

(1. React and JSX basic syntax)

- Finally output to browser DOM will be **HTML with DOM JavaScript**
 - e.g. `<input type="submit" onclick="myFunc();" class="style1" >Delete item 101</...>`
- Before that, **React components** (Possibly w. Mui-React components) to be rendered as HTML etc
 - e.g. `<Button onClick={()=>myFunc()} classNames= >Delete item 101</...>`
- Before that we might have **JavaScript run** so that dynamic values are evaluated, e.g. `{item.id}`.
 - e.g. `<Button onClick={()=>myFunc()} >Delete item {item.id}</...>`
- Before that we might **write** things in **TypeScript**, tsc compiled to JS

2. Single-page application, SPA

- Browser fetches/loads only one version of the page from the server, In the beginning.
- After that with JavaScript running on the browser, we update that one page and to the user it looks like we are navigating through different pages.
- React routing is able to make it look like we are navigating.

3. React component state and props.

- State created with the **useState** hook(s).
- Updated with the set (setter) function.
- Props given to the Child component. Child component receiving the props and destructuring them into local consts with the **destructuring assignment**.

4. Rendering single and multiple components.

- Mostly wrap your returned components inside parenthesis () to avoid mistakes
- Return just single component, e.g. a list. If multiple components, wrap them into a React fragment, `<> ... </>`
- With multiple components (map etc. list creation) remember to give the unique **key** property, e.g. pick the id of the element as value for the key

5. Material UI components

- use only Material UI components to get the styles defined in the Theme. (=> no HTML elements, p, span, if possible)
- think what HTML elements they return. e.g. so we won't get <p> inside of a <p> which is against HTML standard. Or <div> as direct child of <table> or so on.
 - For some components you can specify which HTML element rendered
- (Start learning Mui React components from very simple examples of yours. Build gradually bigger.)

6. Using the Material UI Theme.

- Define the Theme in a different file with the **createTheme** function. You can define Palettes, give components special custom styles.
- Wrap the whole application inside **ThemeProvider**. That already gives Material UI components their styles.
- If you **only** use Material UI components. Then you'll get the theme styles automatically.
- In rare need to use specific colors or styles from the theme, use **useTheme** hook to get the theme object to your component

Use redPalette and yellowPalette to spot hardcoded styles, colors

- Make your whole project follow the theme styles and ensure that by flipping all colors to some red tints, and then to yellow tints.
- If something did not go red (later yellow) you have hard-coded parts
- Similarly you can use some funny distinctive font from the theme and what is not funny, is hard-coded.

<CssBaseline /> Browser styles reset to Mui's style

- The idea is to reset all style differences between different browsers.
- Gives an explicit style setting for everything and thus
 - The browser differences would hopefully disappear
 - The Material UI styles would stay consistent also in the future

7. Child component updating parent's state

- (Normal case was: Parent passes **data** to the children in props, and children only show it)
- Special case: Parent will pass 'event handler function' (parent state's set function) to the child component(s) in props so that child/children can execute it and thus e.g. 'write' to parent's state.

8. App context

- Creating the context
 - `createContext({thing: 22, thang: "abc" ...})`
- Accessing the context values
 - `useContext()` hook to create local reference to the context
 - + `{myContext.thing}`
- Updating the values in the context
 - `useContext()` hook
 - + `myContext.thing = 33;`

9. The useEffect hook

- For “*side effects*” which means something happening outside the normal “state/prop changes => render happens” -cycle.
 - Usually still essential activity for the app, like fetching data from DB
- useEffect defines three parts:
 - **Action**, function to be run (cannot be awaited, but you can call another function that will wait)
 - **Dependency array**
 - if items given [user.id, project.count], then action runs at mount + if changes in these.
 - Empty [], dependency array => runs only once
 - If missing dependency array => runs at mount/first render + for any update/re-render.
 - (Action function is possibly returning) the **Cleanup function**

10. Routing

- Look at the given pictures for routing. **Basic routing** includes:
 - The **Route definitions**. They are our mappings between a **Path** and a View **React component** that renders that path.
 - Defining **Links** or **Buttons** that trigger navigation to the Routes
 - **Routes** (older component was called **Switch**) that marks the place (div) on the page that will be replaced/filled with the “current page”. It also lists many Route components as content/children, but only one Route’s View switched each time to be rendered there.

11. Nested Routing

- Look at the given pictures for routing. Nested Routing includes:
 - **Relative paths** for children
 - Possible **index route**, rendering the index path.
 - Possibly **shared layout** component.
 - Possibly with **shared context**?

+ Programmatic Routing

- You can also programmatically **manipulate the Routing history stack**: pop, push, replace.

12. Typical List component composition

- (See random example on the next slide. Idea important, not exact match)
- The example is about Buildings, e.g. three buildings on one campus, two on other.
- SRP = Single Responsibility Principle. Each component doing usually only one thing in the app:
 - Providing navigable view: **BuildingListView**
 - Holding and refreshing data list state, maps w. ListItems: **BuildingList**
 - Providing actions for one item: **BuildingListItem** (clickable, x = delete)
 - Just shows data for one item: **BuildingDisplay** / BuildingInfo / B...Details

BuildingListView

BuildingList (this could fetch and hold the data list in its state)

BuildingListItem (is passed one Building object in props, by the parent)

BuildingDisplay (is passed the Building object in props, by the parent)

BuildingListItem (is passed one Building object in props, by the parent)

BuildingDisplay (is passed the Building object in props, by the parent)

BuildingListItem (is passed one Building object in props, by the parent)

BuildingDisplay (is passed the Building object in props, by the parent)

(BuildingListItem could handle clicks and actions per each item.

And BuildingDisplay/Info/(Details) could be just presentational/display component)