# React with TypeScript

# What are we going to cover?

Creating a new React TypeScript project
- Converting an existing ECMAScript project to TypeScript

How our TypeScript code is compiled

Defining components using TypeScript

Adding compile checking to prop
- And using the prop-types NPM package

Compile checking our state

Using React hooks with TypeScript

# Creating a new project

**Create-React-App** supports TypeScript out of the box

- Use: create-react-app my-project –typescript

Creates a project using TypeScript

- Using .tsx files instead of .js

# Converting an existing project

Converting an existing CRA project is simple

Make sure the react-scripts dependency is up to date

- The minimal version is 2.1

Steps

- Rename one or more JavaScript files to TypeScript
- Create an empty tsconfig.json
- Use NPM or Yarn to install:
  - typescript
  - Type definitions of react, react-dom, node, jest
  - Custom NPM packages that don’t have internal type definitions

# Package.json

```
{
  "dependencies": {
    "@types/jest": "24.0.3",
    "@types/node": "11.9.3",
    "@types/react": "16.8.2",
    "@types/react-dom": "16.8.0",
    "react": "^16.8.1",
    "react-dom": "^16.8.1",
    "react-scripts": "2.1.5",
    "typescript": "3.3.3"  },
  "scripts": {
    "start": "react-scripts start",
    "build": "react-scripts build",
    "test": "react-scripts test",
    "eject": "react-scripts eject"
  },  "eslintConfig": {
    "extends": "react-app"
  },  "browserslist": […]
}
```

# TypeScript config

Create-React-App creates a default TypeScript **configuration file**

Most **settings can be changed** as required

- Don’t delete them as CRA will add the defaults again

# Tsconfig.json

```
{
  "compilerOptions": {
    "target": "es5",
    "lib": ["dom", "dom.iterable", "esnext“ ],
    "allowJs": true,
    "skipLibCheck": true,
    "esModuleInterop": true,
    "allowSyntheticDefaultImports": true,
    "strict": true,
    "forceConsistentCasingInFileNames": true,
    "module": "esnext",
    "moduleResolution": "node",
    "resolveJsonModule": true,
    "isolatedModules": true,
    "noEmit": true,
    "jsx": "preserve"
  },
  "include": [
    "src"
  ]
}
```

# Compiling TypeScript

React-Scripts uses the **Babel preset-typescript** to process TypeScript

- Strips type info and lets Babel process the remaining modern JavaScript

**Type checking** is done using the ForkTsCheckerWebpackPlugin

# Statically analyzing code

Create-React-App uses **ES-Lint to analyze code**

- A limited set of rules is enabled by default

There is an **eslintConfig** setting in the package.json

- Updating has no effect on CRA

# Authoring components

**Class based components** can still inherit from Component

The Component class is defined with **three optional generic arguments**

- The prop type
- The state type
- The snapshot type

The **prop and state types** can be either an interface or type alias

- Using type alias is more flexible

There are **several types** to use for variables referencing component types

- Class based variables are defines as type ComponentClass<P>
- Functional components are defined as type FunctionComponent<P>
- A ComponentType<P> can be either a class based or a functional component

# A class based component

```
type PropTypes = { value: string; };
type StateTypes = { count: number; };
class MyComponent extends Component<PropTypes, StateTypes> {
  state = {
    count: 0
  };
  render() {
    const { value } = this.props;
    const { count } = this.state;
    return (
      <>
        <div>Prop: {value}</div>
        <div>State: {count}</div>
      </>
    );
  }
}
```

# A function based component

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

type HelloProps = {
  firstName: string;
};

const Hello: FunctionComponent<HelloProps> =
  (props: HelloProps) => (
    <div>Hello {props.firstName}</div>
  );

export default Hello;
```

# Component props

**Properties for a child component** are type checked by the compiler

- Missing or wrong type of props will result in a compile error

On a class based component props are read-only

Adding a static **defaultProps** is supported

- Matching keys in the props type automatically become optional where the component is used

The Visual Studio Code rename refactoring works with components

# Component props

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

type HelloProps = {
  firstName: string;
};

export default class Hello extends Component<HelloProps> {
  static defaultProps = {
    name: "Stranger"
  };

  render() {
    return <div>Hello {this.props.firstName}</div>;
  }
}
```

# Using prop-type definitions

Using **prop-type definitions** can still be useful

- Evaluated at runtime where TypeScript is evaluated at compile time
- But only at development time

Use **InferProps<typeof propTypes>** to derive a TypeScript type

# Using prop-type definitions

```
import React, { Component } from "react";
import { string, InferProps } from "prop-types";

const helloPropTypes = {
  firstName: string.isRequired
};

type HelloProps = InferProps<typeof helloPropTypes>;

export default class Hello extends Component<HelloProps>
{
  static propTypes = helloPropTypes;

  render() {
    return <div>Hello {this.props.firstName}</div>;
  }
}
```

# Accessing prop types

A component **props type** can be **exposed** along with the component

- Usually the consumer doesn't need to know the type

Exposing props type can be useful if a consumer wants to re-export them

- Possibly with some extra props

If the prop type is not exposed it can be **derived** from the component

# Deriving prop types

```
import React, {
  ComponentProps
} from "react";
import Hello from "./hello";

type HelloProps =
  ComponentProps<typeof Hello>;
```

# Component state

The **state property** is also read-only

The class based **setState()** function takes a **Pick<S, K>** of the state type as the argument

# React Hooks

Most React hooks take a **generic type argument**

- Usually not need because of type inferencing

# Components can be generic as well

Define a **generic type** as part of the component usage

- Can be useful with low level components that are reused in different places

# Generic components

```
// Definition

const Display = <T extends any>(props:
DisplayProps<T>) => (

  <div>Value: {props.value}</div>

);



// Usage

<Display<string> value="" />
```

# Higher order components

Creating **higher order functions** is almost the same as with JavaScript

Make sure to expose the original prop type

# Higher order components

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


export default function withErrorBoundary<P>(
  WrappedComponent: ComponentType<P>
) {
  return class MyHOC extends Component<P> {
    render() {
      return <WrappedComponent {...this.props} />;
    }
  };
}
```

# Conclusion

Creating a new React TypeScript project is easy using CRA

TypeScript code is compiled using Babel and the TypeScript preset

Writing components in TypeScript is not that different

- Props and State and now type checked by the compiler

Using prop-types still works the same

- The TypeScript type definition can be inferred from the propTypes object

Using React hooks with TypeScript is easy

- Hooks have generic types where needed
- Usually these are inferred and don’t need to be specified