



Formation TypeScript

Romain Bohdanowicz

Twitter : @bioub - Github : <https://github.com/bioub>

<https://formation.tech/>

Présentations



- Romain Bohdanowicz
Ingénieur EFREI 2008, spécialité en Ingénierie Logicielle



- Expérience
Formateur/Développeur Freelance depuis 2006
Près de 2000 jours de formation animées

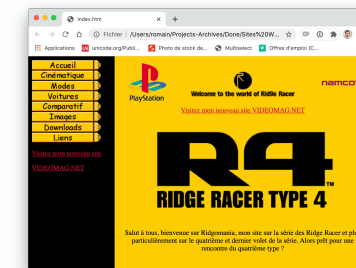
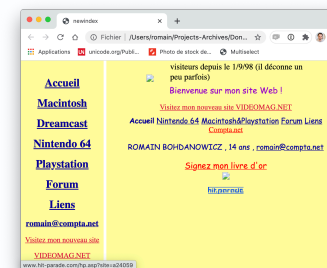
- Langages
Expert : HTML / CSS / JavaScript / TypeScript / PHP / Java
Notions : C / C++ / Objective-C / C# / Python / Bash / Batch



- Certifications
PHP / Zend Framework / Node.js



- A propos
Premier site web à 12 ans (HTML/JS/PHP)
Triathlète du dimanche





Introduction

TypeScript - Introduction



- TypeScript : JavaScript + Typage statique + quelques nouvelles syntaxes (ex: Enum, Décorateurs...)
 - TypeScript est un langage créé par Microsoft, construit comme un sur-ensemble d'ECMAScript
 - Pour pouvoir exécuter le code il faut le transpiler (ou compiler) en JavaScript, c'est à dire le transformer en JavaScript avec le compilateur
 - A quelques exceptions près et selon la configuration, le JavaScript est valide en TypeScript
 - Le principal intérêt de TypeScript est l'ajout d'un typage statique

TypeScript - Installation



- Installation

- `npm i typescript -D`

- Le package typescript contient 2 binaires :

- `tsc` : le transpileur / compilateur
 - `tsserver` : encapsule le compilateur + un language server qui remonte via un protocole JSON les informations de complétion, erreurs, etc à l'IDE

TypeScript - Transpilation



- Création d'un fichier de configuration tsconfig.json
 - `tsc --init`
- Compilation si le fichier tsconfig.json existe dans le projet
 - `tsc`
- Compilation en mode watcher
 - `tsc --watch`
`tsc -w`



- Le principal intérêt de TypeScript est d'ajouter à JavaScript la notion de typage statique :

```
function hello(name: string) {  
    return `Hello ${name.toUpperCase()}`;  
}  
  
console.log(hello('Romain'));
```

- Dans cet exemple, le paramètre *name* de la fonction *hello* est typé statiquement avec *: string*
- Sur des bases de code importantes, cela introduira des contraintes supplémentaires permettant d'améliorer la complétion et la détection d'erreurs, facilitant ainsi le travail à plusieurs (1 développeur qui crée une fonction, 1 autre qui l'utilise)



- Complétion

```
function hello(name: string) {  
  return `Hello ${name.to}`;  
}  
  
console.log(hello('Roma'))
```

Autocomplete suggestions for `to`:

- `toLocaleLower...` (me)
- `toLocaleUpperCase`
- `toLowerCase`
- `toString`

- Détection des erreurs à la compilation

```
function hello(name: string) {  
  return `Hello ${name.toUpperCase()}`;  
}  
  
console.log(hello(123));
```

- En JavaScript l'erreur se produirait à l'exécution mais encore faut-il que la ligne soit exécutée : exécution conditionnelle, exécution lors d'événements...



Les types

TypeScript - Types JavaScript



- Types basiques :

- *boolean*
- *number*
- *string*

```
function createUser(username: string, age: number, isActive: boolean) {  
  return {  
    username,  
    age,  
    isActive,  
  };  
}
```

TypeScript - Typage statique



▸ Tableaux

```
const firstNames: string[] = ['Romain', 'Edouard'];  
const colors: Array<string> = ['blue', 'white', 'red'];
```

▸ Tuples

```
const email: [string, boolean] = ['romain.bohdanowicz@gmail.com', true];
```

▸ Enum

```
enum Choice {Yes, No, Maybe}  
  
const c1: Choice = Choice.Yes;  
const choiceName: string = Choice[1];
```

▸ Never

```
function error(message: string): never {  
    throw new Error(message);  
}
```

TypeScript - Typage statique



- Any

```
let anyType: any = 12;  
anyType = "now a string string";  
anyType = false;  
anyType = {  
  firstName: 'Romain'  
};
```

- Void

```
function withoutReturn(): void {  
  console.log('Do something')  
}
```

- Null et undefined

```
let u: undefined = undefined;  
let n: null = null;
```

TypeScript - Assertion de type



- Le compilateur ne peut pas toujours déterminer le type adéquat :

```
const formElt = document.querySelector('#myForm');  
const url = formElt.action; // error TS2339: Property 'action' does not exist on  
type 'Element'.
```

- Il faut alors lui préciser, 2 syntaxes possibles

```
let formElt = <HTMLFormElement> document.querySelector('#myForm');  
const url = formElt.action;
```

```
let formElt = document.querySelector('#myForm') as HTMLFormElement;  
const url = formElt.action;
```

TypeScript - Inférence de type



- TypeScript peut parfois déterminer automatiquement le type :

```
const title = 'First Names';  
console.log(title.toUpperCase());  
  
const names = ['Romain', 'Edouard'];  
for (let n of names) {  
    console.log(n.toUpperCase());  
}
```

TypeScript - Fonctions



- On peut typer les paramètres d'entrées et de retour d'une fonction

```
function hello(name: string): string {  
    return `Hello ${name.toUpperCase()} !`;  
}
```

- On peut également utiliser

```
function useCallback(cb: Function) {  
    cb();  
}  
  
useCallback(() => {});
```

TypeScript - Interfaces



- Pour documenter un objet on utilise une interface
 - Anonyme

```
function helloInterface(contact: {firstName: string}) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```

- Nommée

```
interface ContactInterface {  
    firstName: string;  
}  
  
function helloNamedInterface(contact: ContactInterface) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```


TypeScript - Interfaces



- Les propriétés peuvent être :
 - optionnelles (ici *lastName*)
 - en lecture seule, après l'initialisation (ici *age*)
 - non déclarées (avec les crochets)

```
interface ContactInterface {  
    firstName: string;  
    lastName?: string;  
    readonly age: number;  
    [propName: string]: any;  
}  
  
function helloNamedInterface(contact: ContactInterface) {  
    console.log(`Hello ${contact.firstName.toUpperCase()}`);  
}
```



- Quelques différences avec JavaScript sur le mot clé class
 - On doit déclarer les propriétés
 - On peut définir une visibilité pour chaque membre : *public*, *private*, *protected*

```
class Contact {  
  private firstName: string;  
  
  constructor(firstName: string) {  
    this.firstName = firstName;  
  }  
  
  hello() {  
    return `Hello my name is ${this.firstName}`;  
  }  
}  
  
const romain = new Contact('Romain');  
console.log(romain.hello()); // Hello my name is Romain
```

TypeScript - Classes



- Une classe peut
 - Hériter d'une autre classe (comme en JS)
 - Implémenter une interface
 - Être utilisée comme type

```
interface Writable {  
  write(data: string): Writable;  
}  
  
class FileLogger implements Writable {  
  write(data: string): Writable {  
    console.log(`Write ${data}`);  
    return this;  
  }  
}
```

TypeScript - Génériques



- Permet de paramétrer le type de certaines méthodes

```
class Stack<T> {  
  private data: Array<T> = [];  
  push(val: T) {  
    this.data.push(val);  
  }  
  pop(): T {  
    return this.data.pop();  
  }  
  peek(): T {  
    return this.data[this.data.length - 1];  
  }  
}  
  
const strStack = new Stack<string>();  
strStack.push('html');  
strStack.push('body');  
strStack.push('h1');  
console.log(strStack.peek().toUpperCase()); // H1  
console.log(strStack.pop().toUpperCase()); // H1  
console.log(strStack.peek().toUpperCase()); // BODY
```

TypeScript - Décorateurs



- Permettent l'ajout de fonctionnalités aux classes ou membre d'une classe en annotant plutôt que via du code à l'utilisation
- Norme à l'étude en JavaScript par le TC39
<https://github.com/tc39/proposal-decorators>
- Supporté de manière expérimentale en TypeScript
- Pour activer leur support il faut éditer le tsconfig.json ou passer une option au compilateur

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "experimentalDecorators": true  
  }  
}
```

TypeScript - Décorateurs



▸ Décorateur de classes

```
'use strict';

function Freeze(FunctionConstructor) {
    Object.freeze(FunctionConstructor);
}

@Freeze
class MyMaths {
    static sum(a, b) {
        return Number(a) + Number(b);
    }
}

try {
    MyMaths['subtract'] = function(a, b) {
        return a - b;
    };
}
catch(err) {
    // Cannot add property subtract, object is not extensible
    console.log(err.message);
}
```



▸ Décorateur de propriétés

```
import 'reflect-metadata';

const minLengthMetadataKey = Symbol("minLength");

function MinLength(length: number) {
  return Reflect.metadata(minLengthMetadataKey, length);
}

function validateMinLength(target: any, propertyKey: string): boolean {
  const length = Reflect.getMetadata(minLengthMetadataKey, target, propertyKey);
  return target[propertyKey].length >= length;
}

class Contact {
  @MinLength(7)
  protected firstName;

  constructor(firstName: string) {
    this.firstName = firstName;
  }

  isValid(): boolean {
    return validateMinLength(this, 'firstName');
  }
}

const romain = new Contact('Romain');
console.log(romain.isValid()); // false
```



Configuration

Configuration - Introduction



- Le compilateur peut se configurer :
 - on peut le rendre plus ou moins sensible à certaines erreurs
 - on peut agir sur le code généré
 - ...
- Pour générer un fichier de configuration on peut utiliser la commande :
`tsc --init`
- Les fichiers de configuration TypeScript sont au format JSON mais acceptent les commentaires



- *target*
permet de définir quelle syntaxe sera utilisée dans le code généré
ex: es3, es5, es6/es2015, es2016, ..., es2022...
En général les toolchains privilégie Babel pour la compatibilité
- *module*
le format du module en sortie
ex: amd, commonjs, es2015 (import/export), es2020 (import(), import.meta), es2022 (top level await), node16 (dépend de la clé type du package.json)
- *moduleResolution*
comment seront résolus les imports
ex: node16 (dans Node.js), bundler (les extensions sont optionnelles pour les imports ESM)
- *outDir*
répertoire où sera généré le code compilé