**Introduction**

Imagine checking your bank account online. You have $1,000. The website is updated overnight, and you check again in the morning. Your balance is $257.43. Where did your money go? Is that truly your balance?

You report this to customer service. Thousands of other users report similar issues. Customers close accounts.

Back at the bank a software error is found to be the cause. The bank’s developers did not run tests on the software before deploying it to users. Money did not vanish but its amounts were printed incorrectly to the website.

Errors in software are inevitable. Unchecked, these errors can have painful and costly impacts on users and developers. In 2002, a study commissioned by the US Department of Commerce’s National Institute of Standards and Technology concluded that software errors cost the US economy about $59 billion annually.

To avoid those costs, software professionals use *automated testing*. During and after production, they can run an *automated test suite* to give themselves confidence that their products are free of errors and work as expected.

This lesson will give you the knowledge and practice to discuss these concepts. By the end of this lesson you will be able to:

* Define an *automated test suite*
* Describe how a test suite is used in software development
* Explain the benefits of *automated testing*

\*Study available here: [*Software Errors Cost U.S. Economy $59.5 Billion Annually*, 2002](https://web.archive.org/web/20090610052743/http:/www.nist.gov/public_affairs/releases/n02-10.htm).

**Instructions**

In this exercise you have access to a terminal and browser. You can start, stop, and test a web application from here.

1. Type npm start into the terminal and hit Enter.
2. Reload the page in the browser.

As a user of the Cake Bar site you suspect there are some errors.

* Try to enter your name and click the order button. Your name should be populated at the bottom of the page (you may have to scroll down). Does it work as expected?
* Try to pick a cake type and order. Does this feature work as expected?
* Continue interacting with the website. Can you find any other errors?

**Manual Testing**

*Software testing* is the process of assessing the completeness and quality of computer software. Usually this is done by running a part of a system (like a web application) and comparing the actual behavior to the expected behavior.

One way to perform software testing is *manual testing*. Manual testing is a form of testing done by a human interacting with a system. With web apps, this might be clicking, dragging, and typing through a webpage. A list of actions and expected behaviors would be given. If the observed behavior doesn’t match the expected behavior, the application has an error.

Errors, like the ones you may have found in the provided web app, are also called *bugs*. A bug is an error, fault, or flaw in software that makes a system behave in unexpected ways. As you read in the last exercise, these unexpected behaviors can cause harm to users. Ideally testing catches bugs before they are sent to users.

**Instructions**

The same application is provided. As a manual test engineer for this app, you have been provided some features to test. Manually test the web app by following the list provided:

* **Behavior 1**: Run npm start in the terminal and reload the page. Scroll down and expect the webpage to start with a blank order.
* **Behavior 2**: Type a name and click “Place Order”. Scroll down and expect “Deliver to:” to display the submitted name.
* **Behavior 3**: A blank name will not overwrite the existing name.
* **Behavior 4**: Select a cake type and place the order. Expect “Cake” to display the selected cake type.
* **Behavior 5**: Check some fillings. Expect “Fillings” to display your selected fillings.
* **Behavior 6**: Choose a stack size. Expect “Pancake Count:” to display the number equivalent to the stack size, e.g. “Double” is “2”.

**Automated Testing**

How long did it take to manually test the application in the previous exercise? If you repeated the process 100 times, how often do you think you would make a mistake?

In a company, someone must be paid to do that work, so every hour of manual testing has a cost. The cost of testing can be reduced and the quality can be improved with *automated testing*.

*Automated testing* is the use of software to control the execution of tests and the comparison of actual behavior to expected behavior. All the testing you just did (and more) could be performed by a computer program.

Compared to manual testing, automated testing is

* Faster: it tests more of your product in less time.
* More reliable: it’s less prone to error than a human is .
* Maintainable: you can review, edit, and extend a collection of tests.

Rather than hire a testing team at the end of development, professional developers can run their automated tests after every change. The workflow might look like this:

1. Write code and corresponding tests
2. Enter a command into a terminal to run tests
3. If the app behaves as intended, all tests should pass. Development is complete.
4. If it does not behave as intended, at least one test should fail. Fix code and return to step 2.

**Instructions**

**1.**

This web app is set up with automated tests. Run them by entering npm test in the terminal.

Wait for the test output to appear.

The tests align with the expected behaviors in the previous exercise. Which method caught more bugs? Which was faster?

const {assert} = require('chai');

describe('User visits index', () => {

  describe('to post an order', () => {

    /\*

     \* Run `npm start` in the terminal and reload the page. Expect the webpage to be visible and the order form to be empty.

     \*/

    // Edit the line below

    it('starts with an empty order (Behavior 1)', () => {

      browser.url('/');

      assert.equal(browser.getText('#deliver-to span'), '');

      assert.equal(browser.getText('#cake-type span'), '');

      assert.equal(browser.getText('#fillings span'), '');

      assert.equal(browser.getText('#size span'), '');

    });

    /\*

     \* Type a name and click "Place Order". Expect "Deliver to:" to display the submitted name. (You may need to scroll down.)

     \*/

    it('displays the submitted name (Behavior 2)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    it('does not overwrite name if blank name submitted (Behavior 3)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    /\*

     \* Select a cake type and place the order. Expect "Cake" to display the selected type.

     \*/

    it('displays the selected cake type (Behavior 4)', () => {

      const cakeType = 'Whole Wheat';

      browser.url('/');

      browser.click('#whole-wheat');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#cake-type'), cakeType);

    });

    /\*

     \* Check some fillings. Expect "Fillings" to display your selection.

     \*/

    it('displays multiple fillings (Behavior 5)', () => {

      const firstChoice = 'Strawberries';

      const secondChoice = 'Banana';

      browser.url('/');

      browser.click('#strawberries');

      browser.click('#banana');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#fillings'), firstChoice);

      assert.include(browser.getText('#fillings'), secondChoice);

    });

    /\*

     \* Choose a stack size. Expect "Pancake Count:" to display the number equivalent to the stack size, e.g. "Double" is "2".

     \*/

    it('displays the number equivalent to the stack size (Behavior 6)', () => {

      const optionText = 'Septuple Stack';

      const optionNum = '7';

      browser.url('/');

      browser.selectByVisibleText('#select-stack', optionText)

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#size'), optionNum);

    });

  });

});

**The Test Suite**

Tests are written with code, just like the rest of your web app. You can refer to the code defining your app as *implementation code*, and the code defining your tests as *test code*.

A collection of tests for a web application is called a *test suite*. In the last exercise, you ran a test suite with npm test. In that case the test suite contained all tests for the application.

Test code is included with and structured similarly to implementation code. Often times changes to test code are associated with changes to implementation code and vice versa. Both are easier to maintain when they are stored in the same place.

For example, if implementation code is written in index.js then the corresponding test code may be written in index-test.js.

**Instructions**

**1.**

All bugs have been fixed!

* Confirm that by running the test suite (this may take some time).
* Check your work once you see that all tests pass.

Checkpoint 2 Passed

Hint

Run npm test in the terminal. The correct output must be shown in the terminal to pass this checkpoint.

**2.**

In the test output, find the string starts with a blank order. That string defines part of the “User visits index” feature in the application, so find it in the **test/features/user-visits-index-test.js** test file.

Around line 9, change starts with a blank order to starts with an empty order in the test file.

Checkpoint 3 Passed

**3.**

Run npm test.

Check your work once you see starts with an empty order in your test output.

Checkpoint 4 Passed

Hint

Run npm test in the terminal, review the output, then check your work.

**4.**

That test is interacting with the index webpage, which is controlled by **index.handlebars**. Add some text to make the order NOT empty.

At the bottom of **index.handlebars**, around line 122, add your name next to {{order.name}}. It should look like this:

<h2 id="deliver-to">deliver to: <span>My-Name{{order.name}}</span></h2>

Checkpoint 5 Passed

**5.**

Run the test suite. Your change in **index.handlebars** affected the outcome of tests in **features/user-visits-index-test.js**! When you make the order NOT empty, the starts with an empty order test fails. (You can scroll up the terminal to see more information.)

Checkpoint 6 Passed

Hint

Run npm test in the terminal, review the output, then check your work.

const {assert} = require('chai');

describe('User visits index', () => {

  describe('to post an order', () => {

    /\*

     \* Run `npm start` in the terminal and reload the page. Expect the webpage to be visible and the order form to be empty.

     \*/

    // Edit the line below

    it('starts with an empty order (Behavior 1)', () => {

      browser.url('/');

      assert.equal(browser.getText('#deliver-to span'), '');

      assert.equal(browser.getText('#cake-type span'), '');

      assert.equal(browser.getText('#fillings span'), '');

      assert.equal(browser.getText('#size span'), '');

    });

    /\*

     \* Type a name and click "Place Order". Expect "Deliver to:" to display the submitted name. (You may need to scroll down.)

     \*/

    it('displays the submitted name (Behavior 2)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    it('does not overwrite name if blank name submitted (Behavior 3)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    /\*

     \* Select a cake type and place the order. Expect "Cake" to display the selected type.

     \*/

    it('displays the selected cake type (Behavior 4)', () => {

      const cakeType = 'Whole Wheat';

      browser.url('/');

      browser.click('#whole-wheat');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#cake-type'), cakeType);

    });

    /\*

     \* Check some fillings. Expect "Fillings" to display your selection.

     \*/

    it('displays multiple fillings (Behavior 5)', () => {

      const firstChoice = 'Strawberries';

      const secondChoice = 'Banana';

      browser.url('/');

      browser.click('#strawberries');

      browser.click('#banana');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#fillings'), firstChoice);

      assert.include(browser.getText('#fillings'), secondChoice);

    });

    /\*

     \* Choose a stack size. Expect "Pancake Count:" to display the number equivalent to the stack size, e.g. "Double" is "2".

     \*/

    it('displays the number equivalent to the stack size (Behavior 6)', () => {

      const optionText = 'Septuple Stack';

      const optionNum = '7';

      browser.url('/');

      browser.selectByVisibleText('#select-stack', optionText)

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#size'), optionNum);

    });

  });

});

# Tests As Documentation

Imagine explaining this Cake Bar app to someone else. How does it behave? Does it rely on other software? How do you run it on a computer? You could read every line in every file to figure that out. Or you could read the documentation.

Documentation is any content separate from implementation code that explains how it works or how to use it. It may provide more concise summaries and explanation than the implementation code can.

Documentation can come in many forms, including plain text, diagrams…and tests! Tests as documentation provide what many other forms cannot: both human-readable text to describe the application and machine-executable code to confirm the app works as described.

This code block from the Cake Bar app describes and tests the “name” functionality.

it('accepts the customer name', () => {

const name = 'Hungry Person';

browser.url('/');

browser.setValue('#name', name);

browser.click('#submit-order');

browser.url('/');

assert.include(browser.getText('#deliver-to'), name);

});

You can read the description in plain English terms: it accepts the customer name. You can run the test to confirm the functionality works as described.

**Instructions**

**1.**

Copy and paste the below code into the **test/features/user-visits-index-test.js** file. This part of the test explains the functionality in easy-to-read text. When you’re done, click Run.

Paste this test structure at the bottom of the file, near line 91.

describe('to clear an order', () => {

it('deletes the selected options', () => {

});

});

Checkpoint 2 Passed

**2.**

Copy and paste the body of the test within that block (underneath the line containing it). This part of the test will confirm that the clear order button deletes the selected options. When you’re done, click Run.

const name = 'Indecisive Person';

const time = '10:00';

browser.url('/');

browser.setValue('#name', name);

browser.selectByVisibleText('#select-pickUp', time)

browser.click('#submit-order');

browser.click('#clear-order');

browser.url('/');

assert.equal(browser.getText('#deliver-to'), '');

assert.equal(browser.getText('#cake-type'), '');

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

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

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

Checkpoint 3 Passed

**3.**

Run the test suite. You should see the new test fail. By running the suite you’ve confirmed that the application does not behave as expected!

Checkpoint 4 Passed

Hint

Run npm test in the terminal.

**4.**

Test this manually. Run npm start in the terminal, reload the page, and see that the clear order button doesn’t exist yet!

const {assert} = require('chai');

describe('User visits index', () => {

  describe('to post an order', () => {

    /\*

     \* Run `npm start` in the terminal and reload the page. Expect the webpage to be visible and the order form to be empty.

     \*/

    it('starts with a blank order (Behavior 1)', () => {

      browser.url('/');

      assert.equal(browser.getText('#deliver-to span'), '');

      assert.equal(browser.getText('#cake-type span'), '');

      assert.equal(browser.getText('#fillings span'), '');

      assert.equal(browser.getText('#size span'), '');

    });

    /\*

     \* Type a name and click "Place Order". Expect "Deliver to:" to display the submitted name. (You may need to scroll down.)

     \*/

    it('displays the submitted name (Behavior 2)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    it('does not overwrite name if blank name submitted (Behavior 3)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    /\*

     \* Select a cake type and place the order. Expect "Cake" to display the selected type.

     \*/

    it('displays the selected cake type (Behavior 4)', () => {

      const cakeType = 'Whole Wheat';

      browser.url('/');

      browser.click('#whole-wheat');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#cake-type'), cakeType);

    });

    /\*

     \* Check some fillings. Expect "Fillings" to display your selection.

     \*/

    it('displays multiple fillings (Behavior 5)', () => {

      const firstChoice = 'Strawberries';

      const secondChoice = 'Banana';

      browser.url('/');

      browser.click('#strawberries');

      browser.click('#banana');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#fillings'), firstChoice);

      assert.include(browser.getText('#fillings'), secondChoice);

    });

    /\*

     \* Choose a stack size. Expect "Pancake Count:" to display the number equivalent to the stack size, e.g. "Double" is "2".

     \*/

    it('displays the number equivalent to the stack size (Behavior 6)', () => {

      const optionText = 'Septuple Stack';

      const optionNum = '7';

      browser.url('/');

      browser.selectByVisibleText('#select-stack', optionText)

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#size'), optionNum);

    });

  });

  // Add "clear order" test here

  describe('to clear an order', () => {

  it('deletes the selected options', () => {

const name = 'Indecisive Person';

      const time = '10:00';

      browser.url('/');

      browser.setValue('#name', name);

      browser.selectByVisibleText('#select-pickUp', time)

      browser.click('#submit-order');

      browser.click('#clear-order');

      browser.url('/');

      assert.equal(browser.getText('#deliver-to'), '');

      assert.equal(browser.getText('#cake-type'), '');

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

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

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

  });

});

});

**Regression**

The new “clear order” button feature has been implemented to satisfy the new test.

When adding a new feature to your product, it’s possible that something will break. If that break occurs within a feature developed earlier, it is called *regression*. When functionality previously developed and tested stops working, you may say the *functionality regressed*.

Running an automated test suite is fast and repeatable, which means you can run tests after every change to confirm that old features still work. If they have regressed, the test output should notify you.

**Instructions**

**1.**

Code to implement the new “clear order” feature has been added. Start the app and manually test the new and old button features:

* Click “Clear”. Expect the order to be empty.
* Enter an order and click “Place Order”. Expect the order to be populated with your selections. Do both buttons behave as expected?

When you’re done, check work.

Checkpoint 2 Passed

Hint

Manually test by running npm start then reloading the page.

**2.**

Stop the app by pressing CTRL + C (the control button and C key at the same time), then run the test suite.

You can scroll up to see that the new feature test (“to clear an order”) is passing and the old tests are failing. You found a regression!

When you’re done, check work.

Checkpoint 3 Passed

Hint

Make sure to use CTRL + C to stop the app before running tests. Run npm test in the terminal to run the tests.

**3.**

Here’s the bug: The “clear” button is using /place-order instead of /clear-order, making both buttons clear the order!

Fix the regression, step 1: In **views/index.handlebars** around line 117, find the form

<form action="/place-order" method="post">

and replace it with

<form action="/clear-order" method="post">

Click Check Work when you’ve replaced the code.

Checkpoint 4 Passed

**4.**

Fix the regression, step 2: In **routes/index.js** around line 13, replace the first instance of

router.post('/place-order', async (req, res) => {

with

router.post('/clear-order', async (req, res) => {

Click Check Work when you’ve replaced the code.

Checkpoint 5 Passed

**5.**

Run the test suite again to confirm the fix works as expected.

If you don’t see 7 passing tests, double check that clear order is on the right line in each file!

When you’re done, check work.

Checkpoint 6 Passed

Hint

Run npm test in the terminal, review the output, then check your work.

  <div class="tracker">

    <div id="tracker-meter" class="tracker-meter">

      <div class="image-container">

        <img alt-text="bug-image" src="images/bug.svg">

      </div>

    </div>

    <div class="tracker-name">

      <p>BUG-O-METER</p>

    </div>

  </div>

  <div class="banner">

    <h1>CAKE BAR</h1>

  </div>

  <div class="body-container">

    <div class="menu">

      <form id="cake-form" action="/place-order" method="post">

        <div class="form-container">

          <div class="form-row">

            <h2>Enter your name</h2>

          </div>

          <div class="form-row">

            <div class="name-flex">

              <div class="name-col-1">

                <input type="text" id="name" name="name" value="">

              </div>

            </div>

          </div>

          <div class="form-row">

            <h2>Pick a cake type</h2>

          </div>

          <div class="form-row">

            <div class="input-col">

              <label for="plain">

                <input type="radio" id="plain" name="cakeType" value="Plain">Plain<br>

              </label>

            </div>

            <div class="input-col">

              <label for="whole-wheat">

                <input type="radio" id="whole-wheat" name="cakeType" value="Whole Wheat">Whole Wheat<br>

              </label>

            </div>

          </div>

          <div class="form-row">

            <h2>Add fillings</h2>

          </div>

          <div class="form-row">

            <div class="input-col">

              <label for="strawberries">

                <input type="checkbox" id="strawberries" name="fillings" value="Strawberries">Strawberries<br>

              </label>

              <label for="blueberries">

                <input type="checkbox" id="blueberries" name="fillings" value="Blueberries">Blueberries<br>

              </label>

              <label for="banana">

                <input type="checkbox" id="banana" name="fillings" value="Banana">Banana<br>

              </label>

              <label for="apple">

                <input type="checkbox" id="apple" name="fillings" value="Apple">Apple<br>

              </label>

            </div>

            <div class="input-col">

              <label for="macadamia-nuts">

                <input type="checkbox" id="macadamia-nuts" name="fillings" value="Macadamia Nuts">Macadamia Nuts<br>

              </label>

              <label for="sprinkles">

                <input type="checkbox" id="sprinkles" name="fillings" value="Sprinkles">Sprinkles<br>

              </label>

              <label for="chocolate-chips">

                <input type="checkbox" id="chocolate-chips" name="fillings" value="Chocolate Chips">Chocolate chips<br>

              </label>

              <label for="bacon">

                <input type="checkbox" id="bacon" name="fillings" value="Bacon">Bacon<br>

              </label>

            </div>

          </div>

          <div class="form-row">

            <div class="input-col">

              <h2>Choose a size</h2>

            </div>

          </div>

          <div class="form-row select-row">

            <div class="select-col">

              <div class="styled-select">

                <select id="select-stack" name="size">

                  <option id="single" value="1">Single Stack</option>

                  <option id="double" value="2">Double Stack</option>

                  <option id="triple" value="3">Triple Stack</option>

                  <option id="quadruple" value="4">Quadruple Stack</option>

                  <option id="quintuple" value="5">Quintuple Stack</option>

                  <option id="sextuple" value="6">Sextuple Stack</option>

                  <option id="septuple" value="7">Septuple Stack</option>

                  <option id="octuple" value="8">Octuple Stack</option>

                  <option id="nonuple" value="9">Nonuple Stack</option>

                  <option id="decuple" value="10">Decuple Stack</option>

                  <option id="centuple" value="100">Centuple Stack</option>

                </select>

              </div>

            </div>

          </div>

          <div class="form-row place-order-div">

            <input class="button" id="submit-order" type="submit" value="Place order">

          </div>

        </div>

      </form>

      <!-- Change the below action to fix the regression -->

      <form action="/clear-order" method="post">

        <div class="form-container">

          <div class="form-row place-order-div">

            <input class="button" id="clear-order" type="submit" value="Clear">

          </div>

        </div>

      </form>

    </div>

    <div class="order">

      <h2 id="deliver-to">deliver to: <span>{{order.name}}</span></h2>

      <h2 id="cake-type">cake: <span>{{order.cakeType}}</span></h2>

      <h2 id="fillings">fillings: <span>{{order.fillings}}</span></h2>

      <h2 id="size">pancake count: <span>{{order.size}}</span></h2>

    </div>

  </div>

**Review**

Writing automated tests takes time, but the cost is outweighed by the benefits. Automated testing

* Increases confidence that your product works as expected (compared to manual testing)
* Improves upon documentation
* Reduces the likelihood of regression

You also learned

* Where and why test code is stored alongside implementation code
* Terms to help communicate the benefit of testing: *manual testing*, *automated testing*, *test suite*, *bug*, *documentation*, and *regression*

**Instructions**

**1.**

Run the test suite to confirm the app is working as expected.

Checkpoint 2 Passed

Hint

Run npm test in the terminal.

**2.**

Start the app, reload the page, and enjoy a bug-free Cake Bar.

Checkpoint 3 Passed

Hint

Run npm start in the terminal, then reload the page.

const {assert} = require('chai');

describe('User visits index', () => {

  describe('to post an order', () => {

    /\*

     \* Run `npm start` in the terminal and reload the page. Expect the webpage to be visible and the order form to be empty.

     \*/

    it('starts with a blank order (Behavior 1)', () => {

      browser.url('/');

      assert.equal(browser.getText('#deliver-to span'), '');

      assert.equal(browser.getText('#cake-type span'), '');

      assert.equal(browser.getText('#fillings span'), '');

      assert.equal(browser.getText('#size span'), '');

    });

    /\*

     \* Type a name and click "Place Order". Expect "Deliver to:" to display the submitted name. (You may need to scroll down.)

     \*/

    it('displays the submitted name (Behavior 2)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    it('does not overwrite name if blank name submitted (Behavior 3)', () => {

      const name = 'Hungry Person';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.url('/');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#deliver-to'), name);

    });

    /\*

     \* Select a cake type and place the order. Expect "Cake" to display the selected type.

     \*/

    it('displays the selected cake type (Behavior 4)', () => {

      const cakeType = 'Whole Wheat';

      browser.url('/');

      browser.click('#whole-wheat');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#cake-type'), cakeType);

    });

    /\*

     \* Check some fillings. Expect "Fillings" to display your selection.

     \*/

    it('displays multiple fillings (Behavior 5)', () => {

      const firstChoice = 'Strawberries';

      const secondChoice = 'Banana';

      browser.url('/');

      browser.click('#strawberries');

      browser.click('#banana');

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#fillings'), firstChoice);

      assert.include(browser.getText('#fillings'), secondChoice);

    });

    /\*

     \* Choose a stack size. Expect "Pancake Count:" to display the number equivalent to the stack size, e.g. "Double" is "2".

     \*/

    it('displays the number equivalent to the stack size (Behavior 6)', () => {

      const optionText = 'Septuple Stack';

      const optionNum = '7';

      browser.url('/');

      browser.selectByVisibleText('#select-stack', optionText)

      browser.click('#submit-order');

      browser.url('/');

      assert.include(browser.getText('#size'), optionNum);

    });

  });

  describe('to clear an order', () => {

    it('deletes the selected options', () => {

      const name = 'Indecisive Person';

      const time = '10:00';

      browser.url('/');

      browser.setValue('#name', name);

      browser.click('#submit-order');

      browser.click('#clear-order');

      browser.url('/');

      assert.equal(browser.getText('#deliver-to span'), '');

      assert.equal(browser.getText('#cake-type span'), '');

      assert.equal(browser.getText('#fillings span'), '');

      assert.equal(browser.getText('#size span'), '');

    });

  });

});