Introduction To Typescript React

Jonathan Van Dyck – Nexios IT

Main Concepts

1. Functional components & JSX
2. Props
3. Hooks
    1. useState
    2. useEffect
    3. useMemo / useCallback
4. Context

1. Functional Components & JSX

Javascript:

```
const Welcome = (props) => {
  return <h1>Hello, {props.name}</h1>;
}
```

Typescript:

```
interface IProps {
  name: string;
}

const Welcome = (props: IProps) => {
  return <h1>Hello, {props.name}</h1>;
};
```

1. Functional Components & JSX

Usage:

```
{/* // component with no children */}
<Welcome name="Jonathan" />

{/* // component with children */}
<Welcome>Jonathan</Welcome>
```

2. Props

```tsx
interface IUser {
  name: string;
  initials: string;
}

interface IProps {
  author: IUser;
  text: ReactNode;
  date?: Date;
  onClickAvatar: (author: IUser) => void;
}

const Comment = ({ author, date = new Date(), text, onClickAvatar }: IProps) => {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar onClick={() => onClickAvatar(author)} user={author} />
        <div className="UserInfo-name">{author.name}</div>
      </div>
      <div className="Comment-text">{text}</div>
      <div className="Comment-date">{formatDate(date)}</div>
    </div>
  );
};
```

3. Hooks

- Provide life cycle functionality for Function Components
- Name always follows the format useSomething
- useState
- useEffect
- useMemo / useCallback
- … Write your own!

3.1. useState

```tsx
interface IProps {
  defaultCount?: number;
}

const Example = ({ defaultCount = 0 }: IProps) => {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState<number>(defaultCount);

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

3.2. useEffect

```tsx
const MessageSubmit = () => {
  const [value, setValue] = useState<string>("");
  const [submitEnabled, setSubmitEnabled] = useState<boolean>();

  useEffect(() => {
    const isValid = value && value !== "";
    setSubmitEnabled(isValid);
  }, [value]);

  const handleChangeInput = (e: ChangeEvent<HTMLInputElement>) => {
    setValue(e.target.value);
  };

  return (
    <div>
      <input value={value} onChange={handleChangeInput} />
      <button disabled={!submitEnabled}>SUBMIT</button>
    </div>
  );
};
```

3.2. useEffect

```tsx
const WindowWidthDisplay = () => {
  const [windowWidthSize, setWindowWidthSize] = useState<number>(0);

  useEffect(() => {
    const handleResize = (e: UIEvent) => {
      const { width } = document.body.getBoundingClientRect();

      setWindowWidthSize(Math.ceil(width));
    };

    window.addEventListener("resize", handleResize);

    return () => window.removeEventListener("resize", handleResize);
  }, []);

  return <h1>The window size {windowWidthSize} pixels</h1>;
};
```

3.3. useMemo

```tsx
const MessageSubmit = () => {
  const [value, setValue] = useState<string>("");

  const handleChangeInput = (e: ChangeEvent<HTMLInputElement>) => {
    setValue(e.target.value);
  };

  const submitEnabled: boolean = useMemo(() => value && value !== "", [value]);

  return (
    <div>
      <input value={value} onChange={handleChangeInput} />
      <button disabled={!submitEnabled}>SUBMIT</button>
    </div>
  );
};
```

3.4. useCallback

```tsx
const MessageSubmit = () => {
  const [value, setValue] = useState<string>("");

  const handleChangeInput = useCallback((e: ChangeEvent<HTMLInputElement>) => {
    setValue(e.target.value);
  }, [setValue]);

  const handleSubmit = useCallback(() => {
    // submit message here
  }, [value]);

  const submitEnabled: boolean = useMemo(() => value && value !== "", [value]);

  return (
    <div>
      <input value={value} onChange={handleChangeInput} />
      <button onClick={handleSubmit} disabled={!submitEnabled}>SUBMIT</button>
    </div>
  );
};
```

4. Context

- Props & State need to be passed further down the component tree
- Context allows us to make state & logic available to a component and all of its children
- Normally used for global data like authentication & user data or style themes
- Can also be used to have a global data store
- Context Provider & Context Consumer

4. Context

```tsx
export interface IUserContext {
  user?: IUser;
  jwt?: string;
  signIn: (email: string, password: string) => Promise<LoginResult>;
  signOut: () => void;
}

const UserContext = createContext<IUserContext | null>(null);

export const UserContextProvider = ({ children }: IProviderProps) => {
  const [user, setUser] = useState<IUser | undefined>();
  const [jwt, setJwt] = useState<string>();

  const signIn = async (email: string, password: string): Promise<LoginResult> => {
    // sign in user here & save user data & jwt
    const response = await SignIn(email, password);
    setUser(response.user);
    setJwt(response.jwt);
  };

  const signOut = () => {
    setUser(undefined);
    setJwt(undefined);
  };

  return <UserContext.Provider value={{ signIn, signOut, user, jwt }}>{children}</UserContext.Provider>;
};

export default MessageSubmit;
```

4. Context

```
// Create a hook to access the user context
export const useUserContext = (): IUserContext => {
  const context = useContext<IUserContext | null>(UserContext);

  if (!context) {
    throw new Error("User context must be used within a Provider.");
  }
  return context;
};
```

Consume context via the hook:

```
const { jwt, user } = useUserContext();
```

Any Questions?

- More detailed information in the workshop playbook & official documentation