React + TypeScript

Part 1: Fundamentals

Hi! I am Mosh Hamedani. I'm a software engineer with over 20 years of experience and I've taught millions of people how to code and become professional software engineers through my YouTube channel and coding school (Code with Mosh).

This PDF is part of my **Ultimate React course** where you will learn everything you need to know from the absolute basics to more advanced concepts. You can find the full course on my website.

https://codewithmosh.com

https://www.youtube.com/c/programmingwithmosh

https://twitter.com/moshhamedani

https://www.facebook.com/programmingwithmosh/

Table of Content

Getting Started……………………………………………………………………..4

Building Components……………………………………………………………..6

Styling Components………..……………………………………………………….10

Managing Component State……………..……………………………………….13

Building Forms……………………………………………………………………….17

Connecting to the Backend……………………………………………………….22

Getting Started

Terms

Components	JSX
JavaScript Framework	DOM
JavaScript Library	Virtual DOM

Summary

- React is a JavaScript library for building dynamic and interactive user interfaces.
- In React applications, we don’t query and update the DOM. Instead, we describe our application using small, reusable components. React will take care of efficiently creating and updating DOM elements.
- React components can be created using a function or a class. Function-based components are the preferred approach as they’re more concise and easier to work with.
- JSX stands for JavaScript XML. It is a syntax that allows us to write components that combine HTML and JavaScript in a readable and expressive way, making it easier to create complex user interfaces.
- When our application starts, React takes a tree of components and builds a JavaScript data structure called the virtual DOM. This virtual DOM is different from the actual DOM in the browser. It’s a lightweight, in-memory representation of our component tree.

- When the state or the data of a component changes, React updates the corresponding node in the virtual DOM to reflect the new state. Then, it compares the current version of virtual DOM with the previous version to identify the nodes that should be updated. It'll then update those nodes in the actual DOM.

- In browser-based apps, updating the DOM is done by a companion library called ReactDOM. In mobile apps, React Native uses native components to render the user interface.

- Since React is just a library and not a framework like Angular or Vue, we often need other tools for concerns such as routing, state management, internationalization, form validation, etc.

Building Components

Terms

Fragment
Immutable
Props
State hook

Summary

- In React apps, a component can only return a single element. To return multiple elements, we wrap them in a fragment, which is represented by empty angle brackets.

- To render a list in JSX, we use the 'array.map()' method. When mapping items, each item must have a unique key, which can be a string or a number.

- To conditionally render content, we can use an 'if' statement or a ternary operator.

- We use the state hook to define state (data that can change over time) in a component. A hook is a function that allows us to tap into built-in features in React.

- Components can optionally have props (short for properties) to accept input.

- We can pass data and functions to a component using props. Functions are used to notify the parent (consumer) of a component about certain events that occur in the component, such as an item being clicked or selected.

- We should treat props as immutable (read-only) and not modify them.

- When the state or props of a component change, React will re-render the component and update the DOM accordingly.

- In React apps, a component can only return a single element. To return multiple elements, we wrap them in a fragment, which is represented by empty angle brackets.
- To render a list in JSX, we use the 'array.map()' method. When mapping items, each item must have a unique key, which can be a string or a number.
- To conditionally render content, we can use an 'if' statement or a ternary operator.
- We use the state hook to define state (data that can change over time) in a component. A hook is a function that allows us to tap into built-in features in React.
- Components can optionally have props (short for properties) to accept input.
- We can pass data and functions to a component using props. Functions are used to notify the parent (consumer) of a component about certain events that occur in the component, such as an item being clicked or selected.
- We should treat props as immutable (read-only) and not modify them.
- When the state or props of a component change, React will re-render the component and update the DOM accordingly.

CREATING A COMPONENT

```
const Message = () => {
  return <h1>Hello World</h1>;
}

export default Message;
```

RENDERING A LIST

```
const Component = () => {
  const items = ['a', 'b', 'c'];
  return (
    <ul>
      {items.map((item) => (
        <li key={item}>item</li>
      ))}
    </ul>
  );
};
```

CONDITIONAL RENDERING

```
{items.length === 0 ? 'a' : 'b'}
{items.length === 0 && 'a'}
```

HANDLING EVENTS

```
<button onClick={() => console.log('clicked')}></button>;
```

DEFINING STATE

```
const [name, setName] = useState('');
```

PROPS

```
interface Props {
  name: string;
}

const Component = ({ name }: Props) => {
  return <p>{name}</p>
};
```

PASSING CHILDREN

```
interface Props {
  children: ReactNode
}

const Component = ({ children }: Props) => {
  return <div>{children}</div>
};
```

Styling Components

Terms

CSS-in-JS	Inline styles
CSS modules	Modular
Implementation details	Separation of concerns
Interface	Vanilla CSS

Summary

- We have several options for styling React components, including vanilla CSS, CSS modules, CSS-in-JS, and inline styles.

- With vanilla CSS, we write our component styles in a separate CSS file and import it into the component file. However, we may encounter conflicts if the same CSS classes are defined in multiple files.

- CSS modules resolve this issue by generating unique class names during the build process.

- With CSS-in-JS, we define all the styles for a component alongside its code. Like CSS modules, this provides scoping for CSS classes and eliminates conflicts. It also makes it easier for us to change or delete a component without affecting other components.

- The separation of concerns principle suggests that we divide a program into distinct sections or modules where each section handles a specific functionality. It helps us build modular and maintainable applications.

- With this principle, the complexity and implementation details of a module are hidden behind a well-defined interface.

- Separation of concerns is not just about organizing code into files, but rather dividing areas of functionality. Therefore, CSS-in-JS does not violate the separation of concerns principle as all the complexity for a component remains hidden behind its interface.

- Although inline styles are easy to apply, they can make our code difficult to maintain over time and should only be used as a last resort.

- We can add icons to our applications using the react-icons library.

- There are several UI libraries available that can assist us in quickly building beautiful and modern applications. Some popular options include Bootstrap, Material UI, TailwindCSS, DaisyUI, ChakraUI, and more.

VANILLA CSS

```
import './ListGroup.css';

function ListGroup() {
  return <ul className="list-group"></ul>;
}
```

CSS MODULES

```
import styles from './ListGroup.module.css';

function ListGroup() {
  return <ul className={styles.listGroup}></ul>;
}
```

CSS-IN-JS

```
import styled from 'styled-components';

const List = styled.ul`
  list-style: none;
`;

function ListGroup() {
  return <List></List>;
}
```

Managing Component State

Terms

Asynchronous
Lifting state
Pure component
Strict mode

Summary

- The state hook allows us to add state to function components.
- Hooks can only be called at the top level of components.
- State variables, unlike local variables in a function, stay in memory as long as the component is visible on the screen. This is because state is tied to the component instance, and React will destroy the component and its state when it is removed from the screen.
- React updates state in an asynchronous manner, so updates are not applied immediately. Instead, they're batched and applied at once after all event handlers have finished execution. Once the state is updated, React re-renders our component.
- Group related state variables into an object to keep them organized.
- Avoid deeply nested state objects as they can be hard to update and maintain.
- To keep state as minimal as possible, avoid redundant state variables that can be computed from existing variables.
- A pure function is one that always returns the same result given the same input. Pure functions should not modify objects outside of the function.

- React expects our function components to be pure. A pure component should always return the same JSX given the same input.

- To keep our components pure, we should avoid making changes during the render phase.

- Strict mode helps us catch potential problems such as impure components. Starting from React 18, it is enabled by default. It renders our components twice in development mode to detect any potential side effects.

- When updating objects or arrays, we should treat them as immutable objects. Instead of mutating them, we should create new objects or arrays to update the state.

- Immer is a library that can help us update objects and arrays in a more concise and mutable way.

- To share state between components, we should lift the state up to the closest parent component and pass it down as props to child components.

- The component that holds some state should be the one that updates it. If a child component needs to update some state, it should notify the parent component using a callback function passed down as a prop.

UPDATING OBJECTS

```
const [drink, setDrink] = useState({
  title: 'Americano',
  price: 5
});

setDrink({ ...drink, price: 2 });
```

UPDATING NESTED OBJECTS

```
const [customer, setCustomer] = useState({
  name: 'John',
  address: {
    city: 'San Francisco',
    zipCode: 94111
  }
});

setCustomer({
  ...customer,
  address: { ...customer.address, zipCode: 94112 },
});
```

UPDATING ARRAYS

```
const [tags, setTags] = useState(['a', 'b']);

// Adding
setTags([...tags, 'c']);

// Removing
setTags(tags.filter(tag => tag !== 'a'));

// Updating
setTags(tags.map(tag => tag === 'a' ? 'A' : tag));
```

UPDATING ARRAY OF OBJECTS

```
const [bugs, setBugs] = useState([
  { id: 1, title: 'Bug 1', fixed: false },
  { id: 2, title: 'Bug 2', fixed: false },
]);

setBugs(bugs.map(bug =>
          bug.id === 1 ? { ...bug, fixed: true } : bug));
```

UPDATING WITH IMMER

```
import produce from 'immer';

setBugs(produce(draft => {
  const bug = draft.find(bug => bug.id === 1);
  if (bug) bug.fixed = true;
}));
```

Building Forms

Terms

React Hook Form
Ref hook
Schema-based validation libraries
Zod

Summary

- To handle form submissions, we set the onSubmit attribute of the form element.
- We can use the ref hook to access elements in the DOM. This technique is often used to read the value of input fields upon submitting a form.
- We can also use the state hook to create state variables and update them as the user types into input fields. With this technique, every time the user types a character into an input field, the component containing the form gets re-rendered. While in theory this can cause a performance penalty, in practice this is often negligible.
- React Hook Form is a popular library that helps us build forms quickly with less code. With React Hook Form, we no longer have to worry about using the ref or state hooks to manage the form state.
- React Hook Form supports the standard HTML attributes for data validation such as required, minLength, etc.
- We can validate our forms using schema-based validation libraries such as joi, yup, zod, etc. With these libraries, we can define all our validation rules in a single place called a schema.

HANDLING FORM SUBMISSION

```
const App = () => {
  const handleSubmit = (event: FormEvent) => {
    event.preventDefault();
    console.log('Submitted');
  };

  return (
    <form onSubmit={handleSubmit}>
    </form>
  );
};
```

ACCESSING INPUT FIELDS USING THE REF HOOK

```
const App = () => {
  const nameRef = useRef<HTMLInputElement>(null);

  const handleSubmit = (event: FormEvent) => {
    event.preventDefault();

    if (nameRef.current)
      console.log(nameRef.current.value);
  };

  return (
    <form onSubmit={handleSubmit}>
      <input ref={nameRef} type="text" />
    </form>
  );
};
```

MANAGING FORM STATE USING THE STATE HOOK

```
const App = () => {
  const [name, setName] = useState('');

  return (
    <form>
      <input
        type="text"
        value={name}
        onChange={(event) => setName(event.target.value)}
      />
    </form>
  );
};
```

MANAGING FORM STATE USING REACT HOOK FORM

```
import { FieldValues, useForm } from 'react-hook-form';

const App = () => {
  const { register, handleSubmit } = useForm();

  const onSubmit = (data: FieldValues) => {
    console.log('Submitting the form', data);
  }

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} type="text" />
    </form>
  );
};
```

VALIDATION USING HTML5 ATTRIBUTES

```
const App = () => {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm<FormData>();

  const onSubmit = (data: FieldValues) => {
    console.log('Submitting the form', data);
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name', { required: true })} type="text" />
      {errors.name?.type === 'required' && <p>Name is required.</p>}
    </form>
  );
};
```

DISABLING THE SUBMIT BUTTON

```
const App = () => {
  const {
    formState: { isValid },
  } = useForm<FormData>();

  return (
    <form>
      <button disabled={!isValid}>Submit</button>
    </form>
  );
};
```

SCHEMA-BASED VALIDATION WITH ZOD

```
import { FieldValues, useForm } from 'react-hook-form';
import { z } from 'zod';
import { zodResolver } from '@hookform/resolvers/zod';

const schema = z.object({
  name: z.string().min(3),
});

type FormData = z.infer<typeof schema>;

const App = () => {
  const {
    register,
    handleSubmit,
    formState: { errors },
  } = useForm<FormData>({ resolver: zodResolver(schema) });

  const onSubmit = (data: FieldValues) => {
    console.log('Submitting the form', data);
  };

  return (
    <form onSubmit={handleSubmit(onSubmit)}>
      <input {...register('name')} type="text" />
      {errors.name && <p>{errors.name.message}</p>}
    </form>
  );
};
```

Connecting to the Backend

Terms

Axios	HTTP
Back-end	HTTP request
Effect hook	HTTP response
Front-end	Side effects

Summary

- We use the effect hook to perform side effects, such as fetching data or updating the DOM.
- The effect hook takes a function that performs the side effect and an optional array of dependencies. Whenever the dependencies change, the effect hook runs again.
- To clean up any resources that were created by the effect hook, we can include a clean-up function that runs when the component unmounts or the dependencies change.
- React is a library for building front-end user interfaces, but to create complete apps, we also need a back-end server to handle business logic, data storage, and other functionality.
- The communication between the front-end and the back-end happens over HTTP, the same protocol that powers the web. The front-end sends an HTTP request to the back-end, and the back-end sends an HTTP response back.
- Each HTTP request and response contains a header and a body. The header provides metadata about the message, such as the content type and HTTP status code, while the body contains the actual data being sent or received.

- To send HTTP requests to the backend, we can use axios, a popular JavaScript library. axios makes it easy to send requests.
- When we send HTTP requests with the effect hook, we should provide a clean-up function to cancel the request if the component is unmounted before the response is received. This is important to prevent errors, especially if the user navigates to a different page while the request is still pending.
- When sending HTTP requests, we must handle errors properly. This can be done using try-catch blocks or by handling the error in the promise chain using .catch().
- Custom hooks are a way to reuse code logic between multiple components. By encapsulating logic in a custom hook, we can create reusable pieces of code that can be shared across components without duplicating the code. Custom hooks can be used to handle common tasks, such as fetching data, and can help to make our code more organized and easier to maintain.

USING THE EFFECT HOOK

```
function App() {
  useEffect(() => {
    document.title = 'App';
  }, []);
}
```

FETCHING DATA WITH AXIOS

```
const [users, setUsers] = useState<User[]>([]);

useEffect(() => {
  axios.get<User[]>('http://...')
    .then((res) => setUsers(res.data));
}, []);
```

HANDLING ERRORS

```
const [error, setError] = useState('');

useEffect(() => {
  axios.get<User[]>('http://...')
    .then((res) => setUsers(res.data))
    .catch(err => setError(err.message));
}, []);
```

CANCELLING AN HTTP REQUEST

```
useEffect(() => {
  const controller = new AbortController();

  axios.get<User[]>('http://...', { signal: controller.signal })
    .then((res) => setUsers(res.data))
    .catch(err => {
      if (err instanceof CanceledError) return;
      setError(err.message)
    });

  return () => controller.abort();
}, []);
```

DELETING DATA

```
axios.delete('http://...')
```

CREATING DATA

```
axios.post('http://...', newUser)
```

UPDATING DATA

```
axios.put('http://...', updatedUser)
```