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

Developing Wordguesser Using TDD and Guard

Goals: Use test-driven development (TDD) based on the tests we've provided to develop the game logic for Wordguesser, 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 WordGuesser game class, ready to be "wrapped" in SaaS using Sinatra.

Overview

Our Web-based word-guessing game will work 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/wordguesser_game_spec.rb`. It specifies behaviors that it expects from the class `lib/wordguesser_game.rb`. Initially, we have added `, :pending => true` to every spec, so when Autotest first runs these, you should see no test cases yet, and the report "0 examples, 0 failures."

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 1 examples, 1 failure.

The `describe 'new'` block means "the following block of tests describe the behavior of a 'new' WordGuesserGame instance." The `WordGuesserGame.new` 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 `wordguesser_game.rb` ?

▶ According to the tests in this `describe` block, what instance variables is a WordGuesserGame expected to have?

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 `wordguesser_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://randomword.saasbook.info/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.

Next: Part 2 - RESTful Thinking