

Developpement Web Avancé

Full stack web app with Spring boot
and Angular

Description du projet et environnement du travail

Description du sujet

Nous souhaitons présenter notre projet de développement d'une application web dédiée à la recherche et au développement au sein de notre entreprise IA-Technology.

Voir Document

Préparation de l'environnement

Utiliser la version ultimate de intelliij



Wamp, Xamp ou autre pour :

- Un serveur web
- Un serveur de données Sql



Postman pour le test
de l'Api



Méthodologie -Agile-

- Travailler en groupe de 5-6 étudiants et partager le projet entre vous sur git en m'invitant pour poursuivre vos travaux.
- Essayer de créer un backlog produit.
- Planifier des sprints de 14 jours.
- Répartir les tâches entre vous.

Partie1 : Spring Boot & Spring Data

NB : Utiliser le code partagé sur classroom

Pourquoi Spring Boot ?

Spring Boot est construit sur le framework Spring et contient toutes les fonctionnalités de Spring. Et il devient le favori des développeurs de nos jours en raison de son environnement de production rapide qui permet aux développeurs de se concentrer directement sur la logique au lieu de lutter avec la configuration et la mise en place.

Spring Boot est un framework basé sur les microservices et créer une application prête pour la production prend très peu de temps. Voici quelques-unes des fonctionnalités de Spring Boot :

Cela permet d'éviter la lourde configuration XML présente dans Spring.

- Il permet une maintenance et une création faciles des points de terminaison REST.
- Il inclut un serveur Tomcat intégré.
- Le déploiement est simple, les fichiers war et jar peuvent être facilement déployés sur le serveur Tomcat.

Spring Data-Java Persistence

Contexte

On suppose les deux entités suivantes :

Département	Employé
<pre>@Entity public class Departement { @Id @GeneratedValue(strategy=GenerationType.IDENTITY) private Long id; private String nom; // ... autres attributs et getters/setters }</pre>	<pre>@Entity public class Employe { @Id @GeneratedValue(strategy=GenerationType.IDENTITY) private Long id; private String nom; // ... autres attributs et getters/setters }</pre>

Relation One-to-One (Unidirectionnelle)

Un employé a un et un seul département (le département "possède" l'employé).

```
@Entity  
public class Employe {  
    // ...  
    // CascadeType.ALL pour propager les opérations (persist, remove)  
    @OneToOne(cascade = CascadeType.ALL)  
        // Spécifie la colonne de clé étrangère dans la table Employe private Departement departement;  
        @JoinColumn(name = "departement_id")  
    // ... }
```

Relation One-to-One (Bidirectionnelle)

Un employé a un département, et un département a un employé (les deux entités se connaissent).

```
@Entity [REDACTED]
public class Employe {
    // ...
    @OneToOne(mappedBy = "employe", cascade = CascadeType.ALL) // "mappedBy" pour la bidirectionnalité
    private Departement departement;
    // ...
}

@Entity [REDACTED]
public class Departement {
    // ...
    @OneToOne(mappedBy = "departement") // "mappedBy" indique que l'entité Employe est propriétaire de la relation
    private Employe employe;
    // ...
}
```

Relation One-to-Many (Bidirectionnelle)

Un département a plusieurs employés, et un employé appartient à un département.

```
@Entity [REDACTED]
public class Departement {
    // ...
    @OneToMany(mappedBy = "departement", cascade = CascadeType.ALL)
    private List<Employe> employes; // Utilisez une collection (List, Set)
    // ...
}

@Entity [REDACTED]
public class Employe {
    // ...
    @ManyToOne
    @JoinColumn(name = "departement_id")
    private Departement departement;
    // ...
}
```

Relation Many-to-Many (Bidirectionnelle)

Plusieurs employés peuvent travailler dans plusieurs départements (et vice-versa). C'est souvent modélisé avec une table de jointure.

```
@Entity [REDACTED]
public class Employe {
    // ...
    @ManyToMany(mappedBy = "employes")
    private List departements;
    // ...
}

@Entity [REDACTED]
public class Departement {
    // ...
    @ManyToMany
    @JoinTable(
        name = "employe_departement", // Nom de la table de jointure
        joinColumns = @JoinColumn(name = "employe_id"), // Colonne(s) de clé étrangère pour Employe
        inverseJoinColumns = @JoinColumn(name = "departement_id") // Colonne(s) de clé étrangère pour Departement
    )
    private List<Employe> employes;
    // ...
}
```

Relation Many-to-Many (Bidirectionnelle) avec une Entité de Jointure

Parfois, la table de jointure a des attributs propres (par exemple, la date de début d'un employé dans un département). Dans ce cas, on crée une entité pour la jointure.

Etape1: Entité Employe

```
@Entity
public class Employe {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "employe", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<DepartmentEmployee> departmentEmployees = new HashSet<>();

    //...
}
```

Etape2: Entité Departement

```
@Entity
public class Employe {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String nom;

    @OneToMany(mappedBy = "employe", cascade = CascadeType.ALL, orphanRemoval = true)
    private Set<DepartmentEmployee> departmentEmployees = new HashSet<>();
    //...
}
```

Etape3: Entité Intermédiaire DepartmentEmployee

```
@Entity
public class DepartmentEmployee {

    @EmbeddedId
    private DepartmentEmployeeId id;

    @ManyToOne
    @MapsId("employeId")
    @JoinColumn(name = "employe_id")
    private Employe employe;

    @ManyToOne
    @MapsId("departementId")
    @JoinColumn(name = "departement_id")
    private Departement departement;
}
```

Le design pattern Jparepository

JpaRepository

- JpaRepository est une extension spécifique de Repository pour JPA (Java Persistence API).
- Il contient l'API complète de CrudRepository et PagingAndSortingRepository.
- Donc, il contient l'API pour les opérations CRUD de base ainsi que l'API pour la pagination et le tri.

Syntaxe :

```
public interface JpaRepository<T, ID>  
extends PagingAndSortingRepository<T, ID>, QueryByExampleExecutor<T>
```

Avec :

T : Type de domaine que le dépôt gère (Généralement le nom de la classe Entité/Modèle)

ID : Type de l'identifiant de l'entité que le dépôt gère (Généralement la classe d'enveloppement de votre @Id qui est créée à l'intérieur de la classe Entité/Modèle)

Les méthodes hérités de Crudrepository-1-

save(S entity) : Sauvegarde une entité. Si l'entité existe déjà, elle est mise à jour.

saveAll(Iterable<S> entities) : Sauvegarde une collection d'entités.

findById(ID id) : Retourne une entité par son identifiant.

existsById(ID id) : Vérifie si une entité existe par son identifiant.

findAll() : Retourne toutes les entités.

findAllById(Iterable<ID> ids) : Retourne une collection d'entités par leurs identifiants.

count() : Retourne le nombre total d'entités.

Les méthodes hérités de Crudrepository-2-

deleteById(ID id) : Supprime une entité par son identifiant.

delete(S entity) : Supprime une entité.

deleteAllById(Iterable<? extends ID> ids) : Supprime une collection d'entités par leurs identifiants.

deleteAll(Iterable<? extends S> entities) : Supprime une collection d'entités.

deleteAll() : Supprime toutes les entités.

Méthodes héritées de PagingAndSortingRepository :

findAll(Pageable pageable) : Retourne une page d'entités en fonction des paramètres de pagination.

findAll(Sort sort) : Retourne toutes les entités triées selon les critères spécifiés.

Méthodes spécifiques à JpaRepository

flush() : Force la synchronisation des modifications en attente avec la base de données.

saveAndFlush(S entity) : Sauvegarde une entité et force la synchronisation immédiate avec la base de données.

deleteInBatch(Iterable<S> entities) : Supprime une collection d'entités en une seule requête batch.

deleteAllInBatch() : Supprime toutes les entités en une seule requête batch.

getOne(ID id) : Retourne une référence à l'entité par son identifiant (chargement paresseux).

findAll(Specification<S> spec) : Retourne toutes les entités correspondant à une spécification donnée.

findAll(Specification<S> spec, Pageable pageable) : Retourne une page d'entités correspondant à une spécification donnée.

findAll(Specification<S> spec, Sort sort) : Retourne toutes les entités correspondant à une spécification donnée, triées selon les critères spécifiés.

Exemple-Step1 : Créer l'entité

```
@Entity
```

```
public class Departement {
```

```
    @Id
```

```
    @GeneratedValue(strategy = GenerationType.IDENTITY)
```

```
    private Long id;
```

```
    private String nom;...
```

```
}
```

Step2 : Créer DepartementRepository

```
package com.example.demo.repository;  
import com.example.demo.entity.Departement;  
import org.springframework.data.jpa.repository.JpaRepository;  
import org.springframework.stereotype.Repository;  
  
@Repository  
public interface DepartementRepository extends  
JpaRepository<Departement, Long> {  
}
```

Step 3 : La couche Service

Méthodes de CrudRepository -1-

***** Sauvegarde un département**

```
public Departement save(Departement departement) {  
    return departementRepository.save(departement);  
}
```

***** Sauvegarde une liste de départements**

```
public List<Departement> saveAll(Iterable<Departement> departements) {  
    return departementRepository.saveAll(departements);  
}
```

Step 3 : La couche Service

Méthodes de CrudRepository -2-

***** Récupère un département par son ID.**

```
public Optional<Departement> findById(Long id) {  
    return departementRepository.findById(id);  
}
```

***** Vérifie si un département existe par son ID.**

```
public boolean existsById(Long id) {  
    return departementRepository.existsById(id);  
}
```

Step 3 : La couche Service

Méthodes de CrudRepository -3-

***** Récupère tous les départements.**

```
public List<Departement> findAll() {  
    return departementRepository.findAll();  
}
```

***** Récupère une liste de départements par leurs IDs.**

```
public List<Departement> findAllById(Iterable<Long> ids) {  
    return departementRepository.findAllById(ids);  
}
```

Step 3 : La couche Service

Méthodes de CrudRepository -4-

***** Compte le nombre total de départements.**

```
public long count() {  
    return departementRepository.count();  
}
```

**** * Supprime un département par son ID.**

```
public void deleteById(Long id) {  
    departementRepository.deleteById(id);  
}
```

Step 3 : La couche Service

Méthodes de CrudRepository -4-

***** Supprime un département après la récupération de l'ID**

```
public void delete(Departement  
departement) {  
  
departementRepository.delete(de  
partement);  
  
}
```

**** * Supprime une liste de départements.**

```
public void  
deleteAll(Iterable<Departement>  
departements) {  
  
departementRepository.deleteAll(  
departements);  
}
```

***** Supprime tous les départements.**

```
public void deleteAll() {  
  
departementRepository.deleteAll(  
);  
}
```

Step 3 : La couche Service

Méthodes de PagingAndSortingRepository -5-

***** Récupère tous les départements triés.**

```
public List<Departement> findAll(Sort sort) {  
    return departementRepository.findAll(sort);  
}
```

**** Supprime un département après la récupération de l'ID**

```
public void delete(Departement departement) {  
    departementRepository.delete(departement);  
}
```

Step 3 : La couche Service

Méthodes de JpaRepository -6-

***** Sauvegarde un département et force la synchronisation avec la base de données**

```
public Departement saveAndFlush(Departement departement) {  
    return departementRepository.saveAndFlush(departement);  
}
```

**** * Force la synchronisation des modifications en attente avec la base de données.**

```
public void flush() {  
    departementRepository.flush();  
}
```

Step 3 : La couche Service

Méthodes de JpaRepository -7-

***** Supprime une liste de départements en une seule requête batch.**

```
public void deleteAllInBatch(Iterable<Departement> departements) {  
    departementRepository.deleteAllInBatch(departements);  
}
```

***** Supprime tous les départements en une seule requête batch.**

```
public void deleteAllInBatch() {  
    departementRepository.deleteAllInBatch();  
}
```

Step 3 : La couche Service

Méthodes de JpaRepository -8-

Créer une classe de spécification pour filtrer les départements par nom :

```
public class DepartementSpecification {  
    //Spécification pour filtrer les départements par nom.  
    public static Specification<Departement> nomContains(String nom) {  
        return (root, query, criteriaBuilder) -> {  
            if (nom == null || nom.isEmpty()) {  
                return criteriaBuilder.conjunction(); // Aucun filtre si le nom est vide  
            }  
            return criteriaBuilder.like(criteriaBuilder.lower(root.get("nom")), "%" + nom.toLowerCase() + "%");  
        };}  
};}
```

Step 3 : La couche Service

Méthodes de JpaRepository -9-

```
@Service
public class DepartementService {

    @Autowired
    private DepartementRepository departementRepository;

    /**
     * Retourne toutes les entités correspondant à une spécification donnée.
     */
    public List<Departement> findAll(Specification<Departement> spec) {
        return departementRepository.findAll(spec);
    }

    /**
     * Retourne une page d'entités correspondant à une spécification donnée.
     */
    public Page<Departement> findAll(Specification<Departement> spec, Pageable pageable) {
        return departementRepository.findAll(spec, pageable);
    }

    /**
     * Retourne toutes les entités correspondant à une spécification donnée, triées selon les critères spécifiés
     */
    public List<Departement> findAll(Specification<Departement> spec, Sort sort) {
        return departementRepository.findAll(spec, sort);
    }
}
```

Step 3 : La couche Service

Méthodes de JpaRepository -10-

comment exposer ces méthodes dans un contrôleur REST ?

```
*** Recherche des départements avec une spécification
@GetMapping("/search")
public List<Departement> searchDepartements(@RequestParam(required = false) String nom) {
    Specification<Departement> spec = DepartementSpecification.nomContains(nom);
    return departementService.findAll(spec);
}

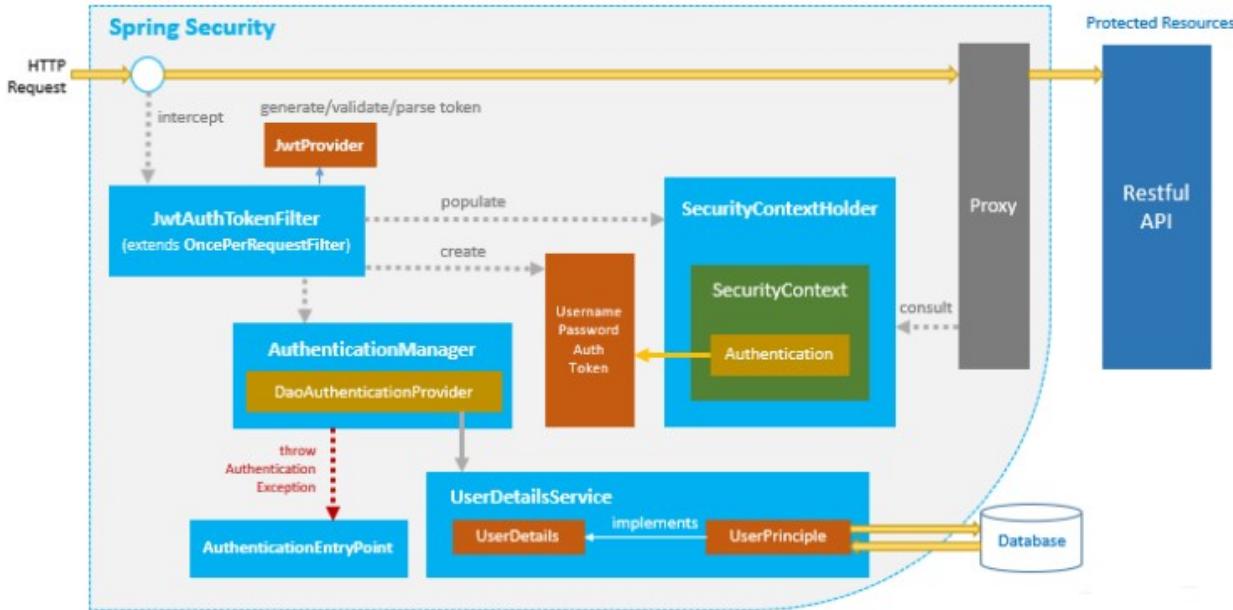
*** Recherche paginée des départements avec une spécification
@GetMapping("/search/paged")
public Page<Departement> searchDepartementsPaged(
    @RequestParam(required = false) String nom,
    Pageable pageable) {
    Specification<Departement> spec = DepartementSpecification.nomContains(nom);
    return departementService.findAll(spec, pageable);
}

*** Recherche triée des départements avec une spécification.
@GetMapping("/search/sorted")
public List<Departement> searchDepartementsSorted(
    @RequestParam(required = false) String nom,
    Sort sort) {
    Specification<Departement> spec = DepartementSpecification.nomContains(nom);
    return departementService.findAll(spec, sort);
}
```

Partie2: Spring Sécurity

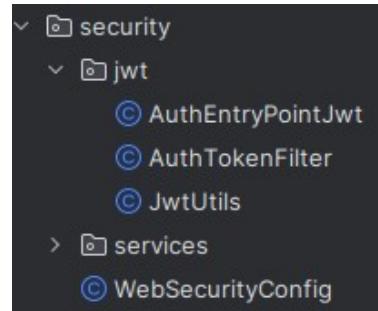
Objectifs

- Assurer une authentification forte
- Gestion des rôles utilisateurs



Étape 1 : Authentification-1-

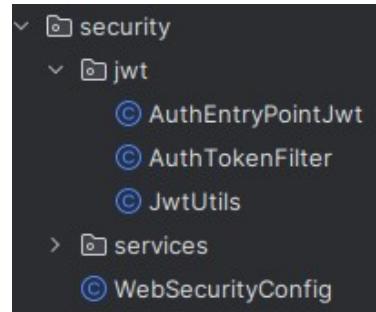
Créer une classe **JwtUtils** pour :



- Créer des tokens JWT pour authentifier des utilisateurs.
- Extraire des informations de ces tokens.
- Vérifier la validité de ces tokens.

Étape 1 : Authentification-2-

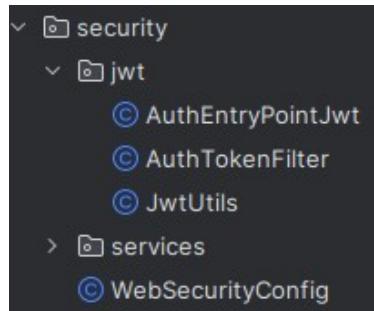
Créer un filtre JWT pour l'authentification à travers la classe AuthTokenFilter



- Ce filtre interceptera les requêtes HTTP.
- Extraira le token JWT du header Authorization, et, s'il est valide, configurera le contexte de sécurité.

Étape 1 : Authentification-3-

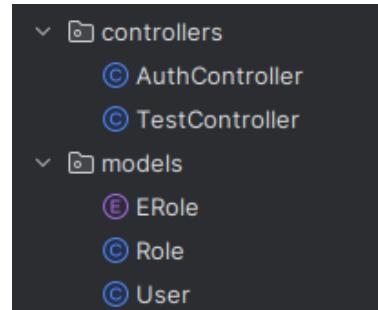
Configurer Spring Security pour utiliser le filtre JWT avec la classe **WebSecurityConfig**



Il faut maintenant intégrer le JwtRequestFilter dans la configuration de Spring Security afin que toutes les requêtes passent par ce filtre.

Étape 1 : Authentification-4-

Créer un endpoint pour l'authentification et la génération du token dans **AuthController**



Il est courant d'exposer une URL (par exemple /authenticate) qui reçoit les identifiants d'un utilisateur, les authentifie et renvoie un token JWT.

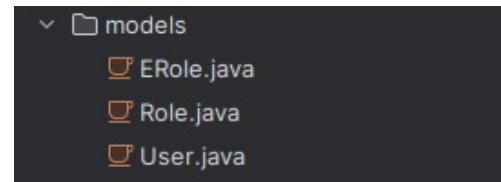
Étape 2 : Rôles utilisateurs-1-

Définir les rôles et l'implémentation de UserDetails :

Il vous faut représenter les rôles dans votre application.

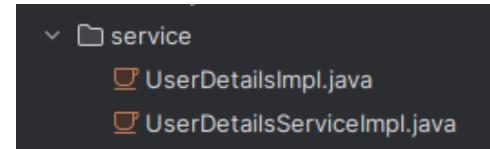
Par exemple, vous pouvez utiliser une énumération pour les rôles et une classe utilisateur qui contiendra une collection de rôles.

Ensuite, vous créerez une classe personnalisée qui implémente UserDetails afin d'intégrer ces rôles dans Spring Security.



Étape 2 : Rôles utilisateurs-2-

Implémenter UserDetails dans une classe personnalisée



Cette classe permettra à Spring Security de récupérer les informations (notamment les rôles) de l'utilisateur.

Remarque : Dans un projet réel, vous utiliserez probablement une entité JPA pour représenter l'utilisateur et stocker les rôles dans une table associée.

Étape 2 : Rôles utilisateurs-3-

Avant de générer le token, vous devez récupérer les rôles (ou autorités) de l'utilisateur. La manière de faire dépend de votre système d'authentification.

Typiquement, vous aurez un service qui récupère ces informations depuis une base de données ou un autre système.

Modifier la classe utilitaire JWT pour inclure les rôles

Pour que le token JWT contienne également les informations de rôles, nous allons modifier la méthode de génération du token afin d'ajouter ces informations dans les claims.

Étape 2 : Rôles utilisateurs-4-

Mettre à jour le contrôleur d'authentification

Il faut maintenant utiliser la nouvelle méthode de génération du token qui prend en compte les rôles.

Exemple de mise à jour du contrôleur :

```
final UserDetails userDetails =  
userDetailsService.loadUserByUsername(authRequest.getUsername());
```

```
final String jwt = jwtUtil.generateToken(userDetails);
```

Étape 2 : Rôles utilisateurs-5-

Restreindre l'accès aux endpoints en fonction des rôles

Dans la configuration de Spring Security, vous pouvez définir quelles URL sont accessibles à quels rôles.

```
@Bean
public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
    http.csrf().disable()
        // Définition des règles d'accès
        .authorizeRequests()
            .antMatchers("/authenticate").permitAll()
            // Endpoints réservés aux ADMIN
            .antMatchers("/admin/**").hasRole("ADMIN")
            // Endpoints réservés aux MODERATEURS et ADMIN
            .antMatchers("/moderateur/**").hasAnyRole("MODERATEUR", "ADMIN")
            // Endpoints réservés aux utilisateurs authentifiés (USER, MODERATEUR ou ADMIN)
            .antMatchers("/user/**").hasAnyRole("USER", "MODERATEUR", "ADMIN")
            .anyRequest().authenticated()
        .and()
        // La gestion des sessions est désactivée (stateless)
        .sessionManagement().sessionCreationPolicy(SessionCreationPolicy.STATELESS);

    // Ajout du filtre JWT avant le filtre standard d'authentification
    http.addFilterBefore(jwtRequestFilter, UsernamePasswordAuthenticationFilter.class);

    return http.build();
}
```

Étape 2 : Rôles utilisateurs-6-

Finalement les ressources filtrées par utilisateurs sont prêtes pour l'utilisation.

Exemple :

```
@RestController
public class UserController {

    @GetMapping("/user/hello")
    public String userHello() {
        return "Bonjour utilisateur!";
    }
}
```

Partie 3 : Développer la partie Front-end sous Angular

Nous allons utiliser l'api sans le jwt

Introduction à Angular

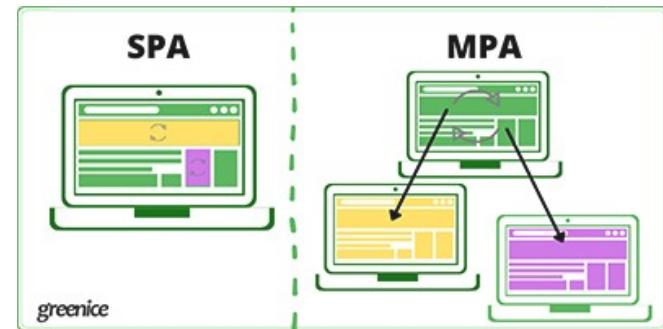
Dans ce cours, nous allons réaliser une application CRUD en Angular pour gérer des employés. L'application communiquera avec une API REST développée avec Spring Boot. Cette API expose les opérations suivantes :

- Création d'un employé (POST /api/employees)
- Lecture (liste complète ou par id) (GET /api/employees et GET /api/employees/{id})
- Mise à jour d'un employé (PUT /api/employees/{id})
- Suppression d'un employé (DELETE /api/employees/{id})

Angular ?

Un framework JavaScript open-source de haut niveau.

- Conçu pour développer des applications web monopages (SPA) riches et performantes.
- Développé par Google et est régulièrement mis à jour pour suivre les dernières tendances du développement web.
- AngularJS : La première version, a été lancée en 2010 et est basée sur **Javascript**.
- Face à l'évolution rapide des technologies web, Google a décidé de repenser complètement le framework et a lancé Angular 2 (aujourd'hui Angular) en 2016 et est basée sur **TypeScript**.



Angular -un Framework Populaire-

Caractéristique	Angular	React	Vue
Architecture	Très structurée	Composants	Flexible
Taille	Plus volumineux	Léger	Entre les deux
Communauté	Grande et active	Très grande et active	En forte croissance
Adoption	Largement utilisé dans les entreprises	Très populaire dans les startups	Populaire de jour en jour

Préparer l'environnement

1 - On utilisera le Node Package Manager, ou npm. Vous pouvez télécharger et installer la dernière version de Node sur <https://nodejs.org/fr/>, ce qui installera également npm.

2- Ensuite : utilisez cette commande pour l'installation : **npm install -g @angular/cli**

3- Création du projet :

Dans un terminal, exécutez : - ng new my-project-frontend

- cd my-project-frontend

Création du modèle de données

- Définir la structure d'un employé pour être utilisé dans l'application.
 - Faciliter la communication entre le service et les composants.
- => Créez un fichier employe.ts dans le dossier src/app :

```
// src/app/employe.ts
export interface Employe {
    id?: number; // Optionnel lors de la création
    name: string;
// Vous pouvez ajouter d'autres propriétés comme email, poste, etc.
}
```

Création du service Angular

- Centraliser l'accès à l'API.
- Utiliser le module HttpClient d'Angular pour les requêtes HTTP.

Étape 1 : Génération du service

Dans le terminal, générez le service tapez : ng generate service employe

Étape 2 : Implémentation du service

Modifiez le fichier employe.service.ts pour implémenter les méthodes CRUD :

```
export class EmployeService {  
    // URL de base de l'API Spring Boot  
    private baseUrl = 'http://localhost:8080/api/employes';  
    constructor(private http: HttpClient) { }  
}
```

```
// Récupérer tous les employés  
getEmployes(): Observable<Employe[]> {  
    return this.http.get<Employe[]>(`${this.baseUrl}`);  
}  
.....de même pour les autres ressources
```

Création des composants Angular-1-

a. Composant de la liste des employés

pour :

- Afficher tous les employés récupérés depuis l'API.
- Proposer des actions de suppression et de redirection vers la mise à jour.

Génération du composant :

ng generate component list-employees

```
export class ListEmployesComponent implements OnInit {  
  employes: Employe[] = [];  
  constructor(private employeService: EmployeService) { }  
  ngOnInit(): void {  
    this.getEmployes();  
  }  
  getEmployes(): void {  
    this.employeService.getEmployes().subscribe(data => {  
      this.employes = data;  
    });}  
}
```

Création des composants Angular-2-

b- Composant de création d'un employé

pour :

- Offrir un formulaire pour saisir les informations d'un nouvel employé.
- Envoyer les données saisies à l'API pour créer un nouvel enregistrement.

Génération du composant :

ng g c create-employe

```
export class CreateEmployeComponent implements OnInit {  
  employe: Employe = { name: '' };  
  constructor(private employeService: EmployeService, private router: Router) {}  
  ngOnInit(): void {}  
  saveEmploye(): void {  
    this.employeService.createEmploye(this.employe).subscribe(  
      data => {  
        console.log('Employé créé : ', data);  
        this.goToEmployeList();  
      },  
      error => console.error(error) );  
  }  
}
```

Création des composants Angular-3-

c-Composant de mise à jour d'un employé

- Récupérer les données d'un employé existant via son identifiant.
- Modifier les informations et les envoyer à l'API pour mise à jour.

Génération du composant :

ng g c update-employe

Configuration du routage

Définir des routes pour naviguer entre les différents composants dans **app.routing.ts**

Configurer les liens dans les composants pour une navigation fluide.

Création/Modification du module de routage

Ouvrez ou créez le fichier **app-routing.module.ts** et configurez les routes :

```
const routes: Routes = [
  { path: '', redirectTo: 'employes', pathMatch: 'full' },
  { path: 'employes', component: ListEmployesComponent },
  { path: 'create', component: CreateEmployeComponent },
  { path: 'update/:id', component: UpdateEmployeComponent }
];
```

Configuration du module principal

Importer les modules nécessaires : HttpClientModule pour les requêtes HTTP et FormsModule pour le binding des formulaires.

Déclarer les composants dans le module principal.

Ouvrez le fichier app.module.ts et assurez-vous que tout est bien configuré :

```
@NgModule({
  declarations: [
    AppComponent,
    ListEmployesComponent,
    CreateEmployeComponent,
    UpdateEmployeComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: []
})
```

Pistes d'amélioration-1-

Gestion des erreurs : Implémentez des messages d'erreur et une gestion plus fine des erreurs HTTP.

Validation de formulaires : Utilisez les formulaires réactifs (Reactive Forms) pour une validation plus robuste.

Styles et design : Intégrez des frameworks CSS comme Bootstrap ou Angular Material pour améliorer l'interface.

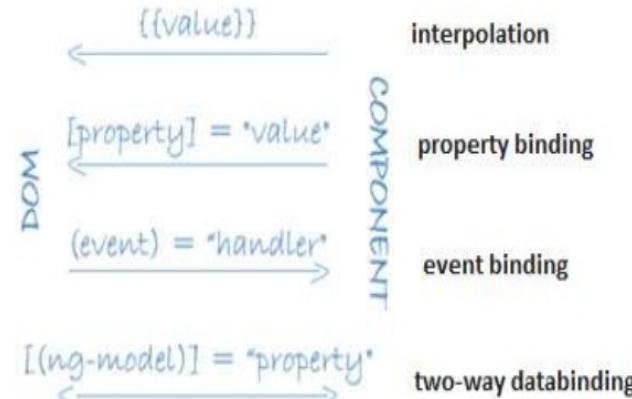
Sécurité : Ajoutez une gestion de l'authentification et autorisation pour sécuriser les endpoints.

Pistes d'amélioration-2-

1- Angular forms :

- Template-Driven Forms
- Reactive Forms

2- Data Binding



Pour aller plus loin-2-

Cycle de vie d'un Composant

Un composant passe par plusieurs phases depuis sa création jusqu'à sa destruction.

Angular maintient et suit ces différentes phases en utilisant des méthodes appelées « hooks ».

On peut alors à chaque phase implémenter une logique.

Ces méthodes se trouvent dans des interfaces dans la librairie « @angular/core »



Cycle de vie d'un Composant-2-

Méthode	Rôle
ngOnChanges	il est appelé lorsqu'un input est défini ou modifié de l'extérieur. L'état des modifications sur les inputs est fourni en paramètre
ngOnInit	il est appelé une seule fois et permet de réaliser l'initialisation du composant, qu'elle soit lourde ou asynchrone (on ne touche pas au constructeur pour ça)
ngDoCheck	il est appelé après chaque détection de changements
ngAfterContentInit	il est appelé une fois que le contenu externe est projeté dans le composant (transclusion)

Méthode	Rôle
ngAfterContentChecked	il est appelé chaque fois qu'une vérification du contenu externe (transclusion) est faite
ngAfterViewInit	il est appelé dès lors que la vue du composant ainsi que celle de ses enfants sont initialisés
ngAfterViewChecked	il est appelé après chaque vérification des vues du composant et des vues des composants enfants.
ngOnDestroy	il est appelé juste avant que le composant soit détruit par Angular. Il permet alors de réaliser le nettoyage adéquat de son composant. C'est ici qu'on veut se désabonner des Observables ainsi que des events handlers sur lesquels le composant s'est abonné.

Pour aller plus loin-3-

Interaction entre composants

1- Communication Parent → Enfant

Se fait via la liaison de propriété à l'aide du décorateur @Input.

2- Communication Enfant → Parent

Pour envoyer des données du composant enfant vers le composant parent, on utilise le décorateur @Output avec un objet EventEmitter.

3- Communication entre Composants Frères via un Service Partagé

Lorsque deux composants frères (ou éloignés dans l'arborescence) doivent communiquer, il est courant d'utiliser un service partagé avec un mécanisme réactif (par exemple, Subject ou BehaviorSubject de RxJS)

4- Utilisation de ViewChild pour accéder à un composant enfant

Le décorateur @ViewChild permet à un composant parent d'accéder à une instance de son composant enfant afin d'appeler directement ses méthodes ou accéder à ses propriétés.

Pour aller plus loin-4-

HTTP interceptor

Un HTTP Interceptor est un mécanisme d'Angular permettant d'intercepter et de modifier les requêtes HTTP sortantes ainsi que les réponses entrantes. Dans le contexte d'une authentification par JWT, il sert principalement à ajouter automatiquement le token dans l'en-tête (header) de chaque requête sécurisée.

- 1- Interception des requêtes : Avant qu'une requête HTTP ne soit envoyée, l'interceptor vérifie si un JWT est disponible (par exemple, dans un service d'authentification ou stocké localement).
- 2- Modification de la requête : Si le token existe, l'interceptor ajoute le JWT dans l'en-tête de la requête (souvent dans le champ Authorization avec le préfixe "Bearer").
- 3- Transmission et gestion des réponses : L'interceptor peut également surveiller les réponses pour détecter des erreurs liées à l'authentification ou pour renouveler le token en cas d'expiration.