Hangperson: a scaffolded (!) ESaaS getting-started assignment

(v1.1, September 2015. Written by Armando Fox and Nick Herson) (some edits by mverdicchio 21 September 2015) (refinements by Armando Fox September 2017) (adapted for CS673 by Prabhat Vaish)

In this assignment you'll be introduced to part of the basic cycle of creating SaaS in a disciplined way.

Learning Goals

After completing this assignment, you will be able to:

- Create and deploy simple SaaS apps in your development environment, and deploy them to the public cloud
- Practice the basic workflow of test-driven development (TDD), in which tests are written before the code (so they fail when first run) and code is then added to make them pass
- Understand how SaaS frameworks such as Sinatra support the conceptual components of a three-tier SaaS application architecture
- Understand the challenges of adapting a non-SaaS application to a SaaS environment, including how to identify and manage application state
- Understand one use case of service-oriented architecture, in which your SaaS app relies on an external service's API (application programming interface) to provide part of the SaaS app's functionality.

Prerequisites

- You should be familiar with Ruby basics, for example by completing the Ruby Intro assignment.
- You should have read ESaaS Chapter 2, "The Architecture of SaaS Applications". You should have getting acquainted with all materials for lecture 2.
- You will need "survival level" Unix command-line skills and facility with an editor to edit code files.

NOTE: You may find the Sinatra documentation helpful to have on hand.

Introduction

The full Agile/XP cycle we follow in ESaaS includes talking to the customer, using BDD to develop scenarios, turning those scenarios into runnable integration/acceptance tests with Cucumber, using those scenarios plus TDD to drive the creation of actual code, and deploying the result of each iteration's work to the cloud. Your class project will also follow the same cycle.

In this introductory assignment, we've provided RSpec unit tests to let you use TDD to develop game logic for the popular word-guessing game Hangman. In the full Agile/XP cycle, you'd develop these tests yourself as you code.

You'll then use the Sinatra framework to make the Hangman game available as SaaS. Adapting the game logic for SaaS will introduce you to thinking about RESTful routes and service-oriented architecture. As you develop the "SaaS-ified" Hangman game, you'll use Cucumber to describe how gameplay will work from the player's point of view and as "full stack" integration tests that will drive SaaS development. In the full Agile/XP cycle, you'd develop Cucumber scenarios yourself based on consultation with the customer, and create the necessary *step definitions* (Cucumber code that turns plain-English scenarios into runnable tests). In this assignment we provide both the scenarios and step definitions for you.

You'll deploy your game to the cloud using Heroku, giving you experience in automating SaaS deployment.

Why Sinatra?

This assignment uses the simple Sinatra framework rather than Rails, so that you can focus on tools, mechanics, and SaaS concepts, all of which will readily map to Rails later. Since our app doesn't have a database and has very few functions, Sinatra is an easy way to get started.

Part 0: Demystifying SaaS app creation

Goal: Understand the steps needed to create, version, and deploy a SaaS app, including tracking the libraries it depends on so that your production and development environments are as similar as possible.

What you will do: Create a simple "hello world" app using the Sinatra framework, version it properly, and deploy it to Heroku.

Creating and versioning a simple SaaS app

SaaS apps are developed on your computer but *deployed to production* on a server that others can access. We try to minimize the differences between the development and production *environments*, to avoid difficult-to-diagnose problems in which something works one way on your development computer but a different way (or not at all) when that code is deployed to production.

We have two mechanisms for keeping the development and production environments consistent. The first is *version control*, such as Git, for the app's code. But since almost all apps also rely on *libraries* written by others, such as *gems* in the case of Ruby, we need a way to keep track of which versions of which libraries our app has been tested with, so that the same ones are used in development and production.

Happily, Ruby has a wonderful system for managing gem dependencies: a gem called **Bundler** looks for a file called `Gemfile` in the *app root directory* of each project. The `Gemfile` contains a list of gems and versions your app depends on. Bundler verifies that those gems, and any others that they in turn depend on, are properly installed on your system and accessible to the app.
Mkfor
Let's start with the following steps:

- Create a new empty directory to hold your new app, and use `git init` in that directory to start versioning it with Git.
- In that directory, create a new file called `Gemfile` (the capitalization is important) with the following contents. This file will be a permanent part of your app and will travel with your app anywhere it goes:

```
source 'https://rubygems.org'
ruby '>= 2.2.0', '< 3.0.0'

gem 'sinatra', '>= 2.0.1'
```

The first line says that the preferred place to download any necessary gems is https://rubygems.org, which is where the Ruby community registers "production ready" gems.

The second line specifies which version of the Ruby language interpreter is required. If we omitted this line, Bundler wouldn't try to verify which version of Ruby is available; there are subtle differences between the versions, and not all gems work with all versions, so it's best to specify this.

The last line says we need version 2.0.1 or later of the `sinatra` gem. In some cases we don't need to specify which version of a gem we want; in this case we do specify it because we rely on some features that are absent from earlier versions of Sinatra.

Run Bundler

Run the command `bundle`, which examines your `Gemfile` to make sure the correct gems (and, where specified, the correct versions) are available, and tries to install them otherwise. This will create a new file `Gemfile.lock`, *which you should place under version control.*
To place under version control, use these commands:

```
$ git add .
$ git commit -m "Set up the Gemfile"
```

The first command stages all changed files for committing. The second command commits the staged files with the comment in the quotes. You can repeat these commands to commit future changes. Remember that these are LOCAL commits -- if you want these changes on GitHub, you'll need to do a git push command, which we will show later.

What's the difference between the purpose and contents of `Gemfile` and `Gemfile.lock`? Which file is needed to completely reproduce the development environment's gems in the production environment?

`Gemfile` specifies the gems you need and in some cases the constraints on which version(s) are acceptable. `Gemfile.lock` records the *actual* versions found, not only of the gems you specified explicitly but also any other gems on which they depend, so it is the file used by the production environment to reproduce the gems available in the development environment.

After running `bundle`, why are there gems listed in `Gemfile.lock` that were not listed in `Gemfile`?

Bundler looked up the information for each Gem you requested (in this case, only `sinatra`) and realized that it depends on other gems, which in turn depend on still others, so it recursively installed all those dependencies. For example, the `rack` appserver is a gem, and while you didn't explicitly request it, `sinatra` depends on it. This is an example of the power of automation:

rather than requiring you (the app developer) to understand every Gem dependency, Bundler automates that process and lets you focus only on your app's top-level dependencies.

Create a simple SaaS app with Sinatra

As we learned in lecture 2, SaaS apps require a web server to receive HTTP requests from the outside world, and an application server that "connects" your app's logic to the web server. For development, we will use `webrick`, a very simple Ruby-based web server that would be inappropriate for production but is fine for development. In both development and production, we will use the `rack` Ruby-based application server, which supports Ruby apps written in various frameworks including Sinatra and Rails.

As we discussed in the class, a SaaS app essentially recognizes and responds to HTTP requests corresponding to the application's *routes* (recall that a route consists of an HTTP method such as `GET` or `POST` plus a URI). Sinatra provides an extremely lightweight shorthand for matching a route with the app code to be executed when a request using that route arrives from the Web server.

Create a file in your project called `app.rb` containing the following:

```
require 'sinatra'

class MyApp < Sinatra::Base
  get '/' do
    "<!DOCTYPE html><html><head></head><body><h1>Hello World</h1></body></html>"
  end
end
```

The `get` method is provided by the `Sinatra::Base` class, from which our `MyApp` class inherits; `Sinatra::Base` is available because we load the Sinatra library on line 1.

What *two* steps did we take earlier to guarantee that the Sinatra library is available to load in line 1?

We specified `gem 'sinatra'` in the `Gemfile` *and* successfully ran `bundle` to confirm that the gem is installed and "lock" the correct version of it in `Gemfile.lock`.

As you see from the above simple example, Sinatra lets you write functions that match an incoming HTTP route, in this case `GET '/'` (the root URL), a very simple HTML document containing the string `Hello World` will be returned to the presentation tier as the result of the request.

To run our app, we have to start the application server and presentation tier (web) server. The `rack` application server is controlled by a file `config.ru`, which you must now create and add to version control, containing the following:

```
require './app'
run MyApp
```

The first line tells Rack that our app lives in the file `app.rb`, which you created above to hold your app's code. We have to explicitly state that our `app` file is located in the current directory (.) because `require` normally looks only in standard system directories to find gems. If you're developing locally, you're now ready to test-drive our simple app with this command line:

```
$ bundle exec rackup
```

This command starts the Rack appserver and the WEBrick webserver. Prefixing it with `bundle exec` ensures that you are running with the gems specified in `Gemfile.lock`. Rack will look for `config.ru` and attempt to start our app based on the information there. If you're developing locally, you can visit `localhost:9292` in your browser to see the webapp. It will open in a new tab in the IDE if you click on it, but you should open up a fresh browser tab and paste in that URL.

Point a new Web browser tab at the running app's URL and verify that you can see "Hello World".

What happens if you try to visit a non-root URL such as `https://workspace-username.c9.io/hello` and why? (your URL root will vary. Try http://localhost:9292/hello)
You'll get a humorous error message from the Sinatra framework, since you don't have a route matching `get '/hello'` in your app. Since Sinatra is a SaaS framework, the error message is packaged up in a Web page and delivered to your browser.

You should now have the following files under version control: `Gemfile`, `Gemfile.lock`, `app.rb`, `config.ru`. This is a minimal SaaS app: the app file itself, the list of explicitly required gems, the list of actual gems installed including the dependencies implied by the required gems, and a configuration file telling the appserver how to start the app.

Modify the app

Modify `app.rb` so that instead of "Hello World" it prints "Goodbye World". Save your changes to `app.rb` and try refreshing your browser tab where the app is running.

No changes? Confused?

Now go back to the shell window where you ran `rackup` and press Ctrl-C to stop Rack. Then type `bundle exec rackup` again, and once it is running, go back to your browser tab with your app and refresh the page. This time it should work.

What this shows you is that if you modify your app while it's running, you have to restart Rack in order for it to "see" those changes. Since restarting it manually is tedious, we'll use the `rerun` gem, which restarts Rack automatically when it sees changes to files in the app's directory. (Rails does this for you by default during development, as we'll see, but Sinatra doesn't.)

You're probably already thinking: "Aha! If our app depends on this additional gem, we should add it to the Gemfile and run bundle to make sure it's really present." Good thinking. But it may also occur to you that this particular gem wouldn't be necessary in a production environment: we only need it as a tool while developing. Fortunately, there's a way to tell Bundler that some gems are only necessary in certain environments. Add the following to the Gemfile (it doesn't matter where):

```
group :development do
  gem 'rerun'
end
```

Now run `bundle install` to have it download the `rerun` gem and any dependencies, if they aren't already in place.

Any gem specifications inside the `group :development` block will only be examined if bundle is run in the development environment. (The other environments you can specify are :test and :production, and you can define new environments yourself.) Gem specifications outside of any group block are assumed to apply in all environments.

Say `bundle exec rerun -- rackup -p $PORT -o $IP` in the terminal window to start your app and verify the app is running. There are more details on rerun's usage available in the gem's GitHub README. Gem's are usually on GitHub and their README's full of helpful instructions about how to use them.

In this case we are prefixing with `bundle exec` again in order to ensure we are using the gems in the Gemfile.lock, and the `--` symbol is there to assert that the command we want rerun to operate with is `rackup -p $PORT -o $IP`. We could achieve the same effect with `bundle exec rerun "rackup -p $PORT -o $IP"`. They are equivalent. More importantly any detected

changes will now cause the server to restart automatically, similar to the use of `guard` to auto re-run specs when files change.

Modify `app.rb` to print a different message, and verify that the change is detected by refreshing your browser tab with the running app. Also before we move on you should commit your latest changes to git.

Deploy to Heroku

Heroku is a cloud platform-as-a-service (PaaS) where we can deploy our Sinatra (and later Rails) applications. If you don't have an account yet, go sign up at http://www.heroku.com. You'll need your login and password for the next step.

Install Heroku CLI following instructions.

Log in to your Heroku account by typing the command: `heroku login -i` in the terminal. This will connect you to your Heroku account.

While in the root directory of your project (not your whole workspace), type `heroku create` to create a new project in Heroku. This will tell the Heroku service to prepare for some incoming code, and locally it will add a remote git repository for you called `heroku`. Next, make sure you stage and commit all changes locally as instructed above (i.e. `git add`, `git commit`, etc).

Earlier we saw that to run the app locally you run `rackup` to start the Rack appserver, and Rack looks in `config.ru` to determine how to start your Sinatra app. How do you tell a production environment how to start an appserver or other processes necessary to receive requests and start your app? In the case of Heroku, this is done with a special file named `Procfile`, which specifies one or more types of Heroku processes your app will use, and how to start each one. The most basic Heroku process type is called a Dyno, or "web worker". One Dyno can serve one user request at a time. Since we're on Heroku's free tier, we can only have one Dyno. Let's create a file named `Procfile`, and only this as the name (i.e. Procfile.txt is not valid). Write the following line in your `Procfile`:

```
web: bundle exec rackup config.ru -p $PORT
```

This tells Heroku to start a single web worker (Dyno) using essentially the same command line you used to start Rack locally. Note that in some cases, a `Procfile` is not necessary since Heroku can infer from your files how to start the app. However, it's always better to be explicit.

Your local repo is now ready to deploy to Heroku:

```
$ git push heroku master
```

(`master` refers to which branch of the remote Heroku repo we are pushing to. We'll learn about branches later in the course, but for now, suffice it to say that you can only deploy to the `master` branch on Heroku.) This push will create a running instance of your app at some URL ending with `herokuapp.com`. Enter that URL in a new browser tab to see your app running live. Congratulations, you did it--your app is live!

Summary

- You started a new application project by creating a `Gemfile` specifying which gems you need and running `bundle` to verify that they're available and create the `Gemfile.lock` file that records the versions of gems actually in use.
- You created a Sinatra app in the file `app.rb`, pointed Rack at this file in `config.ru`, and used `rackup` to start the appserver and the WEBrick web server.
- You learned that changing the app's code doesn't automatically cause Rack to reload the app. To save the work of restarting the app manually every time you make a change, you used the `rerun` gem, adding it to the Gemfile in a way that specifies you won't need it in production, only during development.
- You versioned the important files containing not only your app's code but the necessary info to reproduce all the libraries it relies on and the file that starts up the app.
- You deployed this simple app to Heroku.

Part 1: Hangperson

With all this machinery in mind, clone this repo and let's work on Hangperson.

```
$ git clone https://github.com/saasbook/hw-sinatra-saas-hangperson
$ cd hw-sinatra-saas-hangperson
$ bundle
```

Developing Hangperson Using TDD and Guard

Goals: Use test-driven development (TDD) based on the tests we've provided to develop the game logic for Hangman, which forces you to think about what data is necessary to capture the game's state. This will be important when you SaaS-ify the game in the next part.

What you will do: Use `autotest`, our provided test cases will be re-run each time you make a change to the app code. One by one, the tests will go from red (failing) to green (passing) as you create the app code. By the time you're done, you'll have a working Hangperson game class, ready to be "wrapped" in SaaS using Sinatra.

Overview

Our Web-based version of the popular game "hangman" works as follows:

- The computer picks a random word
- The player guesses letters in order to guess the word
- If the player guesses the word before making seven wrong guesses of letters, they win; otherwise they lose. (Guessing the same letter repeatedly is simply ignored.)
- A letter that has already been guessed or is a non-alphabet character is considered "invalid", i.e. it is not a "valid" guess

To make the game fun to play, each time you start a new game the app will actually retrieve a random English word from a remote server, so every game will be different. This feature will introduce you not only to using an external service (the random-word generator) as a "building block" in a **Service-Oriented Architecture**, but also how a Cucumber scenario can test such an app deterministically with tests that **break the dependency** on the external service at testing time.

- In the app's root directory, say `bundle exec autotest`.

This will fire up the Autotest framework, which looks for various files to figure out what kind of app you're testing and what test framework you're using. In our case, it will discover the file called `.rspec`, which contains RSpec options and indicates we're using the RSpec testing framework. Autotest will therefore look for test files under `spec/` and the corresponding class files in `lib/`.

We've provided a set of 18 test cases to help you develop the game class. Take a look at `spec/hangperson_game_spec.rb`. It specifies behaviors that it expects from the class `lib/hangperson_game.rb`. Initially, we have added `, :pending => true` to every spec, so when Autotest first runs these, you should see the test case names printed in yellow, and the report "18 examples, 0 failures, 18 pending."

Now, with Autotest still running, delete `, :pending => true` from line 12, and save the file. You should immediately see Autotest wake up and re-run the tests. You should now have 18 examples, 1 failure, 17 pending.

The `describe 'new'` block means "the following block of tests describe the behavior of a 'new' HangpersonGame instance." The `hangpersonGame` line causes a new instance to be created, and the next lines verify the presence and values of instance variables.

Self Check Questions

According to our test cases, how many arguments does the game class constructor expect, and therefore what will the first line of the method definition look like that you must add to `hangperson_game.rb`?
One argument (in this example, "glorp"), and since constructors in Ruby are always named `initialize`, the first line will be `def initialize(new_word)` or something similar.

According to the tests in this `describe` block, what instance variables is a HangpersonGame expected to have?
`@word`, `@guesses`, and `@wrong_guesses`.

In order to make this failing test pass you'll need to create getters and setters for the instance variables mentioned in the self check tests above. Hint: use `attr_accessor`. When you've done this successfully and saved `hangperson_game.rb`, `autotest` should wake up again and the examples that were previously failing should now be passing (green).

Continue in this manner, removing `, :pending => true` from one or two examples at a time working your way down the specs, until you've implemented all the instance methods of the game class: `guess`, which processes a guess and modifies the instance variables `wrong_guesses` and `guesses` accordingly; `check_win_or_lose`, which returns one of

the symbols `:win`, `:lose`, or `:play` depending on the current game state;
and `word_with_guesses`, which substitutes the correct guesses made so far into the word.

Debugging Tip

When running tests, you can insert the Ruby command `byebug` into your app code to drop into the command-line debugger and inspect variables and so on. Type `h` for help at the debug prompt. Type `c` to leave the debugger and continue running your code.

- Take a look at the code in the class method `get_random_word`, which retrieves a random word from a Web service we found that does just that. Use the following command to verify that the Web service actually works this way. Run it several times to verify that you get different words.

```
$ curl --data '' http://watchout4snakes.com/wo4snakes/Random/RandomWord
```

(`--data` is necessary to force `curl` to do a POST rather than a GET. Normally the argument to `--data` would be the encoded form fields, but in this case no form fields are needed.)

Using `curl` is a great way to debug interactions with external services. `man curl` for (much) more detail on this powerful command-line tool.

Part 2: RESTful thinking for HangPerson

Note: Part 2 is just reading/background info for Part 3.

Goals: Understand how to expose your app's behaviors as RESTful actions in a SaaS environment, and how to preserve game state across (stateless) HTTP requests to your app using the appserver's provided abstraction for cookies.

What you will do: Create a Sinatra app that makes use of the HangpersonGame logic developed in the previous part, allowing you to play Hangperson via a browser.

Game State

Unlike a shrinkwrapped app, SaaS runs over the stateless HTTP protocol, with each HTTP request causing something to happen in the app. And because HTTP is stateless, we must think carefully about the following 2 questions:

1. What is the total state needed in order for the next HTTP request to pick up the game where the previous request left off?

2. What are the different game actions, and how should HTTP requests map to those actions?

The widely-used mechanism for maintaining state between a browser and a SaaS server is a **cookie**. The server can put whatever it wants in the cookie (up to a length limit of 4K bytes); the browser promises to send the cookie back to the server on each subsequent request. Since each user's browser gets its own cookie, the cookie can effectively be used to maintain per-user state.

In most SaaS apps, the amount of information associated with a user's session is too large to fit into the 4KB allowed in a cookie, so as we'll see in Rails, the cookie information is more often used to hold a pointer to state that lives in a database. But for this simple example, the game state is small enough that we can keep it directly in the session cookie.

Self Check Question

Enumerate the minimal game state that must be maintained during a game of Hangperson.

The secret word; the list of letters that have been guessed correctly; the list of letters that have been guessed incorrectly. Conveniently, the well-factored HangpersonGame class encapsulates this state using its instance variables, as proper object-oriented design recommends.

The game as a RESTful resource

Self Check Question

Enumerate the player actions that could cause changes in game state.
Guess a letter: possibly modifies the lists of correct or incorrect guesses; possibly results in winning or losing the game.

Start new game: chooses a new word and sets the incorrect and correct guess lists to empty.

In a service-oriented architecture, we do not expose internal state directly; instead we expose a set of HTTP requests that either display or perform some operation on a hypothetical underlying **resource**. **The trickiest and most important part of RESTful design is modeling what your resources are and what operations are possible on them.**

In our case, we can think of the game itself as the underlying resource. Doing so results in some important design decisions about how routes (URLs) will map to actions and about the game code itself.

Since we've already identified the game state and player actions that could change it, it makes sense to define the game itself as a class. An instance of that class is a game, and represents the resource being manipulated by our SaaS app.

Mapping resource routes to HTTP requests

Our initial list of operations on the resource might look like this, where we've also given a suggestive name to each action:

1. `create`: Create a new game
2. `show`: Show the status of the current game
3. `guess`: Guess a letter

Self Check Question

For a good RESTful design, which of the resource operations should be handled by HTTP GET and which ones should be handled by HTTP POST?
Operations handled with `GET` should not have side effects on the resource, so `show` can be handled by a `GET`, but `create` and `guess` (which modify game state) should use `POST`. (In fact, in a true service-oriented architecture we can also choose to use other HTTP verbs like `PUT` and `DELETE`, but we won't cover that in this assignment.)

HTTP is a request-reply protocol and the Web browser is fundamentally a request-reply user interface, so each action by the user must result in something being displayed by the browser. For the "show status of current game" action, it's pretty clear that what we should show is the HTML representation of the current game, as the `word_with_guesses` method of our game class does. (In a fancier implementation, we would arrange to draw an image of part of the hanging person.)

But when the player guesses a letter--whether the guess is correct or not--what should be the "HTML representation" of the result of that action?

Answering this question is where the design of many Web apps falters.

In terms of game play, what probably makes most sense is after the player submits a guess, display the new game state resulting from the guess. **But we already have a RESTful action for displaying the game state.** So we can plan to use an **HTTP redirect** to make use of that action.

This is an important distinction, because an HTTP redirect triggers an entirely new HTTP request. Since that new request does not "know" what letter was guessed, all of the responsibility for **changing** the game state is associated with the guess-a-letter RESTful action, and all of the responsibility for **displaying** the current state of the game without changing it is associated with the display-status action. This is quite different from a scenario in which the guess-a-letter action **also** displays the game state, because in that case, the game-display action would have access to what letter was guessed. Good RESTful design will keep these responsibilities separate, so that each RESTful action does exactly one thing.

A similar argument applies to the create-new-game action. The responsibility of creating a new game object rests with that action (no pun intended); but once the new game object is created, we already have an action for displaying the current game state.

So we can start mapping our RESTful actions in terms of HTTP requests as follows, using some simple URIs for the routes:

Route and action	Resource operation	Web result
`GET /show`	show game state	display correct & wrong guesses so far

Route and action	Resource operation	Web result
`POST /guess`	update game state with new guessed letter	redirect to `show`
`POST /create`	create new game	redirect to `show`

In a true service-oriented architecture, we'd be nearly done. But a site experienced through a Web browser is not quite a true service-oriented architecture.

Why? Because a human Web user needs a way to `POST` a form. A `GET` can be accomplished by just typing a URL into the browser's address bar, but a `POST` can only happen when the user submits an HTML form (or, as we'll see later, when AJAX code in JavaScript triggers an HTTP action).

So to start a new game, we actually need to provide the user a way to post the form that will trigger the `POST /create` action. You can think of the resource in question as "the opportunity to create a new game." So we can add another row to our table of routes:

`GET /new`	give human user a chance to start new game	display a form that includes a "start new game" button

Similarly, how does the human user generate the `POST` for guessing a new letter? Since we already have an action for displaying the current game state (`show`), it would be easy to include on that same HTML page a "guess a letter" form that, when submitted, generates the `POST /guess` action.

We will see this pattern mirrored later in Rails: a typical resource (such as the information about a player) will have `create` and `update` operations, but to allow a human being to provide the data used to create or update a player record, we will have to provide `new` and `edit` actions respectively that allow the user to enter the information on an HTML form.

Self Check Questions

Why is it appropriate for the `new` action to use `GET` rather than `POST`?

The `new` action doesn't by itself cause any state change: it just returns a form that the player can submit.

Explain why the `GET /new` action wouldn't be needed if your Hangperson game was called as a service in a true service-oriented architecture.
In a true SOA, the service that calls Hangperson can generate an HTTP `POST` request directly. The only reason for the `new` action is to provide the human Web user a way to generate that request.

Lastly, when the game is over (whether win or lose), we shouldn't be accepting any more guesses. Since we're planning for our `show` page to include a letter-guess form, perhaps we should have a different type of `show` action when the game has ended---one that does **not** include a way for the player to guess a letter, but (perhaps) does include a button to start a new game. We can even have separate pages for winning and losing, both of which give the player the chance to start a new game. Since the `show` action can certainly tell if the game is over, it can conditionally redirect to the `win` or `lose` action when called.
The routes for each of the RESTful actions in the game, based on the description of what the route should do:

Show game state, allow player to enter guess; may redirect to Win or Lose	GET /show
Display form that can generate `POST /create`	GET /new
Start new game; redirects to Show Game after changing state	POST /create
Process guess; redirects to Show Game after changing state	POST /guess
Show "you win" page with button to start new game	GET /win
Show "you lose" page with button to start new game	GET /lose

Summary of the design

You may be itchy about not writing any code yet, but you have finished the most difficult and important task: defining the application's basic resources and how the RESTful routes will map them to actions in a SaaS app. To summarize:

- We already have a class to encapsulate the game itself, with instance variables that capture the game's essential state and instance methods that operate on it when the player makes guesses. In the model-view-controller (MVC) paradigm, this is our model.

- Using Sinatra, we will expose operations on the model via RESTful HTTP requests. In MVC, this is our controller.
- We will create HTML views and forms to represent the game state, to allow submitting a guess, to allow starting a new game, and to display a message when the player wins or loses. In MVC, these are our views.

Note that Sinatra does not really enforce MVC or any other design pattern---if anything, it's closest to the Page Controller pattern, where we explicitly match up each RESTful request with an HTML view---but it's a simple enough framework that we can use it to implement MVC in this app since we have only one model. As we'll see later, more powerful MVC-focused frameworks like Rails are much more productive for creating apps that have many types of models.

Part 3: Connecting HangpersonGame to Sinatra

You've already met Sinatra. Here's what's new in the Sinatra app skeleton `app.rb` that we provide for Hangperson:

- `before do...end` is a block of code executed *before* every SaaS request
- `after do...end` is executed *after* every SaaS request
- The calls `erb :` *action* cause Sinatra to look for the file `views/`*action*`.erb` and run them through the Embedded Ruby processor, which looks for constructions `<%= like this %>`, executes the Ruby code inside, and substitutes the result. The code is executed in the same context as the call to `erb`, so the code can "see" any instance variables set up in the `get` or `post` blocks.

Self Check Question

`@game` in this context is an instance variable of what class? (Careful-- tricky!)
It's an instance variable of the `HangpersonApp` class in the app.rb file. Remember we are dealing with two Ruby classes here: the `HangpersonGame` class encapsulates the game logic itself (that is, the Model in model-view-controller), whereas `HangpersonApp` encapsulates the logic that lets us deliver the game as SaaS (you can roughly think of it as the Controller logic plus the ability to render the views via `erb`).

The Session

We've already identified the items necessary to maintain game state, and encapsulated them in the game class. Since HTTP is stateless, when a new HTTP request comes in, there is no notion of the "current game". What we need to do, therefore, is save the game object in some way between requests.

If the game object were large, we'd probably store it in a database on the server, and place an identifier to the correct database record into the cookie. (In fact, as we'll see, this is exactly what Rails apps do.) But since our game state is small, we can just put the whole thing in the cookie. Sinatra's `session` library lets us do this: in the context of the Sinatra app, anything we place into the special "magic" hash `session[]` is preserved across requests. In fact, objects placed there are *serialized* into a text-friendly form that is preserved for us. This behavior is switched on by the Sinatra call `enable :sessions` in `app.rb`.

There is one other session-like object we will use. In some cases above, one action will perform some state change and then redirect to another action, such as when the Guess

action (triggered by `POST /guess`) redirects to the Show action (`GET /show`) to redisplay the game state after each guess. But what if the Guess action wants to display a message to the player, such as to inform them that they have erroneously repeated a guess? The problem is that since every request is stateless, we need to get that message "across" the redirect, just as we need to preserve game state "across" HTTP requests.

To do this, we use the `sinatra-flash` gem, which you can see in the Gemfile. `flash[]` is a hash for remembering short messages that persist until the *very next* request (usually a redirect), and are then erased.

Self Check Question

Why does this save work compared to just storing those messages in the `session[]` hash?
When we put something in `session[]` it stays there until we delete it. The common case for a message that must survive a redirect is that it should only be shown once; `flash[]` includes the extra functionality of erasing the messages after the next request.

Running the Sinatra app

As before, run the shell command `bundle exec rackup` to start the app, or `bundle exec rerun -- rackup` if you want to rerun the app each time you make a code change.

Self Check Question

Based on the output from running this command, what is the full URL you need to visit in order to visit the New Game page?
The Ruby code `get '/new' do...` in `app.rb` renders the New Game page, so the full URL is in the form `http://localhost:9292/new`

Visit this URL and verify that the Start New Game page appears.

Self Check Question

Where is the HTML code for this page?
It's in `views/new.erb`, which is processed into HTML by the `erb :new` directive.

Verify that when you click the New Game button, you get an error. This is because we've deliberately left the `<form>` that encloses this button incomplete: we haven't specified where the form should post to. We'll do that next, but we'll do it in a test-driven way.

But first, let's get our app onto Heroku. This is actually a critical step. We need to ensure that our app will run on heroku **before** we start making significant changes.

- First, run `bundle install` to make sure our Gemfile and Gemfile.lock are in sync.
- Next, type `git add .` to stage all changed files (including Gemfile.lock)
- Then type `git commit -m "Ready for Heroku!"` to commit all local changes.
- Next, type `heroku login` and authenticate.
- Since this is the first time we're telling Heroku about the Hangperson app, we must type `heroku create` to have Heroku prepare to recieve this code and to have it create a git reference for referencing the new remote repository.
- Then, type `git push heroku master` to push your code to Heroku.
- When you want to update Heroku later, you only need to commit your changes to git locally, then push to Heroku as in the last step.
- Verify that the Heroku-deployed Hangperson behaves the same as your development version before continuing. A few lines up from the bottom of the Heroku output in the terminal should have a URL ending in herokuapp.com. Find that, copy it to the clipboard, and paste it into a browser tab to see the current app.
- Verify the broken functionality by clicking the new game button.

Part 4: Introducing Cucumber

Cucumber is a remarkable tool for writing high-level integration and acceptance tests, terms with which you're already familiar. We'll learn much more about Cucumber later, but for now we will use it to *drive* the development of your app's code.

Just as you used RSpec to "drive" the creation of the class's methods, you'll next use Cucumber to drive the creation of the SaaS code.

Normally, the cycle would be:

1. Use Cucumber scenario to express end-to-end behavior of a scenario
2. As you start writing the code to make each step of the scenario pass, use RSpec to drive the creation of that step's code
3. Repeat until all scenario steps are passing green

In this assignment we're skipping the middle step since the goal is to give you an overview of all parts of the process. Also, in the first step, for your own apps you'd be creating the Cucumber scenarios yourself; in this assignment we've provided them for you.

Cucumber lets you express integration-test scenarios, which you'll find in the `features` directory in `.feature` files. You'll also see a `step_definitions` subdirectory with a single file `game_steps.rb`, containing the code used when each "step" in a scenario is executed as part of the test.

As an integration testing tool, Cucumber can be used to test almost any kind of software system as long as there is a way to *simulate* the system and a way to *inspect* the system's behavior. You select a *back end* for Cucumber based on how the end system is to be simulated and inspected.

Since a SaaS server is simulated by issuing HTTP requests, and its behavior can be inspected by looking at the HTML pages served, we configure Cucumber to use Capybara, a Ruby-based browser simulator that includes a domain-specific language for simulating browser actions and inspecting the SaaS server's responses to those actions.

Self Check Questions

Read the section on "Using Capybara with Cucumber" on Capybara's home page. Which step definitions use Capybara to simulate the server as a browser would? Which step definitions use Capybara to inspect the app's response to the stimulus?

Step definitions that use `visit`, `click_button`, `fill_in` are simulating a browser by visiting a page and/or filling in a form on that page and clicking its buttons. Those that use `have_content` are inspecting the output.

Looking at `features/guess.feature`, what is the role of the three lines following the "Feature:" heading?
They are comments showing the purpose and actors of this story. Cucumber won't execute them.

In the same file, looking at the scenario step `Given I start a new game with word "garply"`, what lines in `game_steps.rb` will be invoked when Cucumber tries to execute this step, and what is the role of the string `"garply"` in the step?
Lines 13-16 of the file will execute. Since a step is chosen by matching a regular expression, `word` will match the first (and in this case only) parenthesis capture group in the regexp, which in this example is `garply`.

Get your first scenario to pass

We'll first get the "I start a new game" scenario to pass; you'll then use the same techniques to make the other scenarios pass, thereby completing the app. So take a look at the step definition for "I start a new game with word...".

You already saw that you can load the new game page, but get an error when clicking the button for actually creating a new game. You'll now reproduce this behavior with a Cuke scenario.

Self Check Question

When the "browser simulator" in Capybara issues the `visit '/new'` request, Capybara will do an HTTP GET to the partial URL `/new` on the app. Why do you think `visit` always does a GET, rather than giving the option to do either a GET or a POST in a given step?
Cucumber/Capybara is only supposed to be able to do what a human user can do. As we discussed earlier, the only way a human user can cause a POST to happen via a web browser is submitting an HTML form, which is accomplished by `click_button` in Capybara.

Run the "new game" scenario with:

```
$ cucumber features/start_new_game.feature
```

If you get an error about Cucumber like this one, just follow the advice and run `bundle install` first.

```
~/workspace/hw-sinatra-saas-hangperson (master) $ cucumber
features/start_new_game.feature
Could not find proper version of cucumber (2.0.0) in any of the sources
Run `bundle install` to install missing gems.
```

The scenario fails because the `<form>` tag in `views/new.erb` is incorrect and incomplete in the information that tells the browser what URL to post the form to. Based on the table of routes we developed in an earlier section, fill in the `<form>` tag's attributes appropriately. You can inspect what happens for various routes in app.rb, but you don't need to edit this file yet. (Hint: if you get stuck, take a look at `show.erb` (at the bottom) for a similar example of a filled in form tag.)

The create-new-game code in the Sinatra app should do the following:

- Call the HangpersonGame class method `get_random_word`
- Create a new instance of HangpersonGame using that word
- Redirect the browser to the `show` action

View how these steps are actualized in the app.rb file under the `post /create do` route. Now stage and commit all files locally, then `git push heroku master` to deploy to Heroku again and manually verify this improved behavior.

Self Check Question

What is the significance of using `Given` vs. `When` vs. `Then` in the feature file? What happens if you switch them around? Conduct a simple experiment to find out, then confirm your results by using Google.

The keywords are all aliases for the same method. Which one you use is determined by what makes the scenario most readable.

Develop the scenario for guessing a letter

For this scenario, in `features/guess.feature`, we've already provided a correct `show.erb` HTML file that submits the player's guess to the `guess` action. You already have a `HangpersonGame#guess` instance method that has the needed functionality.

Self Check Question

In `game_steps.rb`, look at the code for "I start a new game..." step, and in particular the `stub_request` command. Given the hint that that command is provided by a Gem (library) called `webmock`, what's going on with that line, and why is it needed? (Use Google if needed.)

Webmock lets our tests "intercept" HTTP requests coming **from** our app and directed to another service. In this case, it's intercepting the POST request (the same one you manually did with `curl` in an earlier part of the assignment) and faking the reply value. This lets us enforce deterministic behavior of our tests, and also means we're not hitting the real external server each time our test runs.

The special Sinatra hash `params[]` has a key-value pair for each nonblank field on a submitted form: the key is the symbolized `name` attribute of the form field and the value is what the user typed into that field, or in the case of a checkbox or radiobutton, the browser-specified values indicating if it's checked or unchecked. ("Symbolized" means the string is converted to a symbol, so `"foo"` becomes `:foo`.)

Self Check Question

In your Sinatra code for processing a guess, what expression would you use to extract *just the first character* of what the user typed in the letter-guess field of the form in `show.erb`? **CAUTION:** if the user typed nothing, there won't be any matching key in `params[]`, so dereferencing the form field will give `nil`. In that case, your code should return the empty string rather than an error.
`params[:guess].to_s[0]` or its equivalent. `to_s` converts `nil` to the empty string in case the form field was left blank (and therefore not included in `params` at all). `[0]` grabs the first character only; for an empty string, it returns an empty string.

In the `guess` code in the Sinatra app.rb file, you should:

- Extract the letter submitted on the form. (given above and in the code for you)
- Use that letter as a guess on the current game. (add this code in)
- Redirect to the `show` action so the player can see the result of their guess. (done for you as well)

While you're here, read the comments in the file. They give clues for future steps in this assignment.

When finished adding that code, verify that all the steps in `features/guess.feature` now pass by running cucumber for that .feature file.

- Debugging tip: The Capybara command `save_and_open_page` placed in a step definition will cause the step to open a Web browser window showing what the page looks like at that point in the scenario. The functionality is provided in part by a gem called `launchy` which is in the Gemfile.

Part 5: Corner Cases

THIS IS OPTIONAL. Completing this section will earn you extra credit.

By now you should be familiar with the cycle:

1. Pick a new scenario to work on (you should have 2/4 working at this point)
2. Run the scenario and watch it fail
3. Develop code that makes each step of the scenario pass
4. Repeat till all steps passing.

Use this process to develop the code for the remaining actions `win` and `lose`. You will need to add code to the `show` action that checks whether the game state it is about to show is actually a winning or losing state, and if so, it should redirect to the appropriate `win` or `lose` action. Recall that your game logic model has a method for testing if the current game state is win, lose, or keep playing. The scenario `game_over.feature` tests these behaviors in your SaaS app. Push to Heroku and make sure everything still works. Give yourself a break and play a few rounds of hangperson.

While you're playing, what happens if you directly add `/win` to the end of your app's URL? Make sure the player cannot cheat by simply visiting `GET /win`. Consider how to modify the actions for `win`, `lose`, and `show` to prevent this behavior.

- What to submit: Make sure all Cucumber scenarios are passing. A shorthand way to run all of them is `cucumber features/` which runs all `.feature` files in the given directory.

Submission Instructions

When all scenarios are passing, deploy to Heroku and submit the URL of your deployed game in a file named `sinatra-url.txt`. You can create your text file using the `echo` command like this:

```
echo 'my-app-12345.herokuapp.com' > sinatra-url.txt
```

You would of course change 'my-app-12345' to match your Heroku URL.

You also need to submit hangperson_game.rb file as well.

Conclusion

This assignment has served as a microcosm or miniature tour of the entire course: during the rest of the course we will investigate each of these in much more detail, and we will also add new techniques---

- *Test-driven development (TDD)* will let you write much more detailed tests for your code and determine its **coverage**, that is, how thoroughly your tests exercise your code. We will use **RSpec** to do test-first development, in which we write tests before we write the code, watch the test fail ("red"), quickly write just enough code to make the test pass ("green"), clean up (refactor) the code, and go on to the next test. We will use the `autotest` tool to help us get into a rhythm of red--green--refactor. In this assignment we provided the specs for you; when designing your own app, you'll write them yourself.

- *Code metrics* will give us insight into the quality of our code: is it concise? Is it factored in a way that minimizes the cost of making changes and enhancements? Does a particular class try to do too much (or too little)? We will use **CodeClimate** (among other tools) to help us understand the answers. We can check both quantitative metrics, such as test coverage and complexity of a single method, and qualitative ones, such as adherence to the *SOLID Principles* of object-oriented design.

- *Refactoring* means modifying the structure of your code to improve its quality (maintainability, readability, modifiability) while preserving its behavior. We will learn to identify *antipatterns* -- warning signs of deteriorating quality in your code -- and opportunities to fix them, sometimes by applying *design patterns* that have emerged as "templates" capturing an effective solution to a class of similar problems.