

***Nikola Nikolic 2023***

**The content course**

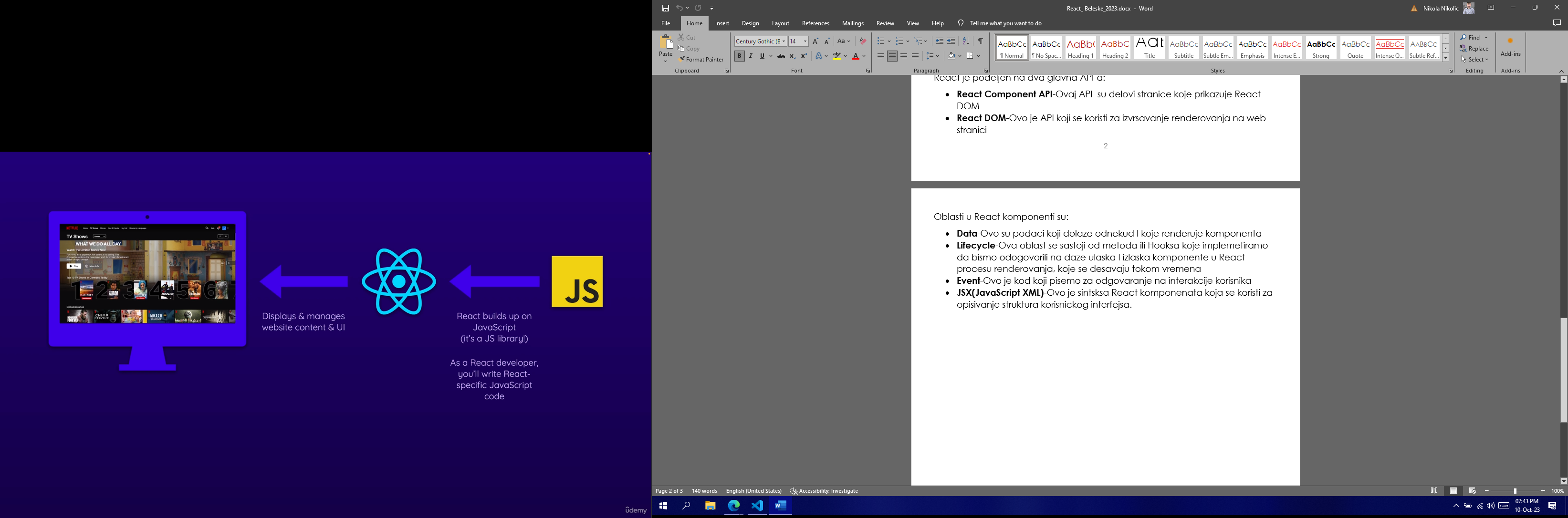
1. **Introduction to React**
2. **Start project**
3. **React Essentials-Components,JSX,Props,State**

* Building User Interfaces with Components
* Using, Sharing & Outputting Data
* HandlingUser Events
* Building Interactive Uls with State

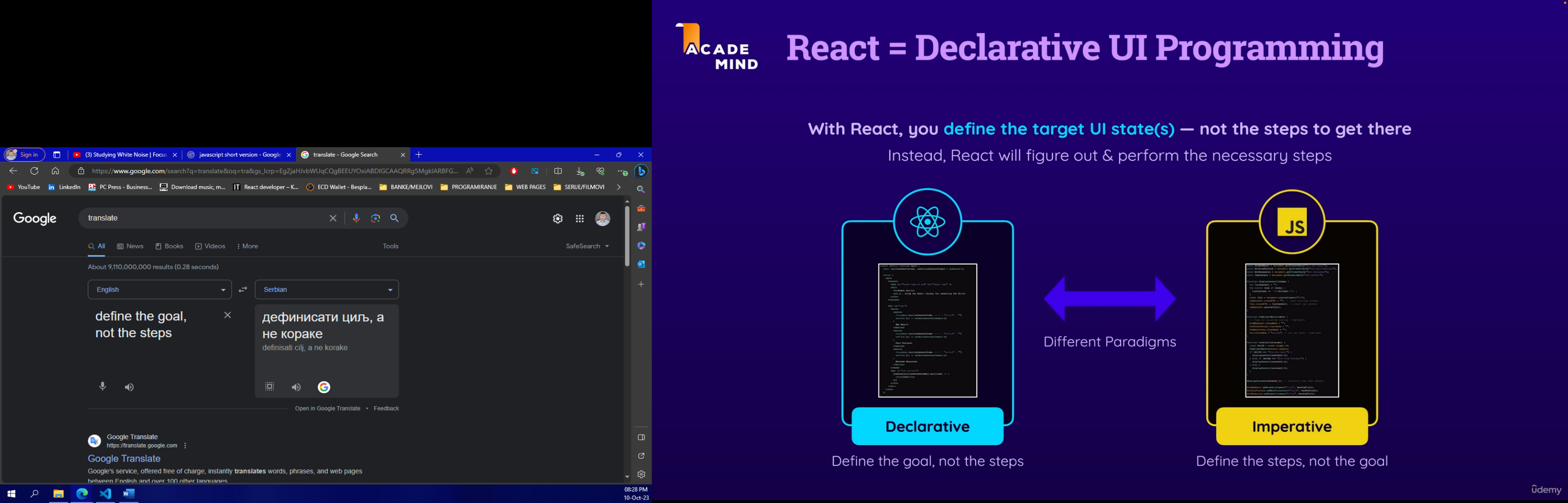
1. **React Essentials-Deep Dive**

* Behind the Scenes of JSX
* Structuring Components and State
* Advances State Usage
* Patterns & Best Practices

**Uvod u React**



React je Javascript biblioteka za kreiranje korisnickog interfejsa(UI-a). React se smatra slojem prikaza u aplikaciji.



**3.React Essentials-Components,JSX,Props,State..**

* **A Closer Look: Components & File Extensions**

At this point, you've built a first custom component and you, of course, also worked with the App component.

For the moment, both components are stored in the App.jsx file (this will change later though).

.jsx is a file extension that's not supported by the browser! It's working because you're working in a React project that supports this special extension. Because this extension *"tells"* the underlying build process (which is running behind the scenes when the development server is running) that a file contains JSX code (which is also not supported by browsers).

It's important to understand that it's really just that build process that cares about this extension.

And therefore, you'll also find React projects that **don't use** .jsx but instead just .js as a file extension. And in those .js files, you'll also find JSX code. Because it simply depends on the underlying build process which extension is expected when using this JSX syntax in a file.

Since it doesn't work in the browser either way, there is no hard rule regarding this. Instead, you'll find projects that require .jsx (like the project setup we use in this course) and you'll find projects that also support .js (with JSX code inside).

I'm emphasizing this here so that you're not confused if you encounter React projects that don't use .jsx files.

In addition, you'll also find projects that require the**file extension as part of file imports** (e.g., import App from './App.jsx') and you'll find other projects that don't require this (i.e., there, you could just use import App from './App').

This, again, has nothing to do with the browser or *"standard JavaScript"* - instead it simply depends on the requirements of the code build process that's part of the project setup you chose.

* **More Prop Syntaxes**

Beyond the various ways of setting and extracting props about which you learned in the previous lecture, there are **even more ways of dealing** with props. But no worries, you'll see all these different features & syntaxes in action throughout the course!

**Passing a Single Prop Object**

If you got data that's already organized as a JavaScript object, you can pass that object as a single prop value instead of splitting it across multiple props.

I.e., instead of

1. <CoreConcept
2. title={CORE\_CONCEPTS[0].title}
3. description={CORE\_CONCEPTS[0].description}
4. image={CORE\_CONCEPTS[0].image} />

or

1. <CoreConcept
2. {...CORE\_CONCEPTS[0]} />

you could also pass a single concept (or any name of your choice) prop to the CoreConcept component:

1. <CoreConcept
2. concept={CORE\_CONCEPTS[0]} />

In the CoreConcept component, you would then get that one single prop:

1. export default function CoreConcept({ concept }) {
2. // Use concept.title, concept.description etc.
3. // Or destructure the concept object: const { title, description, image } = concept;
4. }

It is entirely up to you which syntax & approach you prefer.

* **Grouping Received Props Into a Single Object**

You can also pass multiple props to a component and then, in the component function, group them into a single object via JavaScript's ["Rest Property"](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#rest_property) syntax.

I.e., if a component is used like this:

1. <CoreConcept
2. title={CORE\_CONCEPTS[0].title}
3. description={CORE\_CONCEPTS[0].description}
4. image={CORE\_CONCEPTS[0].image} />

You could group the received props into a single object like this:

1. export default function CoreConcept({ ...concept }) {
2. // ...concept groups multiple values into a single object
3. // Use concept.title, concept.description etc.
4. // Or destructure the concept object: const { title, description, image } = concept;
5. }

If that syntax is a bit confusing - worry not! You'll also see concrete examples for this syntax (and for why you might want to use it in certain situations) throughout the course!

**Default Prop Values**

Sometimes, you'll build components that may receive an optional prop. For example, a custom Button component may receive a type prop.

So the Button component should be usable either with a type being set:

1. <Button type="submit" caption="My Button" />

Or without it:

1. <Button caption="My Button" />

To make this component work, you might want to set a default value for the type prop - in case it's not passed.

This can easily be achieved since JavaScript supports default values when using object destructuring:

1. export default function Button({ caption, type = "submit" }) {
2. // caption has no default value, type has a default value of "submit"
3. }

* **Component Composition**

Your task is to create a reusable Card component that takes a name prop as an input and, in addition, can be wrapped around any JSX code.

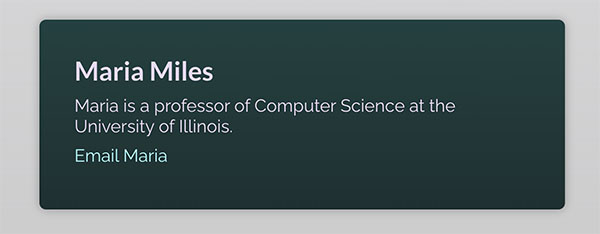
Use the already existing Card.js file to create the Card component in there. You can add the card CSS class to the main wrapping element in that component for some styling.

The name prop should be output as a title inside the Card component, the wrapped JSX code should be output below that title.

For example, the final Card component, should be usable like this:

1. <Card name="Maria Miles">
2. <p>
3. Maria is a professor of Computer Science at the University of Illinois.
4. </p>
5. <p>
6. <a href="mailto:blake@example.com">Email Maria</a>
7. </p>
8. </Card>

This should yield the following visual **output**:



*You can, but don't have to, tweak and edit the JSX code returned by the App component.*

* **Configuring Event Handlers**

Your task is to **edit** the <button> in the App component such that the already defined handleCreateUser function is called with a value for name.

So you must not hard-code the value that should be assigned to user.name in the handleCreateUser function but instead pass it as a value for the name parameter when a click event on the <button> occurs.

You **don't** have to care about any value that might be entered into the <input> field - it's just there for decoration purposes.

**Working with State**

You're working on a part of an online shop where a discounted price should be displayed on the screen once the user clicked a button.

Your task is to add an **event listener** to listen for **clicks** on the button that's already included in the App component.

Upon a button click, the **price should change** from $100 to $75.

**Add a state value** to the existing App component function and make sure the state value is both **updated** upon button clicks and **output** as part of the JSX code.

---

**Important:** In this Udemy environment, you **CAN'T** import & use useState like this:

1. import { useState } from 'react';
2. ...
3. useState();

Instead, import & use it like this (in your component):

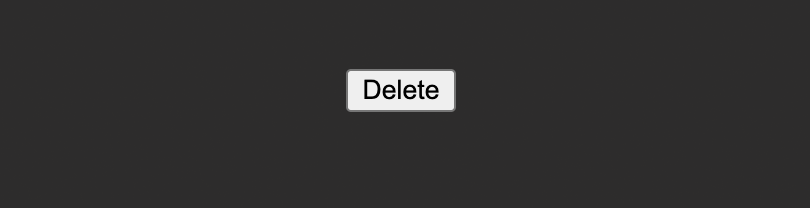
1. import React from 'react';
2. ...
3. React.useState();

* **Conditional Content**

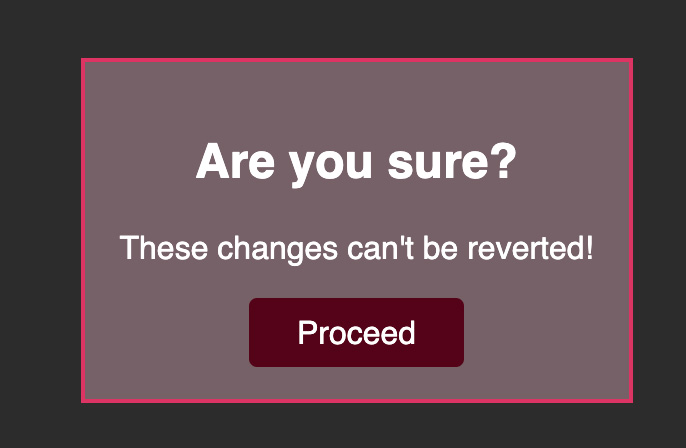
You're working on a part of a web app that's responsible for **showing a warning** when a user is about to perform a dangerous action.

Therefore, your task is to **conditionally** show a warning box once a user has **clicked** a specific button. Inside that warning dialog, another button allows users to **dismiss the warning** (i.e., remove the warning box from the screen).

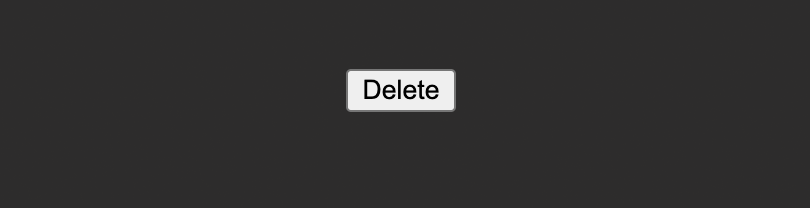
The finished app should display this UI, if the <button> has not been clicked yet:



And this UI, once the button was clicked:



Once the "Proceed" button was clicked, the warning box should be removed again:



For this task, you must react to clicks on both <button> elements that are part of the starting code. The second button, outside of the <div> with the id="alert", should show the <div id="alert"> (and all its content). The button inside that <div> should then hide it again (i.e., remove it from the DOM).

It's up to you whether you want to use a ternary expression or store the conditionally shown JSX code in a variable.

* **Dynamic Styling**

Your task is to **dynamically** apply a CSS class (active) to the <p>Style me</p> element in the provided React app.

The class should be applied when the <button> is clicked for the first time.

Here's how the finished app should look like BEFORE the button was clicked:



Here's how it should look like AFTER the button was clicked:

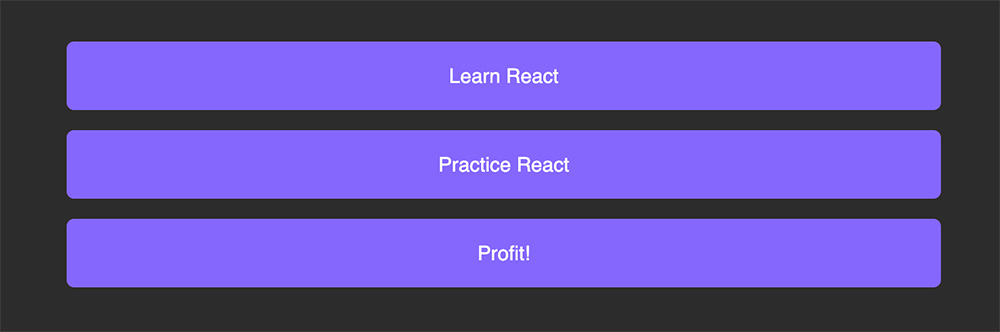


* **Dynamic List Content**

You're working on a "Todo List" web app and your task is to **output a list of dummy todo items** **dynamically**. For this task, a Todo component has been prepared for you, though you must still add some code to it to **receive and output** the todo text.

To be more precise: In the App component, you should transform the DUMMY\_TODOS array that's provided to you (which must not be changed!) to a list of JSX elements (<Todo> elements to be precise). Every Todo component item must receive and output the todo text via a prop called text.

The final UI should look like this:



**4. React Essentials-Deep Dive**

* **Using Fragments**

Your task is to edit the existing Summary component such that it outputs the following content:

1. <h1>Summary</h1>
2. <p>{text}</p>

Inside the Summary component, this content must not be wrapped by any other HTML element!

For example, this code would be wrong:

1. <div>
2. <h1>Summary</h1>
3. <p>{text}</p>
4. </div>

* **Forwarding Props**

Your task is to work on the Input component such that it either returns a <textarea> element or an <input> element, depending on whether a richText prop set on Input is true or false.

I.e., if used like this:

1. <Input richText />

the Input component should render a <textarea>. Otherwise, it should render an <input>.

In addition, the Input component should forward all other props directly to the returned <textarea> or <input> elements.

I.e., it should be usable like this:

1. <Input type="text" placeholder="Your name" />

*(as seen in the existing App.js file)*

The final UI should look like this:



* **Creating Flexible Components**

Your task is to build a **highly re-usable**, custom Button component that can be used in all the following ways *(also see the code in the App.js file)*:

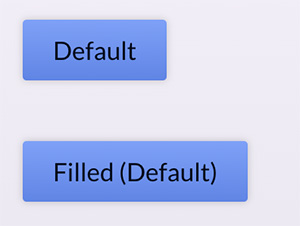
**"Filled" mode (default):**

1. <Button>Default</Button>

or

1. <Button mode="filled">Filled</Button>

should yield buttons that looks like this:



**"Outline" mode:**

1. <Button mode="outline">Outline</Button>

should yield a button that looks like this:



**"Text-only" mode:**

1. <Button mode="text">Text</Button>

should yield a button that looks like this:



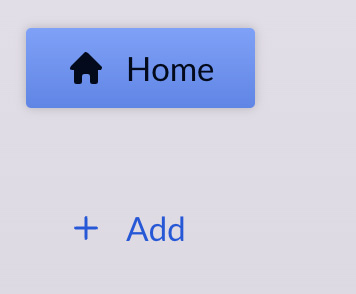
**With Icon:**

1. <Button Icon={HomeIcon}>Home</Button>

or

1. <Button Icon={PlusIcon} mode="text">
2. Add
3. </Button>

should yield buttons that look like this:



*Hint: To make sure the icon becomes visible (if passed correctly to the component & used in there), wrap the icon component in the button with a <span> that has the class "button-icon" on it.*

You find all the styles (CSS classes) that are required to build a button that supports these different "modes" in the provided index.css file!

All buttons need a button CSS class - and then, depending on their mode, additional classes.

In addition, the custom Button component **must accept** all **standard props** that could be set on the built-in <button>. These props should be **forwarded** to the default <button> element that will be used in the custom Button component.

Your task therefore is to work on the Button component provided in the Button.js file. **Don't** add multiple custom components, instead work on that **one** provided component and make sure that it supports all these different modes & features. Also make sure, that if no mode is set, the *"filled"* mode is assumed as a default.