Exploring Typescript

Typescript is an interesting language. It's a typed superset of Javascript that you can compile to usable Javascript, either ES5 or later yearly versions of the language.

Because it's not straigth Javascript it requires compilation before it can be used in Node or browsers.

Using built-in tools to work with Typescript

The easiest way to use Typescript is to install the tool themselves and then run them through NPM scripts we set up in `package.json`.

Compiling

To compile Typescript we need the Typescript compiler (TSC) that comes bundled with the NPM `typescript` package. To install it run the following command.

```
npm i -D typescript
```

To run the compiler add the following line to the scripts section of your `package.json`.

```
"compile": "tsc ./src/**/*.ts"
```

And run the command with:

```
npm run compile
```

This will convert all the files ending with `.ts` under the src directory regardless of how deep they are in the hierarchy.

Linting

Linting Typescript is a little more complicated than Javascript. We still use ESLint to lint and fix our code but we need to make sure that when we initialize the linter we tell it that we'll be working with Typescript so that it installs the appropriate packages.

Run:

```
npx eslint --init
```

And make sure that when the installer asks you **Does your project use TypeScript?** you answer yes.

It will then install all the packages needed to lint both JS and TS files.

Add the following lines to the script section of your **package.json**

```
"lint": "eslin"eslint ./src/**/*.ts""fix": "eslint --fix ./src/**/*.ts",
```

And run the commands as follows:

To lint run: `npm run lint`.

To lint and fix errors, run: `npm run fix`.

Using Gulp to work with Typescript

If you use Gulp to run and build other aspects of your project it would make sense to use it to process Typescript as well.

The instructions on the next sections assume that you haven't installed or used Gulp before. If you have, some of these instructions may be redundant.

First install the Gulp CLI globally, this will give you the `gulp` command to make your life easier.

```
npm install -g gulp-cli
```

Inside your project run the following command to install Gulp.

```
npm install -D gulp@4
```

Now we're ready to install and work with Typescript.

Compiling

Before working with Typescript we need to install them. To do so run the following command:

```
npm install -D gulp-typescript \
 typescript \
 merge2
```

Once the packages are installed

```
const gulp = require('gulp');
const ts = require('gulp-typescript');
const m'gulp-typescript'erge2');

gulp.task('default', function() {
  const tsResult = gulp.src('js'default'/**/*.ts')
    .pipe(ts({
      declaration: true
  }));

  return merge([
    tsResult.dts.pipe(gulp.dest('dist/js/definitions')),
    tsResult.js.pipe(gulp.dest('dist/js/js'))
  ]);
});
```

Linting

Linting Typescript can be a little hard to understand. We still use ESLint with Typescript presets. There used to be a TSLint application but the creators decided to merge their work with ESLint.

```
npm i -D @typescript-eslint/eslint-plugin \
@typescript-eslint/parser \
gulp-eslint eslint
```

We add the require statement at the top of the file, along with the other declarations.

```
const eslint = require('gulp-eslint');
```

We then add a task to lint our files. Because it uses the ESLint configuration that we created when working on setting up the command line, there is no need to configure the Gulp task itself.

One way to do it may look like this:

```
gulp.task('lint', function () {
  return gulp.src(['./src/**/*.ts']'./src/**/*.ts'int())
    .pipe(eslint.format())
    .pipe(eslint.failAfterError());
});
```

And that's it, we now have a working process to compile and lint Typescript files using NPM and a build system (Gulp in this case).

We'll now move to talking about the language itself.

Things to know about Typescript

Typescript is good to usee but at times it can be really infuriating to learn how to use it and to use it properly.

When you bring in Javascript files to convert to Typescript it will give you many surprises and not all of them are intuitive or easy to decipher.

It's important to remember that ***while Typescript will transpile to Javascript it's superset of the ECMAScript specification, and you need to learn the differences***.

These are not all the things I've learned but they are the most important to me.

When to add types and when to let the compiler do its thing

The Typescript compiler is really good at infering (guessing) the type of your parameters, variables or return values so it's usually OK to let it do its thing.

It is only when we get an unexpected value or when the compiler gives an error that we should explicitly add types to your code.

Pay particular attention when the compiler tells you that there's a type mismatch. For example, an error like `Type '1234' is not assignable to type 'string'` may indicate that we need to be explicit about types (or it may mean we made a mistake, it's always possible).

Take for example the following function signature in Javascript.

```
function setRootVar(name, otname, value) {}
```

When I wrote it I instinctively knew that name and otname were strings and value was a number that would be cast as a string to accommodate CSS requirements.

But Typescript saw it as this:

```
function setRootVar(name: any,
  otname: any,
  value: any) {}
```

The any type tells the Typescript compiler to take any value you pass in and not

check for validity; This defeats the purpose of type checking.

To make sure that the code works as we intended it to we have to explicitly add type declarations to the parameters.

As we said, the name is a string, otname is a string but it's optional and value will become a string so we'll define it as one from the beginning.

```
function setRootVar(name: string,
  value: string,
  otname?: string) {
   // body of the function here
 }
```

Using an optional parameter also forced me to change the order of parameters. Optional parameters must be the last ones on the list.

Declare your types first, then build around them

Typescript checks are concerned with the shape of an object and will use that shape to check if we're doing the right thing.

As we start working with code either from scratch or modifying an existing codebase we may want to start by defining the types that we want to use in an interface.

Let's assume that we defined a `person` interface with three values, two strings and an optional string.

```
interface Person {
  firstName: string;
  lastName: string;
  userName: string;
};
```

Then we can use the interface everywhere we need to identify a person. Below are some examples:

The first one is a person.

```
function createPerson(person: Person): void {
  console.log(person.firstName);
  console.log(person.lastName);
  console.log(person.userName);
}
```

The next example is an administrator. In this example we'll extend the Person interface with additional information that is only relevant for administrators.

```
interface Administrator extends Person {
  signedRelease: boolean;
  accountEnabled: boolean;
}
```

Because the Administrator interface extends Person, we get everything from Person in addition to what we get from Administrator. We're saying an administrator is a person.

```
function createAdmin(admin: Administrator): void {
  // These come from Person
  console.log(admin.firstName);
  console.log(admin.lastName);
  console.log(admin.userName);
  // These come from Administrator
  console.log(admin.signedRelease);
  console.log(admin.accountEnabled);
}
```

Another thing worth exploring is function overload. We can have multiple versions of a function with different parameters that perform different functions based on the type of the parameter.

This is different than generic types because we do know the type of the parameters ahead of time and we code the different functions to handle them.

Generic types

One of the first things I saw and learned about Typescript was the idea of generic types.

There are times when we don't know what parameters we want to use, whether the kind of parameters we want to use wil change over time, or whether we will want to use the same function in different contexts.

```
function id<T>(arg: T): T {
  return arg;
}
```

The id can be a string or a number. Rather than hardcode the type we can use the generic function and decide when we instantiate it what type will it have.

This is legal:

```
let outputString = id<string>("myString");
// outputString: string;
```

So is this:

```
let outputNumber = id<number>(984323243);
// outputNumber: number;
```

This gives us additional flexibility when writing code that will grow along with our project.

Declaration Files (yours and theirs)

Type declarations are ways of providing Type information about JavaScript code base (which by their nature of being JavaScript lacks any type information) to the TypeScript compiler. The type declarations are usually in external files with a `.d.ts` extension.

npm is the recommended tool for managing declaration files. When managing

declaration files with npm, the TypeScript compiler would automatically find the declaration files.

If you want to generate declarations for your own projects you can use the d flag for TSC. This will generate the declarations for you. It is not required but it may be good to have.

A final note about declarations. You're not guaranteed to find declaration files on NPM for all scripts and modules you use so you may have to create them yourself.

Most of the time the creation is simple and the error in your editor will tell you what to do. Other times you will have to create the complete declaration file; Use the Declaration Reference and the Declration Deep Dive when building it.

Documenting Typescript code

I documented Javascript code with JSDoc in comments before the functions or the code.

Typescript supports a subset of JSDoc so I can leverage some Typescript functionality in my existing comments.

typedoc provides a Typescript-specific documentation system. I may try it in a Typescript-only project but I'm hesitatnt to throw away all the JSDoc documentation I've already written.

Working with HTML

Working with HTML in Typescript is more complicated than the equivalent Javascript.

Because we need to add types to elements that are not strictly part of the script, we must use type assertions to make sure the compiler understands what we want to do.

Take the following code snippet as an example:

```
const weight = document.getElementById('robotoWeight');
const weightSlider = document.querySelector('.weightSlider');
```

```
weightSlider.innerHTML = weight.value;
```

The compiler has no way of knowing what type of element `weight` or `weightSlider` reference because it lacks the context of the page the script will run in. The compiler throws an error on the last line of the example because HTMLElement doesn't have a value property.

One solution is to use type assertions to modify the code to specify what we mean and what type of HTML elements we're referencing to.

```
const weight = document.getElementById('robotoWeight') as HTMLInputElement
const weightSlider = document.querySelector('.weightSlider') as HTMLEleme
weightSlider.innerHTML = weight.value;
```

So now the compiler knows what type of HTML element weight references and that it has a value attribtue. Problem solved.

The HTMLElement Interface and its children provide a comprehensive list of all the elements you can cast to when working with Typescript.

There's a lot more to learn about Typescript but so far it's boiled to this:

Be disciplined in working with Typescript as it won't hesitate to fail the compilation and tell you where you were wrong, and stare at you while you try and figure out what the errors mean.