Well-Tested Full-Stack To-Do Items

In this project, you will test a full-stack JavaScript and HTML application! You will write tests to make sure the code that was written for the project will meet the expectations of the requirements. Your tests will not have to be exhaustive. Instead, there are guidelines for your tests in each test file. Use those guidelines to implement the Web application.

The upcoming video provides you a full walk-through of the system as it is created. Then, once you understand how the application works from watching it be built, you will need to apply your knowledge of writing tests.

It may be hard. However, stick with it. You'll do great. Just take your time, write good tests, and you will be amazed at how much confidence that you will gain in writing code that comes together.

One of the ways that you can make this project more enjoyable is to vary the way that you pair on it. For each step,

- Discuss what the next feature is that you want to write
- One person writes a unit test to test the code
- Both examine the code and determine if there is any duplication to refactor into common functions or classes
- Loop, but swap who writes the unit test and who writes the code

At the end, you will leverage your test by swapping out the mechanism used to generate the HTML. This is the other part of writing good tests: tests give you the confidence to change code. If you do something that is "wrong" in that it breaks current expectations, the tests will tell you!

Tests are such an important part of a developers life. While some developers will complain about having to write them, when you start working on an existing "legacy" code base, making changes can cause a lot of stress unless you have the work of other developers' tests to make sure you don't unintentionally change a method in such a way to break code that you're not working on.

To get started,

- clone the project fromhttps://github.com/appacademy-starters/testing-an-existing-app-project
- change directory into the project
- run `npm install`to install the modules

If you want to run the server, type `node server.js`and go tohttp://localhost:3000/itemsto see what it does. You can add categories, add items, search for items, and complete items.

The code that you will test are the functions used to create the functionality of the data and the creation of the views, not the actual HTTP server. The video after shows a person writing the entire application. You will see what each piece does. Then, you will understand the *intent* of the code that you have to test.

Create And Serve the Category Screen

You'll now write the tests for the part of the application that shows the list of categories. That code, in **server.js**looks like this.

```
const filePath = path.join(__dirname, 'category-list-screen.html');
const template = await fs.promises.readFile(filePath, 'utf-8');
const html = mergeCategories(template, categories, 'li');
res.setHeader('Content-Type', 'text/html');
res.writeHead(200);
res.write(html);
```

You need to write tests for the function `mergeCategories()`for the portion that outputs the HTML for list items. Open the file **test/merge-categories-spec.js**. You will see

```
describe("mergeCategories()", () => {
  context("Using <li> tags", () => {
    const template = `
      <div>
        <ul>
          <!-- Content here -->
        </ul>
      </div>
    `;

    it("should return no <li>s for no categories", () => {
      expect.fail('please write this test');
    });

    it("should return a single <li> for one category", () => {
      expect.fail('please write this test');
    });

    it("should return an <li> for each category", () => {
      expect.fail('please write this test');
    });
  });

  // more code ...
```

The `context`block is for writing tests for when we use `mergeCategories()`and pass it `<li>`tags.

You will need to write tests in all the `it`blocks. Just replace the `expect.fail`calls with your own tests. (`expect.fail`is a chai assertion to force a spec to fail so we are using it for all the unwritten tests so that when you run `npm test`you will see all the tests you haven't written failing)

Open **merge-categories.js**to review the code before writing the tests.

The `mergeCategories`function takes a string through its `template`parameter, a list of strings through its `categories`parameter and an HTML tag through it's `tagName`parameter.

It then replaces the HTML comment `<!-- Content here ..>`with the newly created `<li>`tags (one for each catagory) and returns a new string of HTML.

Use the `template`variable that is available to you for these tests.

The first test

The first test reads

```
it("should return no <li>s for no categories", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three As* of unit testing.

In the *arrange* section, you will need to create an empty array for the `categories` and store it in a variable. You will use the variable in the action.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains an empty array as the second argument, and the string 'li' for the tag name as the third argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true using the include assertion provided by Chai:

- To make sure that the method doesn't *remove* the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<ul>"
  - Assert that it contains the string "
"
- To make sure that the method doesn't *add* the wrong things
  - Assert that it does not contain the string "<li>"
  - Assert that it does not contain the string "
"
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Here's what the test could look like.

```
it("should return no LIs for no categories", () => {
  const categories = [];
  const result = mergeCategories(template, categories, 'li');
  expect(result).to.contain('<div>');
  expect(result).to.contain('</div>');
  expect(result).to.contain('<ul>');
  expect(result).to.contain('</ul>');
  expect(result).to.not.contain('<li>');
  expect(result).to.not.contain('</li>');
});
```

Notice we are using `contain` here instead of `include`. `contain` is an alias to `include` that chai provides, and it reads better here than `include`.

The second test

The second test reads

```
it("should return a single <li> for one categories", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three As* of unit testing.

In the *arrange* section, you will need to create an array for the `categories` argument that contains a single string and store it in a variable. You will use the variable in the action and the value that you typed in the assertion.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains the array with the single value as the second argument, and the string 'li' for the tag name as the third argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<ul>"
  - Assert that it contains the string "
"
- To make sure that the method *adds*the right things
  - Assert that it does contain the string "
your string here
" where "your string here" is the single value that you placed in the array
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Here's what the test could look like.

```
it("should return a single LI for one categories", () => {
  const categories = ['Cat 1'];
  const result = mergeCategories(template, categories, 'li');
  expect(result).to.contain('<div>');
  expect(result).to.contain('</div>');
  expect(result).to.contain('<ul>');
  expect(result).to.contain('</ul>');
  expect(result).to.contain('<li>Cat 1</li>');
});
```

The third test

The third test reads

```
it("should return an <li> for each category", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail`line with a test that properly follows the *Three A*sof unit testing.

In the *arrange*section, you will need to create an array for the `categories`argument that contains multiple strings and store it in a variable. You will use the variable in the action and the values that you typed in the assertion.

In the *act*section, you will invoke the `mergeCategories`function with the`template`as the first argument, the variable that contains the array with the multiple values as the second argument, and the string 'li' for the tag name as the third argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<ul>"
  - Assert that it contains the string "
"
- To make sure that the method *adds*the right things, for *each*of the values that you put in your categories array:
  - Assert that it does contain the string "
value n
" where "value n" is one of the values in your array
- To make sure it replaces what you expect it to replace

- Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Here's what that code could look like.

```
it("should return an LI for each category", () => {
  const categories = ['Cat 1', 'Cat 2', 'Cat 3'];
  const result = mergeCategories(template, categories, 'li');
  expect(result).to.contain('<div>');
  expect(result).to.contain('</div>');
  expect(result).to.contain('<ul>');
  expect(result).to.contain('</ul>');
  expect(result).to.contain('<li>Cat 1</li>');
  expect(result).to.contain('<li>Cat 2</li>');
  expect(result).to.contain('<li>Cat 3</li>');
});
```

You have won this round!

Save Submitted Category Information

You'll now write the tests for the part of the application that saves a category when it is submitted. That code, in **server.js**looks like this and is what happens when a new category is sent to the server in an HTTP POST.

```
else if (req.url === "/categories" && req.method === 'POST') {
    const body = await getBodyFromRequest(req);
    const newCategory = getValueFromBody(body, 'categoryName')
    categories = saveCategories(categories, newCategory);
    res.setHeader('Location', '/categories');
    res.writeHead(302);
}
```

There are three main functions we need to test in this block of code.

Here's what they all three do in a nutshell:

`getBodyFromRequest`- Gets the raw POST body string from the HTTP POST request`getValueFromBody`- Parses this raw string into individual values representing the new categories`saveCategories`- Saves the new categories into the existing list of categories;

You need to write tests for all three functions `getBodyFromRequest`,`getValueFromBody`, and `saveCategories`.

Testing getting the body from the request (`getBodyFromRequest()`)

Open up **get-body-from-request.js**and review it.

The `getBodyFromRequest()`function takes one argument `req`which is an IncomingMessageobject. It returns a Promise which means it is an *asynchronous*function.

The `IncomingMessage`stored in `req`contains properties like `url`and `method`, but also a stream of data that was sent to us by the browser. We call this stream of data the POST "body".

"GET" requests have *no*data in the body, ever. "POST" submissions almost *always*contain data. Because this is a POST that the code is handling, the code needs to read *all*of the data from the stream. To do that, it listens for two events, the "data" event and the "end" event.

When data shows up for the server to read, it has to do it in chunks because we can't predict how much data the browser will be sending the server and the data could be huge!

The code to do that is

```
req.on('data', chunk => {
  data += chunk;
});
```

The callback will be called everytime the server receives a chunk of data from the browser. Then we just append the incoming chunk of data to the existing data variable with `+=`.

We will continue to do this as long as the server is still receiving chunks.

When the data finishes arriving at the server, the "end" event occurs. That signals the code that it has finished arriving and the Promise in the method can finish with a call to `resolve`passing it the `data`. That is this piece of code from `getBodyFromRequest`:

```
req.on('end', () => {
  resolve(data);
});
```

This is a hard one to test because you need to test those events. The stream of data inherits from a class `EventEmitter`. You can use an instance of the `EventEmitter` class to test this code. This is called *a stub* or *a fake* because it's not a real `IncomingMessage`. You can trigger an event using the `emit` method which takes the name of the event as the first parameter and, as an optional second parameter, any data.

For Example:

```
const fakeReq = new EventEmitter();
fakeReq.emit('end');
```

would emit the "end" event.

Another thing that makes this hard is that it is an *asynchronous* test which means that you **must** use the `done` method that mocha provides as part of the test callback. If everything is ok, then you call `done` without any arguments. If something bad happens, you call `done` with the error message.

You can see an example in this `it` block from the **get-body-from-request-spec.js** file. The `done` function is the first argument to the `it` callback.

```
  it('returns an empty string for no body', done => {
    expect.fail('please write this test');
  });
```

This should remind you of the `resolve` function in Promises, it's a similar pattern.

The first body request test

For the first test, *returns an empty string for no body*, the following code uses the `EventEmitter` stored in `fakeReq` (which is created in the `beforeEach` block) as the fake request to test the `getBodyFromRequest` function.

Write your assertion in the `then` handler of the promise returned by `getBodyFromRequest`. Check to see if the value in `body` is an empty string. If it is, the function works as you expect and you should call `done()`. If not, you should call `done` with an error message. The comments in the `then` function are there to guide you to do that.

```
it('returns an empty string for no body', done => {
  // Arrange
  const bodyPromise = getBodyFromRequest(fakeReq);

  // Act
  // This next line emits an event using
  // emit(event name, optional data)
  fakeReq.emit('end');

  // Assert
  bodyPromise
    .then(body => {
      // Write the following code:
      // Determine if body is equal to ""
      // If it is, call done()
      // If it is not, call
      //   done(`Failed. Got "${body}"`)
    });
});
```

The second body request test

For the second test, *returns the data read from the stream*, use the `EventEmitter` stored in `fakeReq` as the fake request to test the `getBodyFromRequest` function. This time, though, you need to emit some "data" events before you emit the "end" event to test the data-gathering functionality of the method.

From the last section, you know that the signature for the `emit` method is

```
eventEmitter.emit('event name', 'optional data');
```

In the cases below, the event name is "data" and the optional data is stored in `data1` and `data2`. So, you should have *two* calls to `emit` before the `fakeReq.emit('end');`. You can see space for you to write those calls.

Then, in the `then` handler of the Promise, you should check to see if the value in `body` is the same as `data1 + data2`. If it is, the function works as you expect and you should call `done()`. If not, you should call `done` with an error message. The comments in the `then` function are there to guide you to do that.

```
it('returns the data read from the stream', done => {
  // Arrange
  const bodyPromise = getBodyFromRequest(fakeReq);
  const data1 = "This is some";
  const data2 = " data from the browser";

  // Act
  // Write code to emit a "data" event with
  // the data stored in data1

  // Write code to emit a "data" event with
  // the data stored in data2

  fakeReq.emit('end');

  // Assert
  bodyPromise
    .then(body => {
      // Write the following code:
      // Determine if body is equal to data1 + data2
      // If it is, call done()
      // If it is not, call
      //   done(`Failed. Got "${body}"`)
    });
});
```

Testing getting the value from the body (`getValueFromBody`)

It's not enough to just get the stream of raw data from the `POST` body, we also need to parse that data into the categories the user is saving.

When someone POSTs a form from the browser to the server, it comes to the server in a format called "x-www-form-urlencoded". This is also sometimes called a "Query String"

"x-www-form-urlencoded" is just a format for data just like JSON is also a format for data. This specific format is made up of key/value pairs. The key/value pairs are in the form "*key=value*". Those pairs are joined together in a single string by using the ampersand character. The following are valid strings contained in the "x-www-form-urlencoded" format.

- **""**: The empty string is a valid x-www-form-urlencoded string.
- **"name=Morgan"**: The key in this case is "name" and the value is "Morgan"
- **"name=Petra&age=31"**: There are two key/value pairs in this, "name" and "Petra", and "age" and "31"
- **"name=Bess&age=&job=Boss"**: There are three key/value pairs in this
  - "name" and "Bess"
  - "age" and "" (I guess they didn't answer?)
  - "job" and "Boss"

If one of the values contains a character that's not a letter or number, it's replaced by a weird number that begins with a percent sign. That's known as URL encoding. The most common replacement is "%20" for the space character. Here are some valid strings with that replacement.

- **"name=Chandra%20K&age=13%20years%20old"**: This string has two key/value pairs
  - "name" and "Chandra K"
  - "age" and "13 years old"

- **"title=Boop%20Lord"**: This string has one key/value pair, "title" and "Boop Lord"

In the tests that you write, you will not have to write these "x-www-form-urlencoded" strings. They will be provided to you in the test. However, you should be able to read them so that you become familiar with how they > work.

Open up the **get-value-from-body.js**file to see the two lines of code in the `getValueFromBody`function that implement this behavior. Notice we are using the built in `querystring`module in Node.js to `parse()`our `x-www-form-urlencoded`string.

`getValueFromBody`takes in two arguments, `body`and `key`. It then parses the `x-www-form-urlencoded`body and returns the value that cooresponds to `key`.

To make sure that it behaves, though, there are multiple tests in **get-value-from-body-spec.js**in the **test**directory. Let's look at those.

There are five tests. The first three have the body *and*key defined for you. The last two have the body defined and you should figure out the key to test.

The first test

The first test is *returns an empty string for an empty body*. So, if the body is empty, regardless of the key, the `getValueFromBody`method returns an empty string.

```
it('returns an empty string for an empty body', () => {
  // Arrange
  const body = "";
  const key = "notThere";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value of ""
  expect.fail('please write this test');
});
```

In this test, you need to write the code that invokes the `getValueFromBody`method with the `body`and `key`arguments. The result that comes back is what you should assert instead of just having it fail.

Take a moment and try to complete that on your own. The following code snippet will show you the solution, so give it a shot figuring out the two lines of code that you need to complete the previous one.

Here's the solution:

```
it('returns an empty string for an empty body', () => {
  // Arrange
  const body = "";
  const key = "notThere";
```

```
  // Act
  // Write code to invoke getValueFromBody and collect
  // the result
  const result = getValueFromBody(body, key);

  // Assert
  // Replace the fail line with an assertion for the
  // expected value of ""
  expect(result).to.equal('');
});
```

The second test

The second test is *returns an empty string for a body without the key*. So, if you ask for the value of a key that is not in the body, the `getValueFromBody` method returns an empty string.

```
it('returns an empty string for a body without the key', () => {
  // Arrange
  const body = "name=Bess&age=29&job=Boss";
  const key = "notThere";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value of ""
  expect.fail('please write this test');
});
```

This code will look very, very similar to the last test. Complete it to make it pass.

The third test

The third test, *returns the value of the key in a simple body*, is also very similar to the past two tests. In this case, you have to compare it to the expected value "Bess".

```
it('returns the value of the key in a simple body', () => {
  const body = "name=Bess";
  const key = "name";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value of "Bess"
  expect.fail('please write this test');
});
```

The fourth test

The fourth test, *returns the value of the key in a complex body*, is also very similar to the past three tests. In this case, you have to choose a key that you want to test from the existing keys in the body and, then, the value that it has so that you can make the assertion at the end.

```
it('returns the value of the key in a complex body', () => {
  const body = "name=Bess&age=29&job=Boss";
  // Select one of the keys in the body

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result
```

```
  // Assert
  // Replace the fail line with an assertion for the
  // expected value for the key that you selected
  expect.fail('please write this test');
});
```

The fifth test

The fifth test, *decodes the return value of URL encoding*, is also very similar to the past three tests. In this case, you will test the value of the "level" key. Complete the code with the correct assertion. Remember that `%20` should be decoded and be turned into the space character.

```
it('decodes the return value of URL encoding', () => {
  const body = "name=Bess&age=29&job=Boss&level=Level%20Thirty-One";
  const key = "level";

  // Act
  // Write code to invoke getValueFromBody and collect
  // the result


  // Assert
  // Replace the fail line with an assertion for the
  // expected value for the "level" key
  expect.fail('please write this test');
});
```

Testing saving the categories

Open up **save-categories.js** and review it. This contains a method that pushes a new category in the `newCategory` parameter onto the argument provided in `categories` (hopefully an array!). Then, it sorts the `categories` array. Finally, it returns a "clone" of the array by just creating a new array with all of the old entries. This is done to keep modifications to the old array from messing with the new array. It is an implementation detail that you just need to test for.

Open **save-categories-spec,js**. It has three tests in it for you to complete.

The first test

In the first test, you must provide the "Act" stage by calling the `saveCategories` method with the provided `categories` and `newCategory` values and store its return value in a variable named "result".

Of note with the first test is that the assertion (that you do not have to write) uses the "include" method to test if a value is in an array.

The second test

In the second test, you must provide the "Assert" stage by writing the assertion to test using a new method named "eql" rather than "equal". Everything else remains the same.

The reason that you use eql instead of `equal` is the "type" of equality each one provides. The `equal` function, which you've used until now, compares objects and arrays only by their instance. That means equality between arrays and objects using `equal` will only pass if they're *the same object in memory*.

```
// Different arrays with the same content
expect(['a', 'b']).to.equal(['a', 'b']); // => FAIL

// Same arrays
const array = ['a', 'b'];
expect(array).to.equal(array); // => PASS
```

The `eql` method performs "member-wise equality". It will compare the values *inside* the array as opposed to the instance of the array. Because of that, both of the previous examples pass with the `eql` method.

```
// Different arrays with the same content
expect(['a', 'b']).to.eql(['a', 'b']); // => PASS

// Same arrays
const array = ['a', 'b'];
expect(array).to.eql(array); // => PASS
```

The third test

In the third test, you must provide the "Arrange" portion. Interestingly, you can really provide any array and string value. That's an easy one.

Et, voila!

It seems that you have fully tested all of the code that it takes to test the "save category" code. Well done!

You win this round, too!

Create And Serve A To-Do Item Form

To display the form that lets you enter new items, it has to create a dropdown that contains the categories that are in the application. This is identical in *intent* to the code that creates a list of categories to display on the category screen. Because of that, here's the code that handles displaying the "new item" screen.

```
else if (req.url === "/items/new" && req.method === 'GET') {
  const filePath = path.join(__dirname, 'todo-form-screen.html');
  const template = await fs.promises.readFile(filePath, 'utf-8');
  const html = mergeCategories(template, categories, 'option');
  res.setHeader('Content-Type', 'text/html');
  res.writeHead(200);
  res.write(html);
}
```

In this case, the `mergeCategories` method is now called with the third argument of "option" rather than "li" as it was before. This is what the last three tests in the **merge-categories-spec.js** file address. You will write tests that generate "option" tags rather than "li" tags. You'll also test that the replacement correctly occurred.

In this case, you'll modify the tests in the second sub-"describe" section, the one that reads "For selects".

The first test

The first test reads

```
it("should return no <option>s for no categories", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three As* of unit testing.

In the *arrange* section, you will need to create an empty array for the `categories` and store it in a variable. You will use the variable in the action.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains an empty array as the second argument, and the string 'option' for the tag name as the third argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true in the return value that you saved in the *act* section using the include assertion provided by Chai:

- To make sure that the method doesn't *remove* the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<select>"
  - Assert that it contains the string "</select>"

- To make sure that the method doesn't *add* the wrong things
  - Assert that it does not contain the string "<option>"
  - Assert that it does not contain the string "</option>"

- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Except for some string differences, this test will look nearly identical to the first test that you did for the `<li>` tags earlier in this project.

The second test

The second test reads

```
it("should return a single <option> for one category", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

In the *arrange* section, you will need to create an array for the `categories` argument that contains a single string and store it in a variable. You will use the variable in the action and the value that you typed in the assertion.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains the array with the single value as the second argument, and the string 'option' for the tag name as the third argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true using the include assertion provided by Chai:

- To make sure that the method doesn't *remove* the wrong things
    - Assert that it contains the string " "
    - Assert that it contains the string " "
    - Assert that it contains the string "<select>"
    - Assert that it contains the string "</select>"
- To make sure that the method *adds* the right things
    - Assert that it does contain the string "<option>your string here</option>" where "your string here" is the single value that you placed in the array
- To make sure it replaces what you expect it to replace
    - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Except for some string differences, this test will look nearly identical to the second test that you did for the `<li>` tags earlier in this project.

The third test

The third test reads

```
it("should return an <option> for each category", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

In the *arrange* section, you will need to create an array for the `categories` argument that contains multiple strings and store it in a variable. You will use the variable in the action and the values that you typed in the assertion.

In the *act* section, you will invoke the `mergeCategories` function with the `template` as the first argument, the variable that contains the array with the

many values as the second argument, and the string 'option' for the tag name as the third argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string " "
  - Assert that it contains the string " "
  - Assert that it contains the string "<select>"
  - Assert that it contains the string "</select>"

- To make sure that the method *adds*the right things, for *each*of the values that you put in your categories array:
  - Assert that it does contain the string "<option>value n</option>" where "value n" is one of the values in your array

- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Except for some string differences, this test will look nearly identical to the third test that you did for the `<li>`tags earlier in this project.

You have won this round!

Save And Show To-Do Items

In this step, you'll test the code for two different handlers, the one that shows the screen that has the list of items on it and the one that handles the creation of a new item. Here are those two parts of the `if-else` block in **server.js**.

```
else if (req.url === "/items" && req.method === 'GET') {
  const filePath = path.join(__dirname, 'list-of-items-screen.html');
  const template = await fs.promises.readFile(filePath, 'utf-8');
  const html = mergeItems(template, items);
  res.setHeader('Content-Type', 'text/html');
  res.writeHead(200);
  res.write(html);
}

else if (req.url === "/items" && req.method === 'POST') {
  const body = await getBodyFromRequest(req);
  const category = getValueFromBody(body, 'category')
  const title = getValueFromBody(body, 'title')
  items = saveItems(items, { title, category });
  res.setHeader('Location', '/items');
  res.writeHead(302);
}
```

What's really great to note here is that you have already tested `getBodyFromRequest` and `getValueFromBody`! That means, out of all that code, there are only two methods for which you must write tests! Those are `mergeItems` and `saveItems`.

Testing the merge items method

This is *really* similar to the `mergeCategories` method that you've now written tests for twice. But, instead of creating an `<li>` or an `<option>`, it creates a row for a table for the items that are passed in and a form that shows a button to complete the item.

Open **merge-items.js** and review that code, please. You can see the loop on lines 5 - 23 that builds the rows of the table. Then, the form is created only if the item is *not* complete. Then, the `<tr>` and its `<td>`s are created. This just means that you will want to test instead of for `<li>`s and `<option>`s, you'll test for many `<td>`s that contain the expected values.

Open **merge-items-spec.js** and see that you have essentially the same tests that you had for the `mergeCategories` function. It may not surprise you to learn that *many* tests look the same, especially if they handle similar functionality. This can get monotonous, at times. It is better, though, to have the protection of tests by investing a little bit of time in writing them as opposed to spending days trying to find a bug that inadvertently got into the code base when someone was writing other code.

The first test

The first test reads

```
it("should return no <tr>s and no <td>s for no items", () => {
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

In the *arrange* section, you will need to create an empty array for the `items` and store it in a variable. You will use the variable in the action.

In the *act* section, you will invoke the `mergeItems` function with the `template` as the first argument and the variable that contains an empty array as the second argument. Store the return value in a variable.

In the *assert* section, assert that each of the following are true using the include assertion provided by Chai on the result of the *act*:

- To make sure that the method doesn't *remove* the wrong things
  - Assert that it contains the string "<table>"
  - Assert that it contains the string "</table>"
  - Assert that it contains the string "<tbody>"
  - Assert that it contains the string "</tbody>"
- To make sure that the method doesn't *add* the wrong things
  - Assert that it does not contain the string "<tr>"
  - Assert that it does not contain the string "</tr>"
  - Assert that it does not contain the string "<td>"
  - Assert that it does not contain the string "</td>"
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

Because you already have examples of what this looks like in `mergeCategories`, please refer to that.

The second test

The second test reads

```
it("should return a single <tr>, four <td>s, and a <form> for one uncompleted ite
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three A*s of unit testing.

If you look at the code in the `mergeItems` method, you can see that it relies on the item to have the following properties:

- `title`
- `category`
- `isComplete`

In the *arrange* section, you will need to create an array for the `items` argument that contains a single item and store it in a variable. You will use the variable in the action and the value that you typed in the assertion. Something like the following would suffice.

```
const items = [
  { title: 'Title 1', category: 'Category 1' },
];
```

In the *act* section, you will invoke the `mergeItems` function with the `template` as the first argument and the variable that contains an empty array as the second argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string "<table>"
  - Assert that it contains the string "</table>"
  - Assert that it contains the string "<tbody>"
  - Assert that it contains the string "</tbody>"
- To make sure that the method *adds*the right things
  - Assert that it contains the string "<tr>"
  - Assert that it contains the string "</tr>"
  - Assert that it contains the string "<td>Title 1</td>"
  - Assert that it contains the string "<td>Category 1</td>"
  - Assert that it contains the string "<form method="POST" action="/items/1">"
- To make sure it replaces what you expect it to replace
  - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

The third test

Now, you will test that *no*form is created when an item is complete. This will be nearly identical to what you just wrote *except*that your item should have an "isComplete" property set to `true`, and you will assert that it does *not*contain the "<form method="POST" action="/items/1">" string.

Replace the `expect.fail`line with a test that properly follows the *Three A*sof unit testing.

In the *arrange*section, you will need to create an array for the `items`argument that contains a single item that is completed and store it in a variable. Something like the following would suffice.

```
const items = [
  { title: 'Title 1', category: 'Category 1', isComplete: true },
];
```

In the *act*section, you will invoke the `mergeItems`function with the`template`as the first argument and the variable that contains an empty array as the second argument. Store the return value in a variable.

In the *assert*section, assert that each of the following are true using theincludeassertion provided by Chai:

- To make sure that the method doesn't *remove*the wrong things
  - Assert that it contains the string "<table>"
  - Assert that it contains the string "</table>"
  - Assert that it contains the string "<tbody>"
  - Assert that it contains the string "</tbody>"
- To make sure that the method *adds*the right things

    - Assert that it contains the string "<tr>"
    - Assert that it contains the string "</tr>"
    - Assert that it contains the string "<td>Title 1</td>"
    - Assert that it contains the string "<td>Category 1</td>"

- To make sure that the method does not add the wrong things
    - Assert that it does not contain the string "<form method="POST" action="/items/1">"

- To make sure it replaces what you expect it to replace
    - Assert that it does not contain the string "<!-- Content here -->"

Run the test to make sure it passes.

The fourth test

Now, try writing the last test `it('should return three <tr>s for three items')` as a combination or extension of the previous two. Check to make sure that you get all of the indexes for the items that you have in your array. Make sure that the "form" elements appear for those items that are not complete.

Testing the save items method

Open the **save-items.js**file and review the function. It merely adds a new item to the array passed in using the `push`method. Then, it creates a clone of the old array using the "spread operator". If you're not familiar with that syntax, don't worry. All it does is make a copy of the array.

Open the **save-items-spec.js**. You will see two methods in there. These are nearly identical to the first and last tests for the `saveCategories`method. Use the same pattern to complete those tests.

Just as a reminder, a solution for the **save-categories-spec.js**file (with comments removed) could look like this.

```
describe("The saveCategories function", () => {
  it('adds the new category to the list', () => {
    const categories = ['Cat 3', 'Cat 2'];
    const newCategory = 'Cat 1';
    const result = saveCategories(categories, newCategory);
    expect(result).to.contain(newCategory);
  });

  it('makes sure the result and the original are different', () => {
    const categories = ['Cat 3', 'Cat 2'];
    const result = saveCategories(categories, 'Cat 1');
    expect(result).to.not.equal(categories);
  });
});
```

Your code will look a lot like this *except*you should have arrays of items and new items, not strings. Remember from the last section, an array of items might look like this:

```
const items = [
  { title: 'Title 1', category: 'Category 1' },
];
```

Run your tests to make sure they pass.

Show And Complete A To-Do Item

You're almost done with testing this application! Have a look at the method that completes a to-do item.

```
else if (req.url.startsWith('/items/') && req.method === 'POST') {
  const index = Number.parseInt(req.url.substring(7)) - 1;
  items[index].isComplete = true;
  res.setHeader('Location', '/items');
  res.writeHead(302);
}
```

That's interesting. There's nothing to test there, no methods. That's some kind of wonderful! On to the next item!

Search For To-Do Items

What may be the most complex set of tests to write (except that weird event emitter thing), search makes you think though what it should do in a variety of cases.

Here's the relevant part of the server that handles a search query.

```
else if (req.url.startsWith('/search') && req.method === 'GET') {
  const [_, query] = req.url.split('?', 2);
  const { term } = querystring.parse(query);
  const filePath = path.join(__dirname, 'search-items-screen.html');
  const template = await fs.promises.readFile(filePath, 'utf-8');
  let foundItems = [];
  if (term) {
    foundItems = searchItems(items, term);
  }
  const html = mergeItems(template, foundItems);
  res.setHeader('Content-Type', 'text/html');
  res.writeHead(200);
  res.write(html);
}
```

You've already tested `mergeItems`, so that's not needed, again. The only method that you will need to test is `searchItems`.

Open **search-items.js** and review how that code is working. It takes a list of `items` and a search `term`. The first thing it does is force the term to lower case.

```
term = term.toLowerCase();
```

Then, it uses the `filter` function on the array to create a new array of items that meet the comparison in the function. The comparison function makes the title lower case and checks to see if the term is contained in that string.

```
return items.filter(x => {
  const title = x.title.toLowerCase();
  return title.indexOf(term) >= 0;
});
```

If the term *is* in the title, then the comparison returns `true` and the `filter` function will add it to the new array. If the term is *not* in the title, the comparison returns `false` and it is not added to the new array.

Here is an example. Supposed you have the following items in your array.

```
[
  { title: 'Go grocery shopping', category: 'Home' },
  { title: 'Play with my puppy', category: 'Pet' },
  { title: 'Shop for a puppy bed', category: 'Pet' },
]
```

Now, say the search term someone entered is "SHOP". This is what happens in the function.

```
Convert "SHOP" to "shop"
Filter the array of items based on the term "shop":
  Item 1:
    Convert "Go grocery shopping" to "go grocery shopping"
    Does it contain the term "shop"? YES
    Add it to the new array
  Item 2:
    Convert "Play with my puppy" to "play with my puppy"
    Does it contain the term "shop"? NO
  Item 3:
    Convert "Shop for a puppy bed" to "shop for a puppy bed"
    Does it contain the term "shop"? YES
    Add it to the new array
Return the new array that contains items 1 and 3
```

So, that's what you want to test for.

Open **search-items-spec.js**. You'll see three tests.

In the first test, you are asked to fix the *arrange*step to declare `items`and `term`given the directions. This is not a trick. It's just declaring those two variables that it's asking you to create.

In the second test, fix the *assert*step to assert the proper length of the result by completely replacing the `expect.fail`line.

In the third test, you are asked to fix the *arrange*step by choosing a string value for `term`that makes the rest of the test pass.

What have you done?

Now that you've done that, you've won the entire game! All of the meaty logic of the game is now well tested. If someone were to come along and try to change the code, when the tests ran, it would check to make sure they didn't accidentally break something in their earnest to add new functionality!

Here's what you did:

- You've looked at, read, and understood other people's code
- You've seen and used a variety of assertions
- You've seen how to do real (not fake) asynchronous testing using the `done`method
- You've invested time in hardening the maintainability of an application

Here's a link to a solution.https://appacademy-open-assets.s3-us-west-1.amazonaws.com/Module-JavaScript/testing/projects/testing-an-existing-project-solution.zip

In the next step, you're going to use the fact that you have tests to radically change the code.

Refactor To Use A Template Engine

Note: the solution project does not have this step included in it because it changes earlier tests.

The hard work you've done with `mergeCategories` and `mergeItems` to make sure that the important parts of the HTML are generated, those are some really good tests that you're now going to use to change the way the entire HTML is generated.

This is the other side of testing. When you can feel confident that what you are doing will not break the code because you have tests that tell you what to do.

You're going to follow some steps to replace the HTML-generating portion of the application. The steps will be explicit, because this is less about learning a library as it is proving to yourself that tests are a good thing.

Hello, handlebars

There's very little chance that you would create a Web application, anymore, and generate your own HTML the way it was done in the `mergeCategories` and `mergeItems` functions. Instead, you would use a "templating engine" which is a library that takes some template (with some fancy instructions) and some data and generates HTML *for you*.

You'll change your tests to use a handlebars style template which looks nearly identical to HTML. This will break your tests. Then, you will change the functions to use the handlebars engine. Then, you will know that they properly handle handlebars templates, so you'll change the HTML files to use that syntax rather than using the "<!-- Content here -->" placeholder.

All you need to know about handlebars

This is just an informative section so you know what will actually be going on in the tests. You're not going to be asked to come up with any of this yourself. This is showing you how you would, in the real world, update existing code and tests in a real application.

When you use the handlebars engine, you pass it two things, a string that contains the template and an object that contains the data that you want to show.

Assume that this is your data object.

```
const data = {
  name: 'Remhai',
  nicknames: [ 'R', 'Rem', 'Remrem' ],
  addresses: [
    { street: '123 Main St', city: 'Memphis', state: 'TN' },
    { street: '2000 9th Ave', city: 'New York', state: 'NY' },
  ],
};
```

In your template, if you want to output the value in the `name` property, you just put the name of the property in double curly brackets.

```
<div>
  Name: {{ name }}
</div>
```

In your template, if you want to output all of the nicknames of the person, you loop over that property using the `#each` helper like this. Then, inside the `#each` "block", you refer to the value of the string itself as `this`.

```
<ul>
  {{#each nicknames}}
    <li>{{ this }}</li>
  {{/each}}
</ul>
```

In your template, if you want to output all of the addresses of the person, you loop over the property using the `#each`helper in which you will use the property names of the objects inside the array. You can use `@index`to give you the current index.

```
<tbody>
  {{#each addresses}}
    <tr>
      <td>{{ @index }}</td>
      <td>{{ street }}</td>
      <td>{{ city }}</td>
      <td>{{ state }}</td>
    </tr>
  {{/each}}
</tbody>
```

If you want to do a conditional, you can just do something like this.

```
{{#if isVisible}}
  <div>You can see me!</div>
{{else}}
  <div></div>
{{/if}}
```

So, that's *handlebars*. Again, it's just so that you can understand the syntax of the tests and HTML that you'll be changing.

Install handlebars

You just need to use `npm` to do this. `npm install handlebars`. Yay!

To do some math, you'll need to install some handlebars helpers. You just need to use `npm` to do this. `npm install handlebars-helpers`. Yay!

Now, change your merge items test

Inside **merge-items-spec.js**, you will change the `template`string. And, that is all you'll change. Instead of having the "<!-- Content here -->", you'll replace that with handlebars code. Then, your tests will fail. Then, you'll make then pass. Once they pass, you'll know you're safe to change the *real*HTML file.

Update the template from what it is now to the following code.

```
const template = `
  <table>
    <tbody>
      {{#each items}}
        <tr>
          <td>{{ add @index 1 }}</td>
          <td>{{ title }}</td>
          <td>{{ category }}</td>
          <td>
            {{#if isComplete}}
            {{else}}
              <form method="POST" action="/items/{{ add @index 1 }}">
                <button class="pure-button">Complete</button>
              </form>
            {{/if}}
          </td>
        </tr>
      {{/each}}
    </tbody>
```

```
  </table>
`;
```

This seems like a lot when compared to the other template we had. However, this moves all of the HTML-generation to the template. There will be no looping and string manipulation in the `mergeItems` function after you're done with it.

Run your tests and make sure they fail. Without a failing test, you don't know what to fix. And, in this case, all three tests fail.

Fix the merge items function

Now that you have this, it's time to update the `mergeItems` function. You'll know you're done when the tests all pass.

Inside `mergeItems`, import the *handlebars* library at the top of your file, the helpers library, and then register the 'math' helpers with the handlebars library according to the helpers documentation. We need this so we can use the `add` helper in the template to add `1` to the `@index`.

```
const handlebars = require('handlebars');
const helpers = require('handlebars-helpers');
helpers.math({ handlebars });
```

Now, delete *everything* inside the function. Replace it with the following lines.

```
const render = handlebars.compile(template);
return render({ items });
```

You've moved the complexity of the HTML generation from the source code to the HTML code. HTML code is easier to change because it's only about the display and generally won't crash your entire application.

Now, change your merge categories tests

Open **merge-categories-spec.js**. Change the first template to this code.

```
const template = `
  <div>
    <ul>
      {{#each categories}}
        <li>{{ this }}</li>
      {{/each}}
    </ul>
  </div>
`;
```

Change the second template to this code.

```
const template = `
  <div>
    <select>
      {{#each categories}}
        <option>{{ this }}</option>
      {{/each}}
    </select>
  </div>
`;
```

Now, your merge categories tests should not work. Run them to make sure.

Fix the merge categories function

Open **merge-categories.js**and import just *handlebars*. There's no math in the templates, so there's no need for the helpers.

```
const handlebars = require('handlebars');
```

Again, delete *everything*inside the function and replace it with the following to make the tests pass, again.

```
const render = handlebars.compile(template);
return render({ categories });
```

AMAZING!

What just happened?

You just performed a major refactor of the application and you knew you did it because you had tests to guide you during the refactor!

Here's an even more amazing thing. You did it without actually running the code! You did it because you had tests that told you if the inputs and outputs matched your expectations!

Now, you can change the content of the HTML pages to use the new *handlebars*syntax. There are only four of them, and you can use the stuff from your tests to update the source code.

Open **category-list-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each categories}}
  <li>{{ this }}</li>
{{/each}}
```

Open **list-of-items-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each items}}
  <tr>
    <td>{{ add @index 1 }}</td>
    <td>{{ title }}</td>
    <td>{{ category }}</td>
    <td>
      {{#if isComplete}}

      {{else}}
        <form method="POST" action="/items/{{ add @index 1 }}">
          <button class="pure-button">Complete</button>
        </form>
      {{/if}}
    </td>
  </tr>
{{/each}}
```

Open **search-items-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each items}}
  <tr>
    <td>{{ add @index 1 }}</td>
    <td>{{ title }}</td>
    <td>{{ category }}</td>
    <td>
      {{#if isComplete}}

      {{else}}
        <form method="POST" action="/items/{{ add @index 1 }}">
```

```
          <button class="pure-button">Complete</button>
        </form>
      {{/if}}
    </td>
  </tr>
{{/each}}
```

Open **todo-form-screen.html**and replace the "<!-- Content here -->" with this handlebars syntax lifted straight from the tests.

```
{{#each categories}}
  <option>{{ this }}</option>
{{/each}}
```

That completes the upgrade of the HTML files. You can run your server using`node server.js`and checkout that everything just works by going tohttp://localhost:3000/items.

Just another note

This may not seem very amazing to you. This unit testing *revolutionized*the way that programmers write software! The fact that you could go into a code base and run tests to see how code works, change code and know that you haven't broken anything, that enabled people to work more confidently that they were not introducing bugs into the software as they were going along.

This is the stuff dreams are made of.

So, good work. Good work