formation-angular2

Travaux Pratiques

zenika
ARCHITECTURE INFORMATIQUE

Pré-requis

Installation

Installer les technos demandées :

- NodeJS / NPM
- GIT

TP 1 : Démarrer une application Angular

Dans ce premier TP, nous allons initier notre première application **Angular**, qui sera réutilisée dans les TPs suivant.

L'initialisation de cette application se décomposera en plusieurs étapes :

- Création d'un projet Angular avec `@angular/cli`
- Implémentation de la page principale
- Création du composant principal
- Lancement du serveur afin de tester

Création du projet

L'application, que nous allons implémenter, sera initialisée via l'outil `@angular/cli`. Cet outil va automatiser :

- la création et la configuration du squelette de l'application
- la gestion des dépendances
- Téléchargez `@angular/cli` en utilisant `NPM`. Ce module nécessite une version récente de *NodeJS*
- Depuis votre console, créez un nouveau projet via la commande `ng new Application --style=scss`
- Regardez la structure de l'application tout juste créée
  - dépendances installées
  - configuration TypeScript
  - les différents fichiers TypeScript
- Une fois cette étape terminée, vous pouvez à présent lancer votre application en exécutant la commande `npm start`. Cette commande va prendre en charge la compilation de vos sources et le lancement d'un serveur.

Modification de l'application

Même si nous n'avons pas encore abordé les concepts du framework, nous allons faire des petites modifications afin de prendre en main la structure de notre application.

- Le composant principal devra contenir le code HTML suivant :

```
<h1>Welcome to {{ title }}!</h1>
```

- La variable `{{ title }}` sera remplacé par le contenu de la propriété `title` dans la classe `Application`. Modifier la valeur de cette propriété pour y mettre votre prénom.

- Vérifiez que vous obtenez bien la toute dernière version de votre application dans le navigateur avec le titre `Welcome to VotrePrénom`.

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.

Lors de la mise en place initiale de l'application dans le TP précédent, toute la configuration nécessaire à la bonne exécution des tests unitaires a été automatiquement réalisée.

Nous allons vérifier que tout fonctionne correctement.

- Lancez les tests via `@angular/cli` et `karma`

```
ng test
```

Après les modifications réalisées au TP1, les premiers tests générés par `@angular/cli` vont échouer.

Vous pouvez jeter un coup d'oeil aux tests générés par `@angular/cli`. Ils seront expliqués plus en détails dans les parties suivantes.

- Corriger les tests générés en remplaçant la valeur du `title` dans les assertions par la valeur que vous avez utilisé.

TP3 : Composants

L'application que nous allons développer tout au long de cette formation, est une application d'e-commerce.

Après avoir reçu du formateur le template principal de l'application, veuillez suivre les étapes suivantes :

- Modifiez le fichier index.html créé dans les TPs précédent, afin d'intégrer le template envoyé par le formateur.

- Tout le code HTML situé entre les balises `body` doit être défini dans le template du composant `AppComponent`

- Le total de votre panier sera défini dans un attribut `total` à rajouter dans `AppComponent` que nous allons initialiser à 0
- Créez un nouveau composant `menu\menu.component.ts` dans lequel vous allez implémenter le menu principal de l'application. Pour créer un nouveau composant Angular, nous allons utiliser la commande `ng generate component menu`
- Remplacez dans le composant `AppComponent` le menu initial par le composant que vous venez de créer.
- Créez une classe `product.ts` dans un répertoire `model`. Pour créer cette nouvelle classe, vous pouvez utiliser la commande `ng generate class model/product`.
- Dans cette classe, définissez les propriétés suivantes:
  - title de type `string`
  - description de type `string`
  - photo de type `string`
  - price de type `number`
- Dans le composant `AppComponent`, instancier un nouveau tableau de `Product` et ajoutez les produits utilisés dans le template.
- Modifier le template pour utiliser ce tableau pour l'affichage des différents produits. Comme nous n'avons pas encore vu la directive `ngFor`, vous êtes obligé de copier/coller le template pour chaque élément du tableau.
- Nous allons à présent externaliser le template utilisé pour afficher un produit dans un nouveau composant `ProductComponent`. Ce composant aura un paramètre `data` correspondant à un objet de type `Product`. Ajoutez ce composant dans le template.
- Nous allons à présent émettre un évènement `addToBasket`, via le composant `ProductComponent`, lorsque l'utilisateur cliquera sur le bouton `Ajoutez au panier`. Cet évènement sera utilisé par le composant `Application` pour mettre à jour la variable `total` créée précédemment.

Tests

- Ajouter `schemas: [CUSTOM_ELEMENTS_SCHEMA]` dans le `configureTestingModule` du composant `App` pour qu'il n'échoue pas sur l'utilisation des composants `app-menu` et `app-product`.
- Remplacer le test de la valeur de `title` par un test de la valeur de `total`
- Remplacer le test du binding de `title` par un test du binding de `total` dans le `header`
- Ajouter un test de la méthode `updatePrice`. L'appeler avec un produit créé pour l'occasion et vérifier que le total a été mis à jour.

- Ajouter un test du binding des produits dans les composants associés. Sélectionner les composants `app-product` et vérifier leur propriété `data`.
- Ajouter un test au composant `app-product` pour le binding des champs `title` et `price`.
- Ajouter un test au composant `app-product` pour le binding de la propriété `src` de l'image.
- Ajouter un test au composant `app-product` pour l'utilisation du bouton. Utiliser un `spy` sur la méthode `emit` de `addToBasket` pour intercepter et valider qu'elle a été appelé.
- Ajouter un test au composant `app-menu` pour valider que le template fonctionne. Tester qu'un texte du template est bien présent, par exemple `Zenika` dans `.navbar-brand`.

TP4 : Les directives Angular

Dans ce TP, nous allons utiliser les directives `ngFor`, `ngIf` et `ngClass` pour dynamiser notre page. Les autres directives d'Angular seront présentées lors du chapitre sur les formulaires.

- Grâce à la directive `ngFor`, itérez sur la liste des `products` afin d'afficher autant de composants `ProductComponent` qu'il y a d'éléments dans ce tableau.
- Dans la classe `Product`, ajoutez une propriété `stock` de type `number`.
- Initiez cette propriété pour tous les produits définis dans le composant `AppComponent`. Nous vous conseillons de mettre une valeur différente pour chaque produit, afin de pouvoir tester les différents cas définis ci-dessous.
- Modifier la méthode `updatePrice` du composant `AppComponent` pour réduire le stock du produit dès que l'on clique sur `Ajouter au panier`.
- Grâce à la directive `ngIf`, affichez un produit, seulement si sa propriété `stock` est supérieure à 0. Vous serez peut-être obligé de revoir l'utilisation du `*ngFor` du point précédent.
- Grâce à la directive `ngClass`, ajoutez une classe CSS `last`, sur l'élément utilisant la classe `thumbnail`, si la propriété `stock` d'un produit atteint 1. Cette classe ne sera utilisée que pour modifier la couleur de fond (`background-color: rgba(255, 0, 0, 0.4)`)

Tests

- Corriger les tests existant en prenant en compte le changement de signature de la classe `Product` (ajout du champ stock). On constatera que le test du binding des produits fonctionne toujours alors que l'implémentation a changé (utilisation du `ngFor`).
- Compléter le test de la méthode `updatePrice` pour vérifier que le stock du produit a bien été diminué.
- Ajouter un test dans `app` vérifiant qu'un produit sans stock n'est pas affiché. Pour ce faire, modifier le tableau `products` avec un nouveau tableau contenant deux produits, un au stock vide, l'autre non. Après avoir lancé `fixture.detectChanges()`, vérifier qu'il n'y a qu'une balise `app-product` et que sa propriété `data` est bien égale au produit ayant du stock.

- Ajouter deux tests dans `app-product`, un vérifiant que la class `last` n'est pas ajoutée si le stock est supérieur à 1, l'autre vérifiant qu'elle l'est si le stock est égal à 1.

TP5 : Injection de Dépendances

Nous allons à présent aborder les services et l'injection de dépendances dans une application Angular.

Nous allons créer deux services :

- ProductService : qui sera en charge de la gestion des produits,
- CustomerService : qui sera en charge du panier de l'utilisateur.
- Veuillez créer un service `ProductService` en utilisant la commande `ng generate service services/Product` dans lequel vous allez définir :
  - un tableau `products` avec les valeurs définies dans le composant `AppComponent.ts`
  - une méthode `getProducts()` : retournera le tableau `products`
  - une méthode `isTheLast(product)` : retournera `true` si le stock d'un produit est égal à 1
  - une méthode `isAvailable(product)` : retournera `true` si le stock d'un produit n'est pas égal à 0
  - une méthode `decreaseStock(product)` : mettra à jour la propriété `stock` du produit spécifié en paramètre
- Veuillez créer un service `CustomerService`, en utilisant la commande `ng generate service services/Customer` dans lequel vous allez définir :
  - une méthode `addProduct(product)` : ajoutera le nouveau produit dans un tableau, ce tableau représente votre panier.
  - une méthode `getTotal()` : calculera le montant total du panier.
- Importez ces deux services dans votre composant `Application`, et modifiez l'implémentation de ce composant afin d'utiliser les différentes méthodes implémentées précédemment.
- Pour terminer ce TP, nous allons externaliser le titre "Bienvenue sur Zenika Ecommerce" dans une variable injectable de type `String` en utilisant un provider de type `Value`

Tests

Avec l'ajout de dépendances à vos composant, les tests réalisés jusque là vont presque tous échouer. En effet, pour utiliser vos composant, il faudra maintenant qu'Angular sache comment résoudre les dépendances de chaque composant.

Souvenez vous que le but de chaque test unitaire est de tester le code de l'élément qu'on est entrain de tester (composant ou service) sans jamais utiliser du code d'un autre élément. Il ne faut donc pas satisfaire les dépendances avec les vrais implémentations mais avec des mocks.

De plus, l'ajout de service a déplacé certaines responsabilités. Certains tests réalisé jusque là dans les composant ne doivent pas être corrigé mais supprimés.

- Dans les tests de `app`, créer une classe `ProductServiceMock` minimaliste qui remplacera `ProductService` ainsi qu'une classe `CustomerService`. Ajouter une propriété `providers` dans le module de test avec `ProductService` et `CustomerService` définie avec leur mock ainsi qu'une valeur pour `welcomeMsg`.
- Dans les tests de `app`, supprimer les tests portant sur le calcul et la mise à jour du total du panier. Le composant n'a plus la responsabilité de ce calcul.
- Dans les tests de `app`, utiliser la fonction `inject` d'Angular pour obtenir des instances des services et la fonction `spyOn` pour faire des espions Jasmine afin de refaire fonctionner les tests existants.
- Dans les tests de `app`, ajouter un test vérifiant la valeur de `welcomeMsg` provenant de l'injection de dépendance soit bien présent dans le header.
- Dans les tests de `app`, ajouter un test vérifiant la bonne exécution de la fonction `updatePrice` en utilisant des espions pour les méthodes `addProduct` et `decreaseStock`.
- Dans les tests de `app-product`, faire fonctionner les tests existants de la même façon que les tests de `app` (avec des mock et des spy).
- Ajouter des tests au `CustomerService`. Un test pour vérifier que le panier est initialisé sans produit, un autre pour valider que `addProduct` ajoute bien le produit au panier et un dernier pour valider le calcul du prix total du panier.
- Ajouter des tests au `ProductService`. Un pour vérifier qu'il y a bien 4 produits au départ, un validant le fonctionnement de la fonction `isTheLast` et un dernier pour la fonction `decreaseStock`.

TP6 : Les Pipes

Nous allons à présent utiliser les `pipes`, afin de formatter le contenu de notre application.

Dans un premier temps, nous allons utiliser les `pipes` disponibles dans le framework : `uppercase` et `currency`.

- Dans le templace du composant `produit`, utiliser le `pipe` `uppercase` afin d'afficher le titre en majuscule
- Dans le template du composant `product`, utiliser le `pipe` `currency` afin d'afficher le prix d'un produit avec la devise *euro* et avec deux chiffres après la virgule.
- Ajoutez également le `pipe` `currency` pour l'affichage du total sur la page principale `app.component.html`
- Pour spécifier la locale du projet, il faut ajouter dans `app.module.ts` les lignes suivantes :

```
import { LOCALE_ID } from '@angular/core';
import { registerLocaleData } from '@angular/common';
import localeFr from '@angular/common/locales/fr';

registerLocaleData(localeFr);
```

et dans la section `providers` du `@NgModule` :

```
{provide: LOCALE_ID, useValue: navigator.language}
```

Nous allons à présent créer notre propre `pipe`, qui va nous permettre de trier une collection de produit par sa propriété `title`.

- Créer un nouveau `pipe` grâce à `@angular/cli`
- Implémenter la méthode de transformation, dans laquelle nous allons trier un tableau via la méthode `sort` du prototype `Array`
- Utiliser votre `pipe` dans le template du `ngFor`
- Nous allons à présent ajouter un paramètre à notre `pipe`. Ce paramètre permettra de définir la propriété sur laquelle le tri doit s'effectuer.

Tests

- Résoudre les nouvelles injections de dépendances afin que les tests existants fonctionnent.
- Ajouter un test de `SortPipe`, passer un tableau de produit au pipe et vérifier que la valeur de retour est bien le tableau trié.

TP7 : Communication avec une API REST

Après avoir reçu de la part du formateur un serveur REST développé en NodeJS, nous allons manipuler cette API pour récupérer la liste des produits, et persister le panier de l'utilisateur.

Pour lancer le serveur REST, vous devez exécuter la commande suivante :

```
cd server
npm install
npm start
```

Le serveur sera disponible via l'URL `http://localhost:8080/rest/`.

Cette API propose plusieurs points d'entrée :

- `GET` sur `/products` retournera la liste des produits
- `GET` sur `/basket` retournera le panier de l'utilisateur
- `POST` sur `/basket` pour ajouter un nouveau produit au panier de l'utilisateur
- Il est nécessaire d'importer le module `HttpClientModule` dans le module `AppModule`

- Nous allons tout d'abord modifier le service `ProductService`. Dans la méthode `getProducts`, nous allons envoyer une requête `HTTP` vers l'API correspondante.
- A la reception de la requête, utiliser l'opérateur `map` pour construire des object Products.
- Modifier le composant `AppComponent` en conséquence.
- Nous allons à présent modifier, de la même façon, le service `CustomerService`.
    - Créez une méthode `getBasket` avec le même fonctionnement que le point précédent
    - Implémentez une méthode `addProduct` dans laquelle nous allons envoyer une méthode `POST` pour ajouter un produit au panier de l'utilisateur.
- Modifiez le composant `AppComponent` afin d'utiliser la nouvelle version des services `CustomerService` et `ProductService`.

Tests

- Dans les tests de `app`, mettre à jour les tests pour s'adapter aux nouvelles signature des services. Dans les mock, utiliser `of()` pour créer des observables à partir d'une valeur de retour (il faudra également `import { of } from 'rxjs';`).
- Dans les tests de `ProductService` et `CustomerService`, ajouter au module de test l'import de `HttpClientTestingModule`. Une fois cela fait, mettre à jour les tests en simulant les réponses du serveur et en prenant en compte les nouvelles signatures des méthodes.

TP8 : Router

Nous allons intégrer dans notre application le routeur proposé par défaut dans le framework.

- Créez deux composants : `home` et `basket`
    - le composant `home` aura la charge d'afficher le contenu de la page que nous avons implémenté dans les TPs précédents
    - le composant `basket` permettra d'afficher, pour l'instant, le contenu du panier de l'utilisateur (via le pipe `json`)
- Ajoutez à votre application la configuration nécessaire pour le fonctionnement du router. Pour cela, nous allons utiliser la méthode `forRoot` mis à disposition par le module `@angular/router`
- Dans le template du composant `Application`, nous allons utiliser la directive `router-outlet` afin d'indiquer le point d'insertion des différentes pages de l'application.
- Ajoutez la directive `routerLink` dans le composant `menu` afin de rediriger l'utilisateur vers les deux composants que nous venons de créer.

Tests

Le routing en lui même est une fonctionnalité du framework Angular. Ce n'est pas le rôle des tests de notre application que de vérifier que le router d'Angular fonctionne correctement. Nous allons donc simplement adapter nos tests pour qu'ils fonctionnent à nouveau.

- La grande majorité de l'intelligence du composant `app` ayant été migré dans le composant `home`, l'ensemble des tests doivent également être migrés.
- Pour chaque composant utilisant des directives du module router, il est nécessaire d'importer le module pour que ces directives passent. Pour définir un routage minimaliste, utiliser l'import `RouterModule.forRoot([], {useHash: true})`.

TP9 : Gestion des Formulaires

Nous allons créer une nouvelle vue qui permettra de valider la commande.

Pour ce faire, nous allons commencer par créer une classe et un service pour gérer la validation.

- Dans un nouveau fichier `model\customer.ts`, créez une nouvelle classe `Customer` ayant les propriétés suivantes :
  - name de type `string`
  - address de type `string`
  - creditCard de type `string`
- Dans le service `service\CustomerService.ts` rajouter une méthode `checkout(customer)` qui doit :
  - faire un `POST` sur `/basket/confirm` pour persister la commande d'un client côté serveur

Pour interagir avec ces nouvelles fonctionnalités, nous allons utiliser le composant `basket` créé précédemment. Il affichera :

- le panier de manière simplifiée (une liste avec le nom et le prix de chaque produit)
- un formulaire permettant de saisir les informations du client.

Ajoutez un lien dans le composant `Home` qui pointe vers la page `/basket`.

Ce formulaire devra respecter les contraintes suivantes :

- Exécution de la méthode `checkout` lorsque l'évènement `ngSubmit` est émis. Après avoir reçu la réponse du serveur, redirigez l'utilisateur sur la page `home`
- un champ `input[text]` pour saisir le nom du client qui devra
  - être lié sur la propriété `name` de l'objet `Customer`
  - être requis (grâce à l'attribut *required*)
  - avoir la class CSS `has-error` s'il n'est pas valide

- un champ `textarea` pour saisir l'adresse du client qui devra
  - être lié sur la propriété `address` de l'objet `Customer`
  - être requis (grâce à l'attribut *required*)
  - avoir la class CSS `has-error` s'il n'est pas valide
- un champ `input[text]` pour saisir un code de carte bleue factice qui devra
  - être lié sur la propriété `creditCard` de l'objet `Customer`
  - être requis (grâce à l'attribut *required*)
  - respecter le pattern `^[0-9]{3}-[0-9]{3}$` qui correspond par exemple à `123-456`
  - avoir la class CSS `has-error` s'il n'est pas valide
  - afficher le message `Invalid pattern. Example : 123-456` si le pattern est incorrect
- un bouton `button[submit]` pour valider la saisie qui devra :
  - être désactivé si tout le formulaire n'est pas valide

Pour information, voici le template à utiliser pour ajouter un champ de formulaire dans votre page :

```
<div class="form-group has-error">
    <label class="control-label" for="name">Name</label>
    <input type="text" id="name" name="name" class="form-control">
</div>
```

Tests

- Dans les tests du composant `basket`, ajouter l'import du module `FormsModule` pour pouvoir gérer toutes les nouvelles directives utilisées.
- Ajouter un test au niveau de la description du panier vérifiant que chaque ligne de la liste contient bien le titre et le prix des produits du panier.
- Ajouter un test de l'activation de la classe `has-error` des champs de formulaire quand la valeur saisie n'est pas valide. Attention, pour que la validation de formulaire se déroule entièrement dans le cadre des tests, vous aurez besoin de toutes ces étapes :

```
const waitValidation = async fixture => {
  fixture.detectChanges();
  await fixture.whenStable();
  fixture.detectChanges();
};
```

- Ajouter un dernier (!) test sur l'activation du bouton submit du formulaire. Changer les valeurs saisies pour changer l'état de validation du formulaire et vérifier que le bouton est actif quand le formulaire est valide et désactivé quand il est invalide.