TypeScript Tooling

What are we going to cover?

Bundling with Webpack

Static analysis with ESLint and TSLint

Formatting with prettier

Unit testing and TypeScript

Bundling with Webpack

Webpack is a very popular module bundler.

- Distributed as a Node.js application using NPM.

The Angular CLI use Webpack out of the box.

- So does Create-React-App.

Different approaches:

- Use the ts-loader and have it process TypeScript
  - The awesome-typescript-loader is a similar alternative
- Use babel and the @babel/preset-typescript
  - This uses the powerful Babel along with the TypeScript compiler

Webpack main concepts

Entry

- The **entry** is the file where to start bundling
- There can be multiple entry points

Output

- Where to write the bundled output
- A minimum of **filename** and **path** is required

Loaders

- How individual files are transformed when loaded
- A minimum of **test** and **use** is required for each rule

Plugins

- For performing actions on bundles

Using the ts-loader

Used to be the most **common loader** for Webpack and TypeScript

- The awesome-typescript-loader can be a good alternative

Uses to TypeScript compiler to compile TypeScript to JavaScript

- Using the standard TypeScript options

Webpack configuration ts-loader

```
module.exports = {
  entry: ‘./src/main.ts',
  output: {
    path: path.resolve(__dirname, 'dist'),
    filename: ‘main-bundle.js'
  },
  module: {
    rules: [
      {test: /\.ts$/, use: 'ts-loader'}
    ]
  }
};
```

Using Babel and the TypeScript preset

Babel 7 has a TypeScript preset

- More flexible output targets with @babel/preset-env then the ts-loader
- Very fast because it doesn’t really do type checking
- Removes TypeScript specific features and leaves Babel to do the transpilation
- A few TypeScript features aren’t supported:
  - Namespaces
  - Bracket style type-assertion/cast syntax
  - Enum merging
  - Legacy-style import/export syntax

Configure Babel as usual and add TypeScript and the preset

- npm install –save-dev typescript @babel/preset-typescript

Webpack configuration babel-loader

```
module.exports = {
  entry: './src/main.ts',
  module: {
    rules: [
      { test: /\.(js|ts)$/, loader: 'babel-loader'
      }
    ]
  },
  resolve: {
    extensions: ['.wasm', '.mjs', '.js','.json','.ts']
  }
};
```

.babelrc with TypeScript

```
{
  "presets": [
    "@babel/typescript",
    "@babel/preset-env"
  ]
}
```

Type checking with WebPack

Both **ts-loader** and **@babel/preset-typescript** don’t do type checking

- The only transpile the TypeScript code to ECMAScript

The webpack **fork-ts-checker-webpack-plugin** can be used to type check your code

- Also used in the configuration generated by Create-React-App

Integrating with other build tools

See the TypeScript documentation:

- https://www.typescriptlang.org/docs/handbook/integrating-with-build-tools.html

Static analysis with TSLint or ESLint

Linting is a type of **static code analysis**

- Frequently used to find problematic patterns or code that doesn’t use best practices

Both **ESLint** and **TSLint** can be used to analyze TypeScript code

The TypeScript team has announced that **ESLint is the future**

TSLint

TSLint is an extensible static analysis tool that checks TypeScript code

- It contains many build in rules

Can be used with **React**

- Both tslint-react and tslint-react-hooks are useful rulesets

The rules in **tslint:recommended** are a good place to start

ESLint

ESLint is a very extensible static analysis tool ECMAScript linting utility

- Originally not intended for TypeScript

The ESLint **configuration** allows for custom parsers

- As long as they produce an Esprima compatible abstract syntax tree
- For TypeScript the @typescript-eslint/parser is recommended

ESLint is designed to be much **more flexible** then TSLint

- The main focus for the TypeScript team

Create-React-App uses ESLint to check TypeScript out of the box

ESLint configuration

ESLint does **nothing by default**

- Configure the environment and rules to start checking

There are many popular **ECMAScript configurations** with rulesets for ESLint

- eslint-config-Airbnb
- eslint-config-google

The ECMAScript configurations will work on **TypeScript code** as well

For **React** the following configurations are a good place to start

- react/recommended
- @typescript-eslint/recommended

ESLint with **Prettier**

- Use the prettier/@typescript-eslint to disable rules that arenafected by Prettier

ESLint configuration

Part one

```
module.exports = {
  parser: "@typescript-eslint/parser",
  env: {
    browser: true,
    es6: true
  },
  parserOptions: {
    ecmaFeatures: {
      jsx: true
    },
    ecmaVersion: 2018,
    sourceType: "module"
  },
  globals: {
    Atomics: "readonly",
    SharedArrayBuffer: "readonly"
  },
  // Remaining code
};
```

ESLint configuration

Part two

```
module.exports = {
  // Previous code
  plugins: ["react"],
  settings: {
    react: {
      version: "detect"
    }
  },
  extends: [
    "react-app",
    "plugin:react/recommended",
    "plugin:@typescript-eslint/recommended",
    "prettier/@typescript-eslint"
  ],
  rules: {
    "@typescript-eslint/prefer-interface": "off"
  }
};
```

Prettier

Prettier makes it easy to **standardize code formatting**

- There are plugins available for most code editors

Can also be used via an **NPM script**

```
"prettier": "prettier --write {src,public}/**/*.{js,ts,jsx,tsx,css,scss,json,html}",
```

Git pre commit rules and Husky

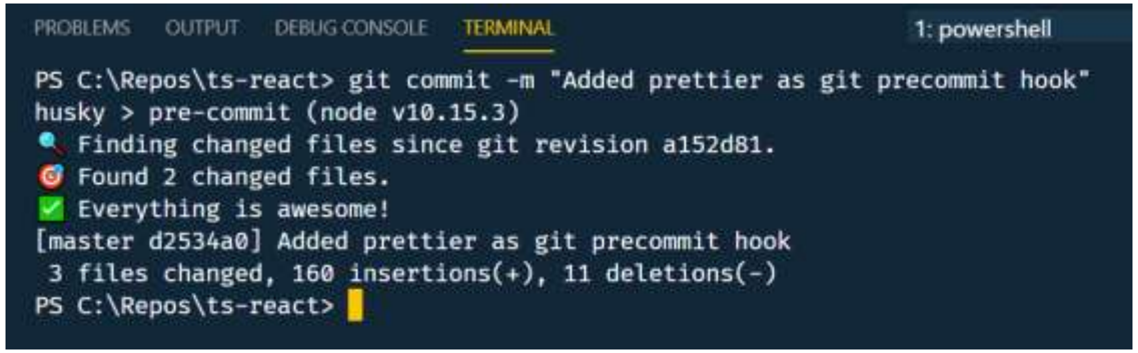
Automate formatting using a GIT pre-commit hook

- Running prettier manually on every pull request becomes tedious

Setup is easy

- NPM install husky and pretty-quick
- Add the following to the package.json

  -
    ```
    "husky": {
        "hooks": {
          "pre-commit": "pretty-quick --staged“
        }
      }
    ```



```
PS C:\Repos\ts-react> git commit -m "Added prettier as git precommit hook"
husky > pre-commit (node v10.15.3)
🔍 Finding changed files since git revision a152d81.
🎯 Found 2 changed files.
✅ Everything is awesome!
[master d2534a0] Added prettier as git precommit hook
 3 files changed, 160 insertions(+), 11 deletions(-)
PS C:\Repos\ts-react>
```

Unit testing with TypeScript

Type checking catches some errors but not all of them.

- Logic errors still require unit testing.

Unit testing TypeScript with Mocha is easy.

- Many other test runners like Jest will work as well
- The Angular-CLI uses Jasmine and Karma to run tests

Mocha requires the **ts-node** compiler to be registered for TypeScript.

- And **ts-node** requires the typescript compiler to be installed

Chai works great for assertions.

- Don’t forget to install the **mocha** and **chai** type definitions

Code under test

```
export default function greet(name){
  return `Hello ${name}`;
}
```

The package.json

```json
{
  "name": "my-app",
  "version": "1.0.0",
  "main": "main.js",
  "scripts": {
    "test": "mocha --require ts-node/register **/*-tests.ts"
  },
  "devDependencies": {
    "@types/chai": "^3.4.34",
    "@types/mocha": "^2.2.39",
    "chai": "^3.5.0",
    "mocha": "^3.2.0",
    "ts-node": "^2.0.0",
    "typescript": "^2.1.5"
  }
}
```

The unit test

```
import 'mocha';
import { expect } from 'chai';
import greet from './greet';

describe('Greet', () => {
  it('should work for Maurice', () => {
    const greeting = greet('Maurice');
    expect(greeting)
      .to.equal('Hello Maurice');
  });
});
```

Conclusion

Webpack is great for bundling the source code.

- Deliver only the code you need to the browser

Static analysis of code

- Both ESLint and TSLint can find a lot of bad practices
- The future is with ESLint

Prettier is a simple and fast way for consistent code formatting

- Use a GIT pre-commit hook to automate code formatting

Unit testing of TypeScript code is no harder than regular ECMAScript.