TypeScript

1 Objectifs pédagogiques

À l'issue de cette formation, vous serez en mesure d'utiliser typescript
La documentation est très bien faite, elle est explicite et vous permet souvent de faire des tests en ligne : https://www.typescriptlang.org/docs/handbook/2/basic-types.html

2 Introduction

Le javascript est devenu incontournable dans le cadre de la programmation web. Les applications générées en JS sont

devenues de plus en plus complètes et complexes.

Leur maintenance s'est avérée difficile car le langage est trop permissif et qu'il n'est pas facile de savoir ce que renvoie une fonction ou ce qu'elle attend en paramètre.

TypeScript, développé par Microsoft, répond à cette problématique et il permet une transition progressive du js vers le ts.

En bref Typescript :

- est développé par Microsoft
- est une surcouche à JavaScript
- est open source
- est un langage à typage fort
- doit être transpilé*
  *Un compilateur est un logiciel qui convertit un langage de haut niveau en langage assembleur de bas niveau. Un transpilateur est un autre logiciel, parfois appelé compilateur source à source, qui convertit un langage de haut niveau en un autre langage de haut niveau.

2.1 Avantages de typescript

- Code plus lisible
- Code plus facilement maintenable
- Beaucoup moins de bug grâce au transpileur

2.2 Succès de TypeScript

- Visual Studio Code est écrit en TypeScript
- Depuis quelques années Angular est écrit en TS
- React et Vue js peuvent être codé en TS

3 Principales fonctionnalités

- Typage statique
- Inférence de type : TS est capable de détecter automatiquement le type en fonction de la valeur
- Transpilation : TS transforme le code en un JS rétrocompatible
- Fonctionnalités majeures :
    - Types,
    - interfaces,
    - combinaisons de types,
    - modules,
    - namespaces,
    - décorateurs,
    - mixins,
    - types avancés

4 Le compilateur

Installation globale

```
npm i -g typescript
```

Installation locale

```
npm i -D typescript
```

4.1 Premier test de compilation

Fichier source index.ts :

```
let test:string = "Hello World!";
```

Compilation :

```
tsc index.ts
```

Résultat dans un fichier de destination

```
var test = "Hello World!";
```

4.1.1 Paramétrages du compilateur

```
tsc --init
```

Modification du fichier tsconfig.json

```
{
  "compilerOptions": {
    "target": "es2016",
    "module": "commonjs",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true
  }

}
```

où :

- Target : indique la version du js dans laquelle le code sera transpilé
- Module : Les modules CommonJS constituent le moyen original de packager du code JavaScript pour Node.js. Node.js prend également en charge la norme de modules ECMAScript utilisée par les navigateurs et autres environnements d'exécution JavaScript.
- EsModuleInterop :
  Par défaut, cette valeur est fausse. Lorsque la valeur est définie comme true, le compilateur TypeScript émettra du JavaScript supplémentaire qui vérifie l'objet exporté pour détecter s'il s'agit d'une exportation par

défaut ou d'un objet d'exportation qui a été écrasé, puis l'utilise en conséquence.

- forceConsistentCasingInFileNames :
  TypeScript suit les règles de sensibilité à la casse du système de fichiers sur lequel il s'exécute. Cela peut être problématique si certains développeurs travaillent dans un système de fichiers sensible à la casse et d'autres non. Si un fichier tente d'importer fileManager.ts en spécifiant ./FileManager.ts, le fichier sera trouvé dans un système de fichiers insensible à la casse, mais pas sur un système de fichiers sensible à la casse.Lorsque cette option est définie, TypeScript émettra une erreur si un programme tente d'inclure un fichier dans une casse différente de celle du disque.
    - SkipLibCheck :
      Permet d’ignorer la vérification du type des fichiers de déclaration. Cela peut permettre de gagner du temps lors de la compilation au détriment de la précision du système de types. Par exemple, deux bibliothèques pourraient définir deux copies du même type de manière incohérente. Plutôt que de procéder à une vérification complète de tous les fichiers d.ts, TypeScript vérifiera le code auquel vous faites spécifiquement référence dans le code source de votre application.

4.2 Lancer la compilation en utilisant le fichier tsconfig.json

```
tsc
```

Voir les options de compilation : https://www.typescriptlang.org/docs/handbook/compiler-options.html

Explication de quelques options de compilation :

- files
  Permet de renseigner un tableau de fichiers ts à inclure dans la compilation
- include
  Permet de renseigner un tableau de fichiers/patterns à compiler
- exclude
  Permet de renseigner un tableau de fichiers/patterns à exclure de la compilation (même s'ils matchent dans l'option include)
- allowJS
  Autorise ou non l'inclusion de fichier JS dans le projet sans générer d'erreur
- module
  Permet de définir le système de module pour le projet

(CommonJS, UMD, AMD, System, ESNext/ES2020)

4.3 Exécution d'un fichier typescript avec ts-node

```
npm install -g ts-node
ts-node fichier.ts
```

5 Les types en TypeScript

5.1 Rappel du JavaScript : typage dynamique

```
let test = "Hello World!";
test = 3;
test = {"name": "Bob"}
```

Inconvénients :

- Erreurs fréquentes de type.
- Nécessite beaucoup plus de tests.
- Aucune documentation vérifiée par le compilateur

5.2 Syntaxe

```
variable: type;
```

Exemple

```
let test: string = "Hello World!";

function hello(text: string): string {
  return `Hello ${text}`;
}

hello("Bob");
```

5.3 Les types communs

```
// Types primitifs
let hidden: boolean = false;
let size: number = 16;
let text: string = "Message";

// Tableaux
const fruits: string[] = ["Orange", "Banane"];

// Tuples
const point: [number, number] = [12.3, 54.23];
let contact: [string, number | string, boolean] =
["Bob", 72, false];

// Enumération
enum Color {
  white,
```

```
  blue,
  red,
}
let c: Color = Color.blue;
```

5.4 Void, undefined, null, never

```
// Quand on ne connaît pas le type à l'avance
let test: any = 4;
test = "Message";

let n: null = null;

let u: undefined = undefined;

// Fonction qui ne renvoie rien
function testVoid(text: string): void {
  console.log(`Hello ${text}`);
}

// Fonction qui génère une exception
function error (message: string): never {
  throw new Error(message);
}
```

5.5 Typer les promesses

Lorsqu’une fonction renvoie une promesse, il est possible de typer le résultat de cette
dernière dans le cas où elle est tenue. Exemple pour une promesse de chaîne de caractères :

```
function iReturnPromiseAfter1Second():
Promise<string> {&amp;nbsp;&amp;nbsp;&amp;nbsp;
&amp;nbsp; return new Promise(
&amp;nbsp;&amp;nbsp;&amp;nbsp; (resolve) ⇒ {setTimeout(() ⇒
resolve(1), 1000); }
&amp;nbsp; );
}
(async () ⇒ {
        const result = await
iReturnPromiseAfter1Second();
        console.log(result);
})();
```

5.6 Les alias de type

Nous utilisons des types d'objet et des types d'union en les écrivant directement dans des annotations de type. C'est pratique, mais il est courant de vouloir utiliser le même type plus d'une fois et d'y faire référence par un seul nom.

Un alias de type est exactement cela : un nom pour n’importe quel type. La syntaxe d'un alias de type est :

```
type Point = {
        x: number;
        y: number;
};
// Exactly the same as the earlier example
function printCoord(pt: Point) {
console.log("The coordinate's x value is " +
pt.x);
console.log("The coordinate's y value is " +
pt.y);
}
printCoord({ x: 100, y: 100 });
```

5.7 Types génériques

```
// On utilise souvent les types génériques quand
on ne connaît pas le type à l'avance mais qu'il
dépend du type de l'argument. Ex :

function identity<T>(arg: T): T {
  return arg;
}
// Dans le code ci-dessus, nous avons créé la
variable de type  « T » qui va être assigné lors
// de l'appel de la fonction. Par exemple, si
l'appel prend la forme :
identiy("Bob");// Alors le retour sera de type
string
```

Exemple avancé de type générique :

```
type RequireAtLeastOne<T> = {
  [K in keyof T]-?: Required<Pick<T, K>> &
Partial<Pick<T, Exclude<keyof T, K>>>
}[keyof T]

interface TaskInterfacePost {
  description: string;
  done: boolean;
  order: number;
}

const test:RequireAtLeastOne<TaskInterfacePost> =
{
  done: true
}
```

6 Les interfaces avec TypeScript

Interface pour un objet avec une méthode

```
interface User {
  firstname: string;
  sayHello(lastname: string): void;
}

const user1: User = {
  firstname: "Bob",
```

```
  sayHello(lastname: string) {
    console.log(`Bonjour, je
m'appelle ${this.firstname} ${lastname}`);
  }
}

user1.sayHello("Dylan");
```

Interface pour fonction

```
interface Searching {
  (critere1: string, critere2: string):
boolean;
}

const searchByName:Searching = function(name,
orderBy){
  console.log("Je cherche !")
  return true;
}
searchByName("toto", 1); // oups -)
```

Interface pour classe

```
interface Checkable {
  check(name: string): boolean;
}

class NameChecker implements Checkable {
```

```
  check(s:string) {
    return s.toLowerCase() ===
"ok";
  }

}
const nm = new NameChecker();
console.log(nm.check("ok"));
```

6.1 Types utilitaires

6.1.1 Record<Keys, Type>

Construit un **type d'objet** dont les clés de propriété sont "Keys" et dont les valeurs de propriété sont de type "Type".

Dans l'exemple ci-dessous, la propriété privée "dependencies" a pour type un objet dont les clés sont des "chaînes de caractères" et dont les valeurs sont de n'importe quel type.

```
const dependencies: Record<string, any> = {"name":
"Bob"};
```

Record permet également de construire un type avec les mêmes propriétés mais avec un autre type pour ces propriétés. Exemple :

```
interface FormValue {
        value: string;
        valid: boolean;
}
interface PonyModel {
        name: object;
        color: object;
        speed: object,
}
const pony: Record<keyof PonyModel, FormValue> = {
        name: { value: 'Rainbow Dash', valid: true
},
        color: { value: 'blue', valid: true },
        speed: { value: '45', valid: true }
};
```

6.1.2 ## `Partial<Type>`

Construit un type où toutes les propriétés du Type sont facultatives. Cet utilitaire renverra un type qui représente tous les sous-ensembles d'un type donné.

6.1.2.1.1 Example

```
interface Todo {
        title: string;
        description: string;
}
function updateTodo(todo: Todo, fieldsToUpdate:
```

```
Partial<Todo>) {
        return {  ...todo,  ...fieldsToUpdate };
}
const todo1 = {
        title: "organize desk",
        description: "clear clutter",
};
const todo2 = updateTodo(todo1,
        { description: "throw out trash" }
);
console.log(todo2);
```

6.1.2.2 Omit<Type, Keys>

Construit un type en sélectionnant l'ensemble des propriétés d'un type et en supprimant ensuite des propriétés keys (chaîne littérale ou union de chaînes littérales) Keys. C'est l'opposé de "Pick"

```
interface Todo {
        title: string;
        description: string;
        completed: boolean;
        createdAt: number;
}

type TodoPreview = Omit<Todo, "description">;

const todo: TodoPreview = {
```

```
        title: "Clean room",
        completed: false,
        createdAt: 1615544252770,
};

console.log(todo);
```

6.2 Readonly

Readonly rend toutes les propriétés d'un objet readonly :

```
interface Todo {
        title: string;
}

const todo: Readonly<Todo> = {
        title: "Delete inactive users",
};

//todo.title = "Hello";
console.log(todo);
```

7 Les classes avec TypeScript

Déclaration des propriétés sans constructeur :

```
class Base {
  private _x = 12;
  get x() {
```

```
    console.log("dans le getter");
    return this._x;
  }
  set x(value: number) {
    console.log("dans le setter");
    this._x = value;
  }
}
const b = new Base();
b.x = 25;
console.log(b.x);
```

Déclaration des propriétés via le constructeur :

```
class Person {
  constructor(private name: string) {
    // No body necessary
  }
  getName() {
    return this.name;
  }
}
const p = new Person("Bob");
console.log(p.getName());

//console.log(p.name); // génère une erreur
```

8 Fichier d.ts

Un fichier .d.ts est un fichier de déclaration TypeScript. Ces fichiers sont utilisés pour fournir des informations sur des types dans le cadre d'une une API écrite en JavaScript. L'idée est de fournir aux développeurs TypeScript un moyen de disposer d'IntelliSense, de la vérification de type au moment de la compilation et d'autres fonctionnalités du compilateur TypeScript tout en travaillant avec les bibliothèques JavaScript existantes.

Par exemple, si vous disposez d'une bibliothèque JavaScript, myLib.js, vous pouvez créer un fichier myLib.d.ts avec des déclarations de type pour l'API exposée par myLib.js. Ensuite, lorsqu'un développeur TypeScript utilise myLib.js, il peut également inclure myLib.d.ts pour les informations de type.

En résumé, les fichiers .d.ts constituent un élément fondamental de l'utilisation des bibliothèques JavaScript existantes dans TypeScript, et ils sont également utilisés dans la définition de la propre bibliothèque standard de TypeScript.

8.1 Intellisense

IntelliSense est un outil de complétion de code embarqué par défaut dans VSC et qui fournit principalement les fonctionallités suivantes :

1. **Code Completion**: fournit une liste de suggestions au fur et à mesure que vous tapez, vous aidant à terminer votre code plus rapidement. Il peut suggérer des variables, des fonctions, des classes, des modules, des mots-clés, etc.
2. Paramètres : lorsque vous appelez une fonction ou une méthode, IntelliSense affiche les paramètres attendus, leurs types et toutes les valeurs par défaut.
3. **Quick Info** : lorsque vous survolez une variable ou une fonction, IntelliSense affiche une info-bulle contenant des informations pertinentes telles que le type, les paramètres, les valeurs de retour et une brève description.
4. **List Members**: répertorie les propriétés et les méthodes disponibles pour un objet.
5. **Automatic Import Statements**: lorsque vous sélectionnez une suggestion pour une classe ou une fonction à partir d'un autre module, IntelliSense peut ajouter automatiquement l'instruction d'importation en haut de votre fichier.

Dans TypeScript, IntelliSense est particulièrement utile car il peut fournir des informations de type, ce qui permet de détecter les problèmes potentiels au moment de la compilation plutôt qu'au moment de l'exécution. Il utilise des

fichiers de déclaration TypeScript (.d.ts) pour collecter ces informations de type.

9 Rétrécissement ou Narrowing

cf : https://www.typescriptlang.org/fr/docs/handbook/2/narrowing.html
Le « rétrécissement » est le processus d'affinement du type d'une variable. Cela se fait souvent à l'aide de 'type guards', qui sont des expressions qui effectuent une vérification à l'exécution (runtime check) et garantissent le type d'une variable.

Le rétrécissement est utile car TypeScript utilise un système de types structurels, ce qui signifie qu'il se soucie de la forme de vos données, et non de leur type nominal. Cela signifie que si vous avez une variable d'un type large (comme n'importe quelle variable ou un type union), **TypeScript vous permettra uniquement d'accéder aux propriétés communes à tous les types possibles de la variable**.

En limitant le type d'une variable, on peut alors accéder à davantage de propriétés.

Voici quelques exemples de rétrécissement :
opérateur typeof :

```
let value: string | number;
  if (typeof value === 'string') {
    // TypeScript knows that value is a string
here
    console.log(value.toUpperCase()); // OK
  } else {
    // TypeScript knows that value is a number
here
    console.log(value.toFixed(2)); // OK
  }
```

Opérateur instanceof :
JavaScript dispose d'un opérateur permettant de vérifier si une valeur est ou non une « instance » d'une autre valeur. Plus précisément, en JavaScript,

```
 x instanceof Foo
```

vérifie si la chaîne de prototypes de x contient Foo.prototype.

```
class Dog {
  bark() {
    console.log('Woof!');
  }
}

class Cat {
  meow() {
```

```
    console.log('Meow!');
  }
}
let pet: Dog | Cat;

if (pet instanceof Dog) {
  // TypeScript knows that pet is a Dog here
  pet.bark(); // OK
} else {
  // TypeScript knows that pet is a Cat here
  pet.meow(); // OK
}
```

Rétrécissement par véracité

La véracité (anglais *truthiness*) est le fait d'utiliser des expressions (sans qu'elles retournent nécessairement un booléen), des `&&`, des `||`, des `if`, des négations booléennes (`!`), et plus encore. Par exemple, les `if` ne s'attendent pas spécialement à ce que l'expression retourne un `boolean`.

```
function getUsersOnlineMessage(numUsersOnline:
number) {
  if (numUsersOnline) {
    return `Il y a ${numUsersOnline} utilisateurs
en ligne !`;
  }
```

```
  return "C'est désert.";
}
```

Custom Type guards :
On peut créer ses propres "guards type" pour affiner un type. Un "guard type" défini par l'utilisateur est une fonction qui renvoie une expression booléenne utilisée pour vérifier le type d'une variable (aussi appelé "type predicate").

```
interface Fish {
  swim: () => void;
}

interface Bird {
  fly: () => void;
}

function isFish(pet: Fish | Bird): pet is Fish {
  return (pet as Fish).swim !== undefined;
}

let pet: Fish | Bird;

if (isFish(pet)) {
  // TypeScript knows that pet is a Fish here
  pet.swim(); // OK
} else {
  // TypeScript knows that pet is a Bird here
```

```
  pet.fly(); // OK
}
```

10 Assertions de type

Les assertions de type sont un moyen de dire au compilateur « faites-moi confiance, je sais ce que je fais ». Une assertion de type est comme une conversion de type dans d'autres langages, mais elle n'effectue aucune vérification ou restructuration particulière des données. Il n'a aucun impact sur l'exécution et est utilisé uniquement par le compilateur. TypeScript suppose que vous, le programmeur, avez effectué toutes les vérifications spéciales dont vous avez besoin.

Par exemple, si vous utilisez document.getElementById, TypeScript sait seulement que cela renverra un HTMLElement, mais vous savez peut-être que votre page aura toujours un HTMLInputElement avec un ID donné.

Dans cette situation, vous pouvez utiliser une assertion de type pour spécifier un type plus spécifique :

```
const myInput = document.getElementById("my-
input") as HTMLInputElement;
```

Vous pouvez également utiliser la syntaxe des chevrons (sauf si le code est dans un fichier .tsx), qui est équivalente :

```
const myInput =
<HTMLInputElement>document.getElementById("my-
input");
```

Attention, étant donné que les assertions de type sont supprimées au moment de la compilation, aucune vérification d'exécution n'est associée à une assertion de type. Il n'y aura pas d'exception ou de valeur nulle générée si l'assertion de type est fausse.

TypeScript autorise uniquement les assertions de type qui se convertissent en une version plus spécifique ou moins spécifique d'un type. Cette règle évite les coercitions « impossibles » telles que :

```
const x = "bonjour" as number ;//impossible !!!
```

La conversion du type « chaîne » en type « numéro » peut être une erreur car aucun des deux types ne se chevauche suffisamment avec l'autre. Si cela était intentionnel, convertissez d'abord l'expression en « unknown ».

Parfois, cette règle peut être trop conservatrice et interdire des coercitions plus complexes qui pourraient être valables.

Si cela se produit, vous pouvez utiliser deux assertions, d'abord pour n'importe lequel (ou inconnu, que nous présenterons plus tard), puis pour le type souhaité :

```
const a = expr as any as T;
```

11 Décorateurs et annotations

Référence : https://www.tektutorialshub.com/angular/angular-decorators/

Un décorateur est une façon de faire de la méta-programmation.

En **TypeScript**, un décorateur est une fonction qui ajoute des comportement à une méthode, une classe, des propriétés ou des paramètres existants. Il permet de modifier le comportement d'un objet, sans changer son code d'origine, mais en étendant ses fonctionnalités.

Attention pour que les décorateurs fonctionnent, il faudra donner la valeur "esnext" à la propriété module et true à la propriété "experimentalDecorators" dans votre fichier tsconfig.json. Exemple de fichier tsconfig.json fonctionnel avec les décorateurs :

```
{
  "compilerOptions": {
```

```
    "target": "es2016",
    "module": "esnext",
    "esModuleInterop": true,
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "skipLibCheck": true,
    "experimentalDecorators": true
  },
  "include": [
    "src/*.ts"
  ],
  "exclude": [
    "node_modules"
  ]
}
```

Un exemple TypeScript fonctionnant avec les décorateurs (cf https://devblogs.microsoft.com/typescript/announcing-typescript-5-0/#decorators) :

```
function loggedMethod(target: any, propertyKey:
string, descriptor: PropertyDescriptor) {
  let originalMethod = descriptor.value;
  descriptor.value = function (...args:
any[]) {
      console.log(`Called
${propertyKey} with args:
${JSON.stringify(args)}`);
      let result =
```

```
originalMethod.apply(this, args);
&amp;nbsp; &amp;nbsp; &amp;nbsp; console.log(`Method
${propertyKey} returned
${JSON.stringify(result)}`);
&amp;nbsp; &amp;nbsp; &amp;nbsp; return result;
&amp;nbsp; }

&amp;nbsp; return descriptor;
}

class Person {
&amp;nbsp; name: string;
&amp;nbsp; constructor(name: string) {
&amp;nbsp; &amp;nbsp; &amp;nbsp; this.name = name;
&amp;nbsp; }

&amp;nbsp; @loggedMethod
&amp;nbsp; greet() {
&amp;nbsp; &amp;nbsp; &amp;nbsp; console.log(`Hello, my name
is ${this.name}.`);
&amp;nbsp; }
}

const p = new Person("Ron");

p.greet();
```

Dans cet exemple, @loggedMethod est un décorateur qui encapsule la méthode greet. Il enregistre les arguments et le

résultat de la méthode et en profite pour exécuter des console.log() avant et après le console.log() de la méthode originale.

Dans **Angular**, les termes « annotation » et « décorateur » sont souvent utilisés de manière interchangeable, mais ils ont des significations légèrement différentes.

Les décorateurs sont une fonctionnalité de TypeScript et sont largement utilisés dans Angular. **Ce sont des fonctions** qui peuvent être préfixées par le symbole @ et placées immédiatement avant le code à décorer.
Des exemples de décorateurs dans Angular incluent **@Component**, **@NgModule**, **@Input**, **@Output**, etc.

Annotation : les annotations dans Angular sont les métadonnées définies sur la classe à l'aide du décorateur. Lorsque vous utilisez un décorateur, vous ajoutez des métadonnées à cette classe, propriété ou méthode. Ces métadonnées sont utilisées par Angular pour comprendre le but de la classe ou de la propriété et comment elle devrait fonctionner. **En d'autres termes, les annotations sont les données que vous transmettez à la fonction décorateur.**
Par exemple

```
 @Component({selector: 'my-app', template:
'<h1>Hello</h1>'})
```

```
  // ici l'objet `{selector: 'my-app', template:
'<h1>Hello</h1>'}` est l'annotation
```

En résumé, les décorateurs sont les fonctions préfixées par @ qui vous permettent d'ajouter des métadonnées/annotations à vos classes, propriétés, méthodes ou paramètres. Les métadonnées/annotations sont les données réelles que vous transmettez à ces décorateurs.

Le décorateur est une fonctionnalité Typescript et ne fait toujours pas partie du Javascript. Il est encore au stade de proposition.

Le but des décorateurs Angular est de stocker des métadonnées sur une classe, une méthode, une propriété ou un paramètre. Lorsque vous configurez un composant, vous fournissez des métadonnées pour cette classe qui indiquent à Angular que vous avez un composant et que ce composant a une configuration spécifique.

En Angular, on utilisera les décorateurs fournis par le framework. Leur rôle est assez simple: ils ajoutent des métadonnées à nos classes, propriétés ou paramètres pour par exemple indiquer "cette classe est un composant", "cette dépendance est optionnelle", "ceci est une propriété spéciale du composant", etc. Ce n’est pas obligatoire de les utiliser, car on peut toujours ajouter les métadonnées manuellement

(pur ES5), mais le code sera plus élégant avec des décorateurs, comme ceux proposés par TypeScript.

Par exemple, l'AppComponent, comme tout component, doit savoir où trouver son modèle. Pour ce faire, nous utilisons le paramètre templateUrl de la directive @Component. Le décorateur @Component accepte également les valeurs pour styleUrls, encapsulation, changeDetection, etc., qu'Angular utilise pour configurer le composant.

12 Installation d'un environnement TypeScript

12.1 Créer un répertoire "myApp" par exemple et placez vous sur ce répertoire

12.2 Installer les dépendances

```
npm install --save-dev typescript webpack webpack-cli rxjs html-webpack-plugin ts-loader webpack-dev-server
```

12.3 Créer le fichier tsconfig.json

```
{
  "compilerOptions": {
    "outDir": "./dist/",
    "sourceMap": true,
    "noImplicitAny": true,
    "module": "commonjs",
    "target": "es6",
    "jsx": "react"
  },
  "include": [
    "./src/**/*"
  ]
}
```

12.4 Créer le fichier webpack.config.js

```
const path = require('path');
const HtmlWebpackPlugin = require('html-webpack-
plugin');

module.exports = {
  mode: "development",
  devServer: {open: true},
  entry: './src/index.ts',
  module: {
    rules: [
      {
```

```
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
    ],
  },
  resolve: {
    extensions: ['.tsx', '.ts', '.js'],
  },
  output: {
    filename: 'bundle.js',
    path: path.resolve(__dirname, 'dist'),
  },
  plugins: [new HtmlWebpackPlugin({ template:
__dirname + '/src/index.html' })],
};
```

12.5 Créer le répertoire src et placer à l'intérieur les fichiers index.html et index.ts

Grâce à webpack, le fichier index.html appellera automatiquement le fichier index.ts qui sera compilé automatiquement.

12.6 Modifier le fichier package.json

```
{
  "name": "tp02autocomplete",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" &&
exit 1",
    "start": "webpack-dev-server",
    "build": "NODE_ENV=production webpack"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "devDependencies": {
    "html-webpack-plugin": "^5.5.3",
    "rxjs": "^7.8.1",
    "ts-loader": "^9.5.1",
    "typescript": "^5.3.2",
    "webpack": "^5.89.0",
    "webpack-cli": "^5.1.4",
    "webpack-dev-server": "^4.15.1"
  }
}
```

12.7 Ajout de bootstrap

```
npm i bootstrap@5
```

Nous allons avoir besoin de "loaders" pour charger des fichiers css

```
npm install  css-loader style-loader  --save-dev
```

Il faut enfin modifier le fichier de config de webpack (webpack.config.js)

```
module: {
    rules: [
      {
        test: /\.tsx?$/,
        use: 'ts-loader',
        exclude: /node_modules/,
      },
      {
        test: /\.css$/,
        use: ["style-loader", "css-loader"],
      },
    ],
  },
```

13 Quelques exercices avec TS

13.1 Convertisseur

https://coopernet.fr/formation/javascript/tp-convertisseur

13.2 Contrôle de formulaire

https://coopernet.fr/formation/javascript/tp-controle-formulaire

13.3 Utilisation d'une API REST

https://coopernet.fr/formation/javascript/tp-api-countries

14 Quelques types spécifiques à React

14.1 Typer les states

```
function App(): React.JSX.Element  {
        // déclaration de states de leur setter respectif
        const [tasks, setTasks] = useState<TaskInterface[]>([]);
        const [error, setError] = useState<string>("");
        // ...  suite du composant ici
}
```

Dans le code ci-dessus, on comprend que :

- la fonction doit renvoyer du JSX
- l'on type le paramètre de la fonction générique useState lors de son appel.

C'est ce qui va nous garantir que le state "tasks" implémente l'interface "TaskInterface" ou que "error" est un state de type "string". Cela implique que le type générique est utilisé par useState pour typer le premier élément du tableau qu'il retourne.

Typer un composant fonctionnel

```
import TaskInterface from
'./../interfaces/TaskInterface';
interface TaskProps {
  task: TaskInterface;
  onClickValidate: (taskId: number) ⇒ void;
}

const Task: React.FC<TaskProps> = ({ task,
onClickValidate }) ⇒ {
  return (
    <section className='d-flex justify-content-
between my-3'>
                // jsx ici
    </section>
  );
}
```

Dans le code ci-dessus, on comprend que la const Task est de type Fonction Component et qu'elle attend des paramètres de type "TaskProps"