- ⏱ 4 hours

Formative Forms

Today you'll be going through the formative experience of building out an Express application that uses HTML forms!

This application allows users to create two types of user accounts: a normal user account and an interesting user account. It keeps track of all of the users in a table on the home page.

When you're done with the project, your web application should have the following features:

1. Site-wide navigation elements that allows users to navigate between the home page and the two other pages.
2. A table that shows all of the existing users.
3. A form that can be used to create a normal user account.
4. A form that can be used to create an interesting user account.

Getting started

- Clone the starter project from https://github.com/appacademy-starters/formative-forms-starter
- Run `npm install` to install the dependencies
- Run `npm test` to run the tests for the project

Phase 0: Intro to the skeleton directory

The skeleton directory has a bare bone `index.js` file that renders "Hello World!" when a user land on `localhost:3000/`. There's also a `users` array that already has one user created for you. Throughout the project, as you add new users, you'll do so by pushing the new users into this array.
It also has a `layout.pug` template that your other templates can extend.
The `layout.pug` imports Bootstrap stylesheets in the `head` element. Feel free to make

your website look fancy with the available Bootstrap styling by adding Bootstrap class names to your elements.
For example, here is the documentation on how to add Bootstrap styles to a form element by adding the Bootstrap classes to each element.

Phase 1: Home page

Pass each of the specs in `01_home.test.js` file. Running `npm test` will run every single test across all five test files. If you only want to run the specs in `01_home.test.js`, run `npm run test1`.

Overall, this project will give you ample opportunity to get familiar with reading specs and then building out features to satisfy those specs. Be sure to check the specs often for guidance!

In this first phase, create an `index.pug` template and update the `app.get("/")` route so that it renders the index template. Remember to render the `users` array so that it's available in the `index.pug` template.
Extend the `index.pug` template to inherit from `layout.pug`. Declare a block named "content" and add a table to render existing users. Follow the specs to create an `h2` header for the table.
Hint: *Read the error messages to see the expected header name.*

Create table headers and columns for each user's information.

Hint: *Take a look at 01_home.test.js to view what columns are expected in the table.*
At the top of the `body` element in `layout.pug`, add navigation links so that users can easily navigate between the home page ("/" route), normal user creation form ("/create" route), and interesting user creation form ("/create-interesting" route).

Phase 2: Create the normal user form

Pass each of the specs in `02_create_form.test.js`. You can run the specs in this file by running `npm run test2`.
In this phase, go ahead and set up this route to use the csurf middleware to render a CSRF token in a hidden input field. Be sure to in the `{cookie: true}` options when creating the middleware function.
Because your application will be using cookies to store the secret CSRF value, go ahead and also set up the cookie-parser middleware as an application-wide middleware function.

Set up a route so that when users land on `/create`, a form renders with the following fields:

- `_csrf` **(hidden field)**
- `firstName`
- `lastName`
- `email`
- `password`
- `confirmedPassword`

Be sure to set correct input `type` and `name` attributes, and also remember to add a label correlating to each input field's `name`.
Make sure your `_csrf` field has an appropriate value from your middleware. At this point, remove some duplication from your Pug template by leveraging mixins. Create a mixin to easily generate `label` and `input` element pairs. Think of how you would create a mixin using `input` attributes as parameters.

Phase 3: Submitting the form

Pass each of the specs in `03_form_submit.test.js`. You can run the specs in this file by running `npm run test3`.
Begin by setting up a `"/create"` route to handle POST requests. Now that you are handling POST requests, you'll need access to `req.body`. Have your application use the `express.urlencoded` middleware to decode the form's request body string into data that can be accessible in `req.body`.
The next step is to protect this route from CSRF attacks by using the middleware function that you set up using the csurf library. Make sure you've included the middleware function in both of the GET and POST `"/create"` routes.
Now set up data validations and create an `errors` array within the `app.post("/create")` route. You want to validate whether the user has provided a `firstName`, `lastName`, `email`, and `password`. Read the error messages from the specs to determine what messages to `push` into your `errors` array.
Time to update the `create.pug` template to render the error messages. Start by creating an unordered list at the top of your `create-form.pug` block. Inside of the unordered list, add a paragraph element with the following content: `The following errors were found:`. Now, iterate through each error to create list items with the error messages. Only render errors if they are present in the `errors` array. How can you determine if there are errors present?
You'll want to be sure that you're pre-filling each input field with already-submitted values so that users don't have to fill out all of the fields again whenever there are errors. How can you update your `input` elements so that already-submitted values are still showing even after a form submission fails a data validation?

Phase 4: Create interesting user form

Pass each of the specs in `04_create_interesting_form.test.js`. You can run the specs in this file by running `npm run test4`.
Notice how this form includes all of the fields from the first form. Reduce code duplication by leveraging the Pug includes feature.
One way you could refactor is to create an `includes` directory inside your views and make the following files:

- `views/includes/errors.pug`
- `views/includes/form-inputs.pug`

Now create a `create-interesting.pug` template for your "interesting user form" and refactor your `create.pug` template to leverage the Pug `includes` feature. Start by dividing your existing `create.pug` template into `errors.pug` and `form-inputs.pug`.
Think of how to use the templates to keep your code DRY.

Phase 5: Submit create interesting user form

Pass each of the specs in `05_interesting_form_submit.test.js`. You can run the specs in this file by running `npm run test5`.
Go ahead and add validations and write error messages for this new form. Because this new form still has the same base fields as the other form, be sure to still run the same validations that you are currently running for the `app.post('/create')` route. If you have not done so already, go ahead and move all of the validations for the first form into a custom middleware function that
both `app.post('/create')` and `app.post('/create-interesting`) can use. Be sure to store those errors on the `req` object so that they can be used in a later middleware function.
Once you've ensured that your base fields are being validated, go ahead and add validations for the new `age` and `favoriteBeatle` fields. Follow the error messages in the specs to determine what your error messages should be. Please write these new validations in a custom middleware function.
Create mixins for the `favoriteBeatle` options. How can you make sure that a user's `favoriteBeatle` is automatically selected when a form with errors re-renders upon submission?
How can you make sure a user's `iceCream` checkbox accurately renders whether the user likes ice cream upon an unsuccessful form submission? When saving the user, be sure to use the checkbox's value (ex: "on") to convert the `user.iceCream` property to a boolean.
Finally, make sure your `index.pug` template is also rendering your new user properties.
You've made it! Confirm that your whole app works as expected by running `npm test`.

Bonus

In a production-level Express application, you'll likely use a library to help handle most of your data validation. One of the most popular data validation libraries is the express-validator library, which gives you a wide range of robust validations right out of the box.

Install this dependency and use it to add a validation to check that the user password being submitted is at least 5 characters long and that it at least has one number in it.

Then, go ahead and migrate any validations that you had on the `age` and `favoriteBeatle` fields to use the express-validator library instead. Once you're done with those fields, continue migrating all other validations to use express-validator and add validations to check *all* user input (e.g, checking that a valid email is submitted).
Did you find this lesson helpful?