TypeScript

Table des matières

TypeScript est un « superset de JavaScript » permettant d'écrire en « orienté objet » (interfaces, classes, constructeurs, etc.) avec un typage fort. On crée un fichier *.ts, le compilateur TypeScript (tsc) génère alors le code JavaScript correspondant. Les fichiers de définitions TypeScript *.d.ts (pour Angular par exemple) permettent notamment d'avoir l'IntelliSense.

Site, documentation

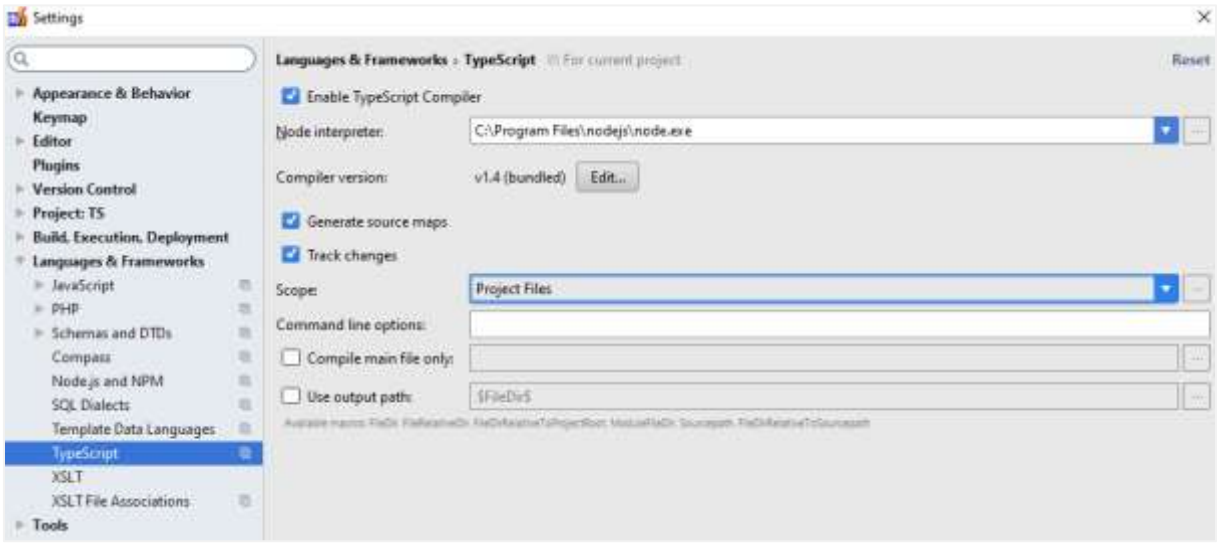
1. Installation

En global avec NPM

```
npm install -g typescript
```

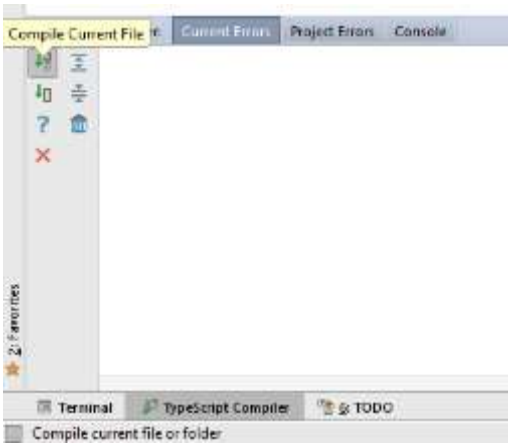
2. Editeurs de texte/ EDI

- Visual Studio, Visual Studio Code
- WebStorm/ PHPStorm

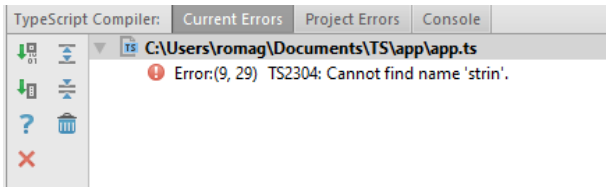


Modifier la version du compiler « edit » … « C:\Users\[user] \AppData\Roaming\npm\node_modules\typescript\lib »

On peut désormais compiler directement depuis le panneau « Typescript compiler » et sélectionner la compilation automatique (à chaque changement)



Affiche également les erreurs pendant la saisie



- Extensions pour **Brackets** « Brackets TypeScript »
- Etc.

3. Compilation

Créer le projet avec des fichiers *.ts …

```
class Item{

    constructor(public name:string){}

    doSomething():string{
        return this.name;
    }
}

var item1 = new Item('Hello');
console.log(item1.doSomething());
```

Depuis une invite de commande naviguer vers le dossier du projet puis indiquer le fichier à compiler

```
tsc
```

Ou tsc avec le fichier à compiler

```
tsc app.ts
```

Pour un watch

```
tsc -w
```

Ou avec un éditeur/edi.

Pour avoir l'aide

```
tsc --help
```

Le JavaScript généré

```
var Item = (function () {
    function Item(name) {
        this.name = name;
    }
    Item.prototype.doSomething = function () {
        return this.name;
    };
    return Item;
})();
var item1 = new Item('Hello');
console.log(item1.doSomething());
//# sourceMappingURL=app.js.map
```

On peut créer une page HTML en référençant le fichier js (exemple « app.js »), ou simplement tester avec Node « node app.js »

4. Fichier de configuration « tsconfig.json »

Documentation, schema

En choisissant ES6 le code généré comportera des class, …

```
{
  "compilerOptions": {
    "target": "ES5",
    "module" : "commonjs",
    "sourceMap": false
  },
  "files": [
    "app/app.ts",
    "app/MyInterface.ts"
  ]
}
```

« **target** » par défaut actuellement es3 ?

« **module** »

Valeurs possibles : "commonjs", "amd", "umd", "system", "es6", "es2015", "none"

Avec « commonjs » défini en module

Les imports sont convertis en « require »

```
var Item_1 = require('./Item');
var item1 = new Item_1.default('abc');
item1.methodOne();
```

Avec « amd » défini en module

```
define(["require", "exports", './Item'], function (require, exports,
Item_1) {
    "use strict";
    var item1 = new Item_1.default('abc');
    item1.methodOne();
});
```

Depuis une invite de commande, naviguer jusqu'au projet et

```
tsc
```

Les fichiers JS sont générés

5. Variables

- var
  - « hoisted » to the top of fucntion
- let et const
  - seulement accessible dans le block ou il est déclaré
  - not « hoisted »

```
function demo(){

    if(true){
        var myvar = 'Hello var!';
        let letvar = 'Hello let!';
    }
    console.log(myvar);
    console.log(letvar); // error (pas de compilation possible)
}
```

Inférence

```
let mystring = 'Hello';
mystring = 10; // error
```

Déclaration du type

```
let mystring : string = 'Hello';
```

6. Types

Basic types

- boolean
- number
- string
- Tableau .. exemple

```
let myarray: number[] = [1, 2, 3];
// ou
let myarray: Array<number> = [1, 2, 3];
```

- any
- void

7. Fonctions avec TypeScript

- Paramètres requis, paramètres optionnels et paramètres avec valeurs par défaut. A la différence de JS ou tous les paramètres sont optionnels.
- On peut avoir plusieurs signatures de fonction.
- On peut indiquer le type de retour (string, void, number[], …)

Type de retour de fonction et paramètres typés, paramètres optionnels

```
function func(first :string, last?: string):string{
  return last ? 'hello ' + first + ' ' + last : 'hello ' + first;
}
console.log(func('Marie'));
console.log(func('Marie','Bellin'));
```

Rest parameters

Avec « … », on peut passer un nombre variable de paramètres qui seront réunis dans le tableau

```
function func(first:string, ...arr:number[]):void{

    for(var val in arr){
        console.log(val);
    }
}
func('Marie',1,2,3);
```

« Overloads »

```
function func(name:string):string;
function func(age:number):string;
function func(param:any):string{
    return 'Param : ' + param;
}
```

```
console.log(func('Marie'));
console.log(func(20));
```

Passer un objet

```
function func(item : { name : string }){
    console.log(item.name);
}

func({ name : 'abc'});
```

Callback

```
function func(name, callback) {
    // ...
    callback();
}
func('abc', function () {

});
```

8. Interface, objet, class

```
class Item {
    private _name;

    constructor(){
        this.name = '';
    }

    set name (value :string){
        this._name = value;
    }

    get name (){
        return this._name;
    }
    private methodOne() : void {

    }

    // public par défaut
    methodTwo () : void {

    }
    static methodStatic () : void{

    }
}
```

Getter, setter dispos quà partir de es5

Code généré

```
var Item = (function () {
    function Item() {
        this.name = '';
    }
    Object.defineProperty(Item.prototype, "name", {
        get: function () {
            return this._name;
        },
        set: function (value) {
            this._name = value;
        },
        enumerable: true,
```

```
            configurable: true
        });
        Item.prototype.methodOne = function () {
        };
        // public par dï¿½faut
        Item.prototype.methodTwo = function () {
        };
        Item.methodStatic = function () {
        };
        return Item;
}());
```

a. Héritage avec « extends »

```
class specificItem extends Item{

}
```

Code généré

```
var specificItem = (function (_super) {
    __extends(specificItem, _super);
    function specificItem() {
        _super.apply(this, arguments);
    }
    return specificItem;
}(Item));
```

b. Interface

Dans un fichier

```
interface MyInterface{
    name : string;
    age : number;
    doSomething: () => string;
}

export { MyInterface };
```

Utilisation

```
import { MyInterface } from './MyInterface';

let myItem : MyInterface = {
    name : 'abc',
    age : 20,
    doSomething : function() : string {
        return this.name;
    }
}
```

Avec « class » on utilise « implements »

```
import { MyInterface } from './MyInterface';

class Item implements MyInterface{
    constructor(public name:string,public age:number){}
    doSomething = function() : string {
        return this.name;
    }
}
```

c. Namespace

```
namespace MyNamespace {

    export class Item {

        constructor(public name:string) {
        }

        methodOne():void {
            console.log(this.name);
        }
    }

}

var item1 = new MyNamespace.Item('abc');
item1.methodOne();
```

Code généré

```
var MyNamespace;
(function (MyNamespace) {
    var Item = (function () {
        function Item(name) {
            this.name = name;
        }
        Item.prototype.methodOne = function () {
            console.log(this.name);
        };
        return Item;
    }());
    MyNamespace.Item = Item;
})(MyNamespace || (MyNamespace = {}));
var item1 = new MyNamespace.Item('abc');
item1.methodOne();
```

d. Equivalent en « modules »

Dans un fichier « Item.js » par exemple

```
export default class Item {

    constructor(public name:string) {
    }

    methodOne():void {
        console.log(this.name);
    }
}
```

Code généré

```
"use strict";
var Item = (function () {
    function Item(name) {
        this.name = name;
    }
    Item.prototype.methodOne = function () {
        console.log(this.name);
    };
    return Item;
}());
Object.defineProperty(exports, "__esModule", { value: true });
exports.default = Item;
```

Dans « app.js »

```
import Item from './Item';

var item1 = new Item('abc');
item1.methodOne();
```

Code généré

```
var Item_1 = require('./Item');
var item1 = new Item_1.default('abc');
item1.methodOne();
```

Générer un bundle avec Browserify

```
browserify app.js -o bundle.js
```

Et référencer seulement ce bundle dans la page HTML

9. Import de librairies tiers

a. Installation de librairies externes

1. Installer avec NPM le module désiré, exemple jQuery

Créer le « package.json » avec « npm init »

Puis

```
npm install jquery --save
```

2. Installer des fichiers de définition :

- Téléchargement direct (fichiers de définition existants)
- NuGet (avec Visual Studio)
- Tsd (mais utilisaer Typings est désormais recommandé)
- Typings

Typings

Github

Installation en global si ce n'est fait

```
npm install -g typings
```

Chercher un fichier de définition, exemple jQuery

```
typings search jquery
```

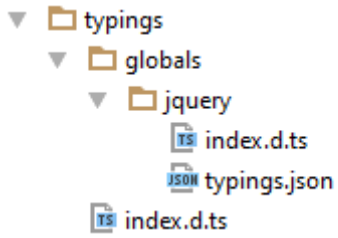
Pour installer un fichier de définition

« dt~ » suivi du nom du fichier

« --global » pour ne pas avoir d'erreur

```
typings install dt~jquery --global
```

Un dossier « typings » est créé



Référencer « index.d.ts » dans tsconfig.json

```
{
  "compilerOptions": {
    "target": "ES5",
    "module" : "commonjs",
    "sourceMap": false
  },
  "files": [
    "app/app.ts",
    "typings/index.d.ts"
  ]
}
```

Dans « app.ts »

Import puis utilisation

```
import * as $ from 'jquery';

$('.container').html('Hello!');
```

Générer les fichiers JS

```
tsc
```

Code généré

```
"use strict";

var $ = require('jquery');
$('.container').html('Hello!');
```

Comme le module utilisé est « **commonjs** »

```
browserify app.js -o bundle.js
```

Dans la **page HML**

Ne référencer que le bundle et ajouter une div pour la démo.

```
<body>
<div class="container"></div>

<script src="app/bundle.js"></script>
</body>
```

b. Créer des fichiers de définition pour une libraire

Documentation

Créer un fichier de définition pour un module qui n'en aurait pas

On crée un fichier *.d.ts et on définit la signature des méthodes

```
declare module "MyItem"{
    export function doSomething(name:string);
}
```

Ajout dans « tsconfig.json » dans les « files » le fichier de définition (ne sera pas compilé)

```
{
// etc.
"files": [
    "myitem.d.ts",
     // etc.
  ]
}
```

Puis utilisation, avoir l'IntelliSense et import

```
/// <reference path="myitem.d.ts" />
import * as MyItem from 'MyItem';
```

10. TypeScript avec Angular (attention assez ancien)

a. Installation

Inclus avec Visual Studio ou …

1. **npm** : Installer « Node Package Manager » Node.js
2. **tsc** : Installer TypeScript avec npm (pour pouvoir compiler les fichiers *.ts vers *.js)

```
npm install -g typescript
```

On peut vérifier la version de tsc installée depuis une invite de commande « tsc –v ».

```
C:\Users\romagny13>tsc -v
message TS6029: Version 1.5.3
```

3. **tsd** : Installer « TypeScript Definition Manager » definitelytyped

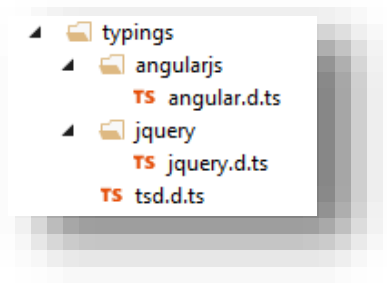
```
npm install tsd -g
```

Puis pour Installer des fichiers de definition *.d.ts… exemple avec Angular. Depuis une invite de commande naviguer jusqu'au dossier du projet puis …

```
tsd install angular --resolve --save
```

```
C:\Users\romagny13\Documents\Visual Studio 2015\Projects\TypeScriptDemo\TypeScriptDemo>tsd install angular --resolve --save
- angularjs / angular
  -> jquery > jquery
>> running install..
>> written 2 files:
  - angularjs/angular.d.ts
  - jquery/jquery.d.ts
```

Le fichier *.d.ts est ajouté dans un dossier « typings » du projet. Ce dossier est créé s'il n'existe pas.



… ainsi qu'un fichier tsd.json à la racine du projet

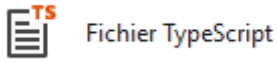
```
{
  "version": "v4",
  "repo": "borisyankov/DefinitelyTyped",
  "ref": "master",
  "path": "typings",
  "bundle": "typings/tsd.d.ts",
  "installed": {
    "angularjs/angular.d.ts": {
      "commit": "e95958ac847d9a343bdc8d7cbc796a5f6da29f71"
    },
    "jquery/jquery.d.ts": {
      "commit": "e95958ac847d9a343bdc8d7cbc796a5f6da29f71"
    }
  }
}
```

Création d'un projet TypeScript

Avec Visual Studio

Visual Studio 2015

Créer un projet « Application Web Asp.Net » vide.



Pour ajouter des fichiers TypeScript *.ts, soit ajouter un nouvel élément … fichier TypeScript, soit utiliser le raccourci du menu contextuel sur le projet.

La compilation des fichiers TypeScript *.ts se fait en même temps que la « génération » du projet.

Avec Visual Studio Code

Visual Studio Code

1. Créer un dossier (vide) pour le projet.
2. Ouvrir ce dossier avec Visual Studio Code
3. Créer un fichier **tsconfig.json**

```
{
        "compilerOptions": {
                "target": "ES5"
        }
}
```

Puis créer des fichiers *.ts. **Compilation** avec le raccourci « **CTRL + SHIFT + B** »

Exemple création d'un fichier « HelloWorld.ts »

```
function SayHello(){
        let message = "Bonjour!";
        alert(message);
}
```

4. A la première compilation configurer le « **Task Runner** »

… le fichier « tasks.json » est généré. La ligne args définit le fichier à compiler

```
// args is the HelloWorld program to compile.
"args": ["HelloWorld.ts"],
```

Le fichier TypeScript

Le fichier JavaScript

Création de la page d'index

```
<html>
	<body>
		<input type="button" onclick="SayHello();" value="Click!"/>
		<script src="HelloWorld.js"></script>
	</body>
</html>
```

Création d'un serveur avec « http-server »

Depuis une invite de commande

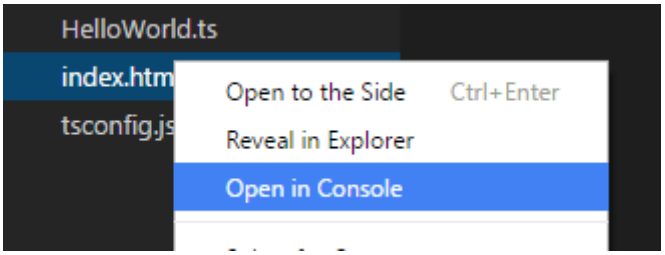
```
npm install -g http-server
```

Lancer le serveur avec la commande

```
http-server
```

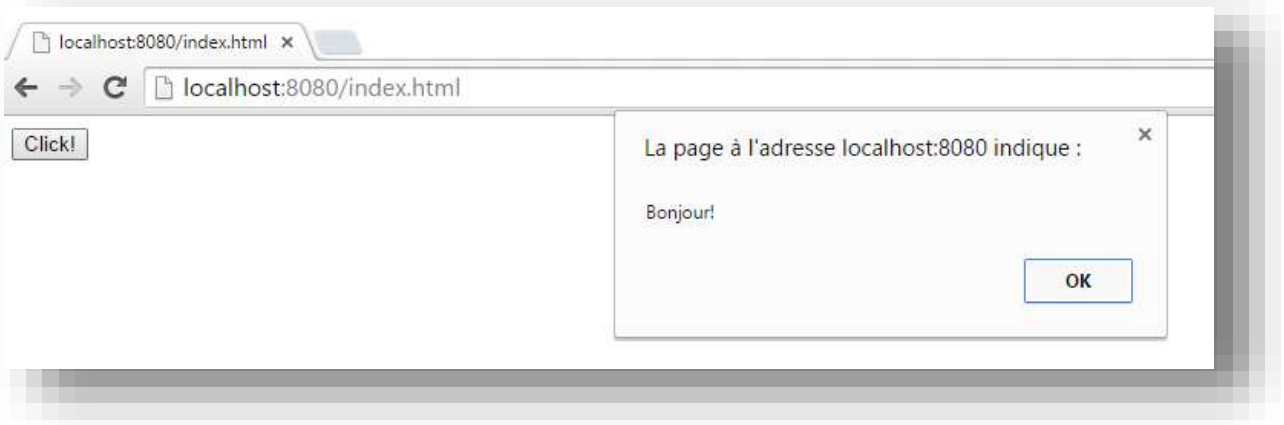
```
c:\Users\romagny13\Documents\TypeScriptDemo>http-server
Starting up http-server, serving ./ on: http://0.0.0.0:8080
Hit CTRL-C to stop the server
```

Note il est possible d'ouvrir une invite de commande depuis Visual Studio Code



(« ctrl+c » pour arrêter le serveur)

Dans browser



b. Fichiers de définition

tsd : installer les fichiers de définition *.d.ts pour chaque dépendance

```
tsd install angular --resolve –save
tsd install angular-route --resolve –save
tsd install angular-resource --resolve –save
// etc.
```

c. Modules

« Main »

```
module app {
    var main = angular.module("app", ["ngRoute"]);

    main.config(routeConfig);

    routeConfig.$inject = ["$routeProvider"];
    function routeConfig($routeProvider: ng.route.IRouteProvider): void {
        $routeProvider.when("/people", {
            templateUrl: "app/people/peopleView.html",
            controller: "PeopleController as vm"
        })
            .otherwise("/people");
    }
}
```

Injection

Routes

« ng » ou

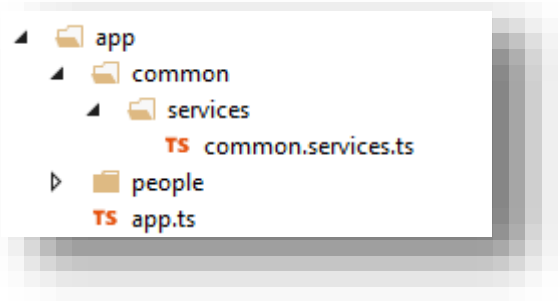
« Sous modules »

Exemple enregistrement du module « common.services »

```
module app.common {
    angular.module("common.services", ["ngResource"]);
}
```

Ajout de la dépendance au module principal

```
module app {
    var main = angular.module("app", ["ngRoute", "ngResource", "common.services"]);
    //etc.
}
```



d. Contrôleurs

```
module app {
    interface IPeopleController {
        title: string;
        people: any[];
    }
    class PeopleController implements IPeopleController {
        title: string;
        people: any[];

        constructor() {
            this.title = "Contacts Twitter";
            this.people = [
                {
                    "id": 1,
                    "name": "Marie Bellin",
                    "twitter": "mariebellin123"
                },
                {
                    "id": 2,
                    "name": "Jérôme Romagny",
                    "twitter": "romagny13"
                }];
        }
        // méthodes
    }
    angular
        .module("app")
        .controller("PeopleController", PeopleController);
}
```

Interface

Classe implémentant l'interface (propriétés en premier, puis constructeur et

Enregistremen

Avec entité

```
module app {
    interface IPeopleController {
        title: string;
        people: app.domain.Person[];
    }
    class PeopleController implements IPeopleController {
        title: string;
        people: app.domain.Person[];

        constructor() {
            this.title = "Contacts Twitter";
            this.people = [
                {
                    id: 1,
                    name: "Marie Bellin",
                    twitter: "mariebellin123"
                },
                {
                    id: 2,
                    name: "Jérôme Romagny",
                    twitter: "romagny13"
                }];
        }
        // méthodes
    }
    angular
        .module("app")
        .controller("PeopleController", PeopleController);
}
```

Ce n'est plus du JSON donc les propriétés ne sont plus entre guillements

e. Entités

Exemple création de « person.ts »

```
interface IPerson {
    name: string;
}
class Person implements IPerson {
    constructor(public name: string)
{
    }
}
```

Equivaut à

```
interface IPerson {
    name: string;
}
class Person implements IPerson {
    name: string;

    constructor(name: string) {
        this.name = name;
    }
}
```

« public » : déclaration + affectation « automatiques »

Autre exemple

```
interface IPerson {
    id: number;
    name: string;
    birthday: Date;
    twitter: string;
    sayHello(): void;
}
class Person implements IPerson {
    constructor(public id: number,
        public name: string,
        public birthday: Date,
        public twitter: string) {
    }
    sayHello(): void {
        alert("Bonjour " + this.name + "!");
    }
}
```

Méthode, définition du type de

Implémentation, constructeur avec

Avec namespace et export

```
module app.domain {
    export interface IPerson {
        id: number;
        name: string;
        birthday: Date;
        twitter: string;
        sayHello(): string;
    }
    export class Person implements IPerson {
        constructor(public id: number,
            public name: string,
            public birthday: Date,
            public twitter: string) {
        }
        sayHello(): string {
            return "Bonjour " + this.name + "!";
        }
    }
}
```

Code généré « person.js »

```
var app;
(function (app) {
    var domain;
    (function (domain) {
        var Person = (function () {
            function Person(id, name, birthday, twitter) {
                this.id = id;
                this.name = name;
                this.birthday = birthday;
                this.twitter = twitter;
            }
            Person.prototype.sayHello = function () {
                return "Bonjour " + this.name + "!";
            };
            return Person;
        })();
        domain.Person = Person;
    })(domain = app.domain || (app.domain = {}));
})(app || (app = {}));
```

Héritage

```
// interface IPerson
// classe Person
    class student extends Person {
        constructor(public id: number,
            public name: string,
            public birthday: Date,
            public twitter: string,
            public school :string) {

            super(id,name,birthday,twitter);
        }
        sayHello(): string {
            return "Bonjour!";
        }
    }
```

Appel au constructeur de la classe base. « super » permet également l'accès aux membres de la classe de

Redéfinition de la méthode de la classe de

f. Services

Exemple $resource

```
module app.common {
    export interface IPeopleService {
        getPeopleResource(): ng.resource.IResourceClass<IPeopleResource>;
    }
    export interface IPeopleResource extends ng.resource.IResource<app.domain.Person> {

    }

    export class PeopleService implements IPeopleService {
        static $inject = ["$resource"];
        constructor(private $resource: ng.resource.IResourceService) {

        }
        getPeopleResource(): ng.resource.IResourceClass<IPeopleResource> {
            return this.$resource("http://localhost:3000/api/people");
        }
    }
    angular
        .module("common.services")
        .service("peopleService", PeopleService);
}
```

Référencer le service dans la page d'index.

```
<!DOCTYPE html>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>TypeScript Demo</title>
       <meta charset="utf-8" />
</head>
<body ng-app="app">
    <div class="container">
        <ng-view></ng-view>
    </div>

    <!-- Lib -->
    <script src="lib/angular.js"></script>
    <script src="lib/angular-resource.js"></script>
    <script src="lib/angular-route.js"></script>


    <!-- app -->
    <script src="app/app.js"></script>

    <!-- Domain -->
    <script src="app/people/person.js"></script>

    <!-- Services -->
    <script src="app/common/services/common.services.js"></script>
    <script src="app/common/services/peopleService.js"></script>

    <!-- Controllers -->
    <script src="app/people/peopleController.js"></script>
</body>
</html>
```

Utilisation du service dans un contrôleur

```
module app {
    // interface
    interface IPeopleController {
        title: string;
        people: app.domain.IPerson[];
    }
    // classe
    class PeopleController implements IPeopleController {
        // propriétés
        title: string;
        people: app.domain.IPerson[];

        // constructeur
        static $inject = ["peopleService"];
        constructor(private peopleService: app.common.PeopleService) {
            this.title = "Contacts Twitter";
            this.people = [];
            var peopleResource = peopleService.getPeopleResource();
            peopleResource.query((data: app.domain.IPerson[]) => {
                this.people = data;
            });
        }
        // méthodes
    }
    // enregistrement
    angular
        .module("app")
        .controller("PeopleController", PeopleController);
}
```

Injection du service

Utilisation du service

Côté serveur (Node.js avec Express)

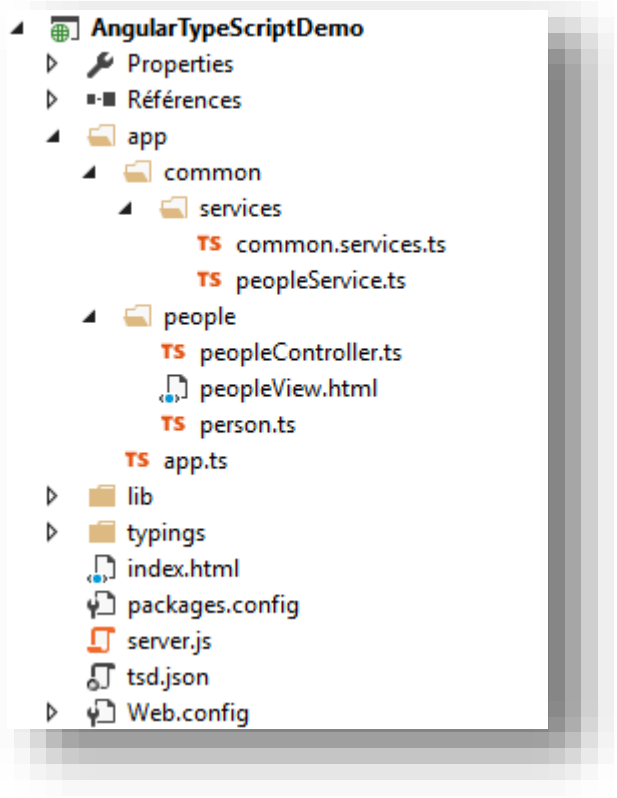
```
var express = require('express');
var app = express();

app.use(function (req, res) {
    res.setHeader('Access-Control-Allow-Origin', '*');
    res.setHeader('Access-Control-Allow-Methods', '*');
    res.setHeader('Access-Control-Allow-Headers', '*');
});
// GET ALL
app.get('/api/people', function (req, res) {

    var people = [
            {
                id: 1,
                name: "M. Bellin",
                twitter: "mariebellin123"
            },
            {
                id: 2,
                name: "J. Romagny",
                twitter: "romagny13"
            }];
    res.set("Content-Type", "application/json");
    res.send(200, people);

});
app.listen(3000);
console.log("Listen on port 3000");
```

Projet


AngularTypeScriptDemo
Properties
Références
app
common
services
common.services.ts
peopleService.ts
people
peopleController.ts
peopleView.html
person.ts
app.ts
lib
typings
index.html
packages.config
server.js
tsd.json
Web.config