Bonus: TypeScript with React

TypeScript is a powerful tool for bringing type safety and developer productivity to JavaScript codebases. It works seamlessly with JSX, and by including the type definitions for React (`@types/react`) and ReactDOM (`@types/react-dom`), you get full support for React in a TypeScript environment.

Adding TypeScript to an Existing React Project

To install the type definitions for React and ReactDOM, run the following command in your terminal:

```
npm install @types/react @types/react-dom
```

Then, in your `tsconfig.json`, make sure you have the following options:

- Include "`dom`" in the "`lib`" array to ensure DOM support (this is typically the default behavior).
- Set the `jsx` option to either "`react-jsx`" or "`react`" depending on your React version. If you're working with a library, check the React documentation for the appropriate option.

TypeScript in React Components

When working with React and TypeScript, every file containing JSX should use the `.tsx` extension. This tells TypeScript that the file includes JSX syntax, enabling type-checking for React components.

A key feature is the ability to define types for a component's props. This ensures correctness and provides better documentation in your editor. Let's modify the `MyButton` component example by adding type annotations for its `title` prop:

```
function MyButton({ title }: { title: string }) {
  return <button>{title}</button>;
}

export default function App() {
  return (
    <div>
```

```
      <h1>Hello, React with TypeScript!</h1>
      <MyButton title="Click me!" />
    </div>
  );
}
```

This example shows how you can type props inline. However, for more complex props or to improve readability, you can define types using `interface` or `type`. For a more in-depth explanation and examples, please refer to the Working with React Props and TypeScript lesson.

Working with React Props and TypeScript

One of the core features of React is passing props to components, and TypeScript allows you to explicitly define the types of those props. This ensures the values passed into components are of the expected types, avoiding common bugs. You can define the shape of the props using either interface or type. Both approaches achieve the same outcome, but each has slight differences and personal preference plays a role in which to choose.

Using `interface` for Props

An `interface` is a natural way to define an object structure in TypeScript. It's often used for larger applications because interfaces are extendable, meaning you can add to them later or combine them with others.

```
interface ButtonProps {
  label: string;
  disabled?: boolean; // Optional prop
}

function MyButton({ label, disabled = false }: ButtonProps) {
  return <button disabled={disabled}>{label}</button>;
}
```

In the above example:

- The `ButtonProps` interface defines the shape of the props.
- `label` is a required `string`, and `disabled` is an optional `boolean`.
- `MyButton` component receives `label` and `disabled`, defaulting `disabled` to `false` if not provided.

Using `type` for Props

A `type` is another way to define the structure of an object, including props. Unlike `interface`, `type` is often used for more specific cases like unions or primitives but can also be used for props.

```
type MyComponentProps = {
  name: string;
  age?: number; // Optional prop
};

const MyComponent: React.FC<MyComponentProps> = ({ name, age }) => (
  <div>
    <p>Name: {name}</p>
    {age && <p>Age: {age}</p>}
  </div>
);
```

In this example:

- The `MyComponentProps` type defines the props structure.
- `name` is a required `string`, while `age` is an optional `number`.
- `MyComponent` displays the `name` and conditionally renders the `age` if provided.

When to Use `interface` vs. `type`

- `interface` is preferred when you need to extend or merge definitions. It's more commonly used in large-scale applications.
- `type` is more flexible for complex types like unions and intersections but can also be used for defining props.

Both approaches are valid, so it's up to you to choose the what fits your use case or coding style.

Typing React Hooks

TypeScript works out of the box with React Hooks. Here are some examples of how to use TypeScript with common hooks like `useState`, `useReducer`, and `useContext`.

`useState`

When you use the `useState` hook, TypeScript will infer the type from the initial value. However, you can also explicitly define the state type:

```
const [count, setCount] = useState<number>(0); // Setting the type to number
```

For union types, such as status states:

```
type Status = "idle" | "loading" | "success" | "error";
const [status, setStatus] = useState<Status>("idle");
```

`useReducer`

The `useReducer` hook allows for more complex state management. You can type the reducer's state and actions:

```
interface State {
  count: number;
}

type Action = { type: "increment" } | { type: "decrement" };

function reducer(state: State, action: Action): State {
  switch (action.type) {
    case "increment":
      return { count: state.count + 1 };
    case "decrement":
      return { count: state.count - 1 };
    default:
      throw new Error("Unknown action");
  }
}

const [state, dispatch] = useReducer(reducer, { count: 0 });
```

`useContext`

The `useContext` hook lets you share values across components without explicitly passing props down the tree:

```
const ThemeContext = createContext<"light" | "dark">("light");
const MyComponent = () => {
  const theme = useContext(ThemeContext);
  return <div>Current theme: {theme}</div>;
};
```

If the context has no initial value or needs null handling, make sure to manage the null state explicitly:

```
const MyContext = createContext<string | null>(null);

const MyComponent = () => {
  const value = useContext(MyContext);
  if (!value) throw new Error("MyComponent must be within MyContext.Provider");
  return <div>{value}</div>;
};
```

`useMemo`

The `useMemo` Hook helps optimize performance by caching the result of a function and recalculating it only when its dependencies change. In TypeScript, you can either rely on TypeScript's inference or be more explicit with a type argument.

```
interface Todo {
  id: number;
  task: string;
  completed: boolean;
}

const visibleTodos = useMemo<Todo[]>(() => filterTodos(todos, tab), [todos, tab]);

function filterTodos(todos: Todo[], tab: string): Todo[] {
  // Filter logic here...
}
```

Here, `visibleTodos` is of type `Todo[]`, inferred from the `filterTodos` function. We can also provide the return type directly as a type argument to `useMemo`.

`useCallback`

`useCallback` returns a stable reference to a function and only re-creates it when dependencies change. It's especially useful for performance optimization in child components or event handlers.

```
const handleClick = useCallback(() => {
  // Function logic...
}, [todos]);

// When strict mode is on, or for explicit typing:
const handleChange =
   useCallback<React.ChangeEventHandler<HTMLInputElement>>((event) => {
      setValue(event.currentTarget.value);
}, [setValue]);
```

Here, `useCallback` is used to memoize the `handleChange` function, and `React.ChangeEventHandler<HTMLInputElement>` is added for explicit typing of the event handler.

`useRef`

The `useRef` Hook returns a mutable object that persists for the lifetime of the component. You can specify the type of the DOM element it refers to using TypeScript.

```
interface InputProps {
  placeholder: string;
}

const Person = ({ placeholder }: InputProps) => {
  const inputRef = useRef<HTMLInputElement>(null);

  return (
    <div>
      <input type="text" ref={inputRef} placeholder={placeholder} />
    </div>
  );
};
```

In this example, we use `useRef` to refer to an `HTMLInputElement`.

Useful Types

The `@types/react` package provides a wide variety of types that make working with React and TypeScript more efficient. Here, we'll cover some of the most common types you'll encounter when working with React components and event handlers.

DOM Events

When handling DOM events in React, the event type is often inferred by TypeScript from the event handler. However, when passing a handler to a component or refactoring, it's best to explicitly set the event type.

Example:

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

export default function Form() {
  const [value, setValue] = useState("Change me");

  function handleChange(event: React.ChangeEvent<HTMLInputElement>) {
    setValue(event.currentTarget.value);
  }

  return (
    <>
      <input value={value} onChange={handleChange} />
      <p>Value: {value}</p>
    </>
  );
}
```

There are many types of DOM events provided by React. The full list can be found in the `@types/react` package, but common examples include:

- `React.MouseEvent`
- `React.KeyboardEvent`
- `React.ChangeEvent`

If you need a generic event type, `React.SyntheticEvent` serves as a base type for all React events.

Children

When defining a component's children in TypeScript, there are two common types you can use to specify what can be passed as children.

1. Using `React.ReactNode`: This is a broad type that can represent anything that can be rendered in JSX (elements, strings, numbers, etc.).

```
interface ModalRendererProps {
  title: string;
  children: React.ReactNode;
}
```

2. Using `React.ReactElement`: This type restricts children to only JSX elements and excludes primitives like strings or numbers.

```
interface ModalRendererProps {
  title: string;
  children: React.ReactElement;
}
```

TypeScript cannot enforce that children must be of a specific JSX type (e.g., only allowing `<li>` elements). This means you cannot specify that a component only accepts certain types of JSX elements through TypeScript's type system.

Style Props

When passing inline styles in React, use the `React.CSSProperties` type to ensure your styles are valid CSS properties and benefit from editor auto-completion.

Example:

```
interface MyComponentProps {
  style: React.CSSProperties;
}

const MyComponent = ({ style }: MyComponentProps) => {
  return <div style={style}>Styled Component</div>;
};
```

Event Types

To determine the correct event type, you can hover over the event handler, and TypeScript will provide the expected event type. If you need a more generic handler or your event isn’t included, fall back on `React.SyntheticEvent`.

Summary

Using TypeScript in a React project brings safety, improved developer experience, and self-documenting code through type annotations. Whether you're managing props, hooks, or context, TypeScript ensures that your React code remains clean and free of common runtime errors. With the flexibility to describe component props and states in detail, TypeScript becomes an invaluable tool for building robust and scalable React applications.

For more advanced use cases, consider exploring union types, generics, and custom hooks with TypeScript.