W4 - PRACTICE

JSX - Dynamic Data - Components

At the end of this practice, you should be able to…

- ✓ Create a new **component** from HTML
- ✓ Translate HTML to **JSX**
- ✓ Understand the basic of **nested components**
- ✓ Draw a **diagram** component from some given code
- ✓ Understand how to display **data dynamically** using curly braces *{xx}* in JSX

How to work?

- ✓ Download **the start code** from the Google classroom
- ✓ For each exercise you can either:
  - Run `npm install`
  - Or move an existing `node_modules` to the exercise folder *(fastest option!)*

How to submit?

- ✓ **Create a repository on GitHub** with the name of this practice:
  Ex: `C2-S1-PRACTICE`
- ✓ **Push your final code** on this GitHub repository (if you are lost, follow this tutorial)
- ✓ Finally, submit on **Google classroom** your GitHub repository URL
  Ex: `https://github.com/thebest/ C2-S1-PRACTICE.git`

Are you lost?

You can read the following documentation to be ready for this practice:
https://www.w3schools.com/react/react_jsx.asp
https://www.w3schools.com/react/react_props.asp
https://www.gatsbyjs.com/docs/how-to/images-and-media/importing-assets-into-files/

Lab4.1

The Prop-Driven Card

Pass an object as a single prop.

```
const TaskItem = (props) => {
  return (
    <div style={{ border: '1px solid black', margin: '5px' }}>
      {/* Display Name and Priority here */}
    </div>
  );
};

function App() {
  const taskData = { id: 101, name: "Buy Milk", priority: "High" };
  return <TaskItem info={taskData} />;
}
```

TODO: Inside **TaskItem**, access the info prop to display the name and priority.

Lab4.2

Level Nesting (The Layout)

Create a hierarchy: App → TaskContainer → TaskItem

```
const TaskItem = () => <li>Individual Task</li>;

const TaskContainer = () => {
  return (
    <fieldset>
      <legend>My List</legend>
      {/* Render 3 TaskItems here */}
    </fieldset>
  );
};

function App() {
  return <TaskContainer />;
}
```

TODO: Modify the components so that `App` is the parent, and ensure `TaskContainer` correctly wraps the list of items

Lab4.3

The Multi-Input State

Sync multiple inputs to a single source of truth

```
function App() {
  const [title, setTitle] = useState("");
  const [desc, setDesc] = useState("");

  return (
    <form>
      <input placeholder="Title" />
      <input placeholder="Description" />
      <p>Preview: {title} - {desc}</p>
    </form>
  );
}
```

TODO: Connect both inputs so that as the user types, the preview paragraph updates instantly

Lab4.4

Functional State Updates

Use the spread operator to add to an array

```
function App() {
  const [list, setList] = useState(["Task 1"]);

  const handleAdd = () => {
    // Task: Add "New Task" to the list without losing old data
  };

  return <button onClick={handleAdd}>Add Item</button>;
}
```

TODO:

1. Implement `handleAdd` using `setList([...list, "New Task"])`
2. Add a text input field (`<input type="text" />`) to allow the user to enter a task name.
3. Display the list of tasks using the map() function, for example: `list.map((task) => <li>{task}</li>)`

Lab4.5

The Deletion Callback

Tell the parent to delete an item from the child

```
const TaskItem = ({ name, onRemove }) => (
  <li>{name} <button onClick={onRemove}>Delete</button></li>
);

function App() {
  const [tasks, setTasks] = useState(["React", "JSX", "Props"]);

  const removeTask = (index) => {
    // Logic to filter the array
  };

  return (
    <ul>
      {tasks.map((t, i) => <TaskItem key={i} name={t} onRemove={() => removeTask(i)} />)}
    </ul>
  );
}
```

TODO: Complete the `removeTask` function using `.filter()`.

Lab4.6

The "Toggle" Signal

Change a status in the parent via a child action.

```
const StatusBadge = ({ active, onToggle }) => (
  <button onClick={onToggle}>{active ? "Done" : "Pending"}</button>
);

function App() {
  const [isDone, setIsDone] = useState(false);
  return <StatusBadge active={isDone} onToggle={() => setIsDone(!isDone)} />;
}
```

TODO: Ensure that clicking the button in the child successfully changes the text from "Pending" to "Done"

Lab4.7

Prop Drilling (The Middleman)

Pass a prop through a component that doesn't use it.

```
function ProfileIcon() {
  return (
    <div>
      {/* TODO: Display the user name here */}
    </div>
  );
}

function Header() {
  return (
    <header>
      <ProfileIcon />
    </header>
  );
}

function App() {
  const user = "John Doe";

  return (
    <div>
      <Header />
    </div>
  );
}
```

TODO: Pass a `user` string from `App` → `Header` → `ProfileIcon`. The `Header` should not use the string, just pass it down

Lab4.8

Dynamic Style Callback

Use a child input to change the parent's CSS

```
const ColorPicker = ({ onColorChange }) => (
  <input type="color" onChange={(e) => onColorChange(e.target.value)} />
);

function App() {
  const [bg, setBg] = useState("#ffffff");
  return (
    <div style={{ backgroundColor: bg, height: '100vh' }}>
      <ColorPicker onColorChange={setBg} />
    </div>
  );
}
```

TODO: Verify that picking a color in the child input updates the background of the parent `div`.

Lab4.9

Handle the "Empty State" UI.

Use a child input to change the parent's CSS

```
function TaskList({ tasks }) {
  return (
    <ul>
      {tasks.map((task, index) => (
        <li key={index}>{task}</li>
      ))}
    </ul>
  );
}

function EmptyMessage() {
  return <p>No tasks available. Please add a task.</p>;
}

function App() {
  const tasks = [];

  return (
    <div>
      {/* TODO: Conditionally render EmptyMessage or TaskList */}
    </div>
  );
}
```

TODO:

1. If your `tasks` array is empty, render a component called `EmptyMessage`. If it has items, render the `TaskList`
2. If `tasks.length === 0`, render `<EmptyMessage />`
3. Otherwise, render `<TaskList tasks={tasks} />`

Lab4.10

The Master Todo Search

Filter a list based on an input state.

```
function App() {
  const [search, setSearch] = useState("");
  const items = ["Apple", "Banana", "Cherry"];

  const filteredItems = items.filter(item => item.includes(search));

  return (
    <div>
      <input onChange={(e) => setSearch(e.target.value)} />
      {/* Map filteredItems here */}
    </div>
  );
}
```

TODO: Successfully map the `filteredItems` array so that typing "a" hides "Cherry".

Lab4.11

The Tailwind Power-Up

1. **Install Tailwind:** https://tailwindcss.com/docs/installation/using-vite
2. Test the Design**:** In your `App.jsx`, use Tailwind classes to style a title.

```jsx
function App() {
  return <h1 className="text-4xl font-bold text-blue-600">Tailwind is Working!</h1>;
}
```

Lab4.12 (CODE STARTER - EXERCISE 1)

Your task is to create your first React **component**!

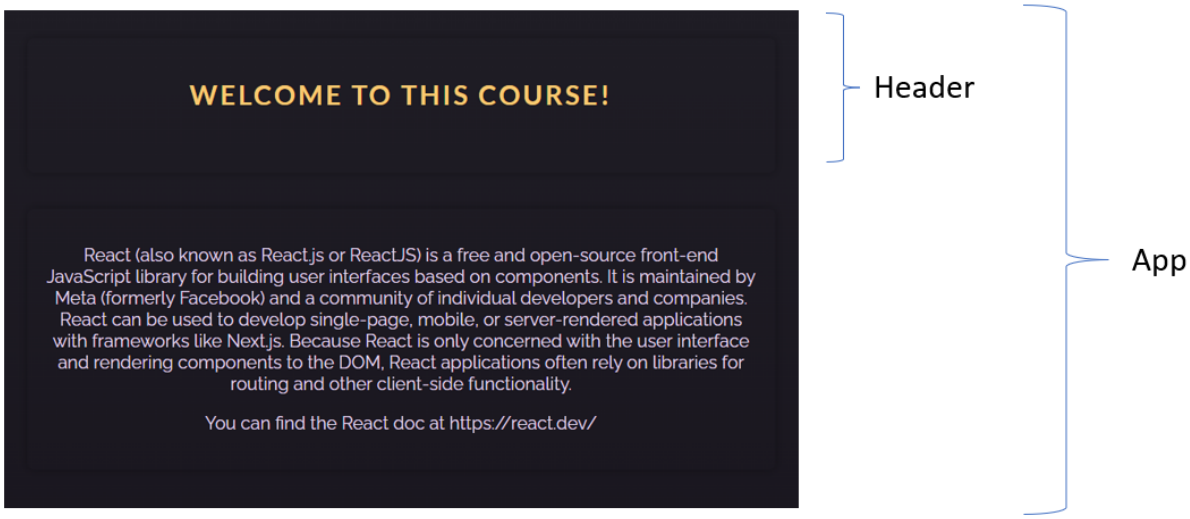
You have an App component, containing the header and the body.

- Create a component **Header** containing the header of the file.
- Change the code in the App component to use this new component

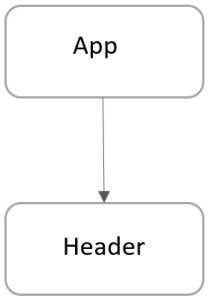
Notes:

- *You can create the component directly in the App file.*

The finished app could look like this:



The finished app diagram component:



Lab4.13 (CODE STARTER - EXERCISE 2)

Well done!
Now your challenge is to **convert some vanilla HTML** into some React JS code!

Q1 – Research on internet and list down the **main differences** between **HTML** and **JSX** syntax

-
-
-

Q2 – The first part is to create an **empty React project** which display Hello

- **Create a new React project** using the following command:

```
npm create vite exercise2 -- --template react
```

- On the root folder, **remove** the following useless file:

```
.eslintrc.cjs
README.md
.gitignore
```

- On `/src` folder remove, **remove** the following useless file:

```
/assets
App.css
```

- Edit the `index.css` and **remove all styles**
- Edit the `App.jsx` and just write a simple code:

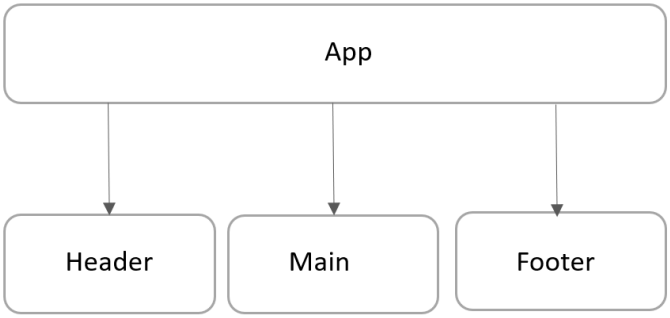
```
function App() {
  return (
    <>
      <p>Hello</p>
    </>
  );
}
export default App;
```

- From the root folder, launch `npm install` and `npm run dev`
- You have now a very simple ReactJS code that displays Hello:

Hello

Q3 – On this second part you need to **adapt** the original HTML code to your new created project:

Your code should be composed of 4 components, as bellow:



- Create a folder `/components`
- In this folder create 3 additional JSX files:
  - `Header.jsx`
  - `Main.jsx`
  - `Footer.jsx`

- Adapt the code from the original HTML code to those 4 compomers (App, header, Body and Footer)
  - Do not forget to **export** your components to use them outside!

- Finally, you can copy the original CSS code to your new project

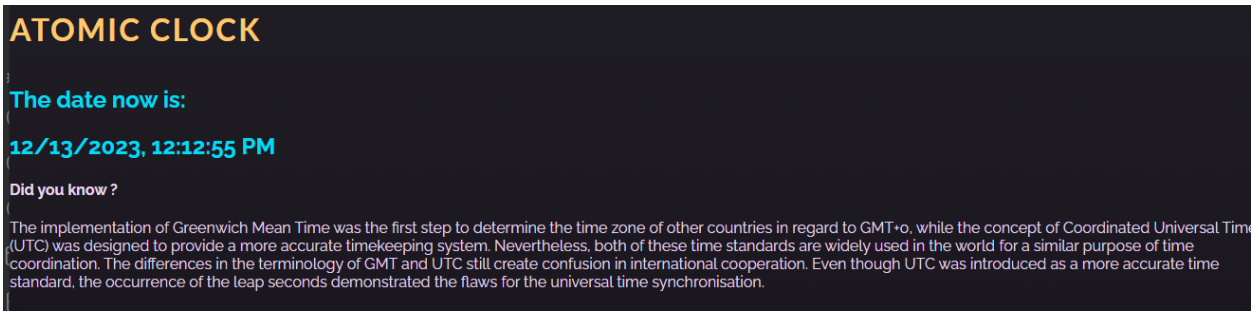
The finished app could look like this:

Lab4.14 (CODE STARTER - EXERCISE 3)

Amazing!

Q1 - Now your challenge is to **draw a diagram component** from some existing React JS code.

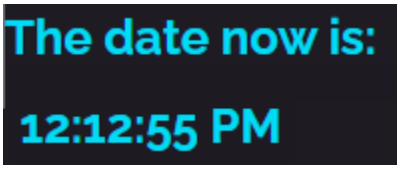
1. Read the code
2. Identify components
3. Draw the diagram component *(using power point or another tool)*



Q2 – Let's **play with dynamic data**:

- In **Header**, change the title to: "The amazing atomic clock"

- In **Time** component, change the code to display only the **time** only (not **date + time**)



Lab4.15 (CODE STARTER - EXERCISE 4)

Amazoooome!
For this last exercise, your challenge is to provide the dynamic data for the 2 following fields:

- The value (15 dollars) converted in Dong
- The value (15 dollars) converted in Euro

Important

- You need to implement and call the functions already provided for you to convert dollar to other devices
- All inputs are disabled: we use them for display only, not to enter any value…