**Introduction**

When testing an application, feature tests expose required tests or implementations at different levels of the application technology stack. One of those levels is the backend server.

Server tests are used to test the server response only, not any front-end rendering of code or user interactions. We “disconnect” the browser and interact directly with the server using requests. The tests define the expected behavior of the interactions and check the actual responses against what we expect.

Server tests are commonly used to test API responses, but we also use server tests for any server response that our application relies on. This can include checking status codes and error messages.

In this lesson, we introduce a suite of technologies and concepts for performing testing on a JavaScript-based server. These include Chai, jsdom, and SuperTest. We also review how to use async/await for asynchronous calls. When adapting this to your project, it needs to be tailored to match the specific technology stack for your project.

**Instructions**

**1.**

In the pane to the right, **user-visits-root-test.js** is pre-populated with feature-test code. Run the test by typing npm test in the terminal.

When you are ready to move on, check your work.

Checkpoint 2 Passed

Hint

Type npm test in the terminal.

**2.**

The test is currently failing — the test is attempting to verify that a message is being saved, but this behavior is not implemented by the server yet. This guides us to implement the code for a desired behavior.

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);

    });

  });

});

**Testing Framework: Chai**

When writing tests, sometimes you’ll find that the tests require calculation steps or inline code to determine if the test is passing. For example, to test if an array foo includes an element bar using Mocha with the built-in Node assertion library, we use the JavaScript includes helper:

assert.ok(foo.includes(bar));

To improve the readability and flow of our tests, we extend the built-in Node assertion library with Chai.

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

The main function in Chai we are using is .include(). This allows us to rewrite the previous example as:

assert.include(foo, bar);

Include also works to check that text contains certain values:

assert.include('foobar', 'bar'); // Evaluates to true

The large set of assertion methods in the chai library enable us to write more expressive tests that are easy for developers to understand.

**Instructions**

**1.**

In **chai-test.js** to the right, we’ve included Chai at the top of the file and set up a describe block with Mocha. Use Chai on line 9 to assert that the foo array contains the number 4. Use npm test to verify the test is passing.

When you are ready to move on, check your work.

Checkpoint 2 Passed

Hint

Use assert.include(). First argument is where you are looking (foo) and the second is what you are looking for (includedNumber).

**2.**

In JavaScript, the .pop() method removes the final element from an array and returns it. Write an assertion to verify that the variable fooPop returned from the .pop() method returns the correct element from the array. Use npm test to verify the test is passing.

When you are ready to move on, check your work.

Checkpoint 3 Passed

Hint

Use assert.equal() to check that fooPop is equal to includedNumber. Be sure to use fooPop as the first argument.

**3.**

Since we popped the only element from the array, foo should be empty. To check this, assert that the length of foo is now zero. Use npm test to verify the test is passing.

When you are ready to move on, check your work.

Checkpoint 4 Passed

Hint

Use assert.equal() to check that foo.length equals 0.

const {assert} = require('chai');

describe('Array', () => {

  describe('.pop()', () => {

    it('should return a value and remove the element from the array', () => {

      // setup

      const foo = [4];

      const includedNumber = 4;

      // check setup

      assert.include(foo, includedNumber);

      // exercise

      const fooPop = foo.pop();

      // asserts

      assert.equal(fooPop, includedNumber);

      assert.equal(foo.length, 0);

    });

  });

});

**Testing HTML Responses**

Our back-end server is serving dynamic HTML to the user. For the homepage, this is located in the **jsdom-test.js** file to the right. It is possible to use .include() to verify that the HTML response contains certain Strings, but gets cumbersome to verify the hierarchical relationships of DOM elements.

We can use the jsdom library to improve this type of assertion. It allows us to select elements of the DOM and check relationships and content. To increase the readability of our tests, we abstracted the jsdom functionality into a custom function, parseTextFromHTML:

const parseTextFromHTML = (htmlAsString, selector) => {

const selectedElement = jsdom(htmlAsString).querySelector(selector);

if (selectedElement !== null) {

return selectedElement.textContent;

} else {

throw new Error(`No element with selector ${selector} found in HTML string`);

}

};

This function takes the HTML response as a string and the desired selector as inputs and returns the textContent of the corresponding element. If no element is found, it will return a TypeError.

**Instructions**

**1.**

In the panel to the right, **jsdom-test.js** is prepopulated with code to test that the string “Hello” is contained within the HTML response. Run the test by typing npm test in the terminal and observe it pass.

When you are ready to move on, check your work.

Checkpoint 2 Passed

Hint

Type npm test in the terminal to the right and press enter.

**2.**

Change the existing assertion to use parseTextFromHTML and assert that the string “Hello” is contained in the #bar element. The first argument should be the HTML string, foo and the second argument should be the selector, '#bar'. Run the test using npm test.

When you are ready to move on, check your work.

Checkpoint 3 Passed

Hint

Pass the HTML string foo as the first argument into parseTextFromHTML and '#bar' as the second argument. Assert that this returns the string "Hello".

**3.**

Modify the HTML string foo to include the string “Hello” in the #bar element (in addition to the ‘#buzz’ element) to pass the test. Run the test using npm test to verify it passes.

When you are ready to move on, check your work.

Checkpoint 4 Passed

Hint

const foo = '<html><div id="bar">Hello</div><div id="buzz">Hello</div><html>';

const {assert} = require('chai');

const {jsdom} = require('jsdom');

const parseTextFromHTML = (htmlAsString, selector) => {

  const selectedElement = jsdom(htmlAsString).querySelector(selector);

  if (selectedElement !== null) {

    return selectedElement.textContent;

  } else {

    throw new Error(`No element with selector ${selector} found in HTML string`);

  }

};

describe('HTML tests', () => {

  describe('#bar', () => {

    it('should include the string "Hello"', () => {

      // setup

      const foo = '<html><div id="bar">Hello</div><div id="buzz">Hello</div><html>';

      //asserts

      assert.include(parseTextFromHTML(foo, '#bar'), 'Hello');

    });

  });

});

**Async / Await**

A server typically handles many requests at a time, but may be only capable of processing a subset of the requests concurrently. One side effect of this is that the server response time is neither instant nor predictable. If no other processes are occurring on the server, requests are handled quickly, but if the server is close to full capacity, the request can take a few seconds or even timeout.

We need a way to receive asynchronous responses from the server and then act on them. The async/await pattern introduced in Node 8 helps us write readable descriptions of the behavior of our application which is an important part of writing good tests.

To use this pattern, define the function with the async keyword. Then, within the function, use the await keyword in front of the asynchronous function you are calling. For example:

const foo = async () => {

console.log(await someAsyncThing());

return true;

}

foo();

Here, we are waiting for someAsyncThing() to return before logging the result to the console.

**Instructions**

**1.**

In **index-test.js** to the right, there is the start of a server test on the root document of our site. There are no assertions yet, but we are attempting to log the server response to the console. Run the test as is and note that we see an “undefined” response logged to the console. (The request method is covered in the next exercise)

When you are ready to move on, check your work.

Checkpoint 2 Passed

Hint

Use npm test in the terminal to run the test.

**2.**

Update the function to use async in the function definition and await for the call to request. Run the tests again using npm test and note the logged response in the console.

When you are ready to move on, check your work.

Checkpoint 3 Passed

Hint

Add async ahead of the () on the line starting with it... And add await before request(app).

const request = require('supertest');

const app = require('../../app');

describe('the homepage', () => {

  it('returns the correct content', async () => {

    const response = await request(app)

      .get('/')

      .send();

      console.log(response.text);

  });

});

**SuperTest**

As you may have noticed in the previous exercise, we are using the function request to make server calls to support our tests. This is actually a reference to the SuperTest library:

const request = require('supertest');

This library was specifically designed for testing Node server responses and integrates well with Mocha and Chai. To use SuperTest, we pass the app object from our app into the request function. To make a GET request, we use .get() with the desired route as the argument:

await request(app)

.get('/')

.send();

It is also possible to perform a POST using SuperTest. We chain any desired properties or inputs to the HTTP call, and use .send() to make the request:

await request(app)

.post('/messages')

.type('form')

.send({author, message});

**Instructions**

**1.**

In the pane to the right, there is the start to a server test on the root document of our site. Chain the .get() method at the end of the request. Pass the appropriate argument to get the root object of our app ('/').

Run npm test to verify the server response is being printed to the console.

When you are ready to move on, check your work.

Hint

Append .get('/') to the end of .request(app)

const request = require('supertest');

const app = require('../../app');

describe('the homepage', () => {

    it('returns the correct content', async () => {

        const response = await request(app).

        get('/');

        console.log(response.text);

    });

});

# Summary

In this lesson we covered a set of technologies used for testing a Node server. These included:

* Chai - a library for extending the built in Node assertion library
* jsdom - a library for interacting and testing the DOM returned by the server (this functionality is encapsulated in our parseTextFromHTML helper function).
* async / await - a pattern for making asynchronous code more readable
* SuperTest - a library for making Node server requests and testing their responses

**Instructions**

**1.**

In the pane to the right there is a test that makes use of all the technologies. Run the test using npm test and verify it passes. In the next lesson, we’ll use these concepts to further explore testing a Node server.

When you are ready to move on, check your work.

Hint

Type npm test in the terminal and press enter

const express = require('express');

const router = express.Router();

router.get('/', async (req, res) => {

  res.send('<h1 id="page-title">My Page</h1>');

});

module.exports = router;

# Introduction

In general, it is up to the developer to make a judgement call on how in depth to write a test. Every test written adds time to the testing cycle and can require maintenance if changes are made to the server behavior. For example, extensively testing failure cases at the feature level might be more than is needed if the error behavior can be fully tested and described at the server level.

As you develop an application, you may realize that you can replace feature tests or reduce them with equal coverage at a lower level. One question to ask when deciding between a full feature test versus a server test is:

“Is it worth trading a slow feature test for a faster server test that doesn’t test the UI?”

Based on the context of the different levels of testing, you should aim to pick the set of tests that gives you the best combination of reliable, complete and fast tests.

In this lesson, you will use server-level testing technologies to test-drive the development of an Express server.

Server tests often provide feedback in terms of HTTP domain concepts, like status codes, header keys and values, and the content of the response body. Let’s take a look at a feature-level test and compare it to a corresponding server test in **messages-test.js** to the right:

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

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

const author = 'user name';

const message ='feature testing with TDD makes me feel empowered to create a better workflow';

browser.url('/');

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

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

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

assert.include(messagesText(), message);

assert.include(messagesText(), author);

});

});

});

When such a test fails due to a non-existent server implementation, the developer needs to dive into the server level and begin the TDD process to drive the server solution.

**Instructions**

**1.**

In the panel to the right, there is a corresponding server test to illustrate the differences between feature and server level tests. Compare the describe block to the feature test above — what’s different?

When you are ready to move on, check your work.

Hint

The main difference is that we are not using the browser component to test anything. Assertions are directly using the server request and responses.

const {assert} = require('chai');

const request = require('supertest');

const {jsdom} = require('jsdom');

const app = require('../../app');

const parseTextFromHTML = (htmlAsString, selector) => {

    const selectedElement = jsdom(htmlAsString).querySelector(selector);

    if (selectedElement !== null) {

      return selectedElement.textContent;

    } else {

      throw new Error(`No element with selector ${selector} found in HTML string`);

    }

};

describe('when the Message is valid', () => {

    it('creates a new message', async () => {

      const author = 'user name';

      const message ='feature testing with TDD makes me feel empowered to create a better workflow';

      //save message

      const response = await request(app)

        .post('/messages')

        .type('form')

        .send({author, message});

      //check response to verify message is saved

      assert.include(parseTextFromHTML(response, '#messages'), message);

    });

});

# Status Codes

Server tests are slightly faster than browser-driven feature tests. Since the web browser is cut out of the test, we are not testing how things are rendered for the user. Instead, we are focused on the server response.

One use of TDD at the server level is to ensure that the HTTP status codes are returned as expected. Verifying status codes provide the most basic level of confidence that the server is functioning correctly. Having a test suite that includes status codes provides a quick check when implementing a new feature that we haven’t accidentally caused a request for valid routes to respond not authorized (401) or not found (404). (Full list of status codes at [httpstatuses.com](https://httpstatuses.com/))

To verify status codes, we are asserting that the response status is equal to the status code integer that our application requires:

assert.equal(response.status, 200);

If we use the “red, green, refactor” approach to implement our server behavior we would start out with an assertion like this and expect it to fail (“red”). We then implement the behavior to pass the test (“green”) and continue to refactor if needed, ensuring the test remains passing.

**Instructions**

**1.**

In **index-test.js** to the right, we started a server test for verifying the homepage returns a 200 (OK) status code. Add an assertion to check that the status code is indeed 200 and run the test using npm test.

When you are ready to move on, check your work.

Checkpoint 2 Passed

Stuck? Get a hint

**2.**

This test failed, but that’s good news! That means we’ve entered the “red” portion of the red, green, refactor approach. Use res.send() within the server implementation for this route in **index.js**. It should return an empty string when a request is made to the home route ('/'). Run the test using npm test and verify it now passes.

Checkpoint 3 Passed

Hint

Add res.send(''); in the response for the '/' route in **index.js**.

const {assert} = require('chai');

const request = require('supertest');

const app = require('../../app');

describe('root page', () => {

  describe('GET request', () => {

    it('returns a 200 status', async () => {

      const response = await request(app).

      get('/');

      assert.equal(response.status, 200);

    });

  });

});

const express = require('express');

const router = express.Router();

router.get('/', (req, res) => {

  res.send('');

});

module.exports = router;

# Refactoring: Route Parameters

In the previous exercise, we checked that the server responded with a specific message. On our home page, the title is constant for everyone, “Messaging App”.

What if we want to create a profile page that is customized for each user?

A straightforward implementation would be to generate hard coded routes for every single user of our app. Think: 'welcome/alice' => '<h1>Your Name is alice</h1>', 'welcome/bob' => '<h1>Your Name is bob</h1>', etc.

Hopefully if you see repetitive code like this, you’ll have an urge to refactor it to something more elegant using a variable route parameter. This allows us to put any username into the url and have the server generate the appropriate response. Think: 'welcome/:username' => '<h1>Your Name is ' + req.params.username +'</h1>'.

If you are using the red, green, refactor approach, you will start with a set of passing (“green”) assertions for the section of code you are looking to improve. With the current behavior captured, you can begin refactoring, knowing that your tests will “catch” you by turning red if you miss something in your approach.

**Instructions**

**1.**

In **profile.js** to the right, we have an implementation to provide customized profile messages for each of our users (alice and bob). Through TDD, we also have a set of assertions in **profile-test.js** to verify we are providing the correct messages to each user.

Run npm test to verify we are green.

When you are ready to move on, check your work.

Checkpoint 2 Passed

Hint

Type ‘npm test’ in the terminal to run the tests.

**2.**

In **profile.js**, replace the string '/alice' in the first profile route with '/:username' to make the username available as a variable.

Run npm test to see how this affected our test.

When you are ready to move on, check your work.

Checkpoint 3 Passed

Hint

Replace '/alice' with '/:username' in **index.js**.

**3.**

Our test is now failing. Edit the welcome message in the route to use the username variable (req.params.username) instead of the hard-coded “alice”.

Run npm test to verify we’re back in the green.

When you are ready to move on, check your work.

Checkpoint 4 Passed

Hint

Use string concatenation to replace alice with the username variable in the response.

**4.**

Remove the second profile route for bob now that we have a variable route doing the work for us.

Run npm test to verify we are still green.

When you are ready to move on, check your work.

Checkpoint 5 Passed

Hint

Delete the route for bob within **profile.js**.

**5.**

At this point, you may decide to delete one of the tests (alice, bob), since the implementation is now dependent on the name as a variable only. This decision depends on the balance between the time your tests take and the cost of missing a potential corner case.

When you are ready to move on, check your work.

const {assert} = require('chai');

const request = require('supertest');

const {jsdom} = require('jsdom');

const app = require('../../app');

const parseTextFromHTML = (htmlAsString, selector) => {

  const selectedElement = jsdom(htmlAsString).querySelector(selector);

  if (selectedElement !== null) {

    return selectedElement.textContent;

  } else {

    throw new Error(`No element with selector ${selector} found in HTML string`);

  }

};

describe('profile page', () => {

  describe('GET request', () => {

    it('greets alice', async () => {

        const response = await request(app).

        get('/profile/alice');

        assert.equal(parseTextFromHTML(response.text, '#welcome-message'), 'Welcome alice!');

    });

    it('greets bob', async () => {

        const response = await request(app).

        get('/profile/bob');

        assert.equal(parseTextFromHTML(response.text, '#welcome-message'), 'Welcome bob!');

    });

  });

});

# Refactoring: Handlebars

Sometimes during the reflection of the refactor phase, you will realize that you can implement something better or more efficiently. In the code so far, we have been responding with inline HTML strings. On a large project, this could make it difficult for the front end developer to organize and maintain.

An improved approach to this is using a templating library like Handlebars to separate the HTML view from the JavaScript controller.

In the web app that you’ve built in this lesson, we’ve placed the templates in the /views folder and have an extension of .handlebars. Our controller will now use render to create the view and pass in any variables:

const param = 'Foo';

res.render('templateName', {param});

The templates are written like regular HTML, but variables can be accessed within the view using double curly braces:

<h1>{{ param }}</h1>

When the view is rendered, it will replace {{ param }} with its actual value:

<h1>Foo</h1>

**Instructions**

**1.**

Consider the previous profile page route implementation in **profile.js** to the right.

Check the existing code using npm test.

When you are ready to move on, check your work.

Checkpoint 2 Passed

Hint

Use npm test to run the tests.

**2.**

We’ve added a blank view called **profile.handlebars** in the views folder.

Copy this HTML response into that file: <h1 id="welcome-message">Welcome {{ username }}</h1>.

When you are ready to move on, check your work.

Checkpoint 3 Passed

Hint

Copy <h1 id="welcome-message">Welcome {{ username }}</h1> into views/profile.handlebars

**3.**

Update the response in **profile.js** to use the newly created view and pass in the username. Use res.render() with the name of the view, 'profile' as the first argument, and the username as the second variable, {username}.

Run the test using npm test.

When you are ready to move on, check your work.

Checkpoint 4 Passed

Hint

Remove the existing res.send() in **profile.js** and replace it with res.render(), passing 'profile' and {username}.

**4.**

It looks like our test caught a small error in our new implementation — can you fix it?

Run npm test to verify it is in the green now.

When you are ready to move on, check your work.

Checkpoint 5 Passed

Hint

There is an exclamation point missing from the end of the response.

const {assert} = require('chai');

const request = require('supertest');

const {jsdom} = require('jsdom');

const app = require('../../app');

const parseTextFromHTML = (htmlAsString, selector) => {

  const selectedElement = jsdom(htmlAsString).querySelector(selector);

  if (selectedElement !== null) {

    return selectedElement.textContent;

  } else {

    throw new Error(`No element with selector ${selector} found in HTML string`);

  }

};

describe('profile page', () => {

  describe('GET request', () => {

    it('greets user with custom message', async () => {

        const username = 'alice';

        const response = await request(app).

        get('/profile/' + username);

        assert.equal(parseTextFromHTML(response.text, '#welcome-message'), 'Welcome ' + username + '!');

    });

  });

});

# API Errors

As mentioned earlier, one of the use cases for server testing is for checking API responses, especially the “sad path” where a user interacts with the server in an unexpected or disallowed manner. We need to make sure our server properly handles invalid passwords, form field errors, etc.

Ensuring the app is designed to withstand these issues and that the error interactions are well bounded is important.

Keep in mind that while there may only be one “happy path” for an interaction (user submits a valid password), there can be many corresponding “sad paths” (password is too short, doesn’t contain special characters, etc). By testing the majority of these on the server level, it saves us from testing them at a more resource intensive level including the user view.

**Instructions**

**1.**

We’ve added a new route to our application. It allows users to POST a message. After this, the user should be redirected to '/'. There is a passing assertion that handles this “happy path”.

We need to check the “sad path” where a user provides insufficient inputs.

We have started a test in **messages-test.js** to check the server responds with an error message when a submission does not contain an author.

Add an assertion to verify the server responds to this case with a status code of 400 (Bad Request).

Run the test using npm test.

When you are ready to move on, check your work.

Checkpoint 2 Passed

Hint

Use assert.equal to check that the response.status is equal to 400. Note that the test will fail - this is ok!

**2.**

The test is failing since we have not implemented this behavior yet. Within **messages.js** uncomment the code block that responds with the 400 response.

Verify the test now passes using npm test.

When you are ready to move on, check your work.

Checkpoint 3 Passed

Hint

Uncomment the whole block in **messages.js** that is commented out.

**3.**

Let’s make sure the server responds with an appropriate error message as well. Add an assertion to our test to verify the server is responding with a message of ‘Every message requires an author’.

Note that this API is returning JSON, so you will access the message content using:

JSON.parse(response.text).message

Run the test using npm test. This should fail since we have not implemented the correct response.

When you are ready to move on, check your work.

Checkpoint 4 Passed

Hint

Use assert.equal() to check that JSON.parse(response.text).message is equal to ‘Every message requires an author’.

**4.**

The test is in the red. Add the correct string (‘Every message requires an author’) to the response in **messages.js**.

Run the test again using npm test.

When you are ready to move on, check your work.

Checkpoint 5 Passed

Hint

Add ‘Every message requires an author’ as the JSON message text within the res.send() in **messages.js**:

{ message: 'Every message requires an author'}

const {assert} = require('chai');

const request = require('supertest');

const {jsdom} = require('jsdom');

const app = require('../../app');

const parseTextFromHTML = (htmlAsString, selector) => {

  const selectedElement = jsdom(htmlAsString).querySelector(selector);

  if (selectedElement !== null) {

    return selectedElement.textContent;

  } else {

    throw new Error(`No element with selector ${selector} found in HTML string`);

  }

};

**Summary**

We used several technologies to write tests for both “happy” and “sad” paths of:

* Server status codes
* Server response content
* Error cases

We also saw how TDD can be used at the server level to guide the implementation of the server code:

* We wrote a failing test
* We wrote the minimal required server code to pass the test
* When we decided or needed to refactor to meet external requirements, we used the existing tests to make sure our refactored code maintained the same end behavior

**Instructions**

**1.**

In this course we moved from the feature level to the server level for our TDD approach. When working with persistent data, you will need to continue down the stack to the model layer. We’ve updated the test from the initial exercise in **messages-test.js** to the right.

Again, we are checking that a valid message is saved. However, instead of checking the returned page contains the new message, we are checking the message is saved to the database layer.

Run the test using npm test.

This test fails, since we have not yet implemented the model layer, which is our next step as a developer.

const {assert} = require('chai');

const request = require('supertest');

const {jsdom} = require('jsdom');

const app = require('../../app');

const parseTextFromHTML = (htmlAsString, selector) => {

    const selectedElement = jsdom(htmlAsString).querySelector(selector);

    if (selectedElement !== null) {

      return selectedElement.textContent;

    } else {

      throw new Error(`No element with selector ${selector} found in HTML string`);

    }

};

describe('when the Message is valid', () => {

    it('creates a new message', async () => {

      const author = 'user name';

      const message ='feature testing with TDD makes me feel empowered to create a better workflow';

      //save message

      const response = await request(app)

        .post('/messages')

        .type('form')

        .send({author, message});

      //check database to verify message is saved

      assert.ok(await Message.findOne({message, author}), 'Creates a Message record');

    });

});