developer Works

유니버설 미들웨어 프레임워크 – OSGi

OSGi 개발 환경 구현 2 – OSGi Bundle 구현

지난 시간에는 OSGi 개발 환경 구축에 앞서 OSGi 애플리케이션 구현을 위한 실행과 개발 환경에 대해 살펴봤다. 그 흐름을 이어서 이번 시간에는 OSGi 애플리케이션(Bundle)을 직접 구현하고 OSGi 프레임워크에 등록, 수행, 테스트, 그리고 원격 관리를 수행하는 일련의 번들 라이프 사이클을 살펴보기로 한다.

6

연 재 순 서

1회 | 2007. 7 | 임베디드를 넘어 엔터프라이즈로!
2회 | 2007. 8 | OSGi 서비스와 활용 사례
3회 | 2007. 9 | OSGi 개발 환경 구현 1 – J2ME & OSGi
4회 | 2007. 10 | OSGi 개발 환경 구현 2 – OSGi Bundle 구현
5회 | 2007. 11 | 이클립스의 핵심, Equinox
6회 | 2007. 12 | 엔터프라이즈로의 확장, Spring-OSGi

김석우 suhgoo.kim@samsung.com |
Polytech 전산학 석사. 현재 삼성전자 선행개발팀에 근무 중이며 센서 네트워크를 활용한 빌딩의 쾌적제어 시스템을 개발하고 있다. 또한 OSGi 기반의 센서 네트워크 컨트롤러를 구상하고 있다.

많은 OSGi 관련 툴들이 나와 있지만 이번 호에서도 역시 IBM의 웹스피어(Websphere) 솔루션을 사용해 구현해 보기로 한다. 구현할 내용은 다음과 같다.

- HttpLogService를 이용한 번들 구현
- OSGi 프레임워크 상에서 번들 Test

IBM에서는 OSGi 관련 솔루션들을 IBM Workplace Client Technology, Micro Edition이라는 이름으로 One-stop 솔루션을 제공하는데, 이클립스 기반의 통합개발 툴인 WSDD (WebSphere Studio Device Developer), OSGi 기반의 미들웨어 툴킷 Micro Environment Toolkit for WebSphere Studio, J2ME 런타임 모듈과 DB2e를 제공하는 WebSphere Everyplace Micro Environment로 구성되어 있다. SMF (Service Management Framework)는 IBM의 OSGi 솔루션 이름이면서 또한 Front-End Solution의 핵심 컴포넌트 가운데 하나이다. IBM의 Front-End Solution을 Workplace Client Technology라고 통칭하기도 하는데 주로 PC 기반의 비즈니스 애플리케이션, 디바이스 콘트롤, 임베디드 및 모바일 디바이스를 타깃으로 운용된다.

OSGi 번들 아키텍처

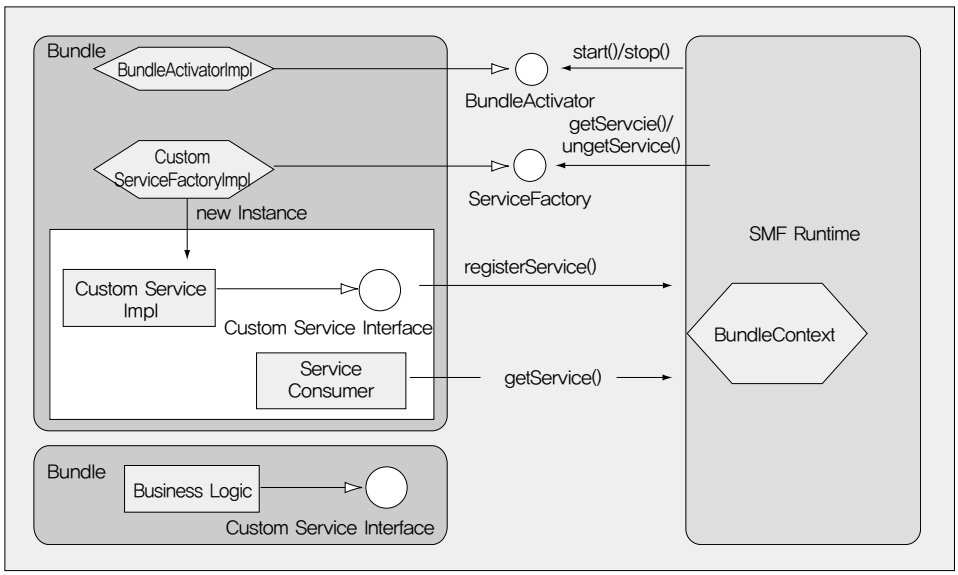
OSGi에서는 모든 기능과 서비스들이 번들에 의해 구현되고 서비스로 운영된다. 번들이 기본적인 구성요소인 이유로는 느슨한 결합(loosely-coupling)과 재사용성(reusability)에 있는데, 번들은 Dependency static sharing과 dynamic services로 각각의 독립성(Dependency)을 유지하게 된다. 그렇다면 번들의 독립성이란 어떤 의미일까? OSGi는 같은 번들이라고 해도 다른 버전의 번들을 동시에 사용할 수 있다. 1.0의 A 번들과 1.5의 번들이 함께 구동될 수 있다는 뜻이다. 그렇게 운영되는 것은 번들이 다음의 구성요소들로 구분되고 요청하는 번들이나 서비스에 인터페이스로 제공되기에 가능하다.

● Bundle manifest file

번들의 가장 기본적인 명칭, 버전, 제공 기능 및 다른 번들과의 static & dynamic interaction 방법들을 서술하고 있다.

● Bundle activator implementation

우리는 번들이 OSGi 시스템 프레임워크로부터 install, update, start, stop되는 일련의 라이프 사이클을 가지고 있음을 알고 있다. 이렇게 번들이 시스템 프레임워크와 연동되고 컨트롤할 수 있게 해주는 것이 바로 OSGi BundleActivator interface이다.



〈그림 1〉 Bundle Process Flowdiagram

● Custom service interface

번들이 반드시 R3, R4의 규약에 나온 서비스들로만 구현될 수 있는 것은 아니다. 각 벤더에서 독자적인 번들을 서비스 형태로 구현할 수 있는데, 이렇게 각 벤더들이 각각 독자적으로 구현한 번들과 서비스를 사용하려고 할 때 필요한 것이 바로 Custom service interface이다.

● Custom service implementation

custom service interface를 정의한 Class 메소드

● Business logics

Business logics 부분은 일종의 번들 제공 기능의 알고리즘이면서 블랙박스라고 할 수 있다. 또한 Java class file이나 다른 resource file을 Invoke해서 사용할 수도 있다.

〈그림 1〉은 OSGi 시스템 프레임워크(IBM에서는 SMF Runtime)와 service consumer bundle 그리고 service provider bundle간의 interaction을 보여주고 있다.

HttpLogService를 이용한 번들 구현

WSDD(WebSphere Studio Device Developer)를 이용해 HttpLog라는 번들을 구현해 보기로 한다. HttpLog 번들은 특정한 HttpServlet에 의해 엑세스된 트랜잭션의 로그 메시지를 추적하고 기록하는 기능을 가진다.

Step 1 : Bundle skeleton 생성

WSDD에서 자바 프로젝트를 통해 HttpLog라는 번들 skeleton을 만든다. 또한 Properties를 통해 OSGi framework library osgi.jar와 servlet.jar의 build path를 수정한다.


〈화면 1〉 Java Build Path 수정

Step 2 : OSGi BundleActivator interface 구현

이제 HttpLogBundleActivator라는 자바 클래스를 작성하면서 OSGi BundleActivator interface를 구현해 보기로 한다. BundleActivator interface는 두 개의 메소드를 정의하는데 start()와 stop()이 바로 그것이다. start() method는 번들이 active될 때 OSGi 시스템 프레임워크로부터 invoke된다. 〈리스트 1〉의 코드에서 BundleContext argument는 delegation object로 프레임워크로부터 invoke될 때 구동되는 Operation set이다.

〈리스트 1〉 HttpLogBundleActivator.java

```
public class HttpLogBundleActivator implements
BundleActivator {
    …
    BundleContext bc;
    public void start(BundleContext context) throws
```

```
Exception {
        bc=context;
        //to add initialization code here

    }
    public void stop(BundleContext context) throws
Exception {

    }
    …
}
```

Step 3 : Register custom service

이제 HttpLogService라는 로그기록을 관리하는 custom service를 구현하기로 한다. 총 3개의 자바 파일을 코딩하는데, 명칭은 각각 다음과 같다. HttpLogService.java, HttpLog ServiceImpl.java, HttpLogServiceFacotry.java. 우리는 custom service를 위한 하나의 interface class와 implement ation class를 구현한다. 예를 들어 HttpLogService interface의 경우 log()라는 하나의 메소드를 정의한다.

〈리스트 2〉 HttpLogServiceImpl.java

```
package httplog;
public interface HttpLogService {
    void log(String message);
}
```

〈리스트 3〉 HttpLogService.java

```
package httplog;
public class HttpLogServiceImpl implements HttpLogService{
    static public List messageArray = new ArrayList();

    public void log(String message) {
        HttpLogServiceImpl.messageArray.add(message);
    }

}
```

일반적으로 어떤 custom services를 요청하는 번들(hosting bundle)이 active될 때 해당 custom services는 시스템 프레임워크에 등록된다. 따라서 Step2에서 구현했던 HttpLogBundle Activator 클래스에 〈리스트 4〉의 start() method를 추가한다.

〈리스트 4〉 HttpLogBundleActivator.java

```
public class HttpLogBundleActivator implements
BundleActivator {
static final String httpLogServiceName =
HttpLogService.class.getName();
    …
public void start(BundleContext context) throws Exception
{
        registerHttpLogService();
    }
private void registerHttpLogService(){
        HttpLogServiceFactory sf = new
HttpLogServiceFactory();
        ServiceRegistration
sr=bc.registerService(httpLogServiceName,sf,null);
    }
}
```

또한 registerService()가 호출될 때 해당 번들은 정의된 custom service name과 OSGi ServiceFactory interface의 서비스 오브젝트, 그리고 해당 서비스의 properties가 포함된 Dictionary object를 제공한다. 각각의 ServiceFactory subclass는 getService()와 ungetService()라는 두 개의 메소드를 갖게 된다. 우리가 작성하는 코드에서는 sf가 하나의 HttpLog Ser viceFactory 타입이면서 또한 HttpLogServiceImpl instances를 generate하는 역할을 한다.

〈리스트 5〉 HttpLogServiceFactory.java

```
package httplog;
import org.osgi.framework.Bundle;
import org.osgi.framework.ServiceFactory;
import org.osgi.framework.ServiceRegistration;

public class HttpLogServiceFactory implements
ServiceFactory{
…
    public Object getService(Bundle bun,
ServiceRegistration sr) {
        return new HttpLogServiceImpl();
    }

    public void ungetService(Bundle bun,
ServiceRegistration sr, Object obj) {
    }
}
```

Step 4 : Consume external service

우리가 작성하는 HttpLog 번들에 의해 제공되는 외부 서비스는 바로 HttpService이다. 사용자는 localhost의 HttpServlet이 액세스한 기록들을 HttpLog 번들이 작성한 레코드를 통해 추적할 수 있다. 이러한 기능을 제공하기 위해 HttpLog 번들은 HttpService가 제공하는 인터페이스를 통해 HttpService 번들을 servlet object로 등록해야만 한다. HttpLog 번들이 start될 때 위의 과정이 자동적으로 발생한다.

〈리스트 6〉 HttpService에 특정 HttpServlet object 등록

```
public class HttpLogBundleActivator implements
BundleActivator {
    …
    static final String servletURI ="/httplog";
    private void initialize(){

        //Import HttpService
        attachToHttp();
        //register HttpLogService
        registerHttpLogService();

    }
    private void attachToHttp(){
        httpContext = new HttpContext(){

            public boolean handleSecurity(HttpServletRequest
request,
                                          HttpServletResponse
response)
            {
                return true;
            }

            public URL getResource(final String name)
            {
                return (URL)
AccessController.doPrivileged(new PrivilegedAction()
                {
                    public Object run()
                    {
                        String resource = name;
                        if (resource.charAt(0) != '/' )
                            resource = "/".concat(resource);
//$NON-NLS-1$
                        return
getClass().getResource(resource);
                    }
                });
            }

            public String getMimeType(String name)
            {
                return null;
            }

        };

        ServiceReference httpSR =
bc.getServiceReference(HttpService.class.getName());
        HttpService http =
(HttpService)bc.getService(httpSR);

        try {
            http.registerServlet(servletURI, new
HttpLogServlet(), null, httpContext);
        } catch (ServletException e) {
            e.printStackTrace();
        } catch (NamespaceException e) {
            e.printStackTrace();
        }
    }
    …
}

//HttpLogServlet.java
package httplog;

import java.io.IOException;
import java.io.PrintWriter;
import java.util.List;

import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import javax.servlet.ServletException;

public class HttpLogServlet extends HttpServlet{

    protected void doGet(HttpServletRequest req,
HttpServletResponse res)
        throws ServletException, IOException
     {
        res.setContentType("text/html; charset=UTF-8");
        PrintWriter out = res.getWriter();

        out.print("<html><head><title>" + "Http Log
Display" + "</title></head>");
        out.print("<body text=\"#000000\" bgcolor=\
"#C0C0C0\"
            link=\"#0000EE\" vlink=\"#551A8B\" alink=
\"#FF0000\">");
        List msgArray = HttpLogServiceImpl.messageArray;

        int len=msgArray.size();
        for (int i=0;i<len;i++){
          String msg= (String)msgArray.get(i);
          out.print(msg);
        }
        out.println("</body></html>");
     }

}
```

〈리스트 6〉을 살펴보면 번들이 초기화될 때 특정한 Http Servlet object가 HttpService에 등록된다. 일반적으로 external service가 구동되기 전에 요청 번들은 OSGi 시스템 프레임워크로부터 external service instance가 어디에 위치해 있는지를 알 필요가 있다. 이러한 일련의 과정은 BundleContext에서 get ServiceReference()와 getService() 메소드를 제공함으로써 수행된다. 두 개의 메소드는 필요로 하는 정보들을 service regis

try에서 찾아 리턴하는 역할을 한다.

Step 5 : Migrate to bundle format

코드를 모두 작성하고 나면 실제 번들을 패키지 형태로 작성하는 과정을 거친다. WSDD에서 File -> New -> SMF를 고르고, 번들 폴더를 선택해서 새로운 번들을 생성한다.

〈화면 2〉
번들 폴더 생성

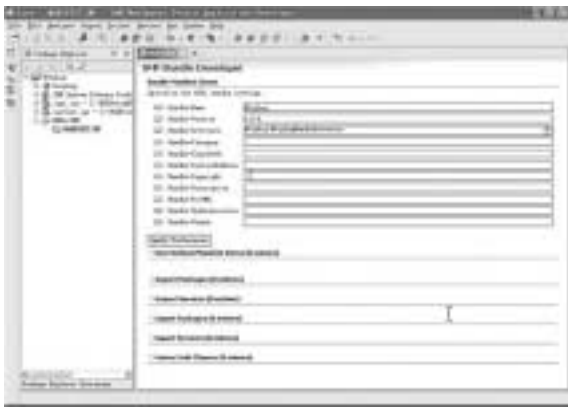
타깃 번들 폴더에서 HttpLog를 선택한다. 나머지 옵션은 선택하지 않는다.

〈화면 3〉 번들 폴더 컨테이너 선택

다음 단계로 MANIFEST를 작성할 단계이다. MANIFEST.MF 파일은 번들들의 독립성과 정보 공유 및 협업을 위해 매우 중요한 정보 파일이다. 반드시 일정한 OSGi 명세(specification)에 의해 작성되어야 한다. WSDD의 MANIFEST editor는 〈화면 4〉처럼 사용자가 쉽게 작성할 수 있게 도움을 준다.

다음은 섹션을 필수적으로 정의해야 하는 항목들로 우리가 작성한 파일을 예로 든 내용이다.

- Bundle-Name : HttpLog.



〈화면 4〉 MANIFEST 작성 화면 (1)

- Bundle-Version : 1.0.0.
- Bundle-Activator : httplog.HttpLogBundleActivator
- Import Services : org.osgi.service.HttpService
- Import Packages : javax.servle, javax.servlet.http, org.osgi.framework, org.osgi.service.http.

OSGi는 하나의 JVM 환경에서 작동한다. 그러나 하나의 VM이라고 하나의 프로세스 또는 싱글 태스크라고 생각하면 안 된다. 앞에서도 이야기했지만, OSGi는 비록 하나의 VM 위에서 구동하더라도 번들간의 독립성과 협업을 위해 각 번들마다 서로 다른 클래스 로더가 구동해 작동하게 되어 있다. 따라서 각 번들은 MANIFEST 파일에서 자신의 클래스 패스와 버전, 기본적인 정보들을 레퍼런스하게 된다.

- Export Packages : httplog.
- Export Services : httplog.HttpLogService.

〈화면 5〉
MANIFEST 작성
화면 (2)

MANIFEST.MF 파일까지 생성하면 이제 번들을 배포할 준비는 모두 마친 셈이다. 번들의 배포는 .jar 패지키 형태로 생성해 이뤄지며 여기서 우리는 httplog.jar란 이름으로 생성한다. WSDD의 Project explorer view에서 HttpLog project를 Export하면 된다.

Step 6 : OSGi 서버로 번들 배포

작성한 번들을 배포하려면 OSGi 시스템 프레임워크가 탑재된 시스템을 찾고 그곳에 추가하면 된다. WSDD에서는 이런 모든 과정을 매우 쉽게 접근할 수 있도록 돕는데, 이를 위해 다음의 방법을 사용한다. 우선 〔Run〕 Window를 실행시켜 OSGi 서버(SMF Server)를 생성하고 구동한다.

〈화면 6〉 SMF Server 생성 및 구동

〈화면 7〉 Bundle Deployment

〈화면 8〉
타깃 디바이스
/ 서버 설정

이렇게 SMF Server를 구동한 후에 Project Explorer에서 〔HttpLog project〕를 클릭하고 SMF -〉 Submit 번들을 선택하면 번들의 배포가 끝난다.

SMF 서버 구동시에 서버가 되는 타깃에 대한 대화상자가 나오는데, 그곳에 아이디와 패스워드, IP 등에 대한 기본적인 액세스 정보들을 적는다. 우리는 로컬PC에서 작업을 하므로 'Admin@localhost:8080/smf'로 기입한다. 이렇게 모든 서버의 구동과 번들의 배포를 마친 후에 〔SMF perspective〕-〔SMF Bundle Servers view〕에서 HttpLog 번들이 SMF 서버에 배포된 것을 확인할 수 있다. IBM 솔루션에서는 이러한 모든 과정들이 WSDD의 다이얼로그 박스(Dialog Box)에서 자동으로 이뤄지지만, 다른 여타의 OSGi 솔루션에서는 각기 다른 방법을 써서 로컬 서버 및 원격 서버(실제 OSGi가 설치된 PC 및 타깃 디바이스)로 배포할 수 있다.

〈화면 9〉 배포된 번들 확인

Step 7 : Bundle testing

이제 우리가 구현한 번들을 배포했고 테스트하는 단계만 남았다. 이 역시도 테스트 번들을 구현하기로 한다. 테스트 번들을 HttpLogTester로 명명하고, 이 번들은 테스트용으로 HttpLog Service를 invoke하는 것이 주된 기능이다. 이 번들이 HttpLog Service를 invoke하면 그 다음 단계로 넘어간다. 여기서 Http log가 화면에 정상적으로 나타난다면 우리가 만든 번들이 성공적으로 작성된 것으로 본다. 이 번들 역시 우리가 거쳐 왔던 step 1, 2를 반복해 bundle skeleton을 생성한다. 빌드 패스에 httplog.jar를 포함하며 〈리스트 7〉의 코드와 같이 HttpLog TesterBundleActivator에 start() method를 코딩한다.

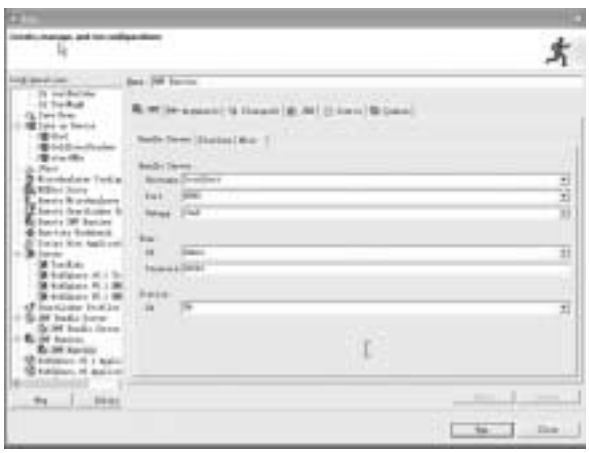
Step 2-6을 반복하며 마지막으로 역시 HttpLogTester 번들을 SMF 서버에 배포한다.

〈리스트 7〉 HttpLogTesterBundleActivator.java

```
class HttpLogTesterBundleActivator implements
BundleContext{
…
    public void start(BundleContext context) throws
Exception {
        bc=context;
        ServiceReference
httpLogSR=bc.getServiceReference(httpLogServiceName);
        HttpLogService httpLog
=(HttpLogService)bc.getService(httpLogSR);
        httpLog.log("Hello world!");
    }
…
}
```

Step 8 : Test HttpLog in SMF runtime

이제 우리는 배포된 HttpLogTester 번들을 실제 install-activate시키는 과정을 거쳐 우리가 만든 번들이 제대로 작동되는지를 확인해야 한다. SMF Runtime을 실행시키고 HttpLogTester를 인스톨한다. 이 모든 과정 역시 WSDD의 SMF Bundle Server view에서 지원된다.



〈화면 10〉 SMF Bundle Server view

〈화면 11〉 번들 인스톨 화면

HttpLogTester 번들이 SMF 번들 서버에 인스톨되는 순간 필요한 번들을 요청하게 되어 HttpLog와 HttpService는 자동적으로 함께 인스톨되고 ready하게 된다.

이제 로그 메시지를 확인하기 위해 웹 브라우저에 'http://localhost/httplog'라고 입력하면 곧 우리는 브라우저에서 해당 로그 결과를 살펴볼 수 있다.

이번 호에서는 OSGi 번들의 기본적인 구조를 비롯해 코드를 통한 구현과 배포, 실행에 대해 각각 살펴봤다. 예제 코드에서 로컬 서버와 번들간의 인터페이스들을 살펴봤지만, 실제 구현되는 코드에서는 DB와의 연동과 원격관리 서버를 통한 업데이트 및 관리, 네이티브 코드(Native Code)와의 연동 등을 통해 OSGi의 장점을 최적화해 사용하고 있다. 특히 방금 이야기한 DB handling & Sync, Bundle Remote Management & Dynamic Update, JNI를 통한 Native code interface 등은 OSGi의 고급 핵심 기술들로 향후 더 많이 사용될 중요한 부분이다.

여타의 OSGi 솔루션들도 마찬가지겠지만, 특히 IBM의 OSGi 솔루션은 이클립스 플러그인 프레임워크(Equinox)의 비즈니스 프로세스 애플리케이션(Expeditor) 등을 거쳐 더욱 강력한 미들웨어 프레임워크(Middleware Framework)로 부각되고 있다. 다음 호에서는 RCP(Rich Client Platform)로 대변되는 이클립스 OSGi 프레임워크인 Equinox를 통해 임베디드에서 데스크탑 애플리케이션, 그리고 플랫폼의 일부분으로까지 확장되는 OSGi의 사례를 살펴본다.

참고 자료

1. Managed mobile clients with OSGi – http://www.ibm.com/developerworks/library/wi-osgi
2. IBM Workplace Client Technology, Micro Edition and Open Services Gateway Initiative (OSGi) –http://www.ibm.com/developerworks/websphere/library/techarticles/0606_salkosuo/0606_salkosuo.html
3. IBM Service Management Framework – http://www-306.ibm.com/software/wireless/smf/index.html