Practical: React page

Initial Due Date: 2023-02-28 8:15AM
Final Due Date: 2023-03-31 4:15PM

Goals

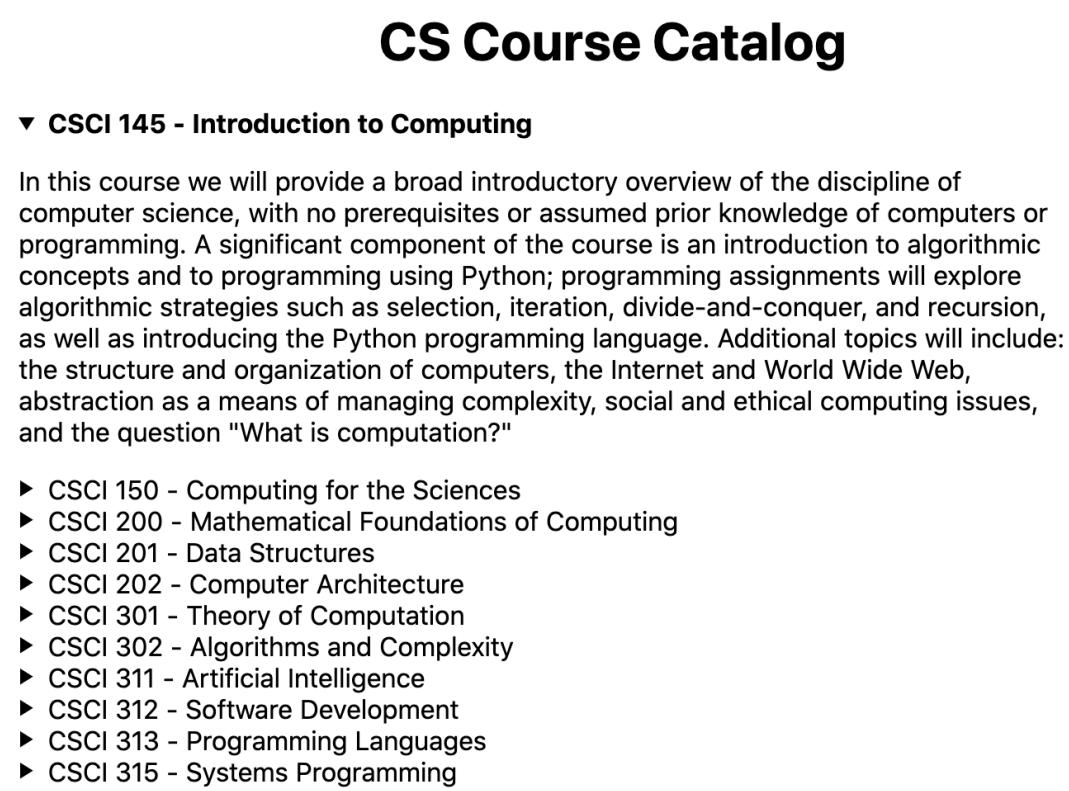
- Create your first React page
- Create a list of components using `map`
- Utilize the `useState` hook
- Add some simple styling with CSS

Prerequisite

1. Create the git repository for your practical by accepting the assignment from GitHub Classroom. This will create a new repository for you with a skeleton application already setup for you.
2. Clone the repository to you computer with `git clone` (get the URL of the repository from GitHub).
3. Open up the `package.json` file and add your name as the author of the package.
4. Install the module dependencies by executing `npm install` in the terminal.

Overview

In this practical, you are going to put together a basic React app that shows the courses in the CS catalog. It looks like this:



Getting situated

I've provided you with the framework of the application, which was created with the `create-next-app` tool. For the moment, we will ignore many of the features of the Next framework. Our focus today is the `index.js` file in `src/pages` that is the root React component for your site and the `components` directory in `src`, where you will find the individual components.

The development server

To run the development server, execute 💻 `npm run dev` in the terminal (or click the "play" button next to `dev` in the NPM Scripts section VSCode). As we discussed in class, there is a certain degree of translation that needs to happen to make your code runnable in the browser. The development server will handle that processing in real time as you update your code and serve your web pages as if it were a normal web server.

Once the development server spins up, you can visit the site in your browser at `localhost:3000`.

Testing

I encourage you to test your code in two ways. Running the development server and interacting with the web page you are building is essential to really understand what is going on. However, I have also given you a collection of tests for your work that you can run with 💻 `npm test` on the command line. You can practice TDD as the tests will all initially fail and then you can slowly get them to pass by following the directions below.

When you start, the number of errors can be daunting, so you can focus your testing a little.

You can run just one test file by putting its name after 💻 `npm test`. For example, 💻 `npm test CourseEntry` will only run the tests in `CourseEntry.test.js`. Notice that I didn't specify the path or even the whole name. Jest will pattern match and run all test files that match. 💻 `npm test Course` will run both `CourseEntry.test.js` and `CourseList.test.js`. You can certainly add the path to make this more specific.

You can also target individual tests inside of a file. You can add `.only` after `test` or `describe` (right before the parentheses) to only run that test or that test suite. Alternately, you can skip a test or test suite by adding `.skip`.

pages

Open up `src/pages/index.js`. This is where we will put the "root" of the application. You will see that the JSX in this file matches what you can see in the development server. Try a simple interaction by adding the following below the "h1" header:

```
<h2>Don't Panic!</h2>
```

When you save the file, the contents displayed in your web browser should change. *Once you have seen this, remove* `<h2>Don't Panic!</h2>`.

The data

We are including the course catalog data as a JSON file. You will find it in `data/cs-courses.json`. If you open it up, you will see that it is an array of objects, one per class. The objects have properties for `number`, `name`, `description`, and optionally, `prerequisites`. Prerequisites are listed as an array of numbers. Example:

```
{
    "number": 431,
    "name": "Computer Networks",
    "description":"Computer networks have had a profound impact on modern society. This course wi
    "prerequisites": [200,315]
}
```

Some prerequisites can be satisfied with one of a collection of options. This is represented using another object with an `or` property like this:

```
{
    "number": 201,
    "name": "Data Structures",
    "description":"In this course we will study the ideas and structures helpful in designing alg
    "prerequisites":[{"or": [145, 150]}]
   }
```

The components

You are going to add two components to the application: `CourseEntry` (which represents a single class), and `CourseList` (which represents the entire list of classes). I have started these already, and you will find the relevant files in `src/components`.

CourseEntry

Take a look at `CourseEntry`. I've given you a working implementation of the component with some basic functionality.

```
export default function CourseEntry({ course }) {
  let summaryClass = "";
  return (
    <details className={styles.description}>
      <summary className={summaryClass}>CSCI {course.number} - {course.name} </summary>
      <p>{course.description}</p>
    </details>
  );
}
```

This makes use of the details HTML tag, which is a less well known HTML tag. It is a simple interactive widget that shows a text summary which can expand to show more detail.

▶ This is a simple details example (click me)

```
<div>
<details>
  <summary>This is a simple details example (click me)</summary>
  <p>When you click it, it expands to show more, well... details. It is actually three separate t
</details>
</div>
```

Looking at the component, you can see that `CourseEntry` takes in a single course as a prop (with the name `course`). It then uses the values in that object to populate the JSX with data.

Javascript Note

The component function is expecting a single object as an argument. By writing the function definition as `function CourseEntry({ course }) {`, we are "destructuring" that single object argument into individual variables by name, i.e., the values of the `courses` property is assigned to the `courses` variable.

CourseList

The `CourseList` component is not as far along. You can see that the component takes in a prop called `courses` (the full list of course objects), but that is all that is included in the skeleton.

```
export default function CourseList({ courses }) {
  return (<></>);
}
```

JSX Note

The empty tags are a convenience in JSX that allow us to have components that are just placeholders (we can also use them to get around the "can only return one element" rule). They don't show up on the web page itself.

Make a list

Now we can flesh out `CourseList`. We need to transform the array of course objects into an array of `CourseEntry` components. Hopefully you immediately thought, `map`! Before the return statement, we are going to build the list of `CourseEntry` components. The basic structure will look like this:

```
const items = courses.map((course) => ());
```

The question is, what is that inner function returning? Well, we want it to be a `CourseEntry` component, and, as it turns out, we aren't restricted to using JSX only in return statements, we can put them anywhere where it makes sense to be talking about a component. Here are we are creating an array of `CourseEntry` components.

```
const items = courses.map((course) => (<CourseEntry />));
```

To use `CourseEntry` in this component, we need to import its definition from the file where it is defined:

```
import CourseEntry from "./CourseEntry";
```

Now that we have the component imported, we can return to creating our `CourseEntry` components. We need to give the `CourseEntry` the course we want it to display, so we will add that as a prop (i.e., within the angle

brackets). Recall that we specify props by name and provide the values in curly brackets. Anything within the curly brackets in JSX is Javascript code, in this case a reference to the `courses` variable.

```
const items = courses.map((course) => (<CourseEntry course={course}/>));
```

When we make a list of React components, it is important that we add a `key` property so that React can uniquely identify individual child components during rendering. When React is diff'ing the virtual DOM and the actual DOM, it needs to have a quick way to tell which component(s) has changed. Without a key, React falls back on simple ordering, and weird things happen if you delete an element out of the middle of a list. The React documentation on lists goes into more detail. The key can be anything, provided it is a unique identifier. In this case, course number will work well (though if we incorporated courses from other departments we would want to use the full name, e.g., "CSCI312"). Note that although `key` is specified like other props, it is not passed through to the component, it is one of several properties used by React itself.

```
const items = courses.map((course) => (<CourseEntry key={course.number} course={course}/>));
```

Now that we have the items, we want to return them. We can only return a single component, so we need a wrapper around all of our `CourseEntry` components. In this instance, a `div` is a good choice. Change the return statement to return a `div`. To put the list of `CourseEntry` components into the `div`, you just need to stick the `items` variable in there (remembering to surround it with curly braces since it is JavaScript). We are going to style the `div` a little, so add `className={styles.listContainer}`as an attribute of the `div` tag (we will learn more about this shortly).

```
return (<div className={styles.listContainer}>{items}</div>);
```

To see the list, you need to put it on the page. Return to `index.js`. First we need to import the `CourseList` component. At the top of the page, add an import statement.

```
import CourseList from "../components/CourseList";
```

While you are doing that, take a moment to notice that the JSON file holding the course data has been imported as the variable `courses`. This is a little trick performed by the development tools where the JSON is parsed into JavaScript if we import it like this. Add a `CourseList` component to the page and pass it the `courses` as a prop like shown below. Recall that we can create React components in JSX, just like we do HTML tags. You should now have a list of courses on your web page.

```
<CourseList courses={courses} />
```

Showing Prerequisites

We are going to add a little bit of interaction to the page. As the user hovers over different courses, we are going to highlight the courses that serve as prerequisites. Since *all* of the `CourseEntry` components will need to see which entry is being hovered over, we are going to need a piece of state that lives above all of them in the hierarchy (i.e., in `CourseList`).

Let's call this piece of state `currentCourse`. At the top of the `CourseList` component, add:

```
const [currentCourse, setCurrentCourse] = useState(null);
```

This gives us our state variable (`currentCourse`) and the method for setting it (`setCurrentCourse`). We passed an initial value of `null` to set the default course selection as no selection.

We need to import `useState` to use it, so put the following line at the top of the file. The curly brace notation is how we import items that are exported from their modules, but that aren't the default export.

```
import { useState } from "react";
```

Push the props down

We need to get the state information to the `CourseEntry` components, so we need to add some new props. In the `CourseList` component, where you create the `CourseEntry` components, add a `currentCourse` property and a `setCurrentCourse` property to the `CourseEntry` component and set their values appropriately (i.e., using the values created by `useState`). This will tell each `CourseEntry` component what the current course is and also provide a way to change it if the component needs to. Finish adding the new properties by going to the `CourseEntry` component and adding them to the destructured `props` argument. Do not use different names for the props as it will break the automated tests.

Changing the current course

We want the current course to change when we hover over it. One of the events we can listen for is the cursor moving over an element on the page. We register a callback for it with `onMouseEnter`, which fires the moment the cursor enters the region defined by the component.

When the cursor enters the region of our `details` component in `CourseEntry.js`, we want it to call `setCurrentCourse` with the course associated with the component. Add the `onMouseEnter` listener to the `details` as shown below.

```
<details lassName={styles.description} onMouseEnter={() => setCurrentCourse(course)} >
```

Now, as the user mouses over the list, the listener will "fire", invoking the `setCurrentCourse` callback to update the current course. To see this in action, add a `console.log(currentCourse)` to the `CourseList` component and then open up the console tab in the browser developer tools (documentation for finding the developer tools Chrome).

Add some styling

Printing things to the console is useful for debugging, we want users to be able to see what they are doing. So, we will change the styling so that the summary of the current course is bolded.

We are using CSS modules for this example, so you will find the stylesheet for `CourseEntry` in `src/styles/CourseEntry.module.css`. I would like you to add a new class in that file that sets the `font-weight` to `bold` (if you aren't sure what this says, please read up on CSS).

```
.current {
  font-weight: bold;
```

Copy

```
}
```

Now we need to apply this CSS class to the `summary` element in `CourseEntry`.To access a class from a CSS module, we import the module (which I've already done for you), and then we can refer to the class name as `styles.class-name` (where `styles` is determined by the import). So, in this instance, it will be `styles.current`.

We only want the style to applied when the course we are rendering is actually the current course, so we are going to set the value conditionally. Create a new variable called `summaryStyle` and initialize it to `""`, i.e., no style by default. If the `course` is equal to `currentCourse`, set the variable to `styles.current` (don't forget to use `===` for the equality check).

Add a `className` attribute to `summary` and set it to `summaryStyle`, e.g, `<summary className={summaryStyle} ...>`. The courses should now show up bold as you mouse over them.

Add styling for prerequisites

Bolding the entry you are over hardly requires state (we could actually do that with pure CSS). We defined state because we are going to also show the prerequisites. Go back to the `CourseEntry.modules.css` CSS module, and add the following styling for prerequisites:

```
.prereq {
  font-weight: bold;
  background: lightslategray;
  color:white;
}
```

Copy

Now, back in `CourseEntry`, we are going to add some more code to check if this course is a prerequisite for the current course. To so we need to check if the course we are rendering is a prerequisite of the current course (either in the pre-requisite list or for an "OR" prerequisite, in the sub list). If the course is a prerequisite of the `currentCourse`, then we will set the style of the `summary` element to our new `prereq` style. A potential implementation is:

```
if (currentCourse && currentCourse.prerequisites) {
  currentCourse.prerequisites.forEach((req) => {
    if (req === course.number) {
      summaryStyle = styles.prereq;
    } else if (req.or) {
      req.or.forEach((altreq) => {
        if (altreq === course.number) {
          summaryStyle = styles.prereq;
        }
      });
    }
  })
}
```

Copy

Finishing up

While this isn't the most complex web application, you now have an example of the React development process and it's component-based approach. Building a more sophisticated application is the same process, just with more and more complex components.

Commit any changes you may have made and then push your changes to GitHub. You should then submit your repository to Gradescope.

Grading

- Should display all courses
- Hovered over courses should be bolded
- Pre-requisites should by highlighted
- Pass all tests
- Pass all ESLint checks

Recall that the Practical exercises are evaluated as "Satisfactory/Not yet satisfactory". Your submission will need to implement all of the required functionality (i.e., pass all the tests) to be Satisfactory (2 points).
