TypeScript

Neil Kingdom

October 5, 2024

Contents

1 Abstract

I've titled this document 'TypeScript' because that is what it primarily concerns. Unlike most of my other documents which specifically target the features of a particular language, this document will be a bit more broad in scope, opting to stray away from mere TypeScript to take a look at the history of web libraries and frameworks. This document is meant to be read as a companion to my other document 'Web Applications'. I felt that there was an exceedingly large amount of content, so some of it has been moved here. I hope that by taking a look into some of the history of writing web applications, you're knowledge of TypeScript will be bolstered all the more.

2 Introduction

TypeScript is a bit of a rare specimen, because it isn't *really* its own language. What I mean by that is that it is really just syntactic sugar on top of JavaScript. Because of this, there don't currently exist any interpreters to my knowledge which interpret TypeScript by itself. Instead, it is common to compile TypeScript into JavaScript first, and then use an existing interpreter to run that. This makes sense because most things are already compatible with JavaScript, and trying to make everyone adopt a secondary language would likely not go over well. As mentioned in the abstract, in this document, we'll be covering TypeScript, but also some other libraries and frameworks to give a better idea of TypeScript's role within the industry.

3 JavaScript

Alongside HTML and CSS, JavaScript (a.k.a. JS) makes up the third person in the web trinity. JavaScript is a dynamically typed language, meaning that it does not strongly enforce types. It is also an interpreted language which depends on a Just-in-Time (JIT) compiler to intepret it at runtime, similar to Python. The language is inherently single-threaded, which is why it relies heavily on promises and futures and asynchronous methods. JavaScript was created in just 10 days by Brendan Eich who was working for Netscape at the time, but is now one of the most utilized languages on the web. JavaScript conforms to something known as the ECMAScript (a.k.a. ES) standard, which is also the standard used for JScript and ActionScript (languages similar to JavaScript, but not identical).

JavaScript can be considered a scripting language. It's original intent was to manipulate elements of the Document Object Model (DOM). As web 2.0 roled around, it became apparent that we needed a method of handling more complex logic within our web apps. Considering JavaScript was already supported in most browsers, and that it was a perfectly capable language, it became the standard for interfacing with the back-end and handling most of the application's logic. Considering TypeScript is essentially just JavaScript with types, I won't be covering JavaScript on its own, but learning TypeScript should equip you with everything you need to understand JavaScript.

4 JQuery

JQuery used to be a very popular library for JavaScript. JQuery is still used on many websites, and still has certain niche applications. Back in the day, we used to use JavaScript to operate on the DOM directly. We would select an element using something like getElementById() and then operate on said element. JQuery provided a wrapper which made selecting elements even easier, and also provided ways to easily manipulate the CSS classes associated with elements. Nowadays, with page rehydration, we avoid operating on the DOM directly, since a) it's very error-prone, and b) it requires reloading the entire DOM any time a change needs to be displayed, rather than simply refreshing the element which was updated.

You can tell you're looking at JQuery code when you see a bunch of $ signs being used everywhere. The $ sign expands to the jQuery() function. The jQuery() function (or just $()) has multiple overloads. It returns

a jQuery object, which differs from a standard JavaScript object. We can grab elements by selector or element. For example:

```
$( "div.foo" ).on( "click", function() {
    $( "span", this ).addClass( "bar" );
});
```

The example above grabs any div element with class ”foo”, then sets its onClick handler to an anonymous function which adds the class ”bar” to any span elements within the outer div element. jQuery is still a good library if you really do require something that can manipulate the DOM directy. For example, it might be a useful tool when writing web scrapers that need to extract specific data. It is not advisable that it be used when creating web applications, however.

5 The V8 Engine

As mentioned earlier, JavaScript runs in the browser. In Chrome, this is accomplished by using the V8 engine. The V8 engine is Google’s JavaScript engine, which contains its own JIT compiler for JS. The V8 engine is written in C++. It implements the ECMAScript standard, as well as Web Assembly (WASM).

6 Node.js

In the previous section, we seen how the V8 engine could interpret JavaScript from within the browser. Only being able to run JavaScript within the browser was a bit of a limitation of the language, however. Enter node.js, a runtime environment which utilizes the V8 engine to allow JavaScript to be able to be ran from anywhere. Node.js is sort of to JS what the JVM is to Java. So long as Node.js can be installed on the user’s native platform, then it can interpret JS code. On a bit of a different note, the Electron framework utilizes Node.js to provide users with tools for building traditional GUI desktop applications. Electron is often heavily criticized for utilizing so much memory since it’s not only built on top of multiple layers of abstraction, but also requires JIT compilation. Still, many apps such as the Atom and VSCode IDEs.

node.js works similarly to Python, in that you can run the command on its own and you’ll enter a sandboxed environment where you can execute JS statements in real time. This is pretty nice when you just want to see what something evaluates to quickly without setting up a whole project.

7 package.json

node.js introduced the package.json file for managing node projects. node.js will search for this file in order to find metadata about the project’s structure. This is essentially equivallent to Java’s MANIFEST.md file or Rust’s cargo.toml file. package.json has a couple properties that can and should be overridden when creating a project in JS or TS. These include the following:

- **name:** Sets the name of the project.
- **version:** Sets the version number for the project.
- **license:** Sets the type of license used for the project.
- **description:** Sets a brief description about the project.

- **keywords:** Sets a brief description about the project.
- **main:** The file containing the entry point of the program.
- **repository:** Used for specifying metadata about an upstream repository e.g. GitHub.
- **scripts:** Creates custom build actions similar to rules in a Makefile.
- **dependencies:** A list of dependencies and their expected versions. Usually managed by npm.
- **devdependencies:** A list of developer dependencies and their expected versions. Usually managed by npm.

8 Node Package Manager

Node Package Manager (NPM) is the official package manager for node.js. npm is a pretty straightforward package manager, though there are a couple of things to note before you go out and start installing a bunch of packages. First of all, npm is notorious for having vulnerabilities in its packages since anyone can easily upload their libraries to the upstream repo, and because the web is such a high priority target for hackers. npm will mark vulnerable packages as such, but this will only occur after the vulnerabilities have been reported and confirmed.

To find a package by keyword, use npm search [search terms]. If a term begins with a forward slash, it shall be interpreted as a regex pattern. When it comes to installing packages, you have a few options that you might want to consider. Before I get carried away, note that you can run npm help <subcommand> to bring up a man page containing additional information about that subcommand. For npm install, the most important option is the -g or –global flag, which will install the package system-wide. The default is to install the package in the user's current directory in a folder called node_modules. As mentioned in the previous section, npm will utilize package.json, alongside node.js. npm primarily utilizes this file to track dependencies and dependency versions. package.json may contain three different subsections for packages. These are: production, developement, and optional dependencies. These subsections correspond to the -P, -D, and -O flags, respectively, when running the npm install subcommand. -P is the default option. These help distinguish which packages are actually critical for the application, and which ones are used for developer tools or environment setup. Running npm install by itself will synchronize any packages located in package.json, meaning that any dependencies which are out of date or which are not installed will either be upgraded or installed, respectively. If your packages become corrupt, it's easiest to delete the node_modules directory and simply run npm install. It should also be noted that most npm subcommands have aliases, so npm i and npm install are equivallent.

You can run npm init to intialize the current directory. Unless the -y option is used, this will prompt you to fill out several of the typical properties in the package.json file. npm init essentially is just a fancy script for generating the package.json file, and that's about it.

We also mentioned in the previous section that we could create custom build actions in package.json within the scripts property. The scripts property can be set to an object containing one or more rules. These rules can be executed using the npm run subcommand. npm defines a couple default rules which can be invoked directly instead of having to use the run subcommand. These include: test, start, restart, and stop. So in other words, rather than invoking npm run start, we could just say npm start instead; but only for the aforementioned aliases, otherwise you must use npm run <command>. Here's an example:

```
"scripts": {
    "start": "node index.js",
    "test": "jest",
    "custom": "./run_script.sh"
}
```

9 Node Package Executor

The Node Package Executor (NPX) is bundled with newer releases of npm. If you do not have it installed, you can install it globally via npm e.g. npm i -g npx. The npx command is capable of executing packages installed with npm from the command line. This allows devs to write CLI tools in JS, upload them to the npm repository and then anyone using that CLI tool can execute it using npx.

10 Managing the Node.js Release Version

We've seen how to manage packages for a project using npm, but what about managing the release of node.js itself? For this, we can use another CLI tool called the Node Version Manger (NVM). nvm is the simplest way of switching the version of node.js that your system is using. This is very helpful when maintaining projects that use different node.js versions. You can list out the node.js versions available to you by running nvm ls-remote. nvm install –lts will install the latest Long Term Release version, nvm install node to install the latest version available, or nvm install <version> with the specific version you want. Likewise running nvm use with any of the same options for install will activate the appropriate node js version on the system.

11 Alternative Package Managers

11.1 pnpm

11.2 Yarn

12 TypeScript as a Dependency

All this talking and yet we are just now getting to the bread and butter of using TypeScript. As mentioned, TS isn't really its own language per se. For this reason, we can actually just install it as a developer dependency for our project e.g. npm i -D typescript. TS should now appear as a dependency in the node_modules directory. If we do an ls on node_modules/typescript/bin, we can find two useful tools for working with TS: tsc and tsserver. Let's look at each of these individually.

12.1 The TypeScript Compiler

The TypeScript Compiler (TSC) is the official compiler for converting from TS to JS.

12.2 TS Server

13 React

NOTE: Update (react is just a library, not a framework) React is arguably the most popular framework on the market in current year. This is mostly in part thanks to its backing by Facebook, and its more mature lifetime (frameworks such as Angular, Vue JS, and Svelte are all significantly more recent). One thing that I think confuses new web developers learning React is its use of JSX. JSX is truly a Frankenstein mishmash of JavaScript and HTML, which makes it sometimes difficult to distinguish. For instance, it is possible using JSX to assign an HTML element to a JavaScript variable.

14 Babel.js

Babel.js is a JavaScript compiler. It's actually more like a transpiler, meaning that it converts high level code into code of essentially the same complexity. Babel.js is primarily used for compiling current versions of JS into older versions for browsers that don't support later versions. Babel.js is also used for compiling JSX and React into JS, which is likely what it's most often used for.

15 JSX

```
const element = <h1>This line of code uses JSX</h1>
```

What React fundamentally allows us to do is create custom JSX elements/components which look like HTML tags and accept attributes in the same manner. This powerful heirarchical pattern allows us to build components of off other pre-existing components, which makes development quicker and less repetitive. Practically speaking, components are created by returning a block of JSX code from a function. A new tag is then generated based off the name of the function. React components must use Pascal casing, which distinguishes them from regular functions. For example, we can create a custom button component with the following code:

```
const CustomButton = (): JSX.Element => {
    const sayHello = () => {
        alert('Hello, World');
    }

    return (
        <div>
            <button
                style={ color: 'red' }
                onClick={ sayHello }>
                My Custom Button
            </button>
        </div>
    );
};

export default CustomButton;
```

Then elsewhere, we can import this component and use it like so (note that App() is the entry point for React):

```
import CustomButton from 'path/to/file/CustomButton';

function App(): JSX.Element => {
    ...
    <CustomButton />
    ...
};

export default App;
```

We can escape JSX and insert JavaScript code using curly braces. This is what we did for the style and onClick attributes in the former example. Likewise, any comments must first be escaped like so:

```
{ /* A comment in JSX */ }
```

React uses hydration to update individual components, rather than reloading the entire DOM each time the state of a component changes. This is, naturally, much faster than attempting to manipulate individual elements in the DOM, which is what something like jQuery does. Once again, looking at the practical application in terms of how this is achieved, React uses something called hooks (essentially a glorified name for callback functions) to update Components when the state is changed. There are many kinds of hooks, which we'll look at shortly, but the relevant kind that we're interested in for updating components on state change are state hooks. You can tell that a function is a hook in React if it begins with the word 'use'. State hooks are created using the useState() function. What this does is return a list with exactly two items. The first item it returns will be the variable that will maintain state, and the second item will be a callback function that can be used to update the state of this variable. The idea is to keep the state variable

as immutable i.e. we should not update the state variable directly, but rather indirectly, via the callback. Typically, we name the state variable some relevant name, and we name the callback the same thing, but prepended with 'set', akin to a setter in OOP languages. One final note before I completely lose you: the useState() hook can take a parameter (in TS this would be of type any) and assign that value as the initial state for the state variable. Let's look at this in code:

```
function App(): JSX.Element {
    const [counter, setCounter] = useState(3);
    return (
        <div>
            <h1>{ counter }</h1>
            <button onClick={ () => { setCounter(counter + 1); } }>
                Increment counter
            </button>
        </div>
    );
}

export default App;
```

16 Hooks

React uses something called hooks, which are just a fancy word for callbacks/function pointers that do a thing. In React, all hooks begin with the word use, followed by the actual hook name. We'll look at some commonly used hooks and why you'd want to implement them in your code. Hooks are idiomatic in React because they typically take advantage of React's page hydration, where lone elements can be updated apart from the rest of the DOM.

16.1 useState

The useState hook is by far the most common. It is used for creating a state variable. We can give this state variable an initial value, which is what the component will render on its initial load. Using the setter that useState provides, we can also update the state variable, which has the side-effect of rehydrating the outer component. As with most hooks, useState returns an array containing both the state variable and setter for said variable. We tend to use array deconstruction to perform multiple assignment as seen in the code snippet below:

```
import React, { useState } from 'react';

const [state, setState] = useState<string>('Initial value');
```

It is common to see this type of code being used when working with React. The setter is almost always given the same name as the state variable but is prepended with 'set', as is the normal convension for getters/setters.

16.2 useEffect

The useEffect hook is very similar to the useState hook, but it is a bit more powerful. The useEffect hook accepts an array of dependencies. These dependencies can be other React components, or they can be variables, such as state variables created with the useState hook (or regular variables too). The function bound to useEffect will run at least once after the outer component renders for the first time. If the dependency list is not empty, then the function will be reinvoked anytime a dependency changes. We can also optionally return a callback which will be executed once the dependencies are unmounted. This return callback is only really useful when using something like a timer that needs to be cleaned up once it expires. Here's an example of useEffect:

```
import React, { useState, useEffect } from "react";

export function App(props) {
    const [count, setCount] = useState(0);

    useEffect(() => {
        console.log("A dependency was updated");
    }, [count]); // Dependency list

    return (
        <div>
            <input value={`Count: ${count}`} type="submit" onClick={() =>
            ↪  setCount(count + 1)}></input>
        </div>
    );
}
```

16.3 useContext

The useContext hook is used to setup context objects. A context object will usually manage some global state which needs to be shared throughout the application. A good example of this is managing a color pallette that will be applied to components throughout the application. This color pallette may need to be updated on the fly to support a light and dark theme. A context object is definitely the way to go in this case. The way this works can be a bit confusing at first, but it's not too bad. We need to create two things: the context object, as well as the context provider. The context object manages the state, while the provider distributes it to all child components starting at the top of the DOM. Here's how we might setup a context object to keep track of the score in a game:

```tsx
// GameContext.tsx
import { createContext, useState } from React;

export const GameContext = createContext(null); // null means no pre-existing state
↪ that we're borrowing

export const GameProvider = ({ children }) => {
    const [score, setScore] = useState(0);

    const incrementScore = () => {
        setScore((prevScore) => prevScore + 1);
    };

    return (
        <GameContext.Provider value={{ score, incrementScore }}>
            { children }
        </GameContext.Provider>
    );
}

// Game.tsx
import { useContext } from React;
import { GameContext } from 'GameContext'

const Game(props) => JSX.Element {
    const { score, incrementScore } = useContext(GameContext); // Access
    ↪ GameContext

    if (playerScoredGoal) {
        incrementScore();
    }

    return (
        <div>
            <h1>Score: {score}</h1>
        </div>
    );
}

// App.tsx
import { GameProvider } from 'GameContext'

const App(): JSX.Element {
    // Wrap all children in the GameProvider
    return (
        <GameProvider>
            <Game />
        </GameProvider>
    );
};

export default App;
```

16.4 useDeferredValue

16.5 useId

17 ESLint

The ECMA Script linter (eslint) is a static code analysis tool which enforces pre-defined code styling and standards. ESLint is not really intended to catch semantic errors so much as it is meant to catch poor code habits. It can enforce things like spacing, indentation, etc.

18 Concurrency in Typescript

JavaScript introduced the concept of promises and futures. I've spoken about promises and futures in many of my other programming language documents, but to summarize, promises and futures are an asynchronous idiom. Async code does not run parallel like with multithreading, but it does run concurrently. This is a requirement for JavaScript, since the JIT is single-threaded. Async portions of code are categorized into coroutines. The cpu is able to perform context switching to jump between coroutines and process code without blocking for a specific line to finish execution. A function marked with the async keyword must return a Promise<T>. Let's first look at how we used to use promises and futures before moving on to the async await syntax so that we can get a better idea of what's happening behind the scenes.

Assume you have some function that pulls data from a database and might take an extended period of time to run. We can simulate this wait time using setTimeout(). Let's take a look at the following code snippet, which will continously run until either the timeout expires, or Math.random() chooses a number greater than 0.5.

```
function fetchData(url: string): Promise<string> {
    return new Promise((resolve, reject) => {
        setTimeout(() => {
            if (Math.random() > 0.5) {
                resolve("Data fetched successfully");
            } else {
                reject(new Error("Network error"));
            }
        }, 100);
    });
}
```

In the code above, we return a string wrapped in a Promise object. When using the new keyword when constructing a Promise object, it will expect a function with the resolve and reject parameters. These are functions which are used to specify whether a task completed successfully (resolve) or failed (reject). We can handle promises in various ways, but the most common are to use the then() and catch() functions, or to use the async/await keywords within a try-catch block. Here is an example of both:

```
// With then() and catch()
fetchData("My SQL query")
    .then((response) => {
        console.log(response);
    })
    .catch((error) => {
        console.error(error);
    });

// With async/await
const fetchDataAsync = async () => {
    try {
        const response = await fetchData("My SQL query");
        console.log(response);
    } catch (error) {
        console.error(error);
    }
}
```

Note that in the async/await version, we need to wrap any code which uses the await keyword in an async block. In this case, we create a fetchDataSync variable which is assigned an anonymous async lambda expression. This lambda expression wraps the call to fetchData in a try-catch block since it needs to be awaited. If the await fails, an exception will be thrown, and we'll end up in the catch block.