<!DOCTYPE html> <!-- The above 3 meta tags *must* come first in the head; any other head content must come *after* these tags -->

<!-- HTML5 shim and Respond.js for IE8 support of HTML5 elements and media queries --> <!-- WARNING: Respond.js doesn't work if you view the page via file:// --> <!--[if lt IE 9]> <![endif]-->

The Ruby on Rails Tutorial

: Korean Edition, Draft

Welcome, 임재룡(R팀) | 목차 | 원문

```
  | <a href="/signout">Log Out</a>
</div>
```

```
<ul id='chapter_navigator' class='sticker'>
  <li class=""><a href="frontmatter.html">Front matter</a> </li>
  <li class=""><a href="beginning.html">Chapter 1: From zero to deploy</a> </li>
  <li class=""><a href="toy_app.html">Chapter 2: A toy app</a> </li>
  <li class="active">Chapter 3: Mostly static pages </li>
  <li class=""><a href="rails_flavored_ruby.html">Chapter 4: Rails-flavored Ruby</
a> </li>
  <li class=""><a href="filling_in_the_layout.html">Chapter 5: Filling in the layo
ut</a> </li>
  <li class=""><a href="modeling_users.html">Chapter 6: Modeling users</a> </li>
  <li class=""><a href="sign_up.html">Chapter 7: Sign up</a> </li>
  <li class=""><a href="log_in_log_out.html">Chapter 8: Log in, log out</a> </li>
  <li class=""><a href="updating_and_deleting_users.html">Chapter 9: Updating, sho
wing, and deleting users</a> </li>
  <li class=""><a href="account_activation_password_reset.html">Chapter 10: Accoun
t activation and password reset</a> </li>
  <li class=""><a href="user_microposts.html">Chapter 11: User microposts</a> </li
>
  <li class=""><a href="following_users.html">Chapter 12: Following users</a> </li
>
</ul>
```

<!DOCTYPE html>

```
<title>ruby_on_rails_tutorial</title>

  <script type="text/x-mathjax-config">
          MathJax.Hub.Config({
    "HTML-CSS": {
      availableFonts: ["TeX"],
    },
    TeX: {
      extensions: ["AMSmath.js", "AMSsymbols.js", "color.js"],
      equationNumbers: {
        autoNumber: "AMS",
        formatNumber: function (n) { return "3" + '.' + n }
      },
      Macros: {
        PolyTeX:    "Poly{\\TeX}",
        PolyTeXnic: "Poly{\\TeX}nic",
        "failing": "\\coloredtext{red}{\\textsc{\\textbf{red}}}",
```

"passing": "\coloredtext{ForestGreen}{\textsc{\textbf{green}}}" } }, showProcessingMessages: false, messageStyle: "none", imageFont: null });

```
  </script>
  <script type="text/javascript" src="MathJax/MathJax.js?config=TeX-AMS-MML_SVG"><
/script>
```

```
<div id="book">
  <div id="cha-static_pages" data-tralics-id="cid15" class="chapter" data-number="
3" data-chapter="static_pages"><h1><a href="static_pages.html#cha-static_pages" cl
ass="heading hyperref"><span class="number">Chapter 3 </span>정적 페이지</a></h1>
```

이번 장에서는, 이후 튜토리얼에서 꾸준히 사용할 전문가 수준의 sample 애플리케이션 개발을 시작한다. sample 애플리케이션은 users, microposts 그리고 로그인과 인증 프레임워크를 포함하는데, 이번 장에서는 이중에서 정적 페이지 제작에 집중한다. 정적 페이지 만들기는 단순하고 쉽지만, 연습에 적합하고 다양한 내용을 배울수 있어서, 첫 애플리케이션 제작에 훌륭한 출발점이 된다.

레일스는 데이터베이스 기반의 동적 웹사이트를 제작하기 적합하지만, 기본 HTML 파일같은 정적 페이지 생성에도 유용하다. 사실, 레일스로 정적 페이지 생성을 하면 독특한 장점이 있다. *적은* 양의 동적인 내용을 쉽게 추가할 수 있다. 이번 장에서는 이런 부분들의 구현 방식을 배울 것이다. 우선 첫번째로 자동화된 테스트환경을 만들 것이며, 이는 작성한 코드가 정확하게 동작하는 것을 더욱 신뢰하도록 도와 줄 것이다. 더불어 훌륭한 테스트 코드는 애플리케이션의 기능 변경 없이 코드의 모양을 변경하는 *리팩터(refactor)* 작업을 안심하고 진행할 수 있게 도와준다.

3.1 Sample 애플리케이션 셋업

시작하기 앞서 이번에도 2장처럼 `sample_app` 란 이름의 새로운 레일스 프로젝트를 목록 3.1과 같이 생성한다.[1] 만약

목록 3.1 의 명령을 실행할 때 “Could not find ’railties’” 라는 에러를 만났다면, 이는 일치하는 레일스가 설치되어 있지 않다는 것을 의미하므로, 목록 1.1 **(목록 1.2로 이동 됩니다.)**를 정확히 수행했는지 다시 한번 확인해야 한다.

목록 3.1: Sample 앱을 새로 만들기

```
$ cd ~/workspace
$ rails _4.2.0_ new sample_app
$ cd sample_app/
```

(섹션 2.1 처럼, 클라우드 IDE 사용자는 앞의 두 장에서 만든 애플케이션과 같은 워크스페이스에서 이 프로젝트를 만들 수 있다. 따라서 새로운 워크스페이스를 만들 필요는 없다.)

섹션 2.1처럼 다음 단계는 텍스트 에디터를 이용해 애플리케이션에 필요한 젬을 `Gemfile` 에 추가 갱신한다. 목록 3.2 는 `test` 그룹내의 젬을 제외하고는목록 1.5과 목록 2.1과 동일하며 이 그룹은 높은 수준의 테스트 환경 설정에 선택적으로 필요하다(섹션 3.7). *노트* : sample 애플리케이션이 의존하는 *모든* 젬을 설치하려면 목록 11.66 코드를 사용한다.

목록 3.2: Sample 앱의 `Gemfile`

```
source 'https://rubygems.org'

gem 'rails',                '4.2.0'
gem 'sass-rails',           '5.0.1'
gem 'uglifier',             '2.5.3'
gem 'coffee-rails',         '4.1.0'
gem 'jquery-rails',         '4.0.3'
gem 'turbolinks',           '2.3.0'
gem 'jbuilder',             '2.2.3'
gem 'sdoc',                 '0.4.0', group: :doc

group :development, :test do
  gem 'sqlite3',     '1.3.9'
  gem 'byebug',      '3.4.0'
  gem 'web-console', '2.0.0.beta3'
  gem 'spring',      '1.1.3'
end

group :test do
  gem 'minitest-reporters', '1.0.5'
  gem 'mini_backtrace',     '0.1.3'
  gem 'guard-minitest',     '2.3.1'
end

group :production do
  gem 'pg',             '0.17.1'
  gem 'rails_12factor', '0.0.2'
end
```

앞의 두 장처럼, `bundle install`를 실행해서 `Gemfile`에 명시한 젬을 설치하는데, 이 때 --without production 옵션을 지정해서 운영 환경용 젬 설치는 건너뛴다.[2]

```
$ bundle install --without production
```

개발과 테스트 환경에서는 SQLite 를 사용하므로, PostgreSQL를 위한 pg 젬은 건너뛴다. 허로쿠는 개발환경과 운영환경에서 서로 다른 종류의 데이터베이스 사용을 추천하지 않지만, sample 애플리케이션에서는 별 다른 차이가 없고, SQLite는 PostgresSQL보다 로컬에 설치하여 사용하기가 *매우* 쉽다.[3] `Gemfile`에 명시된 젬 외의 특정 버전의 젬(레일스 자체도 포함)을 앞서 설치한 적이 있다면, 버전 맞추기 위해 `bundle update` 명령으로 젬들을 *갱신*하는 것이좋다.

```
$ bundle update
```

지금까지의 작업과 함께, 이제 Git 저장소를 초기화하는 것만이 남았다.

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

첫번째 애플리케이션에서와 같이, `README` 파일(애플리케이션의 루트 디렉토리에 위치)을 변경할 것을 추천한다. RDoc 을 마크다운 형식으로 변경한다.

```
$ git mv README.rdoc README.md
```

그리고 목록 3.3 과 같이 파일 내용을 수정한다.

목록 3.3: Sample 앱의 향상된 `README` 파일

```
# Ruby on Rails Tutorial: sample application

This is the sample application for the
[*Ruby on Rails Tutorial:
Learn Web Development with Rails*](http://www.railstutorial.org/)
by [Michael Hartl](http://www.michaelhartl.com/).
```

마지막으로, 변경 내용을 커밋한다.

```
$ git commit -am "Improve the README"
```

섹션 1.4.4에서 사용한 “모든 변경” (`-a`)과 메세지(`-m`) 플래그를 포함한 Git 명령인 `git commit -a -m "Message"`를 기억 속에 떠 올릴 수 있다. 위의 두번째 명령과 같이, Git 에서는 두 개의 플래그를 붙여서 이와 같이 `git commit -am "Message"` 하나로 사용할 수 있다.

앞으로 이 sample 애플리케이션을 지속적으로 사용할 것이므로, Bitbucket에 새로운 저장소를 만들어 올리는게 좋다.

```
$ git remote add origin git@bitbucket.org:<username>/sample_app.git
$ git push -u origin --all # 처음으로 소스 저장소에 올린다
```

나중에 서비스 통합에 골머리 아프지 않기 위해서는, 초기 단계부터 허로쿠에 응용 프로그램을 배포하는 것이 좋다.1장과 2장의 목록 1.8과 목록 1.9 단계에서 “hello, world!” 예제를 커밋하고 허로쿠에 올리는 예제를 추천한다.[4]

```
$ git commit -am "Add hello"
$ heroku create
$ git push heroku master
```

(섹션 1.5 처럼, 지금은 무시해도 될 몇가지 경고를 볼 수 있을 것이다. 섹션 7.5에서 제거할 것이다.) 그림 1.18처럼, 원격 허로쿠 앱 주소에 접속해서 같은 결과를 볼 수 있다.

책의 남은 부분을 진행하면서, 자동으로 원격 백업을 하고, 가능한 신속하게 운영 환경 에러를 인지하기 위해, 주기적으로 배포할 것을 추천한다. 만약 허로쿠에 문제가 생기면, 운영환경 로그를 검색하여 문제의 원인을 찾아야 한다.

```
$ heroku logs
```

노트: 실제 운영 목적으로 애플리케이션을 허로쿠로 배포하여 사용하기 위해서는, 섹션 7.5의 운영환경 웹서버 설정을 따라하기 바란다.

3.2 정적 페이지

섹션 3.1을 마치면, sample 애플리케이션 개발을 위한 준비가 끝난 것이다. 이번 섹션에서는, 일련의 레일스 *액션*과 정적 HTML만을 포함하는 *뷰*를 만들어, 동적 페이지 만들기 위한 첫 단계를 수행한다.[5] 만들게 될 레일스 액션은 *컨트롤러* 안에서 다른 범용의 액션들과 함께 위치하게 된다(섹션 1.3.3의 MVC 에서 C를 의미). 2장에서 컨트롤러를 잠깐 다루었고, 앞으로 REST 설계를 다루면서 더 쉽게 이해할 것이다(6장에서 시작). 바른 이해를 위해, 섹션 1.3(그림 1.4)의 레일스 디렉토리 구조를 기억해라. 이번 섹션에서는 주로 `app/controllers` 와 `app/views` 디렉토리에서 작업한다.

섹션 1.4.4를 기억 속에 떠 올리면, Git 을 사용할때 master 브랜치에서 분리된 별도의 브랜치에서 작업하는것은 좋은 습관이다. Git 버전 관리를 위해 , 아래 명령어를 실행하여 정적 페이지용 브랜치를 생성한 후 체크아웃한다.

```
$ git checkout master
$ git checkout -b static-pages
```

(첫째 코드라인은 master 브랜치로 변경한 후, `static-pages` 관련 브랜치가 `master` 브랜치를 기반으로 한다는 것을 명확하게 해 준다. 이미 master 브랜치에 있다면 이 단계는 무시해도 된다.)

3.2.1정적 페이지 생성

정적 페이지를 시작하려면, 우선 2장의 레일스 `generate` 스크립트를 이용해서 스캐폴딩을 만든다. 정적 페이지를 처리할 컨트롤러를 만들어야 하기 때문에, CamelCase 표기법으로 `StaticPages` 컨트롤러를 만든다. 이 컨트롤러에는 Home 페이지 Help 페이지 About 페이지를 위한 소문자로 `home` , `help` , `about` 액션도 만들 계획이다.

`generate` 스크립트는 액션 목록을 옵션으로 받기 때문에, 명령행에 Home과 Help 페이지를 위한 액션 이름을 추가한다. About 페이지용 액션은 나중에 추가 할 것이다(섹션 3.3). Static Pages 컨트롤러 생성 결과를 목록 3.4에서 확인한다.

목록 3.4: 정적 페이지 컨트롤러 생성.

```
$ rails generate controller StaticPages home help
      create  app/controllers/static_pages_controller.rb
       route  get 'static_pages/help'
       route  get 'static_pages/home'
      invoke  erb
      create    app/views/static_pages
      create    app/views/static_pages/home.html.erb
      create    app/views/static_pages/help.html.erb
      invoke  test_unit
      create    test/controllers/static_pages_controller_test.rb
      invoke  helper
      create    app/helpers/static_pages_helper.rb
      invoke    test_unit
      create      test/helpers/static_pages_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/static_pages.js.coffee
      invoke    scss
      create      app/assets/stylesheets/static_pages.css.scss
```

한편, `rails g` 는 `rails generate` 의 단축형으로 레일스가 지원하는 것 중 하나다(표 3.1). 명확하게 하기위해, 이 튜토리얼에서는 항상 전체 명령어 이름을 사용하지만, 실제 레일스 개발자들은 표 3.1의 단축 명령을 사용한다.

전체 명령	단축명령
`$ rails server`	`$ rails s`
`$ rails console`	`$ rails c`
`$ rails generate`	`$ rails g`
`$ bundle install`	`$ bundle`
`$ rake test`	`$ rake`

표 3.1: 레일스 단축 명령 일부.

Git로 버전관리 중이면, 더 진행하기 전에, Static Pages 컨트롤러 관련 파일을 원격 저장소에 저장하는게 좋다.

```
$ git status
$ git add -A
$ git commit -m "Add a Static Pages controller"
$ git push -u origin static-pages
```

위에서 실행한 마지막 명령은 `static-pages` 브랜치를 Bitbucket에 올리는 것이다. 이후에 푸시를 할 때는 간편하게 다른 인자를 생략할 수 있다.

```
$ git push
```

위에 커밋과 푸시 순서는 저자가 실제 개발에서 일반적으로 따르는 패턴이지만, 이제부터는 중간 커밋은 간단하게 보이도록 생략할 것이다.

목록 3.4에서, CamelCase 표기로 지정한 컨트롤러 이름은, snake case 이름의 파일을 생성한다. 그래서 StaticPages 컨트롤러 이름으로 `static_pages_controller.rb` 라는 파일이 생성된다. 이건 단지 관습이라 snake case 형태로 명령행에 입력해도 동작한다. 이 명령도

```
$ rails generate controller static_pages ...
```

`static_pages_controller.rb` 라는 컨트롤러를 생성한다. 루비는 클래스 이름에 CamelCase 규칙을 사용한다(섹션 4.4). 따라서 저자도 이를 사용하지만, 단지 취향의 문제다. (루비 파일 이름은 대체로 snake case **("sanke case 표기를" 가 더 나을 것 같습니다.)**를 사용하므로, 레일스 제네레이터는 underscore 메서드를 이용해 CamelCase 를 snake case 로 변경한다.)

한편, 실수에 대비해 코드 생성 과정을 되돌리는 방법을 알아두면 좋다.**("코드 생성 실수에 대비해 생성 과정을 되돌리는 ~" 가 읽기에 더 좋은 것 같습니다.)** 글상자 3.1을 보면 레일스에서 코드 생성을 되돌리는 몇가지 기법들을 참고할 수 있다 .

글상자 3.1. ## 되돌리는 방법**("명령을 취소하는 방법" 어떨까요?)**

레일스 애플리케이션 개발 중 매우 신중하게 진행해도 때로는 실수를 저지를 수 있다. 다행이도, 레일스는 되돌리는 몇가지 방법을 제공한다.

일반적인 시나리오는 생성된 코드를 되돌리는 작업이다. 예컨데 컨트롤러 이름에 대한 생각이 바뀌었을 때이다. 레일스는 컨트롤러와 상당수의 관련 보조 파일들을 생성하기 때문에(목록 3.4), 컨트롤러 파일을 제거하는 것은 쉽지 않다. 생성물을 되돌린다는 의미는 핵심 파일뿐 아니라 모든 관련 파일을 제거한다는 의미이다. (사실, 섹션 2.2 과 섹션 2.3에서, rails generate는 자동으로 routes.rb 파일을 수정하기 때문에, 이것도 자동으로 제거하기를 원한다.) 레일스에서는 rails destroy 명령으로 생성된 요소들을 일괄 삭제할 수 있다. 특히, 이 두 명령은 서로의 작업을 취소한다.

```
  $ rails generate controller StaticPages home help
  $ rails destroy  controller StaticPages home help
```

마찬가지로, 6장에서 아래와 같이 *모델*을 만들 것이다.

```
  $ rails generate model User name:string email:string
```

다음 명령으로 취소할수 있다.

```
$ rails destroy model User
```

(이 경우에는, 다른 커맨드라인 옵션을 생략할수 있다. 6장에서, 왜 가능한지 알게된다.)

또 다른 기법은 모델과 관계된 *마이그레이션* 작업을 취소하는 것인데, 간단히 2장에서 보았고 6장 시작 부분에서 더 많이 볼 수 있다. 마이그레이션은 데이터베이스 상태를 변경한다.

```
$ bundle exec rake db:migrate
```

하나의 마이그레이션 단계는 이렇게 되돌릴 수 있다.

```
$ bundle exec rake db:rollback
```

처음으로 돌아가려면, 이렇게 하면된다.

```
$ bundle exec rake db:migrate VERSION=0
```

추측할수 있듯이, 위의 0 대신에 다른 번호로 바꾸면, 해당 버전까지 순서대로 마이그레이션 작업을 하라는 의미한다.

이 기법을 이용해서, 개발시 피할수 없는 혼란 상황을 복구할수 있다.

목록 3.4에서, 제네레이터를 사용해 생성한 Static Pages 컨트롤러는 자동으로 라우트 (`config/routes.rb`) 파일을 갱신하는데, 섹션 1.3.4에서 간단히 살펴 보았다. 라우트 파일은 URL과 웹페이지 사이의 연결을 정의한다 (그림 2.11 참고). 라우트 파일은 레일스에 관한 설정을 모아둔 `config` 디렉토리에 위치한다(그림 3.1).

그림 3.1: sample 애플리케이션의 `config` 디렉토리.

목록 3.4의 `home` 과 `help` 액션을 포함하는 라우트 파일은 목록 3.5에서 볼수 있다.

목록 3.5:

Static Pages 컨트롤러의 `home` 과 `help` 액션을 위한 라우트. config/routes.rb

```
Rails.application.routes.draw do
  get 'static_pages/home'
  get 'static_pages/help'
  .
  .
  .
end
```

아래의 규칙은,

```
get 'static_pages/home'
```

/staticpages/home URL을 Static Pages 컨트롤러의 `home` 액션에 연결한다(매핑한다. 역주). 또한, `get` 메소드를 사용해서, 하이퍼텍스트 전송 규약이 지원하는 기본 *HTTP verbs*인 GET 요청에만 응답하도록 제한하였다(글상자 3.2). 이 경우에, StaticPages 컨트롤러 내부에 만든 `home` 액션은 자동으로 /staticpages/home 주소를 가진다. 레일스 개발 서버를 실행한 결과를 섹션 1.3.2에서 확인해라.:

```
$ rails server -b $IP -p $PORT    # 로컬에서 서버를 띄운다면 'rails server'만 입력한다
```

서버를 실행한 후 웹브라우저 주소에 /static_pages/home 입력하고 이동한다.(그림 3.2)

그림 3.2: 자동으로 생성된 상태의 home 뷰 페이지(/static_pages/home).
글상자 3.2. GET과 그 밖의 관련 HTTP 메소드

하이퍼텍스트 전송 규약(HTTP)은 기본 규칙으로 GET, POST, PATCH, DELETE을 정의한다. 이것은 *클라이언트* 컴퓨터(보통 Firefox 나 Safari 같은 웹브라우저)와 *서버* (보통 Apache 나 Nginx 같은 웹서버) 사이의 통신 관련 규칙이다. (로컬 컴퓨터에서 레일스 애플리케이션을 개발할 때는 물리적으로 같은 기계를 사용하지만, 일반적으로 서비스를 운영할 때는 각각 분리된 상태에서 원격으로 접속된다는 것을 이해하는 것이 중요하다.) HTTP verbs는 *REST 설계 하*에 만들어진 전형적인 (레일스를 포함해서) 웹 프레임워크에서 매우 중요한 역할을 한다. 이에 대해서는 2장에서 간단히 둘러보았고, 7장에서 좀 더 자세히 배울것이다.

GET는 웹 상의 데이터를 *읽을 때 사*용하는 가장 평범한 HTTP 동작을 말한다. 이는 단지 “웹페이지 불러오기”를 의미하고, http://www.google.com/ 이나 http://www.wikipedia.org/ 같은 사이트를 방문할 때 브라우저가 매번 GET 요청을 한다. POST는 다음으로 중요한 동작이다. 이는 브라우저에서 폼 데이터 정보를 전달할 때 사용한다. 레일스 애플리케이션은 보통 무언가를 *생성*할 때 POST 요청을 사용한다(비록 HTTP가 데이터 갱신 작업시에도 POST를 사용하기도 하지만). 예를들어, 등록 폼을 POST로 요청하면 원격 사이트에 새로운 사용자가 만들어진다. 또 다른 두가지 HTTP verbs인 PATCH와 DELETE는 원격 서버에서 데이터 *갱신* 및 *삭제*를 위해 설계되었다. 그러나, 아직 몇몇 브라우저에서 제대로 지원하지 못하기 때문에, 이들 요청 방식은 GET과 POST보다 덜 일반적으로 사용된다. 그러나 몇몇 웹 프레임웍(루비 온 레일스 포함)은 브라우저가 보내는 이런 요청을 *자연스럽게 처리하는 것 처럼 보이게 하는 똑똑한 방법을 가지고 있다. 결과적으로, 레일스는 네 가지 요청 종류 GET, POST, PATCH, DELETE 모두 지원한다.*

이 페이지가 어떻게 만들어 졌는지를 이해하기 위해, 텍스트 에디터에서 StaticPages 컨트롤러 코드를 자세히 보자 . 목록 3.6 같이 보여야 한다. 2장의 데모 Users 와 Microposts 컨트롤러와 다른 점을 주목하자면, StaticPages 컨트롤러는 표준 REST 액션을 사용하지 않는다. 단지 정적 페이지의 모음이다. REST 구조가 모든 문제에 최적의 해결책은 아니다.

목록 3.6: 목록 3.4 명령으로 만들어진 StaticPages 컨트롤러app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end
end
```

목록 3.6에서 `class` 키워드를 볼 수 있는데, 이 경우에 `static_pages_controller.rb` 는 `StaticPagesController` 라는 *클래스*를 정의한다. 클래스는 `def` 키워드로 정의한 `home` 과 `help` 액션 같은 *함수*(*메서드*라고도 불림)을 편하게 구성하는 방법을 제공한다. 섹션 2.3.4에서 언급한 바와 같이, 왼쪽 부등호 `<` 는 `StaticPagesController` 가 레일스 클래스 `ApplicationController` 를 *상속*하는 것을 의미한다. 곧 알게 되겠지만, 이 말은 많은 레일스 전용 기능을 페이지에세도 사용할 수 있다는 것을 의미한다. (클래스와 상속에 관해서는 섹션 4.4.에서 더 자세히 알아볼 것이다.)

이 경우 StaticPages 컨트롤러의 메서드는 비어있다.

```
def home
end

def help
end
```

일반적인 루비에서 이 메서드는 아무것도 하지 않는다. 하지만, 레일스에서는 상황 좀 다르다. `StaticPagesController` 는 루비 클래스 이지만 `ApplicationController` 를 상속 받아서 레일스에 특화된 메서드 동작을 하기 때문이다. /static_pages/home URL을 방문하면 레일스는 StaticPages 컨트롤러를 찾아 `home` 액션 내의 코드를 실행한다. 다음 동작으로 액션에 부합하는 *뷰*(섹션 1.3.3 MVC에서 V에 해당)를 출력한다. `home` 액션이 비어 있기 때문에 이번에는 /static_pages/home 방문시 곧 바로 뷰 출력으로 이어진다. 그렇다면, 뷰에는 어떤 내용이 출력되고, 어디에서 찾아 볼 수 있을까?

목록 3.4에서 제네레이터가 만든 다른 결과를 보면, 액션과 뷰 사이에 어떤 대응 관계를 유추할수 있다. 예를 들면, `home` 액션은 `home.html.erb` 로 명명된 뷰와 대응한다. `.erb` 의미는 섹션 3.4에서 배우게 될것이다. `.html` 부분은 아마 기본적으로 HTML (목록 3.7) 처럼 보여서 그리 놀랄만한 건 아닐 것이다.

목록 3.7: 생성된 Home 뷰 페이지. app/views/static_pages/home.html.erb

```
<h1>StaticPages#home</h1>
<p>Find me in app/views/static_pages/home.html.erb</p>
```

`help` 액션에 대한 뷰도 비슷하다(목록 3.8).

목록 3.8: 생성된 Help 뷰 페이지. app/views/static_pages/help.html.erb

```
<h1>StaticPages#help</h1>
<p>Find me in app/views/static_pages/help.html.erb</p>
```

이 두 가지 뷰 모두 단지 견본에 불과하다. 이 파일은 (`h1` 태그 내에) 최상위 머릿말과 파일의 상대 경로 정보를 포함하는 문단(`p` 태그)을 기본값으로 가진다.

3.2.2 정적 페이지 바꾸기

섹션 3.7에서 아주 적은 양의 동적 내용을 추가할 것이다. 하지만, 지금은 목록 3.7과 목록 3.8에서 이 뷰의 중요한 부분에만 주목하자. 레일스 뷰는 단순히 정적 HTML만을 포함할 수 있다. 이는 레일스 지식이 없어도 목록 3.9와 목록 3.10과 같이 Home과 Help 페이지를 바꿀수 있는 것을 의미한다.

목록 3.9: Home 페이지를 위한 사용자 정의 HTML. app/views/static_pages/home.html.erb

```
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

목록 3.10: Help 페이지를 위한 사용자 정의 HTML. app/views/static_pages/help.html.erb

```
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="http://www.railstutorial.org/#help">Rails Tutorial help section</a>.
  To get help on this sample app, see the
  <a href="http://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
  book</a>.
</p>
```

목록 3.9 과 목록 3.10 의 결과를 그림 3.3 그림 3.4에서 볼수있다.

그림 3.3: 사용자 정의 Home 페이지.

그림 3.4: 사용자 정의 Help 페이지.

3.3 테스트 시작하기

sample 애플리케이션을 위해 Home 과 Help 페이지를 만들었고(섹션 3.2.2), 이제 About 페이지도 추가할 것이다. 이와 같은 기능 변경을 하기 위해서는, *자동화된 테스트*로 요구 사항이 정확히 구현되었는지 검증하는 것이 좋다. 애플리케이션을 만드는 과정에서 *일련의 테스트코드를 작성하는 것은* 애플리케이션 소스 코드의 안정망으로써 그리고 실행 가능한 문서

로써의 역할을 제공한다. 제대로만 작성한다면, 테스트 코드를 작성하는 것은 코드 작업량이 늘어 남에도 불구하고 개발을 더 *빠르게* 만든다. 왜냐하면, 버그를 추적하는 시간을 줄여주기 때문이다. 이것은 테스트 작성에 익숙해 질 경우에만 가능하기 때문에 되도록 빨리 시작해야 할 이유다.

사실상 모든 레일스 개발자가 테스팅이 좋은 생각이라고 동의하지만, 세부 사항에 대해서는 다양한 의견이 있다. 실패한 테스트를 먼저 작성하고, 테스트를 통과하는 애플리케이션 코드를 작성하는 방식으로 진행하는, 테스트 주도 개발(TDD)에는 특히 활발한 토론이 있다.[6] *루비온레일스 튜토리얼*은 TDD 이론을 완벽하게 따르지는 않지만, 유용하다고 판단할 때는 과감하게 TDD를 채택하여 테스트를 가볍고 직관적인 방식으로 접근한다.(글상자 3.3)

글상자 3.3. 테스트를 해야할 때

언제, 어떻게 테스트를 해야할지 결정하려면, *왜* 테스트를 해야 하는지에 대해 생각하면 도움이 된다. 저자의 관점에서, 테스트를 자동화하여 얻을 수 있는 이점 세가지는 다음과 같다.

1. 테스트는 어떤 기능이 알수 없는 이유로 중단되는 *회귀(regressions)* 현상을 막는다.
2. 테스트는 코드 *리팩터*(기능 변화 없는 코드 변경)를 통해서 더 큰 신뢰감을 준다.
3. 테스트는 애플리케이션 코드에 대한 *클라이언트*로 동작하여, 코드 디자인을 결정하고 시스템의 다른 부분과의 인터페이스 결정을 도와 준다.

비록 테스트를 처음 작성할때 위와 같은 이유(장점)가 *필요*하지는 않지만, 많은 상황에서 테스트 주도 개발(TDD)가 가치있는 도구로써의 역할을 한다. 언제, 어떻게 테스트 할지는 얼마나 테스트 작성에 익숙한 가에 달려 있다. 즉, 많은 개발자들은 자신이 테스트 코드 작성에 익숙해 지면서 테스트를 먼저 작성하고 싶어 한다는 것을 알게 된다. 또한 애플리케이션 코드에 비해 상대적으로 테스트 코드가 얼마나 어려운지, 요구 사항을 얼마나 정확히 파악하고 있는지, 미래에 얼마나 깨지기 쉬운지에 따라 테스트 작성 시점과 방법이 달라진다.

이런 맥락에서, 언제 테스트를 먼저(혹은 전부) 해야 할지에 대한 가이드라인을 정해 두는 것이 도움이 된다. 여기에서 저자의 경험을 기반으로 한 몇가지 사항을 제안한다.

- 테스트가 애플리케이션 코드에 비해 매우 짧거나 간단하면, 테스트를 먼저 작성한다.
- 요구되는 동작이 분명하지 않으면, 애플리케이션 코드를 먼저 작성하고 결과에 맞춰 테스트를 작성한다.
- 보안은 최우선 사항이므로, 보안 모델의 테스트 작성 관점에서 에러가 발생하도록 한다.
- 버그 발견할때마다, 애플리케이션 코드를 고치는 것보다, 회귀 방지를 위해 에러를 발생하는 테스트 작성을 우선한다.
- 미래에 자주 변경될 코드 (가령 구체적인 HTML 구조)에 대한 테스트 작성은 피한다.
- 오류가 발생하기 쉬운 코드 테스트에 집중해서, 리팩터링 전에 테스트를 작성한다.

실제로, 위의 가이드 라인은 일반적으로 컨트롤러와 모델 테스트를 먼저 작성하고 (모델, 뷰와 컨트롤러 사이의 기능을 테스트하는) 통합 테스트를 나중에 작성하는 것을 의미한다. 그리고 에러가 발생할 확율이 적은 코드 혹은 변경이 잦은 (뷰 관련 코드) 애플리케이션 코드를 작성할때 종종 테스팅을 완전히 건너 뛸수 있다.

주요 테스트 도구는 *컨트롤러 테스트* (이번 섹션에서 시작), *모델 테스트* (6장에서 시작), 그리고 *통합 테스트* (7장에서 시작)다. 웹 브라우저를 이용한 애플리케이션과의 상호작용을 통해서 사용자의 액션을 재현하여 진행하는 통합 테스트는 특히 강력하다. 결국, 통합 테스트가 기본 테스트 기법이지만, 시작하기에는 컨트롤러 테스트가 용이하다.

3.3.1 첫번째 테스트

이제 애플리케이션에 About 페이지를 추가할 것이다. 보는 바와 같이, 테스트는 매우 짧고 간단해서, 글상자 3.3 가이드 라인을 따라 테스트를 먼저 작성한다. 그 뒤에 실패한 테스트를 이용해서 애플리케이션 코드를 작성할 것이다.

테스트에 도전하기 위해서는 레일스와 루비에 광범위한 지식이 필요하다. 그래서 초기 단계에서, 테스트 작성은 절망적으로 보이기도 하다. 다행히, 이미 가장 어려운 부분은 해결해 놓았다. `rails generate controller` 에서 (목록 3.4) 테스트 시작할 파일을 자동으로 생성해두었기 때문이다.

```
$ ls test/controllers/
static_pages_controller_test.rb
```

이제 아래의 코드를 살펴보자.(목록 3.11).

목록 3.11: StaticPages 컨트롤러의 기본 테스트.**green** test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  test "should get home" do
    get :home
    assert_response :success
  end

  test "should get help" do
    get :help
    assert_response :success
  end
end
```

목록 3.11의 문법을 자세하게 이해하는 것은 중요하지 않지만, 두가지 테스트가 있다는 것을 알 수 있고, 각각은 목록 3.4의 두 컨트롤러 액션에 대한 것이다. 각 테스트는 간단히 액션을 불러와 그 결과가 성공이라는 것을 (가정가설문(假定設定文, *assertion*)을 통해) 검증한다. 여기서 get을 사용한 것은 Home 과 Help 페이지를 일반 웹페이지로 기대하고 GET 요청한다는 것을 알려 준다(글상자 3.2). `:success` 응답은 기초 HTTP 상태 코드의 추상적인 표현이다. (이 경우, 200 OK). 다음 테스트는

```
test "should get home" do
  get :home
  assert_response :success
end
```

" `home` 액션에 GET 요청을 하고 응답으로 '성공(success)' 상태값을 받아서 Home 페이지를 테스트를 하라"를 의미한다.

테스트 주기(cycle)를 시작하려면, 현재 테스트를 통과하는지 확인하기위해 일련의 테스트 코드를 실행해야한다. 다음과 같이 `rake` 유틸리티(글상자 2.1)를 사용한다.[7]

목록 3.12: **green**

```
$ bundle exec rake test
2 tests, 2 assertions, 0 failures, 0 errors, 0 skips
```

요청한 바대로, 처음에 2개의 테스트 코드는 통과한다(**green**)(선택 사항이긴 하지만 섹션 3.7.1의 MiniTest 리포터를 추가하지 않으면, 실제 녹색은 볼 수 없다). 그런데, 테스트 시작시에 약간의 시간지체가 발생하는데 여기에는 두가지 요인이 있다. (1) 레일스 환경을 미리 로딩하기 위해 *Spring 서버*를 맨 처음 실행시 (2) 루비 엔진 시동과 관련한 오버헤드. (두번째 요인은 섹션 3.7.3에 소개된 Guard로 개선할수 있다.)

3.3.2 Red

글상자 3.3에서 언급한 바와 같이, 테스트 주도 개발은 먼저 실패하는 테스트를 작성한 후, 이어서 테스트를 통과하는 애플리케이션 코드를 작성하고, 필요하면 코드를 리펙터링한다. 많은 테스트 도구가 빨간 색은 실패를, 녹색은 테스트 성공을 의미하기 때문에, 이 순서는 때때로 “Red, Green, Refactor” 주기라고 부른다. 이번 섹션에서는, 이 주기의 첫단계로, 실패 테스트를 작성해서 빨간 색를 얻을 것이다. 그리고 나서 섹션 3.3.3에서 녹색을, 섹션 3.4.3에서 리펙터링을 하게 될 것이다. [8]

첫번째 단계는 About 페이지에 대해서 실패 테스트를 작성하는 것이다. 목록 3.11를 참고해서, 목록 3.13처럼 통과할 것으로 기대하는 테스트 코드를 작성한다.

목록 3.13: About 페이지에 대한 테스트. **red** test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  test "should get home" do
    get :home
    assert_response :success
  end

  test "should get help" do
    get :help
    assert_response :success
  end

  test "should get about" do
    get :about
    assert_response :success
  end
end
```

목록 3.13의 강조된 지점에서, Home 과 Help 처럼 “home” 이나 “help” 단어 위치에 “about” 을 넣은 About 페이지용 테스트를 볼 수 있다.

짐작대로, 초기 테스트는 실패다.

목록 3.14: **red**

```
$ bundle exec rake test
3 tests, 2 assertions, 0 failures, 1 errors, 0 skips
```

3.3.3 Green

이제 테스트가 실패(**red**)하기 때문에, 테스트 실패에 따른 에러 메세지를 가이드 삼아 통과 테스트(**green**) 코드를 만들면서, About 페이지를 구현할 것이다.

테스트 결과, 실패를 발생시킨 에러 메세지 확인부터 시작한다.[9]

목록 3.15: **red**

```
$ bundle exec rake test
ActionController::UrlGenerationError:
No route matches {:action=>"about", :controller=>"static_pages"}
```

이 오류 메세지는 액션/컨트롤러 조합과 일치하는 라우트 정의가 없다는 의미로, 라우트 파일에 규칙 한 줄을 추가가 필요가 있다는 것을 뜻한다. 목록 3.5 패턴을 따라서 목록 3.16과 같이 완성해 보자.

목록 3.16: `about` 라우트 추가하기. **red** config/routes.rb

```
Rails.application.routes.draw do
  get 'static_pages/home'
  get 'static_pages/help'
  get 'static_pages/about'
  .
  .
  .
end
```

목록 3.16의 강조 부분은 레일스에게 /static_pages/about URL로 GET 요청이 오면 Static Pages 컨트롤의 `about` 액션으로 라우트하라고 명령한다.

다시 일련의 테스트 코드를 실행하면 여전히 **red** 상태이지만, 이번에는 에러 메세지가 바뀌었다.

목록 3.17: **red**

```
$ bundle exec rake test
AbstractController::ActionNotFound:
The action 'about' could not be found for StaticPagesController
```

오류 메세지는 Static Pages 컨트롤러에 `about` 액션이 없다는 것을 나타내며, 목록 3.6에서 `home` 과 `help` 과

비슷하게, 목록 3.18처럼 추가한다.

목록 3.18: Static Pages 컨트롤러에 `about` 액션 추가. **red** app/controllers/static_pages_controller.rb

```
class StaticPagesController < ApplicationController

  def home
  end

  def help
  end

  def about
  end
end
```

이전처럼, 테스트 결과는 **red**지만, 에러 메세지는 다시 변경되었다.

```
$ bundle exec rake test
ActionView::MissingTemplate: Missing template static_pages/about
```

이 메세지는 템플릿이 없다는 것을 의미하며, 레일스의 맥락에서 볼 때 뷰 파일이 없다는 의미와 같다. 섹션 3.2.1에서 설명했듯이, `home` 액션은 `app/views/static_pages` 디렉토리에 있는 `home.html.erb` 뷰 파일과 연관이 있는데, 이는 같은 디렉토리에 `about.html.erb` 파일을 만들어야 하는 것을 의미한다.

파일 생성 방법은 시스템 설정에 따라 차이가 있지만, 대부분의 텍스트 에디터에서는 파일을 생성하기 원하는 디렉토리 내부에서 컨트롤 키를 누를 상태에서 마우스를 클릭할 때 나타나는 메뉴 중 “새파일(New File)” 메뉴를 선택할 수 있도록 해준다. 또는, 먼저 새파일을 생성한 후 저장할 때 정확한 디렉토리를 지정할 수 있다. 마지막으로, 저자가 좋아하는 묘수는 유닉스 touch 명령어를 사용하는 것이다.

```
$ touch app/views/static_pages/about.html.erb
```

`touch` 명령은 파일이나 디렉토리의 타임 스템프만 수정하도록 만들어 졌지만, 해당 파일이 없으면 빈 파일을 새로 만든다. (만약 클라우드 IDE를 이용중이라면, 섹션 1.3.1 설명대로 파일 트리를 새로 고쳐 보기 해야만 한다.)

해당 디렉토리에 `about.html.erb` 파일을 만든 후, 목록 3.19 와 같이 코드를 작성한다.

목록 3.19: About 페이지 코드. **green** app/views/static_pages/about.html.erb

```
<h1>About</h1>
<p>
  The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a> is a
  <a href="http://www.railstutorial.org/book">book</a> and
  <a href="http://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="http://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample application for the tutorial.
</p>
```

이 시점에서, `rake test` 명령을 실행해서 green 상태를 확인한다. **green**

목록 3.20: **green**

```
$ bundle exec rake test
3 tests, 3 assertions, 0 failures, 0 errors, 0 skips
```

물론, 테스트가 문제없는지 브라우저로 페이지를 열어서 눈으로 확인하는건 좋은 생각이다. (그림 3.5).

그림 3.5: 새로운 About 페이지 (/static_pages/about).

3.3.4 리팩터(Refactor)

이제 **green** 상태는, 코드를 자유롭게 리펙터링할 수 있도록 확신을 준다. 애플리케이션 개발 중에, 종종 코드에서 "냄새(smell)"가 나기 시작하는데, 이는 코드가 복잡하고, 비대해지고 반복코드로 채워지는 것을 의미한다. 물론, 컴퓨터는 코드의 모양을 상관하지 않지만, 인간에게는 중요한 부분이다. 그래서 자주 리팩터링 작업으로 코드를 깨끗하게 유지하는 것은 중요하다. sample 애플리케이션이 너무 작아서 지금 당장 리펙터링하기에는 무리지만, 코드 냄새는 모든 균열에 스며들게 마련이라 섹션 3.4.3에서 리펙터링을 시작할 것이다.

3.4 약간의 동적인 내용이 포함된 페이지

지금까지 정적 페이지에 대한 뷰와 액션을 몇개 만들었고, 이번에는 각 페이지에 *아주 적은 양의* 동적인 부분을 추가할 것이다. 즉, 각 페이지의 타이틀을 동적으로 만드는 작업이다. 페이지 타이틀 표시를 *진정한 의미의* 동적인 내용으로 보기에는 논쟁의 여지가 있지만, 이는 7장에서 추가할 제대로 된 동적인 내용물을 만들기 위한 기반이 된다.

이번 장에서는 Home, Help와 About 각 페이지별로 타이틀이 다르게 나오도록 코드를 수정할 계획이다. 이를 위해서는 각 페이지 뷰에서 `<title>` 태그를 사용해야 한다. 대부분의 브라우저는 title 태그 내용을 브라우저 윈도우의 맨위에 보여준다. 그리고 이는 검색엔진 최적화에도 중요하다. "Red, Green, Refactor" 주기를 사용할 것이다. 먼저 페이지 타이틀 관련 테스트를 추가하고(**red**), 세 개 페이지 각각에 타이틀을 추가한 다음 (**green**), 마지막으로 *레이아웃* 파일을 사용해서 중복되는 코드을 제거할 것이다(Refactor). 이 섹션 끝 무렵에 다다르면, 세 개의 정적 페이지는 "<페이지 이름> | Ruby on Rails Tutorial Sample App"과 같은 형식의 타이틀을 각 페이지 정보에 맞게 갱신한 후 페이지 첫 부분에 표시되도록 할

것이다. (표 3.2).

`rails new` 명령(목록 3.1)은 자동으로 기본 레이아웃 파일을 생성하지만, 처음에는 교육적인 측면에서 이름을 변경해서 이를 무시하게 할수 있다.

```
$ mv app/views/layouts/application.html.erb layout_file   # 파일 이름을 임시로 바꾼다
```

실제 애플리케이션에서 이와 같이 하지 않지만, 이렇게 비활성화 시키면 레이아웃 파일의 목적을 쉽게 이해할수 있다.

페이지명	URL	기본 타이틀	title 변수값
Home	/static_pages/home	`"Ruby on Rails Tutorial Sample App"`	`"Home"`
Help	/static_pages/help	`"Ruby on Rails Tutorial Sample App"`	`"Help"`
About	/static_pages/about	`"Ruby on Rails Tutorial Sample App"`	`"About"`

표 3.2: sample 애플리케이션의 (거의) 정적인 페이지들.

3.4.1 타이틀 테스트(Red)

페이지 타이틀을 추가를 위해, 목록 3.21과 같은 일반적인 웹 페이지 구조를 알아 둘(혹은 리뷰할) 필요가 있다.

목록 3.21: 전형적인 웹 페이지의 HTML 구조.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Greeting</title>
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

목록 3.21에서 보는 구조는 *문서유형* 혹은 doctype이라 불리는 정보를 포함하는데 이는 처음에 선언되어 브라우저에게 사용중인 HTML의 버전을 알려준다(이 경우에 HTML5).[10] 즉, `title` 태그 내에 “Greeting”라는 내용을 가진 `head` 섹션, `p 태그` 내에 “Hello World!” 내용을 가진 `body` 섹션으로 구성되어 있다. (HTML은 여백에 민감하지 않고, 탭과 공백문자는 무시하기 때문에 코드 들여쓰기는 선택 사항이지만 문서 구조를 더 보기 좋게 해준다.)

표 3.2에 각 타이틀별로 간단한 테스트를 작성할 것이며, 이 때 목록 3.13의 `assert_select` 메소드를 사용한다. 이 메소드는 특정 HTML 태그(때때로 “셀렉터(selector)”라고 부르기도 한다.)의 존재 여부를 테스트할 수 있도록 해 준다.[11]

```
assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
```

특히, 위에 코드는 `<title>` 태그가 포함하는 “Home | Ruby on Rails Tutorial Sample App” 문자열을 확인하는 코드이다. 목록 3.22처럼, 세가지 정적 페이지에 모두 이 아이디어를 적용한다.

목록 3.22: 타이틀 테스트가 포함된 Static Pages 컨트롤러 테스트. **red**
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Home | Ruby on Rails Tutorial Sample App"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | Ruby on Rails Tutorial Sample App"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | Ruby on Rails Tutorial Sample App"
  end
end
```

(기본 타이틀 “Ruby on Rails Tutorial Sample App”을 반복하는 것이 성가시면, 섹션 3.6의 연습문제를 참고한다.)

목록 3.22 테스트 코드를 확인하면 아직 **red**상태이다.

목록 3.23: **red**

```
$ bundle exec rake test
3 tests, 6 assertions, 3 failures, 0 errors, 0 skips
```

3.4.2 페이지 타이틀 추가 (Green)

이제 섹션 3.4.1에서 각 페이지를 추가하고 테스트를 통과시킬 것이다. 목록 3.21의 기본 HTML 구조를 사용자 정의 Home 페이지에 적용해서 목록 3.9를 목록 3.24로 만든다.

목록 3.24: Home 페이지 전체 HTML 구조보기. **red** app/views/static_pages/home.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Home | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
      This is the home page for the
      <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

해당 웹페이지가 그림 3.6처럼 보이게 된다.[1][2]

그림 3.6: 타이틀이 있는 Home 페이지.

Help 페이지(목록 3.10)와 About 페이지(목록 3.19)를 기반으로 목록 3.25와 목록 3.26와 같이 코드를 변경한다.

목록 3.25: Help 페이지 전체 HTML 구조. **red** app/views/static_pages/help.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>Help | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="http://www.railstutorial.org/#help">Rails Tutorial help
      section</a>.
      To get help on this sample app, see the
      <a href="http://www.railstutorial.org/book"><em>Ruby on Rails
      Tutorial</em> book</a>.
    </p>
  </body>
</html>
```

목록 3.26: About 페이지 전체 HTML 구조. **green** app/views/static_pages/about.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title>About | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a> is a
      <a href="http://www.railstutorial.org/book">book</a> and
      <a href="http://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="http://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample application for the tutorial.
    </p>
  </body>
</html>
```

이 시점에서, 테스트 코드는 다시 **green**으로 돌아온다.

목록 3.27: **green**

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

3.4.3 레이아웃과 임베디드 루비 (Refactor)

이번 섹션까지, 세가지 페이지를 출력하는 레일스 컨트롤러와 액션을 작성했다. 그런데, 이들은 모두 순수하게 정적인 HTML 이라서 레일스의 능력을 보여주지 않았다. 게다가, 이 코드들은 심각할 정도로 중복된 부분을 포함한다.

- 페이지 타이틀이 거의 동일하다.
- “Ruby on Rails Tutorial Sample App” 은 세가지 타이틀에 공통되는 부분이다.
- 전체 HTML 구조가 각 페이지에 반복된다.

이 반복된 코드는 “Don’t Repeat Yourself” (DRY) 원칙을 위반한다. 이번과 다음 섹션에서 반복된 부분을 제거해서, “코드를 DRY할 것이다.” 그리고 마지막에, 섹션 3.4.2 테스트 코드를 다시 실행해서 타이틀이 올바른지 확인할 것이다.

역설적으로, 중복을 제거하는 첫번째 단계는 중복을 좀 더 추가하는 것이다. 페이지 타이틀을 좀 더 비슷하게, *정확히* 일치되도록 만들 것이다. 이 작업은 한번에 모든 중복을 제거할 수 있도록 좀 더 단순하게 만들어 줄 것이다.

뷰에서 *임베디드 루비*를 사용하는 기술을 도입할 것이다. **(" 임베디드 루비에 관한 링크(설명)이 있으면 좋을 것 같습니다.")** Home, Help와 About 페이지 타이틀에 변수를 포함하도록 할 것이기 때문에, 각 페이지에 서로 다른 타이틀을 표시하기 위해 특별한 레일스 함수인 `provide` 를 사용할 것이다. 목록 3.28에서 `home.html.erb` 내부의 “Home” 타이틀 글자를 어떻게 교체할수 있는지 볼 수 있다.

목록 3.28: 임베디드 루비로 구성한 제목이 적용된 Home 페이지 뷰. **green** app/views/static_pages/home.html.erb

```
<% provide(:title, "Home") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Sample App</h1>
    <p>
      This is the home page for the
      <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
      sample application.
    </p>
  </body>
</html>
```

목록 3.28는 임베디드 루비를 사용한 첫번째 예제이며, *ERb*라고도 부른다. (이제 왜 HTML 뷰가 `.html.erb` 파일 확장자를 가지는지 알 수 있다.) ERb는 웹페이지에서 동적 컨텐츠를 포함하는 대표적인 템플릿 시스템이다. [13]

```
<% provide(:title, "Home") %>
```

<% ... %> 코드는 레일스가 `provide` 를 호출해서 `"Home"` 문자열을 `:title` 라벨에 연결하는 것을 의미한다.[14] 그 다음에, 타이틀의 <%= ... %> 표기 부분에서 루비의 `yield` 함수로 해당 타이틀을 삽입한다.[15]

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

(임베디드 루비에서 이 두가지 표기법의 차이는 <% ... %> 표기는 내부 코드를 *실행*하기만 하는 반면, <%= ... %>는 템플릿에 실행 결과를 *삽입*한다는 점이다.) 페이지의 결과는 정확히 이전과 동일하며, 타이틀의 변경 부분만이 ERb에 의해서 동적으로 생성된다.

섹션 3.4.2의 모든 테스트를 실행하면 여전히 **green**상태다.

목록 3.29: **green**

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

그 다음으로, Help와 About 페이지도 같은 방법으로 교체할 수 있다.(목록 3.30과 목록 3.31).

목록 3.30: 임베디드 루비로 구성한 타이틀이 적용된 Help 페이지 뷰. **green** app/views/static_pages/help.html.erb

```
<% provide(:title, "Help") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>Help</h1>
    <p>
      Get help on the Ruby on Rails Tutorial at the
      <a href="http://www.railstutorial.org/#help">Rails Tutorial help
      section</a>.
      To get help on this sample app, see the
      <a href="http://www.railstutorial.org/book"><em>Ruby on Rails
      Tutorial</em> book</a>.
    </p>
  </body>
</html>
```

목록 3.31: 임베디드 루비로 구성한 타이틀이 적용된 About 페이지 뷰. **green**
app/views/static_pages/about.html.erb

```
<% provide(:title, "About") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    <h1>About</h1>
    <p>
      The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
      Tutorial</em></a> is a
      <a href="http://www.railstutorial.org/book">book</a> and
      <a href="http://screencasts.railstutorial.org/">screencast series</a>
      to teach web development with
      <a href="http://rubyonrails.org/">Ruby on Rails</a>.
      This is the sample application for the tutorial.
    </p>
  </body>
</html>
```

이제 다음과 같이 각 페이지 타이틀의 변경부분을 ERb로 교체했다.

```
<% provide(:title, "The Title") %>
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
  </head>
  <body>
    Contents
  </body>
</html>
```

다시 말해, 모든 페이지가 동일한 구조를 가지게 되는데, 타이틀 태그와 다른 내용의 `body` 태그를 포함한다.

이와 같이 공통된 구조를 별도로 분리하기 위해, 레일스는 `application.html.erb` 로 명명한 *특별한 레이아웃* 파일을 제공한다. 섹션 3.4에서 이름을 임시로 바꾸었는데, 이제 본래 파일 이름으로 되돌린다.

```
$ mv layout_file app/views/layouts/application.html.erb
```

레이아웃 작업을 위해 기본 타이틀을 위의 예제에서 처럼 임베디드 루비로 교체한다.

```
<title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
```

결과 레이아웃은 목록 3.32와 같다.

목록 3.32: sample 애플리케이션 사이트 레이아웃. **green** app/views/layouts/application.html.erb

```
<!DOCTYPE html>
<html>
  <head>
    <title><%= yield(:title) %> | Ruby on Rails Tutorial Sample App</title>
    <%= stylesheet_link_tag    'application', media: 'all',
                                              'data-turbolinks-track' => true %>
    <%= javascript_include_tag 'application', 'data-turbolinks-track' => true %>
    <%= csrf_meta_tags %>
  </head>
  <body>
    <%= yield %>
  </body>
</html>
```

여기서 아래의 특별한 코드라인을 주목한다.

```
<%= yield %>
```

이 코드는 레이아웃에 각 페이지의 응답 내용물을 채워 넣는 역할을 담당한다. 이것이 정확하게 어떤 원리로 동작하는지는

중요치 않다. 다만, 중요한 것은 레이아웃의 역할이다. 예컨데, /static_pages/home 페이지를 방문하면 `home.html.erb` 의 내용이 HTML로 변환된 후 <%= yield %> 부분에 삽입된다.

기본 레일스 레이아웃에는 몇가지 추가 코드가 포함되어 있다는 것은 주목할만 하다.

```
<%= stylesheet_link_tag ... %>
<%= javascript_include_tag "application", ... %>
<%= csrf_meta_tags %>
```

이 코드는 애플리케이션 스타일시트와 자바스크립트를 포함하는 애셋 파이프라인 부분(섹션 5.2.1)과, 악의적인 웹 공격의 한 형태인 크로스 사이트 요청 위조(cross-site request forgery) 방지를 위한 레일스 메소드 `csrf_meta_tags` 를 포함한다.

물론, 목록 3.28, 목록 3.30, 목록 3.31의 뷰는 아직 전체 HTML 레이아웃을 포함하는데, 내부 내용물만 남기고 제거한다. 깨끗하게 제거한 뷰의 결과를 목록 3.33, 목록 3.34, 목록 3.35에서 볼 수 있다.

목록 3.33:HTML 구조가 삭제된 Home 페이지. **green** app/views/static_pages/home.html.erb

```
<% provide(:title, "Home") %>
<h1>Sample App</h1>
<p>
  This is the home page for the
  <a href="http://www.railstutorial.org/">Ruby on Rails Tutorial</a>
  sample application.
</p>
```

목록 3.34: HTML 구조가 삭제된 Help 페이지. **green** app/views/static_pages/help.html.erb

```
<% provide(:title, "Help") %>
<h1>Help</h1>
<p>
  Get help on the Ruby on Rails Tutorial at the
  <a href="http://www.railstutorial.org/#help">Rails Tutorial help section</a>.
  To get help on this sample app, see the
  <a href="http://www.railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
  book</a>.
</p>
```

Listing 3.35: HTML구조가 삭제된 About 페이지. **green** app/views/static_pages/about.html.erb

```
<% provide(:title, "About") %>
<h1>About</h1>
<p>
  The <a href="http://www.railstutorial.org/"><em>Ruby on Rails
  Tutorial</em></a> is a
  <a href="http://www.railstutorial.org/book">book</a> and
  <a href="http://screencasts.railstutorial.org/">screencast series</a>
  to teach web development with
  <a href="http://rubyonrails.org/">Ruby on Rails</a>.
  This is the sample application for the tutorial.
</p>
```

새로 정의된 Home, Help 와 About 뷰 페이지는 예전과 같아 보이지만, 중복된 부분이 훨씬 적어졌다.

저자의 경험으로는 매우 간단한 리펙터링 작업도 에러가 발생하기 쉽고, 잘 못 되기 쉽다. 이는 일련의 테스트 코드를 잘 작성해야 하는 매우 소중한 이유 중 하나다. 모든 페이지를 재차 확인하는 것 보다, (초기에 이 절차는 어렵지 않지만, 그러나 응용 프로그램이 규모가 커질수록 급격히 어려워진다.) 일련의 테스트 코드를 **green** 상태로 유지하는 것으로 간단하게 검증할수 있다.

목록 3.36: **green**

```
$ bundle exec rake test
3 tests, 6 assertions, 0 failures, 0 errors, 0 skips
```

이것이 코드가 여전히 문제가 없음을 입증하는 *증거*가 될 수는 없지만, 향후 발생할 수 있는 버그를 방지하는 안전망 역할을 함으로써 코드가 정확하게 동작할 수 있도록 가능성을 매우 높여준다.

3.4.4 루트 라우트 설정

지금까지 사이트 페이지를 입맛에 맞게 수정했고, 일련의 테스트 코드 상 괜찮은 출발을 했으니, 다음은 애플리케이션 루트 라우트를 설정한다. 섹션 1.3.4 와 섹션 2.2.2 마찬가지로, / 와 원하는 페이지를 연결 하려면 `routes.rb` 파일을 수정해야 한다. 이 경우에는 Home 페이지로 연결할 것이다.(이 시점에서, 섹션 3.1에서 추가한 애플리케이션 컨트롤러의 `hello` 액션을 제거하는 편이 좋다.) 목록 3.37과 같이, 목록 3.5의 `get` 규칙을 변경한다.

```
root 'static_pages#home'
```

이렇게 변경하면 `static_pages/home` URL을 컨트롤러/액션 쌍인 `static_pages#home` 에 연결하게 되는데, / 로 들어오는 GET 요청을 Static Pages 컨트롤러의 `home` 액션으로 연결하라는 의미다. 그림 3.7 에서 라우트 파일 결과를 보자. (목록 3.37의 기존 `static_pages/home` 라우트는 더이상 동작하지 않는다.)

목록 3.37: Home 페이지로 루트 라우트 설정. config/routes.rb

```
Rails.application.routes.draw do
  root 'static_pages#home'
  get  'static_pages/help'
  get  'static_pages/about'
end
```

그림 3.7: 루트(/) 라우트로 연결되는 홈 페이지

3.5 결론

겉으로 보기에는, 이번 장에서는 한 일이 거의 없는 것 같다. 정적 페이지로부터 시작해서 *거의* 정적 페이지로 마무리했다. 그러나 그건 겉모습일 뿐이다. 레일스 컨트롤러, 액션과 뷰에 기초하여 개발하면 동적 컨텐츠를 사이트에 마음껏 추가할 수 있게 된다. 이러한 것들의 동작 원리를 정확히 알아 가는 것이 남은 튜토리얼에서 해야할 과제다.

다음 주제로 이동하기 전, master 브랜치로 이번 토픽 브랜치의 변경내용을 커밋한 후 머지하자. 섹션 3.2에서, 정적 페이지 개발을 위한 Git 브랜치를 만들었다. 아직 커밋하지 않았다면, 일단 중단 지점에서 커밋한다.

```
$ git add -A
$ git commit -m "Finish static pages"
```

그리고, 변경 사항을 섹션 1.4.4과 같은 방식으로 master 브랜치에 반영한다.:[16]

```
$ git checkout master
$ git merge static-pages
```

이와 같이 중단 지점에 도달할때, 원격 저장소로 코드를 푸시하는 것을 권장한다. (섹션 1.4.3의 절차를 따라 Bitbucket 를 이용할수 있다.)

```
$ git push
```

허로쿠로 애플리케이션을 배포하는 것도 추천한다.

```
$ bundle exec rake test
$ git push heroku
```

이처럼 배포전에 테스트 코드를 실행하는 것은 좋은 개발 습관이다.

3.5.1 이번 장에서 배운 것

- 세번째 장에서, 처음부터 새로운 레일스 애플리케이션을 생성하고, 필요한 젬을 설치하며, 작업한 내용을 원격 저장소로 푸시하고, 원격 서버로 서비스를 배포하는 전체 과정을 마쳤다.
- `rails` 스크립트를 사용하여 다음 명령으로 새로운 컨트롤러 생성하기.

`rails generate controller ControllerName <optional action names>`.
- `config/routes.rb` 파일에 새 라우트 규칙 정의하기.
- 레일스 뷰에서 HTML 작성 중간에 임베디드 루비(ERb)로 넣을 수 있다.
- 테스트를 자동화함으로써 새로운 기능 개발을 주도하는 테스트 코드를 작성할 수 있고 신뢰할 수 있는 리펙터링 작업과 회귀 방지를 가능하게 한다.
- 테스트 주도 개발은 “Red, Green, Refactor” 주기를 사용한다.
- 레일스 레이아웃은 애플리케이션의 모든 웹페이지에서 공통적으로 사용할 수 있는 템플릿을 사용할 수 있게 해주어 코드 중복을 제거한다.

3.6 연습문제

- www.railstutorial.org를 방문하여 레일스 튜토리얼을 구매하면 연습문제에 대한 해답을 구할 수 있다.
- 레일스 튜토리얼 이메일 리스트에 등록하면 보너스로 첫번째 toy 앱에 대한 레일스 튜토리얼 치트시트 카드를 얻게 된다.

지금부터 튜토리얼이 끝날때까지, 별도의 토픽 브랜치에서 연습문제를 풀 것을 추천한다.

```
$ git checkout static-pages
$ git checkout -b static-pages-exercises
```

이렇게 하면 메인 튜토리얼과의 충돌을 방지할수 있게 한다.

자신의 해결책이 만족스럽다면, 연습용 브랜치를 원격 저장소로 푸시할 수 있다(설정을 해두었을 경우).

```
<solve first exercise>
$ git commit -am "Eliminate repetition (solves exercise 3.1)"
<solve second exercise>
$ git add -A
$ git commit -m "Add a Contact page (solves exercise 3.2)"
$ git push -u origin static-pages-exercises
$ git checkout master
```

(앞으로의 개발을 대비하기 위해 마지막으로 취한 조치는 master 브랜치로 체크아웃하지만 튜토리얼의 나머지 부분과의 충돌을 피하기위해서 변경 사항을 머지하지 *않는 것이다*.) 다음 장들에서도 물론 브랜치와 커밋 메세지는 달라도, 기본 개념은 동일하다.

1. 아마도 Static Pages 컨트롤러 테스트에서 약간 반복된 부분을 눈치 챘을 것이다(목록 3.22). 특히, 기본 타이틀 “Ruby on Rails Tutorial Sample App”은 테스트할 때마다 변하지 않는 부분이다. 모든 테스트 전에 자동으로 실행되는 특별한 용도의 `setup` 함수를 사용하여 목록 3.38의 테스트 코드가 항상 **green** 상태를 유지하도록 한다.(목록 3.38 에서는 섹션 2.2.2과 4.4.5에서 다룬 *인스턴스 변수*와 섹션 4.2.2에서 언급한 *문자열 삽입*을 사용한다.)
2. sample 애플리케이션용 Contact 페이지를 만든다.[17] 목록 3.13을 모델로 삼아서, 우선 /static_pages/contact URL위치에 “Contact | Ruby on Rails Tutorial Sample App” 이라는 타이틀을 가진 페이지가 존재하는지 확인하는 테스트를 작성한다. 섹션 3.3.3의 About 페이지와 같은 절차로 테스트를 통과시키고, 목록 3.39와 같이 페이지 내

용을 작성한다.(연습 문제를 별도로 유지하기 위해, 목록 3.38의 변경 내용에 목록 3.39을 포함하지 않는다.)

목록 3.38: 기본 타이틀을 이용한 Static Pages 컨트롤러 테스트. **green**
test/controllers/static_pages_controller_test.rb

```
require 'test_helper'

class StaticPagesControllerTest < ActionController::TestCase

  def setup
    @base_title = "Ruby on Rails Tutorial Sample App"
  end

  test "should get home" do
    get :home
    assert_response :success
    assert_select "title", "Home | #{@base_title}"
  end

  test "should get help" do
    get :help
    assert_response :success
    assert_select "title", "Help | #{@base_title}"
  end

  test "should get about" do
    get :about
    assert_response :success
    assert_select "title", "About | #{@base_title}"
  end
end
```

목록 3.39: Contact 페이지를 채울 코드. app/views/static_pages/contact.html.erb

```
<% provide(:title, "Contact") %>
<h1>Contact</h1>
<p>
  Contact the Ruby on Rails Tutorial about the sample app at the
  <a href="http://www.railstutorial.org/#contact">contact page</a>.
</p>
```

3.7 고급 테스트 설정

옵션으로 추가된 이번 섹션에서는 루비온레일스 튜토리얼 스크린 캐스트 시리즈에서 사용했던 테스트 환경을 설정하는 방법에 대해서 설명한다. 주요 내용은 세 부분이다. 즉, 기능이 향상된 통과(pass)/실패(fail) 리포터(섹션 3.7.1), 테스트 실패 시 생성되는 역추적로그(backtrace)를 필터링하는 도구 (섹션 3.7.2), 그리고 파일이 변경되면 자동으로 관련 테스트가 실

행되는 자동 테스트 러너(runner)(섹션 3.7.3)에 대한 것이다. 이번 섹션에서 보게되는 코드는 수준이 높고, 단지 편의 목적으로만 제공한다. 따라서 이번 한번으로 모든 것을 이해하리라고 기대해서는 안된다.

이번 섹션의 변경 사항은 master 브랜치에서 진행한다.

```
$ git checkout master
```

3.7.1 MiniTest 리포터

기본 레일스 테스트가 **red** 과 **green**을 제대로 보여주기 위해서는, 목록 3.40의 코드를 헬퍼 파일에 추가할 것을 권장한다.[18] 이 때 목록 3.2에 포함되어 있는 minitest-reporters 젬을 사용해야 한다.

목록 3.40: **red**와 **green**이 보이도록 설정. test/test_helper.rb

```
ENV['RAILS_ENV'] ||= 'test'
require File.expand_path('../../config/environment', __FILE__)
require 'rails/test_help'
require "minitest/reporters"
Minitest::Reporters.use!

class ActiveSupport::TestCase
  # 모든 테스트에서 사용할 test/fixtures/*.yml 파일 내의 모든 픽스처를 알파벳순으로
  # 준비한다.
  fixtures :all

  # 여기에 모든 테스트에 사용할 헬퍼 메소드를 추가한다...
end
```

결과가 **red** 에서 **green**으로 바뀌면 그림 3.8처럼 클라우드 IDE 에 표시된다.

그림 3.8: 클라우드 IDE에서 **red**에서 **green**으로 바뀐다.

3.7.2 역추적로그 조절기능(Backtrace silencer)

에러가 발생하거나, 테스트가 실패하면, 테스트 러너는 애플리케이션에서 실패한 경로를 의미하는 “스택추적로그”나 “역추적로그”를 보여준다. 역추적로그는 보통 문제를 찾아가는데 매우 유용하지만, 몇몇 시스템(클라우드 IDE를 포함)에서는 레일스 자체 코드는 물론, 다양한 젬의 의존성과 애플리케이션 코드까지도 보여준다. 특히 문제의 원인이 애플리케이션이 의존하는 코드들이 아니고 보통 애플리케이션 그 자체임에도 , 결과로써 보여주는 역추적로그는 종종 불편할 정도로 길다.

이에 대한 해결책은, 원하지 않는 내용을 제거하는 역추적로그 필터다. 목록 3.2의 mini_backtrace 젬으로 *역추적로그 조절기능*을 사용한다. 클라우드 IDE에서, 가장 필요없는 내용은 `rvm` (루비 버전 관리자) 관련 정보다. 그래서 저자는 목록 3.41의 필터로 이들을 제거하였다.

목록 3.41: RVM를 위한 역추적로그 조절기능 추가. config/initializers/backtrace_silencers.rb

```
# 이 파일을 수정하면 반드시 서버를 재시작하라.

# 현재 사용하고 있지만 역추적로그에서 보고 싶지 않은 라이브러리에 대해서
# 역추적로그 조절기능을 추가할 수 있다.
Rails.backtrace_cleaner.add_silencer { |line| line =~ /rvm/ }

# 물론 프레임워크 코드에서 유발된 문제를 디버깅하기 위해서
# 모든 조절기능을 제거할 수도 있다.
# Rails.backtrace_cleaner.remove_silencers!
```

목록 3.41의 주석에 언급한 바와 같이 , 조절기능을 추가한 후에는 로컬 웹서버를 재시작해야 한다.

3.7.3 Guard 를 사용한 테스트 자동화

`rake test` 명령 실행시, 커맨드라인에서 손으로 테스트를 실행해야 한다는 점은 성가신 부분이 될 수 있다. 따라서 이런 불편함을 방지하기 위해, *Guard*를 사용하여 테스트가 자동으로 실행되도록 한다. Guard는 파일 시스템을 감시하는데, 가령 `static_pages_controller_test.rb` 파일을 변경하면, 해당 테스트만을 실행하도록 한다. Guard를 이용할 경우 더 좋은 것은 `home.html.erb` 파일이 변경될 때 관련된 `static_pages_controller_test.rb` 테스트도 자동으로 실행하도록 설정할 수 있다는 점이다.

목록 3.2의 `Gemfile` 은 이미 guard 젬을 포함하고 있어서 초기화만 해주면 된다.

```
$ bundle exec guard init
Writing new Guardfile to /home/ubuntu/workspace/sample_app/Guardfile
00:51:32 - INFO - minitest guard added to Guardfile, feel free to edit it
```

위의 명령을 실행하여 만들어지는 `Guardfile` 을 수정하면, 통합 테스트와 뷰가 갱신될때 Guard가 테스트를 제대로 실행할 것이다. 목록 3.42의 길이와 고급 기능을 감안하여, 그냥 복사-붙여넣기 할 것을 추천한다.)

목록 3.42: 사용자 정의 `Guardfile` .

```
# Guard용 매칭 규칙을 정의한다.
guard :minitest, spring: true, all_on_start: false do
  watch(%r{^test/(.*)/?(.*)_test\.rb$})
  watch('test/test_helper.rb') { 'test' }
  watch('config/routes.rb')    { integration_tests }
  watch(%r{^app/models/(.*?)\.rb$}) do |matches|
    "test/models/#{matches[1]}_test.rb"
  end
  watch(%r{^app/controllers/(.*?)_controller\.rb$}) do |matches|
    resource_tests(matches[1])
  end
  watch(%r{^app/views/([^/]*?)/.*\.html\.erb$}) do |matches|
    ["test/controllers/#{matches[1]}_controller_test.rb"] +
    integration_tests(matches[1])
  end
```

```
  watch(%r{^app/helpers/(.*?)_helper\.rb$}) do |matches|
    integration_tests(matches[1])
  end
  watch('app/views/layouts/application.html.erb') do
    'test/integration/site_layout_test.rb'
  end
  watch('app/helpers/sessions_helper.rb') do
    integration_tests << 'test/helpers/sessions_helper_test.rb'
  end
  watch('app/controllers/sessions_controller.rb') do
    ['test/controllers/sessions_controller_test.rb',
     'test/integration/users_login_test.rb']
  end
  watch('app/controllers/account_activations_controller.rb') do
    'test/integration/users_signup_test.rb'
  end
  watch(%r{app/views/users/*}) do
    resource_tests('users') +
    ['test/integration/microposts_interface_test.rb']
  end
end

# 해당 리소스에 대한 통합 테스트를 반환한다.
def integration_tests(resource = :all)
  if resource == :all
    Dir["test/integration/*"]
  else
    Dir["test/integration/#{resource}_*.rb"]
  end
end

# 해당 리소스에 대한 컨트롤러 테스트를 반환한다.
def controller_test(resource)
  "test/controllers/#{resource}_controller_test.rb"
end

# 해당 리소스에 대한 모든 테스트를 반환한다.
def resource_tests(resource)
  integration_tests(resource) << controller_test(resource)
end
```

이 라인은

```
guard :minitest, spring: true, all_on_start: false do
```

Guard가 실행될 때 전체 테스트 코드 실행을 막고, 레일스의 로딩 시간을 높이기 위해 Spring 서버를 사용하도록 한다.

Guard를 사용할 때 Spring와 Git의 충돌을 방지하려면, 반드시 `spring/` 디렉토리를 `.gitignore` 파일에 넣어

서 해당 파일들이 저장소에 커밋되지 않도록 해야한다. 클라우드 IDE를 사용할 때 방법은 다음과 같다.

1. 파일 탐색창의 오른쪽 상단의 기어 모양 아이콘을 클릭(그림 3.9).
2. “Show hidden files” 선택한 후 애플리케이션 루트에서 `.gitignore` 파일을 확인(그림 3.10).
3. `.gitignore` 파일(그림 3.11)을 더블 클릭하여 파일을 연 후에 목록 3.43 내용을 추가.

그림 3.9: 파일 탐색 창의 (잘 보이지 않는) 기어 아이콘.

그림 3.10: 파일 탐색기에서 숨긴 파일 표시.

그림 3.11: 일반적으로 숨겨진 `.gitignore` 파일을 표시.

목록 3.43: `.gitignore` 파일에 spring을 추가

```
# Git이 특정 파일을 무시 파일로 지정하는 것에 대해 더 많은 것을 알고자 한다면,
# https://help.github.com/articles/ignoring-files 주소를 찾아 본다.
#
# 텍스트 에디터나 운영시스템에서 생성하는 임시 파일을 무시 파일로 등록할 경우에는
# 전역에서 동작하는 무시 파일로 대신 추가할 수 있다.
#   git config --global core.excludesfile '~/.gitignore_global'

# bundler의 설정파일을 무시 파일로 등록한다.
/.bundle

# 기본으로 제공되는 SQLite 데이터베이스를 무시 파일로 등록한다.
/db/*.sqlite3
/db/*.sqlite3-journal

# 모든 로그 파일과 임시 파일을 무시 파일로 등록한다.
/log/*.log
/tmp

# Spring 파일들을 무시 파일로 등록한다.
/spring/*.pid
```

이 글을 작성할 시점에서 Spring 서버가 조금 이상하게 동작할 경우가 발생하기 때문에, 때로는 Spring *프로세스들*이 쌓여서 테스트의 성능 저하를 가져 오기도한다. 테스트가 눈에 띄게 느려진다면, 시스템 프로세스를 찾아서 죽이는 건 좋은 생각이다.(글상자 3.4).

글상자 3.4. 유닉스 프로세스

리눅스나 OS X 같은 유닉스 계열 시스템에서는, *프로세스(process)*라 불리는 잘 정의된 컨테이너에서 사용자와 시스템의 작업이 각각 발생한다. 모든 시스템의 프로세스를 보기 위해서는 ps 명령어에 aux 옵션을 사용한다.

```
$ ps aux
```

유형별로 필터링하려면, ps 결과물을 유닉스 파이프 |를 이용해서 grep 패턴-검사기를 사용한다.

```
$ ps aux | grep spring
ubuntu 12241 0.3 0.5 589960 178416 ? Ssl Sep20 1:46
spring app | sample_app | started 7 hours ago
```

프로세스에 대한 자세한 정보를 볼수 있지만, 가장 중요한것은 첫번째 번호인 *process id*, 혹은 pid 이다. 불필요한 프로세스를 삭제하려면, kill 명령어로 유닉스 종료 시그널 (우연히도 15란 숫자임)을 pid에 전달한다.

```
$ kill -15 12241
```

이는 개별 프로세스를 죽일 때 저자가 추천하는 기술인데, 예를 들어 (ps aux I grep server 명령으로 알아낸 pid를 사용하여) 오동작하는 레일스 서버에 사용할 수 있다. 그러나 때로는 특정 프로세스 이름과 일치하는 모든 프로세스를 죽어야 할때가 있다. 가령 spring 프로세스를 죽이고자 할 때다. 이런 특별한 경우에는, 먼저 spring 중지 명령어를 사용해 본다.

```
$ spring stop
```

때로는 이것이 동작하지 않을때, pkill 명령어를 사용하여 spring 이름을 가지는 모든 프로세스를 죽일 수 있다.

```
$ pkill -15 -f spring
```

예상대로 동작하지 않거나, 프로세스가 얼어버린 것처럼 보일 때는, ps aux 명령을 실행하여 어떤 일이 발생했는지 확인한 후, kill -15 <pid> 혹은 pkill -15 -f <name> 명령을 실행하여 해당 프로세스를 완전히 제거한다.

Guard 설정이 완료되면, 새 터미널을 열고 (섹션 1.3.2, 레일스 서버를 실행할 때 처럼) 다음 명령을 실행한다.

```
$ bundle exec guard
```

목록 3.42의 규칙은 이 튜토리얼에 최적화 되어 있어서 컨트롤러가 변경되면 자동으로 통합테스트가 실행된다. *모든* 테스트를 실행하려면, `guard>` 프롬프트에서 리턴키를 친다.(이는 종종 Spring 서버와 연결 실패 오류를 내기도 한다. 문제를 해결하려면 다시 리턴키를 친다.)

Guard 종료하려면, Ctrl-D를 누른다. Guard에 추가 설정은, 목록 3.42을 참고하고, Guard README와 Guard wiki를 본다.

1. 클라우드 IDE를 사용할 경우, “Goto Anything” 명령이 유용한데, 이는 파일이름의 부분만 입력해도 손쉽게 파일 시스템을 돌아 다닐수 있다. 이와 관련해서, hello, toy, sample 앱들이 한 프로젝트 내에 있다면, 평범한 파일 찾기로는 불편할수 있다. 예를 들어, “Gemfile”을 찾으려면, 여섯가지 가능한 파일을 보여줄텐데, 왜냐하면 각 프로젝트의 `Gemfile` 과 `Gemfile.lock` 가 모두 표시되기 때문이다. 따라서, 진행전에 `workspace` 디렉토리로 이동해서 `rm -rf helloapp/ toyapp/` 명령을 실행하여 처음 두 앱의 삭제를 고려할지도 모른다.(표 1.1) (Bitbucket에 저장한 후에는, 나중에 언제든 복구할 수 있다.) ↕
2. --without production 은 “기억되는 옵션”으로, 다음 `bundle install` 실행시 계속 반영된다. ↕
3. 필자는 개발 환경에서 PostgreSQL 설치하고 설정하는 방법을 배우기를 추천하지만, 지금은 하지 않겠다. 시간이 되면, Google 에서 “install configure postgresql <당신의 시스템>” 과 “rails postgresql setup” 을 검색해서 준비

해 두자. (클라우드 IDE에서, <당신의 시스템> 은 Ubuntu 다.) ↥

4. 2장에 언급대로, 이는 기본 레일스 페이지가 허로쿠에서 깨지는 주요 이유로, 배포 성공여부를 말하기는 힘들다. ↥
5. 여기서 다루는 정적 페이지 제작 방법은 아마 가장 단순한 것이지만, 이 방법만이 최선은 아니다. 실제로 최적화된 방법은 각 요구 사항에 따라 달라진다. 만약 *많은* 정적 페이지가 필요하면, 정적 페이지 컨트롤러를 이용하는 방법은 매우 귀찮아질 수 있지만, 그러나 이번 장의 sample 애플리케이션은 적은양만 다룬다. 많은 양의 정적 페이지가 필요하면, high*voltage* *젬을 고려해 볼 수 있다. 이런 이슈에 대해서는 시간이 오래 경과되긴 했지만 여전히 유용한 <a href="http://blog.hasmanythrough.com/2008/4/2/simple-pages" target="*blank">hasmanythrough 사이트의 simple pages 관련 글을 참고한다. ↥
6. 예를 들어, 레일스 창시자 David Heinemeier Hansson의 “TDD is dead. Long live testing.” ↥
7. 섹션 2.2 언급대로, `bundle exec` 는 섹션1.2.1의 클라우드 IDE를 포함해 몇몇 시스템에서 불필요하다. 그러나 저자는 완전한 설명을 위해 다루었다. 실제로, 저자는 에러가 나지 않으면, `bundle exec` 생략하고, 안되면 `bundle exec` 를 실행해본다. ↥
8. 기본적으로, `rake test` 는 테스트가 실패하면 빨간색을 표시하지만, 통과한다고 녹색을 표시하지는 않는다. 실제 Red–Green 주기를 보려면 섹션 3.7.1 참고한다. ↥
9. 일부 시스템에서, 오류 소스 경로를 위해서 “스택추적로그” 나 “역추적로그”를 위해 스크롤 해야할 수도 있다. 원하지 않는 역추적로그 정보를 제거하고 필터링하려면 섹션 3.7.2 참고 ↥
10. 시간이 경과되면 HTML 도 바뀐다. 명시적으로 문서 종류를 선언하면 미래에 현재 페이지를 출력하는 적당한 브라우저에게 정보를 알려줄 수 있다.간단한 doctype `<!DOCTYPE html>` 은 최신 HTML인 HTML5 의 표준이다. ↥
11. 일반적인 MiniTest 가정가설문은, 레일스 가이드의 가정가설문 표를 참고한다. ↥
12. 이책의 대부분은 Google Chrome 브라우저를 사용하지만, 그림 3.6에서는 Safari 를 사용하는데 왜냐하면 Chrome은 페이지 제목 전체를 표시하지 않기 때문이다. ↥
13. Haml은 두번째로 대중적인 템플릿 시스템이며 저자는 개인적으로 매우 좋아한다. 그러나 튜토리얼에서 사용할 법한 *완전한* 표준은 아니다. ↥
14. 숙련된 레일스 개발자들은 이 지점에서 `content*for*` *사용할것을 예상하지만, 그것은 asset pipeline 에서 잘 동작하지 않는다.* `*provide*` *함수가 대안이다. <a class="arrow" href="#cha-3*footnote-ref-14">↑
15. 이전에 루비를 공부했다면, 레일스가 컨텐츠를 블록에 *출력(yielding)* 하는 것에 의구심을 가질텐데, 그러한 의구심은 합리적이다. 그러나 레일스로 애플리케이션을 개발할 때 알 필요는 없다. ↥
16. 만약 Spring 프로세스 id (pid) 파일이 통합시 덮어쓰기 되면 명령행에서 `rm -f *.pid` 로 삭제해라. ↥
17. 이 연습 문제는 섹션 5.3.1에서 해결된다. ↥
18. 목록 3.40의 코드는 작은 따옴표와 큰 따옴표 문자열이 섞여 있다. `rails new` 가 작은 따옴표 문자열을 생성하는데 반하여 MiniTest reporters documentation는 큰 따옴표 문자열을 이용하기 때문이다. 루비에서 두 종류 문자열의 혼용은 흔한일이다. 더 많은 정보를 원할 경우섹션 4.2.2 를 참고한다. ↥