Refactor To Use a Template Engine

an hour

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}}
```

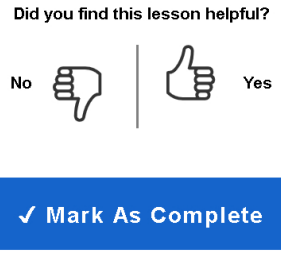
That completes the upgrade of the HTML files. You can run your server using `node server.js` and checkout that everything just works by going to http://localhost:3000/items.

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



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