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 `create-react-app`. In a terminal, type:

```
$ npx create-react-app my-app
```

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.

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. This course will cover both, starting with **class components**.

Hello, World

This is the simplest component can be written as a *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;
- The `props` are the information the component can receive from the outside.

Components

The JSX expression returned from the `render` method can only have **one** root element. When more than one element needs to be returned, it must be wrapped in a containing element:

```
class HelloWorld extends React.Component {
  render() {
    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 `ReactDOM.render` function:

```
ReactDOM.render(helloWorldElement, document.querySelector('#root'))
```

The `ReactDOM.render` function will update the DOM within the `#root` element to match the UI defined in our `helloWorldElement` variable.

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 the name is arbitrary) 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 ReactDOM from 'react-dom'

class App extends React.Component {
  render() {
    return (
      <div>
        <h1>My Awesome Application</h1>
        <HelloWorld />
      </div>
    )
  }
}

ReactDOM.render(<App />, document.querySelector('#root'))
```

Component Composition

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

Avoid calling `ReactDOM.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. *Class components* can access their props using the `props` property on their own instance:

```
class Welcome extends React.Component {
  render() {
    return <h1>Welcome, {this.props.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:

```
class HelloWorld extends React.Component {
  render() {
    return (
      <div>
        {this.props.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*:

```
class Welcome extends React.Component {
  render() {
    return (
      <div>
        {this.props.name
          ? <h1>Hello, {this.props.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 `ReactDOM.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.

```
ReactDOM.render(
  <App />,
  document.querySelector('#root'), // Never touch this element again!
)
```

State

Any *class component* can keep track of an internal **state**. A state is an *object* that can be filled with data of any kind, and this data can change over time following the component's own rules.

Any time the state of a component changes, the component will re-render itself and its children components.

State

The *state* of a component can be *initialized* within its **constructor**:

```
class Clock extends React.Component {
  constructor(props) {
    super(props)
    this.state = {
      now: new Date(),
    }
  }

  render() {
    return <h1>It's now {this.state.now.toLocaleTimeString()}</h1>
  }
}
```

State

The *state* can also be initialized as a *class property*:

```
class Clock extends React.Component {
  state = {
    now: new Date(),
  }

  render() {
    const { now } = this.state // <- Destructuring

    return <h1>It's now {now.toLocaleTimeString()}</h1>
  }
}
```

State

The state can be updated using the `setState` method, available on all class components.

```
constructor(props) {
  super(props)

  setInterval(() => {
    this.setState({ now: new Date() })
  }, 1000)
}
```

State

The call to `setState` notifies React that the state of a component needs to be updated.

The update itself is not *synchronous*: React enqueues it in a list of operations that are going to be performed during the next update cycle.

Calling `setState` will also tell React that the component needs to be re-rendered, causing such update cycle to fire.

State

`setState` can be called in either one of two ways:

- by passing it an object that contains the new *state* (or a portion of it)

```
this.setState({ now: newValue })
```

- by passing it a *callback* that receives the *current state* and returns the new state.

```
this.setState((state) => ({ now: newValue }))
```

State

Since updates to the state can be *batched together* by React to improve performance, our state update should not rely on the state's value available in the current context.

Whenever the *next value* of the state depends in some way from the *current value* of the state, `setState` needs to be called with a callback rather than object.

State

Calling `setState` is necessary for React to know that a state update has happened.

Trying to update the state directly by assigning new values to the state object will result in the component *not updating* due to React not having any knowledge that such update has happened!

```
setInterval(() => {
  // This will **NOT WORK**
  this.state.now = new Date()
}, 1000)
```

Component Lifecycle

All components go through well-defined and specific phases during their lifecycle. These phases are the **mounting** phase, the **update** phase and the **unmounting** phase.

Component Lifecycle - Mounting

The **mounting** phase is the phase of the lifecycle of a component during which the component is created and attached to the application's component tree. It starts with the creation of the component element and ends after the component's element first render.

During this phase three methods are called on the component's class instance: `constructor`, `render` and `componentDidMount`.

Component Lifecycle - Mounting

`constructor`

The constructor is a method native to all JS classes and is called automatically when the class is instanced with the `new` keyword. When using components we do not use the `new` keyword directly, but we let React do it for us behind the scenes.

On a React component, the constructor is called as soon as the component is instanced by React, thefore only once for each instance of the component.

Component Lifecycle - Mounting

`constructor`

We can use the constructor to do all those initializations that do not require for the component to have already rendered something on the DOM. We can, for example, use the constructor to initialize the component's internal state.

It's important to remember that, while running the constructor, the component *has not yet rendered on the screen*.

Component Lifecycle - Mounting

`render`

The `render` method, required for all class components, is called by React every time a component needs to render. This includes the very first time the component renders, and it's therefore part of the *mounting* lifecycle phase.

The `render` method returns a JSX expression that tells React how the component renders itself on the screen.

Component Lifecycle - Mounting

`componentDidMount`

This *optional* method is called right **after** the component has been rendered the first time, and can be used to perform all initializations that require the component to have already rendered, or all those operations that would not have completed before the first render if called within the `constructor`.

We can, for example, start a *data fetching* request, start a timer, manually attach event handlers, connect to websockets etc.

Component Lifecycle - Mounting

```
class MyComponent extends React.Component {
  loadDataFromServer() { /* ... */ }

  componentDidMount() {
    this.loadDataFromServer()
  }

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

Component Lifecycle - Updating

Every time a component needs to be updated it goes through the *updating* lifecycle phase. This phase happens as many times as required as long as the component is part of the application's component tree.

The methods that are called during this phase are `shouldComponentUpdate`, `render` and `componentDidUpdate`.

Component Lifecycle - Updating

`shouldComponentUpdate`

This *optional* method is called by React as soon as the update cycle of a component starts, and must return either `true` or `false` depending on whether we want the component to update or not.

This method has access to both the current state and the current props, via the `this` keyword, and the next props and the next state, received as arguments of the method itself.

Component Lifecycle - Updating

`shouldComponentUpdate`

```
class MyComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return true
  }
}
```

While it's possible to use this method to conditionally stop a component from updating, its use is highly discouraged and should only be considered for very specific use cases.

Component Lifecycle - Updating

`render`

The `render` method is called *every time* React needs to render a component. Props and state *change over time*, either as a consequence of a user interaction or because other events have triggered a change, and every time the component goes through an update cycle it is rendered again by React calling the `render` method.

Component Lifecycle - Updating

`componentDidUpdate`

This *optional* method is called after every render subsequent to the first one. It can be used to compare the old props and the old state with the current ones and use this information to start data fetching operations, connect or disconnect from websockets, start and stop timers, etc.

Component Lifecycle - Updating

```
class MyComponent extends React.Component {
  fetchUser() {
    /* ... */
  }

  componentDidUpdate(prevProps, prevState) {
    if (this.props.userId !== prevProps.userId) {
      this.fetchUser()
    }
  }

  render() {
    /* ... */
  }
}
```

Component Lifecycle - Unmounting

The umounting phase happens whenever a component is *removed* from the application's component tree. This usually happens when a component is rendered conditionally, or when a component is rendered as part of a list of components that changes.

When a component is being unmounted by React, a single method is called: `componentWillUnmount`.

Component Lifecycle - Unmounting

`componentWillUnmount`

This *optional* method is called *right before* a component is removed from the application's component tree, and allows us to perform *cleanup* operations such as disconnecting from websockets, stop timers, etc.

If we have pending operations that might end *after* the component has been removed, we should make sure to cancel them within this method to avoid attempting state updates on a component that has been removed.

Component Lifecycle - Unmounting

```
class MyComponent extends React.Compnent {
  constructor(props) {
    super(props)
    this.state = { now: new Date() }
    this._interval = setInterval(() => {
      this.setState({ now: new Date() })
    }, 1000)
  }
  componentWillUnmount() {
    clearInterval(this._interval)
  }
  render() {
    /* ... */
  }
}
```

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

```
class Counter extends React.Component {
  state = { count: 0 }

  handleCounterIncrement() {
    this.setState((state) => {
      return {
        count: state.count + 1,
      }
    })
  }

  render() {
    return (
      <button onClick={this.handleCounterIncrement.bind(this)}>
        Counter: {this.state.count}
      </button>
    )
  }
}
```

Events

Note that the reference to the *event handler* is explicilty bound to `this`:

```
<button onClick="{this.handleCounterIncrement.bind(this)}">Counter: {this.state.count}</button>
```

This is required because of how the `this` keyword works in JS. The reference to the event handler will be called internally by the `button` element and not directly on the instance of the component. If the reference is not explicitly bound, calling `this.setState` will throw an *exception* because `this` will be `undefined`.

The `this` keyword

In JS, The value of the `this` keyword is determined by the way a function is *called* rather then how a function is *defined*.

A method defined on a class has no special attachment to that class or its instances, and the value of its `this` keyword is determined entirely by how the method itself is called.

The `this` keyword

When calling a method on an object, that method's `this` keyword is automatically bound to that object:

```
class Person {
  constructor(name) {
    this.name = name
  }
  sayHello() {
    return `Hello, I am ${this.name}!`
  }
}

const kate = new Person('Kate')
kate.sayHello() // Hello, I am Kate!
```

The `this` keyword

When saving a reference to a method and calling it through the reference, the `this` keyword is however set to `undefined` (when using *strict mode*, as you should).

```
const kate = new Person('Kate')
// We save a **reference** to the function
const hello = kate.sayHello
// We **invoke** the reference
hello()
// An exception is thrown: cannot access property 'name' on undefined.
```

The `this` keyword

There are several ways to work around this behavior. One of such ways is to use the `bind` method, available on all function references, to create a new function with a specific, predefined value for `this`:

```
const kate = new Person('Kate')
// We save a **reference** to the function returned by `bind`. This reference has
// `this` explictly set to `kate`.
const hello = kate.sayHello.bind(kate)
// We **invoke** the reference
hello()
// "Hello, I am Kate!"
```

The `this` keyword

Another common workaround is to define class methods as arrow functions. An arrow function does not have its own `this` keyword, but rather uses the `this` of the context in which is *defined*:

```
class Person {
  constructor(name) {
    this.name = name
  }

  sayHello = () => {
    return `Hello, I am ${this.name}!`
  }
}
```

The `this` keyword

```
class Counter extends React.Component {
  state = { count: 0 }

  // Defined as an arrow function
  handleCounterIncrement = () => {
    this.setState((state) => {
      return {
        count: state.count + 1,
      }
    })
  }

  render() {
    // No need to bind anymore
    return <button onClick={this.handleCounterIncrement}>Counter: {this.state.count}</button>
  }
}
```

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.

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.

Events

```
class MouseClicker extends React.Component {
  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
    )
  }

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

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.

Events

```jsx
class FancyButton extends React.Component {
  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`.
  }

  render() {
    return (
      <button onClick={this.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.

```
class MyForm extends React.Component {
  state = { username: '' }

  handleUsernameInputChange = (event) => {
    this.setState({ username: event.target.value })
  }

  render() {
    return <input name="username" value={this.state.username} onChange={this.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 a different event handler for each field.

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 the state.

Controlled Components

```
class MyForm extends React.Component {
  state = {
    username: '',
    password: '',
  }

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

    this.setState({
      [name]: value, // Dynamically set the key specified in `name` to `value`
    })
  }

  render() {
    return (
      <div>
        <input name="username" value={this.state.username} onChange={this.handleInputChange} />
        <input name="password" value={this.state.password} type="password" onChange={this.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 class MyUncontrolledForm extends React.Component {
  handleFormSubmit = (event) => {
    event.preventDefault()

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

  render() {
    return (
      <form onSubmit={this.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 allows both components and 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 **should not be overused**.

Refs

```
class MyForm extends React.Component {
  _inputRef = React.createRef()

  handlePrefill = () => {
    this._formRef.current.elements.username.value = 42
  }

  render() {
    return (
      <form ref={_formRef}>
        <input name="age" placeholder="Enter your age..." />
        <button type="button" onClick={this.handlePrefill}>
          Prefill
        </button>
      </form>
    )
  }
}
```

Refs

A *ref* can be created using the `createRef` function, exported by the `react` package.

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

The ref will contain a pointer to the React Element or the DOM Element on its `current` attribute as soon as the Element is rendered.

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 class MyList extends React.Component {
  render() {
    const names = this.props.names.map((name) => <li>{name}</li>)

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

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

Keys

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

```
export class MyList extends React.Component {
  render() {
    const names = this.props.names.map((name, index) => <li key={name + index}>{name}</li>)

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

Keys

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

```
export class MyListItem extends React.Component {
  render() {
    return <li>{this.props.name}</li>
  }
}

export class MyList extends React.Component {
  render() {
    return (
      <ul>
        {this.props.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

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

```
export class MyComponent extends React.Component {
  render() {
    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 class MyComponent extends React.Component {
  render() {
    const MyStyle = {
      backgroundColor: this.props.active ? 'yellow' : '#333',
      color: this.props.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 class MyComponent extends React.Component {
  render() {
    return (
      <div
        style={{
          backgroundColor: this.props.active ? 'yellow' : '#333',
          color: this.props.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 class MyComponent extends React.Component {
  render() {
    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 class Container extends React.Component {
  render() {
    return (
      <div className="container">
        <div className="container-title">{this.props.title}</div>
        <div className="container-content">{this.props.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 class MyComponent extends React.Component {
  render() {
    return (
      <Container title={<h1>My App</h1>}>
        <Welcome name="Jimmy" />
        <p>This is a paragraph.</p>
      </Container>
    )
  }
}
```

Higher Order Components

Higher Order Components (HOCs) are functions that take a component as an argument and return a new component. They are often used to enhance the functionality of a component.

HOCs emerged as a pattern to solve the problem of reusing the same logic for multiple components. Since *inheritance* is considered an *antipattern* in React, which favors *composition* over inheritance, a certain number of patterns have emerged over time to avoid inheritance.

Higher Order Components

Using HOCs is now considered an outdated practice, since more versatile techniques have emerged over time and React has evolved to a point where they are no longer needed.

You might, however, encounter HOCs in React projects, as they have been a very popular pattern for years.

Higher Order Components

Let's take a look at a component that implements a simple logic to track the mouse position:

```
export class MouseTracker extends React.Component {
  state = {
    position: [],
  }

  handleMouseMove = (event) => {
    this.setState({
      position: [event.clientX, event.clientY],
    })
  }

  render() {
    return (
      <div style={{ height: '100%', width: '100%' }} onMouseMove={this.handleMouseMove}>
        <p>The current position is: {this.state.position}</p>
      </div>
    )
  }
}
```

Higher Order Components

A HOC can be written to extract the mouse tracking logic and reuse it with any component that might need it, without having to resort to inheritance of copy-pasting.

Higher Order Components are **functions that take a component as an argument and return a new component**.

Higher Order Components

```
function WithMouseTracking(Component) {
  return class extends React.Component {
    state = {
      position: [],
    }

    handleMouseMove = (event) => {
      this.setState({
        position: [event.clientX, event.clientY],
      })
    }

    render() {
      return (
        <div onMouseMove={this.handleMouseMove}>
          <Component {...this.props} position={this.state.position} />
        </div>
      )
    }
  }
}
```

Higher Order Components

When we have a HOC, we can pass it any component and get a **new** component that we can then use in place of the original one, knowing that it will receive the props passed by the HOC:

```
class MousePositionViewer extends React.Component {
  render() {
    return (
      <div>
        <p>The current position is: {this.props.position}</p>
      </div>
    )
  }
}

export default WithMouseTracking(MousePositionViewer)
```

Higher Order Components

Higher Order Components can be *composed* to create more complex components, but they have some important limitations:

- They offer **static** composition: this means that the component must be enhanced with the HOC before it can be used;
- When reading the code of the enhanced component, it's not clear how the HOC is actually used or which props are passed to it by the HOC;
- Their usage is cumbersome and verbose.

Render Props

Another pattern that emerged in React to solve the problem of reusing the same logic for multiple components is the **render prop**.

A render prop is a prop that a component will receive to implement its *render logic*. This allows a component to implement reusable behavior without implementic a specific render logic, delegating it to its parent component.

Render props are a step forward from HOCs, and are still used quite often.

Render Props

When implementing a render prop, we can use the `render` prop to pass a function to the component.

This function will be called whenever the component is rendered, and will be able receive any argument that parent component will need to implement the presentation logic, in a way similar to how HOCs pass props to the components they enhance.

Render Props

```
export class MouseTracker extends React.Component {
  state = {
    position: [],
  }

  handleMouseMove = (event) => {
    this.setState({
      position: [event.clientX, event.clientY],
    })
  }

  render() {
    return <div onMouseMove={this.handleMouseMove}>{this.props.render(this.state.position)}</div>
  }
}
```

Render Props

When using a component that expects a render prop, we will pass it a function that will be called whenever the component is rendered, and will receive the arguments that the component is exposing to the parent component.

```
export class MyComponent {
  render() {
    return (
      <MouseTracker
        render={(position) => (
          <div>
            <p>The current position is: {position}</p>
          </div>
        )}
      />
    )
  }
}
```

Render Props

Render Props allow a component to expose its internal state or the result of internal logic to its parent component. This makes them a very powerful pattern to reuse logic in different components.

Moreover, render props allow for *dynamic* component composition, and are less opaque than render props.

As a downside, components implementing render props are a little more cumbersome to compose.

Render Props

An alternative to using the `render` prop is to use the `children` prop.

The `children` prop is automatically valorized by React with whatever content is passed as a child of the component. Just as with other props, this content can be *of any type*, even thogh it's usually a component tree.

Render Props

```
export class MyComponent {
  render() {
    return (
      <MouseTracker>
        {(position) => (
          <div>
            <p>The current position is: {position}</p>
          </div>
        )}
      </MouseTracker>
    )
  }
}
```

Render Props

By passing a function as a child to a component, we can use the `children` prop to call function implementing the render logic passed by the parent component.

```
export class MouseTracker extends React.Component {
  state = {
    position: [],
  }

  handleMouseMove = (event) => {
    this.setState({
      position: [event.clientX, event.clientY],
    })
  }

  render() {
    return <div onMouseMove={this.handleMouseMove}>{this.props.children(this.state.position)}</div>
  }
}
```

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 class Root extends React.Component {
  render() {
    state = {
      language: 'en',
    }

    handleLanguageChange = (language) => {
      this.setState({
        language,
      })
    }

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

Context

In the previous 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` value by using the `LanguageContext.Consumer`:

```
export class MyDeeplyNestedComponent extends React.Component {
  render() {
    return (
      <LanguageContext.Consumer>
        {(language) => (
          <div>
            <p>The current language is: {language}</p>
          </div>
        )}
      </LanguageContext.Consumer>
    )
  }
}
```

Context

The `Consumer` component exposes the value of the context through a *render prop*, through which the consuming component will be able access the value.

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.

Function Components

Another way of writing React components is by using *functions*.

A *function component* is entirely defined by a function, which receives props as its only parameter and returns a JSX element that determines what will be rendered by the component.

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

Just as class componens, function components must be **pure** with respect to their props.

Function Components

Function components are different from class components in a couple of ways:

- A function component does *not* have an instance, so it does not have access to `this`;
- They do not have access to lifecycle methods;
- They cannot keep an internal state;
- Not having an instance, they cannot be attached to a `ref`;

Function Components

When writing function components, the props paramenter can be destructured to access props directly:

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

This has the advantage of clearly marking which props the component is expecting (even to your editor), as well as allowing for default values to be set inline:

```
export function Welcome({ name = 'World' }) {
  return <h1>Hello {name}!</h1>
}
```

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 MyComponent() {
  const [count, setCount] = useState(0)

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

  return (
    <div>
      <p>Counter: {count}</p>
      <button onClick={handleUpdateCount}>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 handleUpdateCount() {
    setCount((c) => c + 1)
  }

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

By having an empty dependency array, `useEffect` can be used to *mimick* the `componentDidMount` and `componentWillUnmount` lifecycle methods: the side effect function will be called after the *component did mount*, while the cleanup function will be called when the *component will unmount*.

`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.

In a way, using a dependency array allows us to *mimick* the behavior of the `componentDidUpdate` lifecycle method, but with the added benefit that both the side effect and the cleanup are specific to a set of variabiles instead of the entire component.

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 }) {
  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)
    } catch (error) {
      setError(error)
    } finally {
      setLoading(false)
    }
  }

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

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

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`

```jsx
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.

`useRef`

The `useRef` hook allows us to create a *reference* to a value. The returned ref can be passed to a component's `ref` prop to get access to the DOM node (or to the component itself in case of class components).

`useRef` is also useful when we want to store a value that can be updated but we don't want to re-render the component.

The value of the ref can be accessed using the `current` property of the returned object.

useRef

```
export function Welcome() {
  const [name, setName] = useState('')
  const inputRef = useRef(null)

  function handleNameChange(event) {
    setName(event.target.value)
  }

  useEffect(() => {
    inputRef.current.focus()
  }, [])

  return (
    <div>
      <input type="text" value={name} onChange={handleNameChange} ref={inputRef} />
      {name && <h1>Welcome, {name}!</h1>}
    </div>
  )
}
```

`useContext`

Accessing the value of a context through the consumer can be done using the context's `Consumer` component, but using a render prop can be cumbersome.

An alternative is to use the `useContext` hook:

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

  return (
    <div>
      <h1>Current language: {language}</h1>
    </div>
  )
}
```

`useContext`

The `useContext` hook allows us to access the value of a context without using the Consumer directly.

It expects to receive the Context as its first argument, and will return the value currently stored in the context's Provider component.

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 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*:

```
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:

```
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.

Redux

When building large applications, it's often useful to store the state of the application in a single, central store.

Such central store is handled by a library that is *external* to React, and can exist with or without it. These libraries are called *state management libraries*.

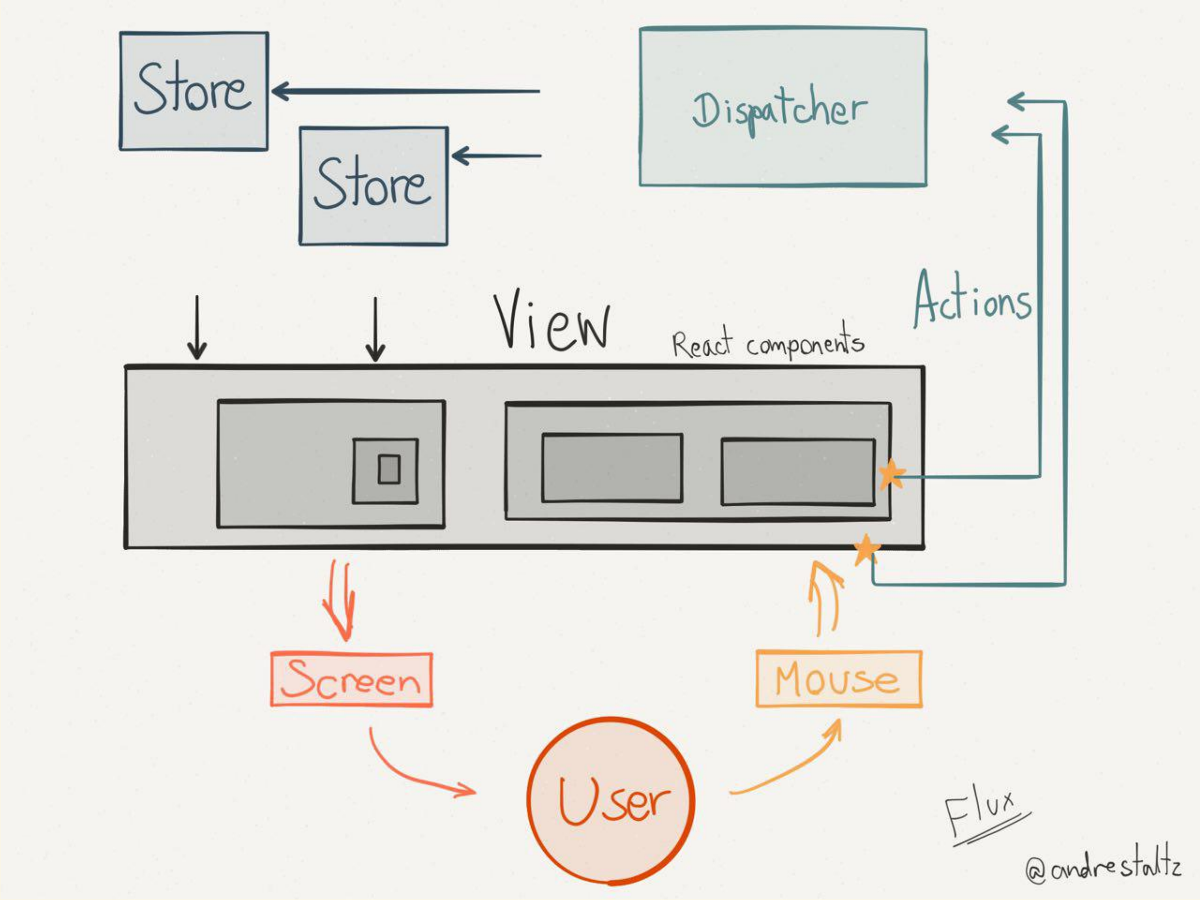
Using *state management libraries* provides an effective way to separate the handling of *data* from the *presentation* logic of an application.

Redux

One of the most popular state management libraries is **Redux**, which is a concrete implementation of a variation of the *Flux* architecture.

Flux is an architectural pattern that implements a *unidirectional data flow*.

Redux



Flow diagram (from André Staltz)

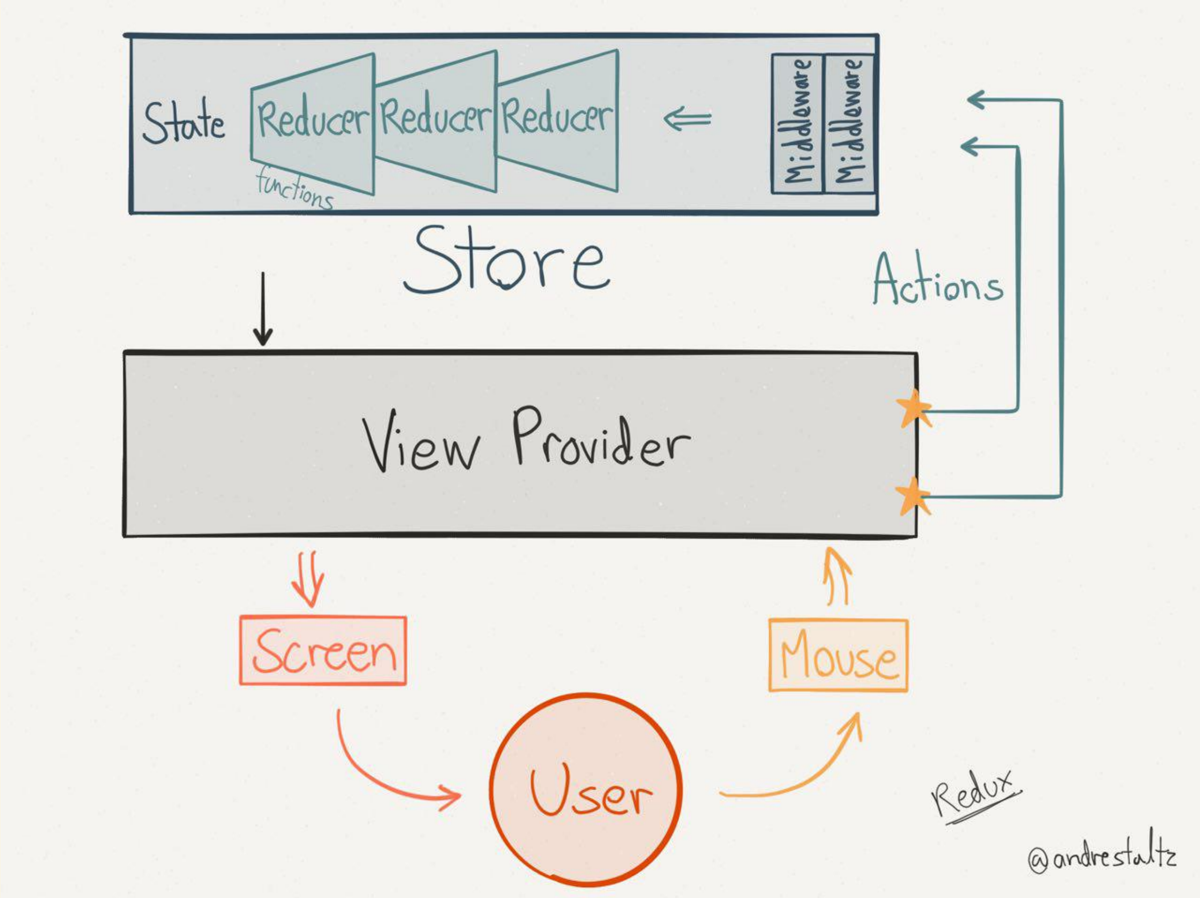
Redux

The *Flux* architecture consists of a number of *stores*, each responsible of a specific *data domain* of the application.

When the user interacts with the application, *actions* are passed to the stores through a *dispatcher*.

Each store will then update its state according to the action, and notify the application of the change so that the UI can be updated.

Redux



Redux diagram (from [André Staltz](#))

Redux

As a variant of *Flux* redux simplifies the architecture by using a *single store* that contains all the application state. Actions are dispatched directly to the Store, which will update its state according to the action by using *reducer functions*.

Before reaching *reducers*, all actions are fed through a number of *middlewars* which can let through, modify or even cancel the action if required.

All updates to the store are then propagated to the UI by using *subscriptions*.

Redux

There components of the Redux architecture are:

- The **Store**, the single source of truth for the application state;
- The **State**, the data that the store contains;
- **Actions**, messages that describe the changes that should be made to the state;
- **Reducers**, functions that take the current state and an action, and return the next state;
- **Middlewares**, functions that intercept actions and can modify, cancel or even replace them;

Redux

The are three *main principles* that define the Redux architecture.

- **Single source of truth**: the store contains the application state, and all components of the application can read it.
- **Immutability**: the state is read-only, and it only ever evolves to a new state returned by a reducer function through an action.
- **Purity**: reducer functions are *pure*, meaning that they have no side effects, and they always return the same output for the same input.

Redux

To use `redux` in our app, we first need to install it through `npm`:

```
$ npm install redux
```

Redux

In order to create a Redux *Store*, at least one reducer needs to be defined. A Reducer is a function that takes the current state and an action, and returns the next state.

When *modelling* a slice of state it can help to start reasoning about the starting value of that slice, and which actions can be used to *evolve* it, so that we can write a reducer accordingly.

Redux

Actions are defined as objects that have a *type* property. The *type* property is a *string* that describes the *action type*. Actions are creating using *action creators* utility functions:

```
export const INCREMENT = 'COUNTER@INCREMENT'
export const DECREMENT = 'COUNTER@DECREMENT'

export function incrementCounter(step = 1) {
  return {
    type: INCREMENT,
    payload: step,
  }
}

export function decrementCounter(step = 1) {
  return {
    type: DECREMENT,
    payload: step,
  }
}
```

Redux

```
function counterReducer(state = 0, action) {
  switch (action.type) {
    case INCREMENT:
      return state + action.payload
    case DECREMENT:
      return state - action.payload
    default:
      return state
  }
}
```

When a *Store* is initialized, each reducer is called with no state and a special action called `@@INIT`. Each reducer is expected to return the initial state it's responsible for.

Redux

After defining all actions and a reducer for at lease one state slice, the *Store* can be initialized:

```
const store = createStore(counterReducer)
```

After a store is created, actions can be dispatched to it by using the `dispatch` method:

```
store.dispatch(incrementCounter(10))
store.dispatch(decrementCounter(2))
store.dispatch(incrementCounter())
```

Redux

To read the state of the store, the `getState` method can be used:

```
const state = store.getState()
```

It's also possible to be notified every time the state changes by using the `subscribe` method:

```
store.subscribe(() => {
  const state = store.getState()

  console.log(state)
})
```

Redux

Since a Store will usually deal with multiple data domains, or *slices*, there will usually be multiple reducers. They can be combined using the `combineReducers` function:

```
const rootReducer = combineReducers({
  counter: counterReducer,
  users: usersReducer,
})

const store = createStore(rootReducer)
```

Redux

Writing reducers, action creators and action types for each slice of state can be cumbersome, and is viewed by many as one of the biggest pain points of using Redux.

A library called Redux Toolkit makes it easier to manage the Redux store by offering a certain number of utility functions.

```
$ npm install @reduxjs/toolkit
```

Redux

Redux Toolkit's `createSlice` function can be used to create a slice of state without having to manually define the reducer, action creators and action types:

```
export const counterState = createSlice({
  name: 'counter',
  initialState: 0,
  reducers: {
    increment: (state, action) => state + action.payload,
    decrement: (state, action) => state - action.payload,
    reset: (state, action) => 0,
  },
})
```

Redux

Action creators and reducer are created *automatically* by the `createSlice` function, which returns an object with the `reducer` and `actions` properties, that can be used to create the store and interact with the created slice:

```
const store = createStore(
  combineReducers({
    counter: counterState.reducer,
    users: usersState.reducer,
  }),
)

store.dispatch(counterState.actions.increment(10))
```

Redux

Reducers defined with the `createSlice` function use the [Immer] library to *emulate* state mutability.

This means that even though the state remains immutable it can be modified within the reducers as if it was mutable, as Immer wil handle the changes and return a new version of the state instead of changing the original one.

Reducers can therefore either modify the state directly or return a new one as regular reducers do.

Redux

```
export const usersSlice = createSlice({
  name: 'users',
  initialState: [],
  reducers: {
    addUser: (state, action) => {
      state.push(action.payload)
    },
    removeUser: (state, action) => state.filter((user) => user.id !== action.payload),
    editUser: (state, action) => {
      const { id, data } = action.payload
      const user = state.find((user) => user.id === id)

      for (const key in data) {
        user[key] = data[key]
      }
    },
    clearUsers: (state, action) => [],
    populateUsers: (state, action) => action.payload,
  },
})
```

Redux

Every time an actions is fed to the store through the `dispatch` method, it will first pass through all the *middlewares*.

A *middleware* is a function that takes an action and returns either the action itself, a new action or nothing at all. It can be used to modify, cancel or even replace the action.

Middlewares can be useful to handle logging, filter certain actions, delay their dispatch or handle them after the execution of a side effect.

Redux

A *middleware* has this signature:

```
const identityMiddleware = (store) => (next) => (action) => next(action)
```

For example, a *logging* middleware could be implemented as such:

```
const loggingMiddleware = (store) => (next) => (action) => {
  console.log('Action received:', action)
  console.log('Current state', store.getState())

  const result = next(action)

  console.log('New state', store.getState())

  return result
}
```

Redux

Once a middleware is defined, it can be added to the store using the `applyMiddleware` method:

```
const store = createStore(rootReducer, applyMiddleware(loggingMiddleware))
```

Middlewares are meant to be composed, so that more than one can be applied to a single store:

```
const store = createStore(rootReducer, applyMiddleware(loggingMiddleware, crashReporterMiddleware))
```

Redux

A common use for middlewares is to handle side effects, such as asynchronous operations.

The most common middleware to handle them is the redux-thunk middleware.

```
$ npm install redux-thunk
```

Redux

A **thunk** is a function that is returned by another function. The thunk receives the store methods `dispatch` and `getState` as parameters, and can trigger a chain of asynchronous operations, as well as dispatch other actions.

By dispatching a *thunk* instead of a regular action, we can trigger asynchronous operations that will eventually dispatch actions to evolve the store's state, instead of evolving it immediately.

Redux

In order to use the thunk middleware, we need to apply it to the store:

```
const store = createStore(rootReducer, applyMiddleware(thunkMiddleware))
```

Once applied, any call to dispatch that receives a *function* instead of an object will cause the function to be called with the store's `dispatch` method as the first argument, and the store's `getState` method as the second argument.

Redux

```
function fetchUser(username) {
  return async function (dispatch, getState) {
    try {
      const response = await fetch(`https://api.github.com/users/${username}`)
      const user = await response.json()

      const { users } = getState()
      if (users.find((u) => u.login === user.login)) {
        return dispatch(editUser(u.login, user))
      }

      return dispatch(addUser(user))
    } catch (error) {
      return dispatch(addError(error))
    }
  }
}

store.dispatch(fetchUser('gianmarcotoso'))
```

Redux

To debug our store's state evolution, we can use a powerful browser extension: **Redux DevTools**.

The Redux DevTools extension allows us to track each action dispatched to the store and see how the state evolves over time following each action.

It even allows us to rewind or fast-forward the state to any one of its values over the course of the application's lifetime!

Redux

In order to use the Redux DevTools, the extension must be installed from either the [Chrome Web Store]() or the [Mozilla add-ons]() page.

Once the extension has been installed, the store of our application must be *enhanced* to use the DevTools. This can be done by installing the `@redux-devtools/extension` package:

```
$ npm install @redux-devtools/extension
```

Redux

Once installed, the store can be enhanced by using the `composeWithDevTools` function exported by the `@redux-devtools/extension` package:

```
const store = createStore(rootReducer, composeWithDevTools(applyMiddleware(thunkMiddleware)))
```

This will make the store of our app show up when opening the extension, allowing us to easily track state evolution over time.

React Redux

Once a store has been created, it can be used in a React application through the `react-redux` library:

```
$ npm install react-redux
```

React Redux

The `react-redux` library exports a `Provider` component, that needs to be used to wrap the application's subtree that needs to be connected to the store:

```
export function Root() {
  return (
    <Provider store={store}>
      <App />
    </Provider>
  )
}
```

React Redux

Once the `App` component has been wrapped by the `Provider`, it can access the Redux store from within any component by using the `useSelector` hook:

```
export function Users() {
  const users = useSelector((state) => state.users)

  return (
    <div>
      {users.map((user) => (
        <div key={user.id}>
          <img src={user.avatar_url} alt={user.login} />
          <p>{user.login}</p>
        </div>
      ))}
    </div>
  )
}
```

React Redux

A *selector* is a function that receives the state of the store and returns a value. This value can either be a value directly extracted from the store's state, or a value computed by using the current store's state:

```
export function ActiveUsers() {
  const users = useSelector((state) => state.users.filter((user) => user.active))

  return (
    <div>
      {users.map((user) => (
        <div key={user.id}>
          <img src={user.avatar_url} alt={user.login} />
          <p>{user.login}</p>
        </div>
      ))}
    </div>
  )
}
```

React Redux

To avoid unnecessary re-renders, selectors can be memoized through the `createSelector` function exported by the `reselect` package, but also included in the `@reduxsjs/toolkit` package.

The advantage of using selectors created with `createSelector` instead of plain functions is that their output will be memoized, and therefore recalculated only when the input changes.

React Redux

```
const activeUsersSelector = createSelector(
  (state) => state.users,
  (users) => users.filter((user) => user.active),
)

export function ActiveUsers() {
  const users = useSelector(activeUsersSelector)

  return (
    <div>
      {users.map((user) => (
        <div key={user.id}>
          <img src={user.avatar_url} alt={user.login} />
          <p>{user.login}</p>
        </div>
      ))}
    </div>
  )
}
```

React Redux

A component can also dispatch actions by accessing the `dispatch` method via the `useDispatch` hook:

```
export function ReduxCounter() {
  const counter = useSelector((state) => state.counter)
  const dispatch = useDispatch()

  return (
    <div>
      <p>{counter}</p>
      <button onClick={() => dispatch(counterSlice.actions.increment())}>+</button>
      <button onClick={() => dispatch(counterSlice.actions.decrement())}>-</button>
    </div>
  )
}
```

Testing

Among the numerous libraries that allow us to test JavaScript applications, the most popular one in the React ecosystem is **Jest**.

Jest is already installed in all applications created with `create-react-app`, but can be installed in any project through `npm`.

Testing

`jest` is a testing library that contains both a **test runner** and an **assertion library**.

A *test runner* is an engine that runs tests and collects their results.

An *assertion library* is a set of functions that allow us to test our code by *asserting* that certain conditions are met when it's executed.

Jest allows us to test both regular JavaScript code and React components (by installing an additional library).

Testing

In order write tests and run them with `jest`, we can create files with either the `.test.js` or `.spec.js` extension, anywhere in the project, and they will be automatically picked up by Jest.

We can also create a `__tests__` folder within our `src` folder and populate it with our tests files.

Jest can be started by running `npm run test` in the project's root directory.

Testing

Let's consider the `sum` function:

```
function sum(...numbers) {
  return numbers.reduce((acc, number) => acc + number, 0)
}
```

In order to test this function we can write the `sum.spec.file`:

```
import { sum } from './sum'

describe('sum', () => {
  test('should return the sum of all numbers', () => {
    expect(sum(1, 2, 3)).toBe(6)
  })
})
```

Testing

Jest's assertion library uses a language that aims to be as close as possible to spoken language, to make tests easier to read.

- `describe` is a function that groups tests together. You should group together tests that *describe* a specific element of your code, such as a function, a component or a hook;
- `it` is a function that defines a test for a specific use case. You should test for both the *success* and the *failure* of your code;

Testing

The `expect` function is used to make an *assertion*, which is a statement that asserts that a certain condition is met.

The result returned from `expect` has a number of functions that allow us to check on the result of the operation, for example the `toBe` function, which asserts that the result of the operation is strictly equal to a given value.

A complete reference to all possible operations is available on Jest's documentation.

Testing

Since Redux reducers are pure functions, we can test them by calling them and asserting the result they return: when called with a specific value for the state and an action, a reducer will always return the same value.

```
describe('users.state', () => {
  it('should add a user', () => {
    const jimmy = { id: 1, name: 'Jimmy', age: 37 }

    const state = usersState.reducer([], usersState.actions.add(jimmy))
    expect(state).toEqual([jimmy])
  })
})
```

Testing

Thunks can also easily be tested in a number of ways. The easiest way of testing a thunk is to call it and assert that the dispatch function is eventually called with a specific action.

```
describe('users.state', () => {
  it('should fetch a user from github', async () => {
    const thunk = fetchUser('gianmarcotoso')
    const mockDispatch = jest.fn()

    await thunk(mockDispatch)

    expect(mockDispatch).toHaveBeenCalledWith(expect.objectContaining({ type: 'users/add' }))
  })
})
```

Testing

When testing, we can *mock* functions: we replace them with *fake* implementations that return a specific value or throw an error.

Mocking is an effective way to make a *unit* test, something that tests the behaviour of a single element of your code.

`jest.fn` is a function that returns a mock function, one that has no specific implementation, but that can be used to assert that it was called with specific arguments.

Testing

When dealing with network requests in unit tests, making requests to a remote server is not a good idea. We can instead create a *mock server* that returns a specific response, so that we can test our code without actually making requests to a remote server.

In order to create a mock server we can use the `msw` library, which we can install through `npm`:

```
$ npm install --save-dev msw
```

Testing

Once we have installed the library, we can create a fake server and make it available for our tests:

```
import { rest } from 'msw'
import { setupServer } from 'msw/node'

const handlers = [
  rest.get('https://api.github.com/users/:username', (req, res, ctx) => {
    return res(
      ctx.json({
        id: 1,
        login: 'gianmarcotoso',
        name: 'gianmarcotoso',
      }),
    )
  }),
]

export const server = setupServer(...handlers)
```

Testing

After creating the server, all we need to do is to start it in all the tests where we need it:

```
describe('users.state', () => {
  beforeAll(() => server.listen())
  afterAll(() => server.close())

  it('should download a user from github', async () => {
    const thunk = fetchUser('gianmarcotoso')
    const mockDispatch = jest.fn()

    await thunk(mockDispatch)

    expect(mockDispatch).toHaveBeenCalledWith(expect.objectContaining({ type: 'users/add' }))
  })
})
```

Testing

Using a mock server is very important, especially when dealing with third party APIs: most APIs have a rate limit, which is a limit on the number of requests we can make in a given time, after which we will be locked out and won't be able to make any more requests for some time.

Some APIs also have a *cost* for each request that is made, so making requests in tests (which are often continuously executed) could result in a very costly bill!

Testing

Having to remember to start and stop a mock server every time we test something that has network interactions is not very practical. We can setup our tests so that the server is started automatically for all of them, by using a `setupTests.js` file:

```
beforeAll(() => server.listen())
afterAll(() => server.close())
```

This file will be automatically executed by Jest before each test, as long as you create it within a `create-react-app` project.

Testing

Testing React components can be done by using React Testing Library, a library dedicated to testing React components.

This library is already installed when building apps with `create-react-app`, but can be added to any project with `npm`:

```
npm install --save-dev @testing-library/react
```

Testing

```
export function Sum({ numbers }) {
  return <h1>{sum(...numbers)}</h1>
}
```

```
import { render } from '@testing-library/react'
import '@testing-library/jest-dom/extend-expect'
import { Sum } from './Sum'

describe('<Sum />', () => {
  it('should render the sum of numbers', () => {
    const { container } = render(<Sum numbers={[1, 2, 3]} />)

    expect(container.firstChild).toHaveTextContent('Sum: 6')
  })
})
```

Since React components require a different set of assertions to be made, we need to import `@testing-library/jest-dom/extend-expect`.

Testing

In order to avoid having to import '@testing-library/jest-dom/extend-expect' in every test, we can import it in the `setupTests.js` file, since it will be automatically executed by Jest before each test.

Testing

Testing a component is done by first *rendering* the component by using the `render` function, exported by the `@testing-library/react` package.

The `render` function returns an object with a `container` property that contains the rendered component.

The `container` property is a DOM node that contains the rendered component, and that we can query to assert that is behaves as expected.

Testing

React Testing Library focuses on testing the DOM that is rendered from a React Component, and not the React component itself.

Since a component is a *presentational element*, it makes sense to test the way it looks on the screen, rather than its implementation details.

Testing

Even though the `render` function returns an object with a `container` property, we should use the `getBy*` functions, available on the `screen` object exported by the library, to assert that certain elements are present in the DOM.

We can, for example, use the `getByTestId` function to get a specific element in the DOM by its `data-testid` attribute.

Testing

```
export function Sum({ numbers }) {
  return <h1 data-testid="sum">Sum: {sum(...numbers)}</h1>
}
```

```
import { render, screen } from '@testing-library/react'
import '@testing-library/jest-dom/extend-expect'
import { Sum } from './Sum'

describe('<Sum />', () => {
  it('should render the sum of numbers', () => {
    render(<Sum numbers={[1, 2, 3]} />)
    const sum = screen.getByTestId('sum')

    expect(sum).toHaveTextContent('Sum: 6')
  })
})
```

Testing

We can also test the behavior of the component by using the `fireEvent` object, which allows us to simulate user interactions.

```
import { render, screen, fireEvent } from '@testing-library/react'
import '@testing-library/jest-dom/extend-expect'
import { TodoList } from './TodoList'

describe('<TodoList />', () => {
  it('should add a new todo', () => {
    render(<TodoList />)
    const input = screen.getByTestId('todo-add-input')
    const button = screen.getByText('Add')

    fireEvent.change(input, { target: { value: 'Learn React' } })
    fireEvent.click(button)

    expect(screen.getByText('Learn React')).toBeInTheDocument()
  })
})
```

Testing

When dealing with components that have asynchronous side effects, we can wait for a UI element to be displayed by using the `findBy*` functions:

```
import { render, screen } from '@testing-library/react'
import '@testing-library/jest-dom/extend-expect'
import { GithubUser } from './GithubUser'

describe('<GithubUser />', () => {
  it('renders the username', async () => {
    render(<GithubUser username="gianmarcotoso" />)
    const user = await screen.findByTestId('user-name')

    expect(user).toHaveTextContent('gianmarcotoso')
  })
})
```

Testing

Hooks can be tested by using the '@testing-library/react-hooks' package, which needs to be installed through `npm`:

```
$ npm install --save-dev @testing-library/react-hooks
```

Testing

This packages exposes some functions, among which are:

- `renderHook`: calls a hook and returns the result of its execution
- `act`: runs a function from a hook and returns the result of its execution

Testing

```
import { act, renderHook } from '@testing-library/react-hooks'

describe('useCounter', () => {
  it('should return the initial state', () => {
    const { result } = renderHook(() => useCounter(0))

    expect(result.current.counter).toBe(0)
  })

  it('should increment the counter', () => {
    const { result } = renderHook(() => useCounter(0))

    act(() => {
      result.current.onIncrement()
    })

    expect(result.current.counter).toBe(1)
  })
})
```

Testing

The `renderHook` function returns an object with a `result` property that contains the result of the hook execution, both current in the `current` property and previous in the `all` property.

In order to trigger an update, we can call hook functions within a callback passed to the `act` function, which will cause the result to be updated.

Testing

When dealing with hooks that handle asynchronous side effects, we can await the result of an operation by using the `waitForNextUpdate` function:

```
describe('useGithubUser', () => {
  it('should return the user', async () => {
    const { result, waitForNextUpdate } = renderHook(() => useGithubUser('gianmarcotoso'))

    await waitForNextUpdate()

    expect(result.current.user).toHaveProperty('login', 'gianmarcotoso')
  })
})
```

Testing

If the asynchronous operation takes more than an update to complete, we can use the `waitFor` function:

```
describe('useGithubUser', () => {
  it('should return the user', async () => {
    const { result, waitForNextUpdate } = renderHook(() => useGithubUser('gianmarcotoso'))

    await waitFor(() => !!result.current.user)

    expect(result.current.user).toHaveProperty('login', 'gianmarcotoso')
  })
})
```

The `waitFor` function receives a callback and returns a `Promise` that will resolve as soon as the callback returns either a truthy value or `undefined`.

Testing

Testing is a very important part of the development process, and it should be done as much as possible.

Testing components, hooks, reducers and functions will greatly help us detect bugs as soon as they are introduced.

We also need to remember that testing should always take into account the cases in which our code *fails* and not only those in which it *succeeds*.

The End

Thank you!