

React.js

Course Material • Hamburg Coding School • May 2023

Outline

Outline	1
Course Goal	4
Front-End Architectures	1
Server-Side Rendering (SSR)	1
Single Page Application (SPA)	1
Universal / Isomorphic Application	2
Recap: Modern JavaScript	3
Arrow Functions	3
Template literals	3
Array Destructuring	3
Object Destructuring	4
Object Spreading	4
Array Spreading	4
Array.map()	4
Array.filter()	4
Ternary Operator	5
React.js	5
React Components	6
Installing React	7
JSX	9
React.Fragment	10
Expressions	11
Props	12
Special Props: children	14
PropTypes (optional knowledge)	15
Composing Components	16
Conditional Rendering	16
Rendering Lists	17
Special property: key	18
Controlled Components	18
Local Component State	19

Handling UI Events	20
onClick	20
onChange	20
Lifecycle Hooks	21
Mounting	21
Updating	21
Unmounting	21
Optional Lifecycle Methods	21
Component Types	22
Function components	22
Class components	23
Styling in React	23
Inline Styles	23
CSS Classes	24
CSS Modules	25
React Developer Tools	26
Routing	27
Setting Up Router	27
Navigational Components with React Router	28
Route Params	29
Accessing Data from APIs	32
Load data with fetch()	32
Using React Effects	32
Local Storage	33
Conditional Rendering for Local Storage	35
Default value	35
Loading State	35
Get current location with the Geolocation API	36
Google Maps for React	37
Center a map to the user's location	38
Google Maps interactions	38
Handling marker clicks	38
Global State Management	39
Context	39
Flux and Redux	40
Storybook	41
Stories	41
Knobs Addon	42
Chakra UI	43

Glossary	45
Useful Links	47

Course Goal

The course goal is to be able to read, understand, modify and write idiomatic React components and apply this knowledge to code bases of all sizes.

Front-End Architectures

Single Page Applications (SPA)

Single page applications are the “default” mode of operation for modern Frontend frameworks like React, Vue or Angular. If you create a new project without any other meta-frameworks, it will be a Single Page Application and React will run on the client (the browser of website visitors).

The HTML webpage which is delivered from the server is almost empty (or with a loading screen) and the content is then rendered at the client side by the loaded Javascript bundle. The “single page” in the name of the concept refers to the single page that the user is requesting from the server. The initial page load will contain all of the application logic.

Compared to an application that is rendered on the server, it takes a little bit more time to get the application up and ready, before the user can read and interact with it.

Single Page Application

Web application, where most HTML is created by JavaScript running on the client side

Server-Side Rendering (SSR)

Server-side rendered pages are built on-the-fly on the server-side – potentially on every single request. The server compiles everything, includes the content and delivers a fully populated, mostly static HTML page to the client.

SSR pages are fast and effective, as long as the server is fast in compiling pages or even pages are cached by the webserver and therefore recompiling pages is not necessary on each request from the client.

This is how many traditional older web-based software systems work. Popular content management systems like WordPress, Drupal, Typo 3, etc. work like this. The most popular language for these kind of systems is PHP. But building server side pages with JavaScript is also getting more and more popular.

Server Side Rendering

Web application where the server hosts and delivers all HTML and CSS

Server-Side Generation (SSG)

Instead of rendering on every request, it is also possible to generate static pages only once during build time, so called **pre rendered pages**, and deliver those directly from the server.

SSR and pre-rendered pages can be well crawled by search engines.

Fortunately Google can also read Javascript and it works very well. But not all search engines are able to do so. If web pages also need to be indexed by other search engines, server side rendered or even pre rendered pages must be considered in.

Server Side Generation

Web application where the server generates and delivers all HTML and CSS only once – typically these are very static pages and don't allow for much interactivity.

💡 **Both, server-side rendered and server-side generated apps** are usually faster and smaller than Single Page Applications, because the browser is faster in parsing HTML than in executing JavaScript and building HTML with that.

Isomorphic / Universal Applications

Universal, or sometimes also called isomorphic applications combine SPA, SSR and SSG.

The universal application consists mainly of one initial request to the server which delivers an already compiled and populated HTML to the client. In other words the first request is typically server-side rendered.

After the initial request, the client-side JavaScript is loaded and executed by the browser and then “takes over” the duty of interactively modifying and updating the page, e.g. by fetching only the necessary data from the server. Everything else is done by the browser client side.

This has become very popular through Meta-Frameworks (that sit on top of React, Vue, Angular, etc.) like Next.js, Nuxt or Astro.

Isomorphic / Universal Apps

Offer a mixture of server-rendered apps and Single Page apps.

All architecture types have their tradeoffs. Developers make their decision depending on what they need for their use-case.

In this course, we are concentrating only on building a ***Single Page App***.

Recap: Modern JavaScript

In React applications, certain types of JavaScript features and syntaxes are very common or preferred. Namely, the following language features are very often used in React apps:

Arrow Functions

```
function sumOfApples(bucket1, bucket2) {  
  const sum = bucket1 + bucket2;  
  return sum;  
}
```

```
const sumOfApples = (bucket1, bucket2) => {  
  const sum = bucket1 + bucket2;  
  return sum;  
}  
  
// or with implicit return:  
const sumOfApples = (bucket1, bucket2) => bucket1 + bucket2;
```

Special case:

If we have only a single parameter, we can omit the brackets.

```
const log = value => console.log('Value:', value);
```

Template literals

```
const person = { name: 'Henning', age: 28, role: 'Engineer'}  
console.log(person.name + ' is ' + person.age + ' years old.')
```

```
const person = { name: 'Henning', age: 28, role: 'Engineer'}  
console.log(`${person.name} is ${person.age} years old.')
```

Array Destructuring

```
const values = ['Henning', 28, 'Engineer']  
const [name, age] = values
```


Object Destructuring

```
const person = { name: 'Henning', age: 28, role: 'Engineer' }  
const {name, age} = person
```

With default values:

```
const person = { age: 28, role: 'Engineer' }  
const { name = 'Henning', age = 28 } = person
```

Object Spreading

```
const person = { name: 'Henning', age: 28 }  
const person2 = { ...person }  
person2.name = 'Nina'  
  
console.log(`${person.name} is ${person.age} years old.`)  
// prints: Henning is 28 years old  
  
console.log(`${person2.name} is ${person2.age} years old.`)  
// prints: Nina is 28 years old
```

Array Spreading

```
const values1 = [1, 2, 3]  
const values2 = [4, 5, 6]  
const allValues = [...values1, ...values2]  
// [1, 2, 3, 4, 5, 6]
```

Array.map()

```
const numbers = [1, 2, 3, 4]  
const doubles = numbers.map(number => number * 2)  
// [2, 4, 6, 8]
```

Array.filter()

```
const numbers = [1, 2, 3, 4, 5, 6, 7, 8]  
const events = numbers.filter(number => number % 2 === 0)  
// [2, 4, 6, 8]
```

Ternary Operator

```
const number = 3
if (number > 5) {
  console.log('Larger than 5')
} else {
  console.log('Lesser or equal 5')
}
// prints: Lesser or equal 5
```

```
const number = 3
number > 5
  ? console.log('Larger than 5')
  : console.log('Lesser or equal 5')
// prints: Lesser or equal 5
```

React.js

"A JavaScript library for building user interfaces"
[<https://reactjs.org/>]

React is often referred to as a JavaScript **framework**, which is actually not accurate. It is rather a library which is very flexible. The difference is that libraries don't dictate rules about how to organize or structure your projects. They are more flexible.

React can easily feel like a framework through its deep integration with other libraries like Redux that impose specific architectural patterns. Redux is a JavaScript library for managing state information in a web application.

React can be used for UI projects of any size. We use React to build a Single Page Application (SPA).

React Components

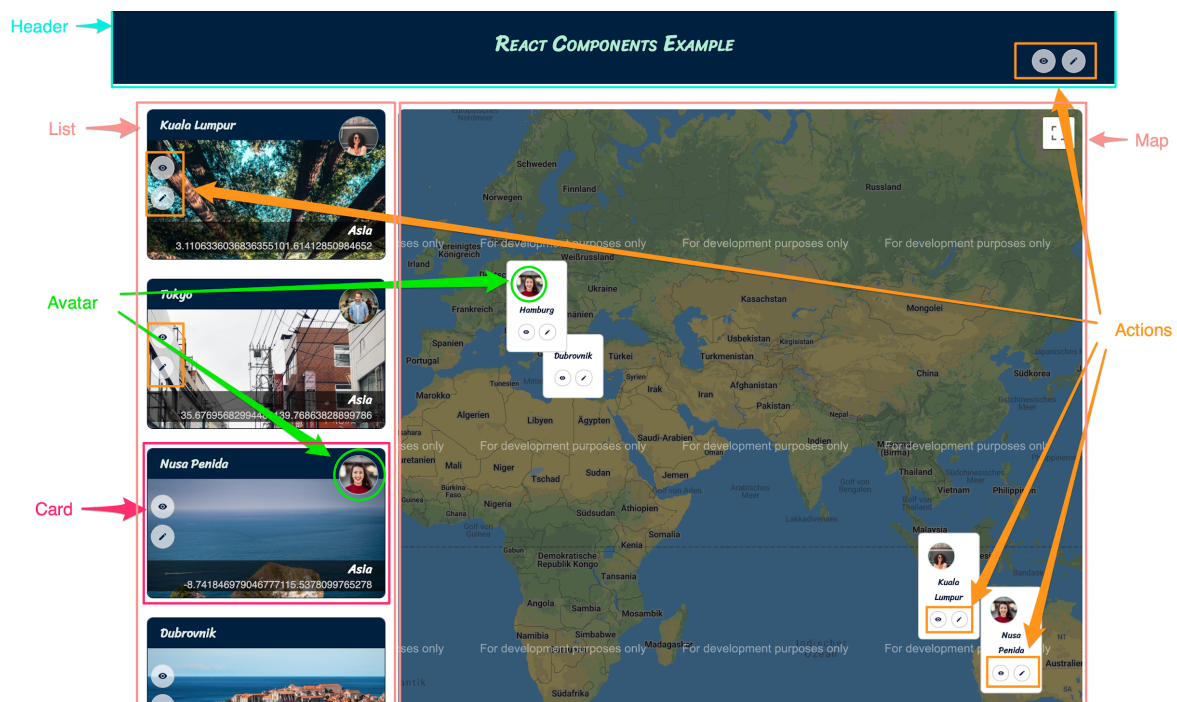
Developing an app with any modern UI framework/library like React (the same is true for Vue or Angular) means **thinking in components**: A React App is built by a bunch of smaller individual components. Components are the basic building blocks of any React application.

Every React application has at least one component: the root component, typically named App in App.jsx.

Often, if a web page gets more complex, you realize that there are a lot of elements that you can use in multiple places. Instead of writing all your code in one single file and not being able to reuse parts of it, a better approach is to extract elements and make them reusable. That is what a component in React is – nothing but an independent and reusable bit of code and one of the reasons why React (and other component-based libraries) became so popular.

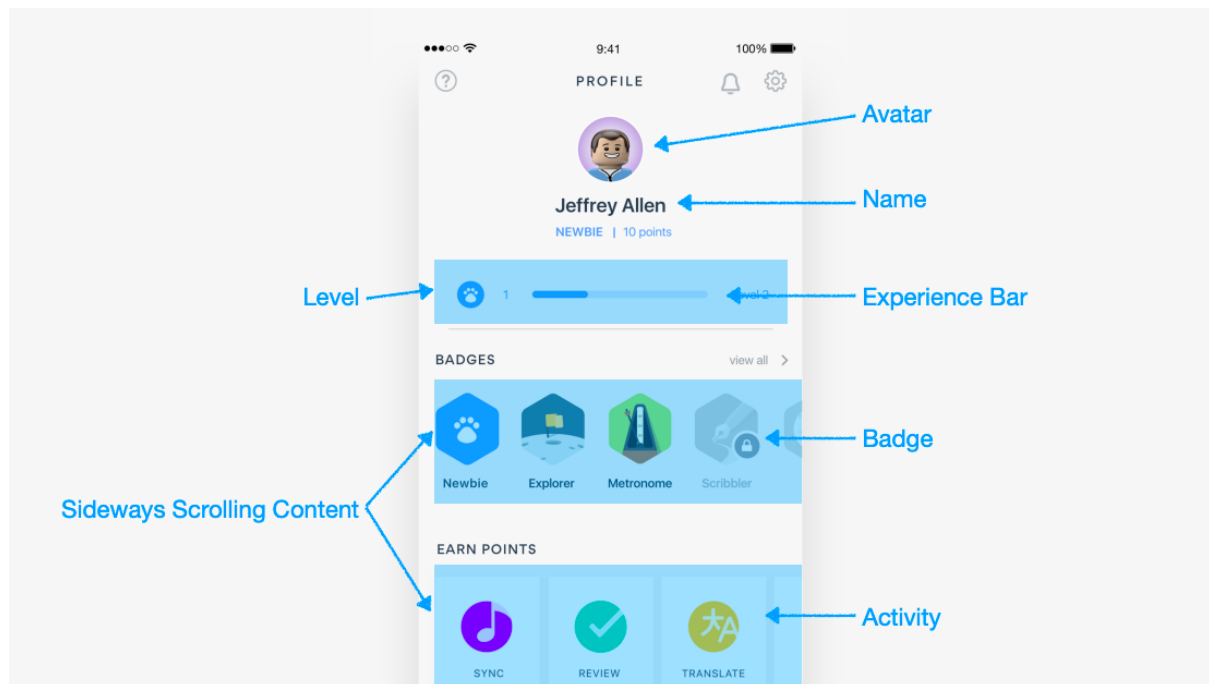
Taking the existing “Travel blog” into consideration, there are a lot of elements which could potentially be reused, e.g.:

- Avatar
- Actions
- Card
- Icon



Another Example: Users Profile Page:

For example, you might have a profile view with an avatar, name and profile information on the profile page, and you also want to show these elements on a dropdown menu.



It would be nice to have an option to have them as small components that we can reuse. We can do that with **React Components**.

Know when to make new components

- Is your code's functionality becoming unwieldy?
- Does it represent its own thing?
- Are you going to reuse your code?

When you answer one of these questions with yes, then you can extract code into a new component.

Installing and Scaffolding React

To install React, we can just use npm, and install it as a package. However, typically we need some other dependencies for a complete project. So instead of creating every project from scratch, we can use a scaffolding tool. `create-react-app` used to be a very popular tool for this task but it has become outdated, so now `vite` has become a faster and smaller alternative to scaffold a new React project.

Creating a new React app with vite is as simple as running this command:

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

Alternatively you can just use:

```
$ npm create vite@latest
```

Which will then show a little wizard/assistant and ask you for the name and the framework you want to use for this project. It is best practice to use lower case with hyphen (-) concatenation as a project name.

There are two essential files for the React Application to get up and running. These files must exist with these exact filenames:

- **/index.html** is the static page template that will be sent to the browser on the initial page load.
- **src/main.jsx** is the JavaScript entry point to execute/start React.

The other files can be renamed or deleted if wished.

Our Single Page Application (SPA) is a bare HTML Skeleton:

```
<html>
  ...
  <body>
    <div id="root"></div>
  </body>
</html>
```

React will attach to the root element, and build the whole application inside of this container:

```
ReactDOM.createRoot(document.getElementById('root')).render(<App />);
```

The **<App />** element in this code segment is not really HTML, it is rather a pseudo element that is used by React. It is a React component and the top level component for the React application.

To create a React component, we use a JavaScript function that returns a block of JSX:

```
function App() {  
  return (  
    <h1>Coding School Blog</h1>  
  )  
}
```

This will put `<h1>Coding School Blog</h1>` onto your screen.



Best Practice: It is common to put each component into its own file.

It is common to give these files a name with a `.jsx` file ending.
File endings with `.js` are common as well.

There is much debate about what is the better way. You can choose either file ending, but once you've chosen, stick to it and stay consistent with your file names.

JSX

You might wonder why it is possible to write HTML (look closely, these elements are NOT a string) inside of a Javascript file. The answer is called **JSX (JavaScript Syntax Extension)**. JSX is a lightweight syntax extension for JavaScript that makes it easier to write markup inside of JavaScript code.

Facebook released JSX to provide a concise syntax for creating complex DOM trees with attributes. They hoped to make React more readable like HTML and XML.

What JSX mainly does, is converting the angle brackets into specific JavaScript function calls.

```
// This JSX block is converted into...  
  
return <p>Coding School Blog</p>  
  
// this raw/vanilla JS function call  
React.createElement("p", null, "Coding School Blog")
```

Although the usage of JSX is optional, it has become the standard template solution in React and can be found in virtually every React project.

It is not really HTML, but an extension for JS that creates HTML elements for us.

```
function App() {  
  return (  
    <section id="blog-section">  
      <h1>Heading</h1>  
      <p>Lorem Ipsum ...</p>  
    </section>  
  )  
}
```

If you would write the above example without JSX, it would look like this:

```
function BlogSection() {  
  return React.createElement(  
    'Section',  
    { id: 'blog-section' },  
    React.createElement('h1', null, 'Heading'),  
    React.createElement('p', null, 'Lorem Ipsum ...')  
  )  
}
```

React.Fragment

For React to be able to do its work, every component must only return one single top-level JSX Element – so you cannot have multiple elements next to each other (siblings). If you want to create a common root element for multiple sibling elements, but you don't want to pick any tag like `div`, then you can choose a `React.Fragment`. It has not semantic meaning and will also not be visible in the DOM.

`<React.Fragment></React.Fragment>` or short `<></>`.

Expressions

We can also use JavaScript **expressions** in JSX.

With expressions, we can use curly brackets { } to include data in our pseudo HTML.

```
function IntroBox() {  
  const person = { name: 'Susi', age: 30}  
  return (  
    <section>  
      <p>Name: {person.name}</p>  
      <p>Age: {person.age}</p>  
      <p>Favorite Number: {23 + 42}</p>  
    </section>  
  )  
}
```


Props

You can define **props** (properties) that you can pass information to child components. Data which is received through props in the child component could not be updated. These data become read-only and immutable.

```
function Header(props) {  
  return <h1>{props.title}</h1>  
}
```

What is passed as props does not necessarily have to be text. It can be an object, any variable, or another component.

```
function Header(props) {  
  return <h1>{props.title}</h1>  
}
```

The props passed as a parameter to the function, could be destructured early in the function signature into variables. Which is common in modern JavaScript. So the above example could be also written like shown in the following example:

```
function Header({title}) {  
  return <h1>{title}</h1>  
}
```

This becomes especially useful if you want to use default values:

```
function Header({ title = 'Header' }) {  
  return <h1>{title}</h1>  
}
```

You can then use the component in other components and pass values for them. This looks even more like HTML:

```
function Header({ title }) {  
  return <h1>{title}</h1>  
}
```

```
function Text({ content }) {  
  return <p>{content}</p>  
}  
  
function BlogSection() {  
  const content = 'Lorem Ipsum ...'  
  return (  
    <section className="blog-section">  
      <Header title="My Blog Post" />  
      <Text content={content} />  
    </section>  
  )  
}
```

In this example, you can see that the value is passed as an attribute to the component, like in this line:

```
const content = 'Lorem Ipsum ...'  
  
<Text contentProp={content} /> // passes the variable to the Text component
```

In the Text component, the value is used like this:

```
function Text({ content }) {  
  return <p>{content}</p>  
}
```

In the end, this will result in the following HTML content:

```
<p>Lorem Ipsum ...</p>
```

This is how you pass variables as props to a component.

Special Props: children

In React, **children** is a special keyword. You cannot, for example, call a variable children. The children keyword is used for objects that represent DOM nodes.

Children of a node are the HTML elements that are nested inside the node.

DOM - Document Object Model The tree of all tags in an HTML document

Node A JavaScript object for a HTML tag with all necessary information (e.g. attributes)

node.children All nested HTML tags of a node

Now, remember that we passed variables as props to a component as an attribute like this:

```
<Text contentProp="Lorem Ipsum..." />
```

What would happen if we instead pass other HTML elements like this?

```
<Text><b>Lorem Ipsum...</b></Text>
```

Per default, React would just ignore that. Text would be empty.

But we can use it, if we use the keyword **children** in the component definition.

```
function Text({ children }) {
  return <p>{children}</p>
}
```

This would then return the following HTML:

```
<p><b>Lorem Ipsum ...</b></p>
```

Now we can use this to nest more complex HTML structures into our component.

```
function Text({ children }) {
  return <p>{children}</p>
}

function BlogSection() {
  return (
    <section>
      <h1>Title</h1>
      <Text><b><i>Lorem</i> Ipsum...</b></Text>
    </section>
  )
}
```

PropTypes (optional knowledge)

React has some built-in type checking abilities. PropTypes become handy when the app grows. With it, you can make your props more strict. This means you can define types for your props as some sort of constraints. If a value or object is passed to the component that does not fulfil these constraints, React is throwing a warning in the console. It avoids a lot of bugs which happen when properties are not correctly passed or expected but not given. For performance reasons, this only happens during development mode.

You can, for example, define that your title and subtitle take only strings and that the title is required (cannot be empty).

```
function Header({ title, subtitle }) {
  return (
    <header>
      <h1>{title}</h1>
      {subtitle !== null ? <p>subtitle</p> : null}
    </header>
  )
}
```

```

    )
  }

  Header.propTypes = {
    title: PropTypes.string.isRequired,
    subtitle: PropTypes.string
  }

```

Composing Components

You can compose multiple components together, and nest them into another.

We can create any tag for any component, e.g. **<BlogSection>**. For this, we need to create a function that has the same name. This is the function that creates our component.

```

function Header() {
  return <h1>Header</h1>
}

function Text() {
  return <p>Lorem Ipsum...</p>
}

function BlogSection() {
  return (
    <section className="blog-section">
      <Header />
      <Text />
    </section>
  )
}

```



Best Practice: It is best practice to create rather strikingly more different components than to put a lot of logic into it that is hard to understand.

Conditional Rendering

In React components, we can use logic to render different content for different conditions. For example:

```

function BlogSection() {
  if (isLoading) {

```

```

    return <p>Loading...</p>
  } else if (isError) {
    return <p>An Error occurred :(</p>
  } else {
    return <p>{blogContent}</p>
  }
}

```

For conditional rendering, the ternary operator becomes useful. In this example, we show either the blog content, or an error:

```

function BlogSection() {
  return (
    <section>
      <h1>{title}</h1>
      {hasError
        ? <p>An error occurred</p>
        : <p>{blogContent}</p>}
      {loadingComments && <p>Loading Comments...</p>}
    </section>
  )
}

```

The second line means that it checks if loadingComments is true, and if yes it adds **<p>Loading Comments...</p>**:

```

{loadingComments && <p>Loading Comments...</p>}

```

Rendering Lists

React has a way to iterate through arrays and define what is displayed for each item.

For this we use **Array.map()**:

```

function NumberList() {
  const numbers = [1, 2, 3, 4, 5]
  return (
    <ol>
      {numbers.map(number => (
        <li>{number}</li>
      ))}
    </ol>
  )
}

```

This would result in:

```
<ol>
  <li>1</li>
  <li>2</li>
  <li>3</li>
  <li>4</li>
  <li>5</li>
</ol>
```

Special property: key

If we display multiple elements that look the same, we want to give it an ID. That way we can distinguish them from each other.

This is useful if we want to manipulate or delete an element. If it has a key, React does not have to iterate through all to find it, but it can just access it directly.

```
function NumberList() {
  const numbers = [1, 2, 3, 4, 5]
  return (
    <ol>
      {numbers.map(number => (
        <li key={number}>{number}</li>
      ))}
    </ol>
  )
}
```

This is the recommended practice. React even shows warning in the dev tools if you don't set the key property in an iteration.

Controlled Components

Using form fields like checkboxes in a component are by default uncontrolled, which means they are not controlled by the component. They have the default HTML behavior and are handled by the DOM. As long as e.g the checkbox is uncontrolled it can be toggled.

```
<input type="checkbox" />
```

To take over the control of form fields by React, the input data has to be handled by the component and not the browser DOM. This is done by the components state (more about that soon in the next section).

```
<input type="checkbox" checked={this.props.checked} />
```

With this the checkbox state only changes, when the components property *checked* changes. Toggling is not anymore controlled by the internal state of the checkbox. Therefore, a click on the checkbox would not have any effect.

To achieve the toggling of the checkbox a click event handler is needed to trigger the change but also the state for *checked* has to be changed. To achieve this it is needed to understand the inner state of components and how it is being changed with React.

Local Component State

Imagine we need to manage a blog entry and to save whether it is published or not. A button is used to toggle the published state. If we want to implement it, we face a very common problem: we need to save the value of the publish state somewhere.

It is good practice to not do that with variables in a global state, but instead inside of a component. Because in the component, we don't want to rely on data coming from outside, but we want to develop and keep the data inside.

This is called **state**: The internal private data of a component, which is fully controlled by the component. It is an object with a set of observable properties that control the behavior of the component.

The state keeps some information that may change over the lifetime of the component. As a member of the component it helps keeping track of the current value. Whenever the state changes the UI is rerendered, to reflect those changes.

In React, the state is not modified directly. To use and modify state properties in function components, React helps us with something called a "hook": the function **useState()**. From that we get the value, and a function to set a new value.

```
import React, { useState } from "react"

function BlogEdit() {
  // initialize state with default value false
  const [published, setPublished] = useState(false)
```



```
// define function for changing the published state
function togglePublished() {
  setPublished(!published)
}

return (
  <div>
    <p>Published: {published}</p>
    <button onClick={togglePublished}>Toggle published</button>
  </div>
)
}
```

It is important to use the observable properties so that React can watch after changes of these. With an update in the state React figures out what part of the state is changed and updates only that part in the real DOM.

Handling UI Events

onClick

In JSX components you can define onClick events like this:

```
function Clicker() {
  function handleClick() {
    console.log('clicked')
  }
  return (<button onClick={handleClick}>Add 1</button>)
}
```

onChange

Similarly, you can listen to change events for input fields. Every time the user types something in the input field (the text in there changes) this event is triggered.

```
function Input() {
  const handleChange = event => {
    console.log('changed')
  }
  return (
    <div>
      <input onChange={handleChange}/>
    </div>
  )
}
```

Lifecycle Hooks

Every component goes through a series of events or phases during its life.

In React there are mainly three phases components go through:

- Mounting
- Updating
- Unmounting

React provides each of these lifecycle methods, that helps to monitor and manipulate what happens within the component. One of the methods is the `render()` method.

It gets called by React when mounting or updating a component. It is the only lifecycle method it's needed to render the DOM for each component.

Mounting

The Mounting phase is mainly the phase where the component is created and inserted into the DOM. In this phase, the component is birthed.

Updating

After the component is mounted it might be needed to be updated. The component gets updated when the state or props changes, hence trigger re-rendering. All of that happens in this phase.

Unmounting

This phase ends the lifecycle of a component.

In this phase the component is removed from the DOM.

Optional Lifecycle Methods

React has several optional lifecycle methods.

- `componentDidMount()` – called after the component is rendered.
- `componentDidUpdate()` – called after the component is updated.
- `componentWillUnmount()` – called when the component is unmounted

Component Types

React comes with two types of components, function components and class components.

On the internet, there are a lot of examples of both. The differences are explained shortly below.

Function components

Function components, as the name says are created by writing a function. It returns a React Element that would be rendered into the screen.

Function components are the status quo of writing modern React applications.

We will only use function components in the course.

```
function App() {  
  return (  
    <p>Hello World</p>  
  )  
}
```

In former React versions function components were also called functional stateless components because they were not able to maintain a state. Think of the state as the data you can store to a specific component.

At the end of 2018 React.js invented some functions (also called hooks in React), one of these: the `useState()` allows to manage the state even in a function component. They were released at the beginning of 2019 (Version 16.8).

Class components

The class components (also called stateful components) use the ES6 class syntax. It provides the ability to handle the internal state and to have lifecycle methods. As explained before, since 2019 it is possible to do this also in function components with React hooks.

```
class App extends React.Component {  
  render() {  
    return <p>Hello World</p>  
  }  
}
```

Styling in React

Inline Styles


You can add inline styles to your component definition like this:

```
<p style={{  
  fontSize: '24px',  
  color: 'red',  
  border: '1px dotted blue',  
  textAlign: 'center'  
}}  
>  
  Lorem  
</p>
```

In HTML a string for the style attribute is used to apply all stylings to the element. In React a JavaScript object is assigned. That is why the value contains two curly braces. The inner ones for the inline styling in the form of a JavaScript object, although the outer curly brace is for the valid JavaScript expression in JSX.

As you might remember, Inline styles have a very high specificity. You use them rarely, mostly if you want to override existing styles.

Also, notice that the style keys are in camelCase. This is also true for all attributes, event handlers in React with few exceptions like `aria-*` and `data-*` attributes which should be in lowercase.

 **Remember:** In inline styles, pseudo elements (like `:hover`) cannot be used.

It is also possible to define the styling in a variable and to assign it to the style where it should be applied.

```
const paragraphStyle = {
  fontSize: '24px',
  color: 'red',
  border: '1px dotted blue',
  textAlign: 'center'
};
```

```
<p style={paragraphStyle}>
  Lorem
</p>
```

CSS Classes

A better way to define styles for your component is to use class names and a css file. This is very similar to what we normally do with HTML and CSS.

You define classes in your CSS as you would normally do.

```
// Component.css
.paragraph {
  font-size: 24px;
  color: red;
  border: 1px dotted blue;
  text-align: center;
}
```

The CSS has to be imported by the components which want to use the class selectors defined within the CSS file.

```
// Component.jsx
import React from 'react';
import './Component.css';
```

Now, in your component, you would apply your style like this:

```
// Component.jsx
<p className="paragraph">Lorem</p>
```

Note that we use **className** instead of **class**, because **class** is a reserved word in JavaScript.

When React creates the HTML files from our component, it creates HTML tags with normal classes from it. This would result in the following:

```
// index.html
<p class="paragraph">Lorem</p>
```

CSS Modules

A CSS Module is a CSS file in which all class names and animation names are scoped locally by default. Used in a React component mean, that styles from a CSS Module can only be applied there. Other components are not influenced by module styling included within another component.

With `create-react-app` it is possible to use CSS Modules. The file names for CSS Modules have to follow the naming convention `[name].module.css`.

A valid filename for a CSS Module is for example:

`Component.module.css`

You can then import it into your JavaScript file where you define your component and work with it there.

```
// Component.module.css
.paragraph {
  color: red;
  border: 1px dotted blue;
  text-align: center;
}
```

```
// Component.jsx
<p className="{styles.paragraph}">Lorem</p>
```

When React generates the HTML and CSS files, it then uses some cryptic postfixes (endings) for the style classes that make sense only to the machine (not humans).

While this makes it harder to read in the inspector, it has a big advantage: the styles are only valid for the component and don't clash with other styles. This keeps the CSS much more clean.

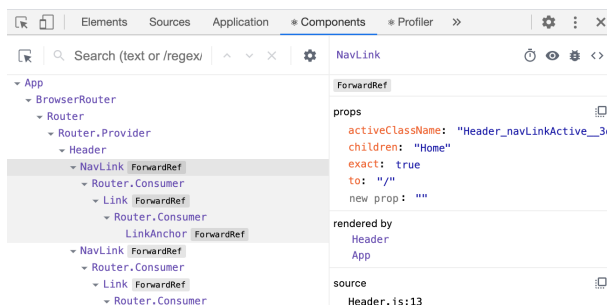
React Developer Tools

React DevTools is available as a built-in extension for Chrome, Firefox and Edge browsers.

It adds two additional tabs in the DevTools:

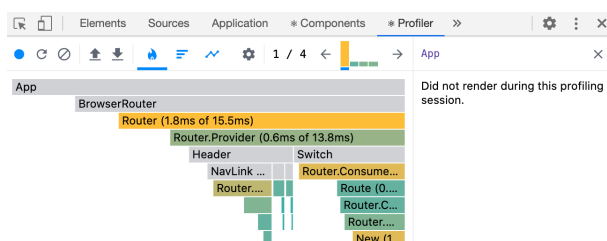
- **Components**

It shows you all components that's rendered in your React App.



- **Profiler**

It collects timing information about each component that's rendered in order to identify performance bottlenecks in React applications.



Routing

With React, we build Single Page Applications (SPA).

This means that everything will be rendered into a single page, and all navigation is happening in the JavaScript on the client side.

The **Router** supports the user to navigate between the pages. In React it is based on components.

Setting Up Router

Initially, the router is not included in the Create React App. Since there are several router libraries to choose from and we don't always need them, Create React App leaves it free to use if and which ones we want to use.

However, the React Router is one of the most popular libraries in this field. To install it, in Node JS we can use the following command:

```
npm install react-router-dom
```


Navigational Components with React Router

In our App component, we create a Router like this:

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom'
// ...

function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link>
          <Link to="/about/">About</Link>
        </nav>
        <Route path="/" exact component={Home} />
        <Route path="/about/" component={About} />
      </div>
    </Router>
  )
}
```

The main application routing is wrapped by the **< Router />** component.

The example above creates two so-called **routes**: **/** and **/about/**

/ will show the component **Home**

/about/ will show the component **About**

You can then use it in the address field of your browser:

https://<hostname>/ will show the component **Home**

https://<hostname>/about/ will show the component **About**

This looks like directories, and it works similarly. It creates “virtual directories” that display certain pages, i.e. components.

The **<Router />** can include many nested routes that render inclusively. It means, whenever a route's path matches the URL path, the router will render the route's component. The **exact** attribute in the **<Route />** ensures that the path must match exactly. If the attribute wouldn't be used here, the router would always show at least the **Home** component, because all paths including the **/** are matching this condition.

You can use **<Switch />**, to achieve that once the first route that matches the path, the Router is not looking for any other matches.

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom'
// ...

function App() {
  return (
    <Router>
      <div>
        <nav>
          <Link to="/">Home</Link>
          <Link to="/about/">About</Link>
        </nav>

        <Switch>
          <Route path="/" exact component={Home} />
          <Route path="/about/" component={About} />
        </Switch>
      </div>
    </Router>
  )
}
```

Route Params

It is possible to include a dynamic part in a route. In this way, the same component can be rendered but contain different contents depending on the dynamic part of the URL. To do this, you add a colon in the path with a parameter name. For instance, you can provide an id to a detail page by adding **:id** to your path.

```
// ...
<Route path="/detail/:id" component={Detail} />
```

In the **<Detail />** component the property **id** is defined in the **match** property provided by the **props**.

```
// Detail.js
import React from 'react';
export default function Detail({match}){
  return (
    <p>{match.params.id}</p>
  )
}
```

Or you can use the Hook **useParams()** provided by the react-router-dom, which returns you all router params for the matching route.

```
// Detail.js

import React from 'react';
import { useParams } from 'react-router-dom';

export default function Detail(){
  const {id} = useParams();

  return (
    <p>{id}</p>
  )
}
```

Often you have to provide additional data to the component of a specific route. For instance, you have a list of amazing people. The overview contains a list of all persons' names. For more information, a detail page should include all the details of a person.

The information (person object) is delivered from the parents component. It is not reasonable to provide all information within the URL. For example, a person should be identified by her nickname. To achieve this, you can define a URL parameter within your route with **:nickname**.


To provide all needed information about one person, we use the **render** attribute in the **<Route />** which gives us the possibility to find the person's data within the **amazingPeopleList** and to pass that to the Detail component.

```
// ...

const getPerson = (nickname) => {
  return amazingPeopleList.find(person => person.nickname === nickname)
}

// ...

<Route path="/detail/:nickname" render={ routeProps => (
  <Detail person={getPerson(routeProps.match.params.nickname)} />
)}>
</Route>
```

 **Documentation:** For more information and a detailed guide, refer to <https://reacttraining.com/react-router/web/guides/quick-start>.

Accessing Data from APIs

Load data with `fetch()`

To get data from a server, we want to interact with its API. For this, we want a way to do this asynchronously, because we need to wait for the response.

This is the modern way of making HTTP requests: **`fetch()`**.

```
fetch("https://swapi.co/api/people")
  .then(response => response.json())
  .then(data => {
    // Do something with the loaded data
  })
  .catch(error => {
    // Handle the error
  })
```

The **`fetch()`** is one of the functions, provided globally, by the Fetch API. The Fetch API provides an interface for fetching resources (including across the network).

The **`fetch()`** function takes at least one argument, the path of the resource you want to fetch. You can pass a second argument, the init options object. With that one, you can define further things. For example, you can set up the HTTP method (e.g. GET or POST, ...).

When the **`fetch()`** promise is resolved, the response instance is returned.

Example APIs:

Poké API: <https://pokeapi.co/>

Chuck Norris Jokes API: <https://api.chucknorris.io/>

Beers: <https://api.punkapi.com/v2/beers>

Using React Effects

The **`useEffect()`** is used to perform side effects in function components. Examples of side effects are data fetching, setting up a subscription, and manually changing the DOM in React components. You can think of a React Hook as **`componentDidMount()`**, **`componentDidUpdate()`** and **`componentWillUnmount()`** combined.

By using **useEffect()** hook React is informed that something has to be run after the component has been rendered. The function which is passed to **useEffect()** (also called “effect”) is remembered by React and gets called every time after the DOM updates were performed.

The effect is usually used for fetching data from an API as soon as the component is ready.

```
const [persons, setPersons] = useState([])
useEffect(() => {
  fetch('https://swapi.dev
/api/people/')
    .then(response => response.json())
    .then(data => {
      setPersons(data.results)
    })
}, [])
```

As an optional second parameter, we can pass an array containing variables that the effect should observe. Only if these variables change, the function will be executed. If we don't pass a second parameter, the function will always be called with every render.

In the example above as the second parameter an empty array is passed to `useEffect()`. This tells React that your effect doesn't depend on any values from props or state, so it never needs to re-run.

Local Storage

When we load data from an API, we need to keep it somewhere so that we don't lose it when we route to another page.

For that, we can use **localStorage**.

This is a JavaScript feature, that we can access via the window element:

[window.localStorage](#)

It can be used for temporarily storing data, like a cache mechanism.

It is a synchronous **key-value** storage.

```
localStorage.setItem('key', value);
```


These operations are possible:

```
// Save data
localStorage.setItem('myData', data);

// Get saved data
localStorage.getItem('myData');

// Remove data item
localStorage.removeItem('myData');

// Remove all data
localStorage.clear();
```

 **Note:** localStorage can only save **Strings** as values!

For more complex data objects, we can save it as JSON.

```
localStorage.setItem('person', JSON.stringify(data))
```

For example:

```
const data = [{ id: 1, name: 'Sascha' }, { id: 2, name: 'Tina' }]

// Create JSON string from data and save it
localStorage.setItem('data', JSON.stringify(data))

// Get data string from local storage and create data object from it
JSON.parse(localStorage.getItem('data'))
```



You can see and manipulate the local storage in the Chrome Dev Tools at
Application > Storage

Conditional Rendering for Local Storage

If we do this for the first time, the local storage will still be empty.

```
const localPerson = JSON.parse(localStorage.getItem('people'))
// localPerson is null

const [person, setPerson] = useState(localPerson)
```

We only show a heading if we have data in the local storage:

```
{ person && <h1>{ person.name }</h1> }
```

Default value

We can make sure that we have at least some default data with the `||` operator:

```
JSON.parse(localStorage.getItem('person')) || '{}'
```

Loading State

We can show a loading indicator if the data is loading, with:

```
{ isLoading === true && <p>Loading...</p> }
```

For this, we need an extra state, like in this example:

```
// ...
const [isLoading, setIsLoading] = useState(false)

function handleClick() {
  setIsLoading(true)

  fetch('https://swapi.co/API/people')
    .then(res => res.json())
    .then(data => {
      setPerson(data)
      const stringData = JSON.stringify(data)
      localStorage.setItem('person', stringData)
      setIsLoading(false)
    })
}
```



```

return (
  <div>
    { person && <h1>{ person.name }</h1> }
    { loading === true && <p>Loading...</p> }
    <button onClick={handleClick}>Load random person</button>
  </div>
)

```

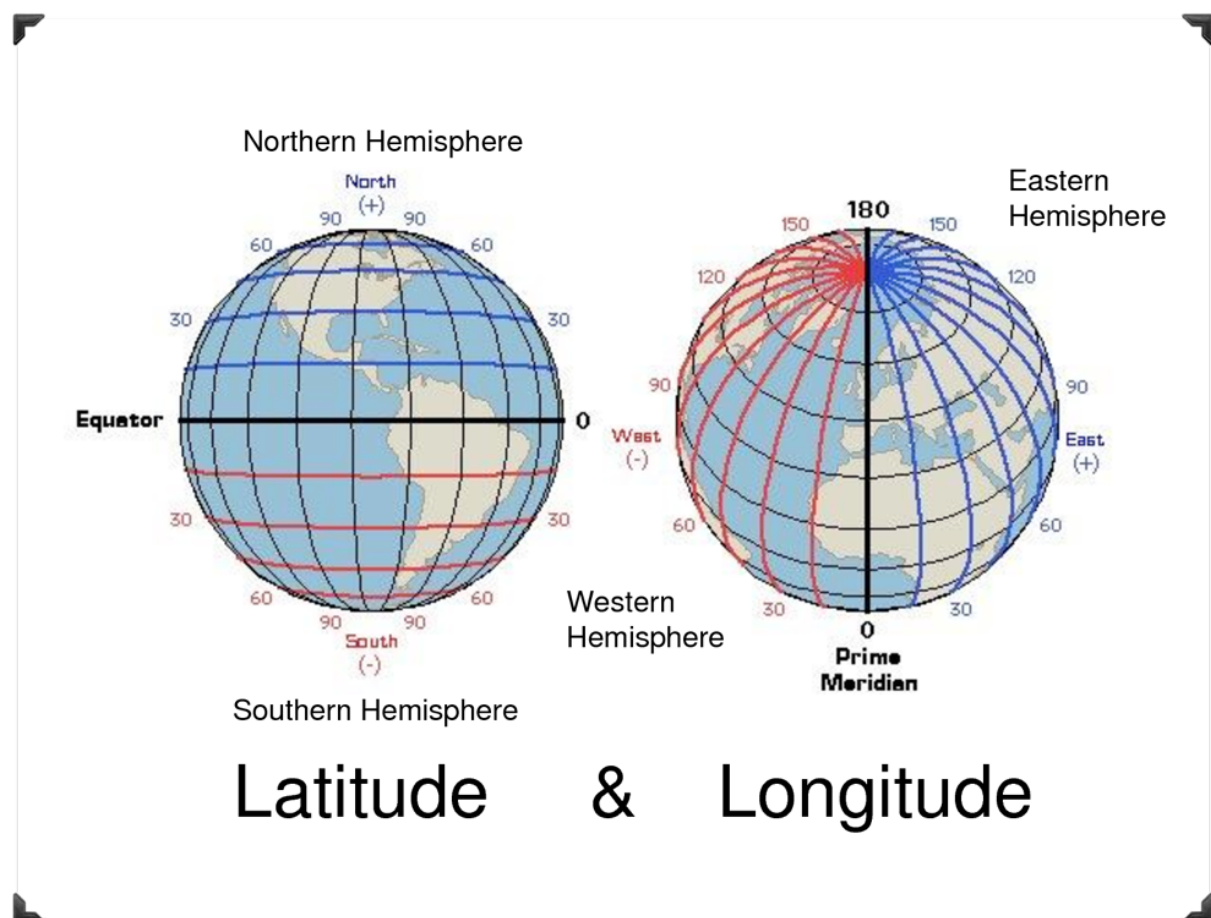
Get current location with the Geolocation API

A location on our globe is usually specified by **longitude** and **latitude**.

Latitude describes the place between north and south pole.

Longitude describes the place to the west or east.

With these two coordinates we can specify any position / geolocation on the globe.



The browser is already giving us an interface for JavaScript (an API) where we can find out our current location: with [navigator.geolocation](#).

The method `.getCurrentPosition()` tells us the geolocation of the device.

```
if (navigator.geolocation) {
  navigator.geolocation.getCurrentPosition(
    (position) => {
      const { latitude, longitude } = position.coords
      console.log('Location:', latitude, longitude)
    },
    (error) => {
      console.log('Oh no:', error)
    }
  )
}
```

This requires **user permission**. A pop-up will ask the user to grant this permission.

If the user doesn't grant the permission, we will get an error, so we need to handle that in our callback.

Google Maps for React

Google Maps provides a package for using the Google Maps API with React. It provides special components for displaying a map etc.

To install **google-map-react** type:

```
npm install --save google-map-react
```

In your JavaScript file, import it like this:

```
import GoogleMapReact from 'google-map-react'
```

Displaying a Google Map then works like this:

```
<div className="App" style={{ height: '100vh' }}>
  <GoogleMapReact
    bootstrapURLKeys={{
      key: '...'
    }}
    defaultCenter={{ lat: -33.9249, lng: 18.4241 }}
    zoom={12}
  ></GoogleMapReact>
</div>
```

Center a map to the user's location

To center the map programmatically to the user's location, use the **center** property of the **GoogleMapReact** component.

```
<GoogleMapReact
  //...
  center={{
    lat: // set user's latitude here
    lng: // set user's longitude here
  }}
></GoogleMapReact>
```

Google Maps interactions

For setting up click event handling, create a function with an **event** parameter, and then set the function on the **GoogleMapReact** component.

```
function handleClick(event) {
  const latitude = event.lat
  const longitude = event.lng
  // TODO: add this as a position to the state
}

<GoogleMapReact onClick={handleClick}>
```

Handling marker clicks

You can also set up click listeners for markers:

```
function handleMarkerClick(key) {
  // handle the marker click here
}

<GoogleMapReact onChildClick={handleMarkerClick}>
```

Global State Management

Context

With **Context** it is possible to pass data through the component tree without having to pass properties down manually at every level. Passing properties to components down in the tree is also known as **Prop-Drilling**.

Context can be considered “global” for a tree of components. It is primarily used when some data for many components must be accessible at different nesting levels.

To use them, a Context must first be created. It optionally takes an argument that contains the default value of the Context. It is only used if the appropriate context of a component that wants to use it, is not available. It is useful, for example, for testing components in isolation.

```
const AContext = React.createContext('a value');
```

To provide data to the Context the Provider is used. All your components which should have access to the Context needs to be wrapped by the **Context.Provider**.

```
<AContext.Provider value={/* some value */}>
  <AComponent />
</AContext.Provider>
```

When React renders a component, which subscribes to this Context object, it will read the current context value from the closest matching Provider above it in the tree.

To subscribe to a Context you can use the **useContext** Hook from React.

```
import AContext from './AContext'
// ...
const value = useContext(AContext);
```

Flux and Redux

In React we learned one way, the Props, to pass data to components. There are also other ways to handle state. When your app becomes greater, drilling props down becomes daunting and tiring for most devs. React-Redux is a javascript library for managing the state in your app. It is globally managed and can be accessed by all components.

Storybook

Storybook is a library that helps creating UI elements in an isolated environment.

Introduction: <https://storybook.js.org/docs/basics/introduction/>

Quickstart: <https://storybook.js.org/docs/guides/quick-start-guide/>

This is a great way to develop all UI variants before you put them on the actual website. It also makes them more testable, because you can access use cases that would be hard to reach on the website.

It makes you think in components, and makes it easy to share and to reuse your components. This is useful, for example, if you work with designers in your team.

You can use Storybook for the documentation of your UI elements.

Stories

Storybook will create so-called stories in your project.

Story files will have the file ending *.stories.js

To create a story, run:

```
npm run storybook
```

This will create two example stories in the folder **/stories**.

You can open the story in the browser at the path **localhost:9009/**.

Knobs Addon

To install the Knobs addon, run this:

```
npm install --save-dev @storybook/addon-knobs
```

Then add this line to your **.storybook/addons.js** file:

```
import '@storybook/addon-knobs/register'
```

Chakra UI

Chakra UI is a component library for React. It provides some ready-made components out of the box that are already styled. You can use them and compose them like lego bricks. It makes it easy to write pretty apps without writing any CSS.

```
// Sample component from airbnb.com

<Box>
  <Image rounded="md" src="https://bit.ly/2k1H1t6"/>
  <Flex align="baseline" mt={2}>
    <Badge variantColor="pink">Plus</Badge>
    <Text
      ml={2}
      textTransform="uppercase"
      fontSize="sm"
      fontWeight="bold"
      color="pink.800"
    >
      Verified &bull; Cape Town
    </Text>
  </Flex>
  <Text mt={2} fontSize="xl" fontWeight="semibold" lineHeight="short">
    Modern, Chic Penthouse with Mountain, City & Sea Views
  </Text>
  <Text mt={2}>$119/night</Text>
  <Flex mt={2} align="center">
    <Box as={MdStar} color="orange.400" />
    <Text ml={1} fontSize="sm"><b>4.84</b> (190)</Text>
  </Flex>
</Box>
```



PLUS VERIFIED • CAPE TOWN

Modern, Chic Penthouse with Mountain, City & Sea Views

\$119/night

★ 4.84 (190)

<https://chakra-ui.com/>

Useful features of Chakra are:

- It makes your app accessible (easy to use for users with disabilities)
- It is themeable (you can switch themes, or provide themes to the user)
- It is composable
- It is fast

This is how an example layout in Chakra looks like:

```
function App() {  
  return (  
    <ThemeProvider theme={theme}>  
      <CSSReset />  
      <Header />  
      <Stack spacing={8} align="center" margin={8}>  
        <Card />  
        <Card />  
        <Card />  
      </Stack>  
      <Composer />  
    </ThemeProvider>  
  )  
}
```

Alternatives

There are some alternatives to Chakra UI, including:

- Material Design <https://material.io/design/>
- Reach UI <https://reacttraining.com/reach-ui/>

Glossary

Server-Rendered App	Web application where the server hosts and delivers all HTML and CSS
Single Page App (SPA)	Web application, where most HTML is created by JavaScript running at the client
Isomorphic / Universal App	Mix-type of server-rendered app and single page app
HTML skeleton	An HTML file that does not have much body content except a div, which will be the target for the JavaScript of React to attach all other HTML elements to.
Component	A part of the UI, e.g. an avatar or a button. Components are usually composites, meaning that they can be composed of other components.
AJAX (Asynchronous Javascript and XML)	Describes a concept of asynchronous data transfer between a browser and the server
JSX	JavaScript Syntax Extension: Pseudo HTML elements that make up components, e.g. <code><BlogSection></code> .
expression	Placeholders for string content in React, that will display the content of a variable. For example: <code><p>Name: {person.name}</p></code>
props	Properties for components. For example: <pre>function Header(props) { return <h1>{props.title}</h1> }</pre>

DOM	Document Object Model: the tree of all tags in an HTML document.
node	A JavaScript object for a HTML tag with all necessary information (e.g. attributes).
node.children	All nested HTML tags of a node.
state	The internal data of a component.
routing	Mapping URLs to pages, e.g. <code>hostname.com/</code> to the index page and <code>hostname.com/about/</code> to the "About" page.
route	The part of the url after the hostname, e.g. <code>/</code> or <code>/about</code> .
router	The piece of JavaScript software that does the routing. In React, we use the <code>react-router-dom</code> package.
API	Application Programming Interface: functionality, that we can access with our programming language. In this course, we use so-called REST APIs to fetch data from servers using HTTP.
Local Storage	A way to save text as key-value pairs in the browser with JavaScript.
Latitude	Latitude describes the place between north and south pole.
Longitude	Longitude describes the place to the west or east.
User permission	A security concept from the browser where the user has to grant permission, e.g. when the

website wants to use the user's location.
Browsers usually show pop-up dialogs for that.

Useful Links

React Tutorial: <https://reactjs.org/tutorial/tutorial.html>

Modern React Documentation (still in beta): <https://beta.reactjs.org/>

React Course on Scrimba: <https://scrimba.com/course/glearnreact>

React Hooks: <https://reactjs.org/docs/hooks-effect.html>

React Router: <https://reacttraining.com/react-router/web/guides/quick-start>

Local Storage: <https://www.robinwieruch.de/local-storage-react>

Storybook: <https://storybook.js.org/docs/basics/introduction/>

Chakra UI: <https://chakra-ui.com/>

Google Maps for React: <https://github.com/google-map-react/google-map-react>

Recap of ES6 on Scrimba: <https://scrimba.com/course/gintrotoes6>