

# **formation-typescript**

## **Travaux Pratiques**



**zenika**  
ARCHITECTURE INFORMATIQUE

# Pré-requis

---

## Installation

Installer les technos demandées

## TP 1: Outilage

---

Dans ce TP, nous allons apprendre à manipuler les différents outils pour développer une application en TypeScript.

Dans un premier temps, nous nous familiariserons avec Node.js et NPM.

Puis ensuite, nous allons pouvoir commencer à compiler notre code en JavaScript, valider la qualité de notre code et enfin automatiser ces tâches à l'aide de Gulp.

### Node.js et NPM

Premièrement, il faut installer Node.js (<https://nodejs.org/download/>).

Une fois installé, créez un nouveau répertoire et exécutez la commande `npm init` pour initier un nouveau projet. Répondez aux questions. Un fichier package.json est créé dans votre répertoire.

### Compiler en Javascript

Pour compiler du TypeScript nous avons besoin d'avoir le CLI TypeScript.

- Installez-le avec NPM :

```
npm install -g typescript
```

- Initier le fichier de configuration TypeScript

```
tsc --init
```

- Copiez ce code dans un fichier nommé `src/app.ts` :

```
function sayHello(nom: string) {
    return 'Bonjour, ' + nom;
}

let user = 'Zenika';

console.log(sayHello(user));
```

- Vous serez peut-être obligé d'installer le fichier de définitions pour NodeJS. Ne vous inquiétez pas, nous allons aborder ce sujet dans la suite de la formation.

```
npm install --save-dev @types/node
```

- Compilez le code en JavaScript avec `tsc src/app.ts` et exécutez le script via `node src/app.js`.

## Automatisation avec Gulp

- Premièrement, il faut installer gulp, toujours via NPM :

```
npm install -g gulp
```

- Créez un fichier `gulpfile.js`.
- Créez une première tâche pour compiler le TypeScript en JavaScript. Nous aurons pour cela besoin des modules `gulp-typescript` et `typescript`.
- Après avoir compilé le code nous allons améliorer les performances en le minifiant. Ajoutez donc à la tâche de build une étape de minification à l'aide du module `gulp-uglify`.
- Ajouter les fichiers modifiés dans un dossier `dist` : `gulp.dest('dist')`
- Définissez la tâche de build comme celle par défaut.

## Linter

Maintenant, nous allons essayer de valider le code avec tslint :

- Créez un fichier `tslint.json` avec quelques règles simples comme :
  - forcer l'utilisation de simple quotes
  - forcer les point-virgule en fin d'instruction
  - tabulations interdites, utilisation d'espaces uniquement
- Installer tslint, en utilisant `npm install --save-dev tslint gulp-tslint`.

- Créez une nouvelle tâche qui permet d'exécuter tslint sur le code, en utilisant le module `gulp-tslint` .
- Ajoutez cette nouvelle tâche comme préambule à la tâche de build.

## Map file

Pour voir l'intérêt des map file, créez un fichier index.html dans votre projet avec ceci :

```
<!doctype html>
<html lang="fr">
  <head> <meta charset="utf-8"> </head>
  <body>
    <script src="dist/app.js"></script>
  </body>
</html>
```

Ouvrez ce fichier avec un navigateur et regardez dans la console. Comme vous pouvez le voir, c'est écrit que "*Bonjour, Zenika*" est renvoyé par la ligne 1 du fichier `app.js` .

Ajoutez le module `gulp-sourcemaps` à vos dépendances et faites en sorte que lors du build un fichier sourcemaps soit créé.

Dorénavant vous pouvez voir que c'est la ligne 7 du code TypeScript qui affiche "*Bonjour, Zenika*".

### Pour aller plus loin :

Pour des raisons d'intégration et de déploiement continu notamment, il faut éviter d'installer des dépendances uniquement en global sur notre système.

Je vous propose donc d'installer Gulp en tant que dépendance de notre projet et de définir deux scripts dans le `package.json` .

Le premier effectuera le build de notre application et le second exécutera tslint sur notre code.

## TP 2 : Les Tests

---

Dans ce second TP, nous allons écrire nos premiers tests unitaires. Ce TP vient au tout début de la formation, afin de vous laisser la possibilité d'écrire de nouveaux tests pour les fonctionnalités que nous allons implémenter dans les TPs suivants.

Le test que nous allons implémenter permettra de s'assurer que les fonctionnalités définies dans la classe `HelloWorld` sont bien celles attendues. Dans cette classe, nous avons une méthode `sayHello` , qui retourne :

- lorsqu'elle est appelée sans paramètre, **Hello World!**.

- lorsqu'elle est appelée avec un paramètre, *par exemple Zenika, Hello Zenika!*.

Dans ce TP, nous allons nous assurer que la méthode **sayHello** retourne bien la chaîne de caractères désirée.

- L'implémentation de la classe que nous allons tester est la suivante. Recopiez cette classe dans le fichier `src/app.ts`

```
export class HelloWorld {
    sayHello(name: string = 'World') {
        return `Hello ${name}!`;
    }
}
```

Dans le code que vous venez de reprendre, plusieurs fonctionnalités du langage TypeScript sont utilisées (classe, paramètre par défaut, String Interpolation, fichiers de définition, ... ), mais seront seulement expliquées dans les prochaines parties de cette formation.

- Installez les modules **AVA** et **Sinon**

```
npm install --save-dev ava sinon @types/sinon
```

- Ajoutez la librairie **es2015** à la configuration du compilateur **TypeScript**

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es5",
    "noImplicitAny": false,
    "sourceMap": false,
    "lib": [
      "es2015"
    ]
  }
}
```

- Créez un répertoire **tests**, dans lequel nous allons créer tous nos tests unitaires.
- Dans ce répertoire, créez un fichier **app.ts**, en respectant la structure suivante :

```

import test from 'ava';
import { HelloWorld } from '../src/app'

let helloWorld: HelloWorld;

test.before(t => {
    helloWorld = new HelloWorld();
});

test(t => {

});

test(t => {

});

```

- Implémentez les deux tests afin de vérifier le bon fonctionnement de la méthode `sayHello`
- Dans le fichier **package.json**, ajoutez un script `test`, permettant de compiler vos fichiers TypeScript et exécuter vos tests AVA.## TP3 : Types et Inférence de types

Nous allons à présent commencer à développer en *TypeScript*. Nous allons tout d'abord nous habituer aux types primitifs proposés par le langage : `string`, `number`, `function`, `array` et `tuple`.

Une suite de tests unitaires est déjà disponible, permettant de tester le résultat de ce TP : `__tests__/tp3.ts`.

Vous pouvez le lancer à tout moment en utilisant cette commande (définie dans le TP précédent):

```
npm test
```

Pour information, devant toutes les objets que vous allez définir à présent, veuillez ajouter le mot clé `export`, afin de pouvoir exécuter les tests. Ce mot clé sera abordé dans une prochaine partie théorique.

- Dans un fichier `src/tp3.ts`, créez une fonction `returnPeopleAndLength`
  - qui prend en paramètre un tableau de `string`
  - qui retourne un tableau de tuples `[string, number]`
  - le tuple contient un élément du tableau passé en paramètre et la taille (en nombre de caractères) de cet élément

- Le paramètre de la méthode `returnPeopleAndLength` possède une valeur par défaut égale à : `['Miles', 'Mick']`
- Exécutez les tests unitaires associés, à présent seulement trois tests doivent échouer.
- Créez une méthode `displayPeopleAndLength`
  - qui prend en paramètre un tableau de `string` optionnel
  - ne retourne aucune donnée
  - exécute la méthode `console.log` pour chaque élément retourné par la méthode `returnPeopleAndLength`
  - la chaîne de caractères affichée doit avoir cette structure : 'Miles contient 5 caractères'
  - utilisez le système d'*interpolation*
- Exécutez les tests unitaires associés, seul un test doit encore échouer (le code sera implémenté au point 8)
- Dans la méthode `displayPeopleAndLength`, utilisez le système de *destructuring* pour manipuler le tableau de tuples.
- Exécutez de nouveau les tests unitaires pour vérifier qu'aucune régression n'a été ajoutée.
- Nous allons à présent utiliser les types `enum`
  - Définir un type `NumberToString` avec les valeurs suivantes : un, deux, trois, quatre, cinq, six, sept, huit, neuf
  - Dans la méthode `displayPeopleAndLength`, ajoutez un paramètre optionnel de type `boolean` appelé `literal`
  - Si ce paramètre est égal à `false` ou `undefined`, le code implémenté précédemment sera exécuté
  - Si ce paramètre est égal à `true`, nous allons
    - filtrer le tableau passé en paramètre pour n'afficher que les chaînes de caractères dont la taille est inférieure ou égale à 9
    - la chaîne de caractères affichée aura la structure suivante : 'Miles contient cinq caractères'
- Exécutez de nouveau les tests unitaires. Ils doivent tous s'exécuter avec succès.

## TP 4 : Classes, Interfaces, Héritage et Générique

---

Nous allons, à travers ce TP, manipuler tous les concepts de classes, interfaces, héritage et

génériques proposés par TypeScript. Des tests unitaires sont déjà disponibles, permettant de tester les différentes étapes de ce TP : `__test__/tp4.ts` .

Vous pouvez les lancer à tout moment en utilisant la commande définie dans le TP précédent.

Toutes les entités seront à créer dans un nouveau fichier `src/tp4.ts`, le découpage en module sera expliqué dans le prochain chapitre.

- Définissez une `enum Music` contenant les clés `JAZZ` et `ROCK`
- Définissez une classe `Musician` ayant les propriétés suivantes :
  - `firstName`
  - `lastName`
  - `age`
  - `style` (de type `Music` non défini)
- Implémentez, dans la classe `Musician` une méthode `toString` qui doit, grâce au système *d'interpolation* retourner une chaîne de la forme `firstName lastName`.
- Définissez deux nouvelles classes `JazzMusician` et `RockStar` qui doivent hériter de la classe `Musician`.

Ces classes permettront de définir la propriété `style` en utilisant les valeurs `Music.JAZZ` et `Music.ROCK`

- Modifiez la méthode `toString` pour qu'elle retourne une chaîne de caractère de la forme `firstName lastName plays style`.
- Modifiez le scope de la propriété `style`. Elle doit être définie avec le scope `private`. Implémentez les accesseurs pour cette propriété.
- Créez une nouvelle classe `Album`. Cette classe aura une propriété `title` de type `string` et une méthode `toString` qui retournera `title`.
- Ajoutez à la classe `Musician` une propriété privée de type `Album[ ]`.
- Nous allons également définir une méthode `addAlbum(album: Album)`. Cette méthode sera définie dans une interface `MusicianI` qui sera implémentée par la classe `Musician`.
- Pour terminer ce TP, nous allons implémenter une méthode générique `display` (à l'extérieur de la classe `Musician`) :
  - Cette méthode prendra un tableau d'objets en paramètre.
  - Pour chaque élément du tableau, nous afficherons sur la console le retour de la

méthode `toString` pour l'objet courant.

- Tous les tests unitaires doivent passer.

## TP 5 - Les Modules

---

Nous allons à présent découper la solution du TP précédent en différents modules, afin d'avoir du code simple et réutilisable.

- Copiez le contenu du fichier `src/tp4.ts` dans un nouveau fichier `src/tp5.ts`
- Copiez le contenu du fichier `__test__/tp4.ts` dans un nouveau fichier `__test__/tp5.ts`
- Créez un nouveau fichier `src/UtilModule.ts` dans lequel vous allez créer et exporter un module `DisplayModule`
  - Ce module exporte une méthode `log` ayant la signature suivante :  
`<T>(value: T): void;`
  - Dans cette méthode, vous devez appeler la méthode `log` de l'objet `console`
- Créez un nouveau fichier `src/Display.ts` qui exposera une seule fonction (utilisation du mot clé `default`)
  - Cette méthode correspond à la méthode `display` implémentée dans le TP précédent
  - Au lieu d'appeler directement la méthode `console.log`, vous devez faire appel à la méthode `log` du module `DisplayModule`
- Exportez tous les objets créés dans le TP précédent (fichier `Musician.ts`) : `Musician`, `Music`, `Album`, `RockStar` et `JazzMusician`
- Ajoutez une méthode `swing` à la classe `JazzMusician` et une méthode `shout` à la classe `RockStar`. Ces deux méthodes affichent respectivement `I'm swinging!` et `I'm shouting!` (utiliser la méthode `log` de `DisplayModule`)
- Modifiez le test, afin d'importer les bons modules. Tous les tests unitaires doivent passer.
- Enfin, dans le fichier `src/tp5.ts` :
  - Utilisez la méthode `log` de `DisplayModule` pour afficher un message d'accueil : **Bienvenue dans ma première application TypeScript**
  - Créez une liste de 2 musiciens (un `JazzMusician` et un `RockStar`)
  - Ajoutez 2 albums au `JazzMusician`

- Affichez la liste des musiciens et la liste des albums ( `src/Display.ts` )
- Bouclez sur la liste des musiciens et affichez `I'm swinging!` / `I'm shouting!` selon leur type
- Compilez votre code et exécutez-le dans un environnement node : vous devez voir le message d'accueil, les deux listes et ce que les musiciens ont à dire

## TP 6 : Fichiers de définitions

---

Nous allons dans ce TP, intégrer la librairie *lodash* afin de bénéficier de la méthode `each` pour itérer sur une collection.

Nous allons utiliser cette méthode à la place des boucles `for` / `forEach`

- Installez via *npm* la librairie *lodash*
- Importez le module téléchargé dans le fichier `src/tp5.ts` de votre application.

```
import * as _ from 'lodash';
```

- Le système d'autocomplétion de votre IDE ne doit pas fonctionner pour cette librairie. Ce problème est normal, car le compilateur ne connaît pas la structure de la librairie `lodash`.
- Dans le répertoire *typings*, créez un nouveau fichier *lodash.d.ts* et importez-le dans le fichier `src/tp.ts`.
- Pour déclarer un composant non défini dans l'environnement TypeScript, nous devons le déclarer via le mot clé `declare`. Dans le fichier de définitions, déclarez un module "lodash"
- Dans ce module, exportez une fonction respectant la signature de la méthode `each` de `lodash`

```
_.each(list, function(objet) { ... });
```

- Remplacez la boucle `for` / `forEach` utilisée dans le fichier `src/tp5.ts`, par cette toute nouvelle méthode. Observez l'autocomplétion qui permet d'accéder aux informations que nous venons de définir. Faites de même pour la boucle contenue dans le fichier `src/Display.ts`
- Nous allons à présent faire la même chose, mais en utilisant le fichier de définitions disponible :

```
npm install --save-dev @types/lodash
```

# TP 7 : Les Décorateurs

---

Dans ce TP, nous allons créer plusieurs décorateurs permettant d'améliorer le code que nous venons d'implémenter dans les TPs précédents :

- Un décorateur `@log` que nous allons utiliser sur une méthode. Elle permettra d'utiliser la méthode `log` du module `DisplayModule` avec en paramètre la valeur de retour
- `@StyleMusic`, un décorateur paramétrable qui, utilisé sur une classe, permettra de définir la propriété `style`
- Dans la configuration du compilateur, la propriété `experimentalDecorators` doit être à `true`.
- Créez un décorateur `@log`, qui sera une annotation qui devra être utilisée sur une méthode
  - Cette méthode permettra d'appeler la méthode `log` du module `DisplayModule` en utilisant le paramètre de la méthode sur laquelle nous avons utilisé le décorateur.
- Modifiez les implémentations des méthodes `swing` et `shout` des classes `JazzMusician` et `RockStar`. Elles doivent à présent retourner la chaîne de caractères précédemment passée en paramètre de la méthode `log`.
- Ajoutez le décorateur `@log` sur ces méthodes.
- Créez un décorateur `@JazzMan` qui sera utilisé sur une classe.
  - Ce décorateur surchargera le constructeur décoré, en définissant la propriété `style`.
- Supprimez l'initialisation de la propriété `style` du constructeur de la classe `JazzMusician`
- Ajoutez le décorateur `@JazzMan` à la classe `JazzMusician`
- Faites de même avec un nouveau décorateur `@Rocker`, et la classe `RockStar`
- Vérifiez que les tests unitaires s'exécutent toujours avec succès.
- Supprimez les décorateurs `@JazzMan` et `@Rocker`. Nous allons les remplacer par un décorateur générique `@StyleMusic`
  - Ajoutez un paramètre correspondant au type de musique