Save Submitted Category Information

30 minutes

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 IncomingMessage object. 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

`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
```

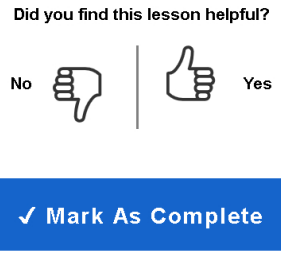
The third test

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!



Finished with this task? Click **Mark as Complete** to continue to the next page!