Lab 4

Installing TypeScript

Playground

There are several ways to start using TypeScript. First, if you want to play around with it and get familiar with the language quickly with minimal setup, you can jump stratight to the TypeScript Playground (https://www.typescriptlang.org/play/) where it loads a TypeScript compiler for you to use in the browser and displays the output along with any errors you might have.

Installing in the project

Alternatively, you can install it globally on your system using `npm install --global typescript` , which will make the `tsc` command available from anywhere in your command line.
It usually makes more sense to include TypeScript directly as a (dev) dependency of your project so that it will be installed (with the same version) for anyone else running your project, including CI/CD and build systems. To do so, just replace the `--global` flag with `--save-dev` . Now, you will need to use the `tsc` command inside a script defined in `package.json` (to have the `node_modules/.bin` folder in its `PATH`) or using `npx` .

We first need to create the `tsconfig.json` file using `npx tsc --init` . The generated file contains all the default values and their explanaitions. The `compilerOptions` we need to change are:

```
  "target": "ESNext",
  "module": "ESNext",
  "moduleResolution": "node"
```

in order to tell TypeScript how to find (resolve) node modules, and according to what module system to generate code (CJS or ESM). Lastly, we should add `"include": ["src/**/*.ts"]` to `tsconfig.json` to let the compiler know where our source code lies.

Adding to the bundler.

However, the more common way to use TypeScript in our projects is as part of our bundling process (so that we don’t have 2 separate compilation steps). We still need to have it installed, so the previous step is still valid, but we also need the Rollup TypeScript plugin (@rollup/plugin-typescript (https://www.npmjs.com/package/@rollup/plugin-typescript)) and its peer dependency tslib (https://www.npmjs.com/package/tslib):

```
npm install --save-dev @rollup/plugin-typescript tslib
```

But if `@rollup/plugin-typescript` depends on `tslib`, shouldn't `npm` automatically install it with it?
Actually, `tslib` is considered a *peer* dependency rather than a direct one. You can read more about it here (https://nodejs.org/es/blog/npm/peer-dependencies/), but for all intents and purposes, they're just dependencies that we need to install manually.

After that, all we need is to update our Rollup configuration as such:

```
import typescript from '@rollup/plugin-typescript';

export default {
  // ...
  plugins: [typescript(), /* Other plugins */],
};
```

Adding TypeScript to our projects

To practice writing TypeScript, let's start by transforming our apps from previous labs into TypeScript and see how easy it is to refactor.

Chat-room app

Since this app already had a bundler and `package.json` configured, our work becomes relatively easy. All we need to do is change our file extension from `.js` to `.ts` (don't forget to update the entry point in Rollup config as well) and the compiler already shows us some (possible) errors.

Transforming

For such a simple application, most of our errors will be fixed by simple type assertions as such:

```
const form = document.getElementById('form') as HTMLFormElement;
const input = document.getElementById('input') as HTMLInputElement;
const messages = document.getElementById('messages') as HTMLUListElement;
```

since we are sure these elements exist and have these exact types. This could be a source of error if any updates happen to the HTML, but it's a tradeoff between ensuring safety and convenience.
These simple changes are enough to fix all errors, and now we have more confidence in our code.

However, we can go further in code readability by documenting that we expect the type of the messages we send and receive to be the same. To do so, we can start by adding a type alias for the message type:

```
type Message = string;
```

and while it's functionally equivalent to the plain `string` type, it carries more meaning to anyone who reads the code. We can now use the `Message` type to annotate the `message` variable we emit to the socket, and also the message we receive from the socket channel listener (which was previously implicitly `any`).

Refactoring

Let's say we now want to update our application, adding the functionality of sending the username along with the message. The first thing to do would be to add an `<input>` element in the HTML for the username and get a reference to it in the JS (TS) code.

Now, the first thing to change in the TypeScript code would be the type representing a message as such:

```
type Message = { username: string; message: string };
```

which means that it now represents an object with these 2 properties.

Immediately we can see errors in both locations where this type is used and can fix them right away (saving us the trouble of having to test the whole code again and modify it while it's running). The fixes are straightforward:

```
const myMessage: Message = {
  username: username.value,
  message: input.value,
};
socket.emit('chat message', myMessage);
```

and

```
li.textContent = msg.username + ': ' + msg.message;
```

and that's all. Everything compiles again without errors and the app works as it should 👍.

Jokes app

And now let’s go back to the previous lab exercise: the app that displays jokes using the Joke API.

First, we have the same type assertions we need on all `document.getElementById` calls, the same way we did for the previous app.

After that, we should update our function signatures to define types for the parameters they accept and the return value. First, the `fetchSomeJoke` function accepts a `string` and returns a Joke object, so we now need to define a `Joke` type:

```
interface Joke {
  type: 'single' | 'twopart';
  joke?: string; // only exists if type == 'single'
  // only exist if type == 'twopart'
  setup?: string;
  delivery?: string;
}
```

Now we should use this type to annotate the object returned by `response.json()` since by default it is defined as returning `any` :

```
const data: Joke = await res.json();
```

We should also use the same type to annotate the parameters of `handleJoke` and its 2 variants. However, we then see that we have errors due to `joke` , `setup` , and `delivery` possibly being `undefined` . But we know for sure that, if the type is `single` , the `joke` property has to be defined. One way to go around this would be another type assertion, but that’s not ideal. The better way would be to use discriminated unions and let TypeScript figure out this fact on its own.

To use discriminated union types, we define separate interfaces for the 2 possible types of jokes with a shared property that *discriminates* (or differntiates) between them, and then one type union that combines them:

```
interface SingleJoke {
  type: 'single';
  joke: string;
}
interface TwopartJoke {
  type: 'twopart';
  setup: string;
  delivery: string;
}
type Joke = SingleJoke | TwopartJoke;
```

And now, we can annotate the parameter of `handleOnePartJoke` as accepting just `SingleJoke`, and conversely with `handleTwoPartJoke`. Due to TypeScript's smart control flow analysis, it understand that inside the `case 'single':` part of the `switch (jokeObj.type)`, the type of `jokeObj` can be narrowed down to just `SingleJoke` rather than the generic `Joke`, and similarly in the other branch.

Homework

Transform your app (from Homework 2) to use TypeScript with a bundler. Do not include the compiled output or `node_modules` in git. Do not leave any type as `any` (even implicitly). Follow the exact criteria on Moodle.