Liste des ressources de cette page

Associer une URL à un composant

En général utilisé lorsqu'on crée une nouvelle page, lier un composant et une URL permet d'afficher le composant lorsque l'URL est entrée dans le navigateur.

Pour ce faire :

1. Ouvrez `front-end/src/app/app.routing.module.ts`.
2. Importez votre composant.
3. Ajoutez un objet Route dans votre liste de routes, en associant l'URL de votre choix (propriété `path`) avec le composant que vous avez importé (propriété `component`).
4. Il faut rajouter dans votre app.component.html le router-outlet qui va afficher le composant à afficher lors de la navigation. Cette balise router-outlet sera remplacé par le composant configuré pour chaque page.

```
<div class="main-content">
    <router-outlet></router-outlet>
</div>
```

Ressources :

- Fichier `front-end/src/app/app.routing.module.ts` dans le projet corrigé.
- Doc Angular : Routes.

Créer un composant

Pour cela, il va falloir d'abord falloir créer un dossier qui va contenir les fichiers du composant. Ce dossier peut être créé n'importe où dans `src/app/`, mais il est conseillé d'établir une structure intelligente pour ne pas passer son temps à chercher les fichiers. Si vous avez besoin d'exemples, n'oubliez pas que le projet support corrigé est là pour ça.

Dans le dossier que vous venez de créer, ajoutez 3 fichiers :

[nomDeVotreComposant].component.html

Il faudra lui ajouter le HTML que vous souhaitez voir s'afficher lorsque votre composant est appelé.

[nomDeVotreComposant].component.scss

C'est là-dedans que vous écrirez le code Sass utilisé pour styliser votre composant.

Sass est une surcouche de CSS (--> vous pouvez donc écrire du CSS classique, il sera parfaitement compris) qui lui ajoute de nouvelles fonctionnalités très pratiques.

Si vous souhaitez développer vos compétences en Sass, parcourez ce tuto.

[nomDeVotreComposant].component.ts

Ce fichier typescript contient toute la logique de votre composant.

Pour fonctionner, il n'a pas besoin de grand chose :

1. Importer Component et OnInit depuis angular/core :
   ```
   import { Component, OnInit } from '@angular/core';
   ```
2. Donner un nom pour la balise servant à appeler le composant, et indiquer où trouver le template et les fichiers de style :
   ```
   @Component({
       selector: 'app-quiz-list',
       templateUrl: './quiz-list.component.html',
       styleUrls: ['./quiz-list.component.scss']
   })
   ```
3. Exporter la classe de votre composant. Elle doit contenir au minimum un constructeur et une fonction ngOnInit vides :
   ```
   export class MyEpicComponent implements OnInit {
       constructor() {}
       ngOnInit(): void {}
   }
   ```

Il faudra ensuite probablement y ajouter de la logique, selon vos besoins.

Ajout du composant dans l'app module

Il faut déclarer maintenant votre composant nouvellement créer dans app.module.ts, et le rajouter dans la liste des declarations.

@NgModule({
declarations: [
AppComponent,
MyEpicComponent]

Créer une page

Pour créer une page, il vous faut 2 choses :

1. Créer un composant (qui sera la page à afficher)
2. Lier ce composant à une URL (pour pouvoir y accéder).

Créer un service

Les services se trouvent dans un dossier à part, au même niveau que le dossier `app` ou que `assets`.
Les services ont plusieurs rôles:

- L'échange de données entre composants. Les composants vont récupérer les données exposées par le service, ils vont pouvoir utiliser les fonctions du service qui vont mettre à jour ses données.
- Garantir la séparation des rôles (separation of concerns principle). Chaque service aura un rôle spécifique lié à un modèle. Nous avons par exemple le `QuizService` qui s'occupe de la gestion des quizzes ou l'`UserService` qui s'occupe des utilisateurs.
- Comme les services sont des entités séparées des composants, partagées, on va les utiliser pour récupérer les données venant de mocks ou de notre serveur. Le service est en charge de s'assurer que les données sont à jour et disponibles en fonction de l'état courant de l'application

Pour créer un service, on crée un fichier `nom-service.service.ts`. On utilise généralement comme `nom-service`, le model qui va être géré par notre service (quiz, user, etc..):

Pour définir une classe comme un service dans Angular et lui permettre de l'injecter dans un composant, il faut utiliser le décorateur @injectable :

```
import { Injectable } from '@angular/core';

...

@Injectable({
  providedIn: 'root'
})
export class QuizService {
```

(La valeur `root` permet de forcer le service à n'être disponible que sous une seule instance au niveau de l'application complète, ce qui permet de s'assurer que chaque composant utilisera la même instance unique, et donc que les éléments qu'il contient seront tous mis à jour en même temps. Dans une utilisation plus avancée, il est possible de définir le champ d'application d'un service à une sous partie de l'application, comme un module de composant).

Ensuite on ajoute les attributs que le service doit contenir, comme par exemple :

```
//The list of quiz. The list is retrieved from the mock.
private quizzes: Quiz[] = QUIZ_LIST; // Ici on initialise la valeur avec un mock QUIZ_LIST
```

Ensuite, le constructeur :

```
// The service's constructor. Le constructeur peut prendre en paramètre les dépendances du service - comme ici, HttpClient qui va permettre de récupérer les données d'un serveur
constructor(private http: HttpClient) {}
```

Et enfin les fonctions utilisables par le service, telles que :

```
addQuizz() { … }

deleteQuizz(id: string) { … }
```

Créer un observable

Les observables sont des flux de données qui peuvent être écoutés pour y récupérer les données.

Dans l'exemple ci-dessous tiré du cours, on initialise l'observable avec une valeur initiale et ensuite on met à jour l'observable avec une nouvelle valeur.

Pourquoi exposer un observable qui contient un tableau de quizz et pas simplement un tableau de quiz ?

Le but est d'avoir une seule source de vérité dans toute l'application, un seul attribut qui contient l'information (dans un service), et de pouvoir souscrire à cet attribut pour que les composants puissent réagir aux changements. Les composants n'ont pas besoin de savoir quel événement met à jour les données (un composant après une action utilisateur, ou de nouvelles données récupérées d'un serveur etc..), ils réagissent juste au changement avec la certitude que les données sont correctes.

Comment créer un observable ?

Dans votre service, créez l'observable souhaité grâce à la classe BehaviorSubject :

```
// Observable containing a list of quizzes, initialized with an empty array.
// Naming convention: Add '$' at the end of the variable name to highlight it as an Observable.
public quizzes$: BehaviorSubject<Quiz[]> = new BehaviorSubject([]);
```

Comment mettre à jour un observable ?

Dans le service où vous avez créez votre observable, là où vous souhaitez mettre à jour la valeur de l'observable, il vous suffit d'utiliser la méthode `next()` :

```
this.quizzes$.next(newValue);
```

> Si vous avez besoin de la valeur de l'observable dans le service même où il est créé (ce qui est souvent le cas), il est conseillé d'avoir un attribut supplémentaire dans votre classe, que vous mettrez à jour en même temps que l'observable.

Subscribe à un Observable

Pour récupérer les données d'un observable dans un composant, il suffit de modifier son constructeur :

1. Ajouter le service en paramètre du constructeur.
2. Appeler la méthode `subscribe()` de l'observable.
3. (Facultatif en théorie) Mettre à jour une variable interne à chaque fois que la valeur de l'Observable est modifiée

```
constructor(..., public quizService: QuizService) {
  this.quizService.quizzes$.subscribe((quizzes: Quiz[]) => {
    this.quizList = quizzes;
  });
}
```

Mock

Un mock sert à simuler un comportement. Dans notre cas, on s'en sert essentiellement pour simuler un appel API : au lieu de récupérer des données de l'API, on va les chercher dans un fichier. Ce fichier est construit pour avoir exactement la même forme que ce qu'on s'attendrait à recevoir de l'API, de manière à faciliter la transition vers les données réelles.

Créer un mock

Dans votre dossier src, vous devriez avoir un dossier `mocks/` ; si ce n'est pas le cas, créez-le. À l'intérieur de ce dossier, créer un fichier par donnée dont vous souhaitez simuler la récupération. Le nom du fichier doit être `xxxx.mock.ts`. Dans ce fichier, il vous faudra simplement importer les modèles dont vous avez besoin, et exporter une variable contenant les données simulées :

```
import { Quiz } from '../models/quiz.model';
...
export const QUIZ_LIST: Quiz[] = [
  {
    id: '1',
    name: 'Les Acteurs',
    theme: 'Actor',
    questions: [],
  },
  {
    id: '2',
```

```
    name: 'Les technos WEB',
    questions: [],
  }
];
```

Utiliser un mock

Pour utiliser un mock dans un composant ou un service, il suffit de l'importer :

```
import { QUIZ_LIST } from '../mocks/quiz-list.mock';
```

Et vous avez accès à `QUIZ_LIST`.

Directives

Les directives en Angular nous permettent de manipuler les éléments HTML de la page, qui ajoutent des comportements aux éléments du DOM. Elles sont utilisées pour manipuler l'apparence et le comportement des éléments, ainsi que pour fournir des fonctionnalités réutilisables à travers les composants.

Voici trois exemples de directives natives en Angular :

*ngFor

*ngFor est une directive structurelle qui permet l'itération d'un tableau ou d'un objet itérable et crée une instance de modèle pour chaque élément.
Voici un exemple avec le composant UserList de la correction:

```
<div class="user" *ngFor="let user of userList">
   <app-user [user]="user" (deleteUser)="deleteUser($event)"></app-user>
</div>
```
```
export class UserListComponent implements OnInit {
  public userList: User[] = [];

  constructor(private userService: UserService) {
    this.userService.users$.subscribe((users: User[]) => {
      this.userList = users;
    });
  }
 }
}
```

*ngIf

*ngIf est une directive structurelle qui inclut ou exclut des éléments de manière conditionnelle sur la base d'une expression booléenne.
Voici un exemple tiré du composant EditQuiz de la correction :
````
````

```
<div *ngIf="quiz">
  <h2>{{quiz.name}}</h2>
  <app-question-form [quiz]="quiz"></app-question-form>
  <app-question-list [quiz]="quiz"></app-question-list>
</div>
````

Il existe de nombreuses autres directives comme `ngSwitch` etc.. Vous pouvez la documentation d'angular sur les directives.

@Input

Dans un composant, les inputs permettent de récupérer les données envoyés par le composant parent.

Pour déclarer un `@Input`, il faut créer un attribut dans la classe du composant et ajouter la décoration `@Input` :

```
@Input()
quiz: Quiz;
```

Pour transmettre la valeur de l'input depuis le composant parent, il faut appeler la variable avec le même nom que dans le composant enfant en utilisant des crochets quand on appelle le composant enfant :

```
<app-quiz [quiz]="currentQuiz"></app-quiz>
```

Dans le code précédent:

- [quiz] est le nom de l'input. Ce nom doit être identique au nom de l'attribut déclaré dans le composant enfant QuizComponent.
- Les crochets [ ] signifient que cet attribut est un Input.

Les données venant d'un @Input ne sont pas disponibles dans le constructor du composant mais à partir du ngOnInit.

Dans Angular, les hooks du cycle de vie des composants sont des méthodes qui sont appelées à des étapes spécifiques de la vie d'un composant.

Le constructeur est la première méthode appelée lorsqu'un composant est créé, tandis que ngOninit est appelé après que les entrées du composant aient été initialisées et que le composant soit prêt à être utilisé.

Il existe de nombreuses autres étapes du cycle de vie d'un composant comme ngAfterViewInit, qui est appelé après l'initialisation de la vue du composant, et ngOnDestroy, qui est appelé juste avant la destruction du composant.

Documentation: **https://angular.io/guide/lifecycle-hooks**

@Output

Le décorateur @Output est utilisé en association avec la classe EventEmitter pour permettre à un composant enfant de communiquer avec son composant parent en envoyant des données et en déclenchant des actions basées sur les interactions de l'utilisateur :

```
import { Output, EventEmitter } from '@angular/core';
...
@Output()
quizSelected: EventEmitter<boolean> = new EventEmitter<boolean>();
...
this.quizSelected.emit(true); // emit event to inform that the quiz has been selected.
```

Dans le parent, on utilise les parenthères au moment de l'appel à l'enfant pour indiquer qu'on attend un output :

```
<app-quiz (quizSelected)="selectQuiz($event)"></app-quiz>'
```

Dans le code précédent:

- (quizSelected) est le nom de l'output. Ce nom doit être identique au nom de l'attribut déclaré dans le composant enfant QuizComponent.
- Les parenthèses signifient que cet attribut est un Output.
- La fonction selectQuiz($event) sera appelée lorsque qu'un événement sera émis par le composant enfant. $event contient la valeur émise par l'événement. Dans notre exemple, un booleén avec pour valeur : true.

Il faudra évidemment créer la méthode `selectQuiz(selected: boolean)` dans le parent.

Pour en savoir plus sur les inputs outputs: https://angular.io/guide/component-interaction

Models

Un modèle est une interface Angular permettant de définir une structure de données que vos composants/fonctions pourront reconnaître et manipuler.

Créer un modèle

Les modèles sont stockés dans un dossier `models/` à la racine du front (au même niveau que le dossier `app/`). Si le dossier n'existe pas, créez-le.

À l'intérieur de ce dossier, créez un fichier `xxxx.models.ts`. À l'intérieur de ce dernier, il vous suffit d'exporter une interface :

```
import { Question } from './question.model';

export interface Quiz {
  id: string;
  name: string;
  theme?: string;
  questions: Question[];
}
```

Dans ce code:

- L'import n'est nécessaire que parce qu'on utilise un modèle personnalisé (Question) comme type d'un attribut de l'interface, il faut donc l'importer pour qu'Angular sache ce qu'est Question.
- Le `?` d'interrogation dans la définition du modèle signifie que ce champ n'est pas obligatoire, un objet Quiz peut être défini sans thème. Les autres champs doivent être définis.

Utiliser un modèle

Pour utiliser un modèle comme type de données dans un composant, il suffit de l'importer :

```
import { Quiz } from '../models/quiz.model';
...
private quizzes: Quiz[] = QUIZ_LIST;
```

Navigation

Pour naviguer d'une page à l'autre, vous avez besoin d'avoir configuré votre route/page sur laquelle vous souhaitez aller.

Il faut au préalable rajouter dans votre app.component.html le router-outlet qui va afficher le composant à afficher lors de la navigation.

```
<div class="main-content">
    <router-outlet></router-outlet>
</div>
```

Une fois cette configuration effectuée, vous avez plusieurs moyens de lancer la navigation depuis un bouton.

- Directement dans la vue du composant en configurant le bouton avec `routerLink`. Exemple :
  Quiz
- Depuis une fonction depuis la classe du composant. Exemple :

```

// Inject router.
constructor(private router: Router, public quizService: QuizService) {}

editQuiz(quiz: Quiz): void {
  this.router.navigate(['/edit-quiz/' + quiz.name]); // navigate
}
```

La fonction editQuiz peut être appelée depuis la vue du composant :

```

<app-quiz (editQuiz)="editQuiz($event)"></app-quiz>

```

Créer un formulaire

Pour créer un formulaire en angular, on va utiliser un objet appelé FormGroup qui va contenir l'ensemble de nos champs de formulaire et va nous permettre d'avoir des fonctionnalités avancées permettant d'évaluer la validité des entrées du formulaire (par exemple : définir que tous les champs doivent être remplis et le formulaire est invalide tant que ça n'est pas le cas), ou de réagir aux changements dans un formulaire :

```
export class QuizFormComponent implements OnInit {   /**
  * QuizForm: Object which manages the form in our component.
  * More information about Reactive Forms: https://angular.io/guide/reactive-forms#step-1-creating-a-formgroup-instance
  */
  public quizForm: FormGroup;

  ...

  constructor(public formBuilder: FormBuilder, public quizService: QuizService) {
    this.quizForm = this.formBuilder.group({
      name: [''],
      theme: ['']
    });
    // You can also add validators to your inputs such as required, maxlength or even create your own validator! Example: name: [' ', Validators.required, Validators.minLength(4)] -> in this example, the field name is a string, is a required field with a minimum length required of 4 characters.
// More information: https://angular.io/guide/form-validation#validating-input-in-reactive-forms
  }
}
```

Ce code importe les classes FormBuilder et FormGroup du module @angular/forms, déclare une propriété quizForm de type FormGroup, et injecte le service FormBuilder dans le constructeur du composant. Dans le constructeur, nous utilisons le service FormBuilder pour créer un quizForm FormGroup avec deux champs de saisie de texte nommés `name` et `theme`, initialisés avec des valeurs de chaînes vides. Nous pouvons ensuite utiliser le groupe de formulaires quizForm dans le modèle de notre composant pour rendre le formulaire et gérer les entrées de l'utilisateur.

- Documentation sur les reactive forms
- Documentation avancée sur la validation de champs

Modifier dynamiquement le style d'un composant

Vous allez avoir besoin de modifier du style dynamiquement à l'intérieur d'un composant, en fonction des configurations que vous avez récupérées depuis un service par exemple.

En angular on a tout ce qu'il faut pour modifier dynamiquement du style, on ne veut pas utiliser des choses comme document.getElementById(id).style.property = value .

On a 2 moyens de faire proprement avec Angular avec ngStyle et ngClass:

ngClass qui va nous permettre de changer dynamiquement la class en fonction de notre logique : https://angular.io/api/common/NgClass

Par exemple: objectif: changer la taille du texte en ajoutant la classe text-big ou text-small sur mon span.

 My text

On définit le CSS des 2 classes (bien souvent, on le définit dans un fichier CSS partagés - comme styles/variables.scss par exemple - pour ne pas avoir à dupliquer le CSS dans tous les composants.

```
.text-big {
  font-size: 20px;
}
.text-small {
  font-size: 10px;
}
```

Dans l'html, le ngClass s'utilise comme ça :

<span [ngClass]="{'text-big': true, 'text-small': false}"> My text

<span [ngClass]="{'text-big': isTextBig, 'text-small': isTextSmall}"> My text

Quand c'est true la class est mise, quand c'est false elle ne l'est pas.

Pour assurer une configuration pour tous les composants, vous aurez certainement besoin d'un service de configuration.

Vous pouvez avoir un ConfigurationService avec un observable qui s'appelle fontConfiguration$ qui contient

{'text-big': true, 'text-small': false}

ou alors une fonction getFontConfiguration dans le service qui renvoie:

```
getFontConfiguration() {
 return {'text-big': true, 'text-small': false};
}
```

et dans les composants:

<span [ngClass]="configurationService.getFontConfiguration()"> My text

Avec cette version, vous pouvez très facilement changer une classe, en rajouter une, modifier la logique dans votre service de configuration et tous les composants auront les changements.

ngStyle: fonctionnement similaire à ngClass mais qui modifie directement la propriétés CSS en question: https://angular.io/api/common/NgStyle

<some-element [ngStyle]="{'font-style': styleExp}">...</some-element>

Reconnaissance de vocale

Certains groupes ont besoin de faire de la reconnaissance vocale pour permettre à l'utilisateur d'interagir différemment au sein d'un quiz.

Voici un exemple d'implémentation de la reconnaissance vocale en utilisant la SpeechRecognition API native en javascript. Information importante: cette API ne fonctionne aujourd'hui que sur Firefox et Chrome, nous l'utilisons à des fins d'expérimentation.

Documentation

Exemple d'implémentation utilisant du JS

Voici 2 exemples d'implémentation en utilisant Angular :

- Exemple simple :

```
@Component({
   selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.scss'],
})
export class AppComponent {

  recognition: any;
  constructor() {
    this.recognition = new webkitSpeechRecognition();
    this.recognition.continuous = true;
    this.recognition.interimResults = true;
    this.recognition.lang = 'en-US';
  }

  startSpeechRecognition() {
    this.recognition.start();
    console.log('Speech recognition started');
    this.recognition.onresult = function(event) {
      var interim_transcript = '';
      for (var i = event.resultIndex; i < event.results.length; ++i) {
        if (event.results[i].isFinal) {
          console.log('Final result: ' + event.results[i][0].transcript);
        } else {
          interim_transcript += event.results[i][0].transcript;
            }
    }
      console.log('Interim result: ' + interim_transcript);
    };
  }
}
```

- Version plus avancée avec grammaire.

```
declare var webkitSpeechRecognition: any;
declare var webkitSpeechRecognitionEvent: any;
@Component ...
export class MyComponent {
  recognition: any;
```

```
  @HostListener("document:keydown", ["$event"])
  onKeydown(event: KeyboardEvent) {
  if (event.key === " ") {
    this.startSpeechRecognition();
  }

  constructor() {
    this.initializeSpeechRecognition();
  }

  initializeSpeechRecognition() {
    this.recognition = new webkitSpeechRecognition();
    var SpeechGrammarList = SpeechGrammarList || (window as any).webkitSpeechGrammarList
    var SpeechRecognitionEvent = SpeechRecognitionEvent || webkitSpeechRecognitionEvent
    var inputs = [ 'ok', 'stop', 'start'];
    if (SpeechGrammarList) {
      var speechRecognitionList = new SpeechGrammarList();
      var grammar = '#JSGF V1.0; grammar colors; public ATTENTION_A_REMPLACER_CF_PLUS_BAS= ' +
inputs.join(' | ') + ' ;'
      speechRecognitionList.addFromString(grammar, 1);
      this.recognition.grammars = speechRecognitionList;
    }
    this.recognition.continuous = true;
    this.recognition.interimResults = true;
    this.recognition.lang = "en-US";
    this.recognition.onresult = (event) => {
    const inputs = event.results[0][0].transcript;
      console.log(`Input received: ${inputs}`);
    };
    this.recognition.onspeechend = function() {
      this.recognition.stop();
    }
    this.recognition.onnomatch = function(event) {
      console.log("no match");
    }
    this.recognition.onerror = function(event) {
      console.log("error");
    }
  }
}

}
```

Notez que la variable ATTENTION_A_REMPLACER_CF_PLUS_BAS doit être remplacé par `< color >`sans espace.

Contacter l'assistance du site

Connecté sous le nom « duong Thi Thanh Tu » (Déconnexion)
Résumé de conservation de données
Obtenir l'app mobile

Fourni par Moodle