Building Your First React App

Intro

In this workshop we introduce you to React. A powerful framework for building high-performance single-page (SPA) modern web applications. We will explore how to transform a static HTML page into a dynamic one composed of interactive and reusable components.

The workshop begins with an introduction to HTML and what it is used for. Then we will explore its limitations in capturing user interactions and present a client-side solution through code (we will give a demonstration). Taming code however is a known challenge in IT, and frameworks come to the rescue. In this regard we introduce React, which is today one of the most common frameworks to support developers with the more-and-more demanding modern web world.

Building upon the initial demo, we will show React capability to streamlining code, making it more compact, readable, and aligned with users' intentions. We will explore how React seamlessly integrates with HTML, empowering developers to isolate components and effortlessly design efficient algorithms.

We then introduce the concept of state and how manage it to supporting state/rendering transitions (especially after users' interaction). We will do so by exploring a more advanced scenario that will highlight the benefits of a framework in which DOM and state manipulation are transparent, simplifying the developer's role and allowing them to focus on implementing unique features.

As complexity and high-performance demand in modern webpages is increasing, efficient page rendering and interactions is paramount. In this regard, React introduced single-page applications (SPA) from the start. In React, SPA is built using a component-based architecture, where different UI components manage their state and interactions. React enables efficient rendering of changes in the user interface by updating only the necessary components, resulting in a fast and interactive user experience without full-page reloads.

Throughout the workshop, participants are invited to code along with the live demos. You will be supported with the installation of the necessary development tools. In addition, all demo code is available on Git for participants to download and study after the workshop.
Of course, the challenges in modern web development go beyond what we cover in this workshop. Challenges like security, interoperability, routing, server-side rendering, SEO, and more only highlight the vast and intricate word of web development. React, as you might have guessed, tackles most of these challenges and is constantly upgrading/improving while maintaining back compatibility (check out the documentation for more details). Our focus in this workshop is on introducing you to HTML/React, showing how interactions are captured and persistently tracked through state management.

Get ready and let's React!

Table of Contents

Content

We begin now with the detailed content of this workshop.

What is HTML?

HTML stands for Hyper-Text Markup Language, initially introduced in the early 90s it played a crucial role in the development of the World Wide Web (*www*). HTML was developed as a markup language to facilitate the sharing and retrieval of documents over the internet. HTML provided a standardized way to structure and format documents, allowing for the creation of hypertext documents with links that could be easily accessed and navigated.

- ***HyperText*** refers to text displayed on a computer or other electronic devices with references (hyperlinks) to other text that a reader (user) can immediately access.
- ***Markup*** indicates that text is surrounded by <markups> to enhance it and allow for formatting. The design of HTML was initially inspired by physical newspaper. In fact, markups such as <header>, <footer>, <body>, <h1>, <p>, and so on resemble the structure of newspapers.
- ***Language*** comes into play as the elements we just described form the foundation for any computer language. We have a domain syntax and semantics, which when combined they help create content (code) that is eventually interpreted by a web browser and displayed accordingly.

We now show how to create a simple HTML page and display it on the browser. On your computer create a file (hello-demo.html), copy the code below, and open it in your browser.

[DEMO 1 - SIMPLE STATIC HELLO-WORLD HTML PAGE]

```
<!DOCTYPE html>
<html>
  <body>
    <div>
      <h1>Hello World!</h1>
      <p>This example contains some basic HTML elements</p>
    </div>
  </body>
</html>
```

What Are HTML Pro and Cons?

If HTML is the current standard for rendering content on the web, what are its pro and cons?

- PRO
    - Very simple and small language. Easy to learn.
    - It is a standard and widely adopted across the web.
    - It is consistent in conveying information to users: same input is interpreted the same way across all HTML versions, same applies to design and layout.
    - It is continuously improved. HTML5, which is the current HTML version, introduced many features and improvements such as audio/video support with <audio> and <video> markups, <canvas> element for creating graphics

and animations, localStorage/sessionStorage API's to provide applications the possibility to persistently store users' data locally on the users' device, and so on.
    - Markups are processable by all modern search engines, making HTML pages easily interpreted by machines for indexing and SEO.
- CONS
    - HTML supports limited interactivity, since it was initially designed for supporting mainly static web content (the <script> markup was initially introduced for injecting code to verify forms and very simple animation).
    - Browser compatibility, as HTML is one, but browsers are many it is sometimes the case that the most advanced markups are not interpreted the same way by different browsers. *Therefore, thorough testing is crucial when developing pages intended to run across various browsers and devices.*

[DEMO – MANIPULATE PREVIOUS H1 TEXT IN THE BROWSER CONSOLE]

In summary, HTML is a foundational technology for conveying information on the web. Over the years, the topic of interactivity, one of HTML's "Achilles' heel", has been extensively discussed in the community. The solution evolved, leaning towards a more extensive support of the <script> to overcome this limitation. The boundaries of <script> evolved over the versions. Today inside the script tag we have the capability to fully access the various elements and properties within the context of a webpage. This includes the ability to interact/modify the Document Object Model (DOM) -*which is the real-time in-memory representation of the currently displayed HTML page*-, make API calls, access localStorage/sessionStorage, manipulate cookies, and more.

Achieving Interactivity in HTML

What are the requirements to use the <script> markup? We can say that the primary requirement, aside from syntactic considerations (such as the use of specific attributes) and semantic rules (like proper placement), is the language interpreted by the <script> markup, which is *JavaScript* (JS). In modern HTML, we can either link a file containing JS code or include JS inline within the file: between <script> and </script>.

JavaScript is a fully fledge computer languages (which we say is Turing-Complete). With JS we can create programs to solve complex problems and algorithms. In the context of HTML, JS is used to interpret and manipulate the DOM based on internal inputs (such as state changes or internal events) or external inputs (like user interactions or API responses).

How does a code in JS look like combined with HTML? In the following, we will show how to add interactivity to the previous code. Specifically, we will add a button that, when pressed, changes definitively the content of our paragraph.

Note the code below, the *innerHTML* property is used to change the content of the HTML element with the id "someText". Update the value of innerHTML at runtime (while the page is being displayed) to see the changes right away on your browser (we do this via code). This

indicates that browser, HTML, memory, and code are all parts of the same entity: manipulating one part affects the others.

[DEMO 2 – CREATE A PROGRAMMABLE BEHAVIOR IN HTML]

```
<!DOCTYPE html>
<html>
  <body>
    <div>
      <h1>Hello World!</h1>
      <p id="someText">Page with basic HTML elements</p>
      <div>
        <button onclick="changeText()">Change text</button>
      </div>
    </div>

    <script>
      function changeText(){
        var someText = document.getElementById("someText")
        someText.innerHTML = "Lorem ipsum dolor sit amet, consectetur adipiscing
elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua."
      }
    </script>
  </body>
</html>
```

Let's think now of a more complex scenarios, such as designing a portal with authentication for online payments, where one of the requirements is to ensure that all payments are processed securely. Another scenario could involve building a collaborative Jira board that allows multiple users to work together simultaneously. These are just a couple of examples, and many others exist.

As we dive into these scenarios, we might find that the JS code presented in the previous demo isn't ideal for handling more intricate situations. Although, JS provides abstractions that enable code encapsulation and support for complex interactions it does not come with native syntax or primitives to manipulate the HTML/DOM seamlessly. This is due to the nature of JS, which is being a general-purpose language. When it comes to DOM manipulation, JS alone becomes cumbersome and requires developers to 'manually' manipulate DOM elements and build the abstractions necessary to handle them.

However, due to its simplicity and browsers' native ability to interpret JS code at runtime, JS has found widespread application in the field of web development.

As applications evolved and certain patterns began to emerge, developers from various communities started developing libraries that organize JavaScript functionalities into controlled behaviors designed specifically for web development. These libraries, more sophisticated versions of which are called frameworks, aim at encapsulating the common patterns in web development, providing a more streamlined (sugar) approach. This allows

developers to focus on designing solutions to problems rather than dealing with the nitty-gritty of implementation and integration with HTML/DOM.

There are various JavaScript frameworks for web development, such as React, Angular, Vue, and so on. Each addressing the same problem: achieving high-performing interactive web applications with code that is readable, reusable, compact, and maintainable, but they do so with different philosophies in mind. While we won't delve into a detailed comparison in this workshop, our choice was guided by the framework with the highest momentum, market share, and developer support. We considered factors such as NPM downloads per week (reflecting the framework's usage), the quality of available documentation (indicative of the framework's maturity), eco-system (indicating many libraries are there out to support us), and Stack Overflow analytics (showing how actively developers are discussing the framework).

As a result of this analysis, React stands out with millions of downloads per week, widespread community support, vast ecosystem that greatly expands React's capabilities, sponsorship by Facebook (its main sponsor), and seamless compatibility with both JavaScript and TypeScript (developed by Microsoft and considered the new JavaScript with support for static typing, partially covered in this workshop).

Developing With React

In the following, we will discuss how to set up React. To begin, make sure to install Node.js on your machines. You can download Node.js from the official website.

Node.js is a powerful runtime enabling server-side JavaScript development. It also includes essential tools like the Node Package Manager (NPM), which is crucial for installing and managing JavaScript packages. Additionally, there's Node Package eXecute (NPX), a tool we'll use for the initial setup of our React application.

Step 1: Set Up Your Project

Create a new directory for your project and navigate to it in your terminal.

[DEMO]

```
mkdir react-typescript-simple-project
cd react-typescript-simple-project
```

Step 2: Initialize a New React App with TypeScript

Create the basic structure for our React app using *create-react-app* with TypeScript template (we choose TypeScript over JavaScript for this demo, to help users understand better the type of data being manipulated and their flow/transformation). The following NPX command will set up the necessary files and configurations for our first React application.

[DEMO]

```
npx create-react-app . --template typescript
```

Step 3: Remove Unnecessary Files

For this demo we will need to remove the unnecessary files and folders (outside to the scope of this workshop). This will help to make our code cleaner and readable. Open the *src* folder and delete all files except *index.tsx, App.tsx* and *index.css*. Remove the contents of *index.css* as well.

Step 4: Rendering Our Hello World in React

Go to App.tsx and replace the content with the following.

[DEMO]

```
const App: React.FC = () => {
  return (
    <div>
      <h1>Hello World!</h1>
      <p>Page with basic HTML elements</p>
    </div>
  );
};
export default App;
```

Step 5: Running The Project

To run the project, go the root folder (*react-typescript-simple-project*) in your terminal and run the following command.

[DEMO]

```
npm run start
```

The *start* command (as many others) is part of the available scripts present in this solution. You can see the full list of available commands in the *package.json* file under the *scripts* section. Note, in this file we mention also the packages (libraries) that are necessary for our solution to run, packages such as React are typically mentioned here. Packages mentioned in this file once installed are available and ready to be used in any part of our project. And bundled together with our custom code into one code that can be referenced/used in the HTML file. More information will follow in the next section.

Wait for the terminal to show the following:

```
Compiled successfully!

You can now view react-typescript-simple-project in the browser.

  Local:            http://localhost:3000
  On Your Network:  http://192.168.178.161:3000

Note that the development build is not optimized.
To create a production build, use npm run build.

webpack compiled successfully
Files successfully emitted, waiting for typecheck results...
Issues checking in progress...
No issues found.
```

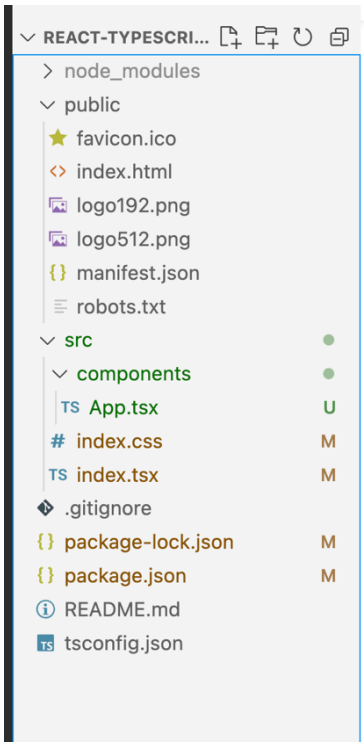
This indicates that your project is compiled and ready to run in your browser. Now, head to http://localhost:3000. You'll see a page containing our *Hello World* message. Unlike the previous HTML demo, where we ran the code directly by executing the file, we now need to navigate to localhost. It's worth noting that Node.js, which we installed in the previous step, is now acting as an actual server, serving our page along with the related React code at the address localhost:3000.

Try opening the inspector now. This allows you to inspect the rendered HTML and understand how React code is naturally transformed into vanilla HTML; of course, there is also JavaScript code involved (see the <script> tag), but this is transparent and served behind the scenes. Running the start command triggered something (a compiler) that took our code and transformed it into HTML, eventually serving it to our Node.js server for download. In the following section, we will explore the project structure in more detail and explain how React code is compiled and interpreted at runtime by our browser.

About React (informative section)

React is a powerful, component-based JavaScript framework that, through a sophisticated state-management and rendering system, enables the creation of high-performance single-page applications (SAP).

In the following, we'll explore key properties of React and discuss their relevance within the context of the project we've just created. Let’s also discuss how React code seamlessly integrates (we say compiles) with HTML. If we followed the previous steps, our code structure should look like the following.



The entry point of our project/code is *index.tsx*, referencing our code stored in *App.tsx*. In the public folder, our entry HTML file called *index.html* can be found. If we were inspect this file we would not find any mention of a *<script>* markup (necessary for code injection). Instead, in the comments, there's a reference to the term "template." This leads us to the following questions: Where is our code mentioned/injected? Moreover, if inside the <script> markups we are only allowed to inject JS code, shouldn't our *App.tsx* be named *App.js*? Also, if *.ts* is the extension for TypeScript files, what does *.tsx* stand for?

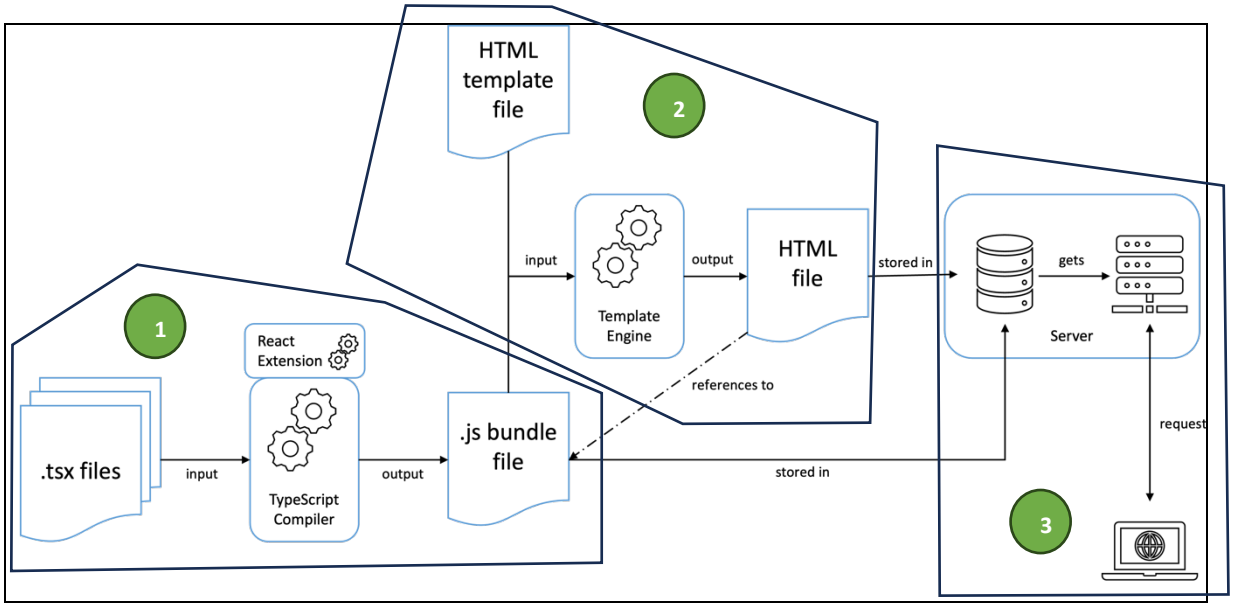
It becomes apparent that a lot is happening behind the scenes. This complexity is part of the tasks a framework (as opposed to a library) handles: facilitating its adoption and abstracting integration layers.

In the diagram below, we aim to provide a simplified overview of how the various parts of a React project come together to deliver a webpage in HTML and JavaScript.

1. React chose the .tsx extension due to some React syntax incompatibility with TypeScript. For example, in React, <...> denotes markup, while in TypeScript, it signifies a generic type argument. React extends the TypeScript compiler to convert *.tsx* files to standard *.ts*. This extension processes *.tsx* files, handling React elements and TypeScript type annotations. Once the React compiler/extension processes the *.tsx* files, the resulting code is then passed to the TypeScript compiler (TSC). TSC checks for type errors and *transpiles* TypeScript code into regular JavaScript code. The final JavaScript code, along with other assets, is bundled and prepared for execution in the browser. This process may involve tools like Webpack, Babel, or other bundlers.

2. At this stage, a new bundled JavaScript file is generated, and all related resources are merged into the HTML template file. This merging is handled by the template engine running under the hood, installed when calling the command `npx create-react-app`. The engine updates information inside the HTML template file to reference the .js file, creating a new HTML file ready for consumption. *create-react-app* is just one of many tools available for quickly setting up (scaffolding) React projects, other alternatives like Next.js, Vue CLI, and Webpack offer similar results, but with different project setups/structure and features.

3. Both the finalized HTML file and the bundled *.js* file are stored and made available to the server. When a user requests our page via the server URL, the server fetches the HTML file and its related resources, including the JavaScript file containing code to implement all dynamic behavior present on the page. These are then served back to the user/client.



Now that we understand how React files are combined and that React supports multiple files, which eventually converge into a single JavaScript bundle, it opens the door to a valuable practice: splitting solution code into separate files, each containing specific behaviours or functionalities. In software engineering, breaking down code into smaller, well-organized chunks is a widely adopted practice. This approach aims to enhance maintainability, reduce error potential, improve readability, and promote reusability.

In the context of React, a common practice is to divide code into files, and within each file, expose what are known as "components." Components in React play a crucial role in structuring and organizing the user interface. In the following section, we will dive into React components by extending our *Hello World* example. This extended Hello World aims to showcase how React expressive capabilities align with fundamental software engineering principles, offering a structured approach to navigate the complexities of building interactive web applications.

Diving Into React Components

The good news is that, without explicitly realizing it, we've already encountered a React component in this workshop. Let's review our previous example, you might have observed that our Hello World code was enclosed within other code.

```
const App: React.FC = () => {
  return (
    …
  )
};
export default App;
```

This code is what we call a *component* and in this specific example we call it *function component*. Notice the *:* next to *App*, this indicates the keyword "is". Instead *React.FC* stands

for "function component". So, if we were to read the first line, it would be: "App is a React Function Component".

So, why do we call it a "function component" and not simply a "component"? The distinction lies in the fact that *App* is not just a straightforward abstraction; it's an abstraction we ideally want to use multiple times. In programming languages, when we look to reuse abstractions, we turn to functions (or other forms of abstractions such as Classes that we do no cover in this workshop).

Think of functions as programmable copy machines in an office, each with a unique label. We hand over a paper to copy (input), and after taking its time, the machine eventually prints (returns) a copy. This process can be repeated throughout the day, with different papers (input) given for copying, resulting in various outputs.

Note, I mentioned that these machines are programmable. Indeed, we can enhance the copying process by installing new updates (code), or if it makes sense, we can completely change their purpose by providing an entirely new behavior (code). In our example above, the abstraction or function we aim to capture is the rendering of a "Hello World".

As previously mentioned, functions may take inputs, which we refer to as parameters (initial inputs provided by the outside world). When used in combination with code we obtain a unique effect that we can either ignore or capture by using the *return* keyword.

In our *App* example, the input provided is nothing. In JavaScript, "nothing" is denoted as *()*. Thus, `const App: React.FC = () => { … }` reads as "App takes no parameters."

If we were to pass inputs, we would represent them inside the parentheses. Input passed to components in React are called properties (*props*). We will explore properties more in detail later. In the example below, *App* takes two inputs, *x* and *y* (both strings). Whether or not we decide to use these inputs depends on the specific implementation.

```
const App: React.FC = ({ x:string, y:string}) => {
  …
};
```

In the next example, *App* takes *x* and *y* and returns their concatenation. *App* (our function) is unaware of the concrete values of *x* and *y*; it only assumes they are strings and prepares the effect for when these parameters turn into specific values. The power of abstraction lies in leaving the determination of values to the caller. A value could be the balance in your bank account, the number you pressed on the screen, your username, and so on.

```
const App: React.FC = ({ x:string, y:string {) => {
  return <div>{x.concat(y)}</div>
};
```

In the following example. We call *App* with the current session's username and email, assuming that a session exists and contains these values. This illustrates how components can be utilized with specific values provided by the caller.

```
…
//normally we would call App like App(session.Username, session.Email);
//but in React we call it like the following to match the HTML syntax
<App x={ session.Username } y={ session.Email }/>
…
```

Let now take the previous Hello World React example and let's add a button that when pressed it changes the value of the paragraph, like in the initial example with plain HTML and JavaScript. Let's update the content of our *App.tsx* with the code below.

[DEMO 3– EXTENDING REACT HELLO WORLD]

```
const App: React.FC = () => {
  return (
    <div>
      <h1>Hello World!</h1>
      <p>This example contains some basic HTML elements</p>
      <div>
        <button>Change text</button>
      </div>
    </div>
  );
};
export default App;
```

If we run the code above, we would end up with a page containing the text Hello World, a paragraph, and a button that if clicked it does nothing. Now we can implement like in the HTML/JavaScript example the same code to change the text, but that would defeat the idea of using React to handle interactions for us. Let's understand how components lifecycle work and how it comes to help to achieve the interaction necessary for the example above.

Components are like mosaics of a big picture, what React does is taking all components that define a project, combine them according to their dependencies and eventually generate the corresponding HTML to render on the page.

How do changes in our code become visible on a page? React facilitates this process through functions such as setState or dispatch. When invoked within a component, these functions tell React to manage the propagation of changes and compute the updated HTML to reflect those changes. When calling setState or dispatch, we need to provide values, the entire new state or portions of it. In React, we should look at components as a triple made of code, data, and functions to modify that data. Components are autonomous entities, capable of self-rendering (check the HTML after the return keyword), holding a state (*useState*), and updating that state, either directly or by delegation through a mechanism known as props.

Let's not update one more time our *App.tsx* with the updated version below the implements the expected behavior.

[DEMO 4 – ADDING BEHAVIOR TO BUTTON]

```
import { useState } from "react";

const App: React.FC = () => {
  const [text, setText] = useState<string>('Page with basic HTML elements');

  const changeText = () => {
    setText ("Some other text.");
  };
```

```
  return (
    <div>
      <h1>Hello World!</h1>
      <p>{text}</p>
      <div>
        <button onClick={ changeText }>Change text</button>
      </div>
    </div>
  );
};
export default App;
```

In this new version, when React calls our App component, it initializes a state using the useState function. Pay attention to the text passed as input; if it's the first time the App function component is called, the state will hold this initial text.

The useState function returns a pair: text and setText (we may of course choose different names). Text simply holds the current text (a stateful value), while setText is a function to update it. When this function is called (as seen in the onClick on the button), it takes the new text and replaces it within the state.

Calling setText triggers React to update the HTML corresponding to the current component (App) and rearrange the HTML of the page to reflect the changes. This is because the code of App depends on the value of text, which is the state of our component.

This behavior is well-documented and follows its lifecycle, which is explained very well in the React documentation.

Note also the use of curly braces *{}* in the HTML code inside the return body. Curly braces serve as an escape hatch into JavaScript expressions within the markup. When rendering TSX/JSX in a React component, this syntax is particularly useful for embedding/rendering dynamic values, such as those coming from the component's state or other variables directly into HTML.

In this exercise, the choice of naming the state variables as *text* and *setText* is contextual to the exercise. It's important to understand that the *useState* function in React is flexible and can handle an arbitrary number of state values. In this case, we are using a simple example with two state values (*text* and *setText*) to showcase the basic functionality of updating and managing state in a React component. However, in more complex scenarios, *useState* can be used with multiple state variables to suit the specific requirements of the application.

In the following example, the *useState* function is used twice, creating two independent state variables: *count* and *text*. The component displays the current count and our text, and two buttons allow you to increment the count and change the text, respectively. Each state variable is updated independently, demonstrating the flexibility of useState for managing multiple pieces of state within a component.

[DEMO 5 – ADDING EXTRA BEHAVIOR]

```
import { useState } from "react";

const App: React.FC = () => {
  const [text, setText] = useState<string>('Page with basic HTML elements');
  const [count, setCount] = useState<number>(0);

  const changeText = () => {
    setText ("Some other text.");
  };
  const incrementCount = () => {
    setCount (count + 1);
  };
  return (
    <div>
      <h1>Hello World!</h1>
      <p>{text}</p>
      <div>Count: {count}</div>
      <div>
        <button onClick={ changeText }>Change text</button>
        <button onClick={ incrementCount }>Increment count</button>
      </div>
    </div>
  );
};
export default App;
```

Nesting Components

Looking at the previous code it becomes obvious that as code starts growing the necessity to group code to make it more manageable and avoid replication becomes important. We already discussed that React supports multiple files and that components may be combined to extend their behavior. In the following we will see how to split the code above into two components. A parent one (in *App.tsx*) displaying the text and the counter, and another one nested to App containing the controllers: our 2 buttons to edit the text and increment the counter. For this part we need to create a new file under *components* and name it *Controllers.tsx*.

We begin with updating our App.tsx.

[DEMO 6 – NESTING COMPONENTS / GROUPING BEHAVIORS]

```
import { useState } from "react";
import Controllers from "./Controllers";

const App: React.FC = () => {

  const [text, setText] = useState<string>('Page with basic HTML elements');
  const [count, setCount] = useState<number>(0);
```

```
  return (
    <div>
      <h1>Hello World!</h1>
      <p>{text}</p>
      <div>Count: {count}</div>
      <Controllers count={count} setCount={setCount} setText={setText} />
    </div>
  );
};

export default App;
```

Note how the code became more compact and that a new element appeared `<Controllers … />`. As we might imagine the code that implements our two buttons is now inside Controllers. Note also the inputs we give to Controllers: `count={count} setCount={setCount} setText={setText}`, we call these in React properties. Properties are a mechanism in React that allow data to propagate over nested components (see count in the example above), including callbacks for updating the various states. In this example we decided to store the state of the text in *App* rather than in *Controllers*, so for controllers to update the text or counter it needs a reference to functions that takes care of this behavior: setCount and setText.

Below the code of the Controllers component.

[DEMO 6 – NESTING COMPONENTS / GROUPING BEHAVIORS]

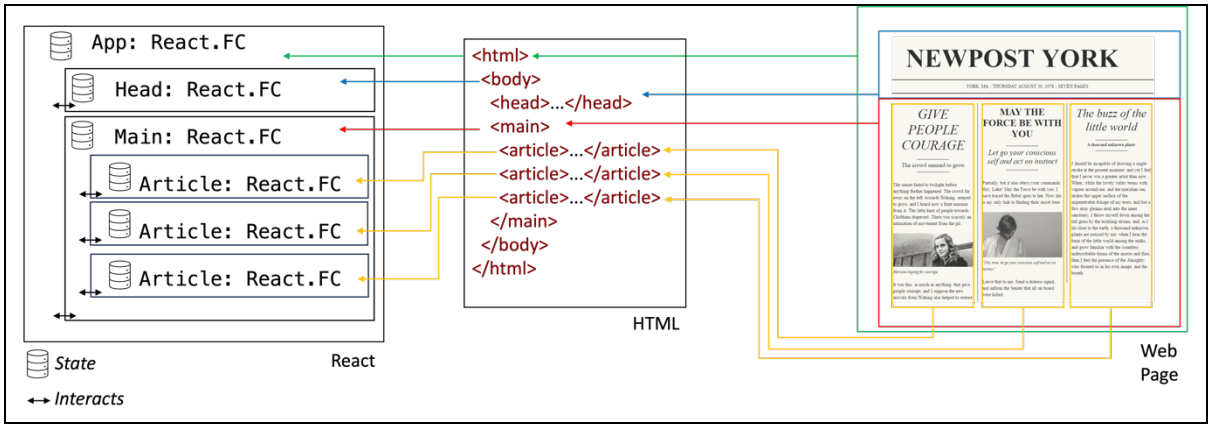
```
interface ControllersProps {
    setText: (newText: string) => void;
    count: number;
    setCount: (newCount: number) => void;
  }

const Controllers: React.FC<ControllersProps> = ({setText, count, setCount}) => {

  const changeText = () => {
    setText ("Some other text.");
  };
  const incrementCount = () => {
    setCount (count + 1);
  };
  return (
    <div>
        <button onClick={ changeText }>Change text</button>
        <button onClick={ incrementCount }>Increment count</button>
    </div>
  );
};
export default Controllers;
```

The *Controllers* component (more specifically *container component* since it holds no state) receives three props (*setText*, *count*, and *setCount*) defined by the *ControllersProps* interface, which acts as a contract specifying their types. The *setText* prop is a function that takes a new text string and updates the text, while *count* is a number representing a count value. The *setCount* prop is a function that takes a new count value and updates the count. Inside the component, there are two buttons: one triggers a function to change the text using *setText*, and the other increments the count using *setCount*. In this concrete example the *Controllers* component serves as a compact user interface for interacting with and modifying text and count values in our React application.

Conclusions

React code is bound to its HTML representation, where each component instance corresponds to an HTML element. React intelligently detects changes in components, organizes them, and seamlessly applies updates to the DOM. It's worth noting that the use of 'id' to reference components or elements in HTML is no longer necessary. The abstraction layer introduced by React allows developers to focus on software development rather than dealing directly with HTML and its limits.

HTML, being a presentation layer, is not well-suited for exhibiting or capturing interactive behavior or executing algorithms but React does. The diagram below summarizes this structure: starting from a web page (what the user sees), each element is mapped to an HTML markup. Despite the static appearance of HTML markup, it is powered by the underlying React engine. Every HTML element (or group of elements) is associated with a component, each maintaining its own state and capable of interacting with one another and responding to user input. This mechanism operates bidirectionally; HTML can present to and capture events from the user, propagating them to React, and React can autonomously manage itself, updating the HTML when necessary, so the appearance on the browser.



The objective of this workshop was to provide a gentle introduction to React development, covering fundamental concepts such as HTML, Components, State, and Props. React, as we've learned, enables developers to build interactive applications, shifting the focus from making a static page with some interactive elements. While React offers plenty of functionalities, our workshop touched into just the basics, offering (I hope) a foundational understanding.

Although we could have explored more in-depth topics like TypeScript or JavaScript (which in itself constitutes an entire course in Computer Science), it wasn't the primary goal of this workshop. The foundational knowledge gained here should empower you to understand what React is and what fundamental problems it addresses.

For those eager to dig deeper into React, I recommend exploring topics like asynchronous behaviour (Promises), React Performance (understanding more components life cycle), Context API, React Routing, Higher-Order Components (HOC), React Native, Server-Side Rendering (SSR), and Progressive Web Apps (PWAs). These areas represent some of the most

interesting aspects of React, each offering solutions to specific challenges in the world of web development.

In conclusion, this workshop serves as a *starting point* into React development, offering a glimpse into essential web development concepts; as well as preparing the groundwork for further studying into advanced topics for those eager to deepen their understanding of both HTML and React.

Necessary software:

- Node.js https://nodejs.org/en
- Visual Studio Code https://code.visualstudio.com/Download

Extra reading

- Intro to HTML https://www.w3schools.com/html/html_intro.asp
- React official site https://react.dev
- TypeScript documentation https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html

Github code

- https://github.com/mabbadi/building-your-first-react-app