Four Days on Rails

compiled by Ki-sun Baek, Dae-yeop Lee

목차

도입

요즘 들어 레일즈에 대한 여러 가지 다소 터무니 없는 주장들이 나오고 있는데, 가령 OnLAMP.com에 게재되었던 한 기사[1]를 예로 들자면, "일반적인 자바 프레임워크를 사용하는 것에 비해 레일즈를 사용할 경우 최소한 10배는 빠르게 웹 애플리케이션을 개발할 수 있다"라고 주장했던 것도 있었다. 이어서 그 기사에서는 PC에 레일즈와 루비를 설치하는 법을 보여주고 실제로 아무런 코딩도 하지 않고 작동하는 '스캐폴드' 애플리케이션을 만드는 것을 보여주었다.

이것이 다소 인상적이긴 하지만 '실제' 웹 개발자들은 그것이 감쪽같이 사람을 홀리는 마술과도 같음을 알 것이다. '실제' 애플리케이션은 그것만큼 단순하지는 않기 때문이다. 그러면 실제로 그 내부는 어떻게 동작하고 그것을 이용해서 '실제' 애플리케이션을 구축하는 것은 얼마나 어려울까?

그런데 여기서부터가 순탄치 않다. 레일즈는 사실 온라인상으로 문서화가 잘 되어 있는 편이고, 30,000개가 넘는 단어들이 수록되어 있는 레퍼런스 매뉴얼 형식의 온라인 문서들 때문에 어쩌면 레일즈를 처음 시작하는 입장에서는 과도하게 문서화가 되어 있는 것으로 느낄지도 모르겠다. 그리고 여러분이 레일즈로 개발할 때 어떤 페이지로 시작할지 알아야 할 필요가 있을 주요 페이지들에 대한 로드맵(레일맵?)은 매뉴얼에 있지 않다.

이 문서는 그러한 틈을 채우는 것에 대하여 설명할 것이다. 여기에서는 이미 여러분의 PC에 루비와 레일즈가 설치되어 있다고 가정한다(아직 설치되어 있지 않다면 Curt가 작성한 기사(Rolling with Ruby on Rails)의 내용에 따라 루비와 레일즈를 설치하길 바란다). 그 기사는 '레일즈 1일차'의 끝으로 여러분을 데려다 줄 것이다.

'레일즈 2일차'에서는 앞서 언급했던 마술의 내부를 살펴볼 것이다. 거기서 여러분은 '스캐폴드' 코드를 보게 될 것이다. 새로운 기능들은 진하게 표시하고 설명은 텍스트로 표시할 것이다. 그리고 각 내용에 대해 좀 더 알아볼 수 있는 레일즈나 루비 문서에 대한 레퍼런스도 달아놓을 것이다.

'레일즈 3일차'에는 스캐폴드를 이용하여 '진짜' 애플리케이션다운 무언가를 만들어 볼 것이다. 그러는 사이 여러분은 자신만의 레일즈 도구상자를 만들게 될 것이다. 가장 중요한 것은 여러분들 또한 온라인 문서화에 익숙해져서 나중에 여러분 스스로 탐험을 계속할 수 있도록 할 수 있어야 한다는 것이다.

'레일즈 4일차'에서는 다른 테이블이 추가될 것이고 관계 무결성(relational integrity) 관리에 있어 몇 가지 복잡한 문제들에 대해 다뤄볼 것이다. 마지막으로 여러분은 실제로 작동하는 애플리케이션을 갖게 될 것이며 이것은 여러분이 레일즈를 시작하기에 충분할 것이다. 그리고 도움이 필요할 경우 어디에서 도움될만한 것들을 찾아보아야 할지에 대해서도 알게 될 것이다.

10배는 빠르게 해보면 안되겠느냐고? 일단 4일 동안만 먼저 해본 다음 판단하시라.

문서화: 이 문서에는 다음의 레퍼런스를 포함하고 있다:

- *Documentation* – http://api.rubyonrails.com에서 볼 수 있는 레일즈 문서(이 문서는 젬 설치의 일부로서 여러분 PC의 다음 위치에 함께 설치된다: `C:\Program Files\ruby\lib\ruby\gems\n.n\doc\actionpack-n.n.n\rdoc\index.html`)

- *Ruby Documentation* – "Programming Ruby – The Pragmatic Programmer's Guide"는 온

1 *Rolling with Ruby on Rails,* Curt Hibbs 20-Jan2005 http://www.onlamp.com/pub/a/onlamp/2005/01/20/rails.html

라인 상에서 볼 수 있으며 http://www.ruby-doc.org/docs/ruby-doc-bundle/ProgrammingRuby/index.html에서 다운로드 할 수 있다.

감사의 말: irc 채널[2]과 메일링 리스트[3]로 많은 도움을 주신 분들께 감사드린다.

버전: 현재 2.3 버전이며 0.12.1 버전의 레일즈를 사용하여 작성하였다. http://rails.homelinux.org에서 최신 버전을 구할 수 있으며 ToDo 소스 코드를 다운로드할 수 있다. 이 문서는 OpenOffice의 'Writer'를 이용하여 작성 및 pdf 파일로 출력하였다.

2 irc://irc.freenode.org/rubyonrails

3 http://lists.rubyonrails.org/mailman/listinfo/rails

레일즈 1일차

‘To Do List’ 애플리케이션

이 문서는 간단한 ‘To Do List’(To Do List는 여러분의 PDA에 있을 법한 종류의 애플리케이션인데, 항목들의 목록을 갖고 있으며, 그것들을 분류하고, 선택적으로 필기를 할 수 있다) 애플리케이션을 구축하는 과정을 따라 진행될 것이다(애플리케이션이 어떻게 생겼는지 슬쩍 한번 보고 싶다면 31페이지에 있는 화면 5를 보라).

레일즈 스크립트 실행하기

다음 예제는 MS-Windows PC용 예제이다. 웹 관련 디렉터리는 `c:\www\webroot` 인데 불필요한 타이핑을 줄이기 위해 `w:` 드라이브로 레이블을 지정하였다.

```
C:\> subst w: c:\www\webroot
C:\> w:
W:\> rails ToDo
W:\> cd ToDo
W:\ToDo>
```

`rails ToDo`를 실행하면 `ToDo\`라는 새 디렉터리가 만들어질 것이며 일련의 파일과 하위 디렉터리들이 그 안에 만들어질 것이다. 새로 만들어진 것들 중 가장 중요한 것들은 다음과 같다.

```
app
     애플리케이션의 가장 핵심적인 내용을 담고 있으며 모델, 뷰, 컨트롤러 및 ‘헬퍼’ 역할을
     하는 하위 디렉터리들로 나뉜다.
config
     애플리케이션에서 사용되는 데이터베이스의 세부내용을 제공하는 databse.yml
     파일을 담고 있다.
log
     애플리케이션에 특징적인(application specific) 로그를 담고 있다.
     주의사항: development.log 파일은 레일즈가 수행하는 모든 액션에 대한
     기록을 유지하는데 오류 추적에 있어 매우 유용하나 주기적으로 비울 필요가 있다.
public
     아파치에서 사용가능한 디렉터리이며 이미지와 자바스크립트, 스타일시트가
     들어있는 하위 디렉터리들을 포함한다.
```

웹 서버에 애플리케이션 추가하기

필자는 개발용 PC에 모든 프로그램들(아파치2, MySQL, 기타 등등)을 운영하고 있기 때문에 다음의 두 단계에서는 브라우저에서 애플리케이션명으로 사용할 이름만 지정하도록 하겠다.

hosts 파일에 애플리케이션 정의하기

```
C:\winnt\system32\drivers\etc\hosts (이 줄 생략)
127.0.0.1 todo
```

아파치 설정 파일에 애플리케이션 정의하기

```
Apache2\conf\httpd.conf
<VirtualHost *>
  ServerName todo
  DocumentRoot /www/webroot/ToDo/public
    <Directory /www/webroot/ToDo/public/>
      Options ExecCGI FollowSymLinks
      AllowOverride all
      Allow from all
      Order allow,deny
    </Directory>
</VirtualHost>
```

fastcgi로 전환하기

여러분의 성격이 그리 느긋하지 않다면(아니면 PC 성능이 좋지 않거나) fastcgi를 활성화할 필요가 있다.

```
public\.htaccess
# 좀 더 나은 성능을 내려면 디스패처를 fastcgi로 교체한다.
RewriteRule ^(.*) $ dispatch.fcgi [QSA,L]
```

레일즈가 작동하고 있는지 확인하기

이제 브라우저에서 `http://todo/`를 입력했을 때 사이트가 나타나야 한다(여러분은 `Congratulations, you've put Ruby on Rails!`라는 문구를 페이지에서 볼 수 있어야 한다).

레일즈 버전

여러분이 이 문서를 읽기전까지 레일즈는 여러 버전을 거쳤을 것이다. 여러분이 이 문서를 다 읽어보려고 한다면 PC에 설치되어 있는 레일즈의 버전을 확인해볼 필요가 있다:

```
W:\ToDo>gem list --local
```

출력되는 버전들이 아래 나열된 것들과 다를 경우 이 문서에서 사용하고 있는 버전으로 다운로드하기를 적극 권장한다. 가령,

```
W:\ToDo>gem install rails --version 0.12.1
```

이라 입력하면 아무것도 손상되지 않는데, 왜냐하면 루비 젬 라이브러리는 여러 버전을 처리할 수 있도록 설계되어 있기 때문이다. 아니면 여러분은 강제로 레일즈가 'To Do List' 애플리케이션에 'Four Days'에서 사용하고 있는 버전을 지정하여 그 버전을 사용하게 할 수도 있다:

config\environment.rb (이 줄 생략)

```
# Require Rails libraries.
require 'rubygems'
require_gem 'activesupport', '= 1.0.4'
require_gem 'activerecord', '= 1.10.1'
require_gem 'actionpack', '= 1.8.1'
require_gem 'actionmailer', '= 0.9.1'
require_gem 'actionwebservice', '= 0.7.1'
require_gem 'rails', '= 0.12.1'
```

동일한 버전을 사용하는 이유는 지극히 단순하다. 'Four Days'에서는 레일즈가 자동으로 만들어 주는 코드를 많이 사용하는데, 레일즈가 코드는 만들어 주지만 -불행히도- 문서는 만들어주지 않기 때문이다(새 버전을 만들어내기 전까지는). 그러므로 여러분의 삶이 편안해지려면 'Four Days'에서 사용한 버전과 동일한 버전으로 맞추도록 한다. 일단 여러분이 'Four Days'를 모두 해보게 되면 무슨 일이 있더라도 가장 최신 버전의 레일즈 버전으로 실행해보고 레일즈 개발자들이 어떠한 개선사항을 추가해 놓았는지 알아두자.

데이터베이스 설정하기

이미 나는 MySQL에 'todos'라는 이름의 새 데이터베이스를 설정해 두었으며 데이터베이스 접속 설정은 config\database.yml 파일에 지정되어 있다.

config\database.yml (이 줄 생략)

```
development:
  adapter: mysql
  database: todos
  host: localhost
  username: foo
  password: bar
```

Categories 테이블 만들기

categories 테이블은 곧 만들어볼 예제에서 사용된다. 그것들은 단순히 카테고리의 목록인데, To Do 목록에서 각 항목들을 그룹짓는 데 사용될 것이다.

MySQL 정의

Categories 테이블

```
CREATE TABLE `categories` (
  `id` smallint(5) unsigned NOT NULL auto_increment,
  `category` varchar(20) NOT NULL default '',
  `created_on` timestamp(14) NOT NULL,
  `updated_on` timestamp(14) NOT NULL,
  PRIMARY KEY (`id`),
  UNIQUE KEY `category_key` (`category`)
) TYPE=MyISAM COMMENT='List of categories';
```

테이블과 필드명에 대해 힌트를 조금 주자면:

- 필드명에 포함된 밑줄(_)은 '사람이 읽기 쉬운' 이름으로 레일즈가 공백으로 변경할 것이다.
- 대소문자가 섞여있는 필드명은 주의하도록 한다. 레일즈 코드중에는 대소문자를 구별하는 것이 있기 때문이다.
- 모든 테이블은 'id'라는 이름의 주키를 가져야 한다. MySQL에서는 이러한 주키들을 numeric auto increment로 지정하는 것이 가장 쉽다.
- 다른 테이블을 연결할 경우에는 동일하게 '_id' 명명규칙을 따라야 한다.
- 레일즈가 자동으로 created at/created on이나 update at/update on라는 이름의 필드를 관리할 것이므로 그것들을 *Documentation:ActiveRecord::Timestamp*에 추가해 놓는 것이 좋다.

Documentation: ActiveRecord::Timestamp

- 유용한 팁: 여러분이 여러 사용자가 사용하는 시스템을 구축하고자 한다면(지금 하고 있는 예제와는 관련이 없지만) 여러분이 lock version(integer default 0)라는 이름의 필드를 추가할 경우 레일즈 또한 낙관적 잠금(optimistic locking)을 수행할 것이다. 여러분이 기억해 두어야 할 것은 update 폼에 히든 필드로 lock_version을 포함하는 것 뿐이다.

Documentation: ActiveRecord::Locking

데이터 모델

아래와 같이 입력하면:

```
W:\ToDo>ruby script/generate model category
    exists     app/models/
    exists     test/unit/
    exists     test/fixtures/
    create     app/models/category.rb
    create     test/unit/category_test.rb
    create     test/fixtures/categories.yml
W:\ToDo>
```

category.rb라는 이름의 빈 파일 하나와 category_controller_test.rb와 categories.yml이라는 두 개의 테스트 파일이 만들어질 것이다. 이어서 곧 데이터 모델에 몇가지를 입력할 것이므로 지금 당장은 비어있는 채로 두도록 하자.

스캐폴드

컨트롤러는 레일즈 애플리케이션에서 핵심이라 할 수 있다.

generate controller 스크립트를 실행

```
W:\ToDo>ruby script/generate controller category
    exists  app/controllers/
    exists  app/helpers/
    create  app/views/category
```

```
    exists  test/functional/
    create  app/controllers/category_controller.rb
    create  test/functional/category_controller_test.rb
    create  app/helpers/category_helper.rb
W:\ToDo>
```

위 스크립트를 실행하면 아래와 같은 파일과 빈 디렉터리가 각각 2개씩 만들어질 것이다:

```
app\controllers\category_controller.rb
app\helpers\category_helper.rb
app\views\categories
app\views\layouts
```

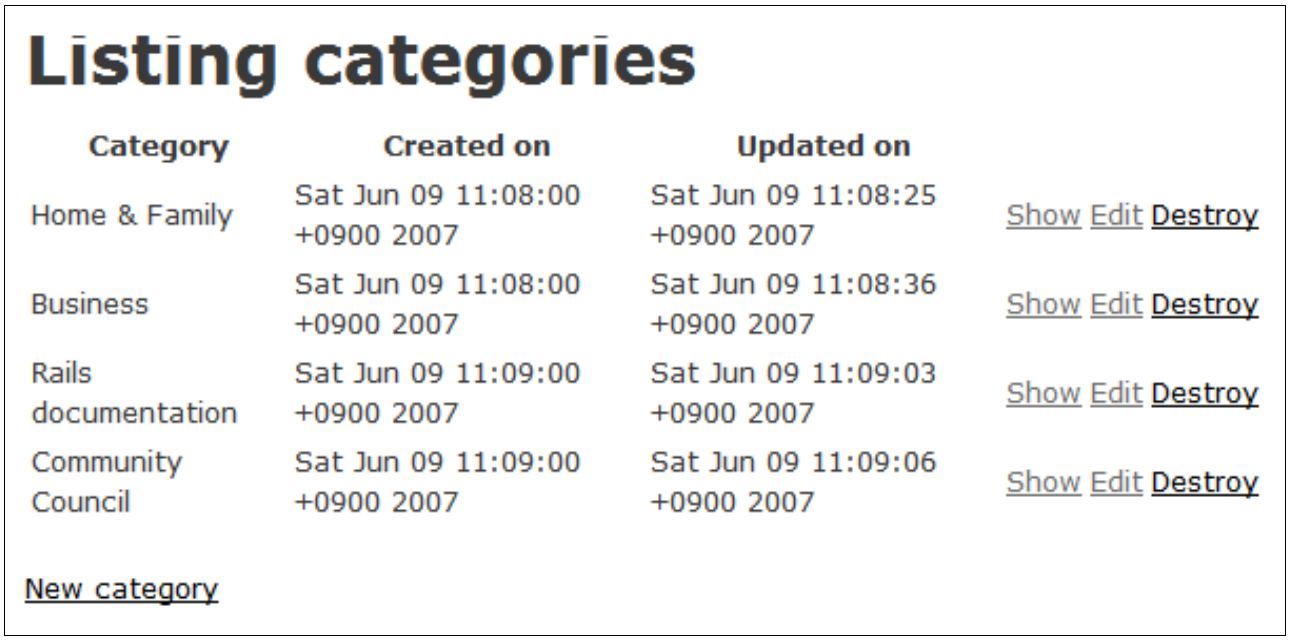
여러분이 *Rolling with Ruby on Rails* 와 같은 레일즈 초심자를 위한 튜토리얼에서 모델과 스캐폴드가 동작하는 것을 아직 보질 못했다면 여기서 그것들을 한번 해보고 나면 하나의 웹 애플리케이션 전체가 단지 한줄의 코드만으로도 작성될 수 있다는 것을 보고 놀라게 될 것이다.

app\controllers\category_controller.rb

```
class CategoryController < ApplicationController
    scaffold :category
end
```

Documentation: ActionController::Scaffolding

브라우저를 열어 `http://todo/category`로 가보면 그것들이 얼마나 영리한지 아마 놀라게 될 것이다. :-)



화면 1: 스캐폴드 'List' 화면

반면 스캐폴드가 얼마만큼 똑똑하지는 않은지 확인해 보려면 똑같이 새 카테고리를 2번 추가해 본다. 레일즈는 'ActiveRecord::StatementInvalid in Category#create'라는 지저분한 오류 메시지를 출력하면서 실행이 중단될 것이다. 여러분은 모델에 유효성 검증을 추가하여 이러한 문제를 수정할 수 있다.

모델 강화하기

모델(Model)은 모든 데이터 관련 규칙들이 저장되는 것인데 여기에는 데이터 유효성 검증과 함께 관계 무결성도 포함된다. 즉, 여러분이 한번 규칙을 정의하기만 하면 데이터가 접근하는 곳마다 그러한 규칙들이 적용될 것임을 의미한다.

데이터 유효성 검증 규칙 만들기

레일즈는 여러분이 (거의) 아무런 노력을 기울이지 않아도 갖가지 오류 처리 방법을 제공해 준다. 이를 한번 보여주기 위해 유효성 검증 규칙을 빈 카테고리 모델에 추가한다:

app\models\category.rb

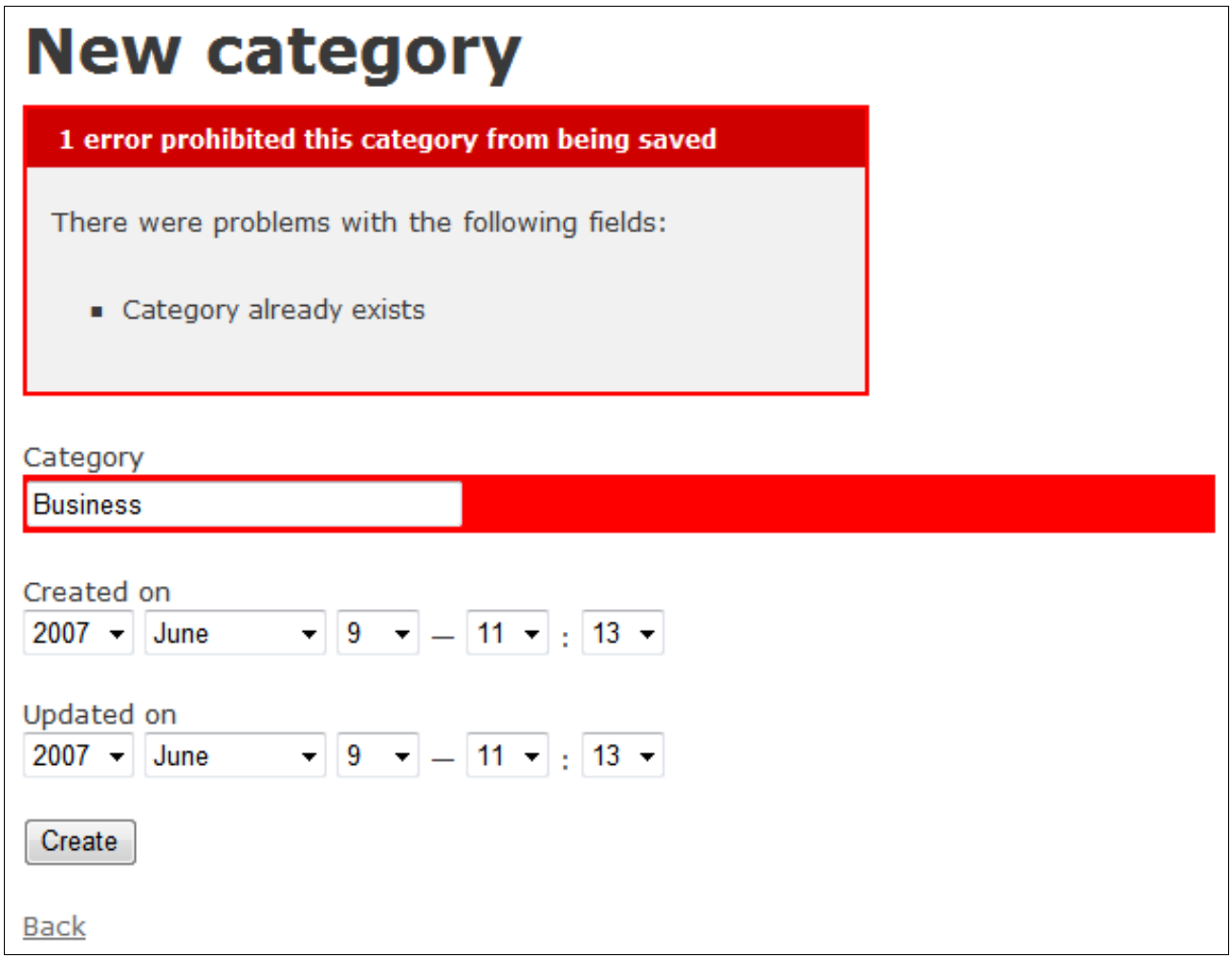
```
class Category < ActiveRecord::Base
    validates_length_of :category, :within => 1..20
    validates_uniqueness_of :category, :message => "already exists"
end
```

위에서 입력한 내용들로 인해 자동적으로 아래 사항들이 확인될 것이다:

- `validates length of:` 필드가 비어있거나 너무 길지는 않는지를 확인한다.
- `validates uniqueness of:` 중복된 값이 검출되지는 않았는지 확인한다. 사실 나는 레일즈의 기본 오류 메시지 중의 하나인 'xxx has already been taken'를 좋아하지 않기 때문에 직접 만든 것을 사용한다. 이것은 레일즈의 일반적인 기능중 하나인데, 즉 기본값을 먼저 이용해 보고 마음에 들지 않으면 재정의하는 것이다.

Documentation: ActiveRecord::Validations::ClassMethods

이것을 한번 시험해 보려면 지금 중복된 레코드를 다시 입력해 본다. 이번에는 레일즈가 그대로 죽지는 않고 오류를 처리할 것이다. 이러한 방식은 서로 대비되는데 사실 사용자 인터페이스에 측면에 있어 그리 큰 문제는 아니다. 공짠데 뭘 바라나?



화면 2: 데이터 오류 잡아내기

레일즈 2일차

여기서부터 더 진행해 나가려면 레일즈의 감춰진 부분에서 어떤 일들이 발생하는지에 알아야할 필요가 있다. 2일차 동안에는 레일즈가 만들어 주는 스캐폴딩 코드들이 도대체 어떠한 의미를 지니는지 풀어헤쳐 나가면서 체계적으로 알아볼 것이다. 스캐폴딩 액션(action)과 함께 레일즈는 필요로 하는 모든 코드들을 동적으로 만들어 낸다. 스캐폴드를 *스크립트로* 실행하여 모든 코드가 디스크에 기록되도록 함으로써 기록된 코드를 검사하여 요구사항에 맞도록 조정하는 작업을 시작할 수 있다.

generate scaffold 스크립트 실행하기

```
W:\ToDo>ruby script/generate scaffold category
      dependency  model
      exists      app/models/
      exists      test/unit/
      exists      test/fixtures/
        skip      app/models/category.rb
        skip      test/unit/category_test.rb
        skip      test/fixtures/categories.yml
      exists      app/controllers/
      exists      app/helpers/
      create      app/views/categories
      exists      test/functional/
      create      app/controllers/categories_controller.rb
      create      test/functional/categories_controller_test.rb
      create      app/helpers/categories_helper.rb
      create      app/views/layouts/categories.rhtml
      create      public/stylesheets/scaffold.css
      create      app/views/categories/list.rhtml
      create      app/views/categories/show.rhtml
      create      app/views/categories/new.rhtml
      create      app/views/categories/edit.rhtml
      create      app/views/categories/_form.rhtml

W:\ToDo>
```

이 스크립트는 컨트롤러, 뷰, 레이아웃, 그리고 스타일 시트까지 하나의 완전한 애플리케이션을 작성하는데 필요한 파일들을 만들어 준다.

여기서 다소 이상야릇한 명명 규칙을 볼 수 있는데, 단수형에서 복수형으로 바뀌었으며 새로운 코드를 사용하려면 브라우저가 `http://todo/categories`를 가리키도록 할 필요가 있다. 사실 혼동을 방지하려면 여러분이 실수로 실행할 경우에 대비하여 `app\controllers\category_controller.rb` 등을 삭제하는 것이 가장 좋다.

생성된 스캐폴드 코드

컨트롤러

이번에는 컨트롤러 내부의 코드들을 살펴보기로 하자. 컨트롤러는 애플리케이션의 프로그래밍 로직이 위치하는 곳이다. 컨트롤러는 뷰를 이용하여 사용자와 상호작용하며 모델을 통해 데이터베이스와 상호작용한다. 여러분은 컨트롤러 코드를 분석하여 애플리케이션이 어떻게 서로서로 연동하는지 알아야 할 필요가 있다.

generate scaffold 스크립트로 만들어진 컨트롤러 코드가 아래에 나타나 있다:

```
\app\controllers\categories_controller.rb
class CategoriesController < ApplicationController
  def index
    list
    render_action 'list'
  end

  def list
    @category_pages, @categories = paginate :category, :per_page => 10
  end

  def show
    @category = Category.find(@params[:id])
  end

  def new
    @category = Category.new
  end

  def create
    @category = Category.new(@params[:category])
    if @category.save
      flash['notice'] = 'Category was successfully created.'
      redirect_to :action => 'list'
    else
      render_action 'new'
    end
  end

  def edit
    @category = Category.find(@params[:id])
  end

  def update
```

```
    @category = Category.find(@params[:id])
    if @category.update_attributes(@params[:category])
      flash['notice'] = 'Category was successfully updated.'
      redirect_to :action => 'show', :id => @category
    else
      render_action 'edit'
    end
  end

  def destroy
    Category.find(@params[:id]).destroy
    redirect_to :action => 'list'
  end
end
```

레일즈 애플리케이션에서 사용자가 액션(예를 들어 'Show')을 선택하면 컨트롤러는 적절한 위치('def show')에 있는 코드를 실행한 다음 기본적으로 동일한 이름의 템플릿('show.rhtml')을 렌더링할 것이다. 기본 동작방식은 재정의될 수 있다:

- render_template는 여러분이 다른 템플릿을 렌더링할 수 있도록 해주는데, 가령 index 액션은 'list' (즉, 'def list')에 대한 코드를 실행한 다음 존재하지 않는 index.rhtml이 아닌 list.rhtml을 렌더링할 것이다.

- redirect_to는 페이지를 이동하며 겉으로는 '302 moved'라는 HTTP 응답을 이용하면서 컨트롤러로 루프백을 수행한다. 가령 destroy 액션이 템플릿을 렌더링할 필요가 없을 때 이러한 방식으로 동작하는데, 원래의 목적(카테고리 제거)을 수행하고 난 후에 단순히 사용자를 list 액션으로 데려가게 하는 역할을 한다.

Documentation: ActionController::Base

컨트롤러는 find, find all, new, save, update atrributes, destory와 같은 액티브레코드(ActiveRecord) 메소드를 이용하여 데이터를 데이터베이스 테이블로 옮기거나 가져온다. 한 가지 알아둘 것은 여러분은 아무런 SQL문도 작성할 필요는 없으나 레일즈가 어떠한 SQL을 사용하는지를 보고 싶을 경우에는 SQL문이 development.log 파일에 모두 기록되기 때문에 그 파일을 참조하면 된다.

Documentation: ActiveRecord::Base

또 한가지 알아둘 것은 가령 테이블내의 레코드를 갱신하는 것과 같은 사용자 관점에서의 논리적인 활동 하나가 어떻게 컨트롤러에 두 번의 활동을 전달할 수 있느냐이다. 즉, 사용자가 'Edit'를 선택할 때 컨트롤러는 모델로부터 사용자가 편집하기를 원하는 레코드를 추출한 다음, 그것을 edit 뷰에 렌더링한다. 사용자가 편집을 마치면 edit 뷰는 update 액션을 호출하여 모델을 갱신한 다음 show 액션을 호출하게 된다.

뷰

뷰는 사용자 인터페이스가 정의되어 있는 곳이다. 레일즈는 3개의 컴포넌트들로부터 최종 HTML 페이지를 렌더링하여 사용자에게 보여준다.

Layout (레이아웃)	Template (템플릿)	Partial (파셜)
위치: app\views\layouts\ 기본값: application.rhtml 혹은 <controller>.rhtml	위치:app\views\<controller>\ 기본값: <action>.rhtml	위치: app\views\<controller>\ 기본값:_<partial>.rhtml

- 레이아웃은 모든 액션에서 사용하는 공통 코드를 제공하며, 일반적으로 브라우저로 전송되는 HTML 문서의 시작과 끝을 구성한다.
- 템플릿은 액션에 특징적인 코드를 제공해 주며 'List'나 'Edit' 코드 등이 여기에 포함된다.
- 파셜(Partial)은 '서브루틴(subroutines)'이라 불리는 공통 코드를 제공하는데, 여러 개의 액션에서 사용될 수 있다. 가령 폼에서 테이블을 구성하는데 사용되는 코드를 예로 들 수 있다.

레이아웃

레일즈의 명명 규칙은 만약 현재 컨트롤러의 이름과 동일한 이름을 가진 템플릿이 app\views\layouts\ 디렉터리 안에 들어 있을 경우 명시적으로 다른 이름이 지정되어 있지 않으면 컨트롤러의 레이아웃으로 지정한다.

현재 컨트롤러와 동일한 이름의 레이아웃이 존재하지 않고 또 명시적으로 할당되어 있는 레이아웃이 없을 경우에는 application.rhtml이나 application.rxml이라는 이름을 가진 레이아웃이 기본 컨트롤러로 지정될 것이다.

스캐폴드 스크립트로 생성된 레이아웃은 다음과 같다:

app\views\layouts\categories.rhtml

```
<html>
<head>
  <title>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>

<%= @content_for_layout %>

</body>
</html>
```

일반적인 HTML에 약간의 루비 코드가 <% %> 태그안에 추가되어 있는 것을 볼 수 있다. 레이아웃은 실행되고 있는 액션과는 상관없이 렌더링 프로세스에 의해 요청될 것이다. 레이아웃은 모든 페이지에서 보여질 <html><head>...</head><body>...</body></html>과 같은 표준 HTML 태그를 포함한다.

굵게 표시된 루비 코드들은 레일즈의 렌더링 과정동안 다음의 규칙에 따라 HTML로 변환된다:

- action_name은 ActionController 메소드로서 컨트롤러가 처리하고 있는 액션의 이름(예, 'List')을 반환하는데, 실행되고 있는 액션에 근거하여 적절한 제목을 페이지에 출력한다.

Documentation: ActionController::Base

- stylesheet_link_tag는 레일즈 헬퍼이며 다소 게으른 방식으로 코드를 생성하며 레일즈에는 이러한 헬퍼들을 상당수 포함하고 있다. stylesheet_link_tag는 단순히 다음의 HTML만을 만들어 낸다:

Documentation: ActionView::Helpers::AssetTagHelper

- content_for_layout은 다음에 설명할 내용의 핵심이다. content_for_layout은 하나의 표준 레이아웃에서 현재 수행되고 있는 액션(예를 들면 'edit', 'new', 'list')에 근거하여 렌더링 시 동적으로 컨텐츠를 삽입할 수 있도록 해준다. 이러한 동적인 컨텐츠들은 동일한 이름을 가진 템플릿으로부터 가져온다. 아래 문서 참조.

Documentation: ActionController::Layout::ClassMethods.

템플릿

레일즈의 명명규칙 하나 : 템플릿은 app\views\categories\'action'.rhtml에서 관리한다.

스캐폴드 스크립트로 만들어진 새로운 .rhtml 파일의 내용은 아래와 같다:

app\views\categories\new.rhtml

```
<h1>New category</h1>

<%= start_form_tag :action => 'create' %>
  <%= render_partial "form" %>
  <%= submit_tag "Create" %>
<%= end_form_tag %>

<%= link_to 'Back', :action => 'list' %>
```

- start_form_tag는 HTML 폼을 시작하는 레일즈 헬퍼이며 여기에서는 <form action="/categories/create" method="post">을 생성하는 역할을 한다.

- submit_tag는 자체적으로 <input name="submit" type="submit" value="Save changes" />를 생성하며, "Create" 파라미터가 기본값인 "Save changes"을 "Create"으로 재정의한다.

- end_form_tag는 단순히 </form>을 출력하는 역할만을 하는데, 지금까지 작성된 레일즈 헬퍼 중에서 가장 유용한 헬퍼는 아니지만 코드 블록을 마무리 하는데는 충분하다.

Documentation: ActionView::Helpers::FormTagHelper

- render_parthial은 파셜 _form.rhtml을 호출하는데 다음 섹션에서 알아보도록 하겠다.

Documentation: ActionView::Partials

- `link_to`는 단순히 링크를 생성하는데 사용되며 HTML의 가장 기본적인 부분을 차지한다. `<a href="/categories/list">Back</a>`

Documentation: ActionView::Helpers::UrlHelper

파셜

레일즈의 명명규칙 둘 : 'foo'라는 파셜은 `app\views\'action'\_foo.rhtml`로 들어간다(시작할 때 밑줄로 시작하는 것임을 주의).

스캐폴드는 동일한 코드를 사용하여 'edit'와 'new' 액션을 모두 처리하는데, 따라서 레일즈가 파셜에 집어넣은 코드는 `render_partial` 메소드를 이용하여 호출된다.

app\views\categories_form.rhtml

```
<%= error_messages_for 'category' %>

<!--[form:category]-->
<p><label for="category_category">Category</label><br/>
<%= text_field 'category', 'category' %></p>

<p><label for="category_created_on">Created on</label><br/>
</p>

<p><label for="category_updated_on">Updated on</label><br/>
</p>
<!--[eoform:category]-->
```

- `error_messages_for`는 앞서 폼 전송시 발생하는 오류 메시지에 대해서 마크업 처리가 이루어진 문자열을 리턴한다. 하나 혹은 그 이상의 오류가 감지될 경우에 만들어지는 HTML 문서는 다음과 같다:

```
<div class="errorExplanation" id="errorExplanation">
  <h2>n errors prohibited this xxx from being saved</h2>
  <p>There were problems with the following fields:</p>
  <ul>
    <li>field_1 error_message_1</li>
    <li>... ...</li>
    <li>field_n error_message_n</li>
  </ul>
</div>
```

이미 우리는 이것을 1일차 10 페이지에 나오는 '화면 2: 데이터 오류 잡아내기'에서 보았는데, 눈여겨 볼 부분은 css 태그가 스캐폴드 스크립트에 의해 만들어진 스타일시트에 들어있는 문장과 서로 일치한다는 것이다.

Documentation: ActionView::Helpers::ActiveRecordHelper

- text_field는 레일즈 헬퍼이며 다음의 HTML을 만들어준다: <input id="category_category" name="category[category]" size="30" type="text" value="" /> 첫번째 파라미터는 테이블명이며 두번째 파라미터는 필드명이다.

Documentation: ActionView::Helpers::FormHelper

레일즈에는 사소한 버그가 하나 있는데, created_on과 updated_on이라는 이름으로 예약된 필드에 대해서는 굳이 입력 필드를 만들지 않아도 그것들에 대한 레이블은 만들어지는 것으로 알려져 있다.

"New" 액션에 대한 렌더링된 뷰

이제 "New" 액션에 대한 응답으로 브라우저에 반환되는 코드를 살펴보고 그것들 모두가 어디에서 만들어졌는지 살펴볼 차례이다. 진한 텍스트는 레이아웃을, 일반 텍스트는 템플릿을, 이탤릭체는 파셜을 의미한다:

app\views\categories\new.rhtml

```
<html>
<head>
<title>Categories: new</title>
<link href="/stylesheets/scaffold.css" media="screen" rel="Stylesheet"
type="text/css" />
</head>
<body>

<h1>New category</h1>

<form action="/categories/create" method="post">

<!--[form:category]-->
<p><label for="category_category">Category</label><br/>
<input id="category_category" name="category[category]" size="30"
type="text" value="" /></p>

<p><label for="category_created_on">Created on</label><br/>
</p>

<p><label for="category_updated_on">Updated on</label><br/>
</p>
<!--[eoform:category]-->

  <input name="submit" type="submit" value="Create" />
```

```
</form>

<a href="/categories/list">Back</a>

</body>
</html>
```

'List' 액션에 대한 뷰 분석

'Edit'와 'Show' 뷰는 'New' 뷰와 유사하다. 'List'는 몇 가지 새로운 기법을 포함한다. 컨트롤러가 'List' 템플릿에 대한 렌더링을 시작하기 전에 다음의 코드를 어떻게 실행하는지 기억하라:

```
    @category_pages, @categories = paginate :category, :per_page => 10
```

paginate는 Categories 테이블로부터 정렬된 레코드를 가진 `@categories` 인스턴스 변수을 가져오는데, 한번에 `:per page` 만큼의 레코드를 가져오며, 다음 페이지와 이전 페이지 등 페이지 탐색에 관한 모든 로직을 포함한다. `@category pages`는 `Paginator` 인스턴스이며 템플릿에서 이러한 것들이 어떻게 사용되는지에 대해서는 이어지는 섹션의 끝부분에서 설명하겠다.

Documentation: ActionController::Pagination

템플릿은 다음과 같다:

```
app\views\categories\list.rhtml
<h1>Listing categories</h1>

<table>
  <tr>
<% for column in Category.content_columns %>
    <th><%= column.human_name %></th>
<% end %>
  </tr>
<% for category in @categories %>
  <tr>
<% for column in Category.content_columns %>

    <td><%=h category.send(column.name) %></td>
<% end %>
    <td><%= link_to 'Show', :action => 'show', :id => category %></td>
    <td><%= link_to 'Edit', :action => 'edit', :id => category %></td>
    <td><%= link_to 'Destroy', {:action => 'destroy', :id => category},
:confirm => "Are you sure?" %></td>
  </tr>
<% end %>
</table>

<%= link_to "Previous page", { :page => @category_pages.current.previous }
if @category_pages.current.previous %>
```

```
<%= link_to "Next page", { :page => @category_pages.current.next } if
@category_pages.current.next %>

<br />

<%= link_to 'New category', :action => 'new' %>
```

- content_columns는 '특수한' 열(주키인 id, '_id'나 '_count'로 끝나는 모든 열, 단일 테이블 상속에 사용되는 모든 열)을 제외한 열 객체의 배열을 반환한다

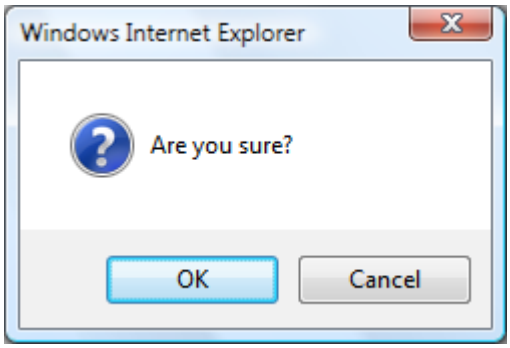
Documentation: ActionController::Base

- human_name은 human_attribute_name의 동의어이며 속성키 이름을 'first_name'이 아닌 'First name'과 같이 일상적인 형식으로 변환해 준다.

Documentation: ActiveRecord::Base

- h는 자동으로 HTML 코드를 이스케이프(escape)해준다. 사용자가 화면에 나타나는 데이터를 입력하도록 해주는 것의 한 가지 문제점은 사용자가 실수로(혹은 악의적으로) 화면에 표시되었을 때 시스템이 망가지는 코드를 입력할 수도 있다는 것이다[4]. 이것으로부터 시스템을 보호하려면 사용자가 입력하는 데이터에 대하여 'HTML 이스케이프'를 수행 하도록 하는 것이 좋은 습관이다. 이는 가령 </table>을 아무런 문제가 발생하지 않는 </table>로 렌더링함을 의미한다. 레일즈에서는 이러한 작업을 정말로 간단하게-보다시피 단순히 'h'만 추가하기만 하면-만들어 준다.

- confirm은 link_to 헬퍼에 대한 유용한 선택 파라미터(optional parameter)인데, 사용자가 링크에 대한 액션을 취하기 전에 Destroy할 것인지에 대한 확인을 하도록 자바스크립트 팝업 창을 만들어준다.



화면 3: 자바스크립트 팝업

Documentation: ActionView::Helpers::UrlHelper

페이징 처리 로직은 다소 복잡하게 얽혀있다. 루비는 if를 한정자로 사용할 수 있는데, *expression if boolean-expression*은 *boolean-expression*이 참일 때만 *expression*을 평가한다. @category pages.current는 paginator의 현재 페이지를 나타내는 Page 객체를 반환하며

ActionController::Pagination::Paginator

4 예를 들면, 사용자가 Category로 "</table>"을 입력했을 때 어떠한 일이 일어날지 생각해 보자.

`@categroy pages.current.previous`는 현재 페이지의 바로 전 페이지를 나타내는 Page 객체를 리턴하거나, 현재 페이지가 첫 페이지일 경우 nil을 반환한다.

ActionController::Pagination::Paginator::Page

그러므로 만약 이동하려는 이전 페이지가 존재할 경우 링크가 보여질 것이며 이전 페이지가 존재하지 않을 경우에는 링크는 보여지지 않을 것이다.

n 페이지에 대한 렌더링된 코드는 다음과 같다:

```
<a href="/categories/list?page=[n-1]">Previous page</a>
<a href="/categories/list?page=[n+1]">Next page</a>
```

생성된 스캐폴드 코드 조정하기

스캐폴드 스크립트에 의해 생성된 코드는 그대로 사용할 수 있을(out of the box) 정도로 완전하며 일단 여러분이 데이터 모델에 충분한 유효성 검증 로직을 추가했다면 상당히 강건(robust)해진다. 그렇지만 레일즈 애플리케이션을 개발하기 위한 모든 것들이 그곳에 포함되어 있다면 프로그래머들은 아마 일자리가 없어질 것이고, 이는 확실히 좋은 현상은 아닐 것이다. :-) 그러므로 스캐폴드 코드를 조금 수정해 보도록 하자:

컨트롤러

'List' 뷰에서는 레코드가 알파벳 순서로 보여질 것으로 기대했었는데, 이렇게 하려면 컨트롤러를 다음과 같이 조금 변경할 필요가 있다:

app\controllers\categories_controller.rb (이 줄 생략)

```
def list
  @category_pages, @categories = paginate :category,
    :per_page => 10, :order_by => 'category'
end
```

Documentation: ActionController::Pagination

이 애플리케이션에서는 `show` 화면은 불필요한데, 왜냐하면 모든 필드가 화면에 적당히 한 줄로 보이기 때문이다. 그러므로 `def show`는 없앨 수 있으며 바로 'Edit' 다음의 `list` 화면으로 되돌아가도록 하자:

app\controllers\categories_controller.rb (이 줄 생략)

```
def update
  @category = Category.find(@params[:id])
  if @category.update_attributes(@params[:category])
    flash['notice'] = 'Category was successfully updated.'
    redirect_to :action => 'list'
  else
    render_action 'edit'
  end
```

```
end
```

flash 메시지는 다음에 보여질 화면을 선택한 후에 보여지는데, 이 경우에는 list 화면에 보여질 것이다. 기본적으로 스캐폴드 스크립트는 플래시 메시지를 보여주지 않으며 아래에서 볼 수 있는 것과 같이 잠깐동안 수정해볼 것이다.

뷰

플래시 메시지 보여주기

레일즈는 '플래시' 메시지를 사용자에게 전달하는 방법을 제공해 주는데, 플래시 메시지란 예를 들면 다음 페이지에서 한번 보여지고 사라지는 '업데이트가 성공적으로 완료되었습니다'와 같은 메시지를 말한다. 이는 레이아웃(이것을 레이아웃에 추가한다는 것은 어떠한 화면에서도 나타날 수 있음을 의미한다)을 조금 변경하여 손쉽게 선택할 수 있다:

app\views\layouts\categories.rhtml<html>

```
<head>
  <title>Categories: <%= controller.action_name %></title>
  <%= stylesheet_link_tag 'scaffold' %>
</head>
<body>
<h1><%=@heading %></h1>
<% if @flash["notice"] %>
<span class="notice">
  <%=h @flash["notice"] %>
</span>
<% end %>
<%= @content_for_layout %>
</body>
</html>
```

Documentation: ActionController::Flash

단순히 스타일시트에 추가하는 것만으로도 플래시 메시지가 좀 더 눈에 잘 띌 것이다.

public\stylesheets\scaffold.css (이 줄 생략)

```
.notice {
  color: red;
}
```

템플릿과 레이아웃간의 변수 공유하기

앞에서 템플릿에 들어있는 <h1>…</h1> 머리말 텍스트를 레이아웃으로 이동시켜 플래시 메시지 상단에 나타나도록 했던 것을 명심하도록 한다. 각각의 템플릿은 서로 다른 머리말을 가질 것이므로 템플릿의 @heading 변수의 값을 지정할 필요가 있다. 레일즈는 이런 작업을 상당히 편리하게 할 수 있는데, 이는 템플릿 변수가 렌더링시 레이아웃에서 사용할 수 있기 때문이다.

이러한 변경사항과 출력형식에 대한 변경작업을 통해 아래와 같이 템플릿을 완성하였다:

app\views\categories\list.rhtml

```
<% @heading = "Categories" %>
<table>
  <tr>
    <th>Category</th>
    <th>Created</th>
    <th>Updated</th>
  </tr>
<% for category in @categories %>
  <tr>
    <td><%=h category["category"] %></td>
    <td><%= category["created_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= category["updated_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= link_to 'Edit', :action => 'edit', :id => category %></td>
    <td><%= link_to 'Delete', {:action => 'destroy', :id => category},
            :confirm => "Are you sure you want to delete this category?"
%></td>
  </tr>
<% end %>
</table>
<br />
<%= link_to 'New category', :action => 'new' %>
<% if @category_pages.page_count>1 %>
<hr />
Page: <%=pagination_links @category_pages %>
<hr />
<% end %>
```

- 나는 기본 날짜 형식을 좋아하지 않기 때문에 루비 메소드인 `strftime()`을 이용하여 날짜와 시간 필드를 원하는 형식으로 맞추었다.

Ruby Documentation: class Time

- `pagination_links`는 주어진 paginator에 대한 기본 HTML 링크를 만들어 준다.

ActionView::Helpers::PaginationHelper

Edit/New 화면을 깔끔하게 정리하기

'New'와 'Edit'에서 사용된 파셜에 약간 변경할 텐데 이번에는 테이블을 사용하여 외관을 좀 더 보기좋게 바꾸고 원치않은 `created_on/updated_on` 레이블을 제거하고, 사용자가 `Category` 필드에 너무 많은 내용을 입력하는 것을 방지할 것이다:

app\views\categories_form.rhtml

```
<%= error_messages_for 'category' %>
<table>
```

```
<tr>
  <td><b><label for="category_category">Category:</label></b></td>
  <td><%= text_field "category", "category", "size"=>20, "maxlength"=>20
%></td>
</tr>
</table>
```

그리고 두 템플릿도 조금 변경해 볼 것이다(특히 `@heading`을 사용하는 부분을 눈여겨 보라).

app\views\categories\Edit.rhtml

```
<% @heading = "Edit Category" %>
<%= start_form_tag :action => 'update', :id => @category %>
  <%= render_partial "form" %>
  <hr />
  <%= submit_tag "Save" %>
<%= end_form_tag %>
<%= link_to 'Back', :action => 'list' %>
```

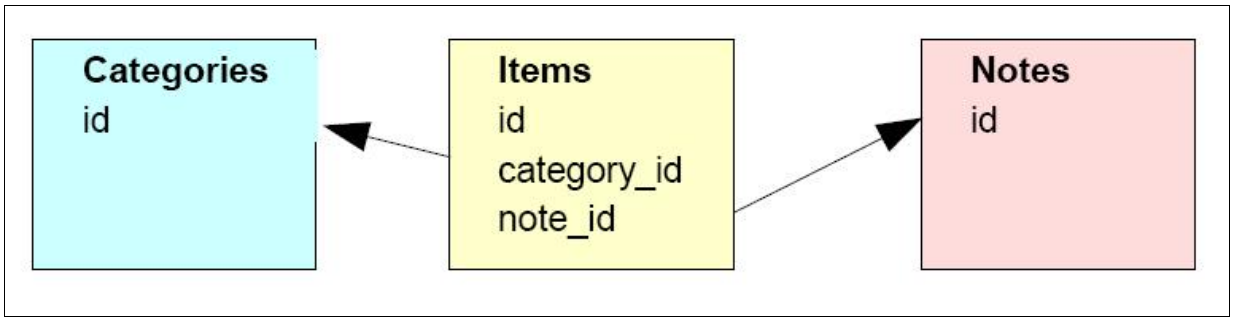
app\views\categories\New.rhtml

```
<% @heading = "New Category" %>
<%= start_form_tag :action => 'create' %>
  <%= render_partial "form" %>
  <hr />
  <%= submit_tag "Save" %>
<%= end_form_tag %>
<%= link_to 'Back', :action => 'list' %>
```

여기까지가 2일차이다. 2일차에서는 카테고리 테이블을 유지보수하는데 사용할, 실제로 작동하는 시스템을 만들어 보았으며 레일즈가 만들어내는 스캐폴드 코드를 제어하는 것을 시작해 보았다.

레일즈 3일차

이제 애플리케이션의 핵심부를 시작할 때가 됐다. Items 테이블은 'To Do'의 목록을 가지고 있다. 목록의 모든 아이템(하나의 할 일)은 Day 2에서 만든 카테고리 중 하나에 포함된다. 하나의 아이템은 하나의 노트(이 테이블은 내일 살펴볼 것이다.)를 가지고 있으며 별도의 테이블로 관리한다. 모든 테이블은 '각각의 레코드를 식별할 수 있는 주키로 'id'를 가지고 있다.



화면 4. 단순화된 데이터 모델

'Items' 테이블

MySQL 테이블 정의

Items 테이블에 있는 필드들은 다음과 같다.

- done – 1은 해당 '할 일' 아이템이 완료 됐다는 것을 나타낸다[5].
- priority – (제일 높은 우선 순위)1 부터 (가장 낮은 우선 순위) 5까지
- description – 무엇을 해야 하는지 자유롭게 적어둔 텍스트
- due_date – 언제까지 해야 하는지 나타낸다.
- category_id – 해당 아이템이 속해 있는 카테고리와의 연결점(Categories 테이블의 'id')
- note_id – 해당 아이템을 설명할 부가적인 노트와의 연결점(Notes 테이블의 'id')
- private – 1은 해당 아이템이 '개인적인' 것임을 나타낸다.

```
Items table
CREATE TABLE items (
  id smallint(5) unsigned NOT NULL auto_increment,
  done tinyint(1) unsigned NOT NULL default '0',
  priority tinyint(1) unsigned NOT NULL default '3',
  description varchar(40) NOT NULL default '',
  due_date date default NULL,
  category_id smallint(5) unsigned NOT NULL default '0',
  note_id smallint(5) unsigned default NULL,
  private tinyint(3) unsigned NOT NULL default '0',
```

5 MySQL은 'boolean'타입을 가지고 있지 않기 때문에 0과 1을 사용하겠다.

```
  created_on timestamp(14) NOT NULL,
  updated_on timestamp(14) NOT NULL,
  PRIMARY KEY (id)
) TYPE=MyISAM COMMENT='List of items to be done';
```

모델

이전과 마찬가지로 레일즈를 사용하여 비어있는 모델 파일을 만들 수 있다:

```
W:\ToDo>ruby script/generate model item
    exists app/models/
    exists test/unit/
    exists test/fixtures/
    create app/models/item.rb
    create test/unit/item_test.rb
    create test/fixtures/items.yml
W:\ToDo>
```

모델 파일을 다음과 같이 수정한다.

app\models\item.rb

```
class Item < ActiveRecord::Base
    belongs_to :category
    validates_associated :category
    validates_format_of :done_before_type_cast, :with => /[01]/,
:message=>"must be 0 or 1"
    validates_inclusion_of :priority, :in=>1..5, :message=>"must be
between 1 (high) and 5 (low)"
    validates_presence_of :description
    validates_length_of :description, :maximum=>40
    validates_format_of :private_before_type_cast, :with => /[01]/,
:message=>"must be 0 or 1"
end
```

테이블 사이의 관계 검증하기

- `belongs_to` 와 `validates_associated` 는 Items 테이블을 Categories 테이블의 item_id 필드로 연결한다.

Documentation: ActiveRecord::Associations::ClassMethods

사용자의 입력 검증하기

- `validates_presence_of`는 사용자의 입력을 받는 필드가 “NOT NULL”인지 확인한다.
- `validates_format_of`는 정규 표현식을 사용하여 사용가 입력한 값을 검증한다.
- 사용자가 숫자가 들어갈 필드에 무언가 입력하면 레일즈는 항상 숫자로 바꾸려고 시도를 한다 (무언가 잘못 되면 0으로 바뀜). 만약 사용자가 실제로 숫자를 입력했는지 확인하고 싶다면

_before_type_cast를 사용하여 사용자가 입력한 값에 접근할 수 있다[6].

- validates_inclusion_of는 사용자가 입력한 값이 허용되는 값의 범위에 포함되는지 확인한다.
- validates_length_of는 사용자가 입력한 값이 저장되지 않을 상황을 방지할 수 있다[7].

Documentation: ActiveRecord::Validations::ClassMethods

'Notes' 테이블

이 테이블은 각각의 'To Do' 아이템에 대한 보다 상세한 정보를 위한 텍스트 필드 하나를 가지고 있다. 물론 이 데이터를 Items 테이블에 속한 하나의 필드로 사용할 수도 있지만, 레일즈를 좀 더 배우기 위해 이렇게 한다. :-)

MySQL 테이블 정의

Notes table

```
CREATE TABLE notes (
  id smallint(6) NOT NULL auto_increment,
  more_notes text NOT NULL,
  created_on timestamp(14) NOT NULL,
  updated_on timestamp(14) NOT NULL,
  PRIMARY KEY (id)
) TYPE=MyISAM COMMENT='Additional optional information for to-dos';
```

모델

비어있는 모델 파일을 생성한다. 새로운 내용은 없다:

app\models\note.rb

```
class Note < ActiveRecord::Base
    validates_presence_of :more_notes
end
```

하지만 이 링크를 Items 모델에 추가하는 것을 잊지 말자.

app\models\item.rb (이 줄 생략)

```
class Item < ActiveRecord::Base
  belongs_to :note
```

---

6 좀 더 명확한 방법으로 다음과 같이 사용할 수 있다. validates_inclusion_of :done_before_type_cast,:in=>"0".."1", :message=>"must be between 0 and 1" - 입력하는 필드가 비어 있을 때 fail 된다.

7 Description 필드에 적용한 두 개의 룰을 하나로 다음과 같이 표현할 수 있다: validates_length_of :description,:within => 1..40

모델을 사용하여 참조 무결성 유지하기

우리가 개발 하는 애플리케이션은 이제 어떤 아이템이든 하나의 노트를 추가할 수 있다. 하지만 사용자가 노트와 연관을 맺고 있는 아이템을 지우려고 할 때는 어떤 일이 벌어질까? 분명히, 아이템을 삭제할 때 연관을 맺고 있는 노트 역시 삭제 하거나 '부모 없는'(연관된 Item이 없는) 노트 레코드를 그냥 남겨 둬야 한다.

모델/뷰/컨트롤러 관점에서 봤을 때 이런 코드는 모델에 속한다. 왜냐고? 나중에 'To Do' 화면에서 Dustbin 아이콘을 클릭하여 아이템을 지울 수 있지만 Purge completed 아이템을 클릭해서도 지울 수도 있기 때문이다. 모델에 그러한 코드를 넣어 둠으로써 어디서 삭제 요청이 오든지 처리할 수 있기 때문이다.

app\models\item.rb (이 줄 생략)

```
  def before_destroy
    unless note_id.nil?
      Note.find(note_id).destroy
    end
  end
```

여기 있는 코드는 아이템 레코드를 삭제하기 전에 해당 Item 레코드에 있는 Note_id 와 동일한 id를 가지는 Notes 테이블의 레코드를 찾아서 삭제한다. 만약 Note_id에 값이 없으면 그러지 않겠지만 :-)

같은 원리로, 만약에 Notes 테이블에서 레코드가 삭제되면 Items 테이블에서 Notes테이블을 참조하는 부분(Note_id)을 제거해야 한다.

app\models\note.rb (이 줄 생략)

```
  def before_destroy
    Item.find_by_note_id(id).update_attribute('note_id',NIL)
  end
end
```

Documentation: ActiveRecord::Callbacks

스캐폴드 사용하기

이제 스캐폴드 코드를 생성하자. Items 테이블과 Notes 테이블에 모두 사용한다. 아직 'Notes'를 작업할 준비가 되지 않았지만, 여기서 생성해 두는 이유는 오늘 코딩 할 부분에서 에러 없이 Notes를 참조할 수 있기 때문이다. 건물을 짓는 것과 비슷하게 스캐폴딩은 한 번에 하나씩 사고 없이 만들 수 있다.

```
W:\ToDo>ruby script/generate scaffold Item
    [snip]
W:\ToDo>ruby script/generate scaffold Note
    [snip]
W:\ToDo>
```

어제 스타일 시트를 변경했기 때문에 "overwrite public/stylesheets/scaffold.css? [Ynaq]" 프롬프트에서 n을 입력한다.

뷰 작업하기

애플리케이션에서 사용할 레이아웃 만들기

지금까지 만들었던 템플릿에서 처음 몇 줄의 코드가 전부 같았기 때문에 이 코드를 에플리케이션 범위의 레이아웃으로 옮겨야겠다. app\views\layouts*.rhtml 파일들을 삭제하고 다음과 같이 application.rhtml을 작성한다.

app\views\layouts\application.rhtml

```
<html>
<head>
  <title><%= @heading %></title>
  <%= stylesheet_link_tag 'todo' %>
<script language="JavaScript">
<!-- Begin
function setFocus() {
  if (document.forms.length > 0) {
    var field = document.forms[0];
    for (i = 0; i < field.length; i++) {
      if ((field.elements[i].type == "text") ||
            (field.elements[i].type == "textarea") ||
            (field.elements[i].type.toString().charAt(0) == "s")) {
        document.forms[0].elements[i].focus();
        break;
      }
    }
  }
}
// End -->
</script>
</head>
<body OnLoad="setFocus()">
  <h1><%=@heading %></h1>
<% if @flash["notice"] %>
<span class="notice">
<%=h @flash["notice"] %>
</span>
<% end %>
<%= @content_for_layout %>
</body>
</html>
```

템플릿에 추가한 @heading은 <h1> 또는 <title> 같은 곳에서 사용한다.

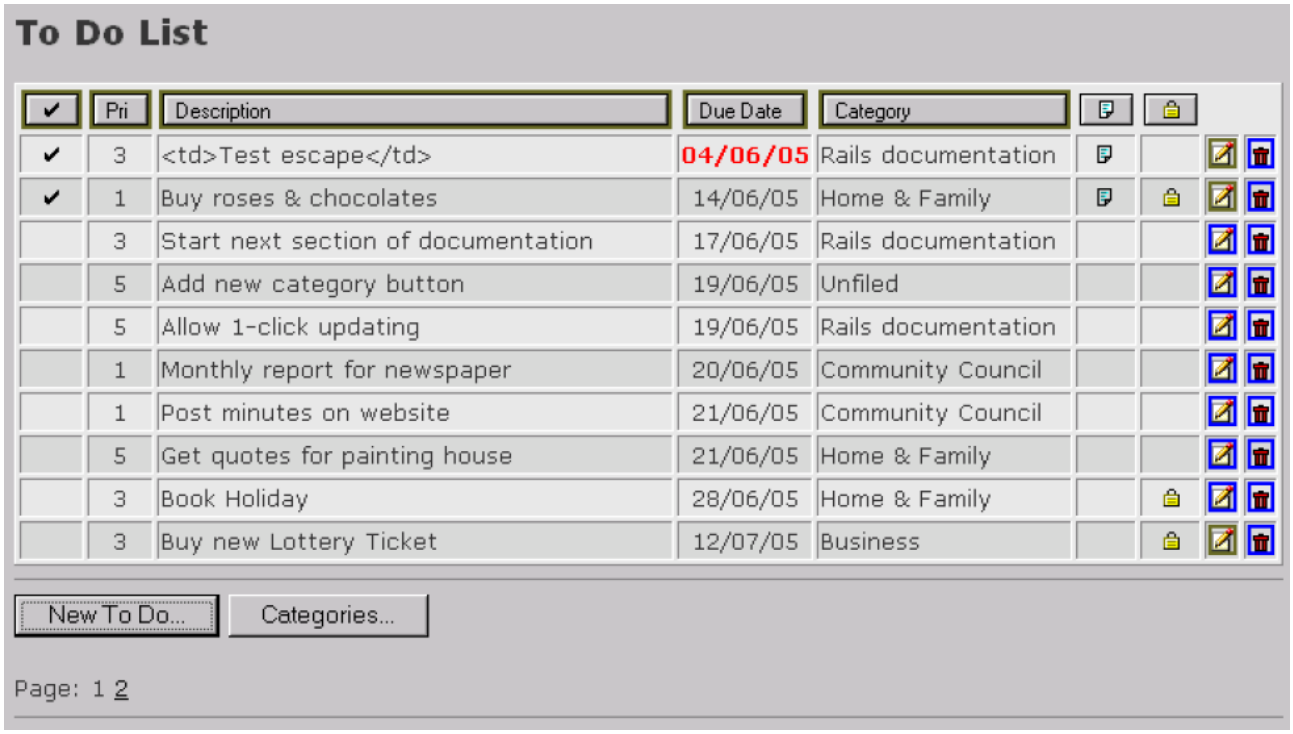
public/stylesheet/scaffold.css를 todo.css로 이름을 변경하고 색상과 테이블의 테두리에 좀 더 이쁘게 레이아웃을 적용한다. 그리고 약간의 자바스크립트를 추가하여 커서를 첫 번째로 입력할 필드에 놓도록 했다..

‘To Do List’ 화면

여기서 만들려고 하는 것은 PalmPilot 또는 PDA 데이크탑 같은 환경에서 동작하는 프로그램처럼 보이도록 할 것이다. 최종 산출물은 화면 5. ‘To Do List’ 화면에 보이는 것과 같다[8].

주의 할 것:

- ‘체크 표시(✔)’ 헤더를 클릭하여 완료한 목록(체크된 상태)을 제거할 수 있다.
- ‘Pri’, ‘Description’, ‘Due Date’, ‘Category’ 컬럼 헤더를 각각 클릭하여 목록을 정렬할 수 있다.
- ‘Done’의 값인 0과1은 ‘체크 표시’아이콘으로 변환한다.
- due date가 지난 목록은 굵은 빨간색 글자로 표시한다.
- 연관된 노트는 ‘노트’ 아이콘으로 표시한다.
- ‘Private’의 값인 0과 1은 자물쇠 모양으로 변환한다.
- 각각의 목록은 화면의 오른쪽에 있는 아이콘들을 클릭하여 편집 또는 삭제 할 수 있다.
- 전체 목록을 보여줄 때 ‘줄무늬’ 효과를 사용한다.
- 화면의 아래에 있는 “New To Do...” 버튼을 클릭하여 새로운 목록을 추가할 수 있다.
- Day 2에서 만든 ‘Categories”로 갈 수 있는 버튼 형태의 링크를 제공한다.



화면 5: ‘To Do List’ 화면

위의 템플릿을 만들기 위해 다음과 같이 작성한다.

app\views\items\list.rhtml

```
<% @heading = "To Do List" %>
<%= start_form_tag :action => 'new' %>
<table>
```

8 스타일시트에 몇 줄을 적어주는 것만으로 화면의 모양과 아이콘들 역시 바꿀 수 있다는 것은 매우 놀라운 일이다.

```
  <tr>
    <th><%= link_to_image "done", {:action => "purge_completed"},
:confirm => "Are you sure you want to permanently delete all completed To
Dos?" %></th>
    <th><%= link_to_image "priority",{:action => "list_by_priority"},
"alt" => "Sort by Priority" %></th>
    <th><%= link_to_image "description",{:action =>
"list_by_description"}, "alt" => "Sort by Description" %></th>
    <th><%= link_to_image "due_date", {:action => "list"}, "alt" => "Sort
by Due Date" %></th>
    <th><%= link_to_image "category", {:action => "list_by_category"},
"alt" => "Sort by Category" %></th>
    <th><%= show_image "note" %></th>
    <th><%= show_image "private" %></th>
    <th> </th>
    <th> </th>
</tr>
<%= render_collection_of_partials "list_stripes", @items %>
</table>
<hr />
<%= submit_tag "New To Do..." %>
<%= submit_tag "Categories...", {:type => 'button',
:onClick=>"parent.location='" + url_for( :controller => 'categories',
:action => 'list' ) + "'" } %>
<%= end_form_tag %>
<%= "Page: " + pagination_links(@item_pages, :params => { :action =>
@params["action"]
|| "index" }) + "<hr />" if @item_pages.page_count>1 %>
```

아이콘을 클릭하여 완료한 '할 일'목록 없애기

클릭할 수 있는 아이콘은 link_to_image을 사용한다. 이 것을 사용하면 기본적으로 pub/images 에서 .png가 붙은 이미지를 찾아서 클릭했을 때 특정 메소드를 실행한다.

:confirm파라미터를 추가하여 해당 작업을 하기 전에 자바스크립트 팝업 창을 보여줄 수 있다.

Documentation: ActionView::Helpers::UrlHelper

'OK'를 클릭하면 purge_completed 메소드를 호출한다. 여기서 컨트롤러에 purge_completed 메소드를 정의해 준다.

app\controllers\items_controller.rb (이 줄 생략)

```
def purge_completed
  Item.destroy_all "done = 1"
  redirect_to :action => 'list'
end
```

Item.destroy_all 은 Items 테이블에서 done 필드가 1인 레코드들을 삭제하고 list 액션을 반환한다.

Documentation: ActiveRecord::Base

컬럼 헤드 클릭하여 정열 순서 변경하기

'Pri'아이콘을 클릭하면 list_by_priority 메소드를 호출한다. 여기서 컨트롤러에 새로운 메소드인 list_by_priority를 다음과 같이 구현한다.

app\controllers\items_controller.rb (이 줄 생략)

```
  def list
    @item_pages, @items = paginate :item,
     :per_page => 10, :order_by => 'due_date,priority'
  end

  def list_by_priority
    @item_pages, @items = paginate :item,
     :per_page => 10, :order_by => 'priority,due_date'
    render_action 'list'
  end
```

list 메소드에서 기본 정렬 순서를 설정하고 list_by_priority 메소드를 만들었다[9]. 그리고 여기서 명시적으로 render_action을 'list'로 설정하여 기본적으로 레일즈에서 list_by_priority를 호출하는 요청을 list 템플릿으로 렌더링 하도록 했다.

Helper 추가하기

Note와 Pricate 컬럼의 헤더 부분은 이미지를 사용할 것이지만 클릭할 수는 없다. show_image라는 이름의 메소드를 작성하여 이미지를 보여주기로 결정했다.

app\helpers\application_helper.rb

```
module ApplicationHelper
  def self.append_features(controller)
    controller.ancestors.include?(ActionController::Base) ?
    controller.add_template_helper(self) : super
  end

  def show_image(src)
    img_options = { "src" => src.include?("/") ? src : "/images/#{src}" }
    img_options["src"] = img_options["src"] + ".png" unless
    img_options["src"].include?(".")
    img_options["border"] = "0"
    tag("img", img_options)
```

9 list_by_description과 list_by_category는 비슷하기 때문에 독자들을 위한 간단한 예제로 남겨두겠다. 하지만 만약 list_by_category 만드는 것이 지겹다면 52페이지의 "*남겨둔 일*"을 참조하기 바란다.

```
  end
end
```

다음과 같이 이 헬퍼를 컨트롤러에 연결할 수 있다.

```
app\controllers\application.rb
class ApplicationController < ActionController::Base
  helper :Application
end
```

여기서 애플리케이션에서 사용할 모든 템플릿을 참조 할 수 있다.

Documentation: ActionView::Helpers

자바스크립트 네비게이션 버튼 사용하기

onClick은 새로운 페이지로 이동하기 위한 버튼을 처리하기 위한 표준 자바스크립트 기법이다. 하지만 레일즈가 상당한 길이의 URL을 다시 만들어 낼 것이므로 레일즈가 적절한 URL을 사용하도록 해줄 필요가 있다. 주어진 controller와 action, url_for이 URL을 반환할 것이다.

Documentation: ActionController::Base

파셜을 사용하여 테이블 포맷팅하기

목록을 보여주는 리스트에 줄무늬 효과를 사용하고 싶다. render_partial 메소드를 사용하여 호출할 수 있는 파셜이 해결책이 될 수 있다.

```
<% for item in @items %>
    <%= render_partial "list_stripes", item %>
<% end %>
```

또는 좀더 경제적으로 render_collection_of_partials을 다음과 같이 사용할 수도 있다:

```
render_collection_of_partials "list_stripes", @items
```

Documentation: ActionView::Partials

레일즈는 list_stripes_counter 라는 연속적인 수를 파셜로 넘긴다. 이것을 사용하여 테이블에 있는 각각의 줄의 배경을 좀 더 밝은 회색 또는 어두운 회색으로 변경할 수 있다. 간단한 방법으로 counter가 홀수인지 짝수인지 확인해서 홀수면 밝은 회색, 짝수면 어두운 회색을 사용하자.

완성된 파셜은 다음과 같다:

```
app\views\items\_list_stripes.rhtml
<tr class="<%= list_stripes_counter.modulo(2).nonzero? ? "dk_gray" :
"lt_gray" %>">
  <td style="text-align: center"><%= list_stripes["done"] == 1 ?
show_image("done_ico.gif") : " " %></td>
  <td style="text-align: center"><%= list_stripes["priority"] %></td>
  <td><%=h list_stripes["description"] %></td>
<% if list_stripes["due_date"].nil? %>
```

```
  <td> </td>
<% else %>
  <%= list_stripes["due_date"] < Date.today ? '<td class="past_due"
style="textalign:center">' : '<td style="text-align: center">' %><%=
list_stripes["due_date"].strftime("%d/%m/%y") %></td>
<% end %>
  <td><%=h list_stripes.category ? list_stripes.category["category"] :
"Unfiled" %></td>
  <td><%= list_stripes["note_id"].nil? ? " " :
show_image("note_ico.gif") %></td>
  <td><%= list_stripes["private"] == 1 ? show_image("private_ico.gif") :
" " %></td>
  <td><%= link_to_image("edit", { :controller => 'items', :action =>
"edit", :id => list_stripes.id }) %></td>
  <td><%= link_to_image("delete", { :controller => 'items', :action =>
"destroy", :id => list_stripes.id }, :confirm => "Are you sure you want to
delete this item?") %></td>
</tr>
```

약간의 루비 코드를 사용하여 counter가 홀수인지 짝수인지 확인한 뒤 class="dk_gray" 또는 class="lt_gray"로 렌더링했다:

```
list_stripes_counter.modulo(2).nonzero? ? "dk_gray" : "lt_gray"
```

첫 번째 물음표가 있는 곳까지의 코드는 list_stripes_counter를 2로 나눈 나머지가 0이 아닌지 물어보고 있다.

Ruby Documentation: class Numeric

물음표 이전의 표현식이 참이 라면 콜론 이전의 값을 반환하고 그렇지 않다면 콜론 다음의 값을 반환한다.

Ruby Documentation: Expressions

위에서 사용한 두 개의 태그 dk_gray와 lt_gray는 스타일 시트에 정의했다.

public\stylesheets\ToDo.css (이 줄 생략)

```
.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
```

주의: 위에서 사용했던 *if then else* 구문을 체크 아이콘을 보여줄지 확인하는 부분에서도 사용했다. 만약 list_stripes["done"]이 1이면 아이콘을 보여주고 아니면 HTML로 빈 공간을 표현하는 문자로 대체한다.

```
list_stripes["done"] == 1 ? show_image("done_ico") : " "
```

데이터 값에 기반하여 포맷팅하기

특정 데이터 아이템을 하이라이팅 하는 것 역시 간단하다. – 예를 들어 이미 지나간 날짜를 확인할 경우

```
list_stripes["due_date"] < Date.today ? '<td class="past_due">' : '<td>'
```

여기서 '<td>'는 스타일 시트에 past_due라는 엔트리를 필요로 한다.

데이터를 가져올 때 손실된 데이터 다루기

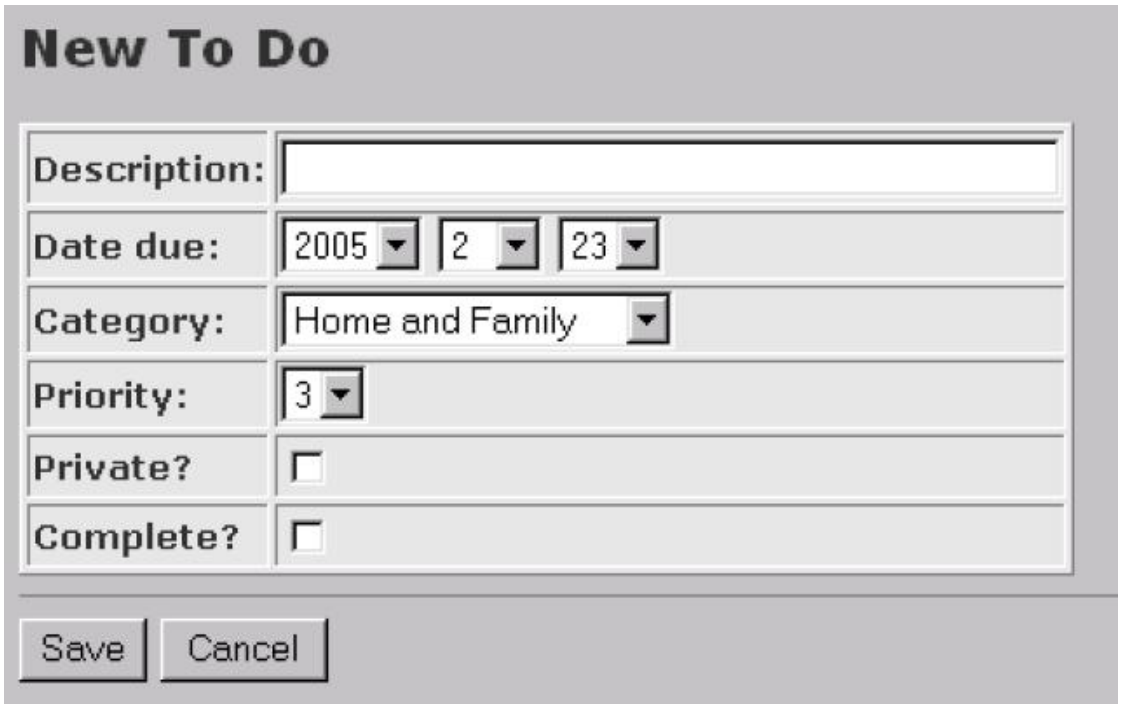
우린 사용자가 할 일 목록에서 사용하고 있는 카테고리를 삭제했을 경우를 시스템에서 대비하길 원한다. 이런 상황이 발생할 경우 카테고리는 "Unfiled"로 보여져야 한다.

```
list_stripes.category ? list_stripes.category["category"] : 'Unfiled'
```

OK. 만약 여기까지 잘 따라왔다면 화면 5와 같은 "할일 목록"을 볼 수 있을 것이다. "할 일 목록"은 31페이지에 나와있다.

"New To Do" 화면

다음으로 넘어가서 "New To Do..." 버튼을 클릭했을 때를 생각해보자. 여기에도 몇 가지 트릭이 숨어있다.



화면 6.새 'To Do' 화면

화면 탬플릿은 최소한 다음과 같다:

app\views\items\new.rhtml

```
<% @heading = "New To Do" %>
<%= error_messages_for 'item' %>
<%= start_form_tag :action => 'create' %>
  <table>
<%= render_partial "form" %>
  </table>
```

```
  <hr />
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='"
+ url_for(:action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

그리고 실제 작업은 “Edit” 액션과 공유하는 작동은 파셜을 사용할 것이다.

```
app\views\items\_form.rhtml
  <tr>
    <td><b>Description: </b></td>
    <td><%= text_field "item", "description", "size" => 40, "maxlength"
=> 40 %></td>
  </tr>
  <tr>
    <td><b>Date due: </b></td>
    <td><%= date_select "item", "due_date", :use_month_numbers => true
%></td>
  </tr>
  <tr>
    <td><b>Category: </b></td>
    <td><select id="item_category_id" name="item[category_id]">
      <%= options_from_collection_for_select @categories, "id",
"category", @item.category_id %>
        </select>
    </td>
  </tr>
  <tr>
    <td><b>Priority: </b></td>
    <% @item.priority = 3 %>
    <td><%= select "item","priority",[1,2,3,4,5] %></td>
  </tr>
  <tr>
    <td><b>Private? </b></td>
    <td><%= check_box "item","private" %></td>
  </tr>
  <tr>
    <td><b>Complete? </b></td>
    <td><%= check_box "item", "done" %></td>
  </tr>
```

*데이터 필드용 **Drop-down** 리스트 만들기*

date_select를 사용하여 데이터를 입력 받을 기본적인 드롭 다운 메뉴를 만들 수 있다:

```
date_select "item", "due_date", :use_month_numbers => true
```

Documentation: ActionView::Helpers::DateHelper

루비에서의 예외 처리

불행히도 date_select는 매우 능청스런 모습으로 2월 31일과 같은 데이터를 받아들일 수 있다. 그러면 레일즈는 이러한 데이터를 데이터베이스에 넣으려고 할 때 죽게 된다. 이러한 저장 실패를 처리하는 방법으로 루비의 예외 처리 메소드인 rescue를 사용한다.

app\controllers\items_controller.rb (이 줄 생략)

```
  def create
    begin
      @item = Item.new(@params[:item])
      if @item.save
        flash['notice'] = 'Item was successfully created.'
        redirect_to :action => 'list_by_priority'
      else
        @categories = Category.find_all
        render_action 'new'
      end
    rescue
      flash['notice'] = 'Item could not be saved.'
      redirect_to :action => 'new'
    end
  end
```

Ruby Documentation: Exceptions, Catch, and Throw

***Lookup Table*에서 드롭 다운 리스트 만들기**

다음의 예는 레일즈에서 코딩 문제를 해결하는 고도로 경제적인 방법이다. 위 예제의 경우는 다음과 같이 적용할 수 있다.

```
options_from_collection_for_select @categories, "id", "category",
@item.category_id
```

options_from_collection_for_select는 카테고리 테이블에 있는 모든 레코드를 읽어 들이고 그것들을 <optionvalue="[value of id]">[value of category]</option> 이렇게 렌더링한다. @item_category_id 에 해당하는 레코드에 selected 태그를 붙인다. 이것으로 충분하지 않을 때는 여러분이 짠 html을 추가할 수도 있다.

Documentation: ActionView::Helpers::FormOptionsHelper

데이터로 만들 드롭 다운 박스는 데이터를 어딘가로부터 가져와야 한다. 이 말은 컨트롤러에 무언가를 추가해야 함을 뜻한다:

app\controllers\items_controller.rb (이 줄 생략)

```
  def new
    @categories = Category.find_all
```

```
    @item = Item.new
  end

  def edit
    @categories = Category.find_all
    @item = Item.find(@params[:id])
  end
```

상수들의 드롭 다운 리스트 만들기

앞에서 살펴본 시나리오와 비슷한 방법이다. 직접 코드로 작성한 리스트의 값들로 선택할 수 있는 박스를 만드는 것이 항상 좋은 생각은 아니다. 테이블에 있는 테이터를 변경하는 것이 코드에 있는 값을 변경하는 것보다 쉽다. 하지만 코드로 리스트 박스를 만들어야 할 경우도 있으며 레일즈도 그렇게 할 수 있다.

```
select "item","priority",[1,2,3,4,5]
```

이전의 코드에서 어떻게 기본값을 설정했는지 살펴보기 바란다.

Documentation: ActionView::Helpers::FormOptionsHelper

체크박스 만들기

이번 요구 사항은 레일즈의 또 다른 헬퍼가 해결한다.

```
check_box "item","private"
```

Documentation: ActionView::Helpers::FormHelper

마무리 하기

스타일시트 조정하기

이 시점에서 “할 일 목록”화면이 잘 작동해야 한다. “새로운 할 일”버튼 역시 마찬가지로 잘 동작해야 한다. 이 때 보이는 화면을 만들기 위해 다음과 같이 스타일 시트를 변경했다.

public\stylesheets\ToDo.css

```
body { background-color: #c6c3c6; color: #333; }

.notice {
  color: red;
  background-color: white;
}
h1 {
  font-family: verdana, arial, helvetica, sans-serif;
  font-size: 14pt;
  font-weight: bold;
}
```

```
table {
  background-color:#e7e7e7;
  border: outset 1px;
    border-collapse: separate;
    border-spacing: 1px;
}

td { border: inset 1px; }
.notice {
  color: red;
  background-color: white;
}
.lt_gray { background-color: #e7e7e7; }
.dk_gray { background-color: #d6d7d6; }
.hightlight_gray { background-color: #4a9284; }
.past_due { color: red }
```

"Edit To Do" 화면

3일째에서 남은 것은 "Edit To Do"화면을 만드는 것인데 이것은 "New To Do"과 매우 비슷하다. 대학을 다닐 때 교재의 내용 중 가장 짜증났었던 부분이 바로 "*이 부분은 독자들을 위한 연습문제로 남겨두겠습니다.*"였다. 따라서 나도 여러분에게 그렇게 하는 것이 좋겠다[10].

여기까지가 3일째 진도가 끝났다. 지금의 애플리케이션은 레일즈 스캐폴드와는 많이 달라 보이지만 그 표면 밑에는 개발을 편리하게 하는 레일즈의 다양한 도구들을 사용했다.

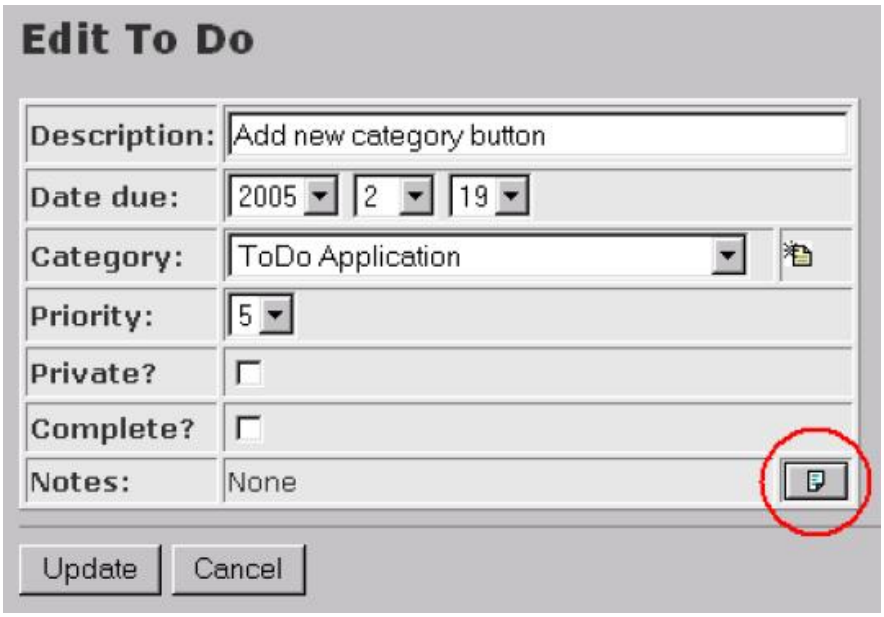
10 하지만 내가 읽었던 대학교 교재의 저자들과는 다르게 나는 4일째에서 해답을 제시해 주겠다. :-) 43페이지의 `app\views\items\edit.rhtml` 을 보면 된다.

레일즈 4일차

“노트” 화면

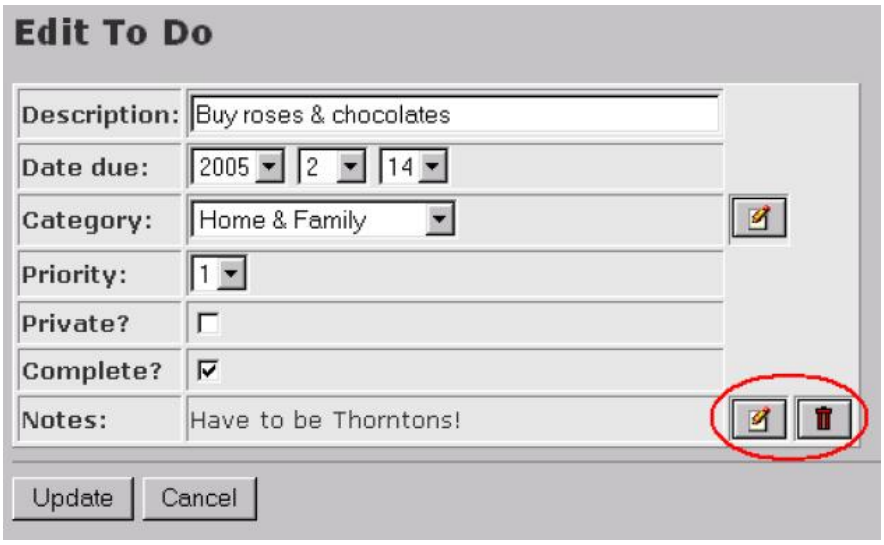
‘Note’를 “Edit To Do”에 연결하기

비록 Notes 스캐폴드가 CRUD 기능을 지원해 주긴 하지만, 사용자가 이를 직접 호출하기를 원하지 않는다. 그대신 만약 어떤 목록이 연관된 노트를 가지고 있지 않을 때, “Edit To Do”화면에서 노트 아이콘을 클릭하여 해당 할 일 목록과 연관된 노트를 생성할 수 있도록 하고 싶다.



화면 7. “Edit To Do”화면에서 새 노트 만들기

만약 노트가 이미 존재한다면 “Edit To Do”화면에서 적당한 아이콘을 클릭하여 수정 또는 삭제하고 싶다.



화면 8. 기존 노트를 편집 혹은 삭제하기

먼저 “Edit To Do”화면을 위한 코드를 살펴보자. 노트가 존재하는지에 따라 아이콘이 어떻게 변하는지 주의해서 보기 바란다. 노트 컨트롤러에서는 이를 어떻게 제어할지 살펴보자.

app\views\items\edit.rhtml

```
<% @heading = "Edit To Do" %>
<%= error_messages_for 'item' %>
<%= start_form_tag :action => 'update', :id => @item %>
  <table>
<%= render_partial "form" %>
    <tr>
      <td><b>Notes: </b></td>
<% if @item.note_id.nil? %>
      <td>None</td>
      <td><%= link_to_image "note", :controller => "notes", :action =>
"new", :id => @item.id %></td>
<% else %>
      <td><%=h @item.note.more_notes %></td>
      <td><%= link_to_image "edit_button", :controller => "notes",
:action => "edit", :id => @item.note_id %></td>
      <td><%= link_to_image "delete_button", {:controller => "notes",
:action => "destroy", :id => @item.note_id }, :confirm => "Are you sure
you want to delete this note?" %></td>
<% end %>
    </tr>
  </table>
  <hr />
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='"
+ url_for( :action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

“Edit Notes” 화면

기존에 존재하는 노트를 수정하는 것은 매우 간단하다. 아래의 코드는 템플릿이다.

app\views\notes\edit.rhtml (이 줄 생략)

```
<% @heading = "Edit Note" %>
<%= start_form_tag :action => 'update', :id => @note %>
  <%= render_partial "form" %>
  <%= submit_tag "Save" %>
  <%= submit_tag "Cancel", {:type => 'button',
:onClick=>"parent.location='" + url_for( :controller => 'items', :action
=> 'list' ) + "'" } %>
<%= end_form_tag %>
```

다음은 여기서 사용하는 Partial이다.

app\views\notes_form.rhtml (이 줄 생략)

```
<table>
  <tr>
    <td><label for="note_more_notes">More notes</label></td>
    <td><%= text_area 'note', 'more_notes' %></td>
  </tr>
</table>
```

노트 테이블에 수정 또는 삭제가 완료 된 뒤 “To Do List” 화면으로 이동하길 원한다.

app\controllers\notes_controller.rb (이 줄 생략)

```
  def update
    @note = Note.find(@params[:id])
    if @note.update_attributes(@params[:note])
      flash['notice'] = 'Note was successfully updated.'
      redirect_to :controller => 'items', :action => 'list'
    else
      render_action 'edit'
    end
  end

  def destroy
    Note.find(@params[:id]).destroy
    redirect_to :controller => 'items', :action => 'list'
  end
```

이전에 대비해 만들었던 참조 무결성 법칙을 기억한다면 노트가 삭제될 때 아이템 테이블에서도 해당 링크가 삭제 될 것이라는 것을 짐작할 수 있다(29페이지의 ‘모델을 사용하여 참조 무결성 유지하기’를 참조하라).

“New Notes”화면

노트를 생성하는 것은 약간 복잡하다. 다음과 같이 하고 싶다.

- 새 노트를 Notes 테이블에 저장할 것이다.
- Notes 테이블에 새로 생성한 레코드의 id를 가져올 것이다.
- 이 id를 연관된 Items 테이블의 notes_id 필드에 저장할 것이다.

세션 변수는 여러 화면 사이에 데이터를 유지하기 위한 유용한 방법을 제공한다. – 이것을 사용하여 Notes 테이블의 id를 저장할 수 있다.

Documentation: ActionController::Base

세션 변수를 사용하여 데이터 저장하기/가져오기

먼저 새로운 Notes 레코드를 생성하려고 할 때 현재 수정하고 있는 할 일 목록(Item)의 id를 넘긴다:

app\views\items\edit.rhtml (이 줄 생략)

```
          <td><%= link_to_image "note", :controller => "notes", :action =>
"new", :id => @item.id %></td>
```

Notes 컨트롤러에 있는 new 메소드는 이 것을 세션 변수에 저장한다.

app\controllers\notes_controller.rb (이 줄 생략)

```
  def new
    @session[:item_id] = @params[:id]
    @note = Note.new
  end
```

“New Notes” 템플릿은 별다른 것이 없다.

app\views\notes\new.rhtml

```
<% @heading = "New Note" %>
<%= start_form_tag :action => 'create' %>
<%= render_partial "form" %>
<%= submit_tag "Save" %>
<%= submit_tag "Cancel", {:type => 'button', :onClick=>"parent.location='"
+ url_for( :controller => 'items', :action => 'list' ) + "'" } %>
<%= end_form_tag %>
```

create 메소드는 session 변수를 가져와서 Items 테이블에 있는 레코드를 찾는데 사용한다. 그러고 나서 Items 테이블에 있는 note_id를 방금 생성한 Note 테이블의 id로 수정하고 아이템 컨트롤러로 돌아간다.

‘Categories’화면 변경하기

이제 해야 할 작업이 얼마 남지 않았다. 이전에 생성했었던 템플릿들을 여기서 사용하는 네비게이션 버튼을 사용하도록 변경할 것이다.

app\views\categories\list.rhtml

```
<% @heading = "Categories" %>
<form action="/categories/new" method="post">
<table>
  <tr>
    <th>Category</th>
    <th>Created</th>
    <th>Updated</th>
  </tr>
<% for category in @categories %>
  <tr>
    <td><%=h category["category"] %></td>
    <td><%= category["created_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= category["updated_on"].strftime("%I:%M %p %d-%b-%y") %></td>
    <td><%= link_to_image 'edit', { :action => 'edit', :id => category.id
} %></td>
    <td><%= link_to_image 'delete', { :action => 'destroy', :id =>
category.id }, :confirm => 'Are you sure you want to delete this
category?' %></td>
  </tr>
```

```
<% end %>
</table>
<hr />
  <input type="submit" value="New Category..." />
  <input type="button" value="To Dos" onClick="parent.location='<%=
url_for( :controller => 'items', :action => 'list' ) %>'">
</form>
```

app\views\categories\new.rhtml

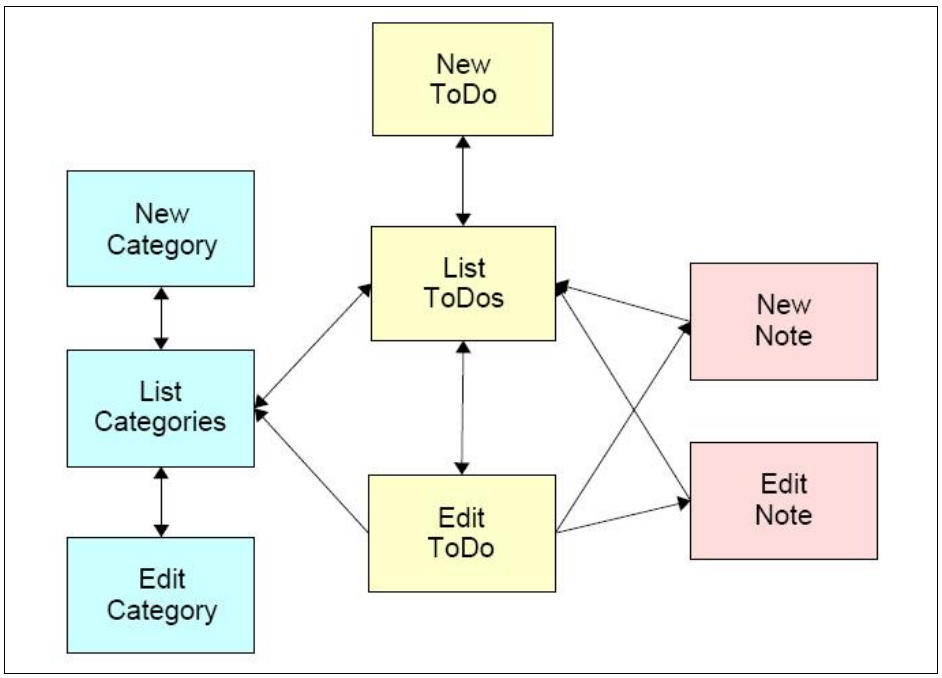
```
<% @heading = "Add new Category" %>
<%= error_messages_for 'category' %>
<%= start_form_tag :action => 'create' %>
  <%= render_partial "form" %>
  <hr />
  <input type="submit" value="Save" />
  <input type="button" value="Cancel" onClick="parent.location='<%=
url_for( :action => 'list' ) %>'">
<%= end_form_tag %>
```

app\views\categories\edit.rhtml

```
<% @heading = "Rename Category" %>
<%= error_messages_for 'category' %>
<%= start_form_tag :action => 'update', :id => @category %>
  <%= render_partial "form" %>
  <hr />
  <input type="submit" value="Update" />
  <input type="button" value="Cancel" onClick="parent.location='<%=
url_for( :action => 'list' ) %>'">
<%= end_form_tag %>
```

시스템 네비게이션

애플리케이션의 최종 네비게이션 경로가 아래에 나와있다. 스캐폴드로 생성한 코드 중에 사용하지 않는 코드(예. `show.rhtml` 파일)를 삭제 한다. 아무런 수고로움 없이 코드를 생성해서 할 일을 다 한 뒤에 필요 없는 코드는 제거할 수 있다는 것이 스캐폴드 코드의 장점이다.



화면 9. 애플리케이션 네비게이션경로

애플리케이션의 홈페이지 설정하기

마지막 단계로, `http://todo`로 접속 했을 때 보이는 기본 화면인 "Welcome to Rails" 를 제거해야 한다. 두 단계로 이루어진다:

- Routes 파일에 홈 페이지 정의를 추가한다.

```
config\routes.rb (이 줄 생략)
    map.connect '', :controller => 'items'
```

- `public\index.html` 을 `public\index.html.orig` 로 변경한다.

애플리케이션 다운로드하기

"To Do" 애플리케이션을 다운로드 하고 실행하고 싶으면 http://rails.homelinux.org 에서 다운로드 할 수 있다. 다운로드 한 다음에

- 레일즈를 사용하여 디렉토리 구조를 생성한다(4페이지의 레일즈 스크립트 실행하기 참조).
- `todo_app.zip` 파일을 새로 생성한 `ToDo` 디렉토리로 다운로드 한다.
- `upzip -o todo_app.zip` 명령을 사용하여 압축을 푼다.
- `public\index.html` 을 `public\index.html.orig` 로 변경한다.
- 예제 데이터베이트를 사용하고 싶으면 `mysql -uroot -p < db/ToDo.sql` 명령을 사용한다.

마지막으로

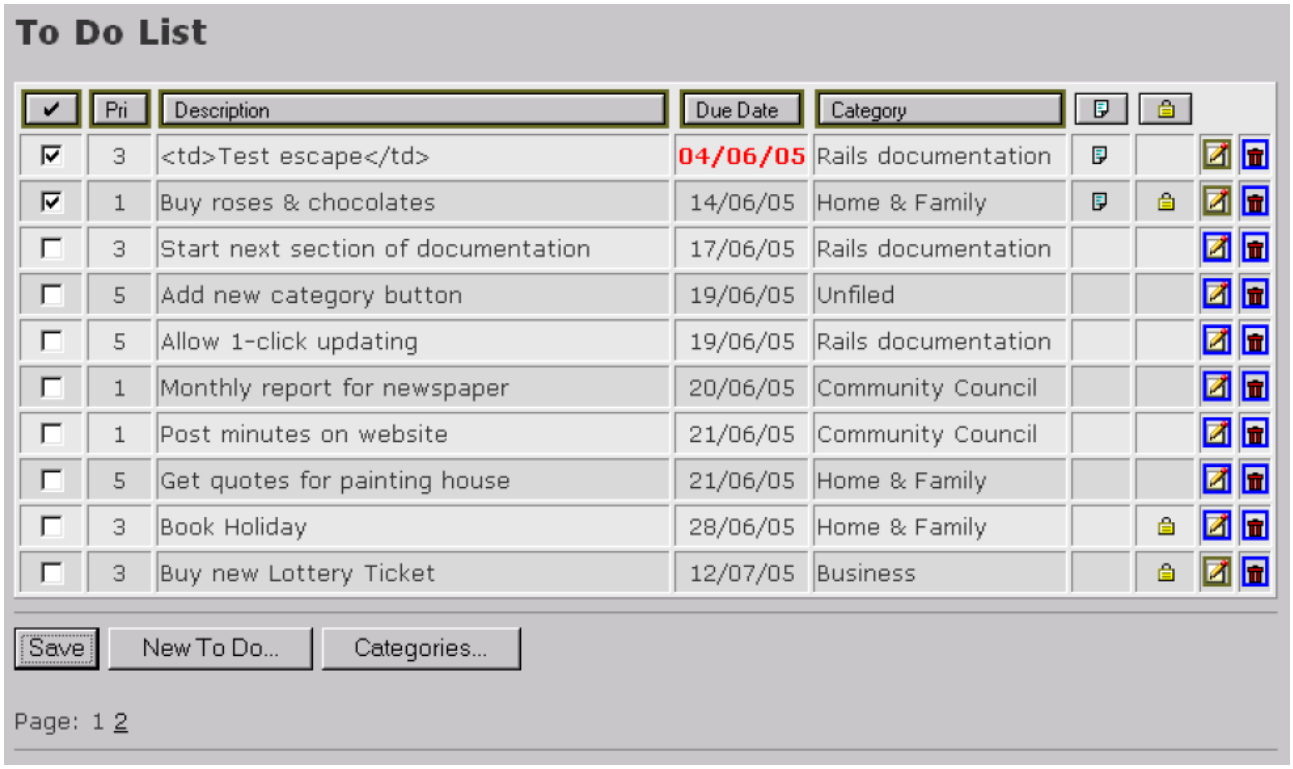
이 문서가 여러분에게 유용했으면 좋겠다. 나는 좋든 안 좋든 피드백을 받는 것을 행복해 한다. 레일즈로 즐거운 코딩을!

부록 – 다시 생각해 볼 부분

“Four Days”를 쓴 뒤, 이 문서의 질을 높이는데 도움이 되는 상당히 많은 양의 피드백을 받았다. 그 중 가장 자주 물어오는 질문이 “어떻게 한 화면에서 여러 개의 레코드를 갱신할 수 있을까?” 라는 것이다. 따라서 여기서 부록으로 FAQ 형식으로 다루고자 한다. 이 부분은 레일즈 개념으로 파악하기 쉽지 않은 부분이며 보다 더 많은 “도우미”들이 나타나길 바란다.

다중 갱신(Multiple Updates)

아래에 있는 화면에서 사용자는 제일 왼쪽에 있는 체크박스를 사용하여 여러 개의 “할 일” 목록을 선택하거나 선택하지 않을 수 있다. 그리고 “저장” 버튼을 클릭하면 데이터베이스에 그 결과를 저장한다.



화면 10. 다중 갱신

뷰

레일즈는 다중 갱신을 별도의 작명 규약(naming convention)으로 지원한다. 여러분이 편집한 레코드의 id를 대괄호[ ]로 감싸서 name에 추가한다. 이것을 사용하여 화면에 있는 목록들 중에 일부를 선택할 수 있다.

우리가 만들려고 하는 HTML로 돌아가보자. `id = 6`인 레코드에서 다음과 같은 모습을 볼 수 있다.

```
<td style="text-align: center">
 <input type="checkbox" id="item_done" name="item[6][done]" value="1"
checked />
 <input name="item[6][done]" type="hidden" value="0" />
```

```
</td>
```

(만약에 체크박스가 체크된 상태가 아니라면 "checked"는 생략된다.)

이런 코드를 생성할 수 있는 방법 중에 하나는 다음과 같다:

```
app\view\items\_list_stripes.rhtm (이 줄 생략)
<td style="text-align: center">
  <%=check_box_tag("item["+list_stripes.id.to_s+"][done]","1",list_stripes
["done"]==1) %>
  <%=hidden_field_tag("item["+list_stripes.id.to_s+"][done]","0") %>
</td>
```

check_box_tag의 파라미터는 name, value = "1", checked = false, option = {}이고 hidden_field_tag의 파라미터는 name, value = nil, option = {} 이다.

Documentation: ActionView::Helpers::FormTagHelper

여기에 물론 저장 버튼이 있어야 한다.

```
app\views\items\list.rhtml (이 줄 생략)
<% @heading = "To Do List" %>
<%= start_form_tag :action => 'updater' %>
<table>
...
</table>
<hr />
<%= submit_tag "Save" %>
<%= submit_tag "New To Do...", {:type => 'button',
:onClick=>"parent.location='" + url_for( :controller => 'items', :action
=> 'new' ) + "'" } %>
<%= submit_tag "Categories...", {:type => 'button',
:onClick=>"parent.location='" + url_for( :controller => 'categories',
:action => 'list' ) + "'" } %>
<%= end_form_tag %>
<%= "Page: " + pagination_links(@item_pages, :params => { :action =>
@params["action"] || "index" }) + "<hr />" if @item_pages.page_count>1 %>
```

컨트롤러

"저장" 버튼을 클릭하여 컨트롤러로 돌아오면 다음과 같다.

```
params: {
  :controller=>"items",
  :item=> {
    "6"=>{"done"=>"0"},
    ... etc...
    "5"=>{"done"=>"1"}
  },
```

```
  :action=>"updater"
}
```

:item 을 살펴보자. 굵은 글자 부분이 "done 필드가 0이고 id가 6인 레코드"다. 여기서는 Items 테이블을 갱신하는 것이 쉽다.

app\controller\items_controller (이 줄 생략)

```
  def updater
    @params[:item].each { |item_id, attr|
      item = Item.find(item_id)
      item.update_attribute(:done,attr[:done])
    }
    redirect_to :action => 'list'
end
```

each는 "6"을 item_id 변수에, "done" => "0"을 attr에 입력한다.

Ruby Documentation: class Array

이 코드는 잘 동작하지만 내부에서 어떻게 돌아가는지 궁금하다면 development.log를 보면 레일즈가 모든 레코드를 가져와서 바뀌었는지 확인한 다음 수정한다. 데이터베이스에 이러한 수정을 가하는 것은 불필요한 작업일 뿐만 아니라 updated_on 의 의도가 변질 되었으며 이것은 전혀 원하던 결과가 아니다. 'done' 이 변경된 레코드만 수정하는 것이 좋으며 별도의 코딩이 필요하다. :-(문자열 타입인 done을 to_i를 사용하여 정수로 바꿔서 비교해야 한다. 이런 종류의 실수를 종종 범할 수 있기 때문에 development.log를 확인하여 레일즈가 예상한대로 동작하는지 자주 확인해 줘야 한다.

app\controller\items_controller (이 줄 생략)

```
  def updater
    @params[:item].each { |item_id, contents|
      item = Item.find(item_id)
      if item.done != contents[:done].to_i
        item.update_attribute(:done,contents[:done])
      end
    }
    redirect_to :action => 'list'
  end
```

사용자 인터페이스에서 고려할 사항

이제 코드는 잘 동작하기 때문에 화면에 있는 모든 필드들을 수정할 수 있도록 변경할 수도 있을 것이다(독자들을 위한 또 다른 연습문제다. :-) 사용자가 어떤 것을 원할까 라는 의문이 생긴다. 만약 사용자가 체크 박스 몇 개를 수정한 다음에 "Save"버튼을 클릭하지 않고 "New To Do..."를 클릭하거나 화면의 정렬 상태를 바꾸면 어떻게 될까? 어떤 작업들을 하기 전에 시스템이 알아서 "Save" 해줘야 할까? 이것 역시 독자들을 위한 좀 더 쉬운 연습문제로 남겨두겠다.

남아있는 작업

32페이지에 list_by_category를 독자들을 위한 간단한 연습문제로 남겨뒀었다. 아쉽지만 보기보다 덜 간단하다는 것을 보여줘야겠다. 사실 여전히 테이블을 필드별로 정렬할 좀 더 멋진 '레일즈'스러운 방법을 찾고 있는 중이다. 약간 끔찍한 코드로 이 글을 마무리 해야겠다.

app\controller\items_controller (이 줄 생략)

```
def list_by_category
  @item_pages = Paginator.new self, Item.count, 10, @params['page']
  @items = Item.find_by_sql 'SELECT i.*, c.category FROM categories c,
items i ' +
    'WHERE ( c.id = i.category_id ) '+
    'ORDER BY c.category ' +
    'LIMIT 10 ' +
    "OFFSET #{@item_pages.current.to_sql[1]}"
  render_action 'list'
end
```

누구라도 좀 더 좋은 해결책이 있다면 나에게 알려주길 바란다. 어찌됐든 실패한 예제라는 것을 재확인 할 겸 위의 코드를 남겨둘 텐데, 레일즈는 여러분들을 그냥 내버려두지 않을 것이며 오히려 여러분들이 최후의 수단으로 구식 코딩이라도 할 수 있도록 해줄 것이다.

즐겨라. 레일즈 코딩을!