Lab 3

The third lab is about routing and form validation, *objectives*:

1. Understanding how a web application can be split into different pages using the React router.
2. Get experience with passing props between components and combining them with parameters from the url.
3. Get some experience with form validation and the html 5 form validation api.

Background

The assignments here assumes you have a working solution for lab 2, i.e. a working react app with three components: `App`, `ComposeSalad`, and `ViewOrder`.

Assignments

1. We are going to move the `ComposeSalad` and `ViewOrder` to separate pages in the application. First, make sure you know what a router is and the basics of the react router, for example by reading this blog post:
`https://codeburst.io/getting-started-with-react-router-5c978f70df91`

2. We will use the react router. Download it from npm, and add it to your project. The start the development web server, in the terminal type (`ctrl-c` first, if you are running the development web server from lab 2):

```
> npm install react-router-dom
> npm start
```

3. We will add the routing to the `App` component. Open `App.js` and add an import:

```
import { BrowserRouter as Router, Route, Link } from "react-router-dom";
```

All routing components, for example `<Link> and <Route>`, must be children of a router in the React component tree. We use `BrowserRouter` since our application is to be run in a web browser. Note that we renamed it to `Router` which is the name we use in JSX. Add a `<Router>` element to your `App`. The `<Link>` do not to be direct child of `Router` so do not worry if you add additional elements between the routing elements. The html code quickly becomes hard to read when it grows, especially when you add wrapping `<div>` elements for styling. It is a good idea to separate the different parts of the page to separate render functions. When adding a new portion to a page, it is good practice to first create an empty React function component and add the the JSX code there. This is what i have:

```
class App {
  render() {
    return (
      <Router>
        <div className="container py-4">
          <Header />
          </* ViewOrder and ComposeSalad components */}
          <Footer />
        </div>
      </Router>
  );}
}
function Header() {
  return (
    <header className="pb-3 mb-4 border-bottom">
      <span className="fs-4">Min egen salladsbar</span>
    </header>
  );
}
```

4. Next, create a navigation bar for your app. When using the react router, use the `<Link to='my-path'>` for links, instead of the native html element `<a href="my-path">`. Use bootstrap classes to style it, see `https://getbootstrap.com/docs/5.1/components/navs-tabs/`. Bellow is the example code adapted for the react router. Place this in a `Navbar` component and add it to `App`.
Option: If you want an responsive design where the menu collapse on small screens, use navbar `https://getbootstrap.com/docs/5.1/components/navbar/`.

```
function Navbar() {
  return (
  <ul className="nav nav-tabs">
    <li className="nav-item">
      <Link className="nav-link" to="compose-salad">
        Komponera en sallad
      </Link>
    </li>
    {/* more links */}
  </ul>);
}
```

Add the code above to the `App` component. Add a second link for the `ViewOrder` alternative. Go to your browser and klick on the links. The path in your browser changes (url), the selected page is highlighted in the menu, and you use the browser back icon to go to the previous page. However, all pages look the same.

5. Let's change this. Based on the url in the address bar of the browser, we want to render either the `ComposeSalad`, or the `ViewOrder` component. To show a component based on the path, you can use the following JSX-code: `<Route path='/compose-salad' component={ComposeSalad}>`. However, it is not enough to just replace `<ComposeSalad ...>` with `<Route ...>`. `<Route ...>` does not interact directly with the browser. Instead, this is done by another component: `<BrowserRouter>`. `<BrowserRouter>` must be an ancestor to `<Route>`. It is probably easiest to place it in the root (note, `<BrowserRouter>` was renamed to `<Router>` during the import):

```
class App extends Component {
  render() {
    return (
      <Router>
        {/* all stuff you had before */}
      </Router>
    );
  }
}
```

Now we can use `<Route ...>`, but there is one more thing. Both `<ComposeSalad>` and `<ViewOrder>` depends on props passed down from `App`. In this situation, we need to use the `<Route path='compose-salad' render={composeSalad}>` pattern. `composeSalad` is a function that takes the route props and returns a react element. The easiest way to create the instance is to use an arrow function with a JSX expression in the body:

```
const composeSalad = (params) => <ComposeSalad {...params} inventory={inventory}
  onOrderSalad={this.onOrderSalad} />;
```

The spread operator, `...params`, was used to pass the router parameters to `<ComposeSalad>`. This is just shorthand for, `match={params.match}`, repeated for all properties of the `params` object. Lets put it all together:

```
class App extends Component {
  render() {
    const composeSalad = (params) => <ComposeSalad {...params} inventory={inventory}
                       onOrderSalad={this.onOrderSalad} />;
    return (
      <Router>
        {/* header and menu bar */}
        <Route path='/compose-salad' render={composeSalad}/>
      </Router>
    );
  }
}
```

Add a route for the `ViewOrder`.

6. Let's explore how to navigate between pages using JavaScript. When the customer orders a salad, they probably want to go to the check out page. We do not have such page in our app, but let's move to the view order page when the user submits the compose salad form. `<Route>` passes the navigation history to the child component. This is a mutable object and if you push a new url to it, the browser will navigate to that state:

```
this.props.history.push('/view-order');
```

Add the code above to a proper place in the code to your app and test it.

Note, automatically changing the page can be confusing for the user and should normally be avoided. However, in some cases it can improve the user experience, for example if you add a "order and checkout" botton to the compose salad page. I strongly encourage you to take a course in interaction design if you want to work with user interfaces, or make sure there is a designer if your team if you prefer to focus on the technical aspects.

7. *Optional assignment 1*: View a "Page not found" component if the user enters an invalid url like `http://localhost:3000/typing-error`. *Hint*: use the `<Switch>` component and a `<Route>` with no path.

8. *Optional assignment 2*: create a component, `ViewIngredient`, that shows the information from the `inventory` object about an ingredient, i.e vegan, lactose et.c. You should be able to navigate to the `ViewIngredient` component by clicking on an ingredient in the `ComposeSalad` component. To solv this, you should:

- Create the component: `ViewIngredient`.
- Add `<Route path='/view-ingredient/:name' ...>` in `App`.
- Use `<Link ...>` around the ingredient names in `ComposeSalad`.

Note the `:name` part of the path. The router will take the matching text from the url and pass it to your component in its `match.params` object, i.e. `this.props.match.params.name` in your `render()` function.

9. Now your app is split to different pages, where each page have a clear functionality. This is good, do not confuse the user by putting too many unrelated things on one page. Let's move on to another important part of the user experience, form validation and feedback. When a user orders a salad we want to make sure that:

   - one foundation is selected
   - one dressing is selected

   If these conditions are not met, an error message should be displayed and the form submission should stop. We will use html 5 form validation, which have a set of predefined constraints. One of them is `required`, which ensures that a value is provided for the form field. Html is text, and the default action is to send a http request, which is also text based, so in this context "a value" means anything but the empty string. First, let's look at `<select>`. If you do not already done this, make sure that there is an invalid default selection for the `<select>` fileds. This is done by adding an invalid `<option>` at the top of the list:

```
<select required ...>
  <option value=''>make a choice...</option>
  ... more options
</select>
```

   Now press the submit button. You should get an error message from your browser. Let's add your own error message and style it with bootstrap. There are two css classes in bootstrap for this: `valid-feedback`, and `invalid-feedback`. They should be used inside a bootstrap `form-group`. The css will hide the styled html element until any of the pseudo class `:valid` or `:invalid` is set for the input in the `form-group` and the css class `was-validated` is set on any parent element. We do not want to show error messages for fields the user have not interacted with, so set the `was-validated` in your `handleChange` and `handleSubmit` functions:

```
handleChange(event) {
  event.target.parentElement.classList.add("was-validated");
  // ...
}
handleSubmit(event){
  event.preventDefault();
  event.target.classList.add("was-validated");
  // ...
}
```

   If you only want to show error messages when the form is submitted, you skip the part in `handleChange`. Note, in `handleChange`, `event` points to the `<select>` element, but we want to update the style for `<div className="form-group">`. Hence the `parentElement`. Next, add the error message to your `<select>` form group:

```
<div className="form-group">
```

```
  <label htmlFor="foundationSelect">Select foundation</label>
  <select required className="form-control" id="foundationSelect" ...>
    <option value=''></option>
    {/* more options */}
  </select>
  <div className="invalid-feedback">required, select one</div>
</div>
```

There is one more thing you need to do:

```
<form onSubmit={this.handleSubmit} noValidate>
```

The attribute noValidate tells the browser not to show its own error message, and makes bootstrap show the content styled with `valid-feedback/invalid-feedback`. The browser still does html 5 constraint validation and updates the pseudo classes `:valid/:invalid`. You can check if a form is valid by calling `formElement.checkValidity()` on the form element, in `handleSubmit`:

```
if(event.target.checkValidity() === false){ ... }
```

10. *Optional assignment 3*: Add validation of the following constraints:

    - one or two proteins are selected
    - at least four, but not more than fifteen extras are selected

    This error is not related to a single input, but rather a group of checkboxes. It is not a good idea to write an error message on each checkbox, rater add an alert box below the group headline, see `https://getbootstrap.com/docs/4.4/components/alerts/`. You can check the constraint in your `handleChange` and `handleSubmit` functions and store the result in the state:

```
constructor(props) {
  super(props);
  this.state = {
    formErrors: {
      ignoreInvalid: false,
      extras: false,
      proteins: false,
    }
  };
}
```

Use this part of the component state to show and hide your alert boxes. Do not bother the user with an error when the first extra is selected. Wait until the form is submitted. After a failed submission, you want to clear the error as soon as the problem is fixed. The attribute `ignoreInvalid` can be used for this, or you might prefer to call it `submissionFailed`.

Editor: Per Andersson

Contributors in alphabetical order:
Alfred Åkesson
Oscar Ammkjaer
Per Andersson

Home: `https://cs.lth.se/edaf90`

Repo: `https://github.com/lunduniversity/webprog`

This compendium is on-going work.
Contributions are welcome!
Contact: `per.andersson@cs.lth.se`