How to create it easilly?

- React team created the create React App. It is a comfortable environment for learning React, and is the best way to start building in React.
- Sets up your development environment so that you can use the latest JavaScript features, provides a nice developer experience, and optimizes your app for production.
- Doesn’t handle backend logic or databases; you can use it with any backend you want.
- Under the hood, it uses ***babel*** and ***webpack***, but you don’t need to know anything about them.
- To deploy to production, run **npm run build**. It will create an optimized build of your app in the build folder.

Lab React20

- Step 1: To use the create-react-app run:

```
> npx create-react-app react_example_20-create_react_app
```

*You may need to add –**use-npm** if you have yarn installed in your system.*

If you get an error of the type 'EPERM : Operation not permitted' you might have spaces on the path to the global cache folder.
Open a command line and do:
> cd c:\Users & dir /x
Get the shortname for your users folder and do:
> npm config set cache "C:/Users/<shortname>/AppData/Roaming/npm-cache" --global

- Step 2: Execute:

```
> cd react_example_20-create_react_app
> npm start
```

  - Congratulations, you have a running React App.

Lab React20

- Step 3: Look into the project structure.
  - */public* and */src* directories, along with the regular .gitignore, node_modules, README.md, and package.json files.
  - in /public, the important file is *index.html*.
  - the /src directory will contain all React code.
- Step 4: see how the environment automatically compiles and updates the React code.
  - Find the line that looks like this in /src/App.js:

```
Edit <code>src/App.js</code> and save to reload.
```

  - replace it with any other text and save the file,
  - localhost:3000 compiles and refreshes with the new data.

Lab React20

- Step 5: Try to recreate the Recipe component from react_example_18-WebpackJSXPassingUpState in this React Application.
- Step 6 : Try to build for production.

Forms

- In HTML the **DOM is the storage**.
  - When working with an input element, the page's DOM maintains that element's value in its DOM node;
  - It's possible to access the value via methods like *document.getElementById('email').value*.
- The React documentation states that "React components **must represent the state of the view at any point in time** and not only at initialization time."
- HTML form elements work a bit differently from other DOM elements in React.

Forms

- Let’s try to create an input element and set its value;
    1) This code represents the view at any state, so the value will always be Mr..

```
render() {
  return <input type="text" name="title" value="Mr." />
}
```

2) This is a better implementation, because it’ll be updated from the state. But what’s the value of *state*?

```
render() {
  return React.createElement('input', {type: 'text', name: 'title', value: this.state.title});
}
```

The ***value*** *attribute is used to set the displayed value on the form element. This is a* **controlled component**.
In this case the value will always be this.state.title, making the React state the source of truth.
In the above example, the displayed value will always be ‘Mr.’

Forms – Controlled Components

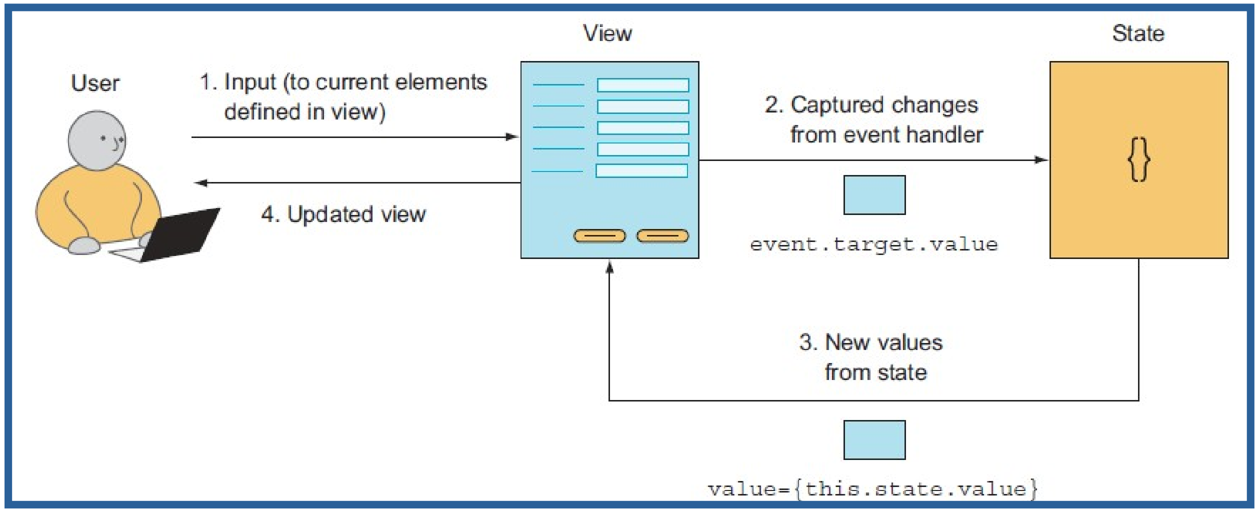
- React is not capable to read what users are typing in form elements.
- We need to implement an event handler to capture changes with *onChange*

```
handleChange(event) {
  this.setState({title: event.target.value})
}
render() {
  return <input type="text" name="title" value={this.state.title}
    onChange={this.handleChange.bind(this)}/>
}
```

- React can handle some Form DOM events: *onChange*, *onInput*, *onInvalid*, *onReset* and *onSubmit* plus the standard React events.

Forms – Controlled Components

- Best practice to sync the internal state with the view:
  1) Define elements in render() using values from state.
  2) Capture changes to a form element as they happen, using onChange.
  3) Update the internal state in the event handler.
  4) New values are saved in state, and then the view is updated by a new render().



*This approach implements **controlled components**. Please take into consideration that it is also possible to implement uncontrolled components (where form data is handled by the DOM itself).*

Tutorial React21

- Step 1: To use the create-react-app run:

```
> npx create-react-app react_example_21-forms_controlled_components
```

- Step 2: Copy to *src/* the jsx files from Tutorial React19:
  - ClockPresenter.jsx
  - Clock.jsx
  - Analog-display.jsx
  - Digital-display.jsx

Tutorial React21

- Step 3: add moment-timezone dependency to the project:

```
> npm i moment-timezone
```

- Step 4: edit analog-display.jsx:

```
import PropTypes from 'prop-types';

//…

export default AnalogDisplay;
```

- Step 5: edit digital-display.jsx:

```
import PropTypes from 'prop-types';

//…

export default DigitalDisplay;
```

Tutorial React21

- Step 6: edit clock.jsx:

```
import React from 'react';
import AnalogDisplay from './analog-display';
import DigitalDisplay from './digital-display';

export default class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {currentTime: new Date().toLocaleString('en-US', {timeZone: this.props.timezone})};
  }
  componentDidMount() {
    this.timerId = setInterval(() => {
      console.log("Updating...");
      this.setState({ currentTime: new Date().toLocaleString('en-US', {timeZone: this.props.timezone})});
    }, 1000);
  }
  componentWillUnmount() {
    clearInterval(this.timerId);
  }
  render() {
    console.log("Rendering...");
    return (
      <div>
        <AnalogDisplay time={this.state.currentTime} />
        <DigitalDisplay time={this.state.currentTime} />
        <p>Displaying timezone : {this.props.timezone}</p>
      </div>
    );
  }
}
```

Tutorial React21

- Step 7: edit clockPresenter.jsx:

```
import React from "react";
import Clock from "./clock";
import momentTZ from "moment-timezone";

export default class ClockPresenter extends React.Component {
  static TIMEZONE_LIST = momentTZ.tz.names();
  constructor(props) {
    super(props);
    this.state = {showClock: true, currentTimeZone: momentTZ.tz.guess()};
  }
  onClick = () => {
    let value = !this.state.showClock;
    this.setState({ showClock: value });
  };
  handleChange = (event) => {
    this.setState({currentTimeZone: event.target.value})
  }
  render() {
    return (
      <div>
        <label>Select Timezone: <select value={this.state.currentTimeZone} onChange={this.handleChange}>
          {ClockPresenter.TIMEZONE_LIST.map((timezone, i) => (<option value={timezone}
key={i}>{timezone}</option>))}</select>
      </label>
        <button onClick={this.onClick}>
          {this.state.showClock ? "Hide" : "Show"}
        </button>
        {this.state.showClock && <Clock timezone={this.state.currentTimeZone} />}
      </div>
    );
  }
```

Let React be "the source of truth".

Tell React the Component changed

Update the timezone property of Clock

Tutorial React21

- Step 8: edit App.js:

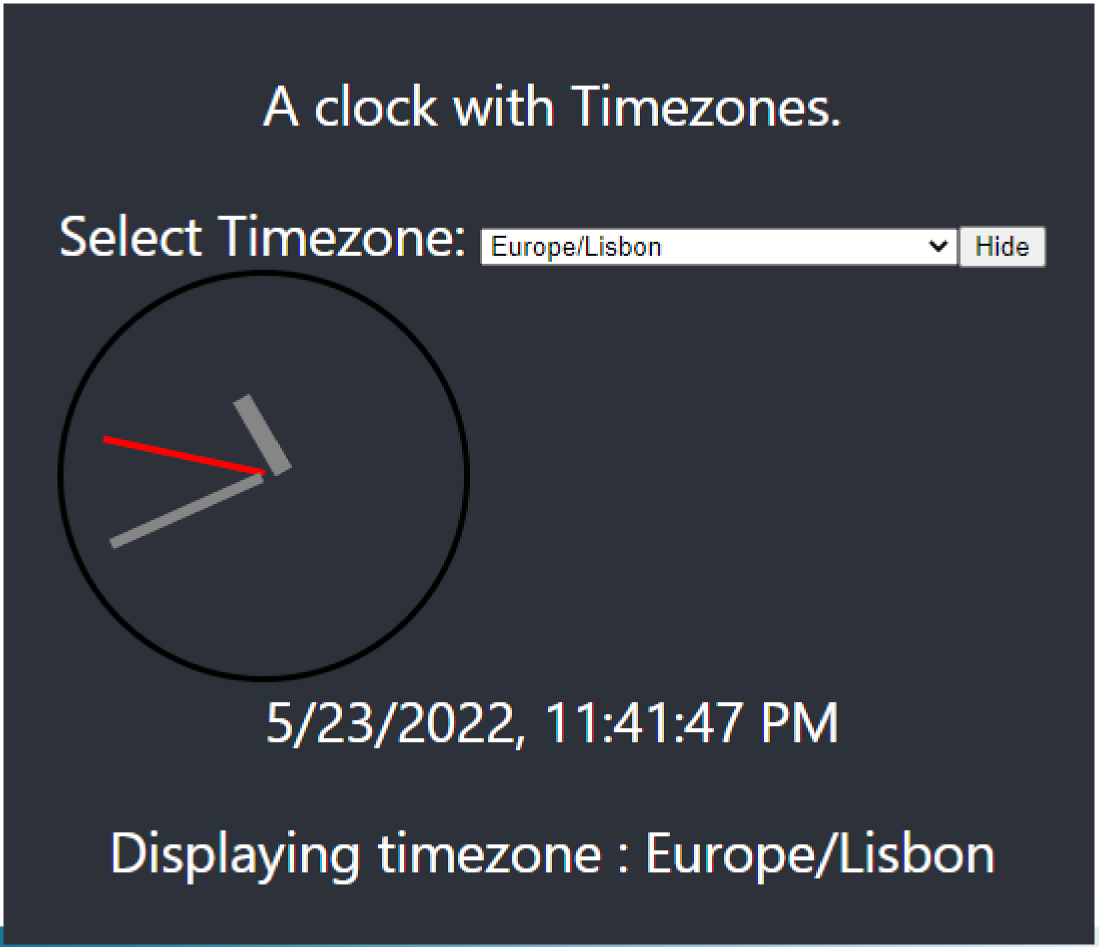
```
import logo from './logo.svg';
import './App.css';
import ClockPresenter from './clockPresenter';

function App() {
  return (
    <div className="App">
      <header className="App-header">
        <p>
          A clock with Timezones.
        </p>
        <ClockPresenter />
      </header>
    </div>
  );
}

export default App;
```

- Run the App:

```
> npm start
```

Tutorial React21

- Try to change the Timezone.



A clock with Timezones.

Select Timezone: Europe/Lisbon Hide

5/23/2022, 11:41:47 PM

Displaying timezone : Europe/Lisbon

References

- Considered an antipattern because when React elements are defined properly, with each element using internal state in sync with the view's state (DOM), the need for references is almost non-existent
- With references, you can get the DOM element (or a node) of a React.js component. This comes in handy when you need to get form element values, but you don't capture changes in the elements.
- Refs are created using *React.createRef()* and attached to React elements via the *ref* attribute.

References

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

- Then when a ref is passed to an element in render, a reference to the node becomes accessible at the current attribute of the ref.

```
const node = this.myRef.current;
```

In the React documentation there are several examples on how to access refs and expose refs to parent components

Forms – Uncontrolled Components

- Uncontrolled components are components whose state (form data) is not handled by React.
- To write an uncontrolled component, instead of writing an event handler for every state update, we can use a ref to get form values from the DOM.

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }
  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>Name: <input type="text" ref={this.input} /></label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

Forms – Uncontrolled Components

- In React, an *<input type="file" />* is always an uncontrolled component because its value can only be set by a user, and not programmatically.

```
class FileInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.fileInput = React.createRef();
  }
  handleSubmit(event) {
    event.preventDefault();
    alert(`Selected file - ${this.fileInput.current.files[0].name}`);
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Upload file:
          <input type="file" ref={this.fileInput} />
        </label>
        <br />
        <button type="submit">Submit</button>
      </form>
    );
  }
}
```

Tutorial React22

- Step 1: Copy react_example_21-forms_controlled_components to **react_example_22-forms_uncontrolled_components** and open in VSCode.

Tutorial React22

- Step 2: Edit ClockPresenter.jsx:

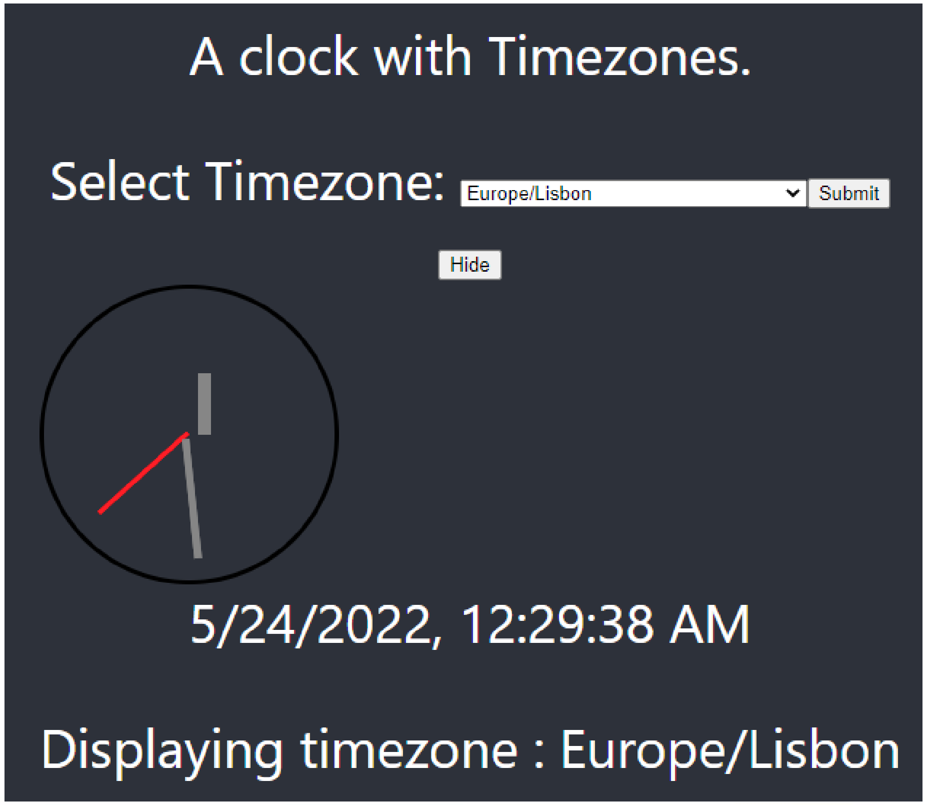
```
//…
  constructor(props) {
    super(props);
    this.timezone = React.createRef();
    this.state = {
      showClock: true,
      currentTimeZone: momentTZ.tz.guess(),
    };
  }
//…
  handleChange = (event) => {
    alert('currentTimeZone: ' +this.timezone.current.value);
    this.setState({ currentTimeZone: this.timezone.current.value });
    event.preventDefault();
  };
  render() {
    return (
      <div>
        <form onSubmit={this.handleChange}>
          <label>Select Timezone: <select defaultValue={this.state.currentTimeZone} ref={this.timezone}>
              {ClockPresenter.TIMEZONE_LIST.map((timezone, i) => (<option value={timezone}
key={i}>{timezone}</option>))}
            </select>
          </label>
          <input type="submit" value="Submit" />
        </form>
        <button onClick={this.onClick}>{this.state.showClock ? "Hide" : "Show"}</button>
        {this.state.showClock && (<Clock timezone={this.state.currentTimeZone} />)}
      </div>
    );
  }
```

Select is now an uncontrolled component

Tutorial React22

- Step 3: Run the App:

```
> npm start
```


A clock with Timezones.

Select Timezone: Europe/Lisbon Submit

Hide

5/24/2022, 12:29:38 AM

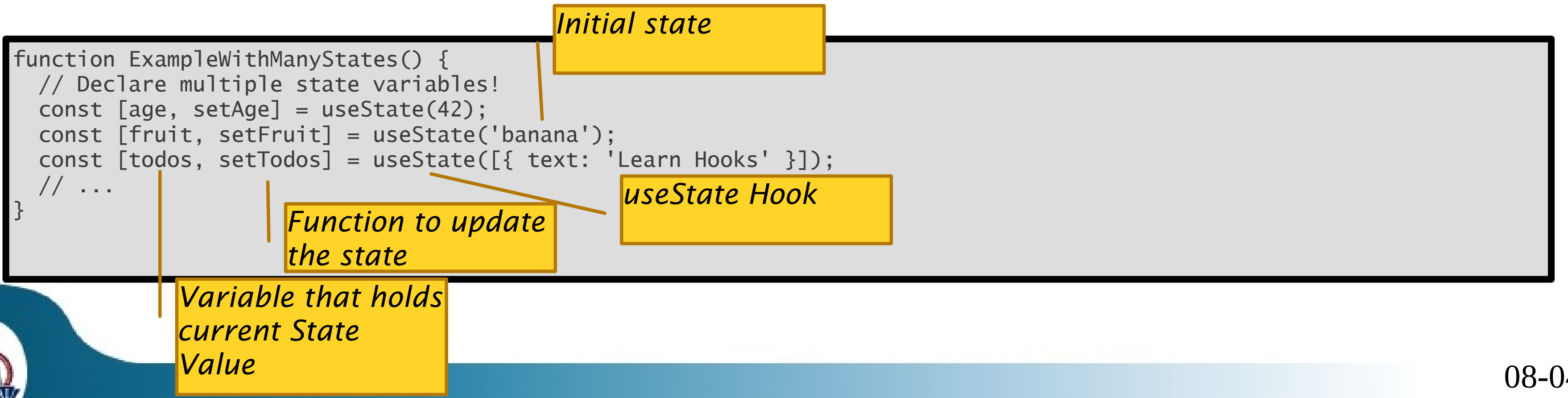
Displaying timezone : Europe/Lisbon

Hooks

- Hooks are a new feature on version 16.8.
- Hooks are 100% backward compatible.
- Hooks complement (don’t replace) knowledge of React concepts, such as props, state, context, refs, and lifecycle. They provide a new way to combine them.
- Hooks are functions that allow to “hook into” React state and lifecycle features from function components.
- Hooks don’t work inside classes.
- React provides a few built-in Hooks. New Hookscan be created to reuse stateful behaviour between different components.

useState Hook

- Called inside a function component to add some local state to it.
- React will preserve this state between re-renders.
- useState() returns a pair: the current state value and a function that lets update it (the state value). The function can be called from an event handler or somewhere else. It’s similar to this.setState in a class, except it doesn’t merge the old and new state together.
- The only argument to useState is the initial state.
- Can be used more than once in a single component.



```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

Initial state
useState Hook
Function to update the state
Variable that holds current State Value

useState Hook

- Declare a state variable called ***count***, and set it to 0.
- Normally, variables “*disappear*” when the function exits but state variables are preserved by React.
- React will remember its current value between re-renders, and provide the most recent one to our function. If we want to update the current count, we can call ***setCount***.

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
```

“array destructuring”. It means that we’re making two new variables fruit and setFruit, where fruit is set to the first value returned by useState, and setFruit is the second

useState Hook

- To read the state:

```
<p>You clicked {count} times</p>
```

- To update the state:

```
<button onClick={() => setCount(count + 1)}>Click me</button>
```

- Complete example:

```
 import React, { useState } from 'react';

 function Example() {
   const [count, setCount] = useState(0);

   return (
     <div>
       <p>You clicked {count} times</p>
       <button onClick={() => setCount(count + 1)}>
        Click me
       </button>
     </div>
   );
 }
```

useEffect Hook

- The Effect Hook allows to perform **side effects in function components**.
- In React class components, the render method itself shouldn't cause side effects. It would be too early. We typically want to perform effects after React has updated the DOM. This is why in React classes, we put side effects into *componentDidMount* and *componentDidUpdate*.
- With useEffect Hook we tell React that the component needs to do something **after render**. By default, it runs both after the first render and after every update.
- Placing *useEffect*() inside the component allows to access state variables (or any props) right from the effect.

"Side Effect" is not a react-specific term. It is a general concept about behaviours of functions. A function is said to have side effect if it trys to modify anything outside its body. For example, if it modidifies a global variable, then it is a side effect. If it makes a network call, it is a side effect as well.

useEffect Hook Example

With a class component

```
class MyButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>Click me</button>
      </div>
    );
  }
}
```

With a function component

```
import React, { useState, useEffect } from 'react';
function MyButton() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

useEffect Hook with dependency

- UseEffect takes a dependency array as a second parameter:

```
useEffect(() => {}, [])
```

- **an empty dependency array** will make useEffect **run only once** after the rendering of the page. It will not re-run on updating of any states.
- To re-run any useEffect on updating of any particular state then pass the key of useState in the dependency array:

```
import React, { useState, useEffect } from 'react';
function MyButton() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    setTimeout(() => {
      setCount(count + 1)
    }, 5000);
  }, [])
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  }, [count]);
  return (
    <div>
      <p>You clicked {count} times</p>
    </div>
  );
}
```

useEffect Hook and cleanup

- Sometimes we might need to use effects that require cleanup when the component is unmounted. For example wen we have a subscription for external data, or we may introduce a memory leak.
- In a ***React class***, you would typically set up a subscription in ***componentDidMount***, and clean it up in ***componentWillUnmount***.
- With **hooks** the cleanup code is a **returned function** from the **useEffect** function. React will run this function when ots time to cleanup.

useEffect Hook Example

With a class component

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
  }
  componentDidMount() {ChatAPI.subscribeToFriendStatus(this.props.friend.id,this.handleStatusChange);}
  componentWillUnmount() {ChatAPI.unsubscribeFromFriendStatus(this.props.friend.id,this.handleStatusChange);}
  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }.bind(this);
  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

With a function component

```
import React, { useState, useEffect } from 'react';
function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
  if (isOnline === null) {return 'Loading...';}
  return isOnline ? 'Online' : 'Offline';
}
```

ESLint plugin for Hooks

- Only Call Hooks from React Functions
- Don’t call Hooks from regular JavaScript functions.
  - Call Hooks from React function components;
  - Call Hooks from custom Hooks.
- There is an ESLint plugin called ***eslint-plugin-react-hooks*** that enforces the above two rules.
- Add this plugin to the project:

```
> npm i -D eslint-plugin-react-hooks
```

- This plugin is included by default in **Create React App**.

Making API calls with AXIOS

- There are a number of different libraries we can use to make HTTP requests.
- AXIOS:
  - has good defaults to work with JSON data;
  - has function names that match any HTTP methods;
  - does more with less code;
  - has better error handling;
  - can be used on the server as well as the client.

Tutorial React23

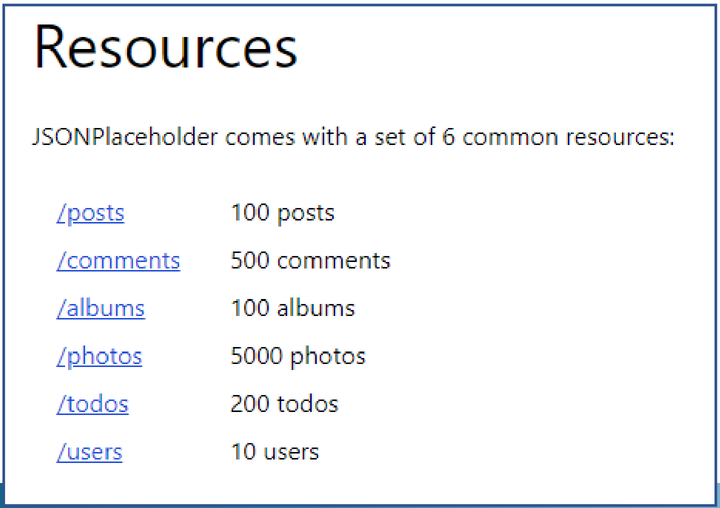
- Step 1: To use the create-react-app run:

```
> npx create-react-app react_example_23-axios
```

- Step 2: Add the Axios module dependency to the project:

```
> cd react_example_23-axios
> npm i axios
```

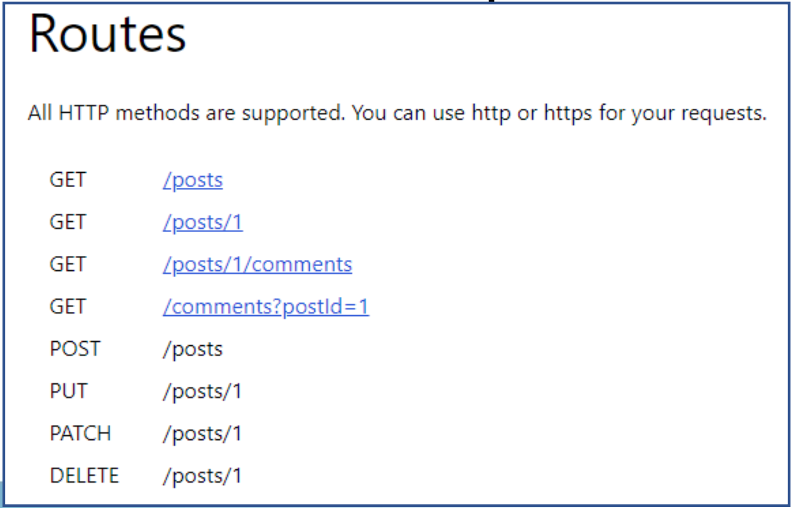
- We will implement a client for the Posts API of JSONPlaceholder. Save and test on each step.



Resources

JSONPlaceholder comes with a set of 6 common resources:

/posts	100 posts
/comments	500 comments
/albums	100 albums
/photos	5000 photos
/todos	200 todos
/users	10 users



Routes

All HTTP methods are supported. You can use http or https for your requests.

GET	/posts
GET	/posts/1
GET	/posts/1/comments
GET	/comments?postId=1
POST	/posts
PUT	/posts/1
PATCH	/posts/1
DELETE	/posts/1

Tutorial React23

- Step 3: Getting Posts. Edit App.js

```
import "./App.css";
import axios from "axios";
import React from "react";

const baseURL = "https://jsonplaceholder.typicode.com/posts";

export default function App() {
  const [posts, setPosts] = React.useState(null);

  React.useEffect(() => {
    axios.get(baseURL).then((response) => {
      setPosts(response.data);
    });
  }, []);

  if (!posts) return null;

  return (
    <div>
      <table>
        <thead><tr><th>id</th><th>title</th><th>UserId</th></tr></thead>
        <tbody>
          {posts.map((post, i) => (
            <tr key={i}>
              <td>{post.id}</td>
              <td>{post.title}</td>
              <td>{post.userId}</td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
};
```

Tutorial React23

- Step 4: Create a new Posts. Edit App.js

```
import "./App.css";
import axios from "axios";
import React from "react";

const baseURL = "https://jsonplaceholder.typicode.com/posts";

export default function App() {
// …
  function createPost() {
    axios.post(baseURL, {title: "Hello World!", body: "This is a new post.",})
      .then((response) => {setPosts(posts.concat( response.data));});
  }
// …
  return (
    <div>
      <table>
        <thead><tr><th>id</th><th>title</th><th>UserId</th></tr></thead>
        <tbody>
          {posts.map((post, i) => (
            <tr key={i}>
              <td>{post.id}</td>
              <td>{post.title}</td>
              <td>{post.userId}</td>
            </tr>
          ))}
        </tbody>
      </table>
      <button onClick={createPost}>Create Post</button>
    </div>
  );
};
```

Tutorial React23

- Step 5: Update a Posts. Edit App.js

```
// …
export default function App() {
// …
  const [, setState] = React.useState(null);
  function updatePost(event) {
    axios.put(`${baseURL}/${event.currentTarget.dataset.index}`, {
        title: "Hello World!",
        body: "This is an updated post.",
        userId: 22
      })
      .then((response) => {
        posts[posts.findIndex((el) => el.id === response.data.id)] = response.data;
        setPosts(posts);
        setState({});
      });
  }
// …
  return (
// …
        <thead><tr><th>id</th><th>title</th><th>UserId</th><th></th><th>Update</th></tr></thead>
        <tbody>
          {posts.map((post, i) => (
            <tr key={i}>
              <td>{post.id}</td>
              <td>{post.title}</td>
              <td>{post.userId}</td>
              <td><button data-index={post.id} onClick={updatePost}>U</button></td>
            </tr>
//…
  );
};
```

Tutorial React23

- Step 5: Delete a Posts. Edit App.js

```
// …
export default function App() {
// …
    function deletePost(event) {
      const deletedId = event.currentTarget.dataset.index;
      axios.delete(`${baseURL}/${deletedId}`)
      .then(() => {
        posts.splice(posts.findIndex((el) => String(el.id) === String(deletedId)) ,1);
        setPosts(posts);
        setState({});
      });
    }
// …
  return (
// …
        <thead><tr><th>id</th><th>title</th><th>UserId</th><th></th><th>Update</th><th>Delete</th></tr></thead>
        <tbody>
          {posts.map((post, i) => (
            <tr key={i}>
              <td>{post.id}</td>
              <td>{post.title}</td>
              <td>{post.userId}</td>
              <td><button data-index={post.id} onClick={updatePost}>U</button></td>
              <td><button data-index={post.id} onClick={deletePost}>D</button></td>
            </tr>
//…
  );
};
```

React UI Component Libraries

- React.js is a library that powers the web user interface (UI).
- While the base library of React.js is solid, there are multiple React Components libraries filled with valuable design elements for React apps or web development projects.
- Several React UI libraries can be found on Github.

React UI Component Libraries Pros

- **Beginner-friendly**: prebuilt components like buttons, form fields, etc. No need to create Components from scratch. Good documentation. Focus on the implementation and customization.
- **Faster prototyping**. With ready-made React components at disposal, quickly create several functioning prototypes. Prove that the design concept is functioning without focus on the details.
- **Saves time**. Not only when prototyping, but also when developing the React project. Write less code. No need to write all the styles.
- **Recognizable components by users**. Innovation helps the project stand out. Too much innovation in UX/UI can put users off. Components in libraries are designed to be universal.
- **Customizable components**. Despite being universal, most Components can be customized enough to make sure the website doesn't look to much like many others.
- **Proven compatibility across devices**. Most prebuilt UI components are mobile-responsive by default. No extra effort into ensuring the React project works on different types of devices.
- **Accessible by default**. Most libraries have built-in accessibility features, fully adhere to WCAG or other standards and best practices. No self-coding semantic tags or keyboard navigation.
- **Crowd-sourced**. UI component libraries often have their communities centered around GitHub. This means developers can raise issues, request features, and also easily become contributors.

React UI Component Libraries Cons

- **Customizing components can be difficult**. The ease of customizing components differs across libraries. With some React libraries getting the expected result can be tricky.
- **Similar web design with other sites**. Choosing a popular library and not customizing the components or theme enough, the site can end up looking very similar to any other sites using the same library.
- **Support relies on the community**. Most React UI libraries don’t offer official support but instead guide their users to Stack Overflow, GitHub, Discord, or other similar channels. With less popular libraries, the community is smaller, and getting help can be more complicated.

React UI Component Libraries Examples

- MUI (formerly Material-UI)
- React-Bootstrap
- Ant Design
- Reactstrap
- Semantic UI React
- Chakra UI
- Theme UI
- Rebass
- Blueprint
- VisX

MUI

- Developed by Google in 2014, MUI (former Material-UI) is a general-purpose customizable component library to build React applications. The folks at Google designed Material-UI as an adaptable system of guidelines, components, and tools to make app building beautiful yet straightforward.

Tutorial React24

- Step 1: To use the create-react-app run:

```
> npx create-react-app react_example_24-mui
```

- Step 2: Start the project (can be in another CLI):

```
> cd react_example_24-mui
> npm start
```

- Step 3: Install MUI dependencies:

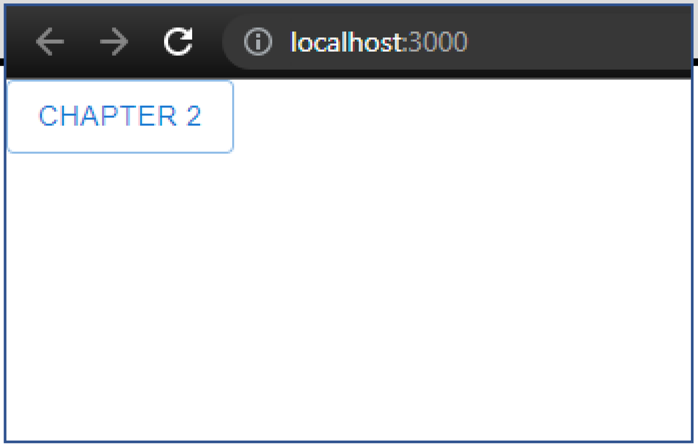
```
> npm install @mui/material @emotion/react @emotion/styled
```

Tutorial React24

- Step 4: Edit App.js

```
import React, { Component } from "react";

import Button from "@mui/material/Button";

class App extends Component {
  render() {
    return (
        <Button variant="outlined" color="primary">
          Chaper 2
        </Button>
    );
  }
}


export default App;
```

Special props for the Button component include:
***variant**: The visual style of the component, either contained, outlined, fab, or empty for the default link style.*
***color**: One of primary, secondary, or default, the same color as if it’s left empty. We’ll cover the customization of these colors later.*
***mini**: If the variant is set to fab (floating action button), then the size of the button is reduced.*



Tutorial React24

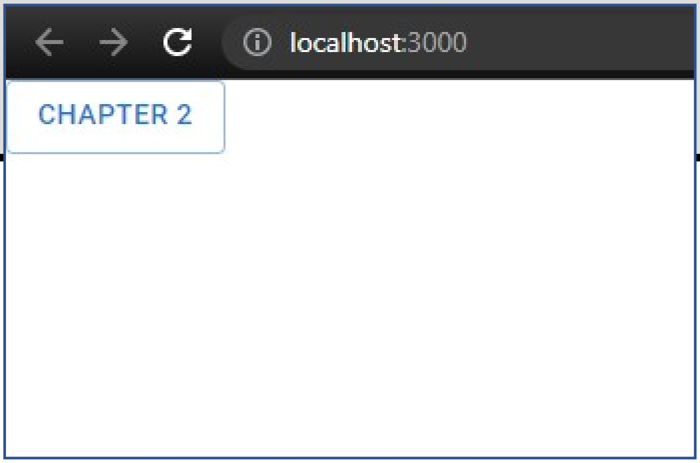
- Step 5: MUI was designed with the Roboto font in mind. Roboto font will not be automatically loaded by MUI. The programmer is responsible for loading any fonts used in the application. Roboto Font has a few easy ways to get installed. Adding it through npm:

```
> npm install @fontsource/roboto
```

Tutorial React24

- Step 6: Edit App.js

```
import "@fontsource/roboto/300.css";
import "@fontsource/roboto/400.css";
import "@fontsource/roboto/500.css";
import "@fontsource/roboto/700.css";

import React, { Component } from "react";
import Button from "@mui/material/Button";

class App extends Component {
  render() {
    return (
        <Button variant="outlined" color="primary">
          Chaper 2
        </Button>
    );
  }
}


export default App;
```



Tutorial React24

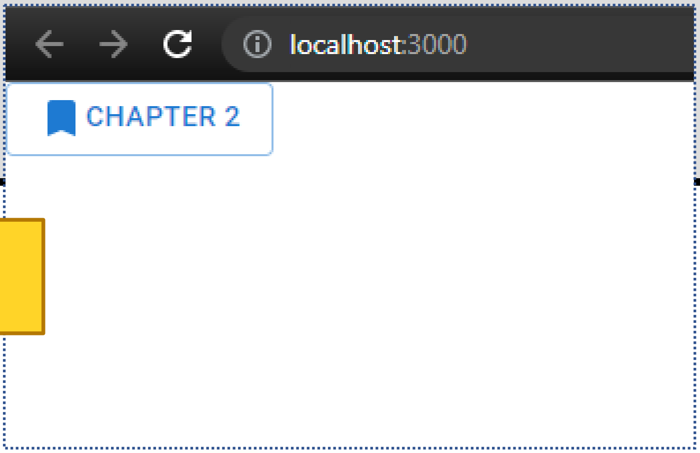
- Step 7: Install prebuilt SVG icons:

```
> npm install @mui/icons-material
```

Tutorial React24

- Step 8: Edit App.js

```
import "@fontsource/roboto/300.css";
import "@fontsource/roboto/400.css";
import "@fontsource/roboto/500.css";
import "@fontsource/roboto/700.css";

import React, { Component } from "react";
import Button from "@mui/material/Button";
import Bookmarks from "@mui/icons-material/Bookmark";

class App extends Component {
  render() {
    return (
      <Button variant="outlined" color="primary">
        <Bookmarks></Bookmarks>
        Chapter 2
      </Button>
    );
  }
}

export default App;
```



SvgIcon material-ui page.
The icons search page (https://mui.com/material-ui/material-icons/).

Tutorial React24

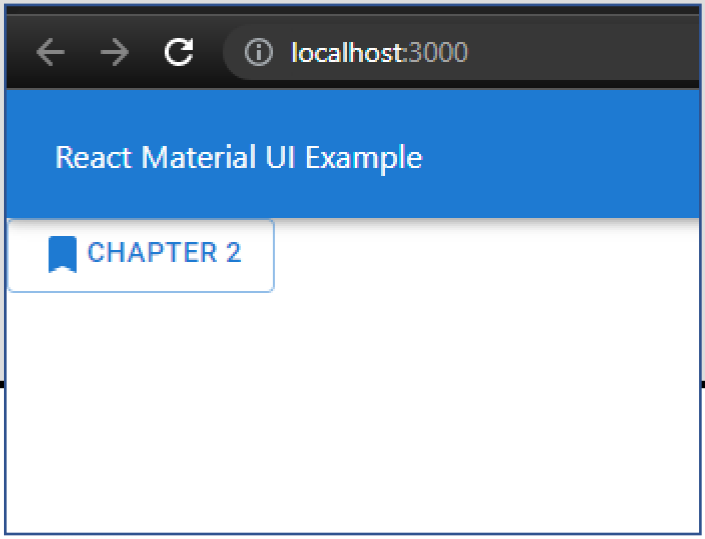
- Step 9: Lets create a Navbar for our content. Create the file **src/Navbar.js** an add the code:

```
// Navbar.js

import React from 'react';
import AppBar from '@mui/material/AppBar';
import Toolbar from '@mui/material/Toolbar';

const NavBar = () => {
    return(
        <div>
        <AppBar position="static">
            <Toolbar>
                React Material UI Example
            </Toolbar>
        </AppBar>
        </div>
    )
}
export default NavBar;
```

Tutorial React24

- Step 10: Edit App.js

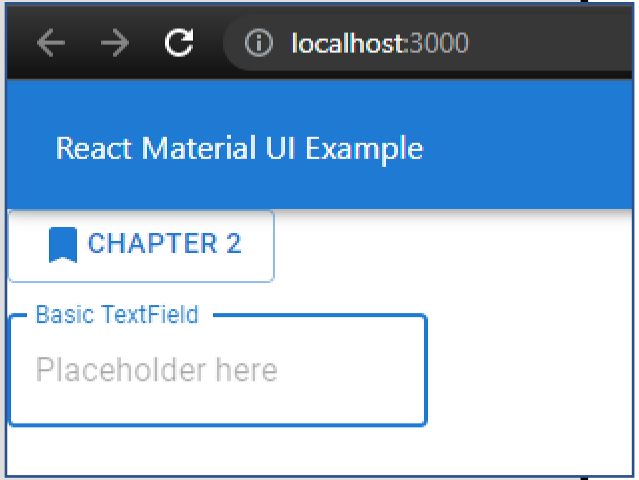
```
import "@fontsource/roboto/300.css";
import "@fontsource/roboto/400.css";
import "@fontsource/roboto/500.css";
import "@fontsource/roboto/700.css";

import React, { Component } from "react";
import Button from "@mui/material/Button";
import Bookmarks from "@mui/icons-material/Bookmark";
import NavBar from "./Navbar";

class App extends Component {
  render() {
    return (
      <div>
        <NavBar />
        <Button variant="outlined" color="primary">
          <Bookmarks></Bookmarks>
          Chapter 2
        </Button>
      </div>
    );
  }
}

export default App;
```



Tutorial React24

- Step 11: Let’s add a TextField. Edit App.js

```
import "@fontsource/roboto/300.css";
import "@fontsource/roboto/400.css";
import "@fontsource/roboto/500.css";
import "@fontsource/roboto/700.css";

import React, { Component } from "react";
import Button from "@mui/material/Button";
import { TextField } from "@mui/material";
import Bookmarks from "@mui/icons-material/Bookmark";
import NavBar from "./Navbar";

class App extends Component {
  render() {
    return (
      <div>
        <NavBar />
        <Button variant="outlined" color="primary">
          <Bookmarks></Bookmarks>
          Chapter 2
        </Button>
        <br />
        <TextField placeholder="Placeholder here" label="Basic TextField" sx={{mt:2}}/>
      </div>
    );
  }
}

export default App;
```



The `sx` prop is a shortcut for defining custom style that has access to the theme.

Tutorial React24

- Step 12: MUI Cards. Create the file **src/Card.js** an add the code:

```
// Card.js

import React from "react";
import PropTypes from "prop-types";
import { createTheme, ThemeProvider } from "@mui/material/styles";
import Card from "@mui/material/Card";
import CardActionArea from "@mui/material/CardActionArea";
import CardActions from "@mui/material/CardActions";
import CardContent from "@mui/material/CardContent";
import CardMedia from "@mui/material/CardMedia";
import Button from "@mui/material/Button";
import Typography from "@mui/material/Typography";
import IMG from "./lizard.png";


// Continues on next slide.


export default MediaCard;
```

Tutorial React24

```
function MediaCard(props) {
  const defaultTheme = createTheme();
  return (
    <ThemeProvider theme={defaultTheme}>
      <Card sx={{ maxWidth: 345 }}>
        <CardActionArea>
          <CardMedia
            sx={{ height: 340, mt:2 }}
            image={IMG}
            title="Contemplative Reptile"
          />
          <CardContent>
            <Typography gutterBottom variant="h5" component="h2">
              Lizard
            </Typography>
            <Typography component="p">
              Lizards are a widespread group of squamate reptiles, with over
              6,000 species, ranging across all continents except Antarctica
            </Typography>
          </CardContent>
        </CardActionArea>
        <CardActions>
          <Button size="small" color="primary">
            Share
          </Button>
          <Button size="small" color="primary">
            Learn More
          </Button>
        </CardActions>
      </Card>
    </ThemeProvider>
  );
}
```

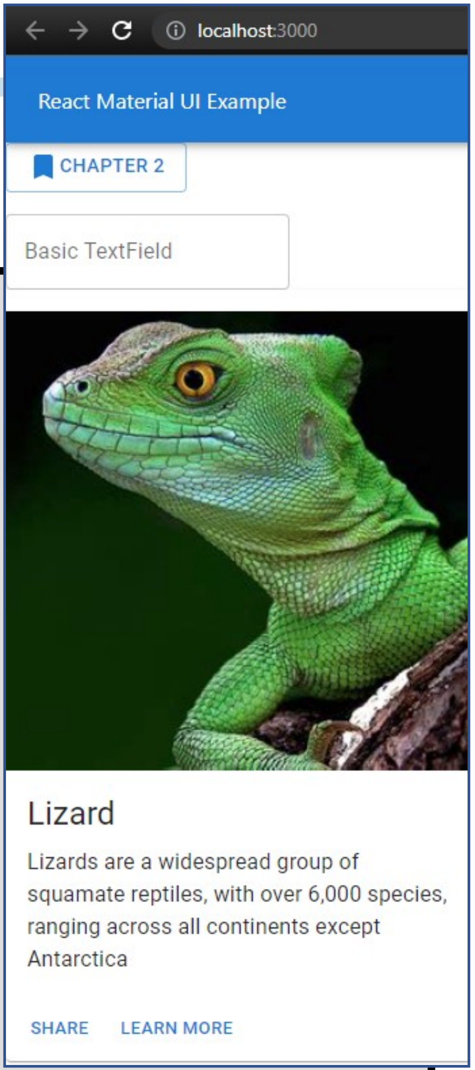
Tutorial React24

- Step 13: Find and download a lizard image. Name it src/lizard.png

Tutorial React24

- Step 14: Edit App.js

```
import "@fontsource/roboto/300.css";
import "@fontsource/roboto/400.css";
import "@fontsource/roboto/500.css";
import "@fontsource/roboto/700.css";

import React, { Component } from "react";
import Button from "@mui/material/Button";
import { TextField } from "@mui/material";
import Bookmarks from "@mui/icons-material/Bookmark";
import NavBar from "./Navbar";
import MediaCard from "./Card";

class App extends Component {
  render() {
    return (
      <div>
        <NavBar />
        <Button variant="outlined" color="primary">
          <Bookmarks></Bookmarks>
          Chapter 2
        </Button>
        <br />
        <TextField placeholder="Placeholder here" label="Basic TextField" sx={{mt:2}}/>
        <MediaCard />
      </div>
    );
  }
}

export default App;
```

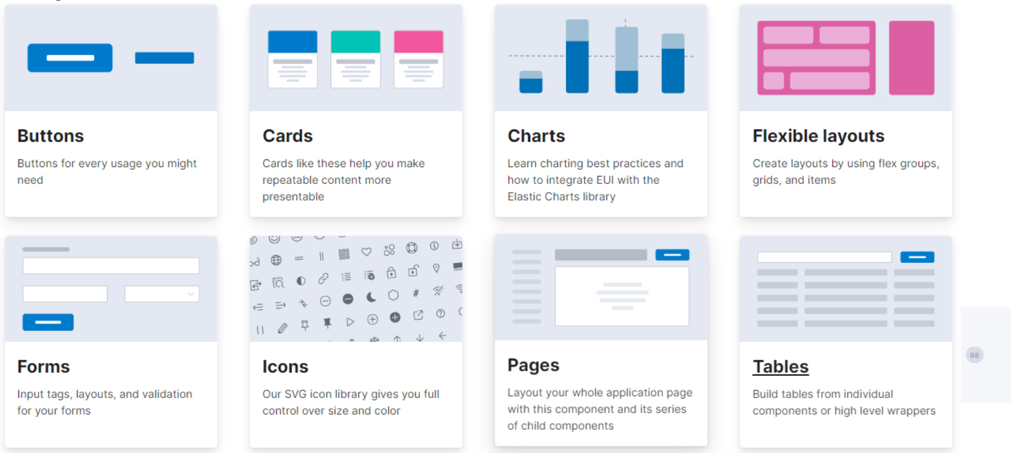


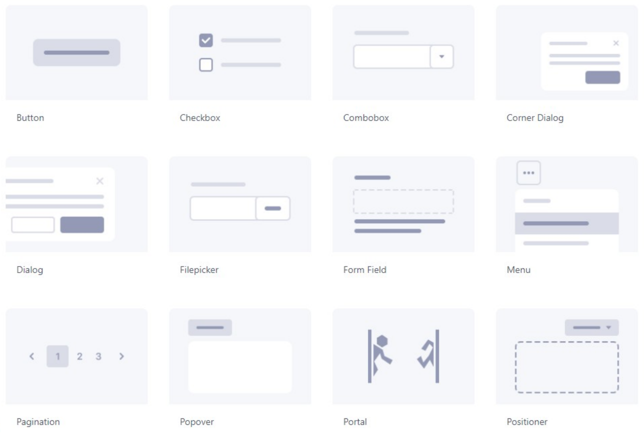
Lizard

Lizards are a widespread group of squamate reptiles, with over 6,000 species, ranging across all continents except Antarctica

Other UI Libraries for React

- We have been using MUI (former Material-UI) but other UI libraries are available (among others):
  - React Bootstrap
  - React Suite
  - Chakra
  - Blueprint
  - PrimeReact
  - Treact
  - Carolina React Admin Dashboard
  - Semantic UI React
  - React Toolbox
  - Elastic UI
  - Evergreen


Buttons
Buttons for every usage you might need
Cards
Cards like these help you make repeatable content more presentable
Charts
Learn charting best practices and how to integrate EUI with the Elastic Charts library
Flexible layouts
Create layouts by using flex groups, grids, and items
Forms
Input tags, layouts, and validation for your forms
Icons
Our SVG icon library gives you full control over size and color
Pages
Layout your whole application page with this component and its series of child components
Tables
Build tables from individual components or high level wrappers



END