TP TypeScript

Nous verrons au cours des travaux pratiques les points suivants :

- TP 1 – Mise en place de l'environnement,
- TP 2 - Création d'un composant,
- TP 3 - Création d'un service et appel d'un webservice,
- TP 4 - Création d'un décorateur,
- TP 5 - Rechercher une annonce,
- TP 6 - Visualiser une annonce.

Objectifs

Les TP ont pour objectif de vous apprendre à développer avec le langage TypeScript tout en utilisant des librairies JavaScript préexistante telles que jQuery, Materialize-css, fetch ou lodash.

Pour ce faire nous allons créer une petite application web de consultation d'annonce de vente immobilière.

Nous verrons par ailleurs les notions suivantes :

- Utiliser les arrows functions,
- Utiliser les mots clefs let et const,
- Créer une classe TypeScript,
- Créer une interface pour typer un objet retourné par un WebService,
- Gérer des templates,
- Utiliser l'héritage pour concevoir des composants,
- Utiliser l'aggregation de paramètre,
- Utiliser les collections ES6,
- Utiliser la librairie fetch pour faire un appel Ajax,
- Créer un décorateur TypeScript pour faciliter la création de composant.

TP 1 - Installation de l'environnement

Prérequis

Vérifier que vous avez les éléments suivants d'installé sur votre poste :

- Node v6 ou plus avec la commande npm –v,
- Git, nous l'utiliserons pour récupérer le projet initial,
- Webstorm (ou un autre IDE)

Installation

Initialisation du projet

Créez un nouveau projet dans votre IDE. Puis dans le terminal, placez-vous sur votre nouveau projet et lancez la commande suivante :

```
npm init
```

> Suivez le guide d'initialisation de la commande. Elle va créer le fichier `package.json` avec les informations du projet.

Installation des modules

Maintenant nous allons installer les modules nécessaires à un projet TypeScript.

Toujours dans le terminal lancez la commande suivante :

```
npm install -g typescript@2.0
npm install --save @types/jquery materialize-css
```

```
npm install --save-dev connect-history-api-fallback http-proxy-middleware lite
tsc --init
```

Nous travaillerons de préférence avec la version 2.0 de TypeScript

La commande `tsc --init` va créer un nouveau fichier `tsconfig.json`. Ce fichier contient les informations nécessaire au compilateur TypeScript pour compiler nos fichiers sources.

En l'état, il nous manque quelques options de compilation dans le `tsconfig.json`.

Voici les options à reporter dans votre `tsconfig.json`:

```
{
  "compilerOptions": {
    "module": "commonjs",
    "target": "es6",
    "noImplicitAny": false,
    "lib": [
      "es6",
      "dom",
      "es2015.collection"
    ],
    "types": [
      "jquery"
    ],
    "experimentalDecorators":true,
    "emitDecoratorMetadata": true,
    "sourceMap": true,
    "declaration": false
  },
  "exclude": ["node_modules"]
}
```

Votre projet est prêt pour compiler du TypeScript

Création du serveur de développement

Nous allons maintenant mettre en place le serveur de développement.

A la racine de projet, créer le fichier bs-config.js et collez le script suivant :

```
var proxyMiddleware = require('http-proxy-middleware');
var fallbackMiddleware = require('connect-history-api-fallback');

module.exports = {
    "port": 8000,
    "files": [
        "./webapp/**/*.{html,htm,css,js}",
        "./node_modules/**/*.{html,htm,css,js}"
    ],
    "server": {
        "baseDir": "webapp",
        "routes": {
            "/node_modules": "node_modules"
        },
```

```
        middleware: {
            1: proxyMiddleware('/api', {
                target: 'http://futurlogement5.azurewebsites.net',
                changeOrigin: true   // for vhosted sites, changes host header
            }),

            2: fallbackMiddleware({
                index: '/index.html',
                verbose: true
            })
        }
    }
};
```

Ensuite il vous faut éditer le `package.json` et ajouter ceci :

```
{
  "scripts": {
     "serve": "lite-server"
  }
}
```

> `scripts` permet d'ajouter de nouvelle commande NPM qui peuvent être appelées en ligne de commande avec `npm run [nom cmd]`.

Notre serveur de développement est prêt. Il nous reste plus qu'à créer un dossier `webapp` avec une première page `index.html`.

Création de l'application web

Voici à quoi la page d'index doit ressembler :

```
<!DOCTYPE html>
<html lang="en">
<head>
    <title>TP</title>
    <!--Import Google Icon Font-->
    <link href="http://fonts.googleapis.com/icon?family=Material+Icons" rel="s
    <!--Import materialize.css-->
    <link type="text/css" rel="stylesheet" href="node_modules/materialize-css/

    <!--Let browser know website is optimized for mobile-->
    <meta name="viewport" content="width=device-width, initial-scale=1.0"/>
</head>
<body>

    <app>

    </app>

    <!-- vendor -->
    <script type="text/javascript" src="node_modules/materialize-css/bin/jquer
    <script type="text/javascript" src="node_modules/materialize-css/bin/mater
    <!-- end vendor -->
```

```
    <script type="text/javascript" src="scripts/app.js"></script>
</body>
</html>
```

Ici il nous manque le script `app.js`. Notez que dans la page d'index nous incluerons toujours le fichier javascript généré par TypeScript et non le fichier `app.ts`.

Dans le dossier `scripts`, créer un nouveau fichier `app.ts`.

L'exercice sera d'intercepter la balise `app` et d'y afficher un `HELLO WORLD` avec jQuery.

> Note : pour lancer votre server `npm run serve`.
>
> Note 2 : pour compiler votre fichier TypeScript `tsc`.
>
> Correction du TP : tp1-solution

TP 2 - Création d'un composant

Notre exercice sera de créer un premier composant `card` basé sur le framework CSS materialize-css (documentation).

Ce composant `card` nous permettre d'afficher un résumé d'une annonce. Nous le réutiliserons pour afficher une liste d'annonce très visuel !

Voici le template materialize-css d'une `card`:

```
<div class="card">
    <div class="card-image">
        <img src="images/path.png">
        <span class="card-title">Titre</span>
    </div>
    <div class="card-stacked">
        <div class="card-content">
            Contenu
        </div>
        <div class="card-action">
            <a href="#">Voir annonce</a>
         </div>
    </div>
</div>
```

> Nous allons construire ce composant ensemble !
>
> Correction du TP : tp2-solution

TP 3 - Création d'un service

Bien que nous n'utilisions pas de framework avancé comme Angular, View.js ou React.js, il est toujours pratique de mettre place des patterns de conception. Cela permet de structurer rapidement notre code et il est ainsi plus facile à porter vers un autre framework.

Nous allons donc mettre en place un service qui aura pour objectif d'appeler le webservice `api/serviceannoncesimmobilieres` pour récupérer la liste des annonces.

Nous verrons comment utiliser la nouvelle API Fetch pour faire une appel Ajax et commment typer une reponse Ajax pour profiter de l'analyse statique du code proposé par TypeScript et les IDE tels que Webstorm.

Créer le service

Notre service sera nommé `RealEstateAdService` par convention et aura une methode `getAll()` qui retournera une Promise avec pour reponse la liste des annonces.

Voici le contrat de service

- Methode : GET
- Url : `api/serviceannoncesimmobilieres`
- Réponse : Liste d'annonce

Structure d'une annonce

- Id : number,
- Titre : string,
- Ville : string,
- Latitude : number,
- Longitude : number,
- Achat : boolean,
- Location : boolean,
- Appartement : boolean,
- Maison : boolean,
- Chateau : boolean,
- Prix : number,
- NbrPieces : number,
- Surface : number,
- Etage : number,
- Ascenceur : boolean,
- Terrasse : boolean,
- Parking : boolean,
- Descriptif : string,
- ListImagesUrl : tableau de string,
- UrlImagePrincipale : tableau de string.

Installation de fetch

```
npm install --save whatwg-fetch @types/whatwg-fetch
```

> N'oubliez pas de rajouter whatwg-fetch à la liste des `types` dans votre `tsconfig.json` !

Exercice 1

A partir de la structure d'annonce créer une interface TypeScript correspondant à sa structure.

Exercice 2

Créer la classe `RealEstateAdService` avec une methode `getAll()`. La methode getAll() doit utiliser l'API `fetch` pour appeler le webservice (documentation).

Exercice 3

Typer le retour de la methode `getAll()` de façon à ce qu'elle expose une Promise de liste d'annonce.

Exercice 4

Utiliser la méthode `getAll()` dans l'application pour afficher la liste d'annonce avec notre composant `card`.

> Correction du TP : tp3-solution

TP 4 - Les décorateurs

L'idée est de déléguer la création de composant à un factory que l'on nommera `ComponentsFactory`.

Cette factory sera ni plus ni moins qu'un registre de composant collecté par le décorateur `@component`.

Voici un exemple de ce qui sera attendu :

```
@component('card')
export class CardComponent extends Component {


    render() {
        return "Hello card";
    }
}

@component('app', Card)
export class AppComponent extends Component {

    render() {
        return `
            <card></card>
        `;
    }

}

ComponentsFactory.bootstrap(AppComponent);
```

> Cette exemple va intercepter tous les éléments `card` et créer une nouvelle instance de `CardComponent`.

Création du ComponentsFactory

Dans le dossier `services` nous allons créer un nouveau fichier `ComponentsFactory.ts`.

Voici une partie de l'implémentation de la factory :

```
module app.services {

    export class ComponentsFactory {
        private static components = new Map<string | any, any | string>();

        /**
         * Add a new component in registry.
         */
        static add(selector: string, target: any, childrens: any[] = []) {

            target.$childrens = childrens;

            this.components.set(selector, target);
            this.components.set(target, selector);
        }

    }
}
```

Création du décorateur

Maintenant nous allons développer le décorateur de classe pour collecter les classes auxquelles il sera associé.

À partir de l'exemple donné dans le cours, créez le décorateur ! Ce décorateur aura deux paramètres :

- selector : le selecteur css associé à la classe composant,
- dependencies : Aggregateur contenant la liste des composants utilisés par la classe en court.

Modification de l'application

Une fois votre décorateur réalisé, nous allons l'utiliser sur la classe `CardComponent` et `AppComponent`. Ajoutez donc votre décorateur `@component` comme présenté dans le premier exemple et vérifiez que le décorateur fonctionne.

> Si vous souhaitez tester votre décorateur, vous pouvez ajouter un console.log() pour voir si votre composant est bien collecté.

A se stade, nous n'avons implémenté qu'une partie de la logique. Le décorateur fait ce qu'on lui demande, mais il fait développer un peu notre `ComponentsFactory` pour qu'il gère complétement le cycle de vie de nos composants.

Nous allons commencer par simplifier nos composants et déléguer un maximum de traitement vers la classe `Component` et `ComponentsFactory`.

Utilisation de classe abstraite

L'utilisation de classe de abstraite permet de fournir une classe de base contenant un certain nombre de fonctionnalité commune à un type de classe, ici les composants.

Voici les possibilités d'une classe abstraite :

- Ne peut être instancié directement. Elle doit donc être étendu (héritage).
- Elle contient des attributs et méthodes avec implémentation.
- Elle peut définir des méthodes abstraites. Ainsi une méthode abstraite doit obligatoirement être implémenté dans la classe fille.

Nous allons donc implémenter cette classe ensemble ! Elle aura pour objectifs :

- De construire l'élément jQuery,
- Exposer des accesseurs à l'élément jQuery et aux attributs de l'élément,
- Stocker le contenu d'origine de la balise,
- De définir une méthode abstraite `render()`.

Maintenant que notre classe abstraite est défini nous pourrons l'utiliser comme dans l'exemple suivant :

```
@component('card')
export class CardComponent extends Component {

    render() {
        return "Hello card";
    }
}
```

Finalisation de ComponentsFactory

Nous allons maintenant agrémenter la classe ComponentsFactory afin de gérer le cycle de vie suivant :

- Initialisation du premier composant,
- Rendu du composant et des composants utilisés par le composant,
- Rappel après création du rendu du composant en cours.

Initialisation du premier composant

Pour gérer l'initialisation du premier composant nous allons rajouter ceci :

```
module app.services {

    export class ComponentsFactory {

        static boostrap(cmpClazz: any) {

            jQuery(document).ready(() => {
                this.render(cmpClazz);
            });

        }

    }
}
```

Implémentation de la méthode render()

Cette méthode prend la classe d'un composant (`Card` par exemple) et recherche dans le DOM tous les élements correspondant au selecteur que nous avons collecté avec le décorateur `@component()`.

```
module app.services {

    export class ComponentsFactory {

        static render(cmpClazz: any) {

            const selector: string = this.components.get(cmpClazz) as string;

            this.fromSelector(selector, cmpClazz)
                .forEach(cmp => cmp.render());

        }
    }
}
```

Implémentation de la méthode fromSelector()

```
module app.services {

    export class ComponentsFactory {
        static fromSelector(selector: string, componentClazz: any) {

            const elements = jQuery(selector);
            const cmps = [];

            elements.each((index, element: Element) => {

                const cmp = new componentClazz(element);

                cmps.push(cmp);

                if (cmp.onInit) {
```

```
                    cmp.onInit();
                }

                cmp.$render = cmp.render;

                cmp.render = function() {
                    this.element.html(this.$render());

                    componentClazz.$childrens.forEach(childrenClazz => {

                        ComponentsFactory.render(childrenClazz);

                    });

                    if (this.afterRender) {
                        this.afterRender();
                    }
                };
            });

            return cmps;
        }
    }
}
```

Maintenant notre `ComponentsFactory` est prête il nous reste plus qu'à adapter notre application !

> Correction du TP : tp4-solution

TP5 - Rechercher une annonce

Nous allons maintenant créer une composant permettant de rechercher une annonce. Cette recherche relancera une nouvelle requête avec les critères de recherche saisie.

L'objectif est d'améliorer la méthode `RealEstateAdService.getAll()` afin de proposer des options de recherche.

Rappel : Le contrat de service

- Methode : GET
- Url : `api/serviceannoncesimmobilieres`
- Réponse : Liste d'annonce

Structure d'une annonce

- Id : number,
- Titre : string,
- Ville : string,
- Latitude : number,
- Longitude : number,
- Achat : boolean,
- Location : boolean,
- Appartement : boolean,
- Maison : boolean,
- Chateau : boolean,
- Prix : number,
- NbrPieces : number,
- Surface : number,
- Etage : number,

- Ascenceur : boolean,
- Terrasse : boolean,
- Parking : boolean,
- Descriptif : string,
- ListImagesUrl : tableau de string,
- UrlImagePrincipale : tableau de string.

Les paramètres de recherche possibles

- Ville : Nom de la ville de recherche (le service veut un nom exacte),
- Achat : S'agit-il d'un achat (True ou False),
- Location : S'agit-il d'une location (True ou False),
- Appartement : S'agit-il d'un appartement (True ou False),
- Maison : S'agit-il d'une maison (True ou False),
- Chateau : S'agit-il d'un chateau (True ou False),
- PrixMin : Prix minimum du bien recherché,
- PrixMax : Prix maximal du bien recherché,
- SurfaceMin : Surface minimal du bien recherché,
- SurfaceMax : Surface maximal du bien recherché.

  Le service s'attend à avoir toutes les options de recherche !

Exemple d'url : `api/serviceannoncesimmobilieres?Ville=paris&Achat=True&Location=False&Appartement=True&Maison=False&Chateau=Fa`

Exercice 1

Créer l'interface `IAdRequest` correspondant aux paramètres.

Exercice 2

Créer un composant permettant de rechercher les annonces par ville.

> Correction du TP : tp5-solution

TP6 - Visualisation d'une annonce

L'objectif est de visualiser une annonce dans une nouvelle page.

Exercice 1

Créez la nouvelle page `edit.html` dans lequel vous appelerez le composant `<app type="edit">` avec l'option `edit` par exemple.

> Vous pouvez faire autrement si vous le souhaitez.

Exercice 2

Il vous faudra récupérer l'id de l'annonce cliqué dans la page précédente et afficher les informations de l'annonce.

Vous avez un service web qui permet de récupérer une annonce !

Le contrat de service

- Methode : GET
- Url : `api/serviceannoncesimmobilieres/{id}`
- Réponse : Une annonce

Vous devrez donc créer une nouvelle méthode dans `RealEstateAdService` pour récupérer une instance annonce.

A vous de jouer !

Exercice 3

Vous devez afficher les photos de l'annonce dans un carousel fournit par Materialize-css. Ensuite vous devez afficher le détail de l'annonce.

Correction du TP : tp6-solution