React

React

React is a library to build user interfaces. The main features of React are:

- **Declarative**: the UI is always defined as a function of its data, and will always update accordingly;
- **Component Based**: each element presented to the user is a *component*, from the smallest building block to the biggest UI element;
- **Portable**: it can power web apps, mobile apps or be rendered server side.

React

When building a UI using an **imperative** approach, we need specify how each update of the UI happens, and when:

```
let counter = 0
const button = document.querySelector('#button')
const counterElement = document.querySelector('#counter')

button.addEventListener('click', function () {
  counter += 1
  counterElement.innerText = counter
})
```

React

When building a UI using a **declarative** approach, we define the UI as a function of its data and expect the UI to update whenever the data does:

```
<div>{counter}</div>
<button onClick="{handleCounterIncrement}">Increment Counter<button></button></button>
```

```
function handleCounterIncrement() {
  setCounter(counter + 1)
}
```

React

A **library** is a *tool* that a developer can use to build an application, but does not necessarily care about all aspects of the application.

A **framework** is an entire toolbox that gives a developer all the tools that are required to build a specific type of application.

React

React, as a library, cares about the *UI* aspect of an application, and how the UI evolves following an interaction from the *User* or a change in the data.

React does **not** have an opinion on how the other parts of the application are implemented, so the developer must decide how to implement routing, data fetching, data persistence, etc.

Components

The main building block and the essential part of React are **components**. A component can be any part of the UI and it needs to be able to do a certain number of things:

- It needs to know how to "draw" itself;
- It needs to know how to react to inputs, if required;
- It needs to be able to receive information from the outside;

JSX

A component needs to know both how to **behave** and how to **appear**. To make the writing of components easier and allow the developer to define both these aspects within the same source file, React uses a superset of JavaScript called *JSX*.

JSX allows us to write regular JS code, but also to use tag-like syntax to define UI elements within the code itself.

```
const button = <button>Click me!</button>
```

JSX

Since JSX is not part of the JS specification, it's not natively supported by any browser or other JS runtimes. When building our app, all JSX code will be *transpiled* to regular JS:

```
const button = <button>Click me!</button>
```

```
const button = _jsx('button', { children: 'Click me!' })
```

The `_jsx` function is automatically imported by the transpiler.

JSX

Since all JSX code gets translated to regular JS, JSX is powerful tool that allows us to **see** the structure of our UI as if it was defined within a regular template, but without losing the power of JavaScript.

JSX

Any JS expression can be easily embedded within a JSX tag:

```
const name = 'Jimmy'
const header = <h1>Hello, {name}!</h1>
```

```
const a = 10
const b = 32
const result = <span>The result is {a + b}</span>
```

```
const sum = (a, b) => a + b
const result = <span>The result is {sum(11, 22)}</span>
```

A new React application

While a new React application can be written from scratch and configured with a vast variety of different tools, the easiest way to create a new application is to use `vite`. In a terminal, type:

```
$ npm create vite@latest my-app
```

And follow the instructions on screen to select React and JavaScript.

A new application will be created inside the `my-app` directory, relative to the path where you executed the command.

A new React Application

After the proccess of creating a new application is completed, you can open the `my-app` directory in you editor to find a base scaffolding.

First, run `npm install` within the `my-app` directory to install all the dependencies of the application. You can try running it by typing `npm run dev` in the terminal.

When done, delete all files from the `src` directory and create a new empty `index.js` file inside `src` before continuing.

Components

React components can be written in two different ways: using classes (**class components**) or using functions (**function components**).

While writing components as *function components* is the most modern approach, *class components* are very common, especially in older applications.

In this course we will focus on *function components*, but a module covering *class components* will be made available to you.

Hello, World

This is the simplest component can be written as a *function*:

```
function HelloWorld() {
  return <h1>Hello, World!</h1>
}
```

This looks like a simple function, but it's actually a React component. It returns the JSX expression that will be used to render the component in the UI.

Hello, World

The same component can also be written as *class*:

```
class HelloWorld extends React.Component {
  render() {
    return <h1>Hello, World!</h1>
  }
}
```

The class extends a base `React.Component` class and implements the `render` function, from which a React element is returned.

Components

When writing a component we are defining the *blueprint* of a UI element:

- The *name* of the class or function is how we use the UI element in our app. It **must always** begin with an uppercase letter;
- The React element (the returned JSX expression) is how the component *appears* when used in the UI;

Components

The JSX expression returned from component's function can only have **one** root element. When more than one element needs to be returned, it must be wrapped in a containing element:

```
export function HelloWorld() {
  return (
    <div>
      <h1>Hello, World!</h1>
      <p>What a beautiful day!</p>
    </div>
  )
}
```

Rendering a Component

Components are included in a JSX expression using a tag-like syntax, as they represent a part of the UI just as the base HTML tags:

```
const helloWorldElement = <HelloWorld />
```

Just like with base DOM components, the JSX syntax supports *self-closing tags* for user-defined components.

Rendering a Component

Since they are part of the UI, just like base DOM components, user-defined components can be mixed with default DOM components as required:

```
const helloWorldElement = (
  <div>
    <HelloWorld />
  </div>
)
```

Note the use of the *round brackets* to enclose multiple tags into separate lines to improve readability.

Rendering a Component

Once we have defined an element using a JSX expression, we can render it inside our page using the `createRoot().render` method:

```
import { createRoot } from 'react-dom/client'

const root = createRoot(document.getElementById('root'))
root.render(helloWorldElement)
```

The `createRoot` function will create a *root node* that will be used to render the element. The `render` method will then render the element inside the root node.

Entrypoint

The `index.js` file is the entrypoint of our application. It's the file that will be loaded by the browser and will run the code that will start and render our application.

Within this file, we will write the code that will create the root node and render the main element of our application:

```
import { createRoot } from 'react-dom/client'
import { HelloWorld } from './HelloWorld'

const root = createRoot(document.getElementById('root'))
root.render(<HelloWorld />)
```

Component Composition

One of the things a component needs to be able to do is to render itself by returning a JSX expression either from the `render` method (for class components) or from the function defining the component itself (for function components).

The JSX expression returned by a component can and will often contain other components, allowing for the composition of multiple components inside a *component tree*.

Component Composition

Most React applications will usually have a single *Root component* (usually called `App` or `Root`, but you can choose the name you like best) that will render other components; these child components will be able, in turn, to render more components.

The *component tree* can get as deep as required, as components can represent everything from a simple button to an entire screen and everything inbetween.

Component Composition

```
import { createRoot } from 'react-dom/client'

function App() {
  return (
    <div>
      <h1>My Awesome Application</h1>
      <HelloWorld />
    </div>
  )
}

const root = createRoot(document.getElementById('root'))
root.render(<App />)
```

Component Composition

Since React allows for components to be composed and an app will usually have a single *Root component*, `createRoot().render` will only ever be called once, at the very start of the application, to render our *Root component*.

Avoid calling `createRoot().render` more than once!

Props

All components need to be able to receive information from the outside.

This information is conveyed through the *props* object, which all components have access to. *Function components* can access their props using the `props` object received as the only parameter of the function itself:

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>
}
```

Props

The *props* object is the **only** parameter received by the function defining the component, and through this object we have access to all the information the component has received from its parent.

We can *destructure* this object to extract the information we need:

```
function Welcome({ name }) {
  return <h1>Hello, {name}</h1>
}
```

Props

A prop can be passed to a component when including it within a JSX expression in a manner similar to HTML attributes:

```
const greeting = <Welcome name="Jimmy" />
```

Props can be of any type. Whenever a prop is not a *string*, it can be passed by embedding an expression with *curly brackets*:

```
const greeting = <Welcome name="Kate" age={21} />
```

Props

There are few strict rules in React, but one of such rules is that **all React components must act like pure functions with respect to their props**.

This means that the `props` object is read-only, and its contents cannot be modified by a component.

Props

When defining a component, a developer needs to think about which data the component will need to receive in order to properly display itself and, if needed, respond to the interactions with the user.

A component, being a *reusable* piece of UI, should expect to receive as many props as needed to display itself properly.

Props

Props are passed down from a *parent component* to its *children components*, and in most applications they will *change over time*.

Every time one or more of the props passed to a component change, the component will automatically re-render itself and all its children components.

Conditional Rendering

Conditional rendering with a JSX expression can be easily achieved by embedding another expression that uses JS's conditional operators, which return the result of the last expression if they all resolve to a truthy value, or the first *falsey* value they encounter.

```
1 && 2 // returns 2
0 && 2 // returns 0
1 === 2 && 'Hello!' // returns false
2 === 2 && 'Hello!' // returns 'Hello!'
```

React will not render anything in place of `false`, but it will render `0`!

Conditional Rendering

Since an embedded expression renders the result of such expression, conditional rendering can be achieved like so:

```
function HelloWorld({ renderGreeting = true }) {
  return (
    <div>
      {renderGreeting && <h1>Hello, World!</h1>}
      <p>What a beautiful day!</p>
    </div>
  )
}
```

Conditional Rendering

An alternative and sometimes useful approach is to use a *ternary operator*:

```
function Welcome ({ name }) {
  return (
    <div>
      { name
        ? <h1>Hello, {name}!</h1>
        : <h1>Hello, World!</h1>
      }
    </div>
  )
}
```

The Virtual DOM

In order to be able to determine which part of the component tree needs to be re-rendered and how these changes are to be committed to the DOM tree - which is what is ultimately displayed in our browser - React uses the **Virtual DOM**.

The Virtual DOM

The Virtual DOM (or VDOM) is an internal data structure that mirrors what is currently being displayed in the browser.

Whenever an update occurs, this update is applied to the VDOM and the difference between the new VDOM and the previous one is used to determine which parts of the actual DOM need to be updated.

The Virtual DOM

It's good practice to ensure that the contents of the *root element* passed to `createRoot().render` are **never** altered by any other library that works on the DOM (such as jQuery, AngularJS, etc.) as such manipulation would make it impossible for React to reconcile any subsequent update inside the VDOM.

```
const root = createRoot(document.getElementById('root'))
root.render(<App />)
```

Events

Most applications need to be able to allow the user to interact with their UI so that the application's state can evolve over time and perform its intended function.

HTML pages expose a certain number of native events on HTML elements, as defined in the HTML standard.

Events

When using React in a browser, all native events are exposed as props on the default elements, using a *lower camel case* variation on their original HTML5 name.

This means that the `onclick` HTML event is exposed as the `onClick` prop on default elements such as `button`, `a`, `div`, etc.

Events

One of the most important differences between HTML and JSX in this regard is that while an HTML event attribute expects a *string* containing the JS code to run when the event occurs, event props on React elements expect a *reference to a function*:

```
<button onclick="handleButtonClick()">Click Me!</button>
```

```
<button onClick={handleButtonClick}>Click Me!</button>
```

Events

To call a function when an event triggers, a reference to that function must be passed as the value to the prop exposing that event on the relevant element. This function is called an *event handler*.

An element does not know, nor does it care, about what happens when an event fires; its only responsibility is to *notify* that the event has occurred by calling the function that it's given by its parent.

Events

By responding to events our app is now interactive, and it's therefore capable of responding to user input:

```
export function AlertClock() {
  function handleButtonClick() {
    alert('Current time: ' + new Date().toLocaleTimeString())
  }

  return (
    <div>
      <p>Click the button to see the current time.</p>
      <button onClick={handleButtonClick}>Click Me!</button>
    </div>
  )
}
```

Hooks

Hooks are a feature of React that makes it possible to add state and other functionality to function components.

Hooks are *functions* that can only be called within function components (or other hooks) and allow to track state variables, execute side effects and many other things that would otherwise not be possible within a function component.

Hooks

React exposes a set of prebuilt *hooks* that can be used as base building blocks for more complex logic, either within the component itself or in custom hooks. Some of the most common hooks are:

- `useState`: a hook that lets you manage the state of a component;
- `useEffect`: a hook that lets you run side effects in a component;
- `useContext`: a hook that lets you access the value of a context;
- `useRef`: a hook that lets you access the DOM node of a component;
- `useMemo`: a hook that lets you memoize expensive computations;

Hooks

Since hooks, as their name itself suggests, *hook* within React internals, there are some hard rules that must be followed when using them:

- Hooks can only be called from inside the body of a function component, or from within another hook;
- Hooks can only be called at the top level;
- The number of hooks called must not change from one render to another;

Hooks

A soft rule is that all custom hooks should have a name starting with `use`, followed by the name of the hook. While this is not enforced, it is *strongly* recommended to follow this convention.

`useState`

`useState` is a hook that lets you track a single *state variable*, and lets you update it.

```
export function Counter() {
  const [count, setCount] = useState(0)

  function handleIncrementCounter() {
    setCount((c) => c + 1)
  }

  return (
    <div>
      <p>Counter: {count}</p>
      <button onClick={handleIncrementCounter}>Increment</button>
    </div>
  )
}
```

`useState`

When called, `useState` returns an array of two values:

- The first value is the current state value;
- The second value is a function that can be used to update the state value.

The argument passed to `useState` is the initial value of the state value.

```
const [count, setCount] = useState(0)
```

`useState`

`useState` tracks a single variable at a time, but you can call it as many times as you want to track multiple state variables.

The *setter* function returned as the second element of the array is a function that can be used to update the value of the state variable in the first element of the array. This funtion can be called with either an immediate value or a function that returns the value to be set.

If passing an immediate value, the value will *overwrite* the previous value, even if it's an object.

`useEffect`

`useEffect` is a hook that lets you run side effects in a component. It has no return value, and expects two arguments:

- A *function* that will be called as the side effect as soon as the component mounts, as well as when the right conditions are met;
- An *array* of values that will be watched, and will cause the side effect to be re-run if any of them change. This array is often reffered to as the *dependency array*;

useEffect

```
export function Counter() {
  const [count, setCount] = useState(0)

  useEffect(() => {
    console.log('Current count:', count)
  }, [count])

  function handleIncrementCounter() {
    setCount((c) => c + 1)
  }

  return (
    <div>
      <p>Counter: {count}</p>
      <button onClick={handleIncrementCounter}>Increment</button>
    </div>
  )
}
```

useEffect

Since the side effect will be always called as soon as the component mounts, the dependency array can be empty: in this case, the side effect will be called only once, after the component mounts.

```
export function Greetings() {
  useEffect(() => {
    console.log('I have mounted!')
  }, [])

  return <h1>Hello!</h1>
}
```

useEffect

A side effect function can (but does not *have to*) return another function, called the *cleanup function*. This function will be called immediately *before* the next time the side effect is called, or before the component unmounts.

```
export function Greetings() {
  useEffect(() => {
    console.log('I have mounted!')

    return () => {
      console.log('I am unmounting...')
    }
  }, [])

  return <h1>Hello!</h1>
}
```

`useEffect`

A cleanup function is also useful when we need to clean up before running a side effect again, for example by clearing an interval before starting a new one, or closing a websocket connection before reconnecting.

Handling Events

When a user interacts with an application, the interaction will often carry over some information. We could, for example, be interested in knowing what text was inputted by the user inside a text field, or which button was pressed when clicking on a button, or even how many fingers were used to tap on the screen.

All events will carry information within them, and the type of information will be different depending on the type of event that was fired.

Handling Events

When an event calls an *event handler* it passes it an `event` object, containing all kinds of information about the fired event. In React, this object is an instance of `SynthethicEvent`.

Since the structure of an event can differ between browsers, React exposes the event within a *Synthetic Event*, which is nothing more than an abstraction that unifies such structure across all browsers.

The structure of a *Synthethic Event* follows the one defined by the W3C for HTML events.

Handling Events

```
function MouseClicker() {
  function handleButtonClick(event) {
    console.log(
      event.target.name, // myButton, the value of the `name` attribute
      event.timestamp,   // The number of ms expired since loading the page
      event.button       // The button used to click
    )
  }

  return (
    <button name="myButton" onClick={handleButtonClick}>Click me!</button>
  )
}
```

Handling Events

When accessing the information about an event, it's usually possible to get a *pointer to the DOM element that fired the event* by accessing the value of `event.target`.

This is often different from `event.currentTarget`, which is the pointer to the DOM element *to which the event handler is attached*.

This subtle difference is important, as HTML events *bubble* upwards after firing.

Handling Events

```
function FancyButton() {
  function handleButtonClick(event) {
    // event.target will point to either the `button` or the `Icon`,
    // depending on which was actually clicked by the user.
    //
    // event.currentTarget will always point to the `button`.
  }

  return (
    <button onClick={handleButtonClick}>
      <Icon name="checkmark" />
      Click the button!
    </button>
  )
}
```

Forms

Building an interactive application will often require the implementation of *Forms*.

Handling forms in React can be done using either *controlled* or *uncontrolled* components.

Forms

A **controlled component** is a component that does not keep an internal state of its content, but relies on the parent component to provide it with its current value and notifies its parent whenever the user attempts to change it.

An **uncontrolled component** is a component that keeps its value within its internal state. It may or may not notify the parent component of a change, but does not rely on the parent to provide it with its current value.

Controlled Components

A *controlled component* relies on its parent to provide it with its current value *at any given time*, and notifies the parent whenever the user attempts to change its value.

```
function MyForm() {
  const [username, setUsername] = useState('')

  function handleUsernameInputChange(event) {
    setUsername(event.target.value)
  }

  return <input name="username" value={username} onChange={handleUsernameInputChange} />
}
```

Controlled Components

Every time a user types into an `input` component, its `onChange` event is fired. The `onChange` event will contain the *new* value inside the `value` attribute of `event.target`, and this new value can be used to update the state of the parent component.

Updating the state will cause the parent component to re-render itself, calling `render` again and passing a new `value` prop to the `input` component.

Controlled Components

A form usually has more than one field, and it's not practical to have different state variables or different event handlers for each of them.

Since the `event` object carries within itself a pointer to the DOM element that has caused the event to fire, we can use a single event handler to handle events fired by multiple fields.

This event handler will make use of *destructuring* and *dynamic property keys* to update the proper part of a state *object*, which will contain all the values of the form.

Controlled Components

```
function MyForm() {
  const [data, setData] = useState({
    username: '',
    password: '',
  })

  function handleInputChange(event) {
    const { name, value } = event.target // Destructure the required attributes

    setData((data) => {
      return {
        ...data, // Keep the previous state
        [name]: value, // Dynamically set the key specified in `name` to `value`
      }
    })
  }

  return (
    <div>
      <input name="username" value={data.username} onChange={handleInputChange} />
      <input name="password" value={data.password} type="password" onChange={handleInputChange} />
    </div>
  )
}
```

Uncontrolled Components

An *uncontrolled component* keeps track of its value within its own state and does not rely on a value passed down by the parent component as a prop.

The parent component will need to be able to access to the DOM element *directly* in order to access the value contained within the *uncontrolled component* or to change its value.

Uncontrolled Components

When writing forms using uncontrolled components, the default HTML Form API should be used to access the value of the input:

```
export function MyUncontrolledForm() {
  function handleFormSubmit(event) {
    event.preventDefault()

    const username = event.target.elements.username.value
    console.log(username)
  }

  return (
    <form onSubmit={handleFormSubmit}>
      <input name="username" />
    </form>
  )
}
```

Uncontrolled Components

The `onSubmit` method of a `form` element is triggered whenever the form itself is submitted: this can happen either when a button of `type` `submit` is clicked, or when the user presses the `Enter` key when focusing one of the form's fields.

Uncontrolled Components

The event handler should call the `preventDefault` method on the event to avoid HTML's default behavior of attempting to perform a `GET` request to the page.

In order to access the form's fields, the `event.target` property should be used by accessing the `elements` object within it. Many browsers support accessing the elements directly on `event.target`, but by using the `elements` object full compatibliity is guaranteed.

Refs

React allow DOM elements to be accessed *directly* through the use of **refs**. Refs are a way to access a component or an element directly in order to access its values or modify it *imperatively* rather than *declaratively*.

Refs can be useful to manage focus and text selection or to integrate React components with animation libraries or other third party libraries.

Refs

A *ref* can be created using the `useRef` hook, exported by the `react` package.

`useRef` will return a ref that can be attached to any component by passing it to the `ref` prop.

The ref will contain a pointer to the DOM Element on its `current` attribute as soon as the Element is rendered, but it will contain the value passed to `useRef` before that (usually `null` or `undefined`).

Refs

```
function MyUncontrolledForm() {
  const _inputRef = useRef(null)

  function handleFocus() {
    _inputRef.current?.focus()
  }

  return (
    <form>
      <input name="username" ref={_inputRef} />
      <button type="button" onClick={handleFocus}>
        Focus the input
      </button>
    </form>
  )
}
```

Rendering Lists

Rendering a list in React is as easy as calling the `map` method on an Array.

The `map` method receives a *callback function* that is then executed for each element within the array, receiving the element itself as its first parameter, and its index in the array as the second one.

```
const numbers = [1, 2, 3, 4]
const double = numbers.map((number, index) => number * 2)
```

Rendering Lists

The `map` function is used to *transform* each element of an array into something else, as defined within the callback function.

The return value of `map` is a **new** array containing the result of calling the callback on all the elements of the original array.

The original array is *not mutated*.

Rendering Lists

Through the `map` function an array of items can be *transformed* into an array of JSX elements:

```
const names = ['Kate', 'Jane', 'John', 'Billy']
const items = names.map((name) => <li>{name}</li>)
```

This is list of elements can then be rendered within a component, as React knows how to handle arrays when they are found within a JSX snippet.

Rendering Lists

```
export function MyList({ names }) {
  const renderNames = names.map((name) => <li>{name}</li>)

  return <ul>{renderNames}</ul>
}
```

```
<MyList names={['Kate', 'Jane', 'John', 'Billy']} />
```

Keys

When rendering an array of elements inside a component, each element **must** have a **unique** and **stable** key prop assigned, as React uses it to identify which items in the array have changed or have been added or removed.

```
export function MyList({ names }) {
  const renderNames = names.map((name, index) => <li key={name + index}>{name}</li>)

  return <ul>{renderNames}</ul>
}
```

Keys

Keys must also always be assigned to outermost element when rendering a list:

```
export function MyListItem({ name }) {
  return <li>{name}</li>
}

export function MyList({ names }) {
  return (
    <ul>
      {names.map((name, index) => (
        <MyListItem key={name + index} name={name} />
      ))}
    </ul>
  )
}
```

Styling Components

As long as React is used to render a *web* application, CSS can be used as easily as with regular HTML, but by using the `className` prop instead of the `class` attribute:

```
<!-- THIS IS HTML -->
<button class="button button-success">Click Me!</button>
```

```
/* This is React */
<button className="button button-success">Click Me!</button>
```

Styling Components

Once you have your `.css` file, you can use it in your React application simply by importing it in your project. CSS rules are always *global*, so you can import your CSS within your `index.js` file:

```
// ...
import './styles.css' // <-- we import the CSS file

const rootElement = document.getElementById('root')
// ...
```

The CSS file will *not* actually get imported into your JS, but will be included in the bundle created by your toolchain of choice (for example Vite).

Styling Components

If you prefer to use `sass` instead of plain CSS, you can do so by simply installing the `sass` package in your project using `npm`:

```
npm install sass
```

After installing the package, you should be able to import `.scss` files directly into your components:

```
import './styles.scss'
```

This will work, provided that your toolchain is correctly configured (as Vite is.)

Styling Components

Another useful tool for styling React components are **CSS Modules**. These are CSS files that are *scoped* to a specific module (the one they are imported in), and allow to use the same class names in different components without worrying about name collisions.

A CSS Module is defined a regular CSS (or SCSS) file, but with the `.module.css` (or `.module.scss`) extension:

```
/* filename: button.module.css */
.button {
  background-color: #333;
  color: white;
  margin: 10px 20px;
}
```

Styling Components

To correctly use a CSS Module, we simply need to import its *classes* into our component:

```
import styles from './button.module.css'

export function MyButton() {
  return <button className={styles.button}>Click Me!</button>
}
```

Styling Components

React also supports *inline* styles through the `style` prop, which can receive an *object* containing the style definition for that element:

```
export function MyComponent() {
  const MyStyle = {
    backgroundColor: '#333',
    color: 'white',
    margin: '10px 20px',
  }

  return <div style={MyStyle}>Hello!</div>
}
```

Styling Components

The advantage of using inline styles is that they can change depending on the component's props or state:

```
export function MyComponent({ active }) {
  const MyStyle = {
    backgroundColor: active ? 'yellow' : '#333',
    color: active ? 'black' : 'white',
    margin: '10px 20px',
  }

  return <div style={MyStyle}>Hello!</div>
}
```

Styling Components

Inline styles can also be passed *directly* to the `style` prop, without an intermediate variable:

```
export function MyComponent({ active }) {
  return (
    <div
      style={{
        backgroundColor: active ? 'yellow' : '#333',
        color: active ? 'black' : 'white',
        margin: '10px 20px',
      }}
    >
      Hello!
    </div>
  )
}
```

Component Composition

Composition is a very important aspect of React, and allows to use components as building blocks that can be composed to build complex UIs.

HTML can be composed easily by nesting them within one another. The same can be done with React components.

Component Composition

When using HTML `div`s, for example, we often nest other tags within them:

```
<div>
  <h1>Hello!</h1>
  <p>This is a paragraph.</p>
</div>
```

Component Composition

React components can be nested as well:

```
export function MyComponent() {
  return (
    <Container title="My App">
      <Welcome name="Jimmy" />
      <p>This is a paragraph.</p>
    </Container>
  )
}
```

Component Composition

Whenever a component receives other components as *children*, it can access them through the `children` prop. By using the `children` prop, the component can render its children within its component subtree:

```
export function Container({ title, children }) {
  return (
    <div className="container">
      <div className="container-title">{title}</div>
      <div className="container-content">{children}</div>
    </div>
  )
}
```

Component Composition

The **only** special property of the `children` prop is that it's automatically filled by React whenever a component has children. A JSX expression can, of course, be passed to *any* prop:

```
export function MyComponent() {
  return (
    <Container title={<h1>My App</h1>}>
      <Welcome name="Jimmy" />
      <p>This is a paragraph.</p>
    </Container>
  )
}
```

Context

When dealing with complex component trees, it's often useful to pass some data down from a topmost ancestor to one or more descendants.

While this can be done by manually passing data down as props, the props need to be passed down to every component in the tree until the desired one is reached and this can prove to be cumbersone, while also adding a lot of *noise* to the code.

Context

React's **Context API** provides a way to pass data down the component tree without having to pass it down manually.

A *context* is a container that *provides* a specific value to any *consumer* that wishes to access it, regardless of its position in the component tree, as long as its contained within said *provider*.

Context

To create a context, the `createContext` function exported by the React library is used:

```
export const LanguageContext = createContext('en')
```

The `createContext` function takes an optional argument, which is the default value of the context.

The value returned by the `createContext` function is an object containing two components: `Provider` and `Consumer`.

Context

```
export function Root() {
  const [language, setLanguage] = useState('en')

  function handleLanguageChange(language) {
    setLanguage(language)
  }

  return (
    <LanguageContext.Provider value={language}>
      <LanguageSelector onLanguageChange={handleLanguageChange} />
      <App />
    </LanguageContext.Provider>
  )
}
```

Context

In this example, the `LanguageContext` object is used to pass the `language` state down the component tree. **Any** component within the `LanguageContext.Provider` component can access the `language`.

Context

The *easiest* way to access the value of a context from a component within the context provider's *component tree* is to use the `useContext` hook:

```
export function MyDeeplyNestedComponent() {
  const language = useContext(LanguageContext)

  return (
    <div>
      <p>The current language is: {language}</p>
    </div>
  )
}
```

Context

The `useContext` hook takes a context object as its argument and returns the value of the context, or the default value of the context if the component is not within the context provider's component tree.

Context

The Context API must be used *sparingly*, as it makes components consuming a context *less reusable*.

Context should only be used when a specific value needs to be accessed by *multiple* components and passing it down the tree manually adds significant *noise* to the code.

Common examples of when to use the Context API are: currently selected language, UI theme, user application settings, etc.

Data Fetching

An application will often have to fetch data from a remote server in order to show it to the user. Browser APIs implement this by using the `fetch` function, which uses a promise-based API:

```
function getGithubUser(username) {
  return fetch(`https://api.github.com/users/${username}`)
    .then((response) => response.json())
}
```

Data Fetching

The `fetch` function has a few peculiarities:

- It returns a promise;
- It does not throw an error if the request returns a status code other than 200, but only when there is a network error;
- The response object allows to use different functions to access the body, depending on the type of the response (`json`, `blob`, `text`).

Data Fetching

We can use a combination of the `useState` and `useEffect` hooks to implement data fetching within a component.

Since we cannot call fetch every time the component renders, we need to use the `useEffect` hook to call `fetch` only at an appropriate time. We can then use the `useState` hook to store the data we get from the server, and render it in the component.

Data Fetching

```
export function GithubUser({ username }) {
  const [user, setUser] = useState(null)

  useEffect(() => {
    if (!username) {
      return
    }

    fetch(`https://api.github.com/users/${username}`)
      .then((response) => response.json())
      .then((user) => setUser(user))
  }, [username])

  return (
    <div>
      {!user && <p>Loading...</p>}
      <h1>{user?.name}</h1>
      <p>{user?.bio}</p>
    </div>
  )
}
```

Data Fetching

Of course, the same logic can be implemented using `async` / `await`. In this case we have to make sure to wrap the `fetch` call inside an `async` function, since the side effect function passed to `useEffect` **cannot** be an `async` function.

Data Fetching

```
export function GithubUser({ username }) {
  const [user, setUser] = useState(null)

  async function fetchUser(username) {
    if (!username) {
      return
    }

    const response = await fetch(`https://api.github.com/users/${username}`)
    const user = await response.json()
    setUser(user)
  }

  useEffect(() => {
    fetchUser(username)
  }, [username])

  return (
    <div>
      {!user && <p>Loading...</p>}
      <h1>{user?.name}</h1>
      <p>{user?.bio}</p>
    </div>
  )
}
```

Data Fetching

When fetching data we usually want to deal with error and loading states. We can use the `useState` hook to store the error and loading states, and render them in the component.

When using `async` / `await` We can also wrap the `fetch` call in a `try/catch` block, and handle the error in the `catch` block.

Data Fetching

```
export function GithubUser({ username }) {
  // ...
  async function fetchUser(username) {
    try {
      setLoading(true)
      const response = await fetch(`https://api.github.com/users/${username}`)
      const user = await response.json()
      setUser(user)
    } catch (error) {
      setError(error)
    } finally {
      setLoading(false)
    }
  }

  useEffect(() => {
    fetchUser(username)
  }, [username])

  return (
    /* ... */
  )
}
```

Custom Hooks

The true power of *hooks* is that they can be used to implement custom, **reusable** logic.

By using other hooks such as the ones provided by React or ones we either wrote ourselves or have been written by others as *base building blocks*, we can define custom logic that can be shared and reused in multiple components.

Custom Hooks

A custom hook is a *function* that can receive any amount of parameters and can have a return value of any type. It must follow the same rules as other hooks, and its code will be executed in a context that will be specific to the instance of the component it is used in.

Custom Hooks

```
export function useCounter(initialValue = 0) {
  const [count, setCount] = useState(initialValue)

  function handleIncrement() {
    setCount((c) => c + 1)
  }

  function handleDecrement() {
    setCount((c) => c - 1)
  }

  return { count, handleIncrement, handleDecrement }
}
```

This simple example shows how to create a custom hook to manage a counter. Any component will be able to use it without needing to know anything about the implementation details of the counter.

Custom Hooks

When writing a custom hook we encapsulate custom logic within it, making its implementation *opaque* to any component that uses it.

```
export function Counter({ initialValue = 0 }) {
  const { count, handleIncrement, handleDecrement } = useCounter(initialValue)

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={handleIncrement}>+</button>
      <button onClick={handleDecrement}>-</button>
    </div>
  )
}
```

Custom Hooks

```
export function useGithubUser(user) {
  const [user, setUser] = useState(null)
  const [error, setError] = useState(null)
  const [loading, setLoading] = useState(false)

  async function fetchUser(username) {
    try {
      setLoading(true)
      const response = await fetch(`https://api.github.com/users/${username}`)
      const user = await response.json()
      setUser(user)s
    } catch (error) {
      setError(error)
    } finally {
      setLoading(false)
    }
  }

  useEffect(() => {
    fetchUser(user)
  }, [user])

  return { user, error, loading }
}
```

Custom Hooks

```
export function GithubUser({ username }) {
  const { user, error, loading } = useGithubUser(username)

  return (
    <div>
      {loading && <p>Loading...</p>}
      {error && <p>Error!</p>}
      <h1>{user?.name}</h1>
    </div>
  )
}
```

By extracting our custom logic from the component we can reuse it in other components and, most importantly, our components will not include any logic that is not *presentational*.

`useCallback`

`useCallback` is a hook provided by React that *memoizes* a function.

If we define a function within a function component, a new version of that function will be created every time the component is rendered. This is normally not a problem, but there are certain edge cases where this creates issues such as infinite loops and, very rarely, performance degradation.

useCallback

```
function Counter({ initialValue = 0 }) {
  const [count, setCount] = useState(initialValue)

  function handleIncrement() {
    setCount((c) => c + 1)
  }

  return (
    <div>
      <h1>{count}</h1>
      <button onClick={handleIncrement}>Increment</button>
    </div>
  )
}
```

In this example, a new version of `handleIncrement` is created every time the `Counter` component is rendered.

`useCallback`

In order to be able to use tha *same* version of a function every time the component is rendered, we need to memoize it. We can do this by using the `useCallback` hook:

```
const handleIncrement = useCallback(function handleIncrement() {
  setCount((c) => c + 1)
}, [])
```

`useCallback` receives a function as its first argument, and an array of dependencies as its second argument. It will return the memoized version of the function or a new version if one of the dependencies has changed.

`useCallback`

The use case for `useCallback` is for when we need to pass functions to optimized components and we want to avoid unnecessary re-renders.

Another use case is when a component receives a function as a prop and uses it as a dependency in a hook (for example `useEffect`). A memoized function will not trigger the execution of an effect unless it has actually change, while passing a non-memoized function will cause the effect to be executed every time the component is rendered.

`useMemo`

There are some cases where we want to memoize a value that is expensive to compute. For example, an array might need to be filtered and sorted before being rendered, and if the array is large enough the impact on performance could be significant.

The `useMemo` hook allows us to memoize and repeat its computation only when its dependencies change:

```
const expensiveValue = useMemo(() => {
  return calculateExpensiveValue()
}, [])
```

`useMemo`

The first parameter passed to `useMemo` is a function that will be called to calculate the memoized value. The value returned by this function will be the one that will be memoized.

As with `useEffect` and `useCallback`, the array passed to `useMemo` as its second parameter is used to determine if the memoized value should be re-calculated. If the array is empty, the memoized value will be calculated only once, when the component is first mounted.

useMemo

```
export function ActiveUsersList({ users }) {
  const activeUsers = useMemo(() => {
    return users.filter((user) => user.isActive)
  }, [users])

  return (
    <div>
      {activeUsers.map((user) => (
        <div key={user.id}>{user.name}</div>
      ))}
    </div>
  )
}
```

This hypothetical component receives a list of *all* users but shows only the active ones. We can memoize the list of active users to avoid re-calculating it every time the component is rendered.

React Router

React Router is **third party** library that handles the routing of our application.

Since React does not provide any kind of routing mechanism, we need to use a third party library to handle the routing, or we need to interact directly with the browser's history API.

React Router

To install React Router, we need to install the `react-router-dom` package:

```
$ npm install --save react-router-dom
```

This will also install the `react-router` package, which is the base package for React Router.

React Router

Once the library is installed, the first step is to wrap our application's root component in the `BrowserRouter` component:

```
export function Root() {
  return (
    <BrowserRouter>
      <App />
    </BrowserRouter>
  )
}
```

This will allow all components within our `App` component to use React Router's features.

React Router

React Router allows us to define *routes*, which are components rendered *conditionally* depending on the current URL.

To define a set of routes, we can use the `Routes` component, passing it any number of `Route` components as children:

```
export function App() {
  return (
    <Routes>
      <Route path="/" element={<Welcome />} />
      <Route path="login" element={<Login />} />
    </Routes>
  )
}
```

React Router

When we define a `Route` we need to specify which *path* it will match. Whenever the current URL matches the route's path, the route's *element* will be rendered.

Since the `element` prop expects a React element, we can pass it *any* React component, complete with any prop we might want to pass it.

React Router

Whenever we want to *navigate* from one route to another, we can use React Router's `Link` component:

```
export function Welcome() {
  return (
    <div>
      <h1>Hello, World!</h1>
      <Link to="/login">Enter the app</Link>
    </div>
  )
}
```

The `Link` component expects a `to` prop, which is the URL that the link will navigate to. It works similarly to the `href` attribute of an `<a>` element.

React Router

It's also possible to manage navigation *imperatively* by using the `useNavigate` hook:

```
export function Welcome() {
  const navigate = useNavigate()

  function handleClick() {
    navigate('/login')
  }

  return (
    <div>
      <h1>Hello, World!</h1>
      <button onClick={handleClick}>Enter the app</button>
    </div>
  )
}
```

React Router

Whenever we define a route, we can tell React Router that that route expects a *parameter* to be passed to it. This is done by passing a string to the route's `path` with a placeholder for the parameter:

```
export function Root() {
  return (
    <Routes>
      <Route path="/:name" element={<Welcome />} />
      <Route path="login" element={<Login />} />
    </Routes>
  )
}
```

Each parameter within a route's path is defined by a placeholder, which is a `:` followed by the parameter's name.

React Router

When rendering a component that is rendered as a Route and expects a parameter, we can use the `useParams` hook to access the parameter:

```
export function Welcome() {
  const { name } = useParams()

  return (
    <div>
      <h1>Hello, {name}!</h1>
    </div>
  )
}
```

The parameter will have the same name as the placeholder we have used to identify it in the route. This makes it possible to have *multiple* parameters in a single route.

React Router

Applications are usually layed out in a hierarchy of routes. We will sometime want to keep showing a *parent* route while showing a *child* route.

React Router

React Route allows for nested routes by passing *children* to the `Route` component:

```
export function Root() {
  return (
    <Routes>
      <Route path="/" element={<Welcome />} />
      <Route path="products" element={<Catalogue />}>
        <Route path=":id" element={<Product />} />
      </Route>
    </Routes>
  )
}
```

React Router

When defining *nested* routes it's not necessary to repeat the whole path for the child route: React Router will append the child route's path to the parent route's path.

React Router

A nested route will be rendered *inside* the parent route's element. In order to render the child route's element inside the parent route's element, we need to include an `Outlet` component inside the parent:

```
export function Catalogue() {
  return (
    <div>
      <h1>My Shop Catalogue</h1>
      <Link to="music-player">Music Player</Link>
      <Link to="acoustic-guitar">Acoustic Guitar</Link>

      <Outlet />
    </div>
  )
}
```

React Router

When rendering nested parametric routes we'll also probably want to render a *parameterless* route, also known as an *index route*:

React Router

```
export function App() {
  return (
    <Routes>
      <Route path="/" element={<Welcome />} />
      <Route path="products" element={<Catalogue />}>
        <Route
          index
          element={
            <div>
              <Link to="music-player">Music Player</Link>
              <Link to="acoustic-guitar">Acoustic Guitar</Link>
            </div>
          }
        />
        <Route path=":id" element={<Product />} />
      </Route>
    </Routes>
  )
}
```

React Router

Another common use case is to render a Route that matches whenever no other Route matches. This is done by using a special path value of `"*"`:

```
export function App() {
  return (
    <Routes>
      <Route path="/" element={<Welcome />} />
      <Route path="*" element={<NotFound />} />
    </Routes>
  )
}
```

Any time the current URL doesn't match any of the other routes, the `NotFound` element will be rendered.

SWR

SWR is a **third party** library that helps us fetch data from a remote server and intelligently cache it, always providing the user with the latest data and avoiding unnecessary network requests.

Data Fetching can be done manually by using default hooks such as `useState` and `useEffect`, but many aspects such as caching, deduplication, and error handling need to be handled manually. SWR takes care of all of these aspects automatically.

SWR

To install SWR we need to install the `swr` package:

```
$ npm install --save swr
```

SWR

Once SWR is installed, any component can benefit from it by importing the `useSWR` hook and passing it a URL and a function to fetch data from the remote server:

SWR

```
import useSWR from 'swr'

const fetcher = (url) => fetch(url).then((r) => r.json())

export function GithubUsers() {
  const { data, error } = useSWR('https://api.github.com/users', fetcher)

  return (
    <ul>
      {!data && !error && <div>Loading...</div>}
      {error && <div>An error has occurred</div>}
      {data.map((user) => (
        <li key={user.id}>{user.login}</li>
      ))}
    </ul>
  )
}
```

SWR

The `fetcher` function is a function that receives the URL and returns a promise. The promise will resolve with the data fetched from the remote server.

```
const fetcher = (url) => fetch(url).then((r) => r.json())
```

We can use either the default `fetch` API or any other third party HTTP client library.

SWR

While components can use `useSWR` directly, it's considered good practice to create a custom hook that wraps `useSWR` for each resource we want to fetch:

```
const fetcher = (url) => fetch(url).then((r) => r.json())

export function useGithubUsers() {
  const { data, error } = useSWR('https://api.github.com/users', fetcher)

  return { users: data ?? [], error, loading: !data && !error }
}
```

By abstracting the fetching logic in a custom hook, components don't need to know about the details of how the data is fetched.

SWR

If our data changes *locally* and we want to trigger a re-render, we can use the `mutate` function, returned by `useSWR`.

The `mutate` function allows to mutate the *local cache* and initiate a re-fetch of the data, in order to update the local cache with fresh data. This is useful to create interfaces that use an optimistic update strategy.

By calling mutate with no arguments, we'll simply trigger a re-fetch of the data.

SWR

```
function useGithubUsers() {
  const { data, error, mutate } = useSWR('https://api.github.com/users', fetcher)

  return { users: data ?? [], error, loading: !data && !error, onRefresh: () => mutate() }
}

function GithubUsers() {
  const { users, error, loading, onRefresh } = useGithubUsers()

  return (
    <ul>
      <button onClick={onRefresh}>Refresh</button>
      {!data && !error && <div>Loading...</div>}
      {error && <div>An error has occurred</div>}
      {data.map((user) => (
        <li key={user.id}>{user.login}</li>
      ))}
    </ul>
  )
}
```

SWR

Writing the `fetcher` function for all requests can be cumbersome. SWR can be configured to use a single fetcher function for all requests by wrapping the App with a `SWRConfig` component:

```
const fetcher = (url) => fetch(url).then((r) => r.json())

function Root() {
  return (
    <SWRConfig
      value={{
        fetcher,
      }}
    >
      <App />
    </SWRConfig>
  )
}
```

SWR

The `SWRConfig` component receives a single `value` prop containing all configuration options that will be shared by all calls to `useSWR`, `fetcher` being one of them.

`SWRConfig` can also be used to specify custom cache invalidation options and other rules used by SWR to handle its data.