--------------------------------------------------------

# React Overview

--------------------------------------------------------

React is an open-source JavaScript library focused on creating declarative user interfaces (UIs) specifically for single-page applications, using a reusable component-based concept. It’s used for handling the view layer and can be used for web and mobile apps. React’s main goal is to be extensive, fast, declarative, flexible, and simple.

**Basic Manual React Example**:  
  
1) index.html:

| <!DOCTYPE html> <**html** lang="en"> <**head**>   <**meta** charset="UTF-8">  <**script** src="https://unpkg.com/react@18/umd/react.development.js" crossorigin></**script**>  <**script** src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></**script**>  <**title**>React Page</**title**>  </**head**> <**body**> <**div** id="root"></**div**>   <**script** src="script.js"></**script**> </**body**> </**html**> |
| --- |

2) script.js:

| const rootElement = document.getElementById('root'); const root = ReactDOM.createRoot(rootElement);  const reactElement = React.createElement('header',  {className: 'site-header'},  React.createElement('h1', {}, 'Hello from React!!!!'),  React.createElement('h2', {}, 'React Subheader) );  root.render(reactElement); |
| --- |

## **The JSX Syntax**

React uses JSX syntax - with that syntax you can write concise HTML/XML-like structures (e.g., DOM like tree structures) in the same file as you write JavaScript code, then Babel will transform these expressions into actual JavaScript code.

**JSX Alternative to the above code:**

1) npm init -y  
  
2) npm install babel-cli@6 babel-preset-react-app@3  
  
3) npx babel --watch src --out-dir . --presets react-app/prod

4) script.js:

| const rootElement = document.getElementById('root'); const root = ReactDOM.createRoot(rootElement);  const reactElement = (  <**header** className="site-header">  <**h1**>Hello from JSX!</**h1**>  <**h2**>React Subheader</**h2**>  </**header**> );  root.render(reactElement); |
| --- |

**The standard way of initiating a React app:**  
  
1) npx create-react-app .  
  
2) npm start

## **Virtual DOM**

In React, for every DOM object, there is a corresponding “virtual DOM object.” A virtual DOM object is a representation of a DOM object, like a lightweight copy. It is exactly the same, but it does not have the power to directly change the layout of the document. Manipulating DOM is slow, but manipulating Virtual DOM is fast as nothing gets drawn on the screen. So each time there is a change in the state of our application, the virtual DOM gets updated first instead of the real DOM.

## **How React Works**

In React, everything is treated as a component, be it a functional component or class component. A component can contain a state. Each time the state of any component is changed, React updates its Virtual DOM tree. Though it may sound that it is ineffective, the cost is not much significant as updating the virtual DOM doesn’t take much time. React maintains two Virtual DOM at each time, one contains the updated Virtual DOM and one which is just the pre-update version of this updated Virtual DOM. It compares the pre-update version with the updated Virtual DOM and figures out what exactly has changed in the DOM like which components have been changed. This process of comparing the current Virtual DOM tree with the previous one is known as ‘diffing’. Once React finds out what exactly has changed then it updates those objects only, on real DOM. React uses something called batch updates to update the real DOM. It just means that the changes to the real DOM are sent in batches instead of sending any update for a single change in the state of a component. This entire process of transforming changes to the real DOM is called Reconciliation.

--------------------------------------------------------

# Component

--------------------------------------------------------

**React Component** - takes an optional input and returns a React element which is rendered on the screen.Components let you split the UI into independent, reusable pieces, and think about each piece in isolation

1) Footer.js:

| export default function Footer() {  return <**div**>All rights reserved &copy;</**div**> } |
| --- |

2) App.js:

| function App() {  return (  <**div** className="App">  <**header** className="App-header">  <**img** src={logo} className="App-logo" alt="logo" />  <**h1**>Hello From React</**h1**>  <**Footer** />  </**header**>  </**div**>  ); } |
| --- |

!!! We may pass information to a component via props (the arguments to the component function):  
  
1) App.js:  
  
 <Header text="Hello React" />

2) Header.js:

| import logo from "./logo.svg";  export const Header = (props) => {   return (  <**header** className="App-header">  <**img** src={logo} className="App-logo" alt="logo" />  <**h1**>{props.text}</**h1**>  </**header**>  ); } |
| --- |

| **Element** | **Component** |
| --- | --- |
| An element is always gets returned by a component. | A component can be functional or a class that optionally takes input and returns an element. |
| The element does not have any methods. | Each component has its life cycle methods. |
| A React element is an object representation of a DOM node. | A component encapsulates a DOM tree. |
| Elements are immutable i,e once created cannot be changed. | The state in a component is mutable. |
| An element can be created using React.createElement( ) with type property. | A component can be declared in different ways like it can be an element class with render() method or can be defined as a function. |
| We cannot use React Hooks with elements as elements are immutable. | React hooks can be used with only functional components |
| Elements are light, stateless and hence it is faster. | It is comparatively slower than elements. |

!!!! Names always start with Uppercase; all of the tags must be closed; information is passed through props or state !!!

**Props** - Props stand for "Properties." It is an object which stores the value of attributes of a tag and works similar to the HTML attributes. It gives a way to pass data from one component to other components. It is similar to function arguments. Props are passed to the component in the same way as arguments passed in a function as such we can not change the props inside the child component. Thus, the props are read-only / immutable inside the child component - you can pass props to children components of the child component, but you can not change them. This can only be done in the parent component and when it happens the props are automatically changed / pre-rendered in the child components.

**Props.Children** - an alternative way to set props to a child component; automatically included in props; it is included between the opening and closing tags of a component; set objects, arrays or HTML elements can be more easily

Basic Props Example:  
  
1) App.js:

| import logo from './logo.svg'; import './App.css'; import {Header} from "./components/Header"; import {BookList} from "./components/BookList";  function App() {   const books = [  {  "title": "Northanger Abbey",  "author": "Austen, Jane",  "year\_written": 1814,  "edition": "Penguin",  "price": 18.2  },  {  "title": "War and Peace",  "author": "Tolstoy, Leo",  "year\_written": 1865,  "edition": "Penguin",  "price": 12.7},  {  "title": "Anna Karenina",  "author": "Tolstoy, Leo",  "year\_written": 1875,  "edition": "Penguin",  "price": 13.5},  {  "title": "Mrs. Dalloway",  "author": "Woolf, Virginia",  "year\_written": 1925,  "edition": "Harcourt Brace",  "price": 25  }  ];   return (  <**div** className="App">  <**header** className="App-header">  <**Header**>{{'title': 'BookList'}}</**Header**>  <**BookList** books={books}/>  <**img** src={logo} className="App-logo" alt="logo"/>  </**header**>  </**div**>  ); }  export default App; |
| --- |

2) components/Header.js:

| export const Header = (props) => {   return <**h1**>{props.children.title}</**h1**>;   }; |
| --- |

3) components/BookList.js:

| import {Book} from "./Book";  export const BookList = (props) => {   return (  <**ul**>  <**li**><**Book**  title={props.books[0].title}  author={props.books[0].author}  year={props.books[0].year\_written}  price={props.books[0].price}  />  </**li**>  <**li**><**Book**  title={props.books[1].title}  author={props.books[1].author}  year={props.books[1].year\_written}  price={props.books[1].price}  />  </**li**>  <**li**><**Book**  title={props.books[2].title}  author={props.books[2].author}  year={props.books[2].year\_written}  price={props.books[2].price}  />  </**li**>  <**li**><**Book**  title={props.books[3].title}  author={props.books[3].author}  year={props.books[3].year\_written}  price={props.books[3].price}  />  </**li**>  </**ul**>  );  } |
| --- |

4) components/Book.js:

| export const Book = (props) => {  return (  <**article**>  <**h2**>{props.title}</**h2**>  <**div**>Year: {props.year}</**div**>  <**div**>Price: ${props.price}</**div**>  <**footer**>  <**span**>Author: {props.author}</**span**>  </**footer**>  </**article**>  ); } |
| --- |

## **State and useState hook**

**State**: a built-in React object that is used to contain data or information about the component. A component’s state can change over time; whenever it changes, the component re-renders. The change in state can happen as a response to user action or system-generated events and these changes determine the behavior of the component and how it will render.   
  
**useState**: a hook which allows us to create a state in a functional component, it accepts an initial state and returns two values: the current state and a function that updates the state.

**Basic State Example**:

| import { useState } from "react";  export const Timer = (props) => {   const [time, setTime] = useState(0);   setTimeout(() => {  setTime(time+1);  }, 1000);    return (  <**div**>  <**h2**>Timer: {time} sec.</**h2**>  </**div**>  )  } |
| --- |

!!!! You can have more than a single state per component !!!!  
  
!!! Instead of setX(x + 1) we use setX(oldX => oldX + 1) to avoid race conditions (in this way the current value of x (oldX) will be used) !!!

**React Events** - synthetic events; almost the same as DOM events; they are using camel case (camelCase); we do not need to set an addEventListener - we just need to set the event as an “attribute”, which equals the function which has to be executed upon the trigger of the event

**Basic React Event**:

| import { useState} from "react";  export const Clicker = (props) => {  const [clicks, setClicks] = useState(0);   const clickHandler = (e) => {   setClicks(oldClicks => oldClicks + 1);   }   return (  <**button** onClick={clickHandler}>{clicks}</**button**>  ); } |
| --- |

**Conditional Rendering**:

1) Via ternary operator:

| import { useState} from "react";  export const Clicker = (props) => {  const [clicks, setClicks] = useState(0);   const clickHandler = (e) => {   setClicks(oldClicks => oldClicks + 1);   }   return (  <**div**>  <**h3**>  {clicks > 10   ? <**h2**>Medium Clicks</**h2**>   : <**h3**>Normal Clicks</**h3**>  }  </**h3**>  <**button** onClick={clickHandler}>{clicks}</**button**>  </**div**>  ); } |
| --- |

2) via boolean operators:

| import {useState} from "react";  export const Clicker = (props) => {  const [clicks, setClicks] = useState(0);   const clickHandler = (e) => {   setClicks(oldClicks => oldClicks + 1);   }   const dangerClicks = clicks > 20;   return (  <**div**>  {dangerClicks && <**h1**>Danger Clicks</**h1**>}  {clicks > 10  ? <**h2**>Medium Clicks</**h2**>  : <**h3**>Normal Clicks</**h3**>  }  <**button** onClick={clickHandler}>{clicks}</**button**>  </**div**>  ); } |
| --- |

3) via if statement:

| import {useState} from "react";  export const Clicker = (props) => {  const [clicks, setClicks] = useState(0);   const clickHandler = (e) => {   setClicks(oldClicks => oldClicks + 1);   }   const dangerClicks = clicks > 20;   if (clicks > 30) {  return <**h1**>Finished Clicks!</**h1**>  }   return (  <**div**>  {dangerClicks && <**h1**>Danger Clicks</**h1**>}  {clicks > 10  ? <**h2**>Medium Clicks</**h2**>  : <**h3**>Normal Clicks</**h3**>  }  <**button** onClick={clickHandler}>{clicks}</**button**>  </**div**>  ); } |
| --- |

!!! if vs ternary operator (? : ) - if is a statement and the ternary operator is an expression !!!!

**Fragments**: we can not return two or more HTML elements / children components without a parent element encapsulating them. Still, we may use fragments to resolve this problem. Fragments let you group a list of children without adding extra nodes to the DOM - the fragment (<>) will not be displayed in the HTML:

| return (  <>  <**td**>Hello</**td**>  <**td**>World</**td**>  </>  ); |
| --- |

Fragments declared with the explicit <React.Fragment> syntax may have keys. A use case for this is mapping a collection to an array of fragments — for example, to create a description list.

--------------------------------------------------------

# Lists and Keys

--------------------------------------------------------

**Lists and Keys** - dynamically working with lists / arrays in React requires keys. Keys are used to uniquely identify elements in a list and they help React to identify what is added, updated and changed. Apart from that they assist React to efficiently update the DOM. The keys must be unique, immutable / not changed during the lifecycle of the application (re-rendering must not change the keys - no random id’s as keys) and it is best for them to not be related to the list. Also, it is not recommended to use indexes as keys - if we do and the list is changed (elements removed for example) the keys will be changed without the values being modified. In other words, if the key is an index, reordering an item changes it. Hence, the component state can get mixed up and may use the old key for a different component instance.

**One solution for keys is the id’s of the elements in the database / API from which we acquire them**

**Side Note**: The main difference between map and forEach is that the map method returns a new array by applying the callback function on each element of an array, while the forEach method doesn't return anything.

**Using map() for iterating over list / array:**

1) BookList.js:

| import {Book} from "./Book";  export const BookList = (props) => {   const bookElements = props.books.map(book => <**Book** {...book} />);   return (  <**ul**>  {bookElements}  </**ul**>  )  } |
| --- |

2) Book.js:

| export const Book = (props) => {  return (  <**li**>  <**article**>  <**h2**>{props.title}</**h2**>  <**div**>{props.year}</**div**>  <**div**>{props.price}</**div**>  <**footer**>  <**span**>Author: {props.author}</**span**>  </**footer**>  </**article**>  </**li**>  ) } |
| --- |

**Direct Iteration in JSX:**

| import {Book} from "./Book";  export const BookList = (props) => {   return (  <**ul**>  {props.books.map(book => <**Book** {...book} />)}  </**ul**>  )  } |
| --- |

!!!! **In the above examples we did not use a key** - even though the above code works, React gives out a Warning in the browser’s Console:  
  
**Warning: Each child in a list should have a unique "key" prop.**

!!!!

**Solution with index (we must be certain that the array will not change):**

| import {Book} from "./Book";  export const BookList = (props) => {   return (  <**ul**>  {props.books.map((book, index) => <**Book** key={index} {...book} />)}  </**ul**>  )  } |
| --- |

--------------------------------------------------------

# Debugging React

--------------------------------------------------------

Besides debugging through the IDE we can debug via:

1. debugger in the JS code (as a breakpoint) - the browser will pause and enter debugging mode after the execution goes to debugger::

| const clickHandler = () => {   setHighlighted(state => !state);  debugger;   }; |
| --- |

1. the React DevTools Chrome extension - through the Components tab in the Chrome’s Dev Tools you may see the components’ states, props, hooks and etc

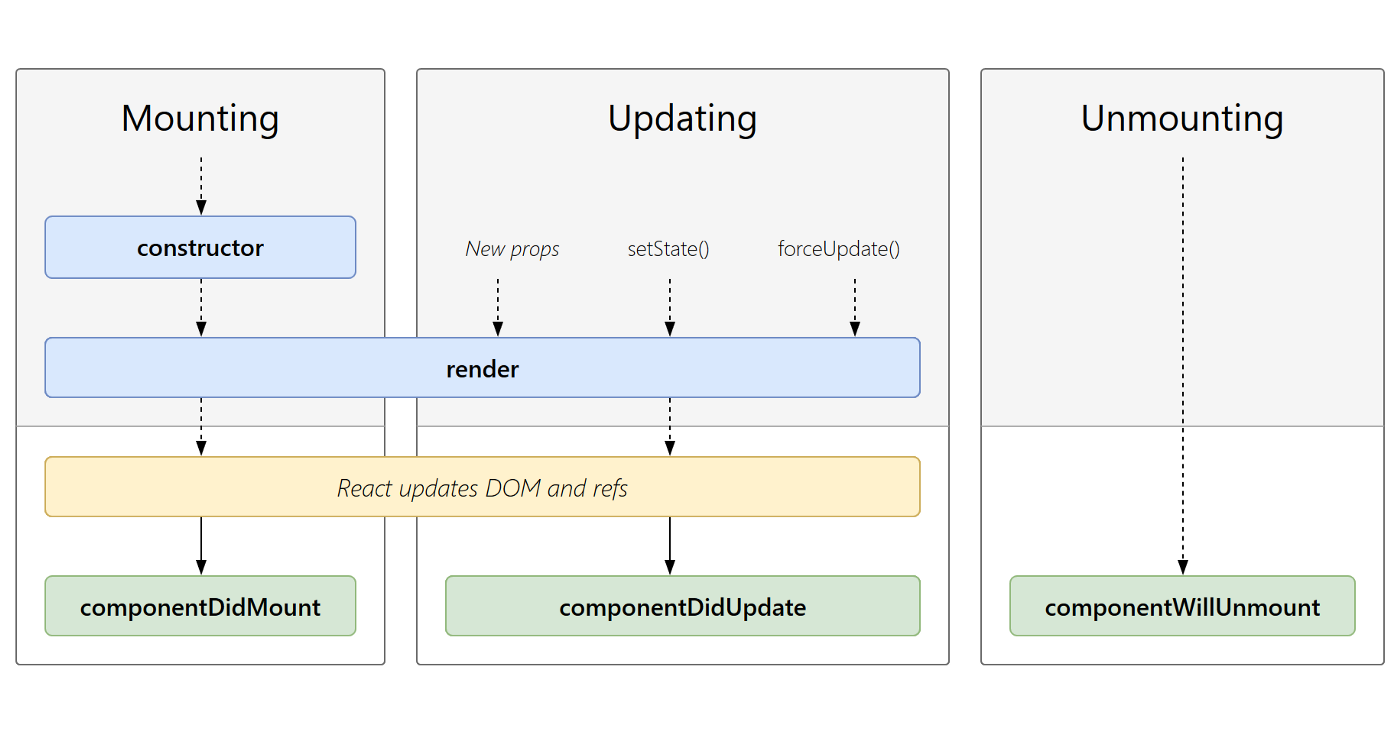
--------------------------------------------------------

# Component Lifecycle

--------------------------------------------------------

After being initialized a React component generally goes through three phases: mounting, updating and unmounting. These three milestones are referred to as the React component lifecycle.

1. Mounting: A component “mounts” when it renders for the first time and it is mounted on the DOM. This is when mounting lifecycle methods get called.
2. Updating: The first time that a component instance renders, it does not update. Starting with the second render, a component updates every time that it renders.
3. Unmounting: A component’s unmounting period occurs when the component is removed from the DOM. This could happen if the DOM is rerendered without the component, or if the user navigates to a different website or closes their web browser.



!!!! A React Component may or may not go through all phases. Sometimes they are never updated. Other times they are never unmounted. A component can even go through the mounting phase and unmounting phase back to back without ever updating. !!!!

React has certain lifecycle-related methods for each of the mentioned phases:

**Mounting:**

1) DEPRECATED **componentWillMount**() Function: As the name clearly suggests, this function is invoked right before the component is mounted on the DOM i.e. this function gets invoked once before the render() function is executed for the first time.

2) **componentDidMount**() Function: Similarly as the previous one this function is invoked right after the component is mounted on the DOM i.e. this function gets invoked once after the render() function is executed for the first time.

**Updating:**

3) DEPRECATED **componentWillReceiveProps**() Function: This is a Props exclusive Function and is independent of States. This function is invoked before a mounted component gets its props reassigned. The function is passed the new set of Props which may or may not be identical to the original Props. Thus checking is a mandatory step in this regard.

4) **setState**() Function: This is not particularly a Lifecycle function and can be invoked explicitly at any instant. This function is used to update the state of a component.

5) **shouldComponentUpdate**() Function: By default, every state or props update re-render the page but this may not always be the desired outcome, sometimes it is desired that updating the page will not be repainted. The shouldComponentUpdate() Function fulfills the requirement by letting React know whether the component’s output will be affected by the update or not. shouldComponentUpdate() is invoked before rendering an already mounted component when new props or state are being received. If returned false then the subsequent steps of rendering will not be carried out. This function can’t be used in the case of forceUpdate(). The Function takes the new Props and new State as the arguments and returns whether to re-render or not.

6) DEPRECATED **componentWillUpdate**() Function: As the name clearly suggests, this function is invoked before the component is rerendered i.e. this function gets invoked once before the render() function is executed after the updation of State or Props.

7) **componentDidUpdate**() Function: Similarly this function is invoked after the component is rerendered i.e. this function gets invoked once after the render() function is executed after the updation of State or Props.

**Unmounting:**  
  
9) **componentWillUnmount**() Function: This function is invoked before the component is finally unmounted from the DOM i.e. this function gets invoked once before the component is removed from the page and this denotes the end of the lifecycle.

--------------------------------------------------------

# useEffect Hook

--------------------------------------------------------

**useEffect** is a tool that lets us interact with the outside world / side effects, but not affect the rendering or performance of the component that it's in; a way to handle performing side effects in what are otherwise pure React components; even simpler - by using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we’ll refer to it as our “effect”), and call it later after performing the DOM updates. Most notably used for fetching data from remote servers, subscriptions and manual editing of the HTML DOM.

!!!!! In the functional development we can use the useEffect hook to access / use / be an equivalent to certain lifecycle methods:

componentDidMount  
shouldComponentUpdate  
componentWillUnmount

componentDidUpdate !!!

!!! the useEffect will always be executed at mounting and then depending on its dependency array !!!

**Basic useEffect example:**

| import {useState} from "react"; import {useEffect} from "react";  export const Book = (props) => {   const [highlighted, setHighlighted] = useState(false);   useEffect(() => {  console.log('Updating highlighted at ' + props.title);  },[highlighted]); //the useEffect will be executed at the initial render  // and then only after the highlighted state is changed   const clickHandler = () => {   setHighlighted(state => !state);   };   let style = {};   if (highlighted) {  style.backgroundColor = 'blue';  }   return (  <**li** onClick={clickHandler} style={style}>  <**article**>  <**h2**>{props.title}</**h2**>  <**div**>Year: {props.year}</**div**>  <**div**>Price: ${props.price}</**div**>  <**footer**>  <**span**>Author: {props.author}</**span**>  </**footer**>  </**article**>  </**li**>  ) }; |
| --- |

**to run the useEffect only once** - dependency array is empty:

| useEffect(() => {  console.log('Mounting');  },[]); |
| --- |

--------------------------------------------------------

# Fetch API

--------------------------------------------------------

The **Fetch API** through the fetch() method allows us to make an HTTP request to the backend. With this method, we can perform different types of operations using HTTP methods like the GET method to request data from an endpoint, POST to send data to an endpoint, and more. It uses promises and it can be used along with async/await with it as well. In a component we can use useEffect to execute the fetch a single time. Otherwise, if we use fetch plus state we will face a loop where the fetch will force the state to rerender, which in turn will force the fetch to rerender.

<https://swapi.dev/>

**Basic Fetch API Example:**

| import {useEffect, useState} from "react";  export const CharacterList = () => {   const [characters , setCharacters] = useState([]);   useEffect(() => {  fetch(`https://swapi.dev/api/people`)  .then(res => res.json())  .then(result => {  setCharacters(result.results);  });  }, []);   return (  <**ul**>  {!characters.length && <**li**>Loading...</**li**>}  {characters.map(x => (  <**li** key={x.name}>{x.name}</**li**>  ))}  </**ul**>  );  }; |
| --- |

--------------------------------------------------------

# CSS in React

--------------------------------------------------------

!!!! We can change the inline CSS of a HTML element by providing an object (with camelCase formatted properties) to the style attribute / tag:



| const style = {};  if (highlighted) {  style.backgroundColor = 'blue';  }   return (  <**li** style={style}> |
| --- |

!!!! Also, we can use ternary operator within the style object:

| <**li** style={{backgroundColor: highlighted ? 'blue' : 'red'}}> |
| --- |

!!!!

**Other ways to change the CSS of an element**:   
  
1) Through a CSS file, imported in the component, preferably named the same way as the component. During building the application, however, all of the CSS will be set in one place, therefore changing the CSS in a single file can affect the whole application:

| import './Book.css';   <**article** className='book-item'> |
| --- |

2) Through third-party solutions such as JSS, styled-components, etc.

3) (Recommended Way) - through CSS modules - a CSS module is a simple CSS file with the key difference being that by default every class name and animation inside the CSS module is scoped locally to the component that is importing it. Therefore, the changes made in the module should not affect the whole application. CSS file name should follow the format ‘filename.module.css’. Finally, we can assign multiple classes to the same element:

| import styles from './Book.module.css';  <**article** className={`${styles['book-item']} ${styles['other-class']}`}> |
| --- |

!!!! The classes in the CSS module use automatically assigned unique names, which is why the changes made in the CSS module to a specific class will only affect the component where the module has been imported:

| <**article** class="Book\_book-item\_\_xEFXN"> |
| --- |

!!!!

**Side Note:** It is not advisable to use IDs as CSS selectors because if another element in the page uses the same/similar style, you would have to write the same CSS again. Even if you don't have more than one element with that style right now, it might come later. Hence it is always advisable to use class.

--------------------------------------------------------

# Passing Information from Child to Parent

--------------------------------------------------------

1) ToDoList.js:

| import {useEffect, useState} from "react"; import {ToDoItem} from "./ToDoItem";  export const ToDoList = () => {   const [todos, setTodos] = useState([]);   useEffect(() => {   fetch('http://localhost:3030/jsonstore/todos')  .then(res => res.json())  .then(result => {  setTodos(Object.values(result));  });    },[]);   const todoClickHandler = (todo) => {   fetch(`http://localhost:3030/jsonstore/todos/${todo.\_id}`, {  method: 'PUT',  headers: {  'content-type': 'application/json'  },  body: JSON.stringify({isCompleted: !todo.isCompleted})  })  .then(res => res.json())  .then(modifiedTodo => {  setTodos(oldTodos => oldTodos.map(x => x.\_id === todo.\_id ? modifiedTodo : x));  });  };   return (  <**table** className="table">  <**thead**>  <**tr**>  <**th** className="table-header-task">Task</**th**>  <**th** className="table-header-status">Status</**th**>  <**th** className="table-header-action">Action</**th**>  </**tr**>  </**thead**>  <**tbody**>  {todos.map((todo) => <**ToDoItem** key={todo.\_id} {...todo} onClick={todoClickHandler}/>)}  </**tbody**>  </**table**>  );  }; |
| --- |

2) ToDoItem.js:

| export const ToDoItem = (props) => {   let className = 'todo';   if (props.isCompleted) {  className += ' is-completed';  }   return (  <**tr** className={className}>  <**td**>{props.text}</**td**>  <**td**>{props.isCompleted ? 'Complete' : 'Incomplete' }</**td**>  <**td** className="todo-action">  <**button** onClick={() => props.onClick(props)} className="btn todo-btn">Change status</**button**>  </**td**>  </**tr**>  );  }; |
| --- |

!!! Whenever the **Change Status** button inside the **ToDoItem** (child) is clicked, it will execute the passed through props **todoClickHandler** function in the ToDoList (parent) with the specific props for the child component (the information we want to be passed to the parent).

In other words we may pass information from the child component to the parent through a function defined in the parent component and passed to the child component through props, which function accepts as argument the info we want to pass from the child to the parent  
  
Shorter: we pass a callback function from the parent to child !!!

**Side Note**: PUT is a method of modifying a resource where the client sends data that updates the entire resource . PATCH is a method of modifying resources where the client sends partial data that is to be updated without modifying the entire data.

!!! If we need to pass information between sibling components (the same parent component) we use a “lifted state”, which is kept in the parent component !!!

--------------------------------------------------------

# Forms in React

--------------------------------------------------------

Inputs in React can be one of two types: **controlled** or **uncontrolled**.

1. The uncontrolled input is similar to the traditional HTML form inputs. The HTML DOM itself handles the form data. In the DOM, the HTML elements maintain their own state that will be updated when the input value changes.
2. With a controlled input, YOU explicitly control the value that the input displays. You have to write code to respond to keypresses, store the current value somewhere (state), and pass that value back to the input to be displayed. It’s a feedback loop with your code in the middle. In this way we may have dynamic values in the inputs, which will be changed per changes in the app. As we rely on React in this case it is the more recommended option.

**Basic Example of How to Get Values from Uncontrolled Forms:**

| import './App.css';  function App() {   const submitHandler = (e) => {   e.preventDefault();   const formData = new FormData(e.currentTarget);  const username = formData.get('username');  const password = formData.get('password');   }   return (  <**div** className="App">  <**header** className="App-header">  <**form** onSubmit={submitHandler}>  <**div**>  <**label** htmlFor="username">Username: </**label**>  <**input** id="username" type="text" name="username"/>  </**div**>  <**div**>  <**label** htmlFor="password">Password: </**label**>  <**input** id="password" type="password" name="password"/>  </**div**>  <**button**>Login</**button**>  </**form**>   </**header**>  </**div**>  ); }  export default App; |
| --- |

**Another example of uncontrolled input with onChange plus defaultValue:**

| import logo from './logo.svg'; import './App.css'; import {useState} from "react";  function App() {   const [username, setUsername] = useState('');   const submitHandler = (e) => {   e.preventDefault();    }   const usernameChangeHandler = (e) => {  const username = e.target.value;  }   return (  <**div** className="App">  <**header** className="App-header">  <**form** onSubmit={submitHandler}>  <**div**>  <**label** htmlFor="username">Username: </**label**>  <**input** id="username" type="text" name="username" defaultValue='Gosho' onChange={usernameChangeHandler}/>  </**div**>  <**div**>  <**label** htmlFor="password">Password: </**label**>  <**input** id="password" type="password" name="password"/>  </**div**>  <**button**>Login</**button**>  </**form**>   </**header**>  </**div**>  ); }  export default App; |
| --- |

**Basic Example of Getting Values from Controlled Form:**

| import logo from './logo.svg'; import './App.css'; import {useState} from "react";  function App() {   const [username, setUsername] = useState('');   const submitHandler = (e) => {   e.preventDefault();    }   const usernameChangeHandler = (e) => {  setUsername(e.target.value);  }   return (  <**div** className="App">  <**header** className="App-header">  <**form** onSubmit={submitHandler}>  <**div**>  <**label** htmlFor="username">Username: </**label**>  <**input** id="username" type="text" name="username" value={username} onChange={usernameChangeHandler}/>  </**div**>  <**div**>  <**label** htmlFor="password">Password: </**label**>  <**input** id="password" type="password" name="password"/>  </**div**>  <**button**>Login</**button**>  </**form**>   </**header**>  </**div**>  ); }  export default App; |
| --- |

**Getting different types of inputs within a controlled form with dynamically available Register button based on Terms of Conditions:**

| import './App.css'; import {useState} from "react";  function App() {   const [values, setValues] = useState({  username: '',  password: '',  age: '',  bio: '',  gender: 'o',  userType: 'corporate',  tac: false   });   const changeHandler = (e) => {  setValues(state => ({  ...state,  [e.target.name]: e.target.type === 'checkbox' ? e.target.checked : e.target.value  }));  };    const submitHandler = (e) => {   e.preventDefault();   }   return (  <**div** className="App">  <**header** className="App-header">  <**form** onSubmit={submitHandler}>  <**div**>  <**label** htmlFor="username">Username: </**label**>  <**input** id="username" type="text" name="username" value={values.username} onChange={changeHandler}/>  </**div**>   <**div**>  <**label** htmlFor="password">Password: </**label**>  <**input** id="password" type="password" name="password" value={values.password} onChange={changeHandler}/>  </**div**>   <**div**>  <**label** htmlFor="age">Age: </**label**>  <**input** id="number" type="age" name="age" value={values.age} onChange={changeHandler}/>  </**div**>   <**div**>  <**label** htmlFor="bio">Bio: </**label**>  <**textarea** name="bio" id="bio" cols="30" rows="10" value={values.bio} onChange={changeHandler}></**textarea**>  </**div**>   <**div**>  <**label** htmlFor="gender">Gender: </**label**>  <**select** name="gender" id="gender" value={values.gender} onChange={changeHandler}>  <**option** value="m">Male</**option**>  <**option** value="f">Female</**option**>  <**option** value="o">Other</**option**>  </**select**>  </**div**>   <**div**>  <**label** htmlFor="individual-user-type">Individual: </**label**>  <**input** type="radio" name="userType" value="individual" id="individual-user-type" checked={values.userType === 'individual'} onChange={changeHandler}/>  <**label** htmlFor="individual-user-type">Corporate: </**label**>  <**input** type="radio" name="userType" value="corporate" id="corporate-user-type" checked={values.userType === 'corporate'} onChange={changeHandler}/>  </**div**>   <**div**>  <**label** htmlFor="tac">Terms and Conditions:</**label**>  <**input** type="checkbox" name="tac" id="tac" checked={values.tac} onChange={changeHandler}/>  </**div**>   <**button** disabled={!values.tac}>Register</**button**>  </**form**>   </**header**>  </**div**>  ); }  export default App; |
| --- |

!!!! For a controlled input you need value and onChange attributes, which are connected to the state related to the input !!!!

## **Basic Input Validation in a Controlled Form:**

1) create boolean state - if the input validation faces error / fails  
  
2) create a handler for onBlur event on the input   
  
3) set the onBlur attribute with the handler on the input  
  
4) set conditional rendering for the error message if the validation fails

| const [firstNameHasError, setFirstNameHasError] = useState(false); |
| --- |

| const [values, setValues] = useState({  firstName: '',  lastName: '',  email: '',  phoneNumber: '',  imageUrl: '',  country: '',  city: '',  street: '',  streetNumber: ''  }); |
| --- |

| const validateFirstName = (e) => {   if (values.firstName.length < 3) {   setFirstNameHasError(true);   } else {  setFirstNameHasError(false);  }   }; |
| --- |

| <**label** htmlFor="firstName">First name</**label**> <**div** className="input-wrapper">  <**span**><**i** className="fa-solid fa-user"></**i**></**span**>  <**input** id="firstName" name="firstName" type="text" value={values.firstName} onChange={changeHandler} onBlur={validateFirstName}/> </**div**> {firstNameHasError && <**p** className="form-error">  First name should be at least 3 characters long! </**p**> } |
| --- |

**Abstract Validation Example:**

| const [errors, setErrors] = useState({}); |
| --- |

| const [values, setValues] = useState({  firstName: '',  lastName: '',  email: '',  phoneNumber: '',  imageUrl: '',  country: '',  city: '',  street: '',  streetNumber: ''  }); |
| --- |

| const minLength = (e, lowerLimit) => {   setErrors(oldValues => ({  ...oldValues,  [e.target.name]: values[e.target.name].length < lowerLimit  }));   }; |
| --- |

| <**label** htmlFor="firstName">First name</**label**> <**div** className="input-wrapper">  <**span**><**i** className="fa-solid fa-user"></**i**></**span**>  <**input** id="firstName" name="firstName" type="text" value={values.firstName}  onChange={changeHandler} onBlur={(e) => minLength(e, 3)}/> </**div**> {  errors.firstName &&  <**p** className="form-error">  First name should be at least 3 characters long!  </**p**> } |
| --- |

| <**input** id="country" name="country" type="text" value={values.country}  onChange={changeHandler} onChange={changeHandler}  onBlur={(e) => minLength(e, 2)}/> </**div**> { errors.country && <**p** className="form-error">  Country should be at least 2 characters long! </**p**> } |
| --- |

--------------------------------------------------------

# useRef Hook

--------------------------------------------------------

A built-in React hook that accepts one argument as the initial value and returns a reference (aka ref). Through ref we can access the DOM elements and the React elements that we have created on our own. The reference is an object which has a property -> current. The reference.current accesses the reference value, and reference.current = newValue updates the reference value. The useRef can be used to directly access DOM nodes (via the ref attribute), as well as persist a mutable value across rerenders of a component.

There are 3 rules to remember about references:

1. The value of the reference is persisted (stays the same) between component re-renderings;
2. Updating a reference doesn't trigger a component re-rendering.
3. ​​Updating a ref is a side effect so it should be done only inside a useEffect (or useLayoutEffect) or inside an event handler.

!!!! You should avoid using reference calls as much as possible. There are only 3 good reasons why you’d need to use the useRef hook.

1. Managing focus, text selection, or media playback
2. Triggering imperative animations
3. Integrating with third-party DOM libraries

!!!!

!!! useRef is like class instance variable for function components. !!!!

**Basic Example of useRef - when the username and age change, they will appear in the RefInput:**

| import './App.css'; import {useEffect, useRef, useState} from "react";  function App() {   const infoRef = useRef();   const [values, setValues] = useState({  username: '',  password: '',  age: '',  bio: '',  gender: 'o',  userType: 'corporate',  tac: false   });   useEffect(() => {   if(values.username && values.age) {   infoRef.current.value = `${values.username} - ${values.age}`;   }   },[values.username, values.age]) ...........  <**div**>  <**label** htmlFor="refInput">Ref Input</**label**>  <**input** name="refInput" type="text" ref={infoRef}/>  </**div**> |
| --- |

--------------------------------------------------------

# Routing in React

--------------------------------------------------------

--------------------------------------------------------

# Useful Links

--------------------------------------------------------

<https://github.com/public-apis/public-apis>

<https://www.geeksforgeeks.org/reactjs-lifecycle-components/>

<https://learnreact.design/posts/react-useref-by-example>

Sandbox Environment: <https://codesandbox.io/>