TDD and the Beginnings of Dynamic Content

TDD and the Beginnings of Dynamic Content

- So as you saw last week in the lecture and practicals we used scaffolding to generate a pretty basic site
- However two things stood out
  - Not much coding was needed
  - However was very limited in the things it could do
  - Limited to adding / editing / deleting from the database

TDD and the Beginnings of Dynamic Content

- Today however we will start to build some static pages
- However some of their content will be build dynamically.
  - Here we will start introducing test driven development (TDD)
  - starting a web server, checking the site then killing the web server is ardious at best

Some initialisation work

- generate a rails project called "sample_app"
- make sure the Gemfile has ruby on rails version 3.2.11 and sqlite3

Some initialisation work

- add in the following lines to your gem file
  - group :development do
  - gem 'rspec-rails'
  - end
  -
  - group :test do
  - gem 'rspec'
  - gem 'webrat'
  - end

Some initialisation work

- do a bundle install
- As we are using TDD for this we have to setup rspec for this project by doing the following
  - "rails generate rspec:install"
- finally initialise a git repositry and add and commit all the files to it.

Some important notes

- Rails is capable of handling truly static web pages i.e HTML files
  - these are all served directly from the public/ directory of your rails project
    - have a look and you'll find a default index.html and {404,422,500}.html
  - Thus if you have some truly static pages there is a place for them to be stored

Some important notes

- Just to prove a point create a public/hello.html with the following contents
  - <!DOCTYPE html>
  - <head><title>Hello World!</title></head>
  - <body><h1>hello world</h1></body>
  - </html>
- Some of you will notice that we are using HTML5 tags here
- Navigate to the page in your webbrowser without the server and you will see it works.

Dynamic static pages

- However, what we are going to do is we will create some dynamic web content that looks like static pages.
- But before we do we need a new branch on our git repository to start writing this feature.
  - We don't want to destroy the mainline
  - "git checkout -b static-pages"
  - creates a new branch called static-pages and switches to that branch

Dynamic static pages

- Because we are using the MVC pattern of design we need to create a new controller for our static pages
  - "rails generate controller Pages home contact"
    - Generates a controller called Pages
    - And a couple of associated actions for a home page and a contact page.
- If you remember from last week we stated that a rails application has a router for web pages
  - let us now have a look at this router

Rails Router

- open up the file config/routes.rb and you will see the following
  - SampleApp::Application.routes.draw do
    - get pages/home
    - get pages/content
  - end
- Let us take the line get pages/home and look at it in more detail

Rails Router

- get pages/home
  - get corresponds to the GET HTTP verb
  - pages/home maps to the home action in the pages controller.
  - thus when a user accesses pages/home using a web browser
    - the browser sends a get action along with the URL
    - The router sees this
    - and calls the home action in the pages controller.

Rails Router

- What this means is you can use any combination of HTTP verb that can be sent from a browser along with any controller action to do what you wish
- We will exploit this in more detail later.
- Now we will see step by step how Rails takes this action and generates a web page

Rails Controllers

- Open up app/controllers/pages_controller.rb and you will see the following
  - class PagesController < ApplicationController
    - def home
    - end
    - def contact
    - end
  - end
- although these methods are empty they still produce pages because of the extension of the ApplicationController class

Rails Controllers

- There are still a few things RoR must do before it can render the page.
- Let's carry on with the get pages/home action
  - when the action is called the code in the the home method in the pages controller is executed.
  - however as there is no code to execute we can proceed to view the result
    - i.e. the View in our MVC
  - All views are stored under the app/views/pages directory.

Rails Controllers

- Under this directory you will find a lot of files with the extension .html.erb
  - these are Embedded Ruby files that render html pages
  - if you look at both of the pages here then you will notice that they contain some html for now
  - but for now we will commit our new controller to the repository but we will stay on the same branch
  - commit -am "added a pages controller"
    - any idea what the -a switch does?

Rails Controllers

- -a tells git to add all files before a commit. (unlike CVS/SVN)
- Now we will get started on TDD
- If you haven't got autotest installed I suggest you do so now

Test Driven Development

- First and foremost you will not really need tests for helpers and views
  - we will handle these tests in the controller tests for now
  - git rm -r spec/views
  - git rm -r spec/helpers
  - recursively removes the spec/views and spec/helpers directories from the git repository
  - However you can get them back by checking out an earlier version of the branch if needs must

Test Driven Development

- That spec directory is where all your RSpec tests will reside.
- Lets have a look at the contents of a test file.
- take spec/controllers
- By the way, if anyone has done Ruby before you will notice that the syntax shown here is not very Ruby like.
  - Ruby is mallable enough to allow for Domain Specific Languages
  - RSpec defines such a DSL

Test Driven Development

- require 'spec_helper'
- describe PagesController do
  - describe "GET 'home' " do
    - it "returns http success" do
      - get 'home'
      - response.should be_success
    - end
  - end
- end

Test Driven Development

- if you notice it looks very close to plain english
- the spec_helper is required anyway as it is fundamental to tests
- the first level of describe indicates which component we are testing.
  - in this case PagesController
- The second level of describe indicates the name of a set of tests to be performed as one unit

Test Driven Development

- The actual tests are described by the it block
- The string in quotes is the name of the test
- get 'home' is telling RSpec to imitate a get home action to the rails controller
- While response.should be_success is telling RSpec that it should recieve a HTTP code 200 (Success) after the command is executed.
- Try running all the tests by using this command (see which pass and fail)
  - rspec spec/

Test Driven Development

- At this point you can run auto test if you wish
  - autotest
- This will automatically run RSpec when you modify and save your files
- There is one small flaw with our test file.
  - because the pages aren't rendered by the tests, those tests will return true by default
  - however by adding a new line in our test file under describe PagesController do
    - render_views
  - RSpec will force render the page.

Test Driven Development

- Now lets go to the red stage of TDD
  - we will define a test for a new page called about however we will write the test first instead of the page
  - Thus in our PagesController Section we add a section similar to GET 'home' above but replacing home with about
- What happens when we run the tests again?

Test Driven Development

- We have entered the red stage
  - i.e. we have a failing test
  - thus to go green we now have to write an about page, however we will use Rails to dynamically construct it instead of making a static page
- First we modify app/controllers/pages_controller.rb and add the following empty method
  - def about
  - end
- Note that the test still fails (i.e. we are still red)

Test Driven Development

- This is because we have no route defined or a view defined
- We will add a route now.
- We want the page to respond to a get request so modify config/routes.rb and add in the line
  - get "pages/about"
- Again our test is still Red because we do not have a view defined but we will add a simple one now

Test Driven Development

- We will add a view now. Open a new file app/views/pages/about.html.erb and add these lines for now
  - <h1>Pages#About</h1>
  - find me in app/views/pages/about.html.erb
- you should find after this when the tests are rerun that we go to Green (i.e. no failed tests)

Test Driven Development

- For now we won't refactor our code because we don't have any code smells yet
  - a code smell is a piece of code that introduces negative design e.g. repeated code, inelegant solutions
  - we will do a little refactoring later.
  - but the tests will be quite valuable when we do refactor

Slightly Dynamic Pages

- The whole point of using RoR is that we can dynamically produce web content.
- Let's take a small step in that direction by creating dynamic titles for our 3 pages.
  - but we will adhere to TDD while we do so

Slightly Dynamic Pages

- We want our pages to have these titles
  - home "Ruby on Rails Tutorial Sample App | Home"
  - contact "Ruby on Rails Tutorial Sample App | Contact"
  - about "Ruby on Rails Tutorial Sample App | About"
- First and foremost define a new test to check that the home page has the right title

Slightly Dynamic Pages

- add a new test in the GET 'home' section of spec\controllers\pages_controller_spec.rb
  - it "should return the right title" do
    - get 'home'
    - response.should have_selector("title", :content => "Ruby on Rails Tutorial Sample App | Home")
  - end
- have_selector asks rspec to take the produced page and check for content in a given tag
  - in this case we are checking the title tags for the desired page title.

Slightly Dynamic Pages

- Our tests will fail again as expected
- However you should define similar tests for the contact and about pages.
- At this point we should modify our app/views/pages/home.html.erb to be a full static html page
  - seems like we are going the opposite direction but will make sense later

Slightly Dynamic Pages

- <!DOCTYPE html>
  - <head><title>Ruby on Rails Tutorial Sample App | Home</title></head>
  - <body>
    - <h1>Sample App</h1>
    - home page for our sample app
  - </body>
- </html>

Slightly Dynamic Pages

- you should define similar pages for your about and contact pages.
  - make sure to create different titles for those pages.
- thus your tests for those pages should pass too.
- Also this is a good example of a code smell called repitition
  - less repitition == smaller code base
  - less repitition == smaller chance of copy paste errors
- Violates the Don't Repeat Yourself (DRY) rule

Instance Variables and Refactoring

- So we will refactor and DRY out our code.
- We will start by adding some ruby code and variables in our actions to later cut down the codebase
- If you notice the titles for all three pages have the same start, but with a different word to end
  - we will capture these in ruby variables and generate titles on the fly

Instance Variables and Refactoring

- We will start by modifing our PagesController home method in app/controllers/PagesController.rb to look like this
  - def home
    - @title = home
  - end
- @ in ruby means we are declaring an instance variable

Instance Variables and Refactoring

- Let's now modify one line of our app/views/pages/home.html.erb with some embeded ruby code
- go from this
  - <title>Ruby on Rails Tutorial Sample App | Home</title>
- to this
  - <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>

Instance Variables and Refactoring

- <%= @title %> is a piece of embedded ruby code
  - tells rails to take the value of @title coming from the controller and place it in this exact spot
- you should find that your tests still work after this change
  - i.e. a successful refactor
- do similar changes to the contact and about pages

More Refactoring and Layouts

- If you notice we still have a lot of repeat HTML lying around.
  - Would be quite useful to get rid of the repeated sections
  - We can do this by using a layout
  - An embedded ruby file that defines content common to all pages
  - there is one to be found in app/views/layouts/application.html.erb

More Refactoring and Layouts

- Put the following text into the layout erb file
  - <!DOCTYPE html>
  - <head>
    - <title>Ruby on Rails Tutorial Sample App | <%= @title %></title>
    - <%= csrf_meta_tag %>
  - </head>
    - <body>
      - <%= yield %>
    - </body>
  - </html>

More Refactoring and Layouts

- A couple of extra tags to mention here.
  - <%= csrf_meta_tag %>
    - asks rails to input some code in here to prefent cross site request forgeries
    - a type of web attack
  - <%= yield %>
    - tells the layout to yield to the page erb for the content that goes in this section

More Refactoring and Layouts

- Now if we modify our page erbs we can rip out the html, head, title, and body tags.
- We should end up with green code but significantly smaller.
- Thus we have sucessful refactoring.