5. React - basics

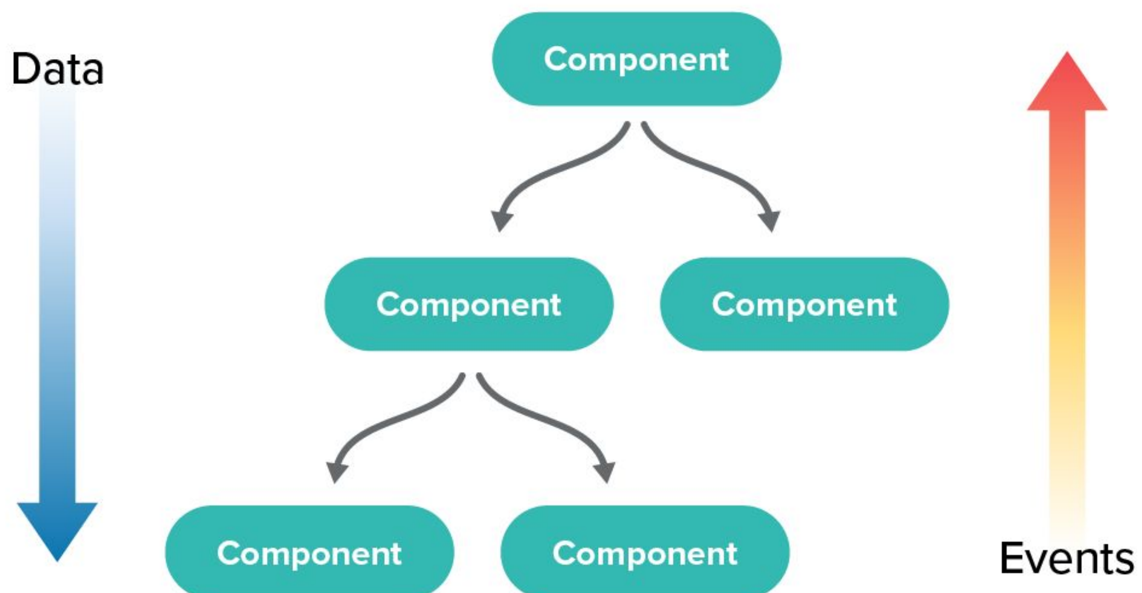
What is React? 1
React component tree 2
Function and Class components 2
Props 3
PropsWithChildren 4
Rerendering 5
State 5
State in Class components 5
State in Function component 6
Tic-Tac-Toe in React 7
Grid system 8
Basics 9
Breakpoints 10
Implementation 11
Custom styles in Material UI 12
makeStyles and useStyles 12
Implementation 13
Implementing logic 14
Mouse events 16
Implementation 17
Material UI icons 19

What is React?

React is a declarative, efficient and flexible JavaScript library for building user interfaces. Components allow you to split the UI into independent, reusable pieces, and to think about each piece in isolation.

React component tree

React components are structured into a hierarchical tree structure. The structure ensures a one-way data flow (via props). The children cannot update the data it receives from its parent. If data needs to be updated, children can receive another prop from the parent, that contains a function to update it. Each component is declaring what will be rendered on the screen based on the props it renders.



Function and Class components

There are two basic ways to define a React component. In the previous lesson it was created by CRA component App. This type of component is called a Function *Component*.

```
const App: React.FC = () => {
    return <div>Hello world</div>;
}
```

You can also write this component via ES6 Class (syntax was mentioned in lesson 2). Class components have a set of methods, each of them evoked in concrete time during the

component's life cycle. In the following example, the App component has only a single lifecycle method `render` which serves the same purpose as the return value of a function component. In other words, it renders JSX on the screen.

```
class App extends React.Component {
  render() {
    return <div />;
  }
}
```

Since we will be using the newer function components and hooks we won't go deep into the class component's lifecycle and other related topics. While React developers said that there are no plans of deprecating class components, we can safely assume that they won't be receiving any new features.

Props

React components accept arbitrary values (called "props" which is short for properties) and return React elements describing what should appear on the screen. We can expand the example of the function component above with props like this.

```
type Props = { name: string };

const App: React.FC<Props> = props => {
  return <div>Hello {props.name}</div>;
};
```

For example `Typography` component from MaterialUI in the following snippet uses a `variant` prop.

```
<Typography variant="h5">Sign in</Typography>
```

It's common to use destructuring on props, so many examples you come across will look like this.

```
const App: React.FC<Props> = ({ name }) => {
  return <div>Hello {name}</div>;
};
```

PropsWithChildren

When you inspect the type of props you get when using TypeScript, you can see that your Props type is wrapped in a PropsWithChildren helper which adds the `children: React.ReactNode` to your props. The `children` prop is a special prop that each component has and it's value is equal to what you pass to it between it's opening and closing tags.

```
type Props = { title: string };

const Button: React.FC<Props> = ({ title, children }) => {
  return <button title={title}>{children}</button>;
};

// Usage
<Button title="Click me">OK</Button>
```

Here you can see one simple example where a button component simply passes the children prop to the DOM element it renders.
You can also change the type of children prop (which TypeScript will also remind you of) to anything else. Most commonly it's used to expect an array of components or in a pattern called **Render props**. You can find this pattern used in various component libraries and it may look something like this.

```
// Type of children is (close: () => void) => ReactNode instead of ReactNode
<Dialog>
  {close => (
    <div>
      <p>Hello</p>
      <button onClick={close}>Back</button>
    </div>
  )}
```

```
</Dialog>
```

Rerendering

Important note about props is that the component is rerendered each time the props change. In class components, you could use multiple methods to change this behaviour, but in function components there was no such a thing until Hooks came along. Hooks are what allows functional components with linear code execution to have things like persistent state and other features, but more on Hooks later.

State

Each component (class or function) is able to store local state. Below we will go over differences in these two approaches.

State in Class components

```
type Props = {};
type State = { foo: string };

class App extends React.Component<Props, State> {
  constructor(props: Props) {
    super(props);
    this.state = {
      foo: 'bar',
    };
  }

  render() {
    return <div>{this.state.foo}</div>;
  }
}
```

State and props are key mechanisms in React and both invoke rerendering the component after the change of it's values. It means that if we have a button element calling `setState` method (reserved method in Class component, which sets values into state) we can change the value of

`this.state.foo` like this...

```
<button onClick={() => this.setState({ foo: 'baz' })}>Click</button>
```

... and then the state will contain the value of `"baz"` under the `foo` key and the component will also be rerendered.

State in Function component

```
import React, { FC, useState } from 'react';

const App: FC = () => {
  const [foo, setFoo] = useState('bar');
  return <div>{foo}</div>;
};
```

In function components there is no *constructor* nor reserved *setState* method. We need to use the already mentioned Hooks, specifically `useState` hook.
This hook returns a tuple consisting of current value (`foo`) and setter (`setFoo`) and takes an optional value which if provided will be the initial value of this state.

Note:
Array destructuring is used to extract the value and setter from the returned value, therefore you can choose any names for these two variables. You should still follow the convention `[x, setX]` whenever possible though.

Note:
If an initial value is provided, the type of the return value is inferred from it. However if there is no initial value you should explicitly provide the type. Also in this case the type will automatically be expanded with `undefined`.

```
const [foo, setFoo] = useState<string>(); // Type of foo is string | undefined
```

There may be cases (we will even come across one in this week's task) where you want to change state based on its current value. Most commonly this is used for boolean state which you simply want to toggle between true and false. For this use case, the set function instead of value can accept a function that provides you with the real and up to date value of its state (this is usually most important when using state setter in asynchronous code).

```
const [flag, setFlag] = useState(false);
const toggleFlag = () => setFlag(prev => !prev);
```

Tic-Tac-Toe in React

Now let's demonstrate all the basics on a Tic-Tac-Toe game example. Our starting point will be a clean Create React App project.

```
yarn create react-app tic-tac-toe --template typescript
```

First create a new `components` folder and do some minor cleanup of CRA template. This project will contain multiple components declared in multiple files so it's useful to organise your files a bit. We can now prepare basic components of the game. Create two new files, `Square.tsx` and `Board.tsx` in the components folder.

Square component will represent one field in the tic tac toe board.

```
import React, { FC } from "react";

const Square: FC = ({ children }) => (
    <div>{children}</div>
);

export default Square;
```

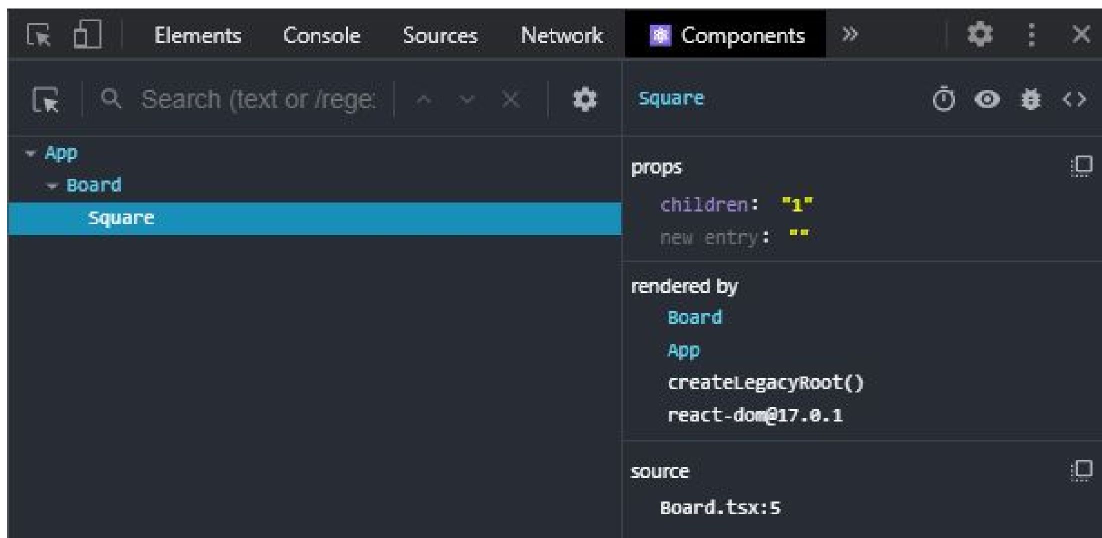
Board component will represent the board of the tic tac toe game, containing Squares.

```
import React, { FC } from "react";
import Square from "./Square";

const Board: FC = () => (
    <Square>1</Square>
);
```

```
export default Board;
```

Now we need to import the Board component in App.tsx and add it into it’s JSX. At this point we created a tree of components App -> Board -> Square. We can inspect it in the Chrome React extension.



Next we should prepare our components and style them. For that we will be using the MaterialUI framework.

```
yarn add @material-ui/core @material-ui/icons @material-ui/styles
```

Note:
Since @material-ui is written in TypeScript and comes with it’s own typings we don’t need to instal any additional @types packages.

Grid system

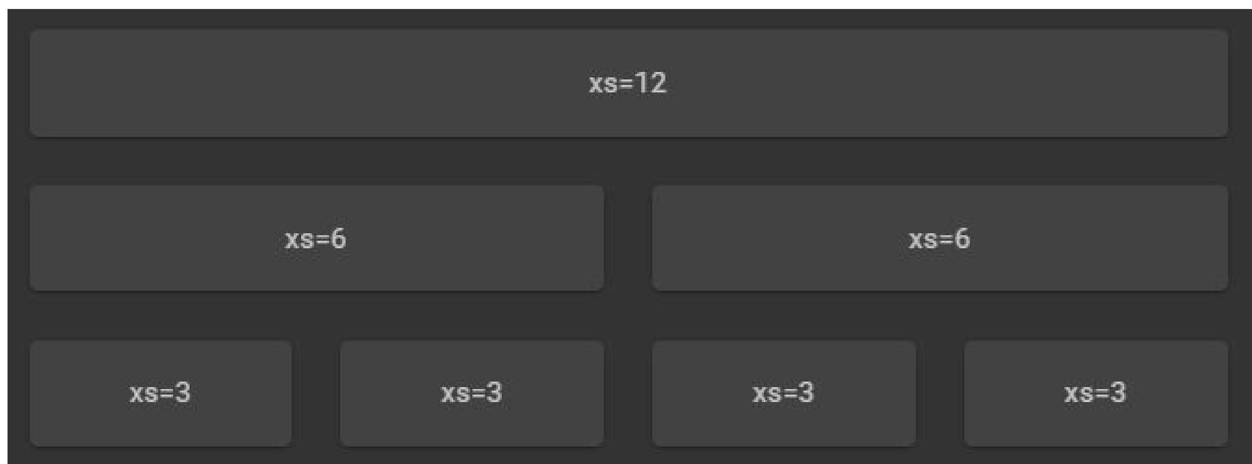
The grid system creates visual consistency between layouts and allows flexibility across many different screen sizes and orientations. It is more related to design of web pages and not something specific to React or Material UI. The grid system from Material UI we will be using follows many common principles of any grid system:

- Based on a 12-column grid layout
- Supports two types of components: containers and items

- Widths of items are specified in percentages, which allows us to create a fluid and responsive layout, relative to their parent element
- In Material UI, there are five grid breakpoints: xs, sm, md, lg and xl

Basics

Breakpoints refer to screen widths at which the specific rule is applied (you can see the exact values and more detailed explanation in [MUI's documentation](#)). Sizes of breakpoints are specified in a mobile first approach, therefore by specifying `xs` size, we also set all larger sizes.



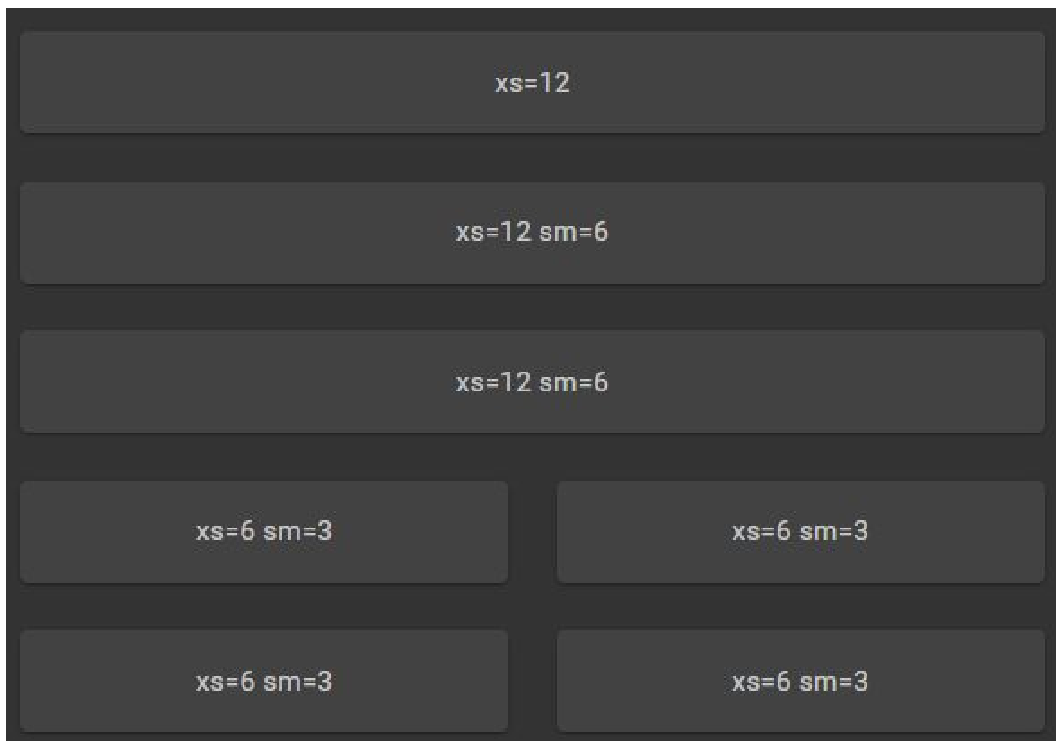

```
<Grid container spacing={3}>
    <Grid item xs={12}>xs=12</Grid>
    <Grid item xs={6}>xs=6</Grid>
    <Grid item xs={6}>xs=6</Grid>
    <Grid item xs={3}>xs=3</Grid>
    <Grid item xs={3}>xs=3</Grid>
    <Grid item xs={3}>xs=3</Grid>
    <Grid item xs={3}>xs=3</Grid>
</Grid>
```

Breakpoints

By providing multiple values, the grid items will be sized based on current breakpoint.

On screens larger than `sm` breakpoint (>=600px):

On screens smaller than `sm` breakpoint (<600px):



```
<Grid container spacing={3}>
    <Grid item xs={12}>xs=12</Grid>
    <Grid item xs={12} sm={6}>xs=12 sm=6</Grid>
    <Grid item xs={12} sm={6}>xs=12 sm=6</Grid>
    <Grid item xs={6} sm={3}>xs=6 sm=3</Grid>
    <Grid item xs={6} sm={3}>xs=6 sm=3</Grid>
    <Grid item xs={6} sm={3}>xs=6 sm=3</Grid>
    <Grid item xs={6} sm={3}>xs=6 sm=3</Grid>
</Grid>
```

Implementation

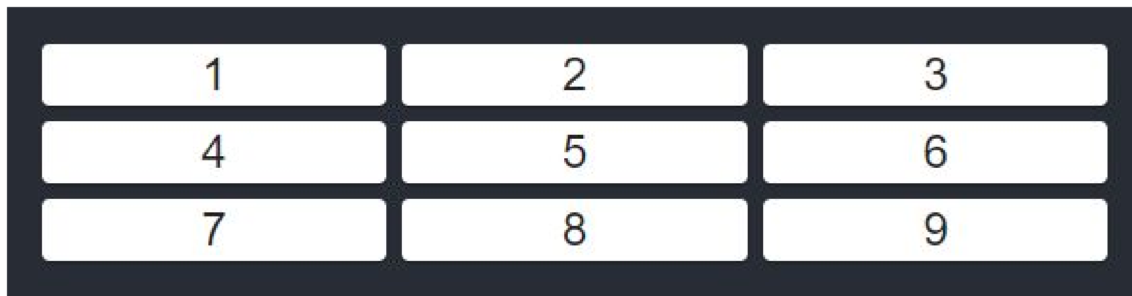
Our tic tac toe board will consist of 3 rows and 3 columns of `Card` components spaced out in a `Grid`. We can use the fact that by default Grid wraps it's items into the next row if they don't fit into it's 12 columns. For styling we will use a `Container` component that will make sure our board isn't stretched on large screens and for styling the squares we can use the `Card` component and it's `CardActionArea` which take care of distinguishing between squares and also styling the click action.

```
const Square: FC = ({ children }) => (
    <Grid item xs={4}>
        <Card>
            <CardActionArea>
                <Typography variant="h5">{children}</Typography>
            </CardActionArea>
        </Card>
    </Grid>
);
```

```
const Board: FC = () => (
    <Container maxWidth="sm">
        <Grid container spacing={1}>
            <Square>1</Square>
            <Square>2</Square>
```

```
            <Square>3</Square>
            <Square>4</Square>
            <Square>5</Square>
            <Square>6</Square>
            <Square>7</Square>
            <Square>8</Square>
            <Square>9</Square>
        </Grid>
    </Container>
);
```

Output on the screen should now look like this:



Custom styles in Material UI

Material UI provides a solution for custom styles based on the CSS-in-JS concept. All styles are held in js files, there is no need to keep static `.css` files. It comes with many powerful features like dynamic styles, theming… We will be using this modern solution too, so let's describe how.

makeStyles and useStyles

The most crucial part is located in package `@material-ui/styles` and it is the `makeStyles` function. With `makeStyles` we can create a hook `useStyles` which injects styles into our app`s markup during runtime. These styles can be found inside the head selection when you inspect the running app.

```
<style data-jss data-meta="makeStyles">
  .makeStyles-card-1 {
    border-radius: 0;
  }
  .makeStyles-action-2 {
    min-height: 150px;
  }
</style>
```

The mechanism behind this feature makes sure that unique class names are always generated for each of our styles object and provides these unique class names for us to use, without having to worry about how it works.

Implementation

Now we can provide some custom styles for our `Square` component to make sure it looks more like a grid and also has a larger area.

```
import { makeStyles } from "@material-ui/styles";

const useStyles = makeStyles({
    card: {
        borderRadius: 0,
    },
    action: {
        minHeight: 150,
    },
});

const Square: FC = ({ children }) => {
    const classes = useStyles();
    return (
        <Grid item xs={4}>
            <Card className={classes.card}>
                <CardActionArea className={classes.action}>
                    <Typography variant="h5">{children}</Typography>
                </CardActionArea>
            </Card>
```

```
        </Grid>
    );
};
```

With these styles applied it should look a bit closer to an actual tic tac toe board.

Implementing logic

Now that we have some basic layout done, we can start working on implementing the functionality of tic tac toe. For a game of tic tac toe to play out we need to keep track of two things. State of the board and which player is supposed to go next. For both of these things we can learn the `useState` hook.

```
const Board: FC = () => {
    // State
    const [player, setPlayer] = useState<Player>("X");
    const [board, setBoard] = useState<BoardState>({});

    // ...
```

With TypeScript we can neatly declare what values our two states should accept. For Player type we can use the X and O string literal values.

```
type Player = "O" | "X";
```

Since we know that our board has exactly 9 fields and each field will either be empty or selected by either player we have few options.

First and the most simple approach would be to use an array of optional players but this does not limit the board to have exactly 9 values.

```
type BoardState = (Player | undefined)[];
```

Next option that solves the length problem is to use a tuple type. This type technically describes our board perfectly but from a code style perspective is not very clean and readable. It would also make it harder to insert elements into arbitrary indexes.

```
type BoardState = [Player?, Player?, Player?, Player?, Player?, Player?,
Player?, Player?, Player?];
```

Here TypeScript's utility types can come to rescue. We can write out the expected indexes as a numeric literal type (there are also more advanced automatic options but for this use case this is sufficient). With these indexes we can create a `Partial Record` of players. This will allow us to write to an arbitrary index of our "array" object and also provides a neat and safe type to describe our index.

```
type Indexes = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8;
type BoardState = Partial<Record<Indexes, Player>>;
```

Now that we have our state described, we can move on to the next task, implementing a function that sets the board's value on a given index and switches to the next player.

```
const Board: FC = () => {
    // State
    const [player, setPlayer] = useState<Player>("X");
    const [board, setBoard] = useState<BoardState>({});
```

```
// Handlers
const onSquareClicked = (index: Indexes) => {
    // Disallow clicking on already set square
    if (board[index]) {
        return;
    }

    // Mark square with current player's symbol
    setBoard((b) => ({ ...b, [index]: player }));
    // Switch to other player
    setPlayer((p) => (p === "O" ? "X" : "O"));
};

// ...
```

Note:
In both cases we want to update the value based on the previous one, therefore we use the function setter variant that provides us with the previous value. With the board we want to keep the old state and just add the new change. For the player we want to set it to the other one.

Mouse events

In order to interact with our app we need to implement and use event handlers. React normalizes events so that they have consistent properties across different browsers. There are many different events and for handling the direct interaction (click or touch) we can use the `onClick` handler. In general, any DOM element can have an onClick handler, but there are few recommendations to keep in mind. Mostly for accessibility purposes, it’s a good practice to provide an onClick handler only on elements that are supposed to be interactable. You can see this property with Chrome’s inspect element tool under the `Role` key.

This makes sure that for example users that can’t use a mouse can select this element by hitting tab on their keyboard.

Implementation

Since CardActionArea is supposed to be interactable and Material UI follows these accessibility principles, it is the correct element to provide a onClick handler to. You can test it out with this code.

```
<CardActionArea onClick={() => alert(children)}>
```

Now that our Square component is properly clickable, we can expand it to expect a callback that is executed when it is clicked.

```
type Props = {
    onClick: () => void;
};

const Square: FC<Props> = ({ onClick, children }) => {
```

```
    const classes = useStyles();
    return (
        <Grid item xs={4}>
            <Card className={classes.card}>
                <CardActionArea className={classes.action} onClick={onClick}>
                    <Typography variant="h5">{children}</Typography>
                </CardActionArea>
            </Card>
        </Grid>
    );
};
```

This change will give us a compilation error since all usages of the Square component in Board are now expecting onClick prop that they did not receive. Let's fix that.

```
return (
    <Container maxWidth="sm">
        <Grid container spacing={1}>
            {([0, 1, 2, 3, 4, 5, 6, 7, 8] as const).map((i) => (
                <Square onClick={() => onSquareClicked(i)}>
                    {board[i] ?? i}
                </Square>
            ))}
        </Grid>
    </Container>
);
```

Instead of copy pasting 9 components we can change the code to work with and array of components. We simply start with an array of our indexes (the `as const` is there so TypeScript interprets them as literals) and then map it into an array of `Square` components.

Note:
You will now see an error in the developer console saying that `Each child in a list should have a unique "key" prop`. This is safe to ignore for now and we will get back to it in a later lecture.

Material UI icons

Now to finalize our Square component we can use Cross and Circle icons instead of rendering plain text. Material UI provides a large set of consistent and clean icons. You can browse through them here and read the documentation here.

We will be using these two icons for our players:

```
import Cross from "@material-ui/icons/Clear";
import Circle from "@material-ui/icons/PanoramaFishEye";
```

To keep our components small and simple we can extract this logic into a new `PlayerIcon` component which we can use in our `Square` component.

```
// Imports...

const useStyles = makeStyles({
    cross: {
        color: "red",
    },
    circle: {
        color: "blue",
    },
});

const PlayerIcon: FC = ({ children }) => {
    const classes = useStyles();
    switch (children) {
        case "X":
            return <Cross className={classes.cross} />;
        case "O":
            return <Circle className={classes.circle} />;
    }
    return <Typography variant="h5">{children}</Typography>;
};
```

```
export default PlayerIcon;
```

And that’s all we will spoil from this week’s task. You can freely use any code from these materials in the assignment but your own creative solutions can and will be awarded.

0	1	○
3	×	○
×	7	×