Introducción a Typescript

Introducción a Typescript

0. Índice de contenidos.

1. Introducción

Desarrollar aplicaciones en Javascript puede convertirse en todo un desafío una vez que empieza a crecer tanto la aplicación como el equipo, originalmente se pensó como un lenguaje para hacer efectos sencillos en la páginas estáticas, y estamos intentándolo usar para hacer aplicaciones mucho más complejas de lo que se pensó originalmente posible.

Aunque Javascript está avanzando a pasos agigantados, la compatibilidad con navegadores antiguos es todavía un problema, por ello la gente de Microsoft creó Typescript, un lenguaje Open Source basado en Javascript y que se integra perfectamente con otro código Javascript, solucionando algunos de los principales problemas que tiene Javascript:

- Falta de tipado fuerte y estático.
- Falta de "Syntactic Sugar" para la creación de clases
- Falta de interfaces (aumenta el acoplamiento ya que obliga a programar hacia la implementación)
- Módulos (parcialmente resuelto con require.js, aunque está lejos de ser perfecto)

Typescript es un lenguaje con una sintaxis bastante parecida a la de C# y Java, por lo que hace fácil para los desarrolladores con experiencia en estos lenguajes el aprender Typescript.

Otra ventaja del tipado estático es que habilita a otras herramientas a mejorar el soporte y la detección de errores sin necesidad de arrancar el código, además de servir cómo documentación al programador.

2. Sintaxis

Tipos básicos

Los tipos básicos que maneja Typescript son `boolean`s, `number`, `string`, `Any` y `Void`.

```
var isDone: boolean = false;
var height: number = 6;
var name: string = "bob";
var list:number[] = [1, 2, 3];
var notSure: any = 4;
function warnUser(): void {
  alert("This is my warning message");
}
```

Los cuatro primeros tipos no hace falta explicarlos ya que cualquier desarrollador estará bastante familiarizado con ellos, el 5º tipo Any es un tipo dinámico, se utiliza principalmente cuando no queremos declarar el tipo o cuando estamos declarando el tipo de una librería de terceros, también se usa en arrays que contienen distintos tipos.

Por último, `void` se utiliza principalmente para declarar el tipo de funciones que no devuelven nada, cómo en el ejemplo de arriba.

Clases e interfaces

```
interface Animal {
  name : string;
  makeSound();
}


class Dog implements Animal {
  constructor(name:string) {
    this.name = name;
  }

  name:string;

  makeSound() {
    return "guau!";
  }
}

function sayHi(animal:Animal) {
  console.log("hi " + animal.name);
}

sayHi(new Dog("Timmy"))
```

Esto, como es de esperar devolverá `hi Timmy` por consola.

Modulos

Este quizás sea uno de los puntos más necesarios a la hora de conseguir una mejor arquitectura en las aplicaciones Javascript.

Typescript lo resuelve usando una sintaxis parecida a la que veremos en Javascript cuando el estándar ES6 se implemente en los navegadores. Typescript tiene dos tipos de módulos internos y externos, a continuación vamos a explicar los externos, ya que queremos que se puedan usar en todos los navegadores (para ello los compila en módulos de Require)

Vamos a repartir una clase / interfaz por fichero, de manera que podamos tener el código bien organizado, basándonos en el ejemplo anterior:

`models/Animal.ts` `models/Dog.ts` `Main.ts`

```
//models/Animal.ts
interface Animal {
  name : string;
  makeSound();
}

export = Animal
```

Podemos ver la keyword `export` que permite que otros módulos/clases utilicen la interfaz Animal.

```
// models/Dog.ts
import Animal = require('Animal')

class Dog implements Animal {
 constructor(name:string) {
  this.name = name;
 }

 name:string;

 makeSound() {
   return "guau!";
 }
}

export = Dog
```

En este ejemplo se ven dos keywords nuevas, `import` y `require`, tras el import nombramos a la variable que vamos a tener disponible en el resto del fichero y con require nombramos el modulo que queremos importar.
Por último, para usar estos módulos en nuestro fichero `Main.ts`:

```
import Dog = require('models/Dog')
import Animal = require('models/Animal')

function sayHi(animal:Animal) {
  console.log("hi " + animal.name);
}

sayHi(new Dog("Timmy"))
```

y el resultado: `hi Timmy`.

Puedes ver todos los ficheros en Github

Genéricos

Los genéricos son muy útiles para hacer código mas reusable, uno de los claros ejemplos son las listas/Arrays, en el siguiente extracto podemos observar como podemos definir el tipo del Array.

```
var animals:Array<Animal> = [new Dog('Timmy'), new Dog('Michael'), new Dog('Dwight')];
animals.forEach(sayHi);
```

Esto puede ayudar sobre todo a hacer librerías y piezas de código más reutilizables.

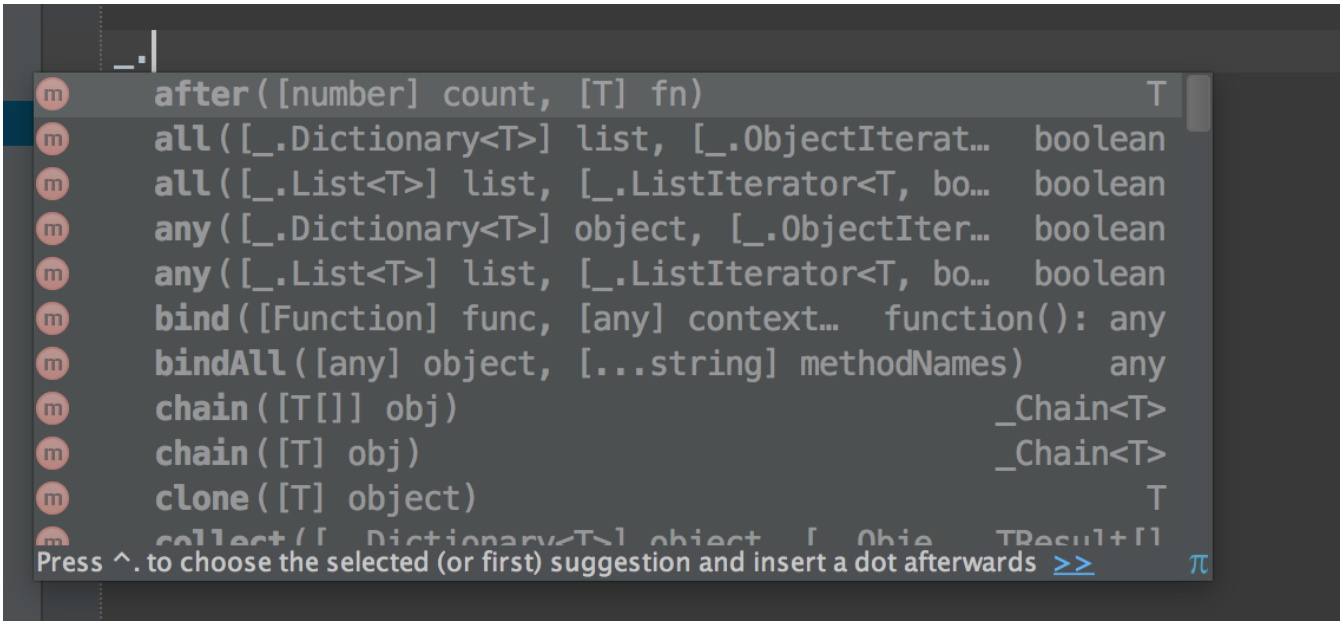
3. Comunicación con código Javascript ya existente

Otra de las ventajas que otorga Typescript es que permite comunicarse con código Javascript ya creado e incluso añadirle “tipos” a través de unos ficheros `d.ts` que indican los tipos que reciben y devuelven las funciones de una librería por ejemplo.

Hay una estupenda comunidad con definiciones para algunas de las librerías más usadas, como pueden ser Underscore, jQuery o Backbone.

Con solo descargarse la definición y añadir una linea de texto en nuestro fichero, nos proporciona tipado para aquellas librerías que no han sido escritas en Typescript.

```
/// <reference path="../libs/underscore.d.ts" />
```



Aquí podemos ver como IntelliJ es capaz de decirnos que tipo espera cada función.

En caso de que usáramos una librería interna o que no tenga declaración de tipos podemos hacerla nosotros, según fuésemos necesitando las distintas funciones.

```
declare var _: {
  each<T, U>(arr: T[], f: (elem: T) => U): U[];
  delay(f: Function, wait: number, ...arguments: any[]): number;
  template(template: string): (model: any) => string;
  bindAll(object: any, ...methodNames: string[]): void;
};
```

Si por lo que fuera necesitáramos acceder desde Javascript al código generado por Typescript es muy sencillo, ya que el código generado es fácilmente legible como podremos comprobar a continuación.

4. Compilador Typescript, Grunt y Gulp.

Para que funcione todo esto hace falta un paso intermedio para convertir el código Typescript en código Javascript, para ello existen varios métodos.

Compilador Typescript

El equipo de Typescript provee una herramienta de linea de comandos para hacer esta compilación, se encuentra en npm y se puede instalar con el siguiente comando:

```
<code>npm install -g typescript</code>
```

Podemos usarlo escribiendo

```
<code>tsc helloworld.ts</code>
```

Si queremos compilar varios ficheros, y que los módulos sean con notación AMD, podríamos hacerlo:

```
<code>tsc foo.ts bar.ts --module amd</code>
```

Esto no escala, ya que poner todos los ficheros uno a uno en la consola, puede resultar muy incomodo, por ello podemos usar Typescript con herramientas como Grunt o Gulp, a continuación voy a poner un ejemplo de como montar la build con Gulp.

Compilando con Gulp

Gulp es un TaskRunner parecido a Grunt (y Maven para los que vengan del mundo Java, sin la parte de dependencias).

La idea es que se ejecute automáticamente el compilador Typescript cada vez que modifiquemos un fichero, y se recargue el navegador con los ficheros Javascript generados.

(A partir de aquí tenemos en cuenta de que tienes instalado Gulp en tu ordenador, puedes leer cómo aquí)

```
gulp.task('compile:typescript', function () {
 gulp.src('app/src/**/*.ts')
 .pipe(ts({
 declarationFiles: true,
 noExternalResolve: true,
 module: 'amd'
}))
.pipe(gulp.dest('dist/scripts'))
});
```

Esto lo que hará será compilar todos los ficheros con la extension `.ts` con las opciones que le pasemos en el objeto json dentro de `ts()` y los pondrá en `dist`.
Esta tarea ha de ser invocada manualmente, el siguiente paso será crear una tarea que detecte los cambios en los ficheros con extension `.ts` y ejecute automáticamente dicha tarea, para ello usaremos la función watch de gulp.

```
gulp.task('default', function () {
  gulp.watch('app/src/**/*.ts', ['compile:typescript'])
});
```

Por último vamos a crear una tarea que nos cree un servidor local donde probar nuestra aplicación, y que cada vez que cambiemos algún fichero se recargue el navegador.

```
// Watch Files For Changes & Reload
gulp.task('serve', ['compile:typescript'], function () {
  browserSync({
  notify: false,
  server: ['dist', 'app']
});
gulp.watch(['app/**/*.html'], reload);
gulp.watch(['app/src/**/*.ts'], ['compile:typescript', reload])
});
```

Puedes ver el fichero entero con todos las dependencias aquí.

5. Revisando el código generado

Por último, vamos a comprobar el código que ha generado el compilador de Typescript, el siguiente código es el que ha generado para el fichero `main.js`

```

```

Cómo se puede observar es código fácilmente legible, y podría ser fácilmente generado por un humano, esto es una de las principales ventajas de Typescript respecto a otros lenguajes que compilan a Javascript.

También podemos observar como se compilan las clases:

```
define(["require", "exports"], function (require, exports) {
    var Dog = (function () {
        function Dog(name) {
            this.name = name;
        }
        Dog.prototype.makeSound = function () {
            return "guau!";
        };
        return Dog;
    })();
    return Dog;
});
```

Como podemos ver no se hace ningún tipo de mención a la interfaz Animal y esto es debido a que la comprobación de que cumpla la interfaz se hace en tiempo de compilación y no en tiempo de ejecución, por lo que no incurrimos en ningún tipo de penalización.

6. Conclusiones

Como hemos podido ver Typescript añade algunas cosas interesantes que echamos de menos en Javascript que ayudaran a mejorar el código y a aumentar la mantenibilidad en proyectos grandes.

También hemos podido observar que el código generado no es “mágico” por lo que no debería dificultar el debugging o la recolección de trazas. Además de que el tipado puede ayudarnos mucho a la hora de hacer nuestro código más robusto y además nos ofrece una documentación de lo que hacen nuestras clases y funciones sin perder las mejores partes de Javascript.

Puedes encontrar el código en el siguiente repositorio de Github