Save and Show Submitted To-Do Items

30 minutes

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 As* 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 item",
  expect.fail('please write this test');
});
```

Replace the `expect.fail` line with a test that properly follows the *Three As* 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 array that contains a single item as the second argument.

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 "<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 As* of 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 array that contains a single completed item as the second argument.

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 "<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.

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