp.swc files by using the `include-libraries` option on the command line, ensure that you use the += operator. This appends rather than overwrites the existing libraries that are included.

To add automated testing support to a Flex Builder project, you also add the automation.swc, automation_agent.swc, and qtp.swc files to the `include-libraries` compiler option.

To add the SWC files to the include-libraries compiler option in Flex Builder:

1. In Flex Builder, select Project > Properties. The Properties dialog box appears.
2. Select Flex Compiler.
3. In the Additional compiler arguments field, enter the following command, and click OK.

   ```
   -include-libraries "Flex SDK 2\frameworks\libs\automation.swc" "Flex SDK
     2\frameworks\libs\automation_agent.swc" "Flex SDK
     2\frameworks\libs\qtp.swc"
   ```

   In Flex Builder, the `include-libraries` compiler option is relative to the Flex Builder installation directory; the default location of this directory on Windows is C:\Program Files\Adobe\Flex Builder 2\.

   If your application uses charts, you must also add the automation_charts.swc file.

Using run-time loading

You can load support for automated testing at run time. This lets you test SWF files that are compiled without automated testing support. To do this, you create a SWF file that *does* load automated testing. In that new SWF file, you use the SWFLoader class to load the other SWF file. The result is that you can test the target SWF file in a testing tool such as QTP, even though the SWF file was not compiled with automated testing support.

To use run-time loading of automated testing support:

1. Create an MXML file with following code:

   ```
   <?xml version="1.0"?>
   <mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
     <mx:SWFLoader source="filename.swf"  width="100%" height="100%"/>
   </mx:Application>
   ```

2. Replace *filename.swf* with the name of your application SWF file that you plan to test. The loaded SWF file does not have to be compiled with automated testing support. It could be any application SWF file that was compiled with Adobe Flex 2.0.1.
3. Save this file as atTemplate.mxml.
4. Compile the atTemplate.mxml file and generate an HTML wrapper for it. Ensure that you include automated testing support.
5. Instruct the QC professional to record tests by requesting the atTemplate.html file.

You are not required to hard code the name of the SWF file to test. Instead, you can pass it dynamically by using a query string parameter. In the following example, you pass the relative path of the SWF file that you plan to load and test by using the `automationswfurl` parameter:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- at/RunTimeLoader.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" layout="absolute"
creationComplete="actionScriptFunction()">
    <mx:Script>
        <![CDATA[
        import flash.external.*;
        public function init():void {
            myLoader.addEventListener(IOErrorEvent.IO_ERROR,
                ioErrorHandler);
        }
        private function ioErrorHandler(event:IOErrorEvent):void {
            trace("ioErrorHandler: " + event);
        }
        public function actionScriptFunction():void {
            init();
            myLoader.source =
                Application.application.parameters.automationswfurl;
        }
        ]]>
    </mx:Script>
    <mx:SWFLoader id="myLoader" width="100%" height="100%"/>
</mx:Application>
```

The following example shows a URL that you could use to request the application:

```
http://localhost/automation.html?automationswfurl=../applications/myapp.swf
```

To use this example, you must convert the query string parameters to a `flashVars` variable in the HTML wrapper. The Adobe Flex Data Services server does this automatically for you if you request an MXML file; however, you can also do convert the query string parameters with any scripting language such as JSP, ASP.Net, or client-side JavaScript.

The following sample HTML wrapper uses JavaScript to convert the `automationswfurl` query string parameter to a `flashVars` variable:

```
<!-- saved from url=(0014)about:internet -->
<html lang="en"><head>
  <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
  <title>Runtime Loading Sample</title>
</head>
<body scroll="no" top="0" left="0" >
  <script language='javascript' charset='utf-8'>
  function getQueryVariable(variable) {
    var atquery = window.location.search.substring(1);
    var atstr = atquery.split("&");
```

```
    for (var i=0;i<atstr.length;i++) {
      var atvar = atstr[i].split("=");
      if (atvar[0] == variable) {
        return atvar[1];
      }
    }
  }

  document.write('<object classid="clsid:D27CDB6E-AE6D-11cf-96B8
    -444553540000" id="automationExample" width="90%" height="90%
    codebase="http://fpdownload.macromedia.com/get/flashplayer/current
    /swflash.cab">');
  document.write('<param name="movie" value="automation.swf"/>');
  document.write('<param name="quality" value="high"/>');
  document.write('<param name="bgcolor" value="#869ca7"/>');
  document.write('<param name="allowScriptAccess" value="sameDomain"/>');
  document.write('<param name="flashvars" value="automationswfurl=
    '+getQueryVariable("automationswfurl")+'"/>');
  document.write('</object>');
</script>
</body>
</html>
```

For more information, see "Communicating with the Wrapper" in *Flex 2 Developer's Guide*.

Testing applications that load external libraries

Applications that load other SWF file libraries require a special setting for automated testing to function properly. A library that is loaded at run time (including run-time shared libraries (RSLs)) must be loaded into the ApplicationDomain of the loading application. If the SWF file used in the application is loaded in a different application domain, automated testing record and playback will not function properly.

The following example shows a library that is loaded into the same ApplicationDomain:

```
import flash.display.*;
import flash.net.URLRequest;
import flash.system.ApplicationDomain;
import flash.system.LoaderContext;

var ldr:Loader = new Loader();

var urlReq:URLRequest = new URLRequest("RuntimeClasses.swf");
var context:LoaderContext = new LoaderContext();
context.applicationDomain = ApplicationDomain.currentDomain;
loader.load(request, context);
```

Creating testable applications

As a Flex developer, there are some techniques that you can employ to make Flex applications as "test friendly" as possible. One of the most important tasks that you can perform is to make sure that objects ares identifiable in the testing tool's scripts. This means that you should set the value of the `id` property for all controls that are tested, and ensure that you use a meaningful string for that `id` property. If you can use unique IDs for each control, the testing scripts are more readable.

Providing meaningful identification of objects

When working with testing tools such as QTP, a QC professional only sees the visual representation of objects in your application. A QC professional generally does not have access to the underlying code. When a QC professional records a script, it's very helpful to see IDs that help the tester identify the object clearly. You should take some time to understand how testing tools interpret Flex applications and determine what names to use for the test objects in the test scripts.

In most cases, testing tools use a visual cue, such as the label of a Button control, to identify the control in the script. Sometimes, however, testing tools use the Flex `id` property of an MXML tag to identify an object in the test script; if there is no value for the `id` property, testing tools use other properties, such as the `childIndex` property.

You should give all testable MXML components an ID to ensure that the test script has a unique identifier to use when referring to that Flex control. You should also try to make these identifiers as human-readable as possible to make it easier for a QC professional to identify that object in the testing script. For example, set the `id` property of a Panel container inside a TabNavigator to *submit_panel* rather than *panel1* or *p1*.

In some cases, QTP does not use the `id` property, but it is a good practice to include it to avoid naming collisions or confusion. For more information about how QTP identifies Flex objects, see *Testing Flex Applications with Mercury QuickTest Professional*.

You should set the value of the `automationName` property for all objects that are part of the application's test. The value of this property appears in the testing scripts. Providing a meaningful name makes it easier for QC professionals to identify that object. For more information about using the `automationName` property, see "Setting the automationName property" on page 334.

Avoiding duplication of objects

If you change the Flex component property that is used by QTP as the object name at run time, QTP creates a second object in its object repository.

For example, if you create a Button control without an `automationName` property, but do not initially set the value of its `label` property, and then later set the value of the `label` property, unexpected test results can occur. This is because QTP uses the value of the `label` property of Button controls to identify an object in its object repository. If you later set the value of the `label` property, or change the value of an existing `label` while the QC professional is recording a test, QTP creates an object in the repository.

So, you should try to understand what properties are used to identify objects in QTP, and try to avoid changing those properties at run time. You should set unique, human-readable `id` or `automationName` properties for all objects that are included in the recorded script.

Coding containers

Containers are different from other kinds of controls because they are used both to record user interactions (such as when a user moves to the next pane in an Accordion container) and to provide unique locations for controls in the testing scripts.

Adding and removing containers from the automation hierarchy

In general, the automated testing feature reduces the amount of detail about nested containers in its scripts. It removes containers that have no impact on the results of the test or on the identification of the controls from the script. This applies to containers that are used exclusively for layout, such as the HBox, VBox and Canvas containers, except when they are being used in multiple-view navigator containers such as the ViewStack, TabNavigator or Accordion containers. In these cases, they are added to the automation hierarchy to provide navigation.

Many composite components use containers, such as Canvas or VBox, to organize their children. These containers do not have any visible impact on the application. So, you usually exclude these containers from being tested because there is no user interaction and no visual need for their operations to be recordable. By excluding a container from being tested, it does not clutter the test scripts and make them harder to read.

To exclude a container from being recorded (but not exclude its children), set the container's `showInAutomationHierarchy` property to `false`. This property is defined by the UIComponent class, so all containers that subclass UIComponent have this property. Children of containers that are not visible in the hierarchy appear as children of the next highest visible parent.

The default value of the `showInAutomationHierarchy` property depends on the type of container. For containers such as Panel, Accordion, Application, DividedBox, and Form, the default value is `true`; for other containers, such as Canvas, HBox, VBox, and FormItem, the default value is `false`.

The following example forces the VBox containers to be included in the test script's hierarchy:

```
<?xml version="1.0"?>
<!-- at/NestedButton.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Panel title="ComboBox Control Example">
        <mx:HBox id="hb">
            <mx:VBox id="vb1" showInAutomationHierarchy="true">
                <mx:Canvas id="c1">
                    <mx:Button id="b1"
                        automationName="Nested Button 1"
                        label="Click Me"
                    />
                </mx:Canvas>
            </mx:VBox>
            <mx:VBox id="vb2" showInAutomationHierarchy="true">
                <mx:Canvas id="c2">
                    <mx:Button id="b2"
                        automationName="Nested Button 2"
                        label="Click Me 2"
                    />
                </mx:Canvas>
            </mx:VBox>
        </mx:HBox>
    </mx:Panel>
</mx:Application>
```

Working with multiview containers

You should avoid using the same label on multiple tabs in multiview containers, such as TabNavigator and Accordion containers. Although it is possible to use the same labels, this is generally not an acceptable UI design practice and can cause problems with control identification in your testing environment. QTP, for example, uses the `label` properties to identify those views to testers. When two labels are the same, QTP uses different strategies to uniquely identify the tabs, which can result in a confusing name list.

Also, dynamically adding children to multiview containers can cause delays that might confuse the testing tool. You should try to avoid this.

Writing the wrapper

In most cases, the testing tool requests a file from a web server that embeds the Flex application. This file, known as the *wrapper*, is often written in HTML, but can also be a JSP, ASP, or other file that browsers interpret. You can request the SWF file directly in the testing tool by using the file protocol, but then you must ensure that the SWF file is trusted.

If you are using Adobe Flex Data Services or Flex Builder, you can generate a wrapper automatically. If you are using the Flex Software Development Kit (SDK), you can use the wrapper templates in the *flex_sdk*/html_templates directory to create a wrapper for your application.

When using a wrapper, your wrapper's `<object>` tag must have an `id` attribute, and the value of the `id` attribute can not contain any periods or hyphens. The convention is to set the `id` to match the name of the root MXML file in the application.

When you use Flex Builder to generate a wrapper, the value of the `id` attribute is the name of the root application file. You do not have to make any changes to this attribute.

When you generate a wrapper with the Flex Data Services server, the object tag's `id` attribute is valid. The following example shows the default object tag for a file named MainApp.swf:

```
<object id='MainApp' classid='clsid:D27CDB6E-AE6D-11cf-96B8-444553540000'
  codebase='http://fpdownload.macromedia.com/get/flashplayer/current/
  swflash.cab' height='600' width='600'>
```

You are not required to change the value of the name in the `<embed>` tag because `<embed>` is used by Netscape-based browsers that do not support the testing feature. The `<object>` tag is used by Microsoft Internet Explorer.

Ensure that you check that the object tag's `id` attribute is the same in the `<script>` and the `<noscript>` blocks of the wrapper.

Understanding the automation framework

This section describes the automation interfaces and shows the flow of the automation framework as you initialize, record, and play back an automatable event.

About the automation interfaces

The Flex class hierarchy includes the following interfaces in the mx.automation.* package that enable automation:

Interface	Description
IAutomationClass	Defines the interface for a component class descriptor.
IAutomationEnvironment	Provides information about the objects and properties of automatable components needed for communicating with agents.
IAutomationEventDescriptor	Defines the interface for an event descriptor.
IAutomationManager	Defines the interface expected from an AutomationManager by the automation module.
IAutomationMethodDescriptor	Defines the interface for a method descriptor.
IAutomationObject	Defines the interface for a delegate object implementing automation for a component.
IAutomationObjectHelper	Provides helper methods for the IAutomationObject interface.
IAutomationPropertyDescriptor	Describes a property of a test object.

About the IAutomationObjectHelper

The IAutomationObjectHelper interface helps the components accomplish the following tasks:

- Replay mouse and keyboard events; the helper generates proper sequence of player level mouse and key events.
- Generate AutomationIDPart for a child: AutomationIDPart would be requested by the Automation for representing a component instance to agents.
- Find a child matching a AutomationIDPart: Automation would request the component to locate a child matching the AutomationIDPart supplied by an agent to it.

- Avoid synchronization issues: Agents invoke methods on Automation requesting operations on components in a sequence. Components may not be ready all the time to perform operations.

For example, an agent can invoke comboBox.Open, comboBox.select "Item1" operations in a sequence. Because it takes time for the drop-down list to open and initialize, it is not possible to run the select operation immediately. You can place a wait request during the open operation execution. The wait request should provide a function for automation, which can be invoked to check the ComboBox control's readiness before invoking the next operation.

Automated testing workflow

Before you automate custom components, you might find it helpful to see the order of events during which Flex's automation framework initializes, records, and plays backs events. The following sections show the steps involved.

Automated testing initialization

1. The user launches the Flex application. Automation initialization code associates component delegate classes with component classes. Component delegate classes implement the IAutomationObject interface.
2. AutomationManager is a mixin. Its instance is created in the mixin `init()` method.
3. The SystemManager initializes the application. Component instances and their corresponding delegate instances are created. Delegate instances add event listeners for events of interest.
4. QTPAgent class is a mixin. In its `init()` method, it registers itself for `SystemManager.APPLICATION_COMPLETE` event. On receiving the event, it creates a QTPAdapter object.
5. QTPAdapter sets up the ExternalInterface function map. QTPAdapter loads the QTP Plugin DLLs by creating the ActiveX object to communicate with QTP.
6. The QTPAdapter requests the XML environment information from the plugin and passes it to the AutomationManager.
7. The XML information is stored in a chain of AutomationClass, AutomationMethodDescriptor, and AutomationPropertyDescriptor objects.

Automated testing recording

1. The user clicks the Record button in QTP.
2. QTP calls the `QTPAdapter.beginRecording()` method. QTPAdapter adds a listener for `AutomationRecordEvent.RECORD` from the AutomationManager.
3. The QTPAdapter notifies AutomationManager about this by calling the `beginRecording()` method. The AutomationManager adds a listener for the `AutomationRecordEvent.RECORD` event from the SystemManager.
4. The user interacts with the application. In this example, suppose the user clicks a Button control.
5. The `ButtonDelegate.clickEventHandler()` method dispatches an AutomationRecordEvent event with the `click` event and Button instance as properties.
6. The AutomationManager `record` event handler determines the important properties of the `click` event from XML information. It converts the values into proper type or format. It dispatches the `record` event.
7. The QTPAdapter event handler receives the event. It calls the `AutomationManager.createID()` method to create the AutomationID object of the button. This object provides a structure for object identification.

   The AutomationID structure is an array of AutomationIDParts. An AutomationIDPart is created by using IAutomationObject. (The `UIComponent.id`, `automationName`, `automationValue`, `childIndex`, and `label` properties of the Button control are read and stored in the object. The `label` property is used because the XML information specifies that this property can be used for identification for the Button.)
8. The QTPAdapter uses the `AutomationManager.getParent()` method to get the logical parent of Button. The AutomationIDPart objects of parent controls are collected at each level up to the application level.
9. All these AutomationIDParts are made part of an AutomationID object.
10. The QTPAdapter sends the information in a call to QTP.
11. At this point, QTP might call the `AutomationManager.getProperties()` method to get property values of Button. The property type information and codec that should be used to modify the value format are gotten from the AutomationPropertyDescriptor.
12. User stops recording. This is propagated by a call to the `QTPAdapter.endRecording()` method.

Automated testing playback

1. The user clicks the Playback button in QTP.
2. The `QTPAdapter.findObject()` method is called to determine whether the object on which the event has to be played back can be found. The AutomationID object is built from the XML data received. The `AutomationManager.resolveIDToSingleObject()` method is invoked to see if QTP can find one unique object matching the AutomationID. The `AutomationManager.getChildren()` method is invoked from application level to find the child object. The `IAutomationObject.numAutomationChildren` property and the `IAutomationObject.getAutomationChildAt()` method are used to navigate the application.
3. The `AutomationManager.isSynchronized()` and `AutomationManager.isVisible()` methods are used to ensure that the object is fully initialized and is visible to receive the event.
4. The `QTPAdpater.run()` method is invoked from QTP to play back the event. The `AutomationManager.replayAutomatableEvent()` method is called to replay the event.
5. AutomationMethodDescriptor for the `click` event on the button is used to copy the property values (if any).
6. The `AutomationManager.replayAutomatableEvent()` method invokes the `IAutomationObject.replayAutomatableEvent()` method on the delegate class. The delegate uses the `IAutomationObjectHelper.replayMouseEvent()` method (or one of the other replay methods, such as `replayKeyboardEvent()`) to play back the event.
7. If there are check points recorded in QTP, the `AutomationManager.getProperties()` method is invoked to verify the values.

Instrumenting events

When you extend Flex components that are already instrumented, you do not have to change anything to ensure that those components' events can be recorded by a testing tool. For example, if you extend a Button class, the class still dispatches the automation events when the Button is clicked, unless you override the Button control's default event dispatching behavior.

Automation events (known in QTP as *operations*) are not the same as Flex events. Flex must dispatch an automation event as a separate action. Flex dispatches them at the same time as Flex events, and uses the same event classes, but you must decide whether to make a Flex event visible to QTP.

Not all events on a control are instrumented. You can instrument additional events by using the instructions in "Instrumenting existing events" on page 325.

If you change the instrumentation of a component, you must edit that component's entry in the TEAFlex.xml file. This is described in "Using the class definitions file" on page 330.

Instrumenting existing events

Events have different levels of relevance for the QC professional. For example, a QC professional is generally interested in recording and playing back a `click` event on a Button control. The QC professional is not generally interested in recording all the events that occur when a user clicks the Button, such as the `mouseOver`, `mouseDown`, `mouseUp`, and `mouseOut` events. For this reason, when a tester clicks on a Button control with the mouse, testing tools only record and play back the `click` event for the Button control and not the other lower-level events.

There are some circumstances where you would want to record events that are normally ignored by the testing tool. But the testing tool's object model only records events that represent the end-user's gesture (such as a click or a drag and drop). This makes a script more readable and it also makes the script robust enough so that it does not fail if you change the application slightly. So, you should carefully consider whether you add a new event to be tested or you can rely on events in the existing object model.

You can see a list of events that QTP can record for each Flex component in the *QTP Object Type Information* document. The Button control, for example, supports the following operations:

- `ChangeFocus`
- `Click`
- `MouseMove`
- `SetFocus`
- `Type`

All of these events except for `MouseMove` are automatically recorded by QTP by default. The QC professional must explicitly add the `MouseMove` event to their QTP script for QTP to play back the event.

However, you can alter the behavior your application so that this event is recorded by the testing tool. To add a new event to be tested, you override the `replayAutomatableEvent()` method of the IAutomationObject interface. Because UIComponent implements this interface, all subclasses of UIComponent (which include all visible Flex controls) can override this method. To override the `replayAutomatableEvent()` method, you create a custom class, and override the method in that class.

The `replayAutomatableEvent()` method has the following signature:

```
public function replayAutomatableEvent(event:Event):Boolean
```

The *`event`* argument is the Event object that is being dispatched. In general, you pass the Event object that triggered the event. Where possible, you pass the specific event, such as a MouseEvent, rather than the generic Event object.

The following example shows a custom Button control that overrides the `replayAutomatableEvent()` method. This method checks for the `mouseMove` event and calls the `replayMouseEvent()` method if it finds that event. Otherwise, it calls its superclass's `replayAutomatableEvent()` method.

```
<?xml version="1.0" encoding="utf-8"?>
<!-- at/CustomButton.mxml -->
<mx:Button xmlns:mx="http://www.adobe.com/2006/mxml">
        <mx:Script>
        <![CDATA[
            import flash.events.Event;
            import flash.events.MouseEvent;
            import mx.automation.Automation;
            import mx.automation.IAutomationObjectHelper;

            override public function
                replayAutomatableEvent(event:Event):Boolean {

                trace('in replayAutomatableEvent()');

                var help:IAutomationObjectHelper =
                    Automation.automationObjectHelper;

                if (event is MouseEvent &&
                    event.type == MouseEvent.MOUSE_MOVE) {
                    return help.replayMouseEvent(this, MouseEvent(event));
                } else {
                  return super.replayAutomatableEvent(event);
                }
            }
        ]]>
    </mx:Script>
</mx:Button>
```

In the application, you call the AutomationManager's `recordAutomatableEvent()` method when the user moves the mouse over the button. The following application uses this custom class:

```
<?xml version="1.0" encoding="utf-8"?>
<!-- at/ButtonApp.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml" xmlns:ns1="*"
initialize="doInit()">
    <mx:Script>
    <![CDATA[
        import mx.automation.*;

        public function doInit():void {
            b1.addEventListener(MouseEvent.MOUSE_MOVE,
                dispatchLowLevelEvent);
        }

        public function dispatchLowLevelEvent(e:MouseEvent):void {
            var help:IAutomationManager = Automation.automationManager;
            help.recordAutomatableEvent(b1,e,false);
        }
    ]]>
    </mx:Script>

    <ns1:CustomButton id="b1"
        toolTip="Mouse moved over"
        label="CustomButton"
    />

</mx:Application>
```

If the event is not one that is currently recordable, therefore, you also must define the new event to QTP in the TEAFlex.xml file. QTP uses this file to define the events, properties, and arguments for each class of test object. For more information, see "Instrumenting events" on page 324. For example, if you wanted to add support for the mouseOver event, you would add the following to the FlexButton's entry in the TEAFlex.xml file:

```
<Operation Name="MouseOver" PropertyType="Method"
  ExposureLevel="CommonUsed">
  <Implementation Class="flash.events::MouseEvent" Type="mouseOver"/>
    <Argument Name="keyModifier" IsMandatory="false" DefaultValue="0">
      <Type VariantType="Enumeration"
        ListOfValuesName="FlexKeyModifierValues"
        Codec="keyModifier"
      />
      <Description>Occurs when the user moves mouse over the
        component</Description>
    </Argument>
</Operation>
```

In the preceding example, however, the `mouseMove` event is already in the FlexButton control's entry in that file, so no editing is necessary. The difference now is that the QC professional does not have to explicitly add the event to their script. After you compile this application and deploy the new TEAFlex.xml file to the QTP testing environment, QTP records the `mouseMove` event for all of the CustomButton objects.

Instrumenting custom components

The process of creating a custom component that supports automated testing is called instrumentation. Flex framework components are instrumented by attaching a delegate class to each component at run time. The *delegate class* defines the methods and properties required to perform instrumentation.

If you extend an existing component that is instrumented, such as a Button control, you inherit its parent's instrumentation, and are not required to do anything else to make that component testable. If you create a component that inherits from UIComponent, you must instrument that class in one of the following ways:

- Create a delegate class that implements the required interfaces.
- Add testing-related code to the component.

You usually instrument components by creating delegate classes. You can also instrument components by adding automation code inside the components, but this is not a recommended practice. It creates tighter coupling between automated testing code and component code, and it forces the automated testing code to be included in a production SWF file.

In both methods of instrumenting a component, you must add your new component's information to a class definitions XML file so that QTP recognizes that component. For more information about this file, see "Using the class definitions file" on page 330.

Consider the following additional factors when you instrument custom components:

- Composition. When instrumenting components, you must consider whether the component is a simple component or a composite component. Composite components are components made up of several other components. For example, a TitleWindow that contains form elements is a composite component.
- Container hierarchy. You should understand how containers are viewed in the automation hierarchy so that the QC professional can easily test the components. Also, you should be aware that you can manipulate the hierarchy to better suit your application by setting some automation-related properties.

- Automation names. Custom components sometimes have ambiguous or unclear default automation names. The ambiguous name makes it more difficult in QTP to determine what component the QTP script is referring to. Component authors can manually set the value of the `automationName` property for all components except item renderers. For item renderers, use the `automationValue`.

Creating a delegate class

To instrument custom components with a delegate, you must do the following:

- Create a delegate class that implements the required interfaces. In most cases, you extend the UIComponentAutomationImpl class. You can instrument any component that implements IUIComponent.
- Register the delegate class with the AutomationManager.
- Define the component in a class definitions XML file.

The delegate class is a separate class that is not embedded in the component code. This helps to reduce the component class size and also keeps automated testing code out of the final production SWF file. All Flex controls have their own delegate classes. These classes are in the mx.automation.delegates.* package. The class names follow a pattern of *Classname*AutomationImpl. For example, the delegate class for a Button control is mx.automation.delegates.controls.ButtonAutomationImpl.

To instrument with a delegate class:

1. Create a delegate class.
2. Mark the delegate class as a mixin by using the `[Mixin]` metadata keyword.
3. Register the delegate with the AutomationManager by calling the `AutomationManager.registerDelegateClass()` method in the `init()` method. The following code is a simple example:

```
[Mixin]
public class MyCompDelegate {
  public static init(root:DisplayObject):void {
    // Pass the component and delegate class information.
    AutomationManager.registerDelegateClass(MyComp, MyCompDelegate);
  }
}
```

   You pass the custom class and the delegate class to the `registerDelegateClass()` method.

4. Add the following code to your delegate class:
   a. Override the getter for the `automationName` property and define its value. This is the name of the object as it appears in QTP. If you are defining an item renderer, use the `automationValue` property instead.
   b. Override the getter for the `automationValue` property and define its value. This is the value of the object in QTP.
   c. In the constructor, add event listeners for events that QTP records.
   d. Override the `replayAutomatableEvent()` method. The AutomationManager calls this method for replaying events. In this method, return whether the replay was successful. You can use methods of the helper classes to replay common events.

      For examples of delegates, see the source code for the Flex controls in the mx.automation.delegates.* packages.
5. Link the delegate class with the application SWF file in one of these ways:
   - Add the following `includes` compiler option and link in the delegate class:
     ```
     mxmlc -includes MyCompDelegate -- FlexApp.mxml
     ```
   - Build a SWC file for the delegate class by using the compc component compiler:
     ```
     compc -source-path+=. -include-classes MyCompDelegate -output
       MyComp.swc
     ```
     Then include this SWC file with your Flex application by using the following `include-libraries` compiler option:
     ```
     mxmlc -include-libraries MyComp.SWC -- FlexApp.mxml
     ```
     This approach is useful if you have many components and delegate classes and want to include them as a single file.
6. After you compile your Flex application with the new delegate class, you must add the new component to QTP's custom class definition XML file. For more information, see "Using the class definitions file" on page 330.

For an example that shows how to instrument a custom component, see "Example: Instrumenting the RandomWalk custom component" on page 339.

Using the class definitions file

The TEAFlex.xml file contains information about all instrumented Flex components. This file provides information about the components to QTP, including what events can be recorded and played back, the name of the component, and the properties that can be tested.

The TEAFlex.xml file is located in the "*QTP_plugin_install*\Flex 2 Plug-in for Mercury QuickTest Pro" directory. QTP recognizes any file in that directory that matches the pattern TEAFlex*.xml, where * can be any string. This directory also contains a TEAFlexCustom.xml file that you can use as a starting point for adding custom component definitions.

The class definitions file describes instrumented components to QTP with the following basic structure:

```
<TypeInformation>
  <ClassInfo>
    <Description/>
    <Implementation/>
    <TypeInfo>
      <Operation/>
      ...
    </TypeInfo>
    <Properties>
      <Property/>
      ...
    </Properties>
  </ClassInfo>
</TypeInformation>
```

The top level tag is `<TypeInformation>`. You define a new class that uses the `<ClassInfo>` tag, which is a child tag of the `<TypeInformation>` tag. The `<ClassInfo>` tag has child tags that further define the instrumented classes. The following table describes these tags:

Tag	Description
ClassInfo	Defines the class that is instrumented, for example, FlexButton. This is the name that QTP uses for the Button control. Attributes of this tag include `Name`, `GenericTypeID`, `Extends`, and `SupportsTabularData`.
Description	Defines the text that appears in QTP to define the component.
Implementation	Defines the class name, as it is known by the Flex compiler, for example, Button or MyComponent.

Tag	Description
TypeInfo	Defines events for this class. Each event is defined in an `<Operation>` child tag, which has two child tags: • The `<Implementation>` child tag associates the operation with the actual event. • Each operation can also define properties of the event object by using an `<Argument>` child tag.
Properties	Defines properties of the class. Each property is defined in a `<Property>` child tag. Inside this tag, you define the property's type, name, and description. For each `Property`, if the `ForDescription` attribute is `true`, the property is used to uniquely identify a component instance in QTP; for example, the `label` property of a Button control. QTP lists this property as part of the object in QTP object repository. If the `ForVerfication` attribute is `true`, the property is visible in the properties dialog box in QTP. If the `ForDefaultVerification` tag is `true`, the property appears selected by default in the dialog box in QTP. This results in verification of the property value in the checkpoint.

The following example adds a new component, MyComponent, to the class definition file. This component has one instrumented event, click:

```
<TypeInformation xsi:noNamespaceSchemaLocation="ClassesDefintions.xsd"
  Priority="0" PackageName="TEA" Load="true" id="Flex" xmlns:xsi="http://
  www.w3.org/2001/XMLSchema-instance">
  <ClassInfo Name="MyComponent" GenericTypeID="mycomponent"
    Extends="FlexObject" SupportsTabularData="false">
    <Description>FlexMyComponent</Description>
    <Implementation Class="MyComponent"/>
    <TypeInfo>
      <Operation Name="Select" PropertyType="Method"
        ExposureLevel="CommonUsed">
        <Implementation Class="myComponentClasses::MyComponentEvent"
          Type="click"/>
      </Operation>
    </TypeInfo>
    <Properties>
      <Property Name="automationClassName" ForDescription="true">
        <Type VariantType="String"/>
        <Description>This is MyComponent.</Description>
      </Property>
      <Property Name="automationName" ForDescription="true">
        <Type VariantType="String"/>
        <Description>The name used by QTP to id an object.</Description>
      </Property>
      <Property Name="className" ForDescription="true">
```

```
            <Type VariantType="String"/>
            <Description>To be written.</Description>
        </Property>
        <Property Name="id" ForDescription="true" ForVerification="true">
            <Type VariantType="String"/>
            <Description>Developer-assigned ID.</Description>
        </Property>
        <Property Name="index" ForDescription="true">
            <Type VariantType="String"/>
            <Description>The index relative to its parent.</Description>
        </Property>
    </Properties>
  </ClassInfo>
  ...
</TypeInformation>
```

You can edit the class definitions file to add a new recordable event to an existing component. To do this, you insert a new `<Operation>` in the control's `<TypeInfo>` block. This includes the implementation class of the event, and any arguments that the event might take.

The following example adds a new event, `MouseOver`, with several arguments to the Button control:

```
<TypeInfo>
  <Operation ExposureLevel="CommonUsed" Name="MouseOver"
  PropertyType="Method">
    <Implementation Class="flash.events::MouseEvent" Type="mouseOver"/>
    <Argument Name="inputType" IsMandatory="false"
      DefaultValue="mouse">
      <Type VariantType="String"/>
    </Argument>
    <Argument Name="shiftKey" IsMandatory="false" DefaultValue="false">
      <Type VariantType="Boolean"/>
    </Argument>
    <Argument Name="ctrlKey" IsMandatory="false" DefaultValue="false">
      <Type VariantType="Boolean"/>
    </Argument>
    <Argument Name="altKey" IsMandatory="false" DefaultValue="false">
      <Type VariantType="Boolean"/>
    </Argument>
  </Operation>
</TypeInfo>
```

When you finish editing the class definitions file, you must distribute the new file to QTP users. They must copy this file manually to the "*QTP_plugin_install*\Flex 2 Plug-in for Mercury QuickTest Pro" directory. When you replace the class definitions file in the QTP environment, you must restart QTP.

Setting the automationName property

The `automationName` property defines the name of a component as it appears in testing scripts. The default value of this property varies depending on the type of component. For example, a Button control's `automationName` is the label of the Button control. Sometimes, the `automationName` is the same as the control's `id` property, but this is not always the case.

For some components, Flex sets the value of the `automationName` property to a recognizable attribute of that component. This helps QC professionals recognize that component in their scripts. Because they do not usually have access to the underlying source code of the application, having a control's visible property define that control can be useful. For example, a Button labeled "Process Form Now" appears in the testing scripts as `FlexButton("Process Form Now")`.

If you implement a new component, or derive from an existing component, you might want to override the default value of the `automationName` property. For example, UIComponent sets the value of the `automationName` to the component's `id` property by default, but some components use their own methods of setting its value.

For example, in the Flex Store sample application, containers are used to create the product thumbnails. A container's default `automationName` (it is the same as the container's `id` property) would not be very useful because it is programmatically generated. So in Flex Store, the custom component that generates a product thumbnail explicitly sets the `automationName` to the product name to make testing the application easier.

The following example from the CatalogPanel.mxml custom component sets the value of the `automationName` property to the name of the item as it appears in the catalog. This is much more recognizable than the default automation name.

```
thumbs[i].automationName = catalog[i].name;
```

The following example sets the `automationName` property of the ComboBox control to “Credit Card List”; rather than using the `id` property, the testing tool typically uses “Credit Card List” to identify the ComboBox in its scripts:

```
<?xml version="1.0"?>
<!-- at/SimpleComboBox.mxml -->
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml">
    <mx:Script>
        <![CDATA[
            [Bindable]
            public var cards: Array = [
                {label:"Visa", data:1},
                {label:"MasterCard", data:2},
                {label:"American Express", data:3}
            ];

            [Bindable]
            public var selectedItem:Object;
        ]]>
    </mx:Script>
    <mx:Panel title="ComboBox Control Example">
        <mx:ComboBox id="cb1" dataProvider="{cards}"
            width="150"
            close="selectedItem=ComboBox(event.target).selectedItem"
            automationName="Credit Card List"
        />

        <mx:VBox width="250">
            <mx:Text
                width="200"
                color="blue"
                text="Select a type of credit card."
            />
            <mx:Label text="You selected: {selectedItem.label}"/>
            <mx:Label text="Data: {selectedItem.data}"/>
        </mx:VBox>
    </mx:Panel>
</mx:Application>
```

If you do not set the value of the `automationName` property, the name of an object in a testing tool is sometimes a property that can change while the application runs. If you set the value of the `automationName` property, testing scripts use that value rather than the default value. For example, by default, QTP uses a Button control’s `label` property as the name of the Button in the script. If the label changes, the script can break. You can prevent this from happening by explicitly setting the value of the `automationName` property.

Buttons that have no label, but have an icon, are recorded by their index number. In this case, you should ensure that you set the `automationName` property to something meaningful so that the QC professional can recognize the Button in the script. This might not be necessary if you set the `toolTip` property of the Button because QTP uses that value if there is no label.

After the value of the `automationName` property is set, you should never change the value during the component's life cycle.

For item renderers, use the `automationValue` property rather than the `automationName` property. You do this by overriding the `createAutomationIDPart()` method and returning a new value that you assign to the `automationName` property, as the following example shows:

```
<mx:List xmlns:mx="http://www.adobe.com/2006/mxml">
  <mx:Script>
    <![CDATA[
      import mx.automation.IAutomationObject;
      override public function
        createAutomationIDPart(item:IAutomationObject):Object {
          var id:Object = super.createAutomationIDPart(item);
          id["automationName"] = id["automationIndex"];
          return id;
      }
    ]]>
  </mx:Script>
</mx:List>
```

This technique works for any container or list-like control to add index values to their children. There is no method for a child to specify an index for itself.

Instrumenting composite components

Composite components are custom components made up of two or more components. A common composite component is a form that contains several text fields, labels, and buttons. Composite components can be MXML files or ActionScript classes.

By default, you can record operations on all instrumented child controls of a container. If you have a Button control inside a custom TitleWindow container, the QA professional can record actions on that Button control just like on any Button control. You can, however, create a composite component in which some of the child controls are instrumented and some are not. To prevent the operations of a child component from being recorded, you override the following methods:

- `numAutomationChildren` getter
- `getAutomationChildAt()`

The `numAutomationChildren` property is a read-only property that stores the number of automatable children that a container has. This property is available on all containers that delegate implementation classes. To exclude some children from being automated, you return a number that is less than the total number of children.

The `getAutomatedChildAt()` method returns the child at the specified index. When you override this method, you return null for the unwanted child at the specified index, but return the other children as you normally would.

The following custom composite component is written in ActionScript. It consists of a VBox container with three buttons (OK, Cancel, and Help). You cannot record the operations of the Help button. You can record the operations of the other Button controls, OK and Cancel. The following example sets the values of the OK and Cancel buttons' `automationName` properties. This makes those button controls easier to recognize in the automated testing tool's scripts.

```
// MyVbox.as
package { // Empty package
    import mx.core.UIComponent;
    import mx.containers.VBox;
    import mx.controls.Button;
    import mx.automation.IAutomationObject;
    import mx.automation.delegates.containers.BoxAutomationImpl;

    public class MyVBox extends VBox {
        public var btnOk : Button;
        public var btnHelp : Button;
        public var btnCancel : Button;

        public function MyVBox():void { // Constructor
        }

        override protected function createChildren():void {
            super.createChildren();

            btnOk = new Button();
            btnOk.label = "OK";
            btnOk.automationName = "OK_custom_form";
            addChild(btnOk);

            btnCancel = new Button();
            btnCancel.label = "Cancel";
            btnCancel.automationName = "Cancel_custom_form";
            addChild(btnCancel);

            btnHelp = new Button();
            btnHelp.label = "Help";
            btnHelp.showInAutomationHierarchy = false;
            addChild(btnHelp);
        }

        override public function get numAutomationChildren():int {
            return 2; //instead of 3
        }

        override public function
            getAutomationChildAt(index:int):IAutomationObject {
            switch(index) {
```

```
                case 0:
                    return btnOk;
                case 1:
                    return btnCancel;
            }
            return null;
        }
    } // Class
} // Package
```

To make this solution more portable, you could create a custom Button control and add a property that determines whether a Button should be testable. You could then set the value of this property based on the Button instance (for example, `btnHelp.useInAutomation = false`), and check against it in the overridden `getAutomationChildAt()` method, before returning null or the button instance.

Example: Instrumenting the RandomWalk custom component

The RandomWalk component is an example of a complex custom component. It has custom events and custom itemRenderers. Showing how it is instrumented can be helpful in instrumenting your custom components.

This section describes the RandomWalk custom component. This source code is located at the following location:

http://demo.quietlyscheming.com/RandomWalk/srcview/RandomWalk.zip

Instrumenting the RandomWalk custom component

The first task when instrumenting a custom component is to create a delegate and add the new component to the class definitions XML file.

To instrument the RandomWalk custom component:

1. Create a RandomWalkDelegate class that extends UIComponentAutomationImpl, similar to RandomWalk extending UIComponent. UIComponentAutomationImpl implements the IAutomationObject interface.

2. Mark the delegate class as a mixin with the `[Mixin]` metadata tag; for example:

```
package {

  ...

[Mixin]
public class RandomWalkDelegate extends UIComponentAutomationImpl {
}
}
```

This results in a call to the static `init()` method in the class when the SWF file loads.

3. Add a public static `init()` method to the delegate class, and add the following code to it:

```
public static init(root:DisplayObject):void {
  Automation.registerDelegateClass(RandomWalk, RandomWalkDelegate);
}
```

4. Add a constructor that takes a RandomWalk object as parameter. Add a call to the super constructor and pass the object as the argument:

```
private var walker:RandomWalk
public function RandomWalkDelegate(randomWalk:RandomWalk) {
  super(randomWalk);
  walker = randomWalk;
}
```

5. Update the TEAFlexCustom.xml file so that QTP recognizes the RandomWalk component. Add the text between the `<TypeInformation>` root tags. The TEAFlexCustom.xml file is located in the "*QTP_plugin_install*\Flex 2 Plug-in for Mercury QuickTest Pro" directory. For more information about the TEAFlexCustom.xml file, see "Using the class definitions file" on page 330.

```
<TypeInformation xsi:noNamespaceSchemaLocation="ClassesDefintions.xsd"
  Priority="0" PackageName="TEA" Load="true" id="Flex" xmlns:xsi="http:/
  /www.w3.org/2001/XMLSchema-instance">
  ...
  <ClassInfo Name="FlexRandomWalk" GenericTypeID="randomwalk"
    Extends="FlexObject" SupportsTabularData="false">
    <Description>FlexRandomWalk</Description>
    <Implementation Class="RandomWalk"/>
    <TypeInfo>
    </TypeInfo>
    <Properties>
      <Property Name="automationClassName" ForDescription="true">
        <Type VariantType="String"/>
        <Description>To be written.</Description>
      </Property>
      <Property Name="automationName" ForDescription="true">
        <Type VariantType="String"/>
        <Description>The name used by the automation system to
```

```
        identify an object.</Description>
      </Property>
      <Property Name="className" ForDescription="true">
        <Type VariantType="String"/>
        <Description>To be written.</Description>
      </Property>
      <Property Name="id" ForDescription="true" ForVerification="true">
        <Type VariantType="String"/>
        <Description>Developer-assigned ID.</Description>
      </Property>
      <Property Name="automationIndex" ForDescription="true">
        <Type VariantType="String"/>
        <Description>The object's index relative to its parent.
        </Description>
      </Property>
    </Properties>
  </ClassInfo>
</TypeInformation>
```

This defines the name of the FlexRandomWalk component and its implementing class. It specifies the `automationClassName`, `automationName`, `className`, `id` and `automationIndex` properties as available for identifying a RandomWalk instance in the Flex application. This is possible because RandomWalk derives from UIComponent, and these properties are defined on that parent class.

If your component has a property that you can use to differentiate between component instances, you can also add that property. For example, the `label` property of a Button control, though not unique when the whole application is considered, can be assumed to be unique within a container; therefore, you can use it as an identification property.

Instrumenting RandomWalk events

The next step in instrumenting a custom component is to identify the important events that must be recorded by QTP. The RandomWalk component dispatches a `RandomWalkEvent.ITEM_CLICK` event. Because this event indicates user navigation, it is important and must be recorded.

To instrument the ITEM_CLICK event:

1. Add an event listener for the event in the RandomWalkDelegate constructor:

```
randomWalk.addEventListener(RandomWalkEvent.ITEM_CLICK,
  itemClickHandler)
```

2. Identify the event in the TEAFlexCustom.xml file so that QTP recognizes the event. Add the following text in the `<TypeInfo>` tag in the TEAFlexCustom.xml file:

```
<Operation Name="Select" PropertyType="Method"
  ExposureLevel="CommonUsed">
  <Implementation Class="randomWalkClasses::RandomWalkEvent"
    Type="itemClick"/>
</Operation>
```

   Adding this block to the TEAFlexCustom.xml file names the event as Select and indicates that it is tied to the RandomWalkEvent class, which is available in randomWalkClasses namespace. It also defines the event of type `itemClick`.

When using the RandomWalk component, users click on items. The component records the item label so that QC professionals can easily recognize their action in the QTP script.

The RandomWalkEvent class has only a single property, `item`, that stores the XML node information. In the RandomWalk component's implementation, the RandomWalkRenderer item renderer is used to display the data on the screen. It is derived from the Label control, which is already instrumented. The Label control returns the label text as its `automationName`, which is what you want to record.

To record the label text:

1. Add a new property, `itemRenderer`, to the RandomWalkEvent class.
2. Add the code to initialize this property for the event before dispatching the event; for example:

```
var rEvent:RandomWalkEvent = new
  RandomWalkEvent(RandomWalkEvent.ITEM_CLICK,node);
rEvent.itemRenderer = child as Label;
dispatchEvent(rEvent);
```

3. Add the new `itemRenderer` property as an argument to the Select operation in the TEAFlexCustom.xml file:

```
<Operation Name="Select" PropertyType="Method"
  ExposureLevel="CommonUsed">
  <Implementation Class="randomWalkClasses::RandomWalkEvent"
    Type="itemClick"/>
    <Argument Name="itemRenderer" IsMandatory="true" >
    <Type VariantType="String" Codec="automationObject"/>
    <Description>User-clicked item.</Description>
  </Argument>
</Operation>
```

This code block specifies the type of argument as String and the Codec as automationObject. The Codec returns the `automationName` of the item renderer.

4. In the event handler, call the `recordAutomatableEvent()` method with `event` as the parameter, as the following example shows:

```
private function itemClickHandler(event:RandomWalkEvent):void {
  recordAutomatableEvent(event);
}
```

In some cases, the component does not dispatch any events or the event that was dispatched does not have the information that is required during the recording or playing back of the test script. In these circumstances, you must create an event class with the required properties. In the component, you create an instance of the event and pass it as a parameter to the `recordAutomatableEvent()` method.

For example, the `ListItemSelect` event has been added in automation code and is used by the List automation delegate to record and play back select operations for list items.

To locate the item renderer object, Automation requires some help. Copy the standard implementation for the following methods:

```
override public function
  createAutomationIDPart(child:IAutomationObject):Object {
  var help:IAutomationObjectHelper = Automation.automationObjectHelper;
  return help.helpCreateIDPart(this, child);
}

override public function resolveAutomationIDPart(part:Object):Array {
  var help:IAutomationObjectHelper = Automation.automationObjectHelper;
  return help.helpResolveIDPart(this, part as AutomationIDPart);
}
```

Preparing RandomWalk for playback

Flex components display the data in a data provider after processing and formatting the data. Playback code requires a way to trace back the visual data to the data provider and the component displaying that data. For example, from the visual label of an item in the List, playback code should be able to find the item renderer that shows the element so that the item's click can be played back.

When playing back a RandomWalkEvent, you must identify the item renderer with the `automationName` given. Because the RandomWalk component has many item renderers that are children which are derived from UIComponent subclasses, you must identify them using the `automationName` property.

The IAutomationObject interface already has APIs to support this. The UIComponentAutomationImpl interface provides default implementations for some methods, but you must override some methods to return information specific to the RandomWalk component.

To prepare the RandomWalk component for playback:

1. Instruct QTP how many renderers are being used by the instance of the RandomWalk component. RandomWalk uses an Array of Arrays for all the renderers. Add the following code in RandomWalkDelegate to find the total number of instances:

```
override public function get numAutomationChildren():int {
  var numChildren:int = 0;
  var renderers:Array = walker.getItemRenderers();
  for (var i:int = 0;i< renderers.length;i++) {
    numChildren += renderers[i].length;
  }
  return numChildren;
}
```

2. Access the `itemRenderers` property in the delegate; however, this property is private. So, you must add the following accessor method to the RandomWalk component:

```
public function getItemRenderers():Array {
  return _renderers;
}
```

   Alternatively, you can make the property public or change its namespace.

3. QTP requests each child renderer. Add the following code to determine the exact renderer and return it:

```
override public function getAutomationChildAt(index:int):
  IAutomationObject {
  var numChildren:int = 0;
  var renderers:Array = walker.getItemRenderers();
  for(var i:int = 0; i < renderers.length; i++) {
    if(index >= numChildren) {
      if(i+1 < renderers.length && (numChildren + renderers[i].length)
        <= index) {
        numChildren += renderers[i].length;
        continue;
      }
      var subIndex:int = index - numChildren;
      var instances:Array = renderers[i];
      return (instances[subIndex] as IAutomationObject);
    }
  }
  return null;
}
```

Linking the delegate to an application

There are two options to link the delegate class with the application SWF file. You can use the `includes` compiler option and link to the delegate as follows:

```
mxmlc -includes RandomWalkDelegate FlexApp.mxml
```

You can also build a SWC file for the delegate class. You then include the SWC file with the Flex application by using the `include-libraries` compiler option, as the following code shows:

```
mxmlc -include-libraries RandomWalkAT.SWC -- FlexApp.mxml
```

This approach is useful if you have many components and many delegate classes.

Adjusting event recording

You can compile and record any application that uses a RandomWalk component. While recording, you might notice that the `FlexLabel().Click` operation is recorded in addition to the `Select` operation. Generally, you do not want to record both operations for each user interaction.

In the following example, you stop the `AutomationRecordEvent.RECORD` event from being recorded. Because it is a bubbling event, you can listen to the event from the children, and prevent the event from being recorded.

To prevent an event from being recorded:

1. Add an event handler in the constructor of the delegate, as the following example shows:

   ```
   obj.addEventListener(AutomationRecordEvent.RECORD, labelRecordHandler);
   ```

2. Prevent the recording by calling the `preventDefault()` method or by stopping the propagation of the event:

   ```
   public function labelRecordHandler(event:AutomationRecordEvent):void {
     // if the event is not from the owning component reject it.
     if (event.replayableEvent.target != uiComponent)
       //event.preventDefault(); can also be used.
       event.stopImmediatePropagation();
   }
   ```

   The RandomWalkDelegate class must handle the playback of the RandomWalkEvent event only. Any other event must be handled by the super class implementation.

3. Override the `replayAutomatableEvent()` method and handle the RandomWalkEvent event:

```
override public function replayAutomatableEvent(event:Event):Boolean {
  if (event is RandomWalkEvent) {
  }
  return super.replayAutomatableEvent(event);
}
```

4. (Optional) To replay the RandomWalkEvent, you must replay a click on the item renderer. To use the `replayClick()` method to play back a click on the item renderer, use the following code:

```
override public function replayAutomatableEvent(event:Event):Boolean {
  var help:IAutomationObjectHelper = Automation.automationObjectHelper;
  if (event is RandomWalkEvent) {
    var rEvent:RandomWalkEvent = event as RandomWalkEvent
    help.replayClick(rEvent.itemRenderer);
    return true;
  } else
    return super.replayAutomatableEvent(event);
  }
```

For playing back an event, the Automation.automationObjectHelper class providers some helper methods, the following table describes:

Method	Description
`replayClick()`	Dispatches the mouseDown, mouseClick and mouseUp events on a UIComponent.
`replayMouseEvent()`	Dispatches a particular mouse event on a UIComponent.
`replayKeyDownKeyUp()`	Dispatches a keyboard event with keyCode and key modifiers specified.
`replayKeyboardEvent()`	Dispatches a particular keyboard event on a UIComponent.

For more information, see the documentation for the IAutomationObjectHelper class in the *Adobe Flex 2 Language Reference*.

5. Record and play back your application.

6. To ensure that the view is updated, add the following code after the call to the `replayClick()` method:

```
override public function replayAutomatableEvent(event:Event):Boolean {
  var help:IAutomationObjectHelper = Automation.automationObjectHelper;
  if (event is RandomWalkEvent) {
    var rEvent:RandomWalkEvent = event as RandomWalkEvent
    help.replayClick(rEvent.itemRenderer);
```

```
        (uiComponent as IInvalidating).validateNow();
        return true;
    } else
        return super.replayAutomatableEvent(event);
}
```

Adjustments like this might be required in the delegate to adjust the behavior of the component during playback because QTP does not wait for actions to be completed. The same can also be achieved by adding `Wait` statements in the QTP script.

You should now be able to compile, record, and play back any application with the RandomWalk component.

Adding checkpoints

To add checkpoints on public properties of the component, you add a small description of the property to the component description in the custom class definitions XML file (TEAFlexCustom.xml). You also add a getter for those public properties.

For the `openChildrenCount` property of the RandomWalk component to appear in checkpoints, add the following element as a child of the `<Properties>` tag:

```
<Property Name="openChildrenCount" ForVerification="true"
  ForDefaultVerification="true">
  <Type VariantType="Integer"/>
  <Description>Number of children open currently.</Description>
</Property>
```

When you use a checkpoint operation with a RandomWalk component, QTP displays the `openChildrenCount` property in the Checkpoint dialog box.

Also, add the following getter method to the RandomWalk class:

```
public function get openChildrenCount():int {
  return numAutomationChildren;
}
```

The `numAutomationChildren` property is inherited from the UIComponent class.

PART 3

Deploying Flex Applications

3

This part describes how to deploy Flex applications.

The following topics are included:

Chapter 15: Deploying Flex Applications 351
Chapter 16: Creating a Wrapper 367
Chapter 17: Using Express Install 391

CHAPTER 15

Deploying Flex Applications

15

When you deploy an application, you make the application accessible to your users. The process of deploying an application is dependent on your application, your application requirements, and your deployment environment. For example, the process of deploying an application on an internal website that is only accessible by company employees might be different from the process for deploying the same application on a public website accessible by anyone.

This topic does not attempt to define the exact set of steps that you use for deploying all applications. Instead, it contains an overview of the deployment process, and a general checklist that you might use when you deploy your application.

Contents

About deploying an application 351
Deployment options. 352
Compiling for deployment 358
Deployment checklist 360

About deploying an application

When you deploy an application, you move the application from your development environment to your deployment environment. After you deploy it, customers have full access to the application.

The deployment process that your organization uses might only require you to copy a Flex application's SWF file from your development server to your deployment server. In many organizations however, the deployment process is more complicated, and involves people from groups outside the development organization. For example, you might have an IT department that maintains your corporate website. The IT department might be responsible for staging, testing, and then deploying your application.

Your application architecture might also require you to deploy more than just a single SWF file. For example, your application might access Runtime Shared Libraries (RSLs) or other assets at run time. You must make sure to copy all required files to your deployment environment.

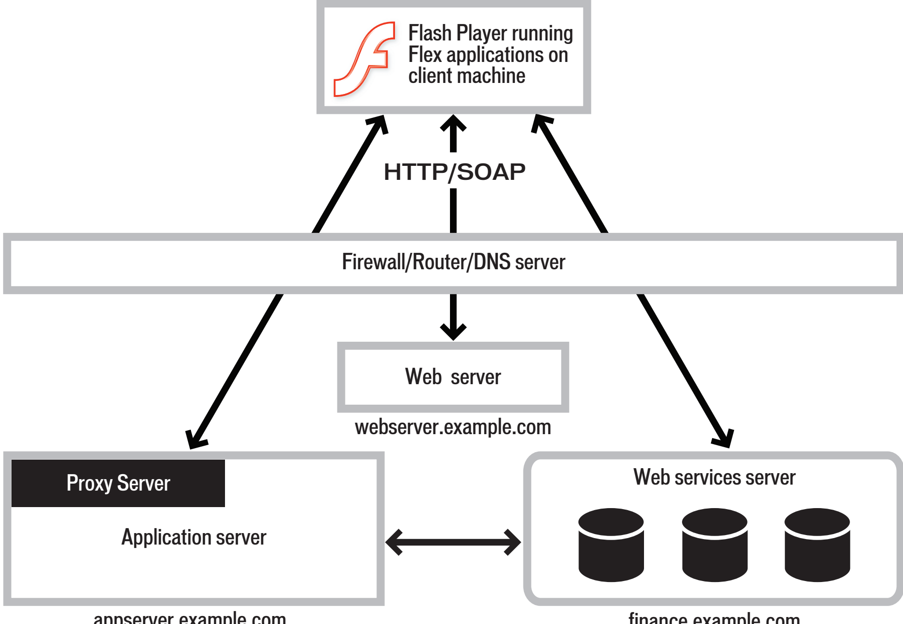
Deployment might also require you to perform operations other than just copying application files to a deployment server. Your application might access data services on your server, or on another server. You must ensure that your data services are accessible by a deployed Flex application that executes on a client's computer.

Deployment options

The process of deploying a Flex application depends on the version of Adobe Flex that you are using. The deployment process is simpler for deploying applications created using Flex 2 SDK than for deploying applications created using Flex Data Services.

Deploying Flex 2 SDK applications

The following example shows a typical deployment environment for a Flex application:



Deploying an application for Flex 2 SDK and Adobe Flex Builder might require you to perform some or all of the following actions:

- Copy the application SWF file to your deployment server. As the previous example shows, you copy the application to webserver.example.com.
- Copy any asset files, such as icons, media files, or other assets, to your deployment server.
- Copy any RSLs to your web server or application server. For more information, see "Deploying RSLs with Flex 2 SDK" on page 353.
- Copy any SWF files required to support Flex features, such as the History Manager. For more information, see "Deploying additional Flex files" on page 353.
- Write a wrapper for the SWF file if you access it from an HTML, JSP, ASP, or another type of page.

  A deployed SWF file can encompass your entire web application, however it is often used as a part of the application. Therefore, users do not typically request the SWF file directly, but request a web page that references the SWF file. Flex Builder and the Flex Data Services web-tier compiler can generate the wrapper for you, or, you can write the wrapper. For more information, see Chapter 16, "Creating a Wrapper," on page 367.
- Create a crossdomain.xml file on the server for data service, if you directly access any data services outside of the domain that serves the SWF file. For more information, see "Accessing data services from a deployed application" on page 354

Deploying RSLs with Flex 2 SDK

When your application uses an RSL, you must make sure to deploy the RSL on your deployment server. You use the `runtime-shared-libraries` option of the Flex compiler to specify the directory location of the RSL at compile time. Ensure that you copy the RSL to the same directory that you specified with `runtime-shared-libraries`. For more information, see Chapter 10, "Using Runtime Shared Libraries," on page 233.

Deploying additional Flex files

The implementation of some Flex features requires that you deploy additional files along with your application's SWF file. For example, if you use the History Manager in your application, you must deploy the history.swf and history.js files along with your application's SWF file. If you use the Flash Player version detection feature, you also must deploy the playerProductInstall.swf file with your SWF file. You typically deploy these files in the same directory as your application's SWF file.

The following table lists the Flex feature that requires additional files that you must deploy to support that feature:

Feature	Files	Comments
History Manager	history.swf, history.js	If you are using Flex Builder or the web-tier compiler, the compiler generates a wrapper for you that references these files. If you are writing your own wrapper, include these files in the wrapper. For more information, see Chapter 16, "Creating a Wrapper," on page 367.
Player detection and deployment	playerProductInstall.swf	If you are using Flex Builder or the web-tier compiler, the compiler generates a wrapper for you that references these files. If you are writing your own wrapper, include these files in the wrapper. For more information, see Chapter 16, "Creating a Wrapper," on page 367.

Accessing data services from a deployed application

In a typical Flex development environment, you build and test your application behind a corporate firewall, where security restrictions are much less strict than when a customer runs the application on their own computer. However, when you deploy the application, it runs on a customers computer outside your firewall. That simple change of location might cause the application to fail if you do not correctly configure your data services to allow external access.

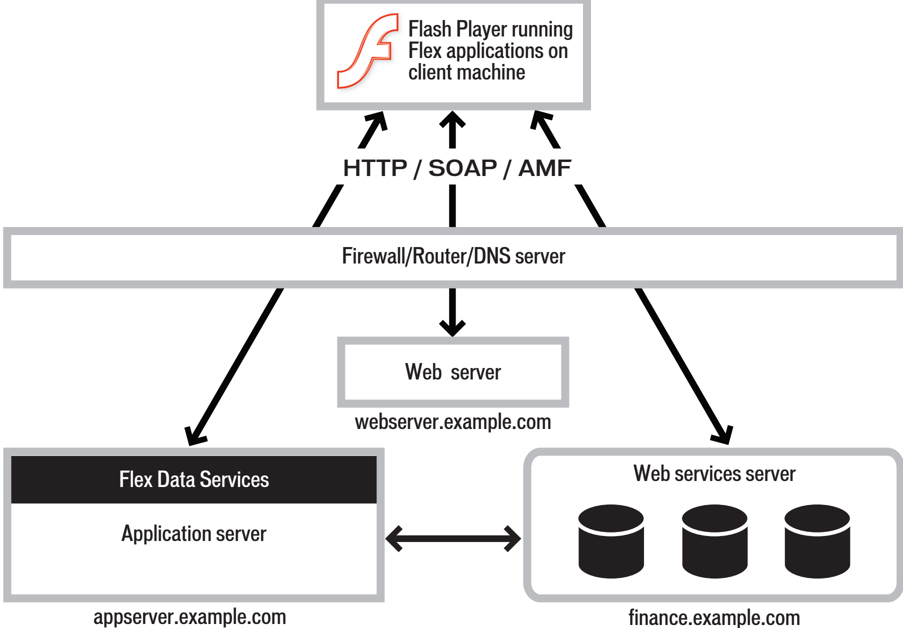
Most run-time accesses to application resources fall into one of the following categories:

- Direct access to asset files on a web server, such as image files.
- Direct access to resources on your J2EE application server.
- Data services requests through a proxy. A proxy redirects that request to the server that handles the data service request.
- Direct access to a data service.

As part of deploying your application, ensure that all run-time data access requests work correctly from the application that is executing outside of your firewall.

Deploying Flex Data Services applications

The following example shows a typical deployment environment for a Flex Data Services application:

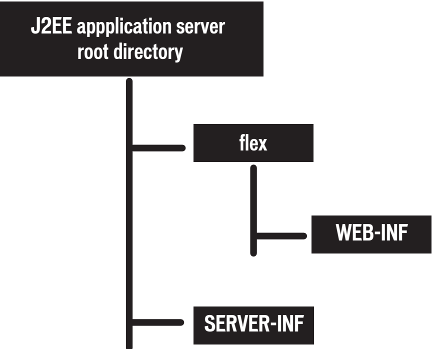


Deploying an application for Flex Data Services requires that you perform the same tasks as you did for Flex 2 SDK, as described in "Deploying Flex 2 SDK applications" on page 352, and perform the following tasks:

- Deploy the Flex Data Services web application on your J2EE application server or servlet container.
- Configure the various services of Flex Data Services, such as the Messaging Service and Proxy Service.

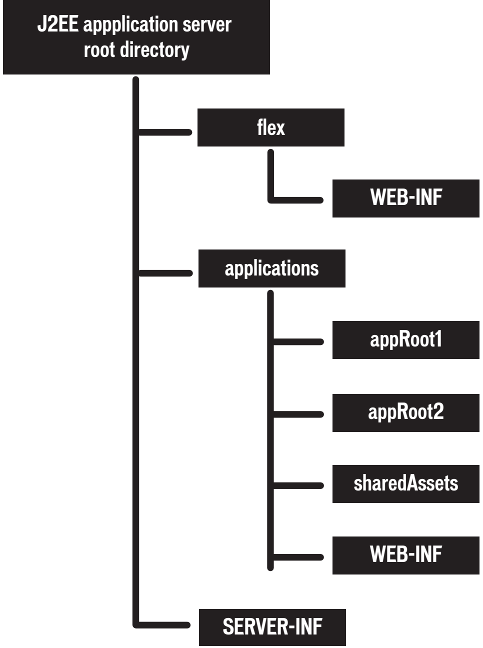
Deploying the Flex Data Services web application

You must deploy the Flex Data Services web application on your deployment server. You can deploy the web application on a J2EE application server or servlet container. The following example shows the directory structure of the Flex Data Services web application:



For a description of this directory structure, see "Flex Data Services installation directory structure" on page 33.

The following example shows a more common approach to web application design. In this example, you deploy your application in its own web application, outside of the Flex Data Services web application:



This configuration has the advantage of isolating your application files from the Flex Data Services web application so that you can package and distribute them separately.

Configuring Flex Data Services

You configure the Flex Proxy Service, Remoting Service, Message Service, and Data Management Service by using the Flex Data Services configuration files. You can copy the configuration files from your development environment to your deployment environment, or you can modify them in the deployment environment.

When you configure Flex Data Services, consider the level of security required by your application. When you are running your application in a development environment, you can use minimal or no security to control access to data services. In a deployment environment, you might decide to restrict access to a privileged group of users by applying a security constraint in a destination definition in the Flex services configuration file. A security constraint ensures that a user is authenticated, by using custom or basic authentication, before they access the destination.

For more information on configuring Flex Data Services, see Chapter 43, “Configuring Data Services,” in *Developing Flex Applications*.

Compiling for deployment

When you create a deployable SWF file, ensure that you compile the application correctly. Typically, you disable certain compiler features, such as the generation of debug output, and enable other options, such as the generation of accessible content.

This section contains an overview of some common compiler options that you might use when you create a deployable SWF file. For a complete list of compiler options, see Chapter 9, “Using the Flex Compilers,” on page 179.

Enabling accessibility

The Flex accessibility option lets you create applications that are accessible to users with disabilities. By default, accessibility is disabled. You enable the accessibility features of Flex components at compile time by using options to a Flex Builder project, setting the `accessible` option to `true` for the command-line compiler, or setting the `<accessible>` tag in the flex-config.xml file to `true`.

If you are using the web-tier compiler of Flex Data Services, you can override the `<accessible>` tag in the flex-config.xml file by appending `?accessible=true` or `?accessible=false` parameters to the query string for your application’s MXML file, as the following example shows:

```
http://mydomain.com/flex/myApp.mxml?accessible=false
```

For more information on creating accessible applications, see Chapter 36, “Creating Accessible Applications,” in *Flex 2 Developer’s Guide*.

Preventing users from viewing your source code

Flex lets you publish your source code with your deployed application. You might want to enable this option during the development process, but disable it for deployment. Or, you might want to include your source code along with your deployed application.

In Flex Builder, you use the Project > Publish Application menu option to specify whether to publish your source code. You can also use the `viewSourceURL` property of the Application class to set the URL of your source code.

Enabling production mode

You enable production mode for Flex Data Services when your application is running live on a public-facing server. Enabling or disabling production mode mainly affects how the web-tier compiler compiles a Flex Data Services application deployed as MXML and ActionScript files.

When production mode is disabled, the default mode, and you deploy your application as MXML and ActionScript files, Flex Data Services automatically recompiles the application on the next request after any source code file is modified. When you enable production mode, applications are not recompiled when source files are modified.

When production mode is enabled, all debugging options to the compiler are set internally to `false`, regardless of any conflicting settings to the compiler. This means the compiler does not generate any debugging information when compiling an application in production mode.

You control production mode by using the `<production-mode>` tag in the flex-webtier-config.xml file. The default value of the `<production-mode>` tag is `false`. To enable production mode, change the value to `true`, as the following example shows:

```
<production-mode>true</production-mode>
```

You must restart Flex Data Services for changes to the production mode to take effect.

Disabling incremental compilation

You can use incremental compilation to decrease the time it takes to compile an application or component library with the Flex application compilers. When incremental compilation is enabled, the compiler inspects changes to the bytecode between revisions and only recompiles the section of bytecode that has changed.

However, if you deploy a Flex Data Services application on a production server as MXML and ActionScript files, you typically compile it on the first access of the application, but do not recompile it on subsequent accesses. Therefore, you can disable incremental compilation by setting the `<incremental>` tag in the flex-config.xml file to `false`. Disabling incremental compilation reduces the amount of memory used by the Flex Data Services web-tier compiler. The `<incremental>` tag is a child tag of the `<compiler>` tag.

For more information, see Chapter 9, "Using the Flex Compilers," on page 179.

Using a headless server

A *headless server* is one that is running UNIX or Linux and often does not have a monitor, keyboard, mouse, or even a graphics card. Headless servers are most commonly encountered in ISPs and ISVs, where available space is at a premium and servers are often mounted in racks. Enabling the headless mode reduces the graphics requirements of the underlying system and can allow for a more efficient use of memory.

If you deploy a Flex application on a headless server, you must set the `headless-server` option of the compiler to `true`. Setting this option to `true` is required to support fonts and SVG images in a nongraphical environment.

If you are using Flex Data Services, you can also enable the headless mode by setting the `<headless-server>` tag in the flex-config.xml file.

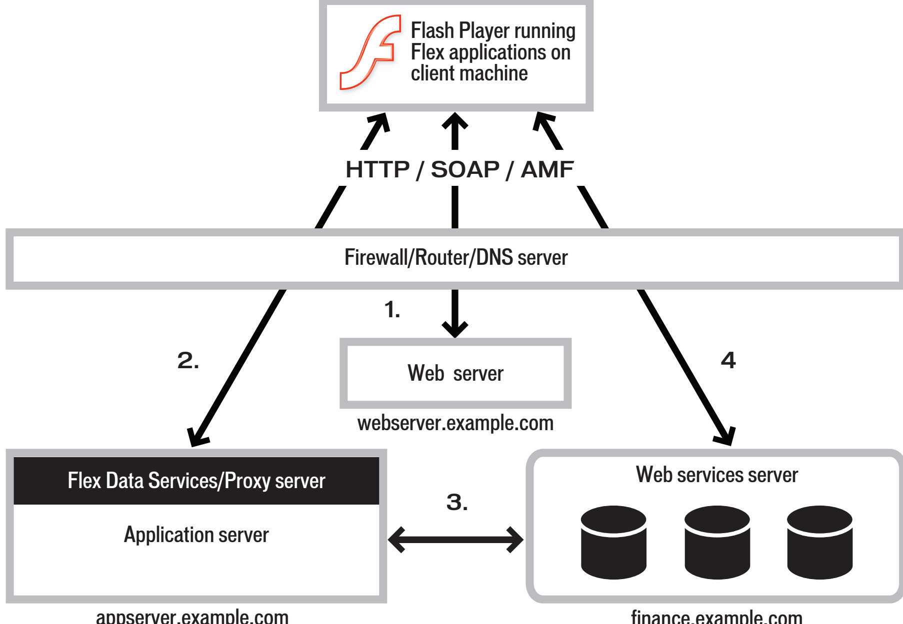
For more information, see "Using headless servers" on page 126.

Deployment checklist

This section contains a checklist of common system configuration issues that customers have found when deploying Flex applications for production. It also contains troubleshooting tips to diagnose common deployment problems.

Types of network access

Deployed Flex applications typically make several types of requests to services within your firewall, as the following example shows:



Most of the deployment issues that customers report are related to network security and routing, and fall into one of the following scenarios:

1. Direct access to resources on a web server, such as image files. In the preceding example, the client directly accesses resources on webserver.example.com.
2. Direct access to resources on your application server. In the preceding example, the client directly accesses resources on appserver.example.com. Ensure that deployed Flex applications can access the appropriate servers.
3. Web services requests through a proxy. A proxy redirects a request to the server that handles the web service. The proxy server can be the Flex Data Services proxy server or your own proxy server. In the preceding example, the client accesses a resource on appserver.example.com, but that request is redirected to finance.example.com. Ensure that you configure the proxy server correctly so that deployed Flex applications can access your web services, or other data services, through the proxy.

4. Direct access of a web service. In the preceding example, the client directly accesses a service on finance.example.com. If a deployed Flex application directly accesses web services, or other data services, ensure that access is allowed.

Step 1. Create a list of server-side resources

Before you start testing your network configuration, make a list of the IP addresses and DNS names of all the servers that a Flex application might access. A Flex application might directly access these servers, for example by using a web service, or another server might access them as part of handling a redirected request.

Enter the information about your servers in the following table:

Name	DNS name	IP address

Flex Data Services includes a web service proxy. Enter information about the server hosting the Flex Data Services web service proxy:

Name	DNS Name	IP Address

Enter information about your web services or any other services accessible from a deployed Flex application:

Name	Location (URL)

Step 2. Verify access from server to server within your firewall

In some cases, an external request to one server can be redirected to another server behind your firewall. A redirected request can occur for a request to a web service or to any file, depending on the system configuration. Where it is necessary, ensure that your servers can communicate with each other so that a redirected request can be properly handled.

To determine if one server, called Server A in this example, can communicate with another server, called Server B, create a temporary file called temp.htm on Server A in its web root directory. Then, log in to Server B and ensure that it can access temp.htm on Server A. Try to access the file by using Server A's DNS name and also its IP address.

Servers can have multiple NIC cards or multiple IP addresses. Ensure that each server can communicate with all of the IP addresses on your other servers.

Also, log in to the server that hosts your web service proxy to make sure that it can access all web services on all other servers. Flex Data Services includes a web service proxy, so log in to the server that hosts Flex Data Services. If your system includes another web service proxy, log in to that server also. You can test the web service proxy by making an HTTP request to the WSDL file for each web service. In the previous example, log in to appserver.example.com and ensure that it can access the WSDL files on finance.example.com.

If any server cannot access the other servers in your system, an external request from a Flex application might also fail. For more information, contact your system administrator.

Step 3. Verify access to your servers from outside the firewall

Some servers might have to be accessed from outside the firewall to handle HTTP, SOAP, or AMF requests from clients. You can use the following methods to determine if a deployed Flex application can access your servers from outside the firewall:

- On each server that can be accessed from outside the firewall, create a temporary file, such as temp.htm, on the server in its web root directory. From a computer outside the firewall, use a browser to make an HTTP request to the temporary file to ensure that an external computer can access it.

  For example, for a file named temp.htm, try accessing it by using the following URL:

  ```
  http://webserver.example.com/server1/temp.htm
  ```

- From a computer outside the firewall, use a browser to make an HTTP request to the WSDL file for each web service that can be accessed from outside the firewall to ensure that the WSDL file can be accessed.

  For example, try accessing the WSDL file for a web service by using the following URL:

  ```
  http://finance.example.com/server1/myWS.wsdl
  ```

You should be able to access the temp.htm file or the WSDL file on all of your servers from outside the firewall. If these requests fail, contact your IT department to determine why the files cannot be accessed.

Step 4. Configure the proxy server

In "Step 3. Verify access to your servers from outside the firewall" on page 364, you ensure that you can directly access your servers and server resources from outside the firewall. When you use the Flex Data Services proxy or your own proxy server to handle requests to data services, you also must ensure that you can access the data services from a deployed Flex application through the proxy.

You typically configure a proxy server with a list of URLs to which the administrator gives access to the proxy. Only the URLs that are allowed by the administrator can pass through the proxy.

With Flex Data Services, you use the configuration files to configure the list of accessible URLs. For another type of proxy server, a different configuration mechanism might be in place. After you configure your proxy server, ensure that the deployed Flex application can access web services and other server-side resources as necessary.

For more information on configuring Flex Data Services, see Chapter 43, "Configuring Data Services," in *Developing Flex Applications*.

Step 5. Create a crossdomain policy file

Your system might be configured to allow a Flex application to directly access server-side resources on different domains or different computers without going through a proxy. These operations fail under the following conditions:

- When the Flex application's SWF file references a URL, and that URL is outside the exact domain of the SWF file that makes the request
- When the Flex application's SWF file references an HTTPS URL, and the SWF file that makes the request is not served over HTTPS

To make a data service or asset available to SWF files in different domains or on different computers, use a crossdomain policy file on the server that hosts the data service or asset. A *crossdomain policy file* is an XML file that provides a way for the server to indicate that its data services and assets are available to SWF files served from certain domains, or from all domains. Any SWF file that is served from a domain specified by the server's policy file is permitted to access a data service or asset from that server. By default, place the crossdomain.xml at the root directory of the server that is serving the data.

For more information on using a cross-domain policy file, see "Using cross-domain policy files" on page 65.

CHAPTER 16

Creating a Wrapper

16

Adobe Flex applications can take the form of a SWF file that you embed in an HTML page by using the `<object>` and `<embed>` tags. The HTML page can also reference an external JavaScript file to embed the Flex application. Collectively, the HTML page and JavaScript file are known as the *wrapper*. This topic describes how to create a wrapper, customize an existing one to include support for history management and Express Install, and how to use the `<object>` and `<embed>` tags.

Contents

About the wrapper 367
Creating a wrapper 370
Adding features to the wrapper 375
About the <object> and <embed> tags 378
Requesting an MXML file without the wrapper 389

About the wrapper

The wrapper is responsible for embedding the Flex application's SWF file in a web page, such as an HTML, ASP, JSP, or Adobe ColdFusion page. In addition, you use the logic in the wrapper to enable history management, Express Install, and to ensure that users both with and without JavaScript enabled in their browsers can access your Flex applications. You can also use the wrapper to pass `flashVars` variables into your Flex applications and use the ExternalInterface API. These topics are described in Chapter 34, "Communicating with the Wrapper," in *Flex 2 Developer's Guide*.

There are several ways to create a wrapper:

- Write a custom wrapper using the instructions in "Creating a wrapper" on page 370.
- Export and customize an HTML wrapper from Flex Builder. For more information, see "About the Flex Builder wrapper" on page 368.
- Generate the HTML wrapper with Flex Data Services. For more information, see "About the wrapper generated by Flex Data Services" on page 368.
- Use the templates provided in the /resources/html-templates directory. For more information, see "About the HTML templates" on page 370.

Flex Builder and the Flex Data Services server generate a wrapper that embeds your Flex application. These wrappers include support for Express Install and history management by default, although you can disable these features or configure them to your specifications. History management lets users navigate the history of their interactions within the Flex application using the browser's Forward and Back buttons. Express Install ensures that your users have a good upgrade experience if their Players require an update.

The mxmlc command-line compiler does not generate a wrapper. You must write it manually using the instructions in "Creating a wrapper" on page 370. You can start out with a simple wrapper that just embeds your Flex application. You can then add history management and Express Install support to your wrapper.

About the Flex Builder wrapper

To view the wrapper generated by Flex Builder, run the current project. Flex Builder generates an HTML page in the same directory as the project's root MXML file. This directory also includes the supporting files such as the history.swf, history.js, history.htm, and AC_OETags.js files.

You can configure the wrapper using the Flex Compiler properties dialog box in Flex Builder. For more information, see the Flex Builder documentation.

About the wrapper generated by Flex Data Services

To view the wrapper generated by Flex Data Services, use your browser to request the MXML file directly. Flex returns an HTML page that embeds references to the compiled version of this file. To access the wrapper, view the page's source in your browser and save it.

Viewing the source in the browser only shows you the final page and not the supporting files that are part of the Flex application's request/response lifecycle. To view the contents of the other files, you can use the JRun sniffer utility. For more information, see "Using the sniffer" on page 411.

You can customize the output of the Flex Data Services wrapper using the settings in the flex-webtier-config.xml file. The following table describes the options:

Option	Description
`use-history-management`	Enables history management support in the wrapper returned by Flex Data Services. The default value is `true`.
`flash-player.use-player-detection`	Enables the insertion of player detection logic into the wrapper that is returned by Flex Data Services. Setting this value to false also disables Express Install. The default value is `true`.
`flash-player.use-express-install`	Enables Express Install support in the wrapper that is returned by Flex Data Services. The default value is `true`.
`flash-player.required-major-version`	Sets the value of the `requiredMajorVersion` variable in the wrapper that is returned by Flex Data Services. The default value is 9.
`flash-player.required-minor-version`	Sets the value of the `requiredMinorVersion` variable in the wrapper that is returned by Flex Data Services. The default value is 0.
`flash-player.required-version-revision`	Sets the value of the `requiredRevision` variable in the wrapper that is returned by Flex Data Services. The default value is 0.
`flash-player.alternate-content-page`	Defines the URL to which a client is redirected if the client does not have the required version of the player and does not install an updated player. This option prints the following line in the alternate content portion of the wrapper: `document.location.replace("`*`your_url_here`*`")` If you define both this option and the `alternate-content-include` option, the `alternate-content-page` option takes precendence.
`flash-player.alternate-content-include`	Defines content to insert in the alternate content location of the wrapper if the client does not have the required version of the player and does not install an updated player.

For more information about Express Install and the variables used by Express Install in the wrapper, see Chapter 17, "Using Express Install," on page 391.

About the HTML templates

Flex Data Services and Flex 2 SDK include a set of HTML templates in the *flex_install_dir*/ resources/html-templates directory. These templates provide a basic wrapper, as well as wrappers that implement various features. The following table describes the templates:

Directory	Description
client-side-detection	Provides scripts that detect the version of the client's player and return alternate content if the client's player does not meet the minimum required version.
client-side-detection-with-history	Provides the same scripts as those in the client-side-detection directory, but adds history management support.
express-installation	Provides scripts that support Express Install.
express-installation-with-history	Provides scripts that support Express Install and history management.
no-player-detection	Provides a basic wrapper that embeds the SWF file by using an external JavaScript file.
no-player-detection-with-history	Provides a basic wrapper with history management support.

For more information about Express Install, see "Adding Express Install to your wrapper" on page 377. For more information about history management, see "Adding history management to your wrapper" on page 377.

Creating a wrapper

You can write your own wrapper for your SWF files rather than use the wrapper generated by Flex Builder or Flex Data Services. Your own wrapper can be simple HTML, or it can be a JavaServer Page (JSP), a ColdFusion page, an Active Server Page (ASP), or anything that can return HTML that is rendered in your client's browser. Typically, you integrate wrapper logic into your website's HTML templates.

This section describes how to write the simplest wrapper possible to get your Flex application running on a web server. It does not include features such as history management and Express Install. These features improve the user experience and should be omitted only upon careful consideration. Instructions for adding these features, which make creating a wrapper considerably more complex, are described in later sections.

Flex Data Services and Flex 2 SDK include files that implement the most basic wrapper in the /resources/html-templates/no-player-detection directory. Other directories in the /resources/html-templates subdirectory define templates with additional features such as history management and Express Install. You can use these files or you can use the instructions in this section to create your own.

The basic wrapper consists of the following:

- **HTML page.** This is the file that the client browser requests. It typically defines two possible experiences (one for users with JavaScript enabled and one for users without JavaScript enabled). This page also references a separate JavaScript file.

  In the provided HTML templates, this file is named index.template.html.
- **JavaScript file.** The JavaScript file referenced by the `<script>` tag in the HTML page includes the following:

  <object> tag This tag embeds the SWF file for Internet Explorer.

  <embed> tag This tag embeds the SWF file for Netscape-based browsers.

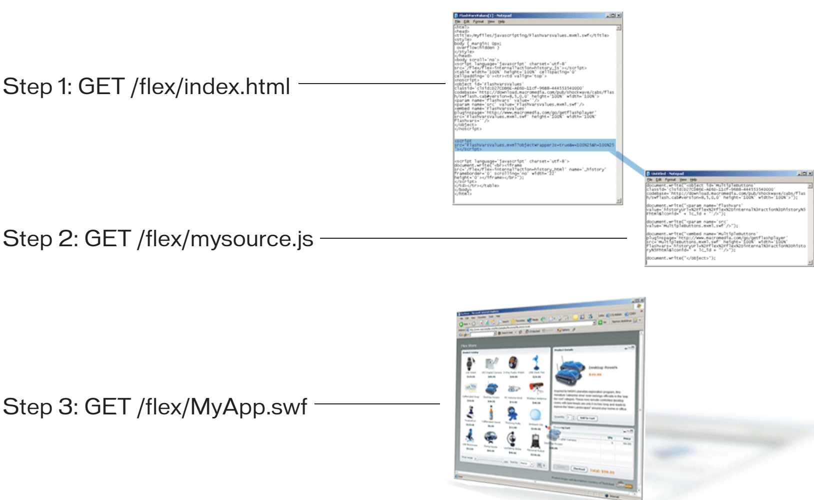
  In the provided HTML templates, the JavaScript file is named AS_OETags.js.

The client first requests the HTML page. If the user's browser has JavaScript enabled, the HTML page then references the JavaScript file. The JavaScript file embeds the Flex application's SWF file.

To make your Flex application respond immediately without user interaction, use a `<script>` tag to load the JavaScript file that contains the `<object>` and `<embed>` tags. Do not write the `<object>` and `<embed>` tags directly in the HTML file. Controls that are internally loaded require activation before they will run in Microsoft Internet Explorer 7 or later. If you load the controls directly in the HTML page, users will be required to activate those controls before they can use them by clicking on the control or giving it focus. This is undesireable because you want your Flex application to run immediately when the page loads, not after the user interacts with the control.

If the client disabled JavaScript in their browser, you typically embed the Flex application directly in the `<noscript>` tag. You can add warnings that they should enable JavaScript or else they will have a less than ideal experience.

The following example illustrates the minimum number of requests that the client browser makes when JavaScript is enabled:



The following example shows the minimum requirements of the HTML page and JavaScript file to embed a Flex application named MyApp:

```
<!-- index.html -->
<!-- saved from url=(0014)about:internet -->
<html>
  <body>
    <script src="mysource.js"></script>
    <noscript>
      <object id='MyApp' classid='clsid:D27CDB6E-AE6D-
        11cf-96B8-444553540000' codebase='http://download.macromedia.com
        /pub/shockwave/cabs/flash/swflash.cab#version=9,0,0,0'
        height='100%' width='100%'>
        <param name='src' value='MyApp.swf'/>
        <embed name='MultipleButtons' pluginspage='http://
          www.macromedia.com/shockwave/download/index.cgi
          ?P1_Prod_Version=ShockwaveFlash' src='MyApp.swf' height='100%'
          width='100%'/>
      </object>
    </noscript>
  </body>
</html>
```

```
<!-- mysource.js -->
document.write("<object id='MultipleButtons' classid='clsid:D27CDB6E-AE6D
  -11cf-96B8-444553540000' codebase='http://download.macromedia.com/pub/
  shockwave/cabs/flash/swflash.cab#version=9,0,0,0' height='100%'
  width='100%'>");
document.write("<param name='src' value='MyApp.swf'/>");
document.write("<embed name='MyApp' src='MyApp.swf'
  pluginspage='http://www.macromedia.com/shockwave/download/index.cgi
  ?P1_Prod_Version=ShockwaveFlash' height='100%' width='100%'/>");
document.write("</object>");
```

Adding the Mark of the Web (MOTW) to your wrapper is optional. However, if you do not add the MOTW to your wrapper, your application might not open in the expected security zone within Internet Explorer. The following example MOTW forces Internet Explorer to open the page in the Internet zone:

```
<!-- saved from url=(0014)about:internet -->
```

In general, add a MOTW when you are previewing pages locally before publishing them on a server. For more information about the MOTW, see http://msdn.microsoft.com/workshop/author/dhtml/overview/motw.asp.

About the HTML page

The wrapper's HTML page includes the following:

<script> block The script block embeds the JavaScript file. This JavaScript file defines the `<object>` and `<embed>` tags that embed the SWF file in the HTML page. This block is for users who have enabled JavaScript in their browser.

<noscript> block The code in the `<noscript>` block uses `<object>` and `<embed>` tags to embed the SWF file in the HTML page for users who have disabled JavaScript in their browser. The `<noscript>` block is useful if your application requires JavaScript (for example, if you use the ExternalInterface API in your application). You can use this block to warn users that they will have limited functionality, or redirect them to another site. For a simple application, however, your `<noscript>` block typically contains identical tags as they are defined in the JavaScript file. For more complex wrappers that include support for Express Install, the `<script>` block can include a considerable amount of JavaScript code.

About the JavaScript file

The JavaScript file consists of a set of `document.write()` methods that write the `<object>` and `<embed>` tags that embed your application. In this example, these tags are identical to the `<object>` and `<embed>` tags used in the HTML page's `<noscript>` block. In more complex configurations, you can add Express Install or history management support to the JavaScript file that is not supported in the HTML page's `<noscript>` block. Remember that the code in the HTML page is for browers that do not support JavaScript.

The `<object>` tag's `codebase` and the `<embed>` tag's `pluginspage` properties add support for basic player version detection and installation. The `codebase` tag defines the minimum version required at the end of the URL (for example, `#version=9,0,0,0`). If a client requests this page with a player version older than that, they are prompted to upgrade their player.

The upgrade experience is considerably better with Express Install. If the user's player does not meet the minimum requirements, the new player is automatically installed for them. You add Express Install by editing your wrapper. For more information, see Chapter 17, "Using Express Install," on page 391.

With a generic wrapper, a user who clicks the Back and Forward buttons in their browser navigates the HTML pages in the browser's history and not the history of their interactions within the Flex application. History management allows users to navigate their interactions with the Flex application by using the Back and Forward buttons in their browser. You can add history management by editing your wrapper. For more information, see Chapter 32, "Using the History Manager," in *Flex 2 Developer's Guide*.

In addition to adding history management and Flash Player detection support to your wrapper, you can use other properties of the `<object>` and `<embed>` tags to add functionality. For more information, see "About the <object> and <embed> tags" on page 378.

Adding features to the wrapper

The default wrapper that Flex Builder and Flex Data Services create includes history management and support for Express Install. The History Manager lets users navigate through a Flex application using the web browser's Back and Forward buttons. Express Install detects if the client has the required version of Flash Player to run the application and installs a newer player on the client if necessary.

Each of these features requires additional files to be deployed with your application. This section gives an overview of the files required by these features. For additional information on implementing these features, see Chapter 32, "Using the History Manager," in *Flex 2 Developer's Guide* and Chapter 17, "Using Express Install," on page 391.

Before adding additional functionality to your custom wrapper, you should understand the issues described in "Customizing the wrapper" on page 375.

A simple application uses only the main wrapper plus a JavaScript file that embeds the Flex application's SWF file. The following files are required by a simple wrapper:

- wrapper.html
- myscript.js
- app.swf

Customizing the wrapper

This section lists some guidelines to follow when you create a custom wrapper.

To customize your HTML wrapper, keep the following guidelines in mind:

- You can use the HTML templates in the /resources/html-templates directory as guides to adding new features to the wrapper. For information about the templates, see "About the HTML templates" on page 370.
- You must embed the SWF file and not the MXML file, even if you are using the web tier compiler at run-time (if your users request the MXML file in their browser). Set the value of the `src` property of the `<object>` tag to *`mxml_filename`*`.mxml.swf` if you use the web-tier compiler. If you use the command-line compiler or Flex Builder, set the value of the `src` property to *`mxml_filename`*`.swf`.

The following example defines the `src` property of the `<object>` tag for an MXML application called MyApp.mxml:

```
<param name='src' value='MyApp.mxml.swf'>
```

The `<embed>` tag uses the `src` property to define the source of the SWF file:

```
src='MyApp.mxml.swf'
```

- Do not include periods or other special characters in the `id` and `name` properties of the `<object>` and `<embed>` tags. These tags identify the SWF object on the page, and you use them when you use the ExternalInterface API. This API lets Flex communicate with the wrapper, and vice versa. For more information about using the ExternalInterface API, see Chapter 34, "Communicating with the Wrapper," in *Flex 2 Developer's Guide*.
- Do not put the contents of the JavaScript file directly in the HTML page. This causes Internet Explorer to prompt the user before enabling Flash Player. If the client has "Disable Script Debugging (Internet Explorer)" unchecked in Internet Explorer's advanced settings, the browser still prompts the user to load the ActiveX plug-in before running it.
- If you use both the `<object>` and the `<embed>` tags in your custom wrapper, use identical values for each attribute to ensure consistent playback across browsers. For more information about the `<object>` and the `<embed>` tags, see "About the <object> and <embed> tags" on page 378.
- To add basic player detection logic without history management or Express Install support, use the templates in the /resources/html-templates/client-side-detection directory.
- To add support for history management, follow the instructions in "Using history management in a custom wrapper" on page 1157.
- To add support for Flash Player detection, follow the instructions in "Configuring Express Install on Flex Data Services" on page 396.
- When using Flex Builder, the default wrapper includes history management and Express Install support. You can disable one or both of these features by using the Compiler Properties dialog box. You can also use this dialog box to set the minimum required version of the Flash Player.
- When using the web-tier compiler with Flex Data Services, you can change some of the wrapper settings in the flex-webtier-config.xml file. These settings are applied to the generated wrapper that Flex returns for a *.mxml file request. For more information on the web-tier compiler, see "Using the web-tier application compiler" on page 183.

Adding Express Install to your wrapper

Express Install is included by default in the wrappers generated by Flex Builder and Flex Data Services. If you write your own wrapper, however, you must add it manually or use the HTML templates in the /resources directory as a base.

Adding Express Install support involves adding JavaScript and VBScript to your main wrapper file, as well as replacing your external JavaScript file with the AC_OETags.js file. In addition, you must deploy another SWF with your application.

The following files are required by a wrapper with Express Install support:

- wrapper.html (with additional version detection logic)
- AC_OETags.js
- playerProductInstall.swf
- app.swf

The AC_OETags.js file defines functions that the wrapper calls to embed the Flex application's SWF file. It works similarly to the simple external JavaScript file described in "About the JavaScript file" on page 374.

In addition to using the AC_OETags.js file, you must also deploy the playerProductInstall.swf file in a location that is accessible by the main application SWF file.

The files required by Express Install are located in the /resources/html-templates/express-installation and /resources/html-templates/express-installation-with-history directories.

For more information about adding support for Express Install to your wrapper, see Chapter 17, "Using Express Install," on page 391.

Adding history management to your wrapper

Support for history management is included by default in the wrappers generated by Flex Builder and Flex Data Services. If you write your own wrapper, however, you must add it manually or use the HTML templates in the /resources/html-templates directory as a base.

To add history management support, you reference the history.js file in a `<script>` tag and the history.htm file in an iframe. The history.js file records actions for history management. You also deploy another SWF file with your application.

The following files are used by wrappers that support history management:

- wrapper.html
- history.js
- myscript.js
- app.swf

- history.htm
- history.swf

The HTML file references the history.swf file. You must deploy this SWF file to a location that is accessible by the main Flex application SWF file.

If your wrapper supports history management but does not include any player version detection logic or support for Express Install, you must combine the contents of the history.js and myscript.js files.

For more information about adding history management support to your wrapper, see "Using history management in a custom wrapper" on page 1157.

About the <object> and <embed> tags

The `<object>` and `<embed>` tags embed your Flex application in the wrapper. They support a set of properties that add additional functionality to the wrapper. These properties let you change the appearance of the SWF file on the page or change some of its properties such as the title or language. If you want to customize your wrapper, you can add these properties to the wrapper.

The `<object>` tag is used by Internet Explorer 3.0 or later on Windows 9x, Windows 2000, Windows NT, Windows ME, and Windows XP platforms or any browser that supports the use of the Flash ActiveX control. The `<embed>` tag is used by Netscape Navigator 2.0 or later, or browsers that support the use of the Netscape-compatible plug-in version of Flash Player.

When an ActiveX-enabled browser loads the HTML page, it reads the values set on the `<object>` and ignores the `<embed>` tag. When browsers using the Flash plug-in load the HTML page, they read the values set on the `<embed>` tag and ignore the `<object>` tag. Make sure that the properties for each tag are identical, unless you want different results depending on the user's browser.

You must set the values of four required properties (`height`, `width`, `classid`, and `codebase`) as attributes in the `<object>` tag. All other properties are optional and you set their values in separate, named `<param>` tags.

The following example shows the required properties as attributes of the `<object>` tag, and four optional properties, `src`, `play`, `loop`, and `quality`, as `<param>` child tags:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="100"
  height="100" codebase="http://active.macromedia.com/flash7/cabs/
  swflash.cab#version=9,0,0,0">
  <param name="src" value="moviename.swf">
  <param name="play" value="true">
  <param name="loop" value="true">
  <param name="quality" value="high">
</object>
```

Although the `src` property is technically an optional tag, without it, there is no reference to the application you want the client to load. Therefore, your wrapper should always set the `src` property in both the `<object>` and `<embed>` tags.

For the `<embed>` tag, all settings are attributes that appear between the angle brackets of the opening `<embed>` tag. The `<embed>` tag requires the `height` and `width` attributes, and the `pluginspage` attribute, which is the equivalent of the `<object>` tag's `codebase` property. The `<embed>` tag does not require a `classid` attribute.

> **TIP** Although the `codebase` and `pluginspage` properties are required, they are not necessarily used if you use Flash Player Detection Kit to detect and install the required version of Flash Player. For more information, see Chapter 17, "Using Express Install," on page 391.

The following example shows a simple `<embed>` tag with the optional `quality` attribute:

```
<embed src="moviename.swf" width="100" height="100" quality="high"
  pluginspage="http://www.macromedia.com/shockwave/
  download/index.cgi?P1_Prod_Version=ShockwaveFlash">
</embed>
```

To use both tags together, position the `<embed>` tag just before the closing `</object>` tag, as the following example shows:

```
<object classid="clsid:d27cdb6e-ae6d-11cf-96b8-444553540000" width="100"
  height="100" codebase="http://active.macromedia.com/flash7/cabs/
  swflash.cab#version=9,0,0,0">
  <param name="src" value="moviename.swf">
  <param name="play" value="true">
  <param name="loop" value="true">
  <param name="quality" value="high">
  <embed src="moviename.swf" width="100" height="100" play="true”
  loop="true" quality="high" pluginspage="http://www.macromedia.com/
  shockwave/download/index.cgi?P1_Prod_Version=ShockwaveFlash">
  </embed>
</object>
```

When you define parameters for the `<object>` tag, also add them as tag properties to the `<embed>` tag so that the SWF file appears the same on the page regardless of the client's browser.

Not all properties are supported by both the `<object>` and the `<embed>` tags. For example, the `id` property is used only by the `<object>` tag, just as the `name` property is used only by the `<embed>` tag.

In some cases, the `<object>` and `<embed>` tag properties duplicate properties that you can set on the `<mx:Application>` tag in the Flex application source code. For example, you can set the `height` and `width` properties of the SWF file on the `<object>` and `<embed>` tags or you can set them on the `<mx:Application>` tag.

The following table describes the supported `<object>` and `<embed>` tag properties:

Property	Type	Description
`align`	String	Specifies the position of the SWF file. The `align` property supports the following values: • `bottom`: Vertically aligns the bottom of the SWF file with the current baseline. This is the default value. • `middle`: Vertically aligns the middle of the SWF file with the current baseline. • `top`: Vertically aligns the top of the SWF file with the top of the current text line. • `left`: Horizontally aligns the SWF file to the left margin. • `right`: Horizontally aligns the SWF file to the right margin.
`allowNetworking`	String	Restricts browser communication. This property affects more APIs than the `allowScriptAccess` property. The `allowNetworking` property supports the following values: • `all`: No networking restrictions. Flash Player behaves normally. This is the default. • `internal`: SWF files cannot call browser navigation or browser interaction APIs (such as the `ExternalInterface.call()`, `fscommand()`, and `navigateToURL()` methods), but can call other networking APIs. • `none`: SWF files cannot call networking or SWF-to-SWF file communication APIs. In addition to the APIs restricted by the `internal` value, these include other methods such as `URLLoader.load()`, `Security.loadPolicyFile()`, and `SharedObject.getLocal()`. For more information, see *Programming ActionScript 3.0*.

Property	Type	Description
`allowScriptAccess`	String	Controls the ability to perform outbound scripting from within the SWF file. The `allowScriptAccess` property can prevent a SWF file hosted from one domain from accessing a script in an HTML page that comes from another domain. Setting `allowScriptAccess` to `never` for all SWF files hosted from another domain can ensure security of scripts located in an HTML page. Valid values are as follows: • `always`: Outbound scripting always succeeds. • `never`: Outbound scripting always fails. • `samedomain`: Outbound scripting succeds only if the application is from the same domain as the HTML page. The default value is `always`. This property affects the following operations: • `ExternalInterface.call()` • `fscommand()` • `navigateToURL()`, when used with `javascript` or another scripting scheme • `navigateToURL()`, when used with window name of `_self`, `_parent`, or `_top`. For more information, see *Programming ActionScript 3.0*.
`archive`	String	Specifies a space-separated list of URIs for archives containing resources used by the application, which may include the resources specified by the `classid` and `data` properties. Preloading archives can result in reduced load times for applications. Archives specified as relative URIs are interpreted relative to the `codebase` property.
`base`	String	Specifies the base directory or URL used to resolve relative path statements in ActionScript.

Property	Type	Description

bgcolor	String	Specifies the background color of the application. Use this property to override the background color setting specified in the SWF file. This property does not affect the background color of the HTML page.
		Valid formats for bgcolor are any #RRGGBB, hexadecimal, or RGB value.
		The Application container’s style uses an image as the default background image. This image obscures any background color settings that you might make. So, to make the value of the bgcolor property display properly, you must clear the Application container’s backgroundImage style property. To do this, you can set it to the value of a space character, as the following example shows:
		<mx:Style>
		   Application {
		      backgroundImage: " ";
		   }
		</mx:Style>

border	int	Specifies the width of the SWF file’s border, in pixels. The default value for this property depends on the user agent.
classid	String	Defines the classid of Flash Player. This identifies the ActiveX control for the browser. Internet Explorer 3.0 or later on Windows 9x, Windows 2000, Windows NT, Windows ME, and Windows XP prompt the user with a dialog box asking if they would like to auto-install Flash Player if it's not already installed. This process can occur without the user having to restart the browser. This property is used for the <object> tag only. For the <object> tag, you set the value of this property as an attribute of the <object> tag and not as a <param> tag.
codebase	String	Identifies the location of Flash Player ActiveX control so that the browser can download it if it is not already installed. This property is used for the <object> tag only. You can modify this property by using the settings of the Flex Data Services server or from Flex Builder. For the <object> tag, you set the value of this property as an attribute of the tag and not as a child <param> tag. Like the pluginspage property, the codebase property is required. However, they are not necessarily used if you use Flash Player Detection Kit to detect and install the required version of Flash Player. For more information, see Chapter 17, “Using Express Install,” on page 391.

Property	Type	Description
codetype	String	Defines the content type of data expected when downloading the application specified by the classid property The codetype property is optional but recommended when the classid property is specified; it lets the browser avoid loading unsupported content types. The default value of the codetype property is the value of the type property.
data	String	Specifies the location of the application's data; for example, instance image data for objects that define images. If the data property is a relative URI, it is relative to the codebase property.
declare	Boolean	Makes the current SWF file's definition a declaration only. The SWF file must be instantiated by a subsequent object definition referring to this declaration.
devicefont	Boolean	Specifies whether static text objects for which the deviceFont option is not selected are drawn using a device font anyway, if the needed fonts are available from the operating system.
dir	String	Specifies the base direction of text in an element's content and attribute values. It also specifies the directionality of tables. Valid values are LTR (left-to-right text or table) and RTL (right-to-left text or table).
flashVars	String	Sends variables to the application. The format is a set of name-value pairs, each separated by an ampersand (&). Browsers support string sizes of up to 64 KB (65535 bytes) in length. The default value of this property is an empty string. If you use query string parameters when requesting an MXML file from the Flex Data Services server, Flex converts them to flashVars properties. For more information on using flashVars to pass variables to Flex applications, see Chapter 34, "Communicating with the Wrapper," in *Flex 2 Developer's Guide*.

Property	Type	Description
height	int	Defines the height, in pixels, of the SWF file. Flash Player makes a best guess to determine the height of the application if none is provided. The browser scales an object or image to match the height and width specified by the author. You can set this value to a fixed number or a percentage value; for example, length='100' or length='50%'. Lengths expressed as percentages are based on the horizontal or vertical space currently available, not on the default size of the SWF file. FireFox browsers do not support percentage-based values. You can also set the height of a Flex application by setting the height property of the <mx:Application> tag in an MXML file. For the <object> tag, you set the value of this property as an attribute of the <object> tag and not as a <param> child tag.
hspace	int	Specifies the amount of white space inserted to the left and right of the SWF file. The default value is not specified, but is generally a small, nonzero length.
id	String	Identifies the SWF file to the host environment (a web browser, for example) so that it can be referenced by using a scripting language such as VBScript or JavaScript. The id property is only used with the <object> tag. It is equivalent to the name property used with the <embed> tag.
lang	String	Specifies the base language of an element's property values and text content. The default value is unknown. The browser can use language information specified using the lang property to control rendering in a variety of ways.
menu	Boolean	Changes the appearance of the menu that appears when users right-click over a Flex application in Flash Player. Set to true to display the entire menu. Set to false to display only the About and Settings options on the menu. The default value is true.
name	String	Identifies the SWF file to the host environment (a web browser, typically) so that it can be referenced by using a scripting language. The name property is only used with the <embed> tag. It is equivalent to the id property used with the <object> tag.

Property	Type	Description
pluginspage	String	Identifies the location of Flash Player plug-in so that the user can download it if it is not already installed. This property is used for the <embed> tag only. You can modify this property by using the settings of the Flex Data Services server or from Flex Builder. Like the codebase property, the pluginspage property is required. However, these properties are not necessarily used if you use Flash Player Detection Kit to detect and install the required version of Flash Player. For more information, see Chapter 17, "Using Express Install," on page 391.
quality	String	Defines the quality of playback in Flash Player. Valid values of quality are low, medium, high, autolow, autohigh, and best. The default value is best. The low setting favors playback speed over appearance and never uses anti-aliasing. The autolow setting emphasizes speed at first but improves appearance whenever possible. Playback begins with anti-aliasing turned off. If Flash Player detects that the processor can handle it, anti-aliasing is turned on. The autohigh setting emphasizes playback speed and appearance equally at first, but sacrifices appearance for playback speed if necessary. Playback begins with anti-aliasing turned on. If the actual frame rate drops below the specified frame rate, anti-aliasing is turned off to improve playback speed. Use this setting to emulate the View > Antialias setting in Flash. The medium setting applies some anti-aliasing and does not smooth bitmaps. The high setting favors appearance over playback speed and always applies anti-aliasing. The best setting provides the best display quality and does not consider playback speed. All output is anti-aliased and all bitmaps are smoothed.

Property	Type	Description
`salign`	String	Positions the SWF file within the browser. Valid values are `L`, `T`, `R`, `B`, `TL`, `TR`, `BL`, and `BR`. `L`, `R`, `T`, and `B` align the SWF file along the left, right, top, or bottom edge, respectively, of the browser window and crop the remaining three sides as needed. `TL` and `TR` align the SWF file to the top-left and top-right corner, respectively, of the browser window and crop the bottom and remaining right or left side as needed. `BL` and `BR` align the SWF file to the bottom-left and bottom-right corner, respectively, of the browser window and crop the top and remaining right or left side as needed.
`scale`	String	Defines how the browser fills the screen with the SWF file. The default value is `showall`. Valid values of the `scale` property are `showall`, `noborder`, and `exactfit`. Set to `showall` to make the entire SWF file visible in the specified area without distortion, while maintaining the original aspect ratio of the SWF file. Borders may appear on two sides of the SWF file. Set to `noborder` to scale the SWF file to fill the specified area, without distortion but possibly with some cropping, while maintaining the original aspect ratio of the SWF file. Set to `exactfit` to make the entire SWF file visible in the specified area without trying to preserve the original aspect ratio. Distortion may occur.
`src`	String	Identifies the location of the SWF file. If you write a custom wrapper but deploy your application as MXML files and not pregenerated SWF files on a Flex Data Services server, use the following naming convention: *movie_name*.mxml.swf Use this property for the `<object>` and `<embed>` tags.
`standby`	String	Defines a message that the browser displays while loading the object’s implementation and data.
`style`	String	Specifies style information for the SWF file. The syntax of the value of the `style` property is determined by the default style sheet language. In CSS, property declarations have the form "name:value" and are separated by a semicolon. Styles set with this property do not affect components or the Application container in the Flex application. Rather, they apply to the SWF file as it appears on the HTML page.

Property	Type	Description
supportembed	Boolean	Determines whether the Netscape-specific <embed> tag is supported. The supportembed property is optional, and the default value is true. Set to false to prevent the <embed> tag from being read by the browser.
tabindex	int	Specifies the position of the SWF file in the tabbing order for the current document. This value must be a number between 0 and 32767. User agents should ignore leading zeros.
title	String	Displays information about the SWF file. Values of the title property can be rendered by browsers or other user agents in different ways. For example, some browsers display the title as a ToolTip. Audio user agents might speak the title information in a similar context.
type	String	Specifies the content type for the data specified by the data property. The type property is optional but recommended when data is specified; it prevents the browser from loading unsupported content types. If the value of this property differs from the HTTP Content-Type returned by the server, the HTTP Content-Type takes precedence.
usemap	String	Associates an image map with the SWF file. The image map is defined by a map element. The value of usemap must match the value of the name attribute of the associated map element.
vspace	int	Specifies the amount of white space inserted above and below the SWF file. The default value is not specified, but is generally a small, nonzero length.

Property	Type	Description
width	int	Defines the width, in pixels, of the SWF file. Flash Player makes a best guess to determine the width of the application if none is provided. Browsers scale an object or image to match the height and width specified by the author. You can set this value to a fixed number or a percentage value. For example, “width=100” or “width=”50%”. Lengths expressed as percentages are based on the horizontal or vertical space currently available, not on the natural size of the SWF file. You can also set the width of a Flex application by setting the width property of the <mx:Application> tag in an MXML file. For the <object> tag, you set the value of this property as an attribute of the <object> tag and not as a <param> tag.
wmode	String	Sets the Window Mode property of the SWF file for transparency, layering, and positioning in the browser. Valid values of wmode are window, opaque, and transparent. Set to window to play the SWF in its own rectangular window on a web page. Set to opaque to hide everything on the page behind it. Set to transparent so that the background of the HTML page shows through all transparent portions of the SWF file. This can slow animation performance. To make sections of your SWF file transparent, you must set the alpha property to 0. To make your application’s background transparent, set the alpha property on the <mx:Application> tag to 0. The wmode property is not supported in all browsers and platforms.

The <object> and <embed> tags can also take additional properties that are not supported by Flex applications. These unsupported properties are listed in “Unsupported properties” on page 389.

Unsupported properties

Some optional Flash Player properties do not apply to Flex applications. These are properties that involve movie frames and looping. The following properties have no effect when used with Flex:

- `loop`
- `play`
- `swliveconnect`

Requesting an MXML file without the wrapper

If you are using the web-tier compiler with Flex Data Services, you can request an MXML file that has not yet been compiled into a SWF and have Flex return only the SWF file (without the wrapper). You do this by appending *.swf to the end of the request string that specifies *.mxml.

For example, the following request returns only a SWF file and no wrapper:

```
http://www.mysite.com/flex/MyApp.mxml.swf
```

CHAPTER 17

Using Express Install

17

This topic describes how to detect if the client has the required version of Flash Player to run your Flex application and upgrade the player if necessary. In most cases, the player is an ActiveX control running inside Microsoft Internet Explorer or a plug-in for Netscape-based browsers. You edit the wrapper to include version detection logic, and logic that installs a newer player on the client if necessary. This feature is known as Express Install.

Express Install requires that the client have Flash Player 6.0.65 or later installed on MacOS or Microsoft Windows, and that the browser has JavaScript enabled.

Contents

About Express Install 391
Editing your wrapper 392
Configuring Express Install on Flex Data Services 396
Alternatives to Express Install 397

About Express Install

After developing an application, you want to ensure that all users can run it and that they have a current version of the Flash Player. Flex includes code and applications that make the updating process for the player simple for you and nearly transparent for the client. These files are located in the /resources/html-templates directory for Flex 2 SDK and Flex Data Services.

The recommended method of ensuring that Flash Player can run the Flex application on the client is to use Express Install. With Express Install, you can detect when users do not have the latest version of Flash Player, and you can initiate an update process that securely installs the latest version of the player from the Adobe website. When the installation is complete, users are directed back your website, where they can run your Flex application.

Express Install runs a SWF file in the existing Flash Player to upgrade users to the latest version of the player. As a result, Express Install requires that Flash Player already be installed on the client, and that it be version 6.0.65 or later. The Express Install feature also relies on JavaScript detection logic in the browser to ensure that the player required to start the process exists. As a result, the browser must have JavaScript enabled for Express Install to work.

If the player on the client is not new enough to support Express Install, you can display alternate content, redirect the user to the Flash Player download page, or initiate another type of Flash Player upgrade experience. For information on using alternative Player upgrade techniques, see "Alternatives to Express Install" on page 397.

Editing your wrapper

After you compile your application into a SWF file, you write a wrapper that embeds that SWF file. Clients request this wrapper directly. You can use Adobe Flex Data Services or Adobe Flex Builder to automatically generate a wrapper, or you can write one yourself. Express Install is included by default in the wrappers generated by Flex Builder and Flex Data Services. You can configure Express Install settings in the flex-webtier-config.xml file for Flex Data Services. For more information, see "About the wrapper generated by Flex Data Services" on page 368.

If you write your own wrapper, however, you must add Express Install manually. Sample wrapper templates are available for Flex Data Services and Flex 2 SDK in the /resources/html-templates directory. For more information, see "About the HTML templates" on page 370.

This section is for users who write their own wrapper and add Express Install support to it. It is also for users who customize a generated wrapper or template. For more information about creating a wrapper, see Chapter 16, "Creating a Wrapper," on page 367.

Adding Express Install script to the wrapper

After you write your own wrapper, follow the steps in this section to add support for Express Install. The steps described here require the following files, which are located in the /resources/html-templates/express-installation directory:

- **index.template.html** Wrapper file that detects the Flash Player version and initiates Express Install if necessary. You integrate this code with your wrapper.
- **AC_OETags.js** Script file that provides methods for Flash Player version detection and embedding your Flex application. You call methods in this file from your wrapper.
- **playerProductInstall.swf** Flash application that initiates Express Install. You deploy this file with your application.

To add Express Install support to your wrapper:

1. Open your wrapper in a text editor.
2. Add or include the script in the index.template.html file to your wrapper. This file is included in the /resources/html-templates/express-installation directory. For a detailed description of this script, see "Understanding the Express Install script" on page 395.
3. In the Globals section of the script, set the minimum version of Flash Player that your users must be running. For Flex 2 applications, the minimum version number is 9.0.0; for example:

   ```
   // Globals
   // Major version of Flash required
   var requiredMajorVersion = 9;
   // Minor version of Flash required
   var requiredMinorVersion = 0;
   // Minor version of Flash required
   var requiredRevision = 0;
   ```

   If you are using Flex Data Services to generate the wrapper, you can set this value in the flex-webtier-config.xml file. If you are using Flex Builder to generate the wrapper, you can set this value in the Detect Flash Version text box in the Compiler Properties dialog box.
4. Deploy the AC_OETags.js file to a location that is accessible to your wrapper. This file is included in the /resources/html-templates/express-installation directory. This file is also generated by Flex Builder and Flex Data Services. The default location is the same directory as the wrapper. If you change it, you must also edit the following line in the wrapper:

   ```
   <script src="AC_OETags.js" language="javascript"></script>
   ```

5. Edit the calls to the `AC_FL_RunContent()` method. This method passes information about the SWF file to be run to the AC_OETags.js external script file. The functions defined in the AC_OETags.js file write the `<object>` and `<embed>` tags for the SWF file.

   There are multiple calls to this method in script blocks that meet different conditions. In the first script block, change the `id` and `name` parameters to match your SWF file's name:

```
if (hasProductInstall && !hasRequestedVersion) {
    AC_FL_RunContent(
    "src", "playerProductInstall",
    "FlashVars", "MMredirectURL="+MMredirectURL+'&MMplayerType='+
      MMPlayerType+'&MMdoctitle='+MMdoctitle+"",
    "width", "100%",
    "height", "100%",
    "align", "middle",
    "id", "MyFirstProject",
    "quality", "high",
    "bgcolor", "#869ca7",
    "name", "MyFirstProject",
    "allowScriptAccess","sameDomain",
    "type", "application/x-shockwave-flash",
    "pluginspage", "http://www.macromedia.com/go/getflashplayer"
  );
}
```

   In the second script block, change the `src`, `id`, and `name` parameters to match your SWF file's name:

```
} else if (hasRequestedVersion) {
  AC_FL_RunContent(
      "src", "MyFirstProject",
      "width", "100%",
      "height", "100%",
      "align", "middle",
      "id", "MyFirstProject",
      "quality", "high",
      "bgcolor", "#869ca7",
      "name", "MyFirstProject",
      "flashvars",'historyUrl=history.htm?&lconid=' + lc_id +'&',
      "allowScriptAccess","sameDomain",
      "type", "application/x-shockwave-flash",
      "pluginspage", "http://www.macromedia.com/go/getflashplayer"
  );
}
```

When you edit these methods, you can also add `flashVars` variables, change the height and width of the SWF file, add history management support, and set other properties of the SWF file, if necessary. For more information about available properties, see “About the <object> and <embed> tags” on page 378. For more information about adding history management support, see Chapter 32, “Using the History Manager,” in *Flex 2 Developer’s Guide*.

6. Replace the value of the `alternateContent` variable with your own custom content. In this step and the next, you can implement alternate methods of upgrading Flash Player for users who do not meet the requirements for Express Install. For more information, see “Alternatives to Express Install” on page 397.
7. Replace the HTML code found within the `<noscript>` tag with your own custom content. For more information, see “Alternatives to Express Install” on page 397.
8. Deploy the playerProductInstall.swf file included in the the /resources/html-templates/express-installation directory to your web server.

   By default, you should deploy this file to the same directory as your Flex application. If you deploy it to another location, you must update the wrapper to point to this new location. This file is also included in your Flex application’s bin directory in Flex Builder.

Understanding the Express Install script

The scripts that implement Express Install in your wrapper are contained in the AC_OETags.js and index.template.html files from the /resources/html-templates/express-installation directory.

The following steps show the order of execution within the Express Install scripts:

For browsers with scripting, the Express Install script:

1. Sets global variables that define the required minimum version of the player.
2. Detects browser type (set the values of the `isIE`, `isWin`, and `isOpera` Boolean properties).
3. Sets the value of the `hasProductInstall` property by calling the `DetectFlashVer()` method.

   This method returns `true` if the current player supports Flash Product Install. This method returns `false` if the current player does not support Flash Product Install. Flash Players later than version 6.0.65 meet this requirement.

4. Sets the value of the `hasRequestedVersion` property by calling the `DetectFlashVer()` method.

   This method returns `true` if the current player is new enough to display the Flex application. This method returns `false` if the current player must be upgraded to display the Flex application.

5. Sets the value of the `MMredirectURL` property to specify the location where the browser is redirected after running Express Install.

6. Sets the value of the `document.title` and `MMdoctitle` properties so the unused browser windows can be closed after running Express Install.

7. Examines the values of the `hasProductInstall` and `hasRequestedVersion` properties:

   a. If the current player is version 6.0.65 or later (`hasProductInstall=true`) but it cannot play the current Flex application (`hasRequestedVersion=false`), run the playerProductInstall.swf file. This upgrades the player with Express Install.

   b. If the current version of the player meets the requirements for playback (`hasRequestedVersion=true`), run the Flex application.

   c. If the current player is earlier than version 6.0.65 (`hasProductInstall=false`) and the version is not new enough (`hasRequestedVersion=false`), show alternate content or upgrade without Express Install.

For browsers without scripting, the Express Install script:

1. Shows the link to the Flash Player download page.

2. Shows alternate content or prompts the user to upgrade without using Express Install.

Configuring Express Install on Flex Data Services

If you are running Flex Data Services, you can generate the HTML wrapper with custom settings for Express Install.

If you generate a wrapper with Flex Data Services, you can configure the major and minor versions required to run your application, and disable version detection in the wrapper.

The settings are in the `<flash-player>` block in the flex-webtier-config.xml file.

For more information, see “About the wrapper generated by Flex Data Services” on page 368.

Alternatives to Express Install

When you add the Express Install script to your wrapper, one of the following results occur when a client requests that wrapper:

- Client runs the application.
- Client upgrades Flash Player by using Express Install and then runs the application.
- Client upgrades Flash Player by using alternative method and then runs the application.
- Client does not upgrade Flash Player and runs alternate content.

Users who do not update the Flash Player version by using Express Install generally fall into the following categories:

- **Browser with disabled scripting** If the browser disables JavaScript, the browser executes content in the `<noscript>` tag of the wrapper. The version detection logic from the index.template.html and AC_OETags.js files is not interpreted when your wrapper is loaded into a browser, nor is Express Install usable.
- **Browser with no Flash Player installed** If the browser has no Flash Player installed but JavaScript is enabled, the browser executes the alternate content area in the `<script>` tag of the wrapper.
- **Browser with Flash Player version earlier than 6.0.65** If Flash Player is not new enough to run the Express Install SWF file (playerProductInstall.swf), but JavaScript is enabled, the browser executes the alternate content area in the `<script>` tag of the wrapper.
- **User refuses Flash Player installation or upgrade** If the user declines to install Flash Player or to upgrade their version of Flash Player, the browser executes the alternate content area in the `<script>` tag of the wrapper. It is up to you to determine whether to provide content in HTML format or some other format that the browser can render without using Flash Player.

In situations where the browser executes alternate content, you can use the `<object>` and `<embed>` tags to embed your Flex application and provide an upgrade and installation path for players that are old or missing.

The `<object>` tag's `codebase` property is used to enforce the player versioning for a Microsoft Internet Explorer browser. The tag adds support for basic player version detection and installation. The `codebase` property defines the minimum version specified at the end of the CAB file's location (for example, `#version=9,0,0`). If the browser requests this page with a player version older than that, the user is prompted to upgrade their player. This installation can occur without the user having to restart the browser.

The `<embed>` tag's `pluginspage` property is used for Firefox, Netscape 8, and Mozilla-based browsers. If there is no plug-in installed, the browser displays a plug-in icon and the text "Click here to get the plug-in." When a user clicks the icon, they are directed to an appropriate location, depending on the type of browser being used, where they can download and install the latest version of Flash Player. The `pluginspage` property does not enforce a required version of the plug-in.

Flash Player is available from the Firefox Plug-in Finder Service. This means that if the Firefox browser does not have the player installed when it encounters an `<embed>` tag, it guides the user through the process of downloading and installing Flash Player through the Finder Service.

Consider adding a note to users of Firefox, Netscape 8, and Mozilla-based browsers that if they have scripting disabled, they should upgrade their Flash Players to version 9 before continuing. You can put this in the `<noscript>` block to ensure that only users with scripting disabled get this message.

These upgrade processes result in a different upgrade experience than the experience of the user who upgrades by using Express Install. Express Install provides a clean installation and returns the user to the original page. These alternative paths require more steps and in some cases do not provide users with a return path to the original application.

You can do basic version detection without using Express Install. For an example of this, see the files in the /resources/html-templates/client-side-detection and /resources/html-templates/client-side-detection-history directories.

Other techniques for upgrading Flash Player without Express Install are described in the Flash Player Detection Kit. These include using server-side logic and writing a custom SWF file to perform version detection and installation. For more information, see the Detection Kit documentation at www.adobe.com/go/fp_detectionkit.

PART 4

Configuring JRun

4

This part describes how to configure the integrated JRun Application Server for Flex Data Services.

The following topic is included:

Chapter 18: Configuring JRun 401

CHAPTER 18

Configuring JRun

18

This topic describes how to work with the integrated JRun application server that is optionally installed with Adobe Flex Data Services. The JRun application server is not a full-featured server, and this version is not intended to be used in a production environment. You can, however, add and remove web applications and servers, configure logging, virtual directories, and other mappings, and perform many other tasks to use the JRun application server in your development environment.

Contents

About JRun application servers 401
Starting and stopping JRun servers. 404
Adding and removing servers 405
Configuring JRun servers 406
Using the sniffer 411

About JRun application servers

Flex Data Services is a web application that you can install on several different Java application servers. One installation option is to include the integrated JRun Java application server on which you install the Flex Data Services web application. This section describes how to administer the integrated JRun application server.

Limitations of the JRun application server

The integrated JRun application server is not meant to be used in a production environment. Use this server only for development purposes.

The following limitations apply to the JRun application server that is integrated with the Flex Data Services installation process as compared to the stand-alone JRun products:

- Is not licensed for deployment
- Cannot be used as a Windows service (you can only start and stop servers from the command line or the JRun Launcher)
- Does not include JRun Management Console (all configuration must be done through the XML configuration files and command-line)
- Does not include web server connectors (you cannot connect the integrated JRun application server to an external webserver such as IIS or Apache)
- Does not include database drivers
- Does not include integrated Pointbase database
- Does not include JRun samples

If you do use JRun in an environment that is accessible to the network, secure it as best as possible. For more information, see "Securing JRun" on page 63.

About the default web applications

JRun has a single server instance on it named default. The following web applications run on this server instance:

- flex
- flex-admin
- samples

You can request each of these web applications using the localhost and the default port number 8700. For example:

```
http://localhost:8700/samples
```

A web application typically consists of web components such as servlets, JSPs, HTML pages, images, a standard deployment descriptor, optionally, JavaBeans and custom tag classes, and other resources. These resources are in a standard, predefined directory structure so that they can be deployed on any web application server.

To use any of these applications, you must first start the default JRun application server. For more information, see "Starting and stopping JRun servers" on page 404.

About the flex web application

The flex web application is deployed on the default JRun server when the integrated JRun application server starts up. It defines the servlets that open the web-tier compiler and the data services and messaging services. You can store your MXML files and other application assets in the /flex directory or a subdirectory, and then open the root MXML file to compile your Flex application into a SWF file. You typically do this in a browser with a URL similar to the following:

```
http://localhost:8700/flex/MyApp.mxml
```

For more information about the servlets used by the flex web application, see "Servlet configuration" on page 171.

For information about the structure of the files used in the flex web application, see"About deploying an application" on page 351.

The flex web application also generates a wrapper. A wrapper is an HTML page with the embedded Flex application's SWF file. It provides history management and Flash Player detection logic, although you can customize the wrapper to be integrated into your website. For more information about the wrapper, see Chapter 16, "Creating a Wrapper," on page 367.

About the flex-admin web application

The flex-admin web application uses Java Management Beans (MBeans) to provide run-time monitoring and management of the services configured in the Flex services configuration files. The run-time monitoring and management console is an example of a Flex client application that provides access to the run-time MBeans. It calls a Remoting Service destination, which is a Java class that makes calls to the MBeans.

For more information about the flex-admin web application, see "Managing services" on page 1390.

About the samples web application

The samples web application contains sample Flex applications that use the Flex Data Services. These samples demonstrate using services such as RPC and messaging. Applications include chat, real-time data feeds, and collaboration.

For more information, see the samples web application's index.htm page.

Starting and stopping JRun servers

You can start and stop a JRun server from the command line or from the JRun Launcher. This section describes how to start and stop the default JRun application server.

To start the default JRun application server from the JRun Launcher:

1. Open a Command Prompt window.
2. Change directories to the \jrun4\bin directory under the Flex Data Services installation folder.
3. Type the following command:

   ```
   jrun
   ```

   JRun opens the JRun Launcher.
4. Select the default JRun server and click the Start button.

You can also use the JRun Launcher to restart and stop the default JRun server.

To start and stop the default JRun application server from the command line:

In Microsoft Windows:

1. Open a Command Prompt window.
2. Change directories to the \jrun4\bin directory under the Flex Data Services installation folder.
3. Start the server by typing the following command:

   ```
   jrun -start default
   ```

4. To stop the server, press Ctrl+C in the active window or type the following command from another window:

   ```
   jrun -stop default
   ```

In Linux and Solaris:

1. Change directories to the /jrun4/bin directory.
2. Start the server by typing the following command:

   ```
   ./jrun -start default
   ```

3. To stop the server, type the following command:

   ```
   ./jrun -stop default
   ```

When the server is running, you can access the index page by requesting the following URL:

```
http://localhost:8700
```

You cannot run JRun as a Windows service.

Adding and removing servers

The default installation of JRun includes a single server named default. You can add any number of additional JRun application servers.

To add a new JRun application server:

1. Copy the *flex_install_dir*/jrun4/servers/default directory and rename it. The name you give this new directory is the name of the new JRun application server.
2. Add a new entry in the *flex_install_dir*/jrun4/lib/servers.xml file that specifies that server's name and path. For example, if you named the copy of the default directory myserver, you add the following block to the servers.xml file:

   ```
   <server>
     <name>myserver</name>
     <directory>{jrun.home}/servers/myserver</directory>
   </server>
   ```

3. Change the default web service port number of the new JRun server, because you will not be able to start it with the existing default server at the same time if they have the same port numbers. This port number is defined by the WebService's `port` property in the following file:

   `flex_install_dir/jrun4/servers/server_name/SERVER-INF/jrun.xml`

   For more information on changing port numbers, see "Changing port numbers" on page 407.
4. Change the default JNDI naming server port number of the new JRun server. This port number is defined by the `java.naming.provider.url` property in the following file:

   `flex_install_dir/jrun4/servers/server_name/SERVER-INF/jndi.properties`

To remove a JRun server, delete the server's directory and remove its entry from the servers.xml file.

Configuring JRun servers

The integrated JRun application server is a lightweight version of the stand-alone JRun Java application server. It is not intended for use in a production environment. As a result, the configuration options are limited.

The following table describes the configuration files you typically edit to configure the JRun servers:

Configuration file	Description
flex-config.xml	Defines web-tier compiler configuration settings. For more information, see Chapter 8, "Flex 2 SDK and Flex Data Services Configuration," on page 161.
jrun.xml	Defines the JRun application server's settings. You use it to configure settings for JRun services such as logging, web server, and J2EE security. The jrun.xml file is in the following location: {*flex_install_dir*}/jrun4/servers/default/SERVER-INF/jrun.xml
jrun-web.xml	Contains web application elements that are specific to the JRun web application server. You typically do not edit this file unless you are adding virtual directories.
web.xml	Configures settings such as servlet mappings and sets the welcome file. Also enables or disables directory browsing on any web application server on which Flex Data Services is deployed. The Flex application's web.xml deployment descriptor file is in the following location: {*flex_install_dir*}/jrun4/servers/default/flex/WEB-INF/web.xml The web.xml file not a JRun-specific file. It is also known as the *deployment descriptor* because it contains information about the deployed Flex web application. For more information on editing the web.xml file, see "Configuring mappings" on page 175.

The settings in the web.xml and flex-config.xml files are not specific to JRun. Settings you make in these files apply to the flex web application on any J2EE server.

Changing port numbers

The JRun web application server primarily serves as a servlet container. It also includes a built-in web server to respond to HTTP requests and return HTTP responses.

All web servers use a TCP/IP port, which is specified in the request string. The default port for web servers is 80. For example, http://www.myhost.com and http://www.myhost.com:80 are the same. Similarly, port 443 is the default port for HTTPS requests.

The default port for the JRun web server is 8700. For example, to access the index page of the flex web application, you specify http://localhost:8700/flex.

When you install JRun, if the default port is already in use, the installer selects the next higher available port and configures the built-in web server to use that port.

To change the port number, edit the following file:

flex_install_dir/jrun4/servers/default/SERVER-INF/jrun.xml

You can change the value of the port attribute of the web service. The following example changes the port number to 8701:

```
<service class="jrun.servlet.http.WebService" name="WebService">
   <attribute name="port">8701</attribute>
</service>
```

Before you change the port number on which the JRun web server listens, consult a list of common well-known port numbers so that you do not cause conflicts with other services running on your computer.

JRun also uses JNDI services which require unique port numbers. These ports are defined in the jndi.properties file, which is in the same directory as the jrun.xml file.

You do not have to change a port number that is set to 0. If a port number is set to 0, then JRun automatically assigns a unique number to that port when the server starts.

Creating a web application

To create a web application, make a copy of the /flex directory and its contents. Rename the copy and store it in the same location under the /servers/default directory. You can now store MXML files in the new application root directory and request them using the following URL:

`http://localhost:8700/`*`new_app_name`*`/myfile.mxml`

Configuring the hot-deploy feature

You can make configuration changes to your flex web application by modifying the flex-config.xml file. After modifying the flex-config.xml file, you must restart the JRun server in order for the changes to take affect. However, you can enable JRun's hot-deploy feature to dynamically redeploy the application without restarting the server.

To configure hot deploy:

1. Open the jrun.xml file under the SERVER-INF directory. The default location is *flex_install_dir*/jrun4/servers/default/SERVER-INF/jrun.xml.
2. Search for the DeployerService section (not the ClusterDeployerService section) within the jrun.xml file.
3. Add the following under <service class="jrun.deployment.DeployerService" name="DeployerService">:

```
<service name="Flex Web Application Factory"
  class="jrun.servlet.WebApplicationFactory">
  <attribute name="mandatoryFingerPrint">WEB-INF/flex/flex-webtier-
    config.xml
  </attribute>
</service>
```

4. Restart the JRun server.

Using virtual directories

You might want to store your application files and resources, such as JSPs, servlets, images, and MXML files, in a directory that is not physically in the JRun server's directory structure. You can do this by mapping a physical directory (or system path) to a virtual location (or resource path). Using virtual directories is especially common in environments that use content management systems or have large teams working on the same project.

To create virtual directories for a JRun application server, you edit the jrun-web.xml file in the server's /WEB-INF directory. You add a `<virtual-mapping>` block, with the `<resource-path>` defining the virtual location of the resources and the `<system-path>` defining the actual location.

The following example maps the c:/main/testing/EN directory to /EN:

```
<jrun-web-app>
  <virtual-mapping>
    <resource-path>/EN</resource-path>
    <system-path>c:/main/testing/EN</system-path>
  </virtual-mapping>
</jrun-web-app>
```

In this example, you can access files in the c:/main/testing/EN directory using a request URL similar to the following:

```
http://mycompany.com:8100/flex/EN
```

You can add virtual mappings to one jrun-web.xml file and have it apply to all applications running on the default server. To do this, you edit the {*jrun_install_dir*}/servers/default/default-ear/default-war/WEB-INF/jrun-web.xml file.

Enabling and disabling directory browsing

You can enable and disable directory browsing. Directory browsing is when a user requests a directory rather than a specific file. If directory browsing is enabled, JRun returns a list of the available files in that directory. If directory browsing is disiabled, JRun returns an error.

Directory browsing is enabled by default. To disable directory browsing, set the value of the `browseDirs` initialization parameter of the FileServlet servlet to `false` in the SERVER-INF/*application_name*-web.xml file, as the following example shows:

```
<servlet>
   <servlet-name>FileServlet</servlet-name>
   <servlet-class>jrun.servlet.file.FileServlet</servlet-class>
   <init-param>
     <param-name>browseDirs</param-name>
     <param-value>false</param-value>
   </init-param>
</servlet>
```

Configuring JRun logging

JRun defines a separate logger for each server. These loggers record application and server events such as initializations, requests, web application deployments, and shutdown messages. You configure the logger in the following file:

flex_install_dir/jrun4/servers/default/SERVER-INF/jrun.xml

You can change the format of the log messages, log levels, and enable JRun's metrics service using this file. By default, JRun logs error, warning, and info messages.

The default location of the default server's log file is *flex_install_dir*/jrun4/logs/default-event.log.

The following example defines the default logger configuration for the default JRun server:

```
<service class="jrunx.logger.LoggerService" name="LoggerService">
  <attribute name="format">{server.date} {log.level}
    {log.message}{log.exception}</attribute>
  <attribute name="errorEnabled">true</attribute>
  <attribute name="warningEnabled">true</attribute>
  <attribute name="infoEnabled">true</attribute>
  <attribute name="debugEnabled">false</attribute>
  <attribute name="metricsEnabled">false</attribute>
  <attribute name="metricsLogFrequency">60</attribute>
  <attribute name="metricsFormat">Web threads (busy/total):
    {jrpp.busyTh}/{jrpp.totalTh} Sessions: {sessions} Total
    Memory={totalMemory} Free={freeMemory}</attribute>
  <service class="jrunx.logger.ThreadedLogEventHandler"
    name="ThreadedLogEventHandler">
    <service class="jrunx.logger.ConsoleLogEventHandler"
      name=":service=ConsoleLogEventHandler"/>
    <service class="jrunx.logger.FileLogEventHandler"
      name="FileLogEventHandler">
      <attribute name="filename">{jrun.rootdir}/logs/{jrun.server.name}
      -event.log</attribute>
      <attribute name="rotationSize">200k</attribute>
      <attribute name="rotationFiles">3</attribute>
      <attribute name="closeDelay">5000</attribute>
      <attribute name="deleteOnExit">false</attribute>
    </service>
  </service>
</service>
```

The default logger configuration does not include debug messages. To enable debug messages, set the value of the `debugEnabled` attribute to `true`, as the following example shows:

```
<attribute name="debugEnabled">true</attribute>
```

The default logger configuration enables both file and console logging. To disable console logging, comment out the following line:

```
<!-- <service class="jrunx.logger.ConsoleLogEventHandler"
  name=":service=ConsoleLogEventHandler"/> -->
```

You can configure a JRun server to write to log-level-specific log files so that you have one log file for debug messages and a separate file for info messages, for example. You do this by adding the `{log.level}` variable to the `filename` attribute, as the following example shows:

```
<attribute name="filename">{jrun.rootdir}/logs/{jrun.server.name}-
  {log.level}-event.log</attribute>
```

The result is that JRun writes the following log files, each containing only the events of that log level:

- default-info.log
- default-debug.log
- default-warn.log
- default-error.log
- default-metrics.log (if metrics are enabled)

Using the sniffer

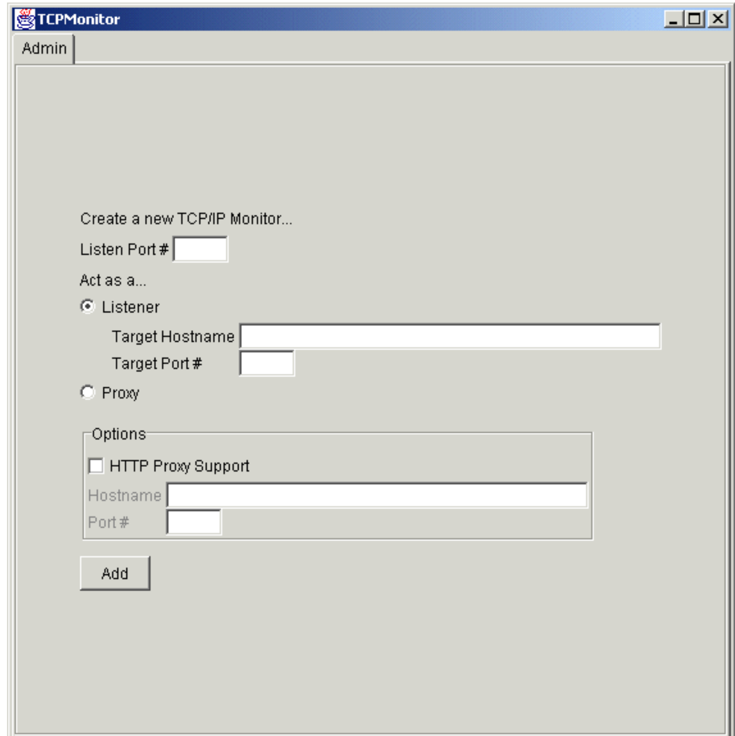
TCPMonitor is a swing-based application that lets you watch the request and response flow of HTTP traffic. You can also watch the request and response flow of data services messages. This application is commonly known as the *sniffer*.

When you use the sniffer, you request a resource but specify the sniffer's listening port. The sniffer then passes this request to the resource on the target port, and returns the results back to the requesting client. Because the sniffer is providing "tunneling" for your requests, it can display all the HTTP request and response traffic.

To run TCPMonitor:

1. On Microsoft Windows and Unix platforms, you can execute the TCPMonitor by launching the sniffer utility in the *flex_install_dir*/jrun4/bin directory.

   The TCPMonitor main window appears:



2. Enter the values in the main window as described in the following table:

Field	Description
Listen Port#	Enter a local port number, such as 8701, to monitor for incoming connections. Instead of requesting the usual port on which your server runs, you request this port. TCPMonitor intercepts the request and forwards it to the Target Port.
Listener	Select Listener to use TCPMonitor as a sniffer service in JRun.
Proxy	Select Proxy to enable proxy support for TCPMonitor.
Target Hostname	Enter the target host to which incoming connections are forwarded. For example, if you are monitoring a service running on your samples JRun server, the hostname is localhost.

Field	Description
Target Port#	Enter the port number on the target machine to which TCPMonitor connects. For example, if you are monitoring Flex on the default JRun server, enter 8700.
HTTP Proxy Support	Select this check box only to configure proxy support for TCPMonitor.

3. To add this profile to your TCPMonitor session, click Add.

   A tab appears for your new tunneled connection.

4. Select the new tab. If there are port conflicts, TCPMonitor alerts you in the Request panel.
5. Request a page using the Listen Port defined in this TCPMonitor session. For example, if you entered 8701 for the Listen Port, enter the following URL in your browser:

   ```
   http://localhost:8701/
   ```

   TCPMonitor displays the current request and response information. For each connection, the request appears in the Request panel and the response appears in the Response panel. TCPMonitor keeps a log of all request-response pairs and lets you view any particular pair by selecting an entry in the top panel.

6. To save results to a file for later viewing, click Save. To clear the top panel of older requests that you do not want to save, click Remove Selected and Remove All.
7. To resend the request that you are currently viewing and view a new response, click Resend. You can edit the request in the Request panel before resending, and test the effects of different requests.
8. To change the ports, click Stop, change the port numbers, and click Start.
9. To add another listener, click the Admin tab and enter the values as described previously.
10. To end this TCPMonitor session, click Close.

Index

Symbols

*.as files 34
*.css files 35, 111, 198
*.mxml files 34
*.swc files 34, 229
@Context token 193
@ContextRoot token 177

A

absolute positioning 115
absolute sizing 115
AC_OETags.js file 377, 394
accessibility
  compiler option 196
  enabling 358
ActionScript
  error handling 79
  keeping generated files 203
ActiveX controls
  activating 371
  upgrading 391
addToCreationQueue() method 147
advisories 85
alignment, SWF files 386
ALL log level 254
allowNetworking property 58, 380
allowScriptAccess property 58, 381
AMF, Secure 75
animations 117
anti-aliasing 117, 385
Application class 147
application compiler
  about 180
  options 196
  SDK 182
  using 195
  web tier 183
  See also compc
  See also mxmlc
web-tier compiler
applicationComplete event 133
applications
  breaking up 102, 234
  compiling 180
  debugging with fdb 270
  default mapping 176
  deploying 60, 351, 360
  embedding in wrappers 373
  files 156
  frame rate 198
  history management 377
  initialization time 89
  login 67, 68
  performance 88
  production mode 61
  profiling 90
  recompiling 96
  root directory 210
  sandboxes 56
  scaleability 88
  scaling 386
  security 51, 76
  SpeedStep 94
  transparency 388
  using SWC files 213, 231
  view source 77
archive option 381
archives. *See* SWC files
AS file type
  about 34
  preventing access 63
assets

- about 158
- externalizing 101
- loading local 66
- remote assets 65
- types 64

authentication
- about 67
- BASIC 68
- container-based 67
- FORM-based 68
- host-based 81
- passwords 81

authorization
- about 67
- passwords 81

axis gutters 128
AxisRenderer object 128

B

background color
- compiler option 198
- wrapper 382

BASIC authentication 68
benchmark, compiler option 196
bitmaps
- asset types 64
- backgrounds 116
- cachePolicy property 117
- optimizing 118
- quality 385

BOM. *See* Byte Order Mark
breakpoints 279
browsers
- cache files 95
- caching 59
- default 271
- disabling history management 125
- security APIs 57
- trusted sites 59

browsing directories 409
building projects 181
Byte Order Mark (BOM) 212
bytecode
- optimization 97

C

Cache-control header 59
cache.dep file 96, 172
cacheAsBitmap property 117, 118
cachePolicy property 117
caching
- cache.dep file 96
- Flash Player 118
- Flex Data Services 126
- fonts and glyphs 174
- HTTP headers 59, 95
- preventing 95
- security implications 59
- server configuration 172
- servers 62

callLater() method 90, 150
Canvas class 115
Capabilities class 250
catalog.xml file 239
CategoryAxis object 128
Channel class 76
channels 75
charts. *See* Flex Charting
charts.swc file 230
charts_rb.swc file 230
checklist, for deployment 360
child controls
- absolute positioning 115
- creating 139
- deferring creation 134
- destroying 142

childDescriptors property 141
class files
- adding to SWC files 224
- keeping generated 203

class selectors 113
classid property 382
client
- caching 59
- logging and debugging 251
- performance 88
- security 54
- *See also* Flash Player

clip masks 119
Clipboard 55
codebase property 382
codetype option 383
command-line compilers
- configuration 165
- leaf nodes 192
- options 196
- shell scripts 189

using 187
See also compc
See also mxmlc
command-line debugger. *See* fdb
compc
about 184, 186, 229
compiling 221
compiling components 220
examples 218
options 216
syntax 187
using 186, 215
See also component compiler
compile times, option 196
compilers
about 179
accessibility 358
adding metadata 211
advanced options 196
command-line configuration 165
configuration files 190
file encoding 212
Flex Data Services configuration 168
incremental 174, 214
library projects 185
logging 264
option precedence 194
options 188
tokens 193
using RSLs 240
using SWC files 213
viewing errors and warnings 207, 227
compiling
about 17, 154
case sensitivity 46
excluding files 199
incremental 124
JVM heap sizes 125
precompiling 63, 124
production mode 359
security 77
strict mode 97, 209
SWC files 229
component compiler
about 184
Flex Builder 186
using 215
See also compc
component descriptors 139
components
childDescriptors property 141
compiling 218, 220, 221, 229
creating 139
destroying 142, 143
distributing 231
lifecycle 133
manifest files 232
ordered creation 143
startup order 132
configuration files
about 153, 161
command-line compiler 165
data-management-config.xml 170
directory structure 162
dump-config compiler option 199
FlashPlayerTrust directory 84
flex-config.xml 165, 406
jrun-web.xml 176, 406
jrun.xml 406
jvm.config 166
loading 205
messaging-config.xml 170
mm.cfg file 83, 178
mms.cfg file 83
proxy-config.xml 170
remoting-config.xml 170
services-config.xml 170
syntax 191
web.xml 406
constraints 68
containers
absolute positioning 115
adding to the queue 144, 147
createComponents() method 139
createLater() method 147
creating 114
creationPolicy property 105, 135
deferred creation 134
multiple-view 136
navigator containers 136
single-view 136
content assets 64
content-size option 173
content-type 383
context root 176, 197
Controller (in Model-View-Controller) 14
cookies. *See* persistence
createComponents() method 139
createComponentsFromDescriptors() method 139
createLater() method 147

creation order 133
creationIndex property 145
creationPolicy property 106
about 105, 134, 145, 149
definition 134
multiple-view containers 136
ordered creation 144
credentials 68
cross-domain policy files 62, 65
cross-scripting 57
crossdomain.xml file 62, 65
CSS
default URL compiler option 198
file type 34
setStyle() method 110
CSSStyleDeclaration class 112, 113

D

data assets 64
Data Management Service, configuration 170
data services, securing 72
data sets, scrolling 120
data storage 54, 82
data validation 78
data-management-config.xml file 170
DataGrid class 120
DataService tag 76
DEBUG log level 254
debug SWF file 197
debug, compiler option 270
debugger
common commands 277
convenience variables 276
setting default browser 271
status 286
using breakpoints 279
See also fdb
debugger version of Flash Player 247
debugging
about 19, 154, 269
client-side 251
compiler option 197
debugger version of Flash Player 247
disabling 97
source command 281
suppressing debug output 81
See also fdb
declarative security 52, 60
default browser
definition 271
setting 271
default file 176
DefaultProperty metadata keyword 299
deferred creation
about 106, 132
example 150
using 134
deferred instantiation. *See* deferred creation
dependencies 98, 100
deploying
about 19
checklist 360
context root 176
Flex Data Services applications 355
headless servers 126
hot deploy 408
production mode 61, 359
RSLs 353
SDK applications 352
SWC files 230
using WEB-INF 62
deprecation, about 228
descriptors 139, 141
design patterns
about 14
Model-View-Controller 14
Struts 15
destinations
logging 253
securing 75
destroying components 142, 143
detection, player. *See* Express Install
directories, virtual 408
directory browsing 177, 409
disk storage 54, 82, 83
displayAsPassword property 81
doLater() method 150
domain-based security
ExternalInterface API 58
navigateToURL() method 58
domains, allowing access 62, 66
download times, reducing 97
drawing 117
drop shadows 128
dump-config compiler option 199
duration property, effects 116
dynamic linking 103, 205, 235

E

effects
  performance 115
elapsed time 90
embed tag
  about 371, 378
  example 379
  unsupported parameters 389
embedded fonts, caching 174
embedding assets
  about 101
  asset types 64
  remote assets 65
encoding 196, 212
endpoints 75
Enterprise Deployment Kit. *See* Express Install
environments, without Windows systems 201
Eolas update 371
ERROR log level 254
ErrorReportingEnable 249
errors
  handling 79
  viewing 227
event handlers, removing 108
event listeners. *See* event handlers
exceptions, security 79
Expires header 59
Express Install
  about 391
  adding 377, 392
  alternatives 397
  configuring on Flex Data Services 396
  disabling 125
  understanding the script 395
ExternalInterface API, security 58

F

fading 115
FATAL log level 254
faults, debugger 287
fdb
  about 269
  configuring 276
  invoking 272
  limitations 270
  shortcuts 271
  using 270
fds.swc file 230
fds_rb.swc file 230
file encoding 212
file system path 408
file types
  about 34, 156
  mapping 63
  preventing access 63
file-watcher-interval option 173
file-watcher-interval setting 173
files
  access 83
  browsing 177
  directory browsing 409
  encoding 196, 212
  size reduction 97
  temporary caching 59
FileServlet 178
filterData property 127
filters
  optimizing 119
  shadows 128
FireFox. *See* browsers
Flash Communication Server. *See* Flash Media Server
Flash Media Server 82
Flash Player
  caching 118
  callLater() method 90
  choosing the version 94
  configuration 178
  configuring debug version 248
  debug output 81
  debugger version 81, 247
  detecting version 250
  displaying errors 80
  embedding assets 64
  memory consumption 92
  mm.cfg file 83
  mms.cfg file 83
  performance 88
  persistent shared objects 82
  playback quality 385
  privacy 55
  redraw regions 118
  sandboxes 55
  security 54, 82
  Settings Manager 84
  Show Redraw Regions 118
  upgrading 377, 391
  See also ActiveX controls
Flash plugin. *See* Flash Player

flashlog.txt file 253
FlashPlayerTrust directory 84
flashVars property 383
Flex 2 SDK, directory structure 36, 49
Flex Builder
  application compiler 181
  building projects 181
  compiling applications 47
  component compiler 186
  directory structure 32
  disabling view source 77
  library projects 185
  wrapper 368
Flex Charting
  axis ranges 127
  AxisRenderer object 128
  CategoryAxis object 128
  directory structure 34
  gutters 128
  LinearAxis object 127
  optimizing 127
  shadows 128
Flex Data Management Service 72
Flex Data Services
  accessing 354
  caching 62, 96, 126
  compiling 48, 168
  compiling, incremental 124
  configuration 17, 170
  deploying web application 27
  directory structure 28, 33, 42, 49
  Express Install 125, 396
  file watcher interval 173
  Flex Data Management Service 72
  JRun 401
  JRun sniffer 96
  JVM configuration 125, 169
  logging 255
  performance 88
  preventing access 63
  production mode 61, 127
  proxy 71
  RPC services security 70
  securing endpoints 75
  securing JRun 63
  security overview 60
  web-tier compiler 183
  wrapper 368
flex web application 402
flex-admin web application 402
flex-config.xml file
  about 165, 406
  actionscript-file-encoding 212
  headless-server 126
flex.swc file 230
FlexForbiddenServlet 63
FlexMxmlServlet 171, 265
FlexSwfServlet 171
fonts, caching 174
form fields, validation 78
FORM-based authentication 68
frame rate, compiler option 198
frames 90, 150
framework.swc file 230, 236
framework_rb.swc file 230
frameworks, Struts 15
functions, queuing 150

G

garbage collection 54
gdb. *See* fdb
generated ActionScript, compiler option 203
getLogger() method 262
getQualifiedClassName() method 100
getTimer() method 89, 90
Global Flash Player Trust directory 84
global style sheets 110
glyphs, caching 174
Grid containers, optimizing 114

H

handlers. *See* event handlers
hardware disabling 83
headless servers
  about 360
  compiler option 201
  java.awt 127
heap size 125, 167
history management
  adding to wrapper 377
  disabling 125
history.js file 377
history.swf file 377
host-based authentication 81
hot-deploy 408
HTML templates
  about 370

Express Install 377, 392
history management 377
See also wrapper
HTML wrapper. *See* wrapper
HTTP headers
Cache-Control 96
Cache-control 59
caching 59
Expires 59, 96
If-Modified-Since 95
Last-Modified 95
Pragma 59, 96
Referer 81
viewing 411
HTTP requests, viewing headers 411
HTTP responses, viewing 411
HTTP sniffer 411
http-maximum-age option 173
HTTPS, security and 75
HTTPService class 70
HttpServletResponse, viewing headers 411

I

ILogger interface 253, 261
include classes, compiler option 218
incremental compilation
about 124, 174, 214
compiler option 202
disabling 359
index file 176
INFO log level 254
initialization
calculating 89
containers 114
styles 110
injection, input validation 78
input validation 78
installing, directory structure 31
instantiation, deferred 134
instantiation, queueing 144
Internet Explorer. *See* browsers
IP spoofing 81
isDebugger property 250

J

J2EE
authentication 67
authorization 67
security constraints 68
security overview 60
J2EE servers, securing JRun 63
JAAS. *See* Java Authentication and Authorization Service
Java Authentication and Authorization Service 67
Java security manager 67
Java Virtual Machine. *See* JVM
JavaScript
AC_OETags.js file 377
history.js file 377
noscript tag 371, 374
script tag 374
wrapper file 371, 374
JRun
about 401
adding and removing servers 405
application mapping 176
configuring servers 406
directory browsing 177, 409
FileServlet 178
hot-deploy 408
logging 410
port numbers 407
security 63
servlet mappings 63
sniffer 96, 411
starting 404
stopping 404
virtual directories 176, 408
web applications 408
jrun-web.xml file 176, 406
jrun.xml file 406
JSP file type, preventing access 63
JVM configuration
about 166
Flex Data Services 169
JVM heap size 125, 167
jvm.config file 166

K

keys, license 161

L

layout containers
optimizing 115

See also containers
layout, optimizing 114
libraries
- adding classes 224
- adding utility classes 224
- component 157
- component compiler 186
- creating 238
- including 202
- including classes 216, 238
- including files 216
- library path compiler option 204
- linking SWC files 201, 229
- linking, dynamic 103
- RSLs 234
- *See also* RSLs
- *See also* SWC files

library path
- compiler option 204
- example 213

library.swf file 239
license keys 161
licenses, version 210
link-report option 98
linking
- about 234
- externs, compiler option 199, 205
- including classes 202
- report 204
- static 103
- SWC files 201, 204, 229
- SWF file dependencies 98

List-based controls 120
listeners. *See* event handlers
live scrolling 120
loading assets, local 66
loadPolicyFile() method 66
local assets 66
local SWF files 61
local-trusted sandbox 56, 84
local-with-filesystem sandbox 55
local-with-networking sandbox 55
LocalConnection class 74
locale, compiler option 205
Log class 253, 262
log targets 253
LogEventLevel, disabling 81
logging
- about 155, 228, 245
- client-side 251
- compiler messages 264
- configuration 171
- custom loggers 259
- destinations 253
- disabling 81
- enabling trace() 83
- errors 249
- JRun 410
- Logging API 252
- message levels 254
- warning 249
- web applications 265
- web-tier compiler 265

Logging API
- custom loggers 259
- debugging output 81
- Flex Data Services 255
- using 252

login modules 67, 68
LogLevelEvent class 254

M

manifest files 231, 232
mappings
- default application 176
- virtual directories 175

Mark of the Web, about 373
masks 119
MaxWarnings 249
MD5 75
media, asset types 64
memory usage
- about 54
- background processing 116
- calculating 92
- disabling SpeedStep 94
- JVM heap size 125, 167

Message Service, configuration 170
MessageBrokerServlet 171
Messaging Service, MessageBrokerServlet 171
messaging tags 76
messaging-config.xml file 170
metadata 211
methods, queuing 150
MiniDebugTarget 253
mm.cfg file 83, 178, 248, 249
mms.cfg file 83
Model (in Model-View-Controller) 14

Model-View-Controller 14, 15
MOTW 373
movie quality 117, 385
MP3, asset types 64
multiple-view containers 134, 136
MVC. *See* Model-View-Controller
MXML file type
  about 34
  preventing access 63
MXML tags, security restrictions 76
mxmlc
  about 182
  options 196
  syntax 182, 187
  See also application compiler

N

namespace, compiler option 206
namespaces, compiling components 221
navigateToURL() method 58
navigator containers
  creationPolicy property 136
  definition 134
nesting containers 114
Netscape Navigator. *See* browsers
network assets 65, 66
noscript tag 371, 374

O

object creation 133
object tag 371, 378
Opera. *See* browsers
optimizing. *See* performance
ordered creation
  about 105, 132
  using 143

P

packages, compiling 220
packet sniffer 411
passwords 81
patterns. *See* design patterns
performance
  about 88
  ActionScript optimizer 206
  compiler option 196
  effects 115
  Flex Charting 127
  improving rendering 117
  incremental compilation 214
  JVM 125
  production mode 127
  RSLs 41, 104, 233
  startup performance 131
persistence
  security 54
  SharedObject class 82
player detection, disabling 125
player. *See* Flash Player
playerglobal.swc file 230
playerProductInstall.swf file 392
plugin. *See* Flash Player
pluginspage property 385
policy files
  cross-domain 65
  example 66
ports
  JRun 407
  securing 62
  sockets 73
positioning child controls 115
Pragma header 59
precompiling 124
preferredIndex 147, 149
privacy
  concerns 55
  mms.cfg file 83
production mode 61, 127, 175, 359
profiling
  disabling SpeedStep 94
  in ActionScript 90
programmatic security 52, 60
projector, defined 74
projects, building 181
proxy
  deployment checklist 364
  destinations configuration 170
  RPC services 71
  secure endpoints 75
proxy-config.xml file 170

Q

quality property 117, 385
queued creation 105

R

RAM
- JVM heap size 125, 167
- usage 92

Real-Time Messaging Protocol. *See* RTMP
realms 68
recycleChildren property 122
redraw regions 117, 118
Remote Procedure Call. *See* RPC
remote sandbox 55
RemoteObject
- class 70
- service configuration 170

remoting-config.xml file 170
removeChild() method 107
removeChildAt() method 107, 142
removeEventListener() method 108
rendering 117
Repeater class 120
reports, linking 204
request object, viewing headers 411
resource bundles
- about 158
- compiling 205, 207, 217

resources, HTML templates 370
roles 68
RPC 70, 76
rpc.swc file 230
RSLs
- about 157
- benefits 237
- compiler option 207
- considerations 235
- creating libraries 238
- defined 41, 104, 233
- deploying 353
- example 241
- linking 103
- using 240

RTMP
- Secure 75
- securing 73
- securing endpoints 75
- using 75

run -time performance 88
Runtime Shared Libraries. *See* RSLs

S

Safarai. *See* browsers
samples web application 402
sandboxes
- client security overview 53
- defined 55
- determining current type 56
- FlashPlayerTrust directory 84
- local-trusted 56
- local-with-filesystem 55
- local-with-networking 55
- persistent data storage 82
- remote 55
- use-network option 56

sandboxType property 56
scaleability 88
scaling, SWF files 386
screen refresh 150
script limits, compiler option 198
script tag 374
scripting. *See* cross-scripting
scriptTimeLimit property 95
scrolling 120
Secure AMF 75
Secure RTMP 75
security
- about 21, 51
- advisorires 85
- allowScriptAccess property 381
- asset types 64
- authentication 67
- authorization 67
- browser APIs 57
- caching 59
- client 53, 54, 82
- cross-domain policy files 62, 65
- data services 72
- data transport 74
- disabling view source 77
- domain-based restrictions 58
- encryption 75
- endpoints 75
- error handling 79
- FlashPlayerTrust directory 84
- host-based authentication 81
- J2EE 60
- JRun 63
- LocalConnection 74
- login modules 67

mm.cfg file 83
mms.cfg file 83
navigateToURL() method 58
other resources 85
precompiling 63
RPC services 70
RTMP 73
sandboxes defined 55
Settings Manager 84
sockets 73
SSL 74
types 52
using input validation 78
security advisories 85
Security class
allowDomain() method 57
allowInsecureDomain() method 57
sandbox type 56
Security Resource Center 85
Security Topic Center 85
SecurityError class 79
servers
adding new JRun servers 405
caching 62, 126, 172
configuring JRun servers 406
deployment checklist 363
headless 360
incremental compiling 124
JRun 63, 401
logging 228, 410
performance 88
security 60
services-config.xml file 170
servlets
FileServlet 178
FlexForbiddenServlet 63
FlexMxmlServlet 171
FlexSwfServlet 171
logging 265
mappings 63
MessageBrokerServlet 171
setStyle() method, optimizing 110
setStyleDeclaration() method 112, 113
Settings Manager 84
shared libraries. *See* RSLs
SharedObject class
security 54
using 82
shell scripts 166, 189
Show Redraw Regions 118
single-view containers 136
sizing controls 115
sniffer 96, 411
sockets 73, 75
source code, disabling view source 77
source path, compiler option 196, 208
SpeedStep 94
spoofing 81
SQL
input validation 78
queries 78
SSL 74
stacktraces
compiler option 210
debugger 286
startup time
about 131, 132
calculating 89
containers 114
deferred creation 106
object creation 104
ordered creation 105, 143
performance 88
reducing 97
RSLs 41, 104, 233
styles 110
static linking 103, 234
storage 54
strict mode 97, 209
Struts 15
style sheets
default, compiler option 198
optimizing 110
styles
AxisRenderer object 128
optimizing 110
setStyle() method 110
setStyleDeclaration() method 112, 113
SWC files
about 156, 229
adding classes 224
caching 229
compiling 186, 218, 220, 221
component compiler 184
distributing 230, 231
file type 34
including classes 216
including files 216
libraries 103
linking 201, 204, 234

manifest files 231, 232
resource bundles 158, 217
themes 157
using 41, 47, 213, 231
SWF files
bytecode optimization 97
caching 59
compiling 17
cross-scripting 57
debugging 197, 270
deploying 19, 60, 351
embedding in wrappers 373
file type 34
FlexSwfServlet 171
including classes 202
libraries 103
linker dependencies 98
loading local assets 66
loading SWF files 58
LocalConnection objects 74
metadata 197, 211
precompiling 63
quality property 117, 385
reducing file size 97, 235
RSLs 104
sandboxes 56
securing 51, 77
transparent 388
trusted 59, 61
using multiple 102
viewing 287
See also applications
SWFLoader class 76, 102
System class, Clipboard 55
system path, virtual directories 175

T

TCPMonitor 411
templates. *See* HTML templates
testing, for performance 88, 89
themes
compiler option 209
compiling 229
creating 226
files 157
timeout 95
Timer class
getTimer() method 89
using 90
tokens 193
trace() method
debugging output 81
enabling 83, 249
file name 249
using 252
TraceOutputFileEnable property 249
TraceOutputFileName property 249
TraceTarget
disabling 81
filters 262
TraceTarget class, using 253
transparent SWF files 388
trusted
local SWF files 61
sandbox 56
sites 59
type checking 100

U

upgrading 161
URL patterns
about 63
security constraints 68
use-network option
local SWF files 61
sandboxes 56
settings 209
user experience, instantiation order 134
user roles 68

V

validation 68, 78
Validator class 78
VBScript, script tag 374
version detection. *See* Express Install
video files, asset types 64
View (in Model-View-Controller) 14
viewSourceURL property 77
virtual directories 175, 408
virtual machine heap sizes 125
virtual machine. *See* JVM

W

WARN log level 254

warnings
- about 249
- compiler options 207, 210
- viewing 227

web applications
- creating on JRun 408
- default 402
- directory structure 357
- flex 402
- flex-admin 402
- logging 265
- samples 402
- using 20

web-tier compiler
- about 183
- caching 173
- configuration 168
- context root 176
- directory browsing 177
- incremental compilation 174, 214
- logging 228, 265
- options 184
- production mode 175, 359
- virtual directories 175
- *See also* application compiler

web.xml file
- about 406
- example 176
- security constraints 68
- servlet mappings 63

WebService class 70

welcome file 176

Window-less environments 201

wmode property 388

wrapper
- about 367
- adding Express Install 377, 392
- adding features 375
- adding history management 377
- alternate content 395
- creating 370
- defined 20
- disabling Express Install 125
- Express Install 391
- ExternalInterface API 58
- Flex Builder 368
- Flex Data Services 368
- HTML templates 370
- Mark of the Web 373
- navigateToURL() method 58
- noscript tag 371
- object and embed tags 371, 378
- quality property 117
- script tag 374
- security 381
- timeout 95
- *See also* HTML templates

X

XML sockets 73

XMLSocket class 75