# Angular (TP)

*Bienvenue à tous !*

*Votre client du jour est le* ***Sanctuaire du jeu*** *de Nantes. Il souhaite que vous produisiez une application afin que ses clients puissent* ***consulter****,* ***commander*** *et* ***acheter*** *des* ***jeux de société*** *sur son site.*

## TP 1 : Créer une application et un premier composant

*Commençons par initialiser notre application et créer notre premier composant !*

#### Actions du TP

* **Créer une application** nommée « SanctuaireDuJeu » avec le **CLI d’Angular**.

*Dans les options d’installation, refuser la mise en place du SSR.*

* **Remplacer le contenu** de la page de démonstration (fichier « app.component.html ») par un texte personnalisé *« Bienvenue au Sanctuaire du jeu ».*
* **Créer un composant « product »** qui servira à afficher un produit (le laisser tel quel pour l’instant).
* **Afficher le composant** « product » dans votre page.
* **Lancer l’application** avec le **CLI d’Angular** (« ng serve -o »).

## TP 2 : Ajouter la library « Angular Material »

*Dans cette partie, nous allons installer Angular Material afin d’embellir notre page d’accueil.*

#### Actions du TP (partie 1) : Installer Angular Material

* **Ajouter la bibliothèque Angular Material**, via le ***schematic*** « @angular/material ».

*Dans les options d’installation, autoriser la mise en place des animations.*

* **Créer un composant « Menu »** avec le ***schematic*** « navigation » d’Angular Material.

*Pour cela, se référer à la documentation officielle d’Angular Material sur les schematics :*

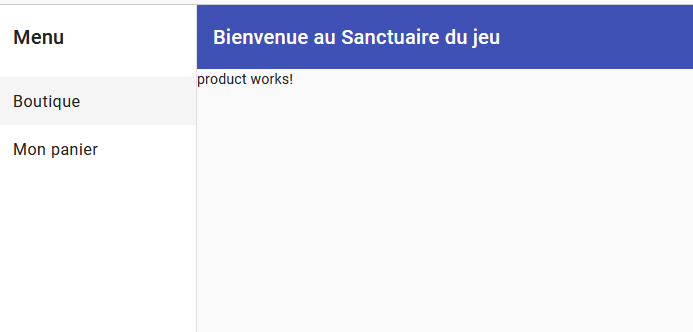
*https://material.angular.io/guide/schematics*

**Attention :** Les schematics ne prennent pas en compte les nouvelles palettes par défaut. Pour compenser cela, ajouter la ligne suivante dans le fichier « angular.json » :

"@angular/material/prebuilt-themes/indigo-pink.css"

|  |
| --- |
| "build": {  ...  "styles": [  "@angular/material/prebuilt-themes/indigo-pink.css",  "src/styles.scss"  ],  },  "test": {  ...  "styles": [  "@angular/material/prebuilt-themes/indigo-pink.css",  "src/styles.scss"  ],  }, |

* **Vérifier** que les nouvelles dépendances ont bien été ajoutées dans le fichier « package.json ».
* **Remplacer le contenu** de la page principale de l’application par le composant « Menu ».
* **Modifier le titre** **dans la toolbar** par « Bienvenue au Sanctuaire du jeu ».
* **Modifier le contenu du menu**: rempalcer les libellés des liens par « Boutique » et « Mon panier ».



*TP2 (menu)*

#### Actions du TP (partie 2) : Valoriser les composants

* **Copier le dossier « img »** (voir « Ressources TP ») dans le dossier « public » du projet.
* **Utiliser la directive « ng-content »** au sein de votre composant « menu » pour pouvoir projeter un contenu. Pour cela, remplacer le commentaire du composant « menu » (**<!-- Add Content here -->**) par « <ng-content /> ».
* **Projeter le composant « Product » dans le composant « Menu »**. Pour cela, aller dans le fichier « app.component.html » et insérer le composant « product » dans les balises du composant « app-menu ».
* **Importer le MatCardModule et le MatButtonModule** dans le composant « product ».
* **Insérer le code du composant « mat-card » dans le composant « product »**.

*Pour cela, se référer à la documentation officielle d’Angular Material sur le composant « mat-card » :*

https://material.angular.io/components/card/examples

* **Modifier votre composant « Product » avec les données du fichier « assets-tp1 »** (voir « Ressources TP »), afin de valoriser les différents éléments du composant « mat-card ».
* **Ajouter un bouton « Acheter »** dans le composant « mat-card ».
* **Ajouter la directive « mat-button » au bouton « Acheter »** pour le styliser.
* **Ajouter les règles CSS adéquates** pour que les composants « Product » s’affichent côte-à-côte (s’il y en a plusieurs) avec une longueur de 400px.

|  |
| --- |
| **Règles CSS (« product.component.css »)** |
| mat-card {      max-width: 400px;  }  :host {      display: inline-block;      margin: 10px;  } |



*TP2 (card)*

## TP 3 : Rendre dynamique le composant « product »

*Il est maintenant temps d’ajouter de nouvelles features à notre application. De plus, nous allons pouvoir factoriser le composant « Product » afin d’en généraliser l’usage à tous les produits de notre catalogue.*

#### Actions du TP

* **Créer une interface** « Product »avec les champs suivants.

**Interface « Product »**

|  |  |  |
| --- | --- | --- |
| **Champ** | **Type** | **Infos** |
| id | Entier | \_ |
| title | Texte | Nom du produit. |
| description | Texte | \_ |
| photo | Texte | Path de la photo à afficher. |
| price | Entier | \_ |
| stock | Entier | \_ |

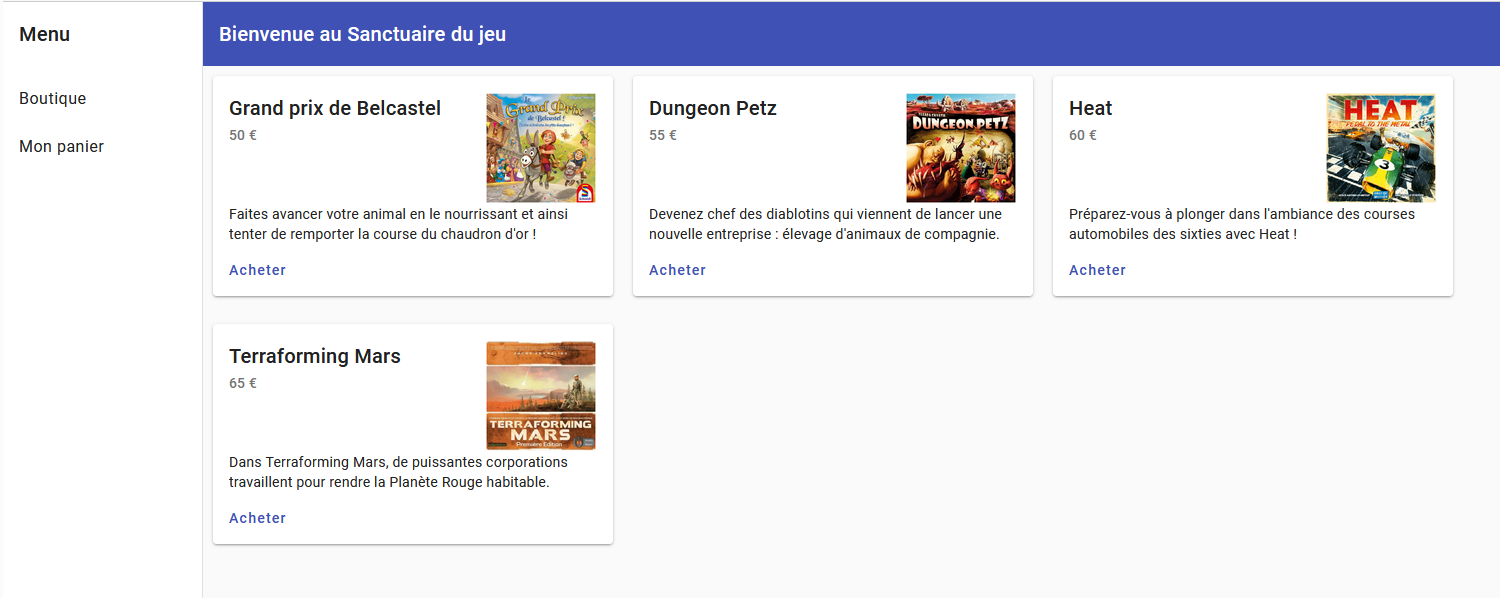
* **Déclarer une property « products » dans le composant « AppComponent »** et utiliser l’interface pour la typer en tableau de Product.
* **Valoriser la property « products »** avec les données du fichier « assets-tp3 » (voir « Ressources TP »).
* **Modifier le composant « AppComponent »** afin d’afficher un composant « Product » par élément présent dans la property « products ».
* **Modifier le composant « Product »** afin d’afficher les données fournies par son composant parent (« AppComponent »).
* **Déclarer une property « total » de type « number »** dans le composant « AppComponent ».
* **Ajouter une balise <p>** dans le composant « AppComponent » afin d’afficher « votre panier est vide » (si le total est égal à 0) ou le total des produits achetés (si le total est supérieur à 0).
* Au clic sur le bouton « Acheter », **décrémenter le stock** et **calculer le total**.

**Attention :** C’est le composant « AppComponent » qui doit être porteur de l’état du stock.

* **Si le stock est de 0**, ne pas afficher le produit concerné.
* **Si le stock est de 1**, appliquer la classe « stock-limite » sur le produit concerné.

*Ne pas oublier d’importer le « CommonModule » pour avoir accès à « ngClass » !*

|  |
| --- |
| **Règles CSS (« product.component.css »)** |
| .stock-limite {      outline: 3px solid red;  } |



## TP 4 : Ajouter un service et un pipe

*Actuellement, le composant « AppComponent » porte deux logiques : la logique d’affichage et la logique métier. A terme, cela risque de rendre la partie TS de ce composant extrêmement complexe. Il va donc falloir transférer la logique métier dans des services dédiés, pour que « AppComponent » ne porte plus que la logique d’affichage.*

#### Actions du TP

* **Créer un service « BasketService »**, et **y déclarer une property « \_basket »**. Cette property doit être **private** (d’où le préfixe « \_ »).
* **Ajouter les méthodes suivantes**.

**Spécifications (« BasketService »)**

|  |  |
| --- | --- |
| **Méthode** | **Description** |
| getBasket() | Récupérer tous les produits du panier. |
| addToBasket(product : Product) | Ajouter 1 produit au panier. |
| getTotal() | Calculer le prix total du panier. |

* **Créer un service « ProductService »**, et **y déclarer une property « \_products »**. Cette property doit être **private** (d’où le préfixe « \_ »).
* **Ajouter les méthodes suivantes**.

*Pour faciliter la déstructuration d’objet, pensez à ne pas nommer le paramètre entrant « id ».*

**Spécifications (« ProductService »)**

|  |  |
| --- | --- |
| **Méthode** | **Description** |
| getProducts() | Récupérer la liste de tous les produits. |
| getProduct(idProduct : number) | Récupérer 1 produit. |
| decreaseProduct(idProduct : number) | Décrémenter le stock d’un produit. |

* **Déplacer le code présent dans « AppComponent » vers les services**, et veiller à répartir correctement les responsabilités entre chaque service.
* La logique de notre catalogue doit être portée par le service « productService ».
* La logique du panier doit être portée par le service « basketService ».
* AppComponent ne doit pas porter de logique métier. Il ne doit pas avoir de properties dédiées à la logique métier : remplacer ses properties « métier » par des getters.
* **Utiliser le CurrencyPipe** pour formater l’affichage du **prix de vente** et le **total du panier**.

**Attention :** VSC peine à faire l’import de la *localeFr*. Vous allez donc devoir l’ajouter manuellement.

*import localeFr from '@angular/common/locales/fr'*

## TP 5 : Appels HTTP

*Nous avons hardcodé les produits de notre catalogue dans la partie front de l’application. En situation professionnelle, nous aurions stockés ces données dans une BDD, accessible depuis un serveur back. Nous allons donc simuler un serveur back et faire des appels HTTP pour être au plus proche de ce qui se fait sur un « vrai » projet.*

#### Actions du TP (partie 1) : Simuler un serveur back

* **Copier le dossier « serveur »** (voir « Ressources TP ») sur votre ordinateur.

*L’emplacement importe peu (ce peut être à la racine du projet).*

* **Exécuter les commandes suivantes depuis le dossier « serveur »**.

*Les endpoints sont maintenant disponibles et exposés à l’adresse suivante :*

*http://localhost:8080/api*

|  |
| --- |
| **Commandes (lancement du serveur « Express »)** |
| ***// Depuis le dossier « serveur »***  npm i  npm start |

**Spécifications (Endpoints)**

|  |  |
| --- | --- |
| **URL/Méthode** | **Description** |
| /api/products  *GET* | **Return :** Product[]  **Effet :** Renvoie tous les produits avec un stock > 0 |
| /api/products/:id  *GET* | **PathVariable :** 'id'  **Return :** Product  **Effet :** Renvoie les données d’un produit |
| /api/basket  *GET* | **Return :** Product[]  **Effet :** Renvoie tous les produits présents dans le panier |
| /api/basket  *POST* | **Body :** Un objet avec un champ « productId »  **Return :** Product (produit ajouté)  **Effet :** Ajoute 1 produit au panier, renvoie le produit ajouté |

#### Actions du TP (partie 2) : Communiquer avec le serveur pour récupérer les données

* **Modifier le service « ProductService »**.
* **Ajouter une méthode « fetchProducts() »**. Faire appel au endpoint « /products » dans cette méthode, et valoriser la property « \_products » du service avec les données renvoyées par le serveur.
* **Modifier la méthode « getProduct() »**. Remplacer le code existant par un appel au endpoint « /products/:id » dans cette méthode.
* **Faire appel à la méthode « fetchProducts() »** dans le cycle de vie approprié de « AppComponent » pour initialiser la property « products ».
* **Modifier le service « BasketService »**.
* **Modifier la méthode « getBasket() »**. Faire appel au endpoint « /basket » (GET) dans cette méthode, et valoriser la property « \_basket » du service avec les données renvoyées par le serveur.
* **Modifier la méthode « addToBasket() »**. Faire appel au endpoint « /basket » (POST) dans cette méthode, et modifier la property « \_basket » du service avec les données renvoyées par le serveur.
* **Déplacer la méthode « decreaseProduct() »** pour qu’elle ne soit exécutée que si la méthode « addToBasket() » est en succès.

**Attention :** La souscription à cet Observable doit se faire dans le composant appelant.

* **Faire appel à la méthode « getBasket() »** dans le cycle de vie approprié de « CatalogComponent » pour initialiser la property « \_basket ».

*Dans le cadre du TP, nous ne modifierons pas davantage les services. L’état du stock est maintenant dépendant du back : si vous souhaitez réinitialiser les données du serveur (et donc, l’état du stock), relancez-le.*

#### Bonus : Ajouter le bouton « reset »

* **Créer un bouton « Reset » dans le composant « Catalog »**.
* **Modifier le service « ProductService »** **et ajouter une méthode « reset() »**. Faire appel au endpoint « /reset » dans cette méthode. En cas de succès, faire appel aux méthodes « fetchProducts() » (productService) et « getBasket() » (basketService).

*Pour éviter les subscribes imbriqués, faire appel à l’opérateur « switchMap() ».*

**Spécifications (Endpoints)**

|  |  |
| --- | --- |
| **URL/Méthode** | **Description** |
| /reset  *GET* | **Return :** Void  **Effet :** Réinitialise le serveur |

## TP 6 : Routing

*Dans cette partie, nous allons développer de nouvelles fonctionnalités pour que notre utilisateur puisse naviguer sur les différentes pages de notre site.*

#### Actions du TP (partie 1) : Routing

* **Créer les Routes suivantes**.

**Routes**

|  |  |
| --- | --- |
| **Path** | **Composant** |
| '' | CatalogComponent |
| 'products/ :id' | ArticleComponent |

* **Ajouter un champ « title » à la Route qui mène à « CatalogComponent »**, afin de renommer l’onglet de la page.
* **Créer un composant « Catalog »**.
* **Transférer le code de « AppComponent » (HTML & TS)** dans le composant « Catalog ».
* **Ajouter la balise <router-outlet /> au composant « AppComponent »** pour y afficher le composant associé à la Route.
* **Créer un composant « Article »**. Pour les besoins du TP, ce composant aura un template HTML minimal.

|  |
| --- |
| **Template HTML (« ArticleComponent »)** |
| <img src="{{ product.photo }}" alt=""/>  <p>{{ product.title }}</p>  <p>{{ product.description }}</p> |

* **Déclarer une property « product » dans le composant « Article »**.
* **Valoriser la property « product » du composant « Article »** avec un appel à la méthode « getProduct() » du service « ProductService ». L’appel HTTP doit se baser sur le pathVariable récupéré depuis l’URL de la page.

#### Actions du TP (partie 2) : RouterLink

* **Composant « Menu » : Remplacer l’attribut « href » par la directive « routerLink »** sur le lien « Boutique ». Au clic sur le lien, l’utilisateur doit être renvoyé vers le composant « Article » associé.

*Ne pas oublier d’importer « RouterLink » pour avoir accès à la directive « routerLink » !*

* **Composant « Product » : Insérer une balise <a> dans la balise « mat-card-title ».** Ajouter une directive « routerLink » à la balise <a>. Au clic sur le lien, l’utilisateur doit être renvoyé vers le composant « Article » associé.

#### Bonus : Route avec « Wild card »

* **Créer un composant « NotFound »**. Créer une Route avec une *WildCard* qui redirige vers ce composant si l’utilisateur tente d’accéder à une URL invalide.

## TP 7 : Forms

*Dernière étape : soumettre notre commande avec un formulaire !*

#### Actions du TP : Créer un formulaire et envoyer la commande

* **Créer un composant « Basket »**.

*Ajoutez les imports nécessaires :*

*FormsModule, MatFormFieldModule, MatInputModule, MatButtonModule*

* **Créer la Route suivante**.

**Routes**

|  |  |
| --- | --- |
| **Path** | **Composant** |
| 'basket' | BasketComponent |

* **Composant « Menu » : Remplacer l’attribut « href » par la directive « routerLink »** sur le lien « Mon panier ». Au clic sur le lien, l’utilisateur doit être renvoyé vers le composant « Basket ».
* **Créer une interface « CommandeFormulaire »** avec les champs suivants.

**Interface « CommandeFormulaire »**

|  |  |  |
| --- | --- | --- |
| **Champ** | **Type** | **Infos** |
| name | Texte | \_ |
| address | Texte | \_ |
| creditCard | number | \_ |

* **Implémenter un formulaire (type « template-driven »)** dans le composant « BasketComponent ».

*Se référer à la documentation officielle d’Angular Material pour embellir les inputs :*

*https://material.angular.io/components/input/overview*

* Ce formulaire comporte 3 champs : « name », « address » et « creditCard ».
* Tous les champs sont obligatoires.
* Le champ « creditCard » doit contenir exactement 6 chiffres pour être valide.
* **Ajouter 1 message d’erreur sous les champs « name »/« address »** si le champ est en erreur.
* **Ajouter 2 messages d’erreur sous le champ « creditCard »**, en fonction de l’erreur (« required » ou « pattern »).
* **Désactiver le bouton « Soumettre »** si le formulaire n’est pas valide.
* **Ajouter la méthode « commander »**. Faire appel au endpoint « /basket/checkout » dans cette méthode.

**Spécifications (Endpoints)**

|  |  |
| --- | --- |
| **URL/Méthode** | **Description** |
| /basket/checkout  *POST* | **Body :** Un objet de type « CheckoutDetails »  **Return :** number  **Effet :** Traite la commande (console.log) et renvoie un numéro de commande |

* **Au clic sur le bouton « Soumettre », déclencher la méthode « commander() »**. Vérifier dans la console du navigateur que la commande a été envoyée.

**Attention :** Le back renverra une erreur 400 si le panier ne contient pas d’article.

#### Bonus : Guard si le panier est vide

* **Créer un composant « EmptyBasket »**.
* **Créer la Route suivante**.

**Routes**

|  |  |
| --- | --- |
| **Path** | **Composant** |
| 'empty' | EmptyBasketComponent |

* **Ajouter un guard à la Route « 'basket' »**. Permettre l’accès à cette Route si le panier n’est pas vide. Si le panier est vide, se servir du Router pour réorienter l’utilisateur vers Route « 'empty' ».