Lab 10: Windows Seven
BDD Web Development

Objective

This lab aims to teach students the basics of performing Web development using behavior-driven development (BDD). Students should already have a basic grasp of BDD principles, and by the end of this lab, will be able to apply those ideas to producing a basic RESTful Web site using BDD on a Sinatra application with Ruby.

Lab format

The lab begins by setting up a basic Web application example. It then introduces Cucumber into the Web development environment and proceeds in demonstrating app development with Cucumber and Sinatra with a guided walkthrough. Students complete the lab by adding a specified new feature to the sample app produced.

Required materials

Laptop, Internet connection, working Ruby 1.8 install with Cucumber

Grading rubric

Questions 1 through 3 are worth 5 points each.	$3 \times 5 = 15$
Tasks 2 and 6 are worth 10 points each.	$2 \times 10 = 20$
Task 3 is worth 5 points.	$1 \times 5 = 5$
Tasks 4 and 5 are worth 15 points each.	$2 \times 15 = 30$
Task 7 is worth 40 points.	$1 \times 40 = 40$
	110 points

Note there is a **20-point** extra credit opportunity on this lab!

Due date

The lab is due **Monday, May 9, 2011 at 11:55pm**.

Turn-in instructions

Create a personal Git repository on GitHub named "Lab10-username", where `username` is your Kerberos username, and commit the following files to it:

- Edited project files from Windows
- A PDF file with the answers to each question *clearly marked.* When you have completed this lab, you will have answered 3 questions.
- A `who.txt` file with your name. If you worked with someone else, include their name in the `who.txt` file.

Once committed, push all files to GitHub. You may work with a partner, as long as each of you pushes your own copy of all of the listed files. Note that files submitted via ANGEL, email, or other methods will receive zero credit. In addition, files that cannot be accessed without additional work (e.g. zip files, LaTeX documents that require compiling) will lose you points. After you push your files to GitHub, please complete the feedback survey!

Behavior-driven Web development

In Lab 9, we dealt with performing behavior-driven development on command-line applications. Testing Web applications, while slightly more complex, is very similar. The basic steps of BDD still apply: with Cucumber, we'll define features and scenarios in plain English, convert those steps to Ruby code, then execute our tests and write our Web application to match.

In this lab, we'll be using the Ruby gem Sinatra to provide a simple framework for doing rapid Web deployment. Sinatra makes it easy to write Web applications by defining "routes," simple handlers for different paths that can be called on a Web application. The canonical sample Sinatra application is:

```
require 'sinatra'

get '/' do
  "Hello world!"
end
```

When run, this app sets up a Sinatra internal web server on port 4567, and any requests to `http://localhost:4567/` will show "Hello world!" on the screen. You can read more about Sintra online. (For additional reading about the pattern of Web development Sinatra promotes, see the Wikipedia article on REST, or Representational State Transfer.)

Once you've read to your satisfaction, let's get started with an app!

Task 1: Install gems

You should have a Ruby install with DevKit available from the previous lab. Open Git Bash or a command prompt and run:

```
gem install sinatra cucumber-sinatra haml thin capybara
```

This will install the Sinatra, haml, Thin, and Capybara gems for use in Ruby projects. In addition, it installs the Cucumber extensions for testing Sinatra-based apps. Contact an instructor or TA if your gems don't install properly.

Task 2: Create directories and files *(10 pts)*

Because we'll be using Cucumber to test a Sinatra app, we need to introduce a little more complexity than the sample shown above. Create the following structure of files and folders, using `touch` to make files from your shell:

```
- Fortune-TeamXX
  - config.ru
  - fortune.rb
  - fortunedata.rb
```

First, download the contents of `fortunedata.rb` from the GitHub page for this lab. Next, open `fortune.rb` and insert the following:

```
require 'sinatra/base'
require 'haml'
require 'thin'

class Fortune < Sinatra::Base

end
```

In our base Fortune file, we've imported the core of the Sinatra gem, and also required two more gems for use in Web development: `haml`, used for HTML-like markup in Ruby, and `thin`, a basic Ruby Web middleware gem that provides some support for Sinatra in serving up Web pages. We've also declared a class for the core of the Fortune application that subclasses `Sinatra::Base`, allowing us to "execute" Fortune as a Sinatra application.

Finally, edit `config.ru` to contain the following:

```
require File.dirname(__FILE__) + "/fortune.rb"

map "/" do
  run Fortune
end
```

This config file does a bit of magic to grab the `fortune.rb` file out of the current directory, then calls the Fortune application we wrote earlier whenever its root is requested over HTTP.

We've just set up a primitive Sinatra application! It won't run yet, but once we define some handlers, we'll be ready to go.

Task 3: Initialize and test Cucumber *(5 pts)*

In Lab 9, we had to create the Cucumber framework ourselves. With `cucumber-sinatra`, though, we can have Ruby do it for us. Open a shell in your project directory, and run:

```
cucumber-sinatra init Fortune fortune.rb
```

You should see the generator create the familiar `features`, `support`, and `step_definitions` folders, along with the necessary Ruby files that Cucumber expects. At this point, let's confirm that Cucumber is working properly. Run:

```
cucumber
```

You should see the by-now familiar output:

```
0 scenarios
0 steps
0m0.000s
```

If Cucumber doesn't run properly, stop and ask your instructor or a TA for help. Once it works, **commit to Git**.

Task 4: Set up the first scenario *(15 pts)*

Now that Cucumber recognizes our web application, let's start writing some features. Create a file `random_fortune.feature` in the `features` directory, and place the following inside:

```
Feature: Random fortune
  In order to receive enlightenment
  As a depressed person
  I want to be given random fortunes

  Scenario: Get a fortune
    Given I am on fortunes
    Then I should see a fortune
```

Save and exit that file, then in the root of your project, run `cucumber` again. You should get something similar to the following output:

```
Feature: Random fortune
  In order to receive enlightenment
  As a depressed person
  I want to be given random fortunes

  Scenario: Get a fortune  # features/random_fortune.feature:6
    Given I am on fortunes # features/step_definitions/web_steps.rb:19
      Can't find mapping from "fortunes" to a path.
      Now, go and add a mapping in /Users/tim/code/Fortune/features/support/paths.rb (RuntimeError)
      ./features/support/paths.rb:24:in `path_to'
      ./features/step_definitions/web_steps.rb:20:in `/^(?:|I )am on (.+)$/'
      features/random_fortune.feature:7:in `Given I am on fortunes'
    Then I should see a fortune   # features/random_fortune.feature:8

Failing Scenarios:
cucumber features/random_fortune.feature:6 # Scenario: Get a fortune

1 scenario (1 failed)
2 steps (1 failed, 1 undefined)
0m0.003s

You can implement step definitions for undefined steps with these snippets:

Then /^I should see a fortune$/ do
  pending # express the regexp above with the code you wish you had
end
```

As usual for this step, we're yellow like a pepper. Open the file `features/support/paths.rb` and add the following text immediately before the "Add more mappings here" comment:

```
when /fortunes/i
  '/fortune/random/'
```

This is a convenience mapping for the step "I am on X" – it lets us refer to different pages in our Sinatra app easily. Now implement the step definition in `features/step_definitions/fortune_steps.rb`:

```
Then /^I should see a fortune$/ do
  fortune = page.find(".fortune")
  fortune.text.should match /^\n.*\n\s*$/
end
```

Once the step definition is implement, our tests should be complete for this portion of the application. Return to the root of your web application and run `cucumber`. You should see:

```
Feature: Random fortune
  In order to receive enlightenment
  As a depressed person
  I want to be given random fortunes

  Scenario: Get a fortune  # features/random_fortune.feature:6
    Given I am on fortunes # features/step_definitions/web_steps.rb:19
    Then I should see a fortune   # features/step_definitions/fortune_steps.rb:1
      Unable to find '.fortune' (Capybara::ElementNotFound)
      ./features/step_definitions/fortune_steps.rb:2:in ‘/^I should see a fortune$/’
      features/random_fortune.feature:8:in ‘Then I should see a fortune’

Failing Scenarios:
cucumber features/random_fortune.feature:6 # Scenario: Get a fortune

1 scenario (1 failed)
2 steps (1 failed, 1 passed)
0m0.241s
```

Uh-oh! **Red like a pepper**! Now we'll move on to satisfy our tests. Remember to **commit to Git**.

Task 5: Introduction to Sinatra and HAML *(15 pts)*

At this point, we need to start making changes to our Web application itself, so it passes our BDD tests. As such, we're going to introduce a couple concepts related to Sinatra that we'll use for the remainder of the lab.

- Sinatra uses **views** to display items in your browser. A view, for our purposes, is a single Web page. In advanced Sinatra apps, we can mix views together to build pages and templates; we won't use these features in this lab.
- A view is generally implemented using **HAML**, the HTML Abstraction Markup Language. HAML simplifies a lot of the boilerplate that goes into Web pages, and removes some of the tedium of writing plain HTML.
- **Routes** are individual paths in your Web application, but the word "route" can also be used to mean the Sinatra handler for a page URI. As such, we can refer to "writing routes" and the like.

With these terms under our collective belt, let's implement the view and route for getting a random fortune. Create a new folder in the root of your application called `views`, and in that folder touch the file `fortune.haml`. Edit that file and include the following text:

```
!!! 5

%html{:lang => "en"}
  %head
    %title Fortune
  %body
```

It's very important to note that *indentation counts* in HAML files. Be sure that your indentation in the HAML matches the nesting you would want in actual HTML; in this case, we want the `head` and `body` elements to be direct children of the `html` element, with the `title` element a child of the `head` element. In addition, we specified an HTML attribute on the `html` tag with a Ruby hash, defining `lang="en"`. This is a complete view that will render with an empty body and the title "Fortune" when called from an appropriate route.

Let's write that route now. In the root of your project, open the file `fortune.rb`, and in the definition of the Fortune class, add the following route block:

```
get '/fortune/random/?' do
  @fortune = $fortunes[rand($fortunes.size)]
  haml :fortune
end
```

This route will handle any requests to a page beginning with the string `/fortune/random` – the question mark at the end handles any bonus data in the URI, discarding it as unnecessary. Inside the `do` block, we specify what to do when this page is called. First, we get a fortune out of the global `$fortunes` array defined in `fortunedata.rb`, then we save it as the class variable `@fortune`. (Remember that variables beginning with an at symbol are classwide in Ruby.) Finally, we call the HAML generator on the `fortune.haml` file.

Note that in this block we depend on the `$fortunes` global array being available. That array is defined in the `fortunedata.rb` file, but we never included that file! Add the following line to the very end of `fortune.rb`:

```
require 'fortunedata'
```

So, we saved a fortune to a variable, but where is it displayed? We need to make one last change to `views/fortune.haml` that takes the fortune stored in that variable and displays it in the page. Recall that we're expecting a fortune to appear in the element `.fortune` for our Cucumber test; as such, we can append to the HAML file (indenting as appropriate):

```
    - if(@fortune)
      .fortune
        = @fortune
```

This entire block goes after the `%body` at the end of the current `fortune.haml` file, and should all be indented under that tag.

Run Cucumber once more, and you should see:

```
Feature: Random fortune
  In order to receive enlightenment
  As a depressed person
  I want to be given random fortunes

  Scenario: Get a fortune  # features/random_fortune.feature:6
    Given I am on fortunes # features/step_definitions/web_steps.rb:19
    Then I should see a fortune   # features/step_definitions/fortune_steps.rb:1

1 scenario (1 passed)
2 steps (2 passed)
0m0.211s
```

Green like a cuke! Remember to **commit to Git**!

Task 6: Dealing with user input *(10 pts)*

What we've done so far is all well and good, but most websites don't depend on their users to navigate to specific pages by URL alone. Instead, they use buttons and links to direct users around. Let's add a button now so that users can get new fortunes on their own, without having to manually refresh the page.

As always with BDD, we first write a new scenario in the `features/random_fortune.feature` file:

```
Scenario: Get a new fortune
  Given I am on fortunes
  When I press "New Fortune"
  Then I should see a fortune
```

This is mostly familiar: we already have the "I am on" and "Then I should see" steps implemented. Run Cucumber now, and we should see that it fails one scenario for want of a single step. More importantly, none of our steps are yellow like a pepper – the Cucumber generator we ran earlier handles the "When I press" step for us!

Since our steps are already understood, all we have to do is change our HAML slightly to include a new button with the text "New Fortune" on it, as expected. Edit `views/fortune.haml` and add the following two lines at the end of the file, *indented so they are direct children of the* `body` *tag*:

```
%form{:action => "", :method => "get"}
  %input{:type => "submit", :id => "new_fortune", :value => "New Fortune"}
```

Run Cucumber again to see that all of our tests are **green like a cuke**!

- [**Question 1** ***(5 pts)***] Explain, in your own words, what HAML is and what advantages it provides developers. What, if any, are the disadvantages of HAML? (Note that HAML is not Ruby-only; it can be compiled to HTML standalone.)
- [**Question 2** ***(5 pts)***] In your opinion, what is the biggest difference between CLI and Web testing with Cucumber?
- [**Question 3** ***(5 pts)***] Many people believe that Cucumber isn't worth the time for CLI testing. Is the same true for small Web applications? What about large (e.g. 1000-page) Web applications?

Task 7: Select specific routes *(40 pts)*

While we love nondeterminism in our Web applications, sometimes users would like to see a specific fortune. To that end, please use BDD to write another Sinatra endpoint that takes a specific fortune ID, and uses it to index into the `$fortunes` array. Call the endpoint `/fortune/get/ID`, where `ID` is the index of the fortune.

To get you started, here's a few quick Sinatra nuggets:

- You can include "arguments" in a Sinatra route by prefixing the argument name with a colon, like so: `get '/fortune/get/:fid' do`.
- Once included in the route, you can access the user-provided value for that argument in the `params` hash. Access it like: `params[:fid]`.
- The `.to_i` function attempts to convert its callee to an integer, so `"2".to_i == 2`. Sinatra arguments are always strings.

Debugging and a bonus

If you're stuck with the output of your Web application, or Sinatra isn't quite giving you what you want, you can run your Web application manually by going to the root of your app and running:

```
thin -R config.ru start
```

Use Ctrl+C to stop the server. Once started, you can view pages at http://localhost:3000.

Finally, if you need a couple extra points in your labs, you can gain up to 20 points on this lab by placing your application on Heroku, a free Ruby-based hosting service. While the TAs are glad to help with the required parts of the lab, for this extra credit adventure, you (and your optional partner) are on your own. Have fun, and if you do decide to take the extra credit, shoot the staff an email letting us know!

Turn-in instructions

Create a personal Git repository on GitHub named "Lab10-username", where `username` is your Kerberos username, and commit the following files to it:

- Edited project files from Windows

- A PDF file with the answers to each question *clearly marked*. When you have completed this lab, you will have answered 3 questions.
- A `who.txt` file with your name. If you worked with someone else, include their name in the `who.txt` file.

Once committed, push all files to GitHub. You may work with a partner, as long as each of you pushes your own copy of all of the listed files. Note that files submitted via ANGEL, email, or other methods will receive zero credit. In addition, files that cannot be accessed without additional work (e.g. zip files, LaTeX documents that require compiling) will lose you points. After you push your files to GitHub, please complete the feedback survey!

Revision History

- May 4, 2011: Sample application written by Eric Stokes
- May 4, 2011: Lab instructions written by Tim Ekl