Cours electif : Création d’application

Cours : API Java Spring Boot & Front Angular (CRUD complet)

1. Contexte : pourquoi une API Java Spring Boot et un front Angular ?

Avant de parler de code, il faut comprendre **dans quel monde tu développes** et **pourquoi cette stack existe**. Sinon, tu écris du code sans voir à quoi il sert dans un vrai projet.

1.1. Applications web modernes

Aujourd’hui, une grande partie des applications professionnelles sont :

- accessibles via un **navigateur web** (Chrome, Edge, Firefox…)
- utilisées sur des **réseaux** (internet, intranet, VPN…)
- partagées entre plusieurs équipes : front, back, ops, métier, etc.
- accessibles sur **PC, tablette, mobile**

L’ancienne approche (pages HTML générées côté serveur, type PHP/JSP) existe toujours, mais on voit de plus en plus :

- des **clients riches** dans le navigateur (SPA : Single Page Application)
- des **API** qui fournissent des données au format JSON

Stack typique :

- **Back-end** : Java + Spring Boot → expose des données via une API REST
- **Front-end** : Angular → affiche et manipule ces données dans le navigateur

---

1.2. Séparation Front / Back : pourquoi deux projets ?

On peut voir le système comme deux blocs indépendants :

1. Le front (Angular)

- Interface que l'utilisateur voit.
- Tourne dans le navigateur.
- Envoie des requêtes HTTP à une API.
- Gère navigation, formulaires, affichage, interactions.

2. Le back (Spring Boot)

- Tourne sur un serveur, une machine ou un conteneur Docker.
- Gère la logique métier : règles, validations, traitements.
- Communique avec la base de données.
- Expose une API standardisée (JSON sur HTTP).

Cette séparation permet :

- de **faire évoluer** l'interface sans toucher à la logique métier, et inversement ;
- de brancher plusieurs clients sur la même API :
  - application Angular
  - application mobile
  - autre service back-end ;
- de **spécialiser les équipes** (front / back) ;
- de **scaler indépendamment** :
  - plus d'instances front si beaucoup d'utilisateurs,
  - plus d'instances back si les traitements sont lourds.

1.3. Types d'applications concernées

Ce couple **API REST + front SPA** apparaît dans de nombreux contextes :

1. **Applications internes d'entreprise**
   - gestion de clients
   - back-office catalogue produit

- gestion de projets, RH, etc.

2. **Applications grand public (SaaS / B2C)**
   - plateforme de réservation
   - dashboard utilisateur pour un service en ligne
   - e-commerce structuré en SPA
3. **Applications hybrides web + mobile**
   - même API Spring Boot consommée par :
     - le front Angular (web)
     - une app mobile (Android / iOS)
4. **Intégrations externes / interopérabilité**
   - systèmes partenaires
   - scripts internes
   - processus d'intégration (ETL, automatisations)

1.4. L'API comme contrat entre systèmes

Quand tu développes une API, tu n'écris pas "juste du Java".

Tu définis un **contrat** :

- **Adresse** : URL (ex. `/api/books`)
- **Méthode HTTP** : GET, POST, PUT, DELETE…
- **Format d'entrée** : JSON envoyé par le client
- **Format de sortie** : JSON renvoyé par le serveur
- **Codes HTTP** : 200, 201, 400, 404, 500, etc.

Ce contrat doit être :

- **clair** (documenté),
- **stable** (pas de changement de champs au hasard),
- **cohérent** (noms, URL, structure).

Le front Angular (ou tout autre client) :

- ne sait pas comment est faite la base,

- ne sait pas quel ORM tu utilises,
- ne voit que l'API.

Dans ton TP, tu vas :

- concevoir une API propre côté Spring Boot,
- écrire un front Angular qui **consomme** cette API comme un client externe.

1.5. Environnements, équipes, cycle de vie

En entreprise, tu ne développes pas seul sur ton PC "en vrac". Il y a :

Environnements typiques

- `dev` : développement
- `test` / `recette` : validation technique et métier
- `prod` : production (vrais utilisateurs)

Souvent, chaque environnement a :

- sa propre base de données ;
- sa configuration (URL, clés, secrets…).

Équipes typiques

- développeurs back (Java/Spring)
- développeurs front (Angular)
- QA / testeurs
- DevOps (CI/CD, infra)

D'où l'importance d'avoir des **API bien définies**.

Cycle de vie simplifié d'une fonctionnalité

1. Le métier exprime un besoin : "Gérer un catalogue de livres".
2. L'équipe technique conçoit :
   - les entités métier (Book, Category, User, etc.)
   - les API (`GET /books` , `POST /books` , etc.)
   - les écrans front (liste, formulaire, détail).
3. Le back implémente et teste l'API.

4. Le front consomme l'API.
5. Tests techniques + métier.
6. Déploiement en production.

Ton TP simule ce cycle à petite échelle.

1.6. Positionnement du TP

Dans ce TP, tu seras dans un cas très fréquent :

- **Back** : petit monolithe Spring Boot exposant une API REST simple.
- **Front** : Angular qui consomme directement cette API.

Tu joueras le rôle :

- de développeur back (conception d'endpoints REST propres),
- de développeur front (consommation correcte de l'API),
- de développeur "full-stack pédagogique" avec la vision globale.

---

2. Java et Spring Boot

2.1. Java côté serveur

Java est l'un des langages les plus utilisés en **back-end** :

- **compilé** → performant,
- **typé** → robuste,
- **portable** ("write once, run anywhere"),
- **pérenne** → beaucoup de systèmes historiques,
- **industriel** → écosystème riche (tests, CI/CD, monitoring).

Historiquement, pour faire du web en Java, on utilisait :

- Java EE (servlets, JSP…),
- des serveurs d'application (GlassFish, WebLogic, JBoss).

Cette approche était souvent lourde à configurer et à déployer.

2.2. Spring Framework

Spring est arrivé pour :

- simplifier le développement Java,
- apporter l'**inversion de contrôle / injection de dépendances** (IoC/DI),
- structurer le code (couches, modules, etc.).

Mais il fallait encore beaucoup de configuration manuelle (XML, etc.).

2.3. Spring Boot : la simplification

Spring Boot (à partir de 2014) change la donne :

- **auto-configuration**,
- **serveur embarqué** (Tomcat, Jetty...),
- **démarrage rapide d'un projet web**,
- très peu de configuration initiale.

Exemple minimal d'application Spring Boot :

```
@SpringBootApplication
publicclassApp {
publicstaticvoidmain(String[] args) {
        SpringApplication.run(App.class, args);
    }
}
```

Plus besoin de déployer sur un serveur externe : il est embarqué.

Résultat : Spring Boot est devenu le standard pour exposer des APIs REST en Java.

2.4. Usages modernes de Spring Boot

Spring Boot permet de construire :

- des **APIs REST**,
- des **microservices**,
- des back monolithiques,
- des systèmes de messaging (Kafka, RabbitMQ...),

- des traitements batch,
- des services sécurisés (Spring Security),
- des services transactionnels avec des bases SQL ou NoSQL.

Dans ce cours, on se concentre sur : **API REST + Base de données + CRUD**.

---

3. API REST : principes et conventions

3.1. REST : définition

REST = **Representational State Transfer**.

C'est un **style architectural**, pas un framework ni un format.

Il repose sur plusieurs contraintes, dont :

1. **Client/Serveur**
2. **Stateless**
3. **Cache**
4. **Interface uniforme (Uniform Interface)**
5. **Système en couches (Layered System)**
6. **Code-on-demand** (optionnel)

3.2. Client / Serveur

Le client (Angular) ne connaît pas :

- la base de données,
- la logique interne,
- l'infrastructure.

Il consomme uniquement l'API via HTTP.

3.3. Stateless

Chaque requête HTTP doit contenir toutes les informations nécessaires.

Le serveur ne garde pas de "session métier" implicite entre deux requêtes REST.

3.4. Ressource et interface uniforme

Une **ressource** est un objet métier manipulable :

- Book
- User
- Product
- Task
- Invoice
- etc.

REST impose une manière uniforme d'adresser ces ressources (URL) et de les manipuler (méthodes HTTP).

Les détails des méthodes et codes HTTP sont vus dans la partie "Principes HTTP".

---

4. Conception d'une API REST dans Spring Boot (couches + CRUD + JPA)

Une API professionnelle en Spring Boot utilise en général 4 couches :

```
HTTP → Controller → Service → Repository → Database
```

4.1. Entity

Une entité représente une table SQL.

```
@Entity
publicclassBook {

@Id
@GeneratedValue
private Long id;

private String title;
private String author;
```

```
// getters/setters
}
```

4.2. Repository (Spring Data JPA)

Un repository est une interface d'accès aux données.

```
publicinterfaceBookRepositoryextendsJpaRepository<Book, Long> {
}
```

Grâce à `JpaRepository`, tu obtiens automatiquement :

```
findAll()
findById(id)
save(entity)
deleteById(id)
```

sans écrire de SQL.

4.3. Service (logique métier)

Le service encapsule la logique métier et fait appel au repository.

```
@Service
publicclassBookService {

privatefinal BookRepository repo;

publicBookService(BookRepository repo) {
this.repo = repo;
   }

public List<Book>findAll() {
return repo.findAll();
   }
```

```
public Bookcreate(Book b) {
return repo.save(b);
   }

public Bookupdate(Long id, Book b) {
       b.setId(id);
return repo.save(b);
   }

publicvoiddelete(Long id) {
       repo.deleteById(id);
   }
}
```

4.4. Controller (exposition REST)

Le controller expose le service via des endpoints HTTP :

```
@RestController
@RequestMapping("/api/books")
publicclassBookController {

privatefinal BookService service;

publicBookController(BookService service) {
this.service = service;
   }

@GetMapping
public List<Book>getAll() {
return service.findAll();
   }

@PostMapping
public ResponseEntity<Book>create(@RequestBody Book b) {
Bookcreated= service.create(b);
return ResponseEntity.status(HttpStatus.CREATED).body(created);
```

```
    }

@PutMapping("{id}")
public Bookupdate(@PathVariable Long id,@RequestBody Book b) {
return service.update(id, b);
    }

@DeleteMapping("{id}")
public ResponseEntity<Void>delete(@PathVariable Long id) {
        service.delete(id);
return ResponseEntity.noContent().build();
    }
}
```

4.5. CRUD, JPA et Spring Data

4.5.1. CRUD : les 4 opérations de base

CRUD décrit les 4 actions fondamentales sur une ressource persistée :

Lettre	Mot	Action
C	Create	créer une donnée
R	Read	lire une donnée
U	Update	modifier une donnée
D	Delete	supprimer une donnée

Correspondances :

CRUD → SQL

CRUD	SQL
Create	INSERT
Read	SELECT
Update	UPDATE
Delete	DELETE

CRUD → HTTP REST

CRUD	HTTP	Exemple
Create	POST	POST /books
Read	GET	GET /books
Update	PUT	PUT /books/7
Delete	DELETE	DELETE /books/7

CRUD → UI Angular

CRUD	Écran typique
Create	Formulaire “Ajouter”
Read	Liste + Détail
Update	Formulaire “Modifier”
Delete	Bouton “Supprimer”

La plupart des applications métiers sont des **CRUD enrichis**.

4.5.2. JPA : Java Persistence API

JPA est une **spécification** (et non une implémentation) qui décrit :

- comment mapper des objets Java vers des tables SQL,
- comment gérer la persistance,
- comment exécuter des requêtes,
- comment gérer le cycle de vie des entités.

Par exemple :

```
@Entity
publicclassBook {

@Id
private Long id;

private String title;
}
```

signifie : “`Book` représente une ligne dans une table SQL”.

Le **mapping objet–relationnel** est appelé **ORM** (Object Relational Mapping).

Les principales implémentations JPA :

- Hibernate (utilisée par défaut par Spring Boot),
- EclipseLink,
- OpenJPA.

4.5.3. Spring Data JPA

Spring Data JPA va plus loin en fournissant :

- des interfaces génériques,
- des méthodes prêtes à l'emploi,
- la gestion de la pagination, tri, etc.,
- l'intégration avec JPA/Hibernate.

Exemple :

```
publicinterfaceBookRepositoryextendsJpaRepository<Book, Long> {
}
```

→ tu obtiens tout le CRUD sans écrire de SQL.

Pourquoi c'est très utilisé ?

- rapidité de développement,
- robustesse,
- gestion des transactions,
- testabilité,
- indépendance vis-à-vis du SGBD,
- code plus propre et maintenable.

Pour ton TP :

1. tu fais du CRUD,
2. ton CRUD s'appuie sur JPA + Hibernate,
3. le repository fait le pont entre Java et SQL,

4. le controller expose ce CRUD via HTTP,
5. Angular consomme ce CRUD via HttpClient.

---

5. Principes HTTP

REST est une façon d'utiliser HTTP. Pour comprendre REST, il faut comprendre HTTP.

5.1. HTTP comme protocole

Un **protocole** est un ensemble de règles de communication.

HTTP (HyperText Transfer Protocol) est le protocole standard du Web.

Dans le cas d'une API :

- il devient **contractuel**,
- il porte des **sémantiques métier** (codes, méthodes),
- il structure l'intégration front/back.

5.2. Modèle client / serveur

Schéma :

```
Client → Requête → Serveur
Serveur → Réponse → Client
```

Dans ce cours :

- client = Angular ou Postman,
- serveur = Spring Boot.

5.3. Requête HTTP

Une requête contient :

1. **Méthode** (GET, POST, PUT, DELETE, PATCH...)
2. **URL**
3. **Headers** (métadonnées)

4. **Corps** (body, optionnel)

Exemple POST :

```
POST /api/books HTTP/1.1
Host: localhost:8080
Content-Type: application/json

{
  "title": "The Hobbit",
  "author": "Tolkien"
}
```

5.4. Réponse HTTP

Une réponse contient :

1. **Code de statut** (200, 201, 400, 404, etc.)
2. **Headers**
3. **Body** (optionnel)

Exemple :

```
HTTP/1.1 201 Created
Content-Type: application/json

{
  "id": 7,
  "title": "The Hobbit",
  "author": "Tolkien"
}
```

5.5. Méthodes HTTP et idempotence

- **GET** : lecture, pas de modification, pas de body. Idempotent.
- **POST** : création. Non idempotent.

- **PUT** : mise à jour / remplacement complet. Idempotent.
- **PATCH** : mise à jour partielle. Pas toujours idempotent.
- **DELETE** : suppression. Idempotent (supprimer 5 fois ne supprime qu'une fois).

5.6. Codes HTTP (principaux)

Classes :

- 1xx : information,
- 2xx : succès,
- 3xx : redirection,
- 4xx : erreur côté client,
- 5xx : erreur côté serveur.

Pour ton TP :

2xx — Succès

- 200 OK : succès générique.
- 201 Created : création réussie (POST).
- 204 No Content : suppression réussie (DELETE).

4xx — Erreur client

- 400 Bad Request : données invalides.
- 401 Unauthorized : non authentifié.
- 403 Forbidden : authentifié mais non autorisé.
- 404 Not Found : ressource inexistante.
- 409 Conflict : conflit (ex. doublon).

5xx — Erreur serveur

- 500 Internal Server Error : erreur interne.
- 503 Service Unavailable : service indisponible.

Une API professionnelle doit renvoyer ces codes correctement.

5.7. Headers HTTP importants

- `Content-Type` : format du corps (ex. `application/json`).
- `Accept` : formats acceptés par le client.
- `Authorization` : jeton d'authentification.
- `User-Agent` : identité du client.
- `Cache-Control` : politique de cache.

Typique pour une API REST JSON :

```
Accept: application/json
Content-Type: application/json
```

---

6. Postman : tester une API

6.1. Rôle de Postman

Postman permet :

- d'envoyer des requêtes HTTP,
- d'inspecter les réponses (codes, headers, body),
- de modifier les headers,
- de tester un CRUD complet,
- de simuler un client REST (avant le front),
- de créer des collections réutilisables,
- d'écrire des tests automatisés.

C'est un outil central pour tester un back avant de brancher un front.

6.2. Ordre de travail recommandé

Cycle typique :

1. Implémenter un endpoint Spring Boot.
2. L'appeler avec Postman.
3. Vérifier :

- code HTTP,
- JSON retourné,
- comportements d'erreur.

4. Corriger côté back si besoin.
5. Une fois stable → brancher Angular.

Ordre universel :

```
Back → Tests API → Front
```

6.3. Tester un CRUD via Postman

1. GET /api/books

→ Affiche la liste.

2. POST /api/books

Body JSON :

```
{
"title":"Dune",
"author":"Frank Herbert"
}
```

→ Attendu : `201 Created`.

3. PUT /api/books/7

Body :

```
{
"title":"Dune Messiah",
"author":"Frank Herbert"
}
```

→ Attendu : `200 OK`.

4. DELETE /api/books/7

→ Attendu : `204 No Content`.

6.4. Tester aussi les erreurs

- POST avec données manquantes → `400`
- GET sur un id qui n'existe pas → `404`
- DELETE sur un id inexistant → `404` ou `204` selon ta politique
- POST en doublon (si interdit) → `409`
- JSON mal formé → `400`

C'est là qu'on mesure la qualité d'une API.

---

7. Panorama des autres types d'API

REST n'est pas le seul style d'API.

7.1. SOAP

- **XML**, contrats stricts en **WSDL**.
- Très normé, utilisé en :
  - banques,
  - assurances,
  - télécoms,
  - administrations.

Avantages : forte standardisation, sécurité, interopérabilité.

Inconvénients : verbeux, tooling lourd, peu souple.

7.2. GraphQL

- Créé par Facebook.
- Le client spécifie les données dont il a besoin.
- Un seul endpoint.

Adapté à :

- interfaces très dynamiques,
- graphes de données complexes,
- contraintes fortes de bande passante (mobile).

7.3. gRPC

- Basé sur HTTP/2, messages binaires, Protocol Buffers.
- Très performant, typé.

Adapté à :

- microservices internes,
- communication service–service,
- scénarios haute performance/faible latence.

7.4. WebSocket

- Canal bidirectionnel et persistant entre client et serveur.
- Contrairement à HTTP classique (requête/réponse ponctuelle).

Utilisé pour :

- chat,
- collaboration temps réel,
- notifications live,
- trading,
- monitoring.

7.5. MQTT

- Protocole léger utilisé en **IoT**, pour des capteurs avec peu de ressources.

7.6. SSE (Server-Sent Events)

- Push unidirectionnel (serveur → client).
- Simpler qu'un WebSocket suivant le cas d'usage.

7.7. Choix rapide selon le besoin

Besoin	Solution courante
CRUD simple	REST
Mobile avec optim réseau	GraphQL
Microservices internes	gRPC
IoT	MQTT
Temps réel	WebSocket
Intégration bancaire / inter-entreprise	SOAP

Dans notre cas :

> afficher / modifier des données métiers via un front Angular
>
> → REST est le choix naturel.

---

8. Architectures d'applications et contexte du TP

8.1. Architecture monolithique

> Tout le système applicatif (modules, services, API) est déployé en un seul bloc.

Caractéristiques :

- un seul projet déployé,
- souvent une seule base de données,
- unifiez la logique métier,
- build et déploiement relativement simples.

Avantages :

- facile à développer,
- facile à tester,
- cohérence transactionnelle,
- idéal pour petite équipe ou MVP/TP.

Inconvénients :

- plus difficile à faire évoluer à très grande échelle,
- scaling global (pas granulaire).

8.2. Architecture microservices

> L'application est découpée en services indépendants (domaines métiers), chacun avec sa base et son cycle de vie.

Caractéristiques :

- découpage par domaine (bounded contexts),
- communication par APIs, gRPC, messaging.

Avantages :

- scalabilité fine,
- déploiement indépendant,
- isolation des pannes.

Inconvénients :

- beaucoup plus complexe,
- fort besoin de DevOps, d'observabilité,
- transactions distribuées plus compliquées.

8.3. Event-Driven, Hexagonale, CQRS (aperçu rapide)

- **Event-Driven** : Kafka, RabbitMQ… pour les flux asynchrones.
- **Hexagonale (Ports/Adapters)** : séparer cœur métier, infrastructure, adaptateurs.
- **CQRS** : séparer clairement lecture (Query) et écriture (Command).

8.4. Architecture choisie pour le TP

Le TP repose sur une **architecture monolithique REST** :

- tu débutes sur ces technologies,
- c'est le modèle le plus répandu pour les CRUD,
- rapide à mettre en place,

- idéal pour apprendre les bases.

Schéma global :

```
Angular (SPA)
      ↓ HTTP/JSON
Spring Boot (Monolithe REST)
      ↓
Base SQL
```

---

9. Angular et les applications web modernes

9.1. Du Web 1.0 aux SPA

Web 1.0 (années 90–2005)

- pages statiques,
- HTML généré côté serveur,
- peu ou pas d'interactivité.

Web 2.0 : Ajax + JS

- arrivée d'AJAX, jQuery,
- modification partielle de page, dynamique,
- mais structure souvent "spaghetti".

SPA (Single Page Application)

Une SPA est une application complète dans le navigateur :

- routage côté client,
- état côté client,
- composants réutilisables,
- appels HTTP,
- formulaires,
- UI réactive.

Modèle très utilisé dans les dashboards métiers, CRM, ERP, SaaS, outils internes.

9.2. Rôle d'Angular

Angular est un **framework complet** de SPA. Il apporte :

- système de composants,
- templating,
- routing,
- formulaires (template-driven et reactive),
- HttpClient,
- injection de dépendances,
- tests unitaires et E2E,
- CLI (génération de code),
- i18n, AOT, SSR/CSR,
- tooling et optimisation.

Il permet de réaliser un front de grande taille de manière homogène et maintenable.

9.3. Pourquoi Angular en entreprise ?

- **Standardisation** : tous les devs Angular écrivent du code similaire.
- **Écosystème intégré** : moins de choix d'outillage à faire.
- **Maintenance long terme** : adapté à des projets sur plusieurs années.
- **Architecture claire** : permet de structurer un gros UI.
- **TypeScript** : correspond bien à la culture des équipes back (Java, .NET).

---

10. TypeScript

10.1. Limites de JavaScript

JavaScript est :

- dynamique,

- non typé,
- permissif.

Exemple :

```
let x =5;
x ="cinq";// valide en JS
```

À grande échelle, cela génère de nombreux bugs.

10.2. Apports de TypeScript

TypeScript est un **superset de JavaScript** :

- tout code JS valide est du TS valide,
- TS ajoute le **typage statique**.

Exemple :

```
letx:number =5;
x ="cinq";// Erreur de compilation
```

Avantages :

- moins de bugs,
- meilleure auto-complétion,
- documentation implicite par les types,
- refactorings plus sûrs,
- meilleure collaboration en équipe.

10.3. Interfaces pour l'appel d'API

Définir un modèle :

```
exportinterfaceBook {
id:number;
title:string;
```

```
author:string;
}
```

permet à Angular de :

- typer les données provenant du back,
- détecter les incohérences JSON / modèle.

Exemple :

```
this.http.get<Book[]>('/api/books')
```

Sans TS, tu serais en “any” et tu ne saurais pas si le JSON correspond.

---

11. Consommer une API REST avec Angular

11.1. HttpClient

Angular fournit `HttpClient` pour faire des requêtes HTTP :

```
this.http.get<Book[]>('http://localhost:8080/api/books');
this.http.post<Book>('http://localhost:8080/api/books', book);
this.http.put<Book>('http://localhost:8080/api/books/7', book);
this.http.delete('http://localhost:8080/api/books/7');
```

11.2. Services Angular

Bonne pratique : ne pas mettre les appels HTTP dans les composants.

On crée un **service** :

```
@Injectable({providedIn:'root' })
exportclassBookService {
private api ='http://localhost:8080/api/books';

constructor(privatehttp:HttpClient) {}
```

```
getAll() {
returnthis.http.get<Book[]>(this.api);
  }

create(book:Book) {
returnthis.http.post<Book>(this.api, book);
  }

update(id:number,book:Book) {
returnthis.http.put<Book>(`${this.api}/${id}`, book);
  }

delete(id:number) {
returnthis.http.delete(`${this.api}/${id}`);
  }
}
```

Puis, dans un composant :

```
this.bookService.getAll().subscribe(books ⇒ {
this.books = books;
});
```

11.3. Asynchronisme et Observables

Les appels réseau sont asynchrones :

- tu ne sais pas quand la réponse arrive,
- tu dois gérer succès / erreur.

`HttpClient` renvoie des **Observables**.

Le composant s'abonne (`subscribe`) pour réagir :

```
this.bookService.getAll().subscribe({
next:books ⇒this.books = books,
error:err ⇒console.error(err)
```

```
});
```

On peut aussi utiliser le `async` pipe dans les templates.

11.4. Découplage front / back

- Angular ne sait pas comment fonctionne ta base.
- Spring ne sait pas ce qu'est ton HTML.

Ils ne communiquent qu'à travers :

- HTTP,
- JSON,
- URLs,
- codes HTTP.

Ce découplage est voulu → il améliore la flexibilité, la maintenabilité, la testabilité.

11.5. CORS (Cross-Origin Resource Sharing)

Quand le front (`http://localhost:4200`) appelle le back (`http://localhost:8080`), c'est un appel **cross-origin**.

CORS doit être configuré côté back, par exemple :

```
@CrossOrigin(origins = "http://localhost:4200")
@RestController
@RequestMapping("/api/books")
publicclassBookController {
// ...
}
```

Sans cela, le navigateur bloque les requêtes (erreur CORS).

---

12. TP : Application complète CRUD (Back + Front)

12.1. Contexte

Tu es développeur full-stack.

Un Product Owner te demande :

> “Nous voulons une application pour gérer un catalogue de livres.”

L’application doit permettre :

- d’ajouter un livre,
- de lister les livres,
- de modifier un livre,
- de supprimer un livre.

12.2. Modèle fonctionnel

Un **Livre** possède :

- `id` (généré par le système),
- `titre`,
- `auteur`,
- `année`,
- `catégorie`.

Fonctionnalités :

Fonction	Description
Create	ajouter un livre
Read	afficher la liste
Update	éditer un livre
Delete	supprimer un livre

12.3. Étapes du TP

Étape 1 — Back-end Spring Boot

1. Créer un projet Spring Boot (Spring Web, Spring Data JPA, H2).
2. Configurer la base H2 en mémoire.
3. Créer l’entité `Book`.

4. Créer le `BookRepository` (Spring Data JPA).
5. Créer le `BookService`.
6. Créer le `BookController` avec un CRUD complet.
7. Tester chaque endpoint dans Postman (cas succès + erreurs).
8. Soigner les codes HTTP.
9. (Bonus) Ajouter Swagger pour la doc.
10. (Bonus) Passer à PostgreSQL.

Condition de passage : le back doit être entièrement fonctionnel **avant** de démarrer l'Angular.

Étape 2 — Front-end Angular

1. Créer un projet Angular.
2. Créer le modèle TypeScript `Book`.
3. Créer le service `BookService` (HttpClient).
4. Brancher les appels vers l'API Spring Boot.
5. Créer un composant "liste des livres".
6. Créer un composant "formulaire livre" (ajout/édition).
7. Mettre en place le routing Angular :
    - `/books` (liste)
    - `/books/new` (création)
    - `/books/:id/edit` (édition)
8. Brancher les actions CRUD complètes.
9. Gérer minimalement les erreurs (messages utilisateur).

Étape 3 — Bonus possibles (non notés mais biens pour le CV)

- validation de formulaires (Angular Reactive Forms),
- utilisation d'Angular Material pour l'UI,
- recherche et filtrage,
- tri et pagination (Spring Data + Angular),

- authentification JWT,
- Docker Compose (API + DB + front),
- déploiement sur un serveur (nginx + Spring Boot).

12.4. Livrables (sur un git par langage que vous m'envoyez par mail pour le 30/01/26) max

- code complet back (Spring Boot),
- code complet front (Angular),
- instructions de lancement (README),
- un court document de synthèse (1 page) expliquant l'architecture.

12.5. Grille d'évaluation possible

Critère	Pondération
Fonctionnalité	40 %
Qualité du code	25 %
Qualité de l'API	15 %
UI/UX minimale	10 %
Documentation	10 %