MUNI inQool a.s.

5. React - advanced

Fragments 1
Conditional rendering 3
React Router 5
Lists and keys 7
Forms 8
Theming 9
Material-UI: custom theme 10

This lesson is aimed at building on your React basics knowledge. Now, we are going to show you the ways of doing things even better than in the previous lesson. All examples extend the Tic-Tac-Toe game created in the previous lesson.

Fragments

React fragments are used to group JSX elements without the need to create unnecessary DOM elements. In earlier versions of React, they were not available. If you wanted to return multiple React elements you had to wrap them into a `<div />` element or an array. Let's demonstrate with an example.

Before
Let's come up with something like this.

```
function Title() {
  // This is not allowed even in the newest version of React.
  // You always have to return a single React element.
  return (
    <h1>I am the first line</h1>
    <p>Hello, my name is second line</p>
  );
}
```

So in the past, the solution to this problem was something like this.

```
function Title() {
  return (
    // Cool, function now returns a single element...
    <div>
      <h1>I am the first line</h1>
      <p>Hello, my name is second line</p>
    </div>
  );
}
```

But imagine, that you don’t want to wrap them into a `<div />` element, because for example layout breaks it breaks your layout.

Now
Now React provides a solution to this problem in form of React Fragment. All will be shown on examples in our Tic-Tac-Toe application. Fragment will not be printed into virtual DOM at all, it is just a wrapper so you don’t need to create additional elements.

```
function returnTwoTexts() {
  return (
    // Cool, function now returns a single element...
    <React.Fragment>
      <h1>I am the first line</h1>
      <p>Hello, my name is second line</p>
    </React.Fragment>
  );
}
```

You can also use a shorthand by omitting the name completely.

```
function returnTwoTexts() {
  return (
    // Cool, function now returns a single element...
    <>
      <h1>I am the first line</h1>
      <p>Hello, my name is second line</p>
    </>
  );
}
```

Conditional rendering

It’s a pattern where part of JSX is rendered depending on evaluation of some condition.
To explain it we can use an example from the previous lesson. Let’s start with it.

```
function renderIcon(value) {
  if (value === 'X') {
    return <Cross className={classes.Xclass} />;
  } else if (value === 'O') {
    return <Circle className={classes.Oclass} />;
  } else {
    return <Typography variant="text">{value}</Typography>;
  }
}
```

`renderIcon` function accepts one parameter and depending on its value returns *Cross*, *Circle* or *Typography* component.

This logic can be moved (and it is a very common approach) directly into JSX.

```
export function Square({ value, handleClick }) {
  const classes = useStyles();

  return (
    <Grid item xs={1}>
      <Card className={classes.card}>
        <CardActionArea
          className={classes.area}
          onClick={() => handleClick('X')}
        >
          {/* Logic from function moved into JSX */}
          {value !== 'X' && value !== 'O' && (
            <Typography variant="text">{value}</Typography>
          )}
          {value === 'X' && <Cross className={classes.Xclass} />}
          {value === 'O' && <Circle className={classes.Oclass} />}
        </CardActionArea>
      </Card>
    </Grid>
  );
}
```

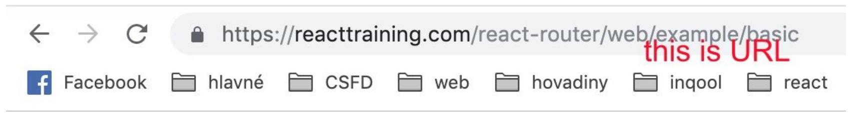
Or you can also use ternary operators “[condition] ? [return-if-true] : [return-if-false]”.

```
{value !== 'X' && value !== 'O' ? (
  <Typography variant="text">{value}</Typography>
) : value === 'X' ? (
  <Cross className={classes.Xclass} />
) : (
  <Circle className={classes.Oclass} />
)}
```

In our case we have 3 possible return values, so we need to nest our ternary operator. It’s not the best practice, because readability of the code suffers, but in some cases it’s acceptable.

React Router

As we spoke before about Single Page Application, it doesn't mean that we are forced to use only one single URL.


https://reacttraining.com/react-router/web/example/basic

Routing is not difficult to understand, behavior is very similar to conditional rendering behavior described before with one difference - rendering content is not dependent on evaluation of condition but on matching the url.

Collection of navigation components is not part of React library, it is separated into `react-router-dom` package. It works like this (yes, copied from docs, I added some explanation comments).

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

// Imagine index page
function Index() {
  return <h2>Home</h2>;
}

// Imagine about page
function About() {
  return <h2>About</h2>;
}

// Imagine page about users
function Users() {
  return <h2>Users</h2>;
}

function AppRouter() {
  return (
    <Router>
      <div>
        {/* Navigation will be rendered on all pages */}
        <nav>
          <ul>
            <li>
              <Link to="/">Home</Link>
            </li>
            <li>
              <Link to="/about/">About</Link>
            </li>
            <li>
              <Link to="/users/">Users</Link>
            </li>
          </ul>
        </nav>
        {/* Index page show only! (used "exact") if pathname is "/" */}
        <Route path="/" exact component={Index} />

        {/* About page show if pathname is "/about/*" */}
        <Route path="/about/" component={About} />

        {/* Page about users show if pathname is "/users/*" */}
        <Route path="/users/" component={Users} />
      </div>
    </Router>
  );
}

export default AppRouter;
```

Lists and keys

We can add into JSX as much logic as we want to. The only limitation should be readability of the source code.

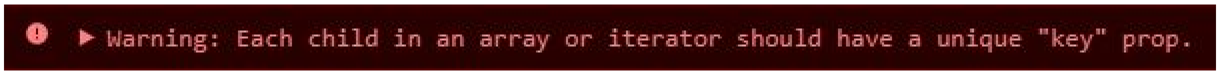
Iterating over an array in JSX is also a very common approach. It's how the lists are built in React applications. Syntax of jS `map` function is following:

```
array.map((currentValue, index, arr) => {
  // modify currentValue
  return currentValue;
});
```

Let's go back to an example of a list we used in the `Board` component in the previous lesson..

```
<Grid item container spacing={1}>
    {/* IndexValues: [0, 1, 2, 3, 4, 5, 6, 7, 8] */}
    {IndexValues.map((i) => (
        <Square onClick={() => onSquareClicked(i)}>
            <PlayerIcon>{board[i] ?? i}</PlayerIcon>
        </Square>
    ))}
</Grid>
```

You should see the below error in your developer console when inspecting the app.



```
▶ Warning: Each child in an array or iterator should have a unique "key" prop.
```

Keys should be given to the elements inside the array to give the elements a stable identity. If you choose not to assign an explicit key to list items then React will by default use indexes as keys. React then compares elements with previous ones and re-render only changed elements. We don't recommend using indexes for keys if the order of items may change. This can negatively impact performance and may cause issues with component state. Though, it is highly recommended to use keys. Moreover, React warns you when you don't have proper keys used.

```
<Grid item container spacing={1}>
    {/* IndexValues: [0, 1, 2, 3, 4, 5, 6, 7, 8] */}
    {IndexValues.map((i) => (
        <Square key={i} onClick={() => onSquareClicked(i)}>
            <PlayerIcon>{board[i] ?? i}</PlayerIcon>
        </Square>
    ))}
</Grid>
```

Key always needs to be on the top-most element you are rendering as a list item, in our case the `Square` component. Although we are in reality still just using indexes as keys, the warning is gone since we explicitly specified those keys (thus React believes we know what we are doing).

Forms

HTML form elements work a little bit differently than static elements because they naturally need to store users input. There are 3 basic HTML form elements: `input`, `textarea`, `select`.

In React there are two approaches we can choose when dealing with form components. They can either be **controlled** or **uncontrolled**. The main difference is that **controlled** elements have their value directly controlled by React while with **uncontrolled** elements you access their value through their `form` element.

Let's demonstrate the differences in an example. To create a new controlled input element we need a state (`useState` hook) and a `onChange` handler.

```
const ControlledForm = () => {
  // We need to provide an initial value to our state
  const [email, setEmail] = useState('');
  return (
    <form
      onSubmit={e => {
        e.preventDefault();
        console.log({ email });
      }}
    >
      <input value={email} onChange={e => setEmail(e.target.value)} />
      <button>Submit</button>
    </form>
  );
};
```

Note:
We must provide an initial value to the useState hook, otherwise React would interpret the input as uncontrolled at first and then complain that you are switching between two approaches after setting it’s value.

To create an uncontrolled input, we must provide it with a name in order to find it in our form in onSubmit event.

```
const UncontrolledForm = () => {
  return (
    <form
      onSubmit={(e: React.FormEvent<HTMLFormElement>) => {
        e.preventDefault();
        console.log({
          email: ((e.target as HTMLFormElement).email as HTMLInputElement)
            ?.value,
        });
      }}
    >
      <input id="email" name="email" />
      <button>Submit</button>
    </form>
  );
};
```

Theming

Often you want to declare a consistent set of rules for the styling of your app, a **theme** (e.g. Dark Mode can also be a theme). It is recommended to extract repeating values in your CSS/JS into constants (good programming practice). For example you may want all of your title texts to have the same font size, and all your divs have the particular padding in the application. To avoid unwanted “magic values” it’s much smarter to extract your paddings, colors etc. into constants. Then, you can easily change them for all your elements in one place. We are going to show you what’s Material-UI approach is.

Material-UI: custom theme

The theme specifies the color of the components, darkness of the surfaces, level of shadow, appropriate opacity of ink elements, etc.

If you wish to customize the theme, you need to use the ThemeProvider component in order to inject a theme into your application. However, this is optional, Material-UI components come with a default theme.

```
import { createMuiTheme } from '@material-ui/core/styles';
import purple from '@material-ui/core/colors/purple';
import green from '@material-ui/core/colors/green';

const theme = createMuiTheme({
  palette: {
    primary: purple,
    secondary: green,
  },
  status: {
    danger: 'orange',
  },
});
```

ThemeProvider relies on the context feature of React to pass the theme down to the components, so you need to make sure that the ThemeProvider is a parent of the components you are trying to customize.

```
const App = () => {
  return (
    <MuiThemeProvider theme={theme}>
      {/* Rest of you app */}
    </MuiThemeProvider>
  );
};
```

You can read more about MUI Theme here and explore all it’s properties here.