**Introduction**

Often the hardest part of creating a full-stack web application is knowing where to start. In this lesson, we will use an outside-in development process. With this approach, we start to build our new functionality at the feature level.

Starting at the feature level of the pyramid (going outside-in) means you begin by writing tests that inform implementation of the code that a user’s browser renders. These tests involve the aspects of your project that users will see and interact with.

In this lesson:

* You will learn a few tools for writing feature-level tests.
* You will go through a few rounds of the TDD cycle at the feature level to build an application that renders user input.
* You will end the lesson “in the red,” with a failing feature-level test that you can only address by “dropping” to the server level.

If you want to continue the journey, you can pick up where you left off in the Codecademy course on testing the server.

# Feature Test Toolbelt

To write the most effective feature tests, we are going to employ a few additional tools. These tools are meant to support JavaScript testing. We will cover their uses in the scope of building a feature test that drives implementation.

### Chai

Node.js has a default assertion library that provides enough functionality to write basic test code. The [Chai testing library](http://chaijs.com/) extends the types of assertions we can make.

Chai is an assertion library for Node.js and browsers that can be paired with any JavaScript testing framework.

### PhantomJS

[PhantomJS](http://phantomjs.org/) is a headless browser scriptable with a JavaScript API, which allows us to write tests that mimic user interaction and then evaluate the results. It does not require us to render the application in a browser.

A browser runs “headless” when it doesn’t render anything to the screen, but runs in the background.

### WebdriverI/O

[WebdriverIO](http://webdriver.io/guide/getstarted/modes.html#The-WDIO-Testrunner) provides methods that allow us to programmatically interact with the user-facing elements of our app in the headless browser that PhantomJS runs.

### Toolbelt High-Level Summary

Phantom allows us to run an instance of a headless browser so you can run tests that mimic user interaction with a web application. WebdriverIO provides the methods to interact with browser values programmatically. We can make assertions against these tests using the Chai assertion library.

**Instructions**

In your **user-visits-root-test.js** file notice the first line of code, which makes Chai’s assertion library available for us to use in our tests.

const {assert} = require("chai")

# Feature Test I

Feature tests exercise behavior by simulating a user navigating the application in a web browser.

Imagine we wanted to create a simple web-based poetry writing application.

The first feature test we want to write is to check our application’s empty state. The functionality we want to test is:

* When a user visits the homepage, the poems section is empty

We want to make sure that when there are no poems in the database, there are no poems rendered on the homepage. This is the application’s empty state.

The testing suite for our poetry app would begin with nested describe blocks like this:

describe('Poetry web app', () => {

describe('user visits root', () => {

});

});

The term ‘root’, refers to our application’s entry point, which in this example is the home page that users will visit in their browser.

Next, we add an it block to describe the behavior we want to test in our app:

describe('Poetry web app', () => {

describe('user visits root', () => {

it('page starts blank', () => {

});

});

});

When a user visits the root of our app, they should have a blank page on which to write their own poem.

#### The Plumbing

Next, we reach for our feature testing toolbelt. We start by to using the global browser variable that is provided by WebdriverI/O.

The browser variable is powerful because it gives us access to the browser that Phantom is running in the background. We can simulate a user interacting with our website by calling different methods on the global browser variable in our test suite.

For example, we can use browser.url() to simulate a user visiting the home page of our application, which is the first behavior we want to test.

The .url method navigates to the URL that is passed to it as an argument. The following line of code would navigate to the Codecademy website in the Phantom browser.

browser.url('https://www.codecademy.com')

In the case of our poetry web app, we will pass in '/' as the argument, which will point the browser to the root file of our project, which in this case is our **index.html**.

The code would look like this:

describe('poetry web app', () => {

describe('user visits root', () => {

it('page starts blank', () => {

browser.url('/');

});

});

});

**Instructions**

**1.**

Imagine you are a developer working on a project that includes creating a web application with a message feature.

The first feature you want to test in your web app is that no messages appear on the page when a user visits the project root.

* Inside the describe blocks in **user-visits-root-test.js** file, write an it block with the string: starts blank, and an empty callback function.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Inside the it block, call the .url() method on the global browser variable and pass '/' as the argument.

const {assert} = require('chai');

describe('User visits root', () => {

  describe('without existing messages', () => {

    it('starts blank', () => {

      browser.url('/')

    });

  });

});

**Feature Test I: Assert**

The last thing our test needs is an assert statement to verify that the behavior we expect is equal to the actual behavior of our code.

We want to make sure our app is in an *empty state*.

We can write a test for this behavior by deciding that poems will be listed in an HTML element with an id attribute set to poems. Then, write an assert statement to verify that the element with the ID poems is empty.

We can do this using the Chai assert.equal method, which evaluates if the two arguments are equal.

In the case of our poetry app, the assert statement would look like this:

assert.equal(browser.getText('#poems'), '')

Because we will render the poetry onto the page as text, we can evaluate the contents of the HTML element as a string.

The .getText method, from WebdriverI/O, gets the text content from the selected DOM element.

Here we are using browser.getText() to evaluate if the text in the element with the ID poems is equal to an empty string.

Our final code for this feature test would look like this:

describe('User visits root', () => {

describe('without existing poems', () => {

it('page starts blank', () => {

browser.url('/');

assert.equal(browser.getText('#poems'), '');

});

});

});

**Instructions**

**1.**

Use assert.equal() to evaluate if an element with ID messages:

<section id="messages"></section>

has no text in it.

Then run your test suite to celebrate being in the red!

Check your work.

Hint

* The first argument in your assert statement should use browser.getText(), and the selector for a DOM element with the ID messages.
* The second argument should be an empty string.

const {assert} = require('chai');

describe('User visits root', () => {

  describe('without existing messages', () => {

    it('starts blank', () => {

      browser.url('/');

      assert.equal(browser.getText('#messages'), '')

    });

  });

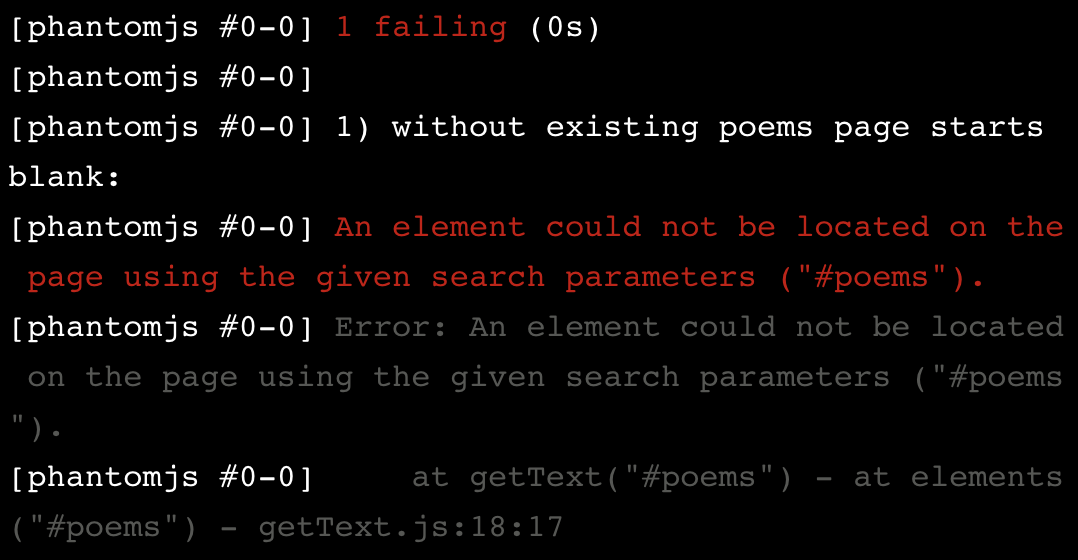
});

# Feature Test I: Passing

Now that we have written our first test with an assert statement, we will run the test and use the error message to drive the next step in our development process.

When executing a feature test that fails, errors will have messages that discuss the failure in terms of HTML (i.e. that text or button that you said would be on the page isn’t on the page) or HTTP (i.e. the request that this page made resulted in a 404 HTTP status because the route you requested didn’t exist).

Considering our poetry web app example, if we ran our test suite we would get an error message that included:

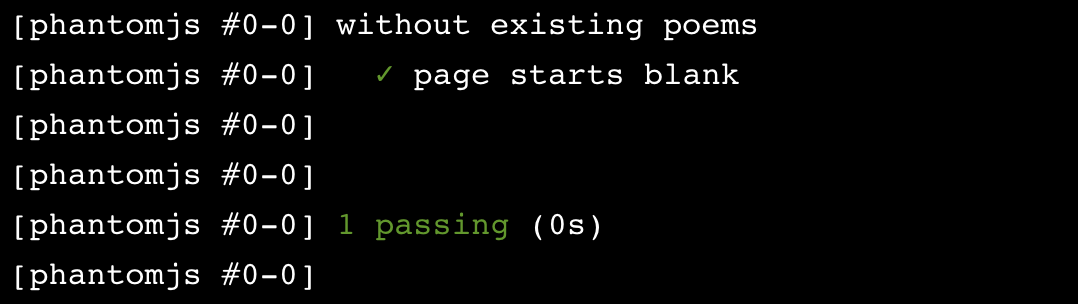


The error message describes the issue in terms of HTML elements and tells us that the element we are expecting does not exist on our page. This is because we have not yet created the HTML in our **index.html** file.

Using a strict TDD approach, we would write just enough HTML code to make our test pass. Let’s do that now.

<section id="poems"></section>

When we run our tests we get a message confirming that it is passing.



We have written our first feature test and moved from the red to the green using a TDD approach.

We expected this test to pass because we haven’t created any poems as part of our test’s setup. Each test is discrete and isolated from the other parts of the project, so we know that the list of poems will be empty.

**Instructions**

**1.**

Run your test suite in the terminal.

Go to **index.html** and create a <section> </section> element that responds to the error message you received when running your test suite.

Run your test suite again in the terminal to make sure you are in the green.

Hint

You need to create a section element with the ID messages.

const {assert} = require('chai');

describe('User visits root', () => {

  describe('without existing messages', () => {

    it('starts blank', () => {

      browser.url('/');

      assert.equal(browser.getText('#messages'),'');

    });

  });

});

<section id="messages"> </section>

# Feature Test II: Setup

Now that we are in the green we should take a moment to consider refactoring our code.

Our code is pretty basic and succinct. We won’t refactor it but will take a moment to consider our current code implementation.

For more on the refactor phase, you can try the Codecademy [Learn Test Driven Development With Mocha](https://www.codecademy.com/courses/learn-mocha-tdd/lessons/tdd-with-mocha/exercises/introduction) Course.

#### Feature Test II

Returning to our poetry app demo, we want to write a test to check if the application saves the title and text of a user’s poem when they press the submit button.

The functionality we want to test is:

1. The user enters text into a text input element (the poem)
2. The user enters text into a second text input element (the title of the poem)
3. The user presses a submit button

Adding the describe and it blocks for this second test would look like this:

describe('demo poetry web app', () => {

it('saves the user poem and title', () => {

});

});

Next, we want to write the setup, exercise, and verification phases of our test.

In the setup phase for this test, we create variables to represent a user’s input to the title and poem fields on the home page.

const title = 'Words Birth Worlds';

const poem = 'Our words are marvelous weapons with which we could behead the sun';

The second test would now look like this:

describe('demo poetry web app', () => {

it('saves the user poem and title', () => {

const title = 'Words Birth Worlds';

const poem = 'Our words are marvelous weapons with which we could behead the sun';

});

});

**Instructions**

**1.**

Go to your test suite for the message app in the **user-visits-root-test.js** file. Notice that we have added the describe and it blocks for your second test. Inside those blocks:

* Create a const variable named message to hold the expected value of the message input, and set it equal to a string with the value:

feature tests often hit every level of the TDD Testing Pyramid

* Below message, create a const variable named author to hold the expected value of the author input, and set it equal to username.

const {assert} = require('chai');

describe('User visits root', () => {

  describe('without existing messages', () => {

    it('starts blank', () => {

      browser.url('/');

      assert.equal(browser.getText('#messages'),'');

    });

  });

  describe('posting a message', () => {

    it('saves the message with the author information', () => {

      const message ='feature tests often hit every level of the TDD Testing Pyramid';

      const author = 'username';

    });

  });

});

# Feature Test II: Exercise

The next step for our poetry web app is to use our browser variable for the exercise phase of the test.

First, we will set the URL of the browser to go to the root of our project using the .url method:

browser.url('/');

Next, we will use the .setValue method, which sends a sequence of keystrokes to an element, based on a string argument.

We will use .setValue() to mimic a user entering the title and poem into the corresponding HTML input elements at the root of our web app.

The first argument passed to .setValue() is the CSS selector that references an HTML element, and the second argument is the value you want to assign that element.

browser.setValue('input[id=title]', title);

browser.setValue('textarea[id=poem]', poem);

In the example above, a text input with the ID of title will be set to a value of title. Also, the textarea with ID poem will be set to the value poem. The variables referenced here are the ones we created in the setup phase.

To complete the exercise phase of our test we would use the .click method to mimic a user clicking on a submit button.

browser.click('input[type=submit]');

Our second test, with the setup and exercise phases, now looks like this:

describe('demo poetry web app', () => {

it('saves the user poem and title', () => {

// Setup

const title = 'Words Birth Worlds';

const poem = 'Our words are marvelous weapons with which we could behead the sun';

// Exercise

browser.url('/');

browser.setValue('input[id=title]', title);

browser.setValue('textarea[id=poem]', poem);

browser.click('input[type=submit]');

});

});

**Instructions**

**1.**

Write the exercise phase of your second feature test for the message web app.

Use the browser.url() method to set the URL of the headless browser to your project root.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

Use browser.setValue() to set the value of an HTML <input> element with the ID author to the value of your variable named author.

Under the line you just wrote, use browser.setValue() to set the value of an HTML <textarea> element with the ID message to the value of your variable named message.

Checkpoint 3 Passed

**3.**

Use browser.click() to simulate a user clicking on an HTML <input> with the type submit.

const {assert} = require('chai');

describe('User visits root', () => {

  describe('without existing messages', () => {

    it('starts blank', () => {

      browser.url('/');

      assert.equal(browser.getText('#messages'),'');

    });

  });

  describe('posting a message', () => {

    it('saves the message with the author information', () => {

      const message ='feature tests often hit every level of the TDD Testing Pyramid';

      const author = 'username';

      browser.url("/")

      browser.setValue('input[id=author]', author);

      browser.setValue('textarea[id=message]', message);

      browser.click('input[type=submit]');

    });

  });

});

**Feature Test II: Verify**

Now that we have programmed the behavior we want to test in the exercise phase, we will write the verification phase of our test. We will compare the actual results of exercising the code with the expected results.

In the case of our poetry app, we want to verify that once a user submits a poem, the section of the app’s webpage that we have decided will display the poems includes that poem.

We have created that element already to make our first feature test pass. It is the following line of code in our **index.html** file:

<section id="poems">

</section>

To add an assert statement to evaluate the behavior of our feature, we will use the browser variable, and .getText() to return the text contents of the element, with the id poem.

The Chai Assertion Library allows us to use the .include method to check if the string that is returned from .getText() includes the substrings of the title and poem that the user has submitted:

assert.include(browser.getText('#poems'), title);

assert.include(browser.getText('#poems'), poem);

In both assert statements the first argument we pass to .include() is the function we created above it.

The .include() method works like this:

assert.include(haystack, needle)

The full second test would now look like this:

const {assert} = require('chai');

describe('User visits root', () => {

describe('demo poetry web app', () => {

it('saves the user poem and title', () => {

// Setup

const title = 'Words Birth Worlds';

const poem = 'Our words are marvelous weapons with which we could behead the sun;

// Exercise

browser.url('/');

browser.setValue('input[id=title]', title);

browser.setValue('textarea[id=poem]', poem);

browser.click('input[type=submit]');

// Verify

assert.include(browser.getText('#poems'), title);

assert.include(browser.getText('#poems'), poem);

});

});

});

**Instructions**

**1.**

Verify your results.

* Write an assert statement that verifies that the value of your variable author is included in the results of your .getText function.
* Write an assert statement that verifies that the value of your variable message is included in the results of your .getText function.

Hint

Assuming that you are testing if needle is included in haystack;

the .include() method works like this:

assert.include(haystack, needle)

You can use browser.getText('#messages') to get the text contents of an HTML element

const {assert} = require('chai');

describe('User visits root', () => {

  describe('without existing messages', () => {

    it('starts blank', () => {

      browser.url('/');

      assert.equal(browser.getText('#messages'),'');

    });

  });

  describe('posting a message', () => {

    it('saves the message with the author information', () => {

      const message ='feature tests often hit every level of the TDD Testing Pyramid';

      const author = 'username';

      browser.url('/');

      browser.setValue('input[id=author]', author);

      browser.setValue('textarea[id=message]', message);

      browser.click('input[type=submit]');

      assert.include( browser.getText('#messages'), message);

      assert.include( browser.getText('#messages'), author);

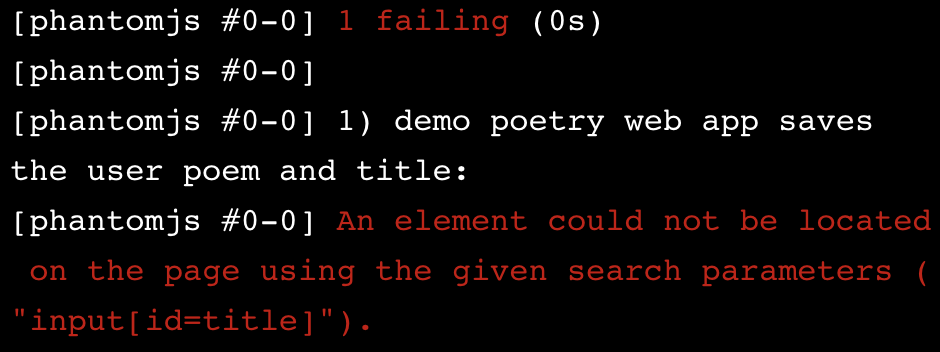
    });

  });

});

**Stuck In The Red**

Now that we have written our second test, it is time to write the minimal implementation code to move us forward in the TDD process. Running the test suite for our poetry web app at this point would give us this error message:



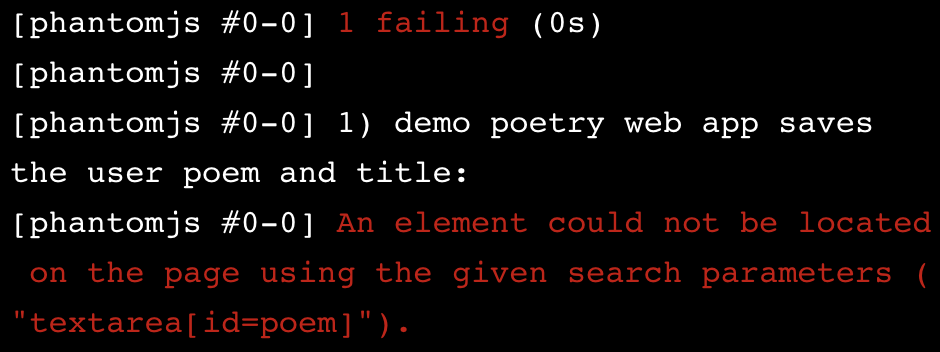
The error message describes the HTML issue that is preventing our test from continuing.

To address this error message, we would create an <input> with the ID, title in our **index.html** file. It would look like this:

<label for="title">Title</label>

<input id="title">

Now when we run our test, we will get a step further and receive an error message that tells us the next line of HTML code we need to write:



This error message tells us we are missing a <textarea> element with the ID, poem. We can address this by adding the following to our **index.html**:

<label for="poem">Your poem:</label>

<textarea id="poem"></textarea>

Running the test again would give us a similar error message concerning the input element with the type equal to submit. This is the submit button referenced in our test code, and we can address this error by adding the following code to our **index.html** file:

<input type="submit">

The complete **index.html** file would now look like this:

<section id="poems">

</section>

<label for="title">Title</label>

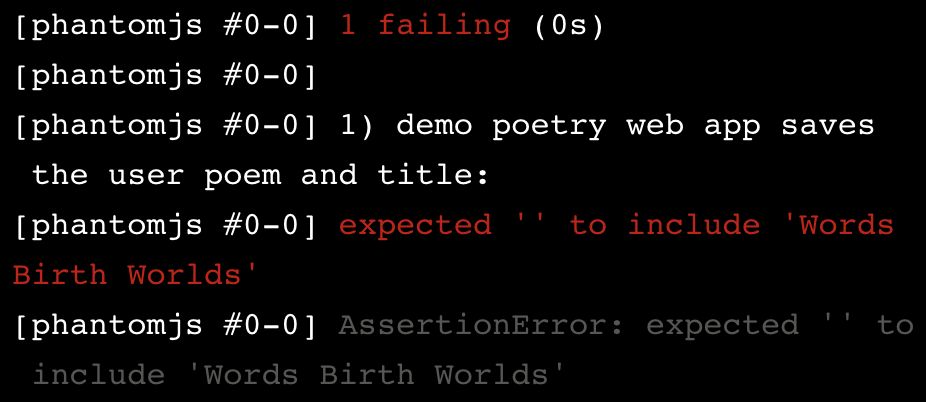
<input id="title">

<label for="poem">Your poem:</label>

<textarea id="poem"></textarea>

<input type="submit">

Running the test suite now would give us an error message like this:



While this error message looks similar to the ones we have been seeing, it is a different type of error message, and it signals the need for a shift in our TDD process.

What’s different here is that the failure comes from the *verification* phase instead of the *exercise* phase. While this isn’t always the case, that means that we’ve changed the implementation code enough to get to the part of the test where we’re specifying behavior, not just the existence of elements.

The kind of test we need to write in response to this error will force us to drop levels in the TDD Testing Pyramid.

**Instructions**

**1.**

Run your test and follow the error messages to address one issue at a time, until you receive an error concerning the *verification* phase of your test. That error that will force you to drop down to a server level test.

* Create an <input> element with the ID, author.
* Check your work
* Run your test suite

Checkpoint 2 Passed

Hint

Make sure the spacing of your HTML code is correct.

**2.**

Create a <textarea> element with the ID, message,

* Check your work
* Run your test suite

Checkpoint 3 Passed

**3.**

Create an <input> element with the type, submit.

* Check your work
* Run your test suite

<section id="messages"></section>

<label for="author">Your name:</label>

<input id="author">

<label for="message">Your message:</label>

<textarea id="message"></textarea>

<input type="submit">

# Review

Congratulations! You’ve built tests for a feature.

You will notice a couple changes to your files and the addition of a web browser. First, let’s take a look at your **index.html** file. You will see we added some tags to the file and added an <h1> element with a title for our webpage.

We also linked to a **style.css** file that will provide some basic styling to the code you have written, so that we can see what our messaging app might look like in a live browser! To view your app, run npm start in the terminal component and then refresh the browser on the right.

Take a look at the **style.css** file in the editor. Feel free to add and change styles in the **style.css** file to customize your web application’s UI.

After making any changes to your **style.css** file, run npm start in your terminal, and refresh your browser to see the changes.

In this lesson, we have been writing tests at the top level of the TDD Testing Pyramid, which concerns the part of the app that users interact with. To make our feature test pass and get back into the green we need to drop down to the server level — which concerns the part of the application that makes ‘POST’ requests to the server.

#### In this lesson you learned:

* When developing a new feature and practicing outside-in development, feature tests are where we’ll typically start.
* Feature tests often incorporate every layer of the application and — using WebDriverI/O and Mocha — exercise features in the same way that a human user would. They’re a good tool for reproducing end-user behavior.
* WebDriverI/O is a Node package that interacts with a “headless” instance of PhantomJS.
* Feedback from feature tests is usually in terms of HTML (i.e. that text or button that you said would be on the page isn’t on the page).
* Because feature tests typically hit every layer of a developer’s stack, they are slower than tests at lower layers, and errors thrown in feature tests can be difficult to interpret and provide little guidance on what the developer can do to resolve them.
* Their value, however, is in developer confidence that the software functions as expected.