Lab 2

This lab is about the react and bootstrap, *objectives*:

1. Understanding how a web page can be styled using css classes.
2. Get experience with basic react usage: components and props.
3. Get some experience using html forms.

This lab will take significantly more time to finish compared to lab 1.

Bootstrap

Open the bootstrap documentation to get an overview of the different bootstrap components to choose from. The pages contains examples, so it is easy to reuse the basic building blocks by copying the template code. Note, the examples uses HTML attribute names, but you must use the equivalent JSX names in an React component. Replace `class` and `for` in the examples with `className` and `htmlFor`, for example replace `class="btn btn-primary"` with `className="btn btn-primary"`
`https://getbootstrap.com/docs/5.3/components/buttons/`

Set up

In the first lab you created JavaScript code to manage custom made salads. In this lab you will create a web page where a user can compose and order salads. On the course canvas page you find the instructions for creating a new react project. Follow this and replace `App.jsx` with the one in canvas. You should now have an app with a headline, a container box listing the extras, and a footer. You will use the `Salad` class from lab 1. Remember to install the `uuid` package:

```
npm install uuid
```

You also need to copy the source file to the `src` directory of your vite project and export the `Salad` class so you can import it in your react code:

```
export default Salad;
```

Assignments

1. Study the relevant material for lecture 3 and 4.

2. You will create React components during the lab. You must use function components with the `useState` hook.

3. Download the template component for composing salads (download from Canvas). All source files are stored in `./src` directory of your project:

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

function ComposeSalad(props) {
  const foundationList = Object.keys(props.inventory).filter(name => props.inventory[name
  const [foundation, setFoundation] = useState('Pasta');
  const [extras, setExtra] = useState({ Bacon: true, Fetaost: true });

  return (
    <div className="continer col-12">
      <div className="row h-200 p-5 bg-light border rounded-3">
        <h2>Välj innehållet i din sallad</h2>
        <fieldset className="col-md-12">
          <label htmlFor="foundation" className="form-label">Välj bas</label>
          <select value="Sallad" className="form-select" id="foundation">
            <option key="Sallad" value="Sallad">Salad</option>
            <option key="Pasta" value="Pasta">Pasta</option>
          </select>
        </fieldset>

      </div>
    </div>
  );
}
export default ComposeSalad;
```

Add it to your App:

```
  import ComposeSalad from './ComposeSalad';

  function App() {
    ...
    return (
        ...
        <ComposeSalad inventory={inventory}></ComposeSalad>
        ...
  );}
```

A few observations:

- Remember to export the component name, otherwise you can't instantiate it outside the file.
- Note the JSX code in the function, it looks like HTML with embedded JavaScript.
- `key={name}` helps react track witch part of the DOM to render when data changes, read about it in the react documentation (Quick Start/Rendering lists).
- `className="container"` is a bootstrap class that adds some styling to the page so it looks nicer. Style other html elements using bootstrap css classes, see the bootstrap docs: `https://getbootstrap.com/docs/5.3/layout/containers/`. You fond more relevant examples under "Forms" in the left menu.
- JSX does not have comments, but you can use embedded JavaScript for that:

```
<span>  {/* this part won't do anything right now */}  </span>
```

4. You should now have a template app running with a html form for choosing the foundation. Open the app in a browser and try to select a different foundation. You can't since it's bound to "Sallad": `<select value="Sallad">`. Update the code so the select instead is bound to the component state, see line 5: `const [foundation, setFoundation] = useState('Pasta')`. Change the the state and see that the selection in the app changes. Chang the state either in the source code, or using the "React developer tools" browser plugin (Components tab in the browser developers tools).

Note for students with prior HTML experience: In React you use the `value` attribute in `<select>` instead of using the `selected` attribute in the `<option selected>` element to pre-select one option. (out of scope of the course).

You have created a read only select element and React warns about this. Open the java script console in the web browser to see the message. Warnings are printed here when React finds some problem. Always have the java script console open to see messages from react. Let's follow Reacts suggestion. Add an event handler to handle change events: `<select onChange={handelFoundation}>` and add the function (inside the body of `ComposeSalad()` since it uses `setFoundation`):

```
function handelFoundation(event) {
  setFoundation(event.target.value);
}
```

Now you have a controlled react component. The browser renders the state of the component and the state is updated when the user interacts with the form.

There is one more thing you need to do for the foundation. The list of options is not complete. Generate the `<option>` elements from inventory, see lab 1 assignment 1. You need to modify your code slightly since we use JSX instead of generating strings.

Reflection question 1: The render function must be a pure function of `props` and the component state, the values returned by `useState()`. What happens if the output of the render function is depending on other data that changes over time?

Reflection question 2: In the code above, the `foundations` array is computed every time the component is rendered. The inventory changes very infrequent so you might think this is inefficient. Can you cache `foundations` so it is only computed when `props.inventory` changes? Hint, read about the second parameter to `useEffect`, "Learn React"/"Escape Hatches"/"Synchronizing with Effects", `https://react.dev/learn/synchronizing-with-effects`. Is it a good idea? Read You Might Not Need an Effect: `https://react.dev/learn/you-might-not-need-an-effect`

5. Complete the form in `ComposeSalad` for the protein, extras and dressing. The initial state of the extras has selected Bacon and Fetaost. Make sure these are pre-selected in the html form (feel free to change the initial state when this works). Read all requirements and hints bellow.

   - Read and understand what should be stored in the component state, see `https://react.dev/learn/thinking-in-react#step-3-find-the-minimal-but-complete-represen`
   - We have two states entities related to salads:
     1. The compose salad form state, which changes while the user is composing a salad. This state should be local to `ComposeSalad`.
     2. The list of salads in the shopping basket. This state is created and modified by `ComposeSalad` when the user submits the form, and viewed by `ViewOrder` which we will write later. This state belongs to a common ancestor of the producer/viewer, the `App` component.
   - You will use a html form with select and check boxes to compose the salad. Use the controlled component pattern. The html form must view the `ComposeSalad` state. Not the other way around (never read the form state from the DOM). Instead, add listeners and update the state based on user actions. The React state is the "single source of truth". (Follow the pattern from the foundation `<select>`, state and event handler above)

- The user must select: one foundation, one protein, two or more extras, and one dressing. The `Salad` class from lab 1 is not suitable to work with here. Instead use a state variable containing a string for each of foundation, protein and dressing. Use a state variable containing an object for the extras, as indicated by the initial state of `extras`. Add and remove properties from the object as the user selects and removes the extras from the salad.
- Remember that state must be immutable. Do not modify the extras object, make a copy for each change. See `https://react.dev/learn/updating-objects-in-state`
- For checkboxes, the state of the DOM-element is stored in the attribute `checked` (for other `<input>` types, the DOM state is stored in the property `value`).
- Do not assign `undefined` to a html attribute. It might be converted to the string `"undefined"`, which is `true`. To avoid this, you can use the JavaScript short circuit behaviour of `||` `<input checked=(extras['Tomat'] || false)>`, or do an explicit type conversion: `Boolean(extras['Tomat'])`, or `!!extras['Tomat']`.
- `<input>` elements have a `name` attribute. Use this to pass additional information to your event handlers: `<input type=checkbox name='Tomat' checked=!!extras['Tomat']>`. Read the name attribute from the event object: `event.target.name`.
- Feel free to use this code, but only if you understand it:
  `const newExtra = { ...extra, [event.target.name]: event.target.checked}`
- You might get a long an hard to read render function as you add more parts to the form. Having many `<div>`, `<select>`, and `<option>` elements in the same render funktion makes it really hard to see what belongs to which part of the GUI. Also, parts of the form is very similar, for example `<select>` for foundation, protein, and dressing. It is good to divide large components to smaller, and use components for repeated code, for example the selects for foundation, protein and dressing. This is what I introduced in my solution:

  ```
      <Select label="Välj innehållet i din sallad"
        onChange={handelFoundation}
        value={foundation}
        options={foundationList}>
      </Select>
  ```

  Reflection question 3: Should you move the foundation state to the `Select` component above? To answer this you need to consider what happens when the user submits the form.
- Remember that the `id` of each HTML element must be unique on a page. You will have three instances of the `<Select>` component and each of the inner `<select>` HTML elements must have different ids. The react hook `useId()` generates id that are unique and persistent over renderings:

  ```
  function Select({ label, onChange, value, options }) {
    const id = useId();
    return (
      ...
      <label htmlFor={id} className="form-label">{label}</label>
      <select onChange={onChange} value={value} className="form-select" id={id}>
    ...
    );
  }
  ```

Layout

- Layout is not the focus of this lab. Do not spend to much time fixing how the page look.
- Bootstrap default is a 12 column layout. You can set the number of columns. This example places 4 elements per row:

```
<div className="row row-cols-4">
  <span className="col">One</span>
  <span className="col">Two</span>
  <span className="col">Three</span>
  <span className="col">Four</span>
  <span className="col">Five</span>
</div>
```

- Use `className="mt-4"` (margin top) to add more space above an element.
- More details can be found in the documentation
  `https://getbootstrap.com/docs/5.3/layout/grid/`
  `https://getbootstrap.com/docs/5.3/utilities/spacing/`

- Requirements:
  - You do not need to support portion size (gourmet salad).
  - You may assume correct input for now, we will add form validation in the next lab.
  - One learning outcome of this lab is for you to get familiar with html and css. Therefore you must use native html tags, e.g. `<input> and <select>`, and style them using `className`. Most real world applications use frameworks, such as ReactBootstrap, which encapsulate the html tags and styling in react components. You should use this approach in the project but not in the labs.
  - You must not read the form DOM state, use event handlers and update the component state variables.
  - Your code must be flexible. If the content of `inventory` changes, your form should reflect these changes. Use iterations in JavaScript (`Array.map` is recommended), avoid hard coding html elements for each ingredient (you may not assume which ingredients are present in inventory). We will start with an empty inventory in lab 4 and add ingredients by fetching them from a rest server. Your `ComposeSalad` must support this.
  - React is based on the *model-view* design pattern. `ComposeSalad` is the view and component state and `this.props` is the model. `ComposeSalad` contains all functionality for viewing the model. `Salad` is not aware of how it is visualised. Do not put any view details, such as html/react elements, in the `salad` class. This makes your data structures portable. You can reuse the `Salad` class in an Angular or Vue.js application.

Reflection question 4: What triggers react to call the render function and update the DOM?
Reflection question 5: When the user change the html form state (DOM), does this change the state of your component?
Reflection question 6: What is the value of `this` in the event handling call-back functions?

6. *optional assignment:* add a “Caesar Salad” quick compose button. When the user clicks the button, the form is pre-filled with the selections for a Caesar sallad.

7. Handle form submission. The salad in the form should be added to a shopping cart when the user submits the form. The shopping cart should be stores in the `App` component.

   - When the form is submitted, you must create a `Salad` object from assignment 1 to store it.
   - The shopping cart is a list of salad objects, use an array if you did not do the optional task in lab 1.
   - The list of salads must be stored in `Apps` state since it will be use by other components later. When the form is submitted, create a Salad object in the callback function of `ComposeSalad` and pass it to `App`. Remember, the user might want to compose several salads, so make sure to copy/create objects when needed.
   - `onSubmit` is the correct event for catching form submission. Avoid `onClick` on the submit button, it will miss submissions done by pressing the enter key in the html form.
   - Clear the form after a salad is ordered, so the customer can start composing a new salad from scratch.
   - The default behaviour of form submission is to send a http GET request to the server. We do not want this since we handle the action internally in the app. Stop the default action by calling `event.preventDefault()`. If you forget this then the app will be reloaded and JavaScript/component state will be lost (submit will result in an empty shopping cart).

   Reflection question 7: How is the prototype chain affected when copying an object with `copy = {...sourceObject}`?

8. Create a react component, `ViewOrder`, to view the shopping cart. The shopping cart should be passed from `App` using props. Instansiate the `ViewOrder` component in `App`, i.e. `<ViewOrder shoppingCart={shoppingCart}>`. This demonstrates the declarative power of react. When the state changes all affected subcomponents will automatically be re-renderd.

   An order can contain several salads. Remember to set the `key` attribute in the repeated html/JSX element. Avoid using array index as key. This can break your application when a salad is removed from the list. This is explained in many blog posts, for example `https://medium.com/@robinpokorny/index-as-a-key-is-an-anti-pattern-e0349aece318`.
   Hint 1: use the `uuid` property in the `Salad` objects as key.

9. *Optional assignment 1:* Add a remove button to the list of salads in the `ViewOrder` component. Remember, `props` are read only. The original list is in the `App` component.

10. *Optional assignment 2:* Add functionality so the user can edit a previously created salad. Add an edit button to each row in the list of salads in the `ViewOrder` component. For conditional rendering of a component you can use any JavaScript condition, `if...then...else` or `{editMode && <ComposeSalad edit={saladToEdit}/>}` . You also need modify the `ComposeSalad` component so it can be used for editing. Use `props` to pass the salad to be edited. If `App` will not initialise this prop, so it will be `undefined`. Use this to determine if the `ComposeSalad` component is in create or edit mode when needed, for example the the text for the submit button (create/update). Note: do not update the salad object in the order until the update button is pressed. This will change the state of `App`. Make sure to copy objects when needed and call the right setState function.

    The edit scenario is a good use case for a modal wrapper around the `ComposeSalad` component. For edit, a pop-up window will appear, and when done the user is back in the

list of the salads.
Hint: Do this assignment in two steps, first add the functionality to view the salad, then continue with changes needed to save the updated salad.

11. This is all for now. Make sure you do not have any warnings from React. Open the console, reload the app, compose a salad and view it. In the next lab we will introduce a router and move the `ComposeSalad` and `ViewOrder` to separate pages.

Editor: Per Andersson

Contributors in alphabetical order:
Ahmad Ghaemi
Alfred Åkesson
Anton Risberg Alaküla
Mattias Nordal
Oskar Damkjaer
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`