المدرسة الوطنية للعلوم التطبيقية - فاس
ⵜⵉⵏⵎⵍ ⵜⴰⵏⴰⵎⵓⵔⵜ ⵏ ⵜⵙⴳⵎⵉⵔⵉⵏ ⵜⵉⵎⵙⵙⵓⵍⵉⵏ
Ecole Nationale des Sciences Appliquées de Fès

جامعة سيدي محمد بن عبد الله بفاس
ⵜⴰⵙⴷⴰⵡⵉⵜ ⵙⵉⴷⵉ ⵎⵓⵃⴰⵎⴻⴷ ⴱⵏ ⵄⴱⴷⵍⵍⴰⵀ ⴱⴼⴰⵙ
UNIVERSITÉ SIDI MOHAMED BEN ABDELLAH DE FES

TP3

Spring Boot-ToDo-App

Réalisé par : Errami Fadwa

ILIA 2, N : 18

Encadré par : Prof IDRISSI KHAMLICHI

I. Introduction :

Ce rapport présente le développement d'une application web de gestion des tâches, réalisée avec **Angular 20** pour le frontend, **Spring Boot** pour le backend et **PostgreSQL** pour la base de données. La gestion et la visualisation des données PostgreSQL sont facilitées via **pgAdmin4**, un outil graphique puissant permettant de créer, modifier et superviser les tables, les requêtes et les utilisateurs de la base.

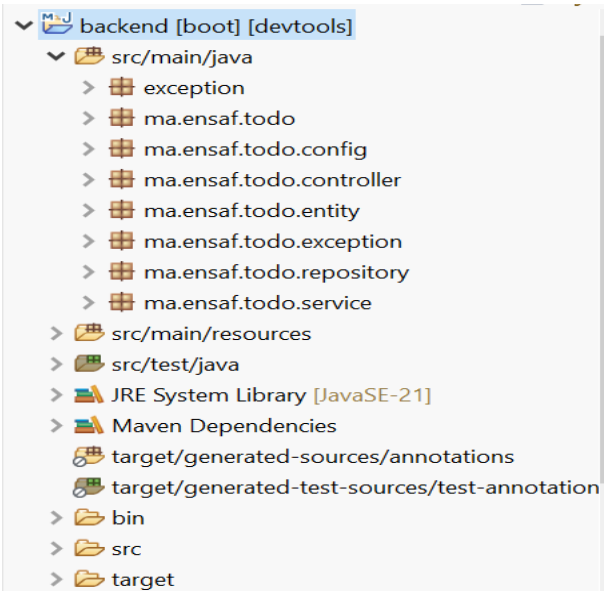
L'application permet de créer, modifier, supprimer et organiser des tâches selon leur priorité et leur statut (à faire ou terminée), avec une persistance fiable et une synchronisation complète entre le frontend et le backend.

Elle intègre les fonctionnalités modernes suivantes :

- **Angular Signal** pour une gestion réactive de l'état des tâches,
- **Angular Material** pour une interface ergonomique et responsive,
- **Angular CDK Drag & Drop** pour réorganiser les tâches facilement et intuitivement,
- Mode sombre automatique et manuel pour améliorer la lisibilité selon les préférences utilisateur,
- **Spring Boot REST API** pour la logique métier et la communication avec le frontend,
- **PostgreSQL** pour la gestion des données, garantissant fiabilité et persistance,
- **pgAdmin 4** pour la gestion graphique et la supervision de la base de données, facilitant l'administration et le débogage.

L'objectif du projet est de fournir un outil complet, performant et intuitif, capable de gérer efficacement les tâches quotidiennes tout en offrant une expérience utilisateur fluide, moderne et sécurisée.

II. Structure du projet Spring Boot



1. Détail des packages

Package	Contenu	Description
controller	TodoController	Gestion des requêtes HTTP (GET, POST, PUT, DELETE) vers l'API REST.
service	TodoService	Logique métier : traitement des tâches, validation, règles de gestion.
repository	TodoRepository	Interface JPA pour la persistance des tâches dans PostgreSQL.
model	Todo	Entité JPA représentant la table todos.

2. Todo.java

```java
package ma.ensaf.todo.entity;

import jakarta.persistence.*;

@Entity
@Table(name = "todos")
public class Todo {

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

    @NotBlank(message = "Le titre est obligatoire")
    private String title;

    private String description;

    private boolean completed = false;

    @Enumerated(EnumType.STRING)
    private Priority priority = Priority.MEDIUM;

    private LocalDateTime createdAt;

       public enum Priority {
        LOW, MEDIUM, HIGH
    }

    @PrePersist
    protected void onCreate() {
        createdAt = LocalDateTime.now();
    }

    public Todo() {}

    public Todo(Long id, String title, String description, boolean completed, Priority priority) {
        this.id = id;
        this.title = title;
        this.description = description;
        this.completed = completed;
        this.priority = priority;
    }
```

Mot-clé / Expression	Description
@Entity	Déclare que la classe est une entité JPA.
@Table(name = "todos")	Nom de la table en base de données.
@Id	Clé primaire de l'entité.
@GeneratedValue(strategy = GenerationType.IDENTITY)	Auto-incrémentation de l'ID.
@NotBlank(message = "...")	Validation : champ obligatoire.
@Enumerated(EnumType.STRING)	Enum stocké en base sous forme de chaîne.
@PrePersist	Méthode exécutée avant l'insertion (initialisation createdAt).
public Todo()	Constructeur par défaut.
public Todo(Long id, String title, String description, boolean completed, Priority priority)	Constructeur complet avec tous les champs.

3. TodoController :

Ce contrôleur expose les endpoints REST permettant de gérer les tâches : liste, création, mise à jour, suppression, filtrage et tri.Il délègue toute la logique métier au TodoService et gère uniquement les requêtes HTTP venant du frontend Angular.

```java
package ma.ensaf.todo.controller;

import java.util.List;

@RestController
@RequestMapping("/api/todos")
@CrossOrigin(origins = "http://localhost:4200", allowCredentials = "true")
public class TodoController {

    private final TodoService service;

    public TodoController(TodoService service) {
        this.service = service;
    }

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

    @GetMapping("/{id}")
    public Todo findById(@PathVariable Long id) {
        return service.findById(id);
    }

    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Todo create(@Valid @RequestBody Todo todo) {
        return service.create(todo);
    }

    @PutMapping("/{id}")
    public Todo update(@PathVariable Long id, @Valid @RequestBody Todo todo) {
        return service.update(id, todo);
    }

    @DeleteMapping("/{id}")
    @ResponseStatus(HttpStatus.NO_CONTENT)
    public void delete(@PathVariable Long id) {
        service.delete(id);
    }

    @GetMapping("/completed")
    public List<Todo> getCompletedTodos(@RequestParam boolean completed) {
        return service.findByCompleted(completed);
    }

    @GetMapping("/by-priority")
    public List<Todo> getTodosByPriority() {
        return service.findAllByPriority();
    }

    @PutMapping("/complete-all")
```

Mot-clé / Expression	Description
@RestController	Déclare une classe API REST.
@RequestMapping	Définit la route principale de l'API.
@CrossOrigin	Autorise les requêtes depuis un autre domaine (ex : Angular).
@PathVariable	Récupère une variable depuis l'URL.
@RequestBody	Récupère le JSON envoyé dans la requête.
@Valid	Active la validation des champs.
@ResponseStatus	Définit le code HTTP renvoyé.
TodoService	Couche métier qui contient la logique.
findAll()	Retourne toutes les tâches.
findById()	Retourne une tâche par ID.
getCompletedTodos()	Filtre les tâches par statut (complétées / non).
getTodosByPriority()	Retourne les tâches triées par priorité.
completeAll()	Marque toutes les tâches comme complétées

4. todo.service :

Ce service gère toute la logique métier des tâches (Todo) : création, mise à jour, suppression, filtrage et recherche. Il communique avec la base de données via TodoRepository

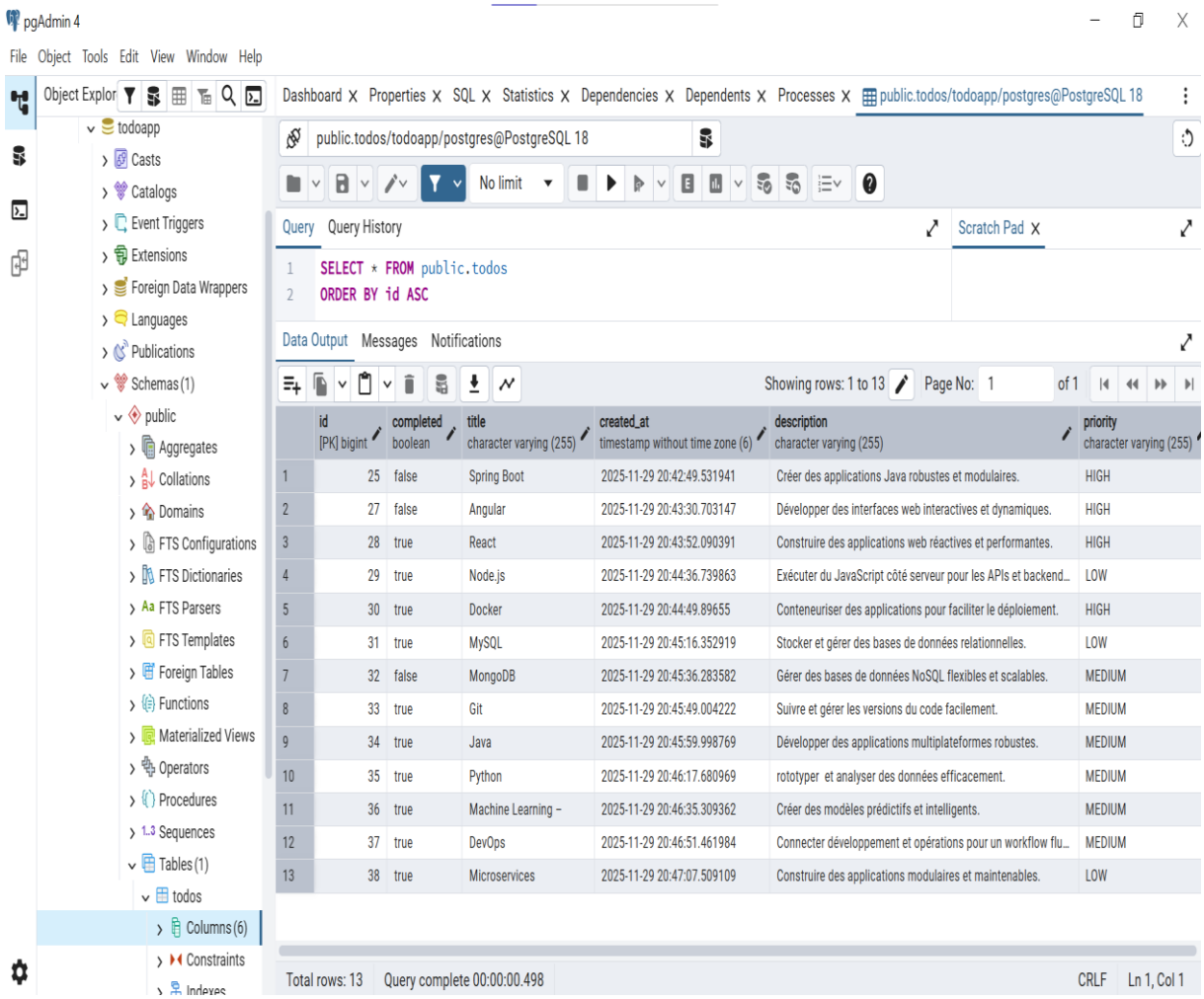
```java
package ma.ensaf.todo.service;

import java.util.List;

@Service
public class TodoService {

    private final TodoRepository repository;

    public TodoService(TodoRepository repository) {
        this.repository = repository;
    }

    public List<Todo> findAll() {
        return repository.findAll();
    }

    public Todo create(Todo todo) {
        return repository.save(todo);
    }

    public Todo update(Long id, Todo todo) {
        Todo existing = repository.findById(id)
                .orElseThrow(() -> new TodoNotFoundException(id));
        existing.setTitle(todo.getTitle());
        existing.setDescription(todo.getDescription());
        existing.setCompleted(todo.isCompleted());
        existing.setPriority(todo.getPriority());
        return repository.save(existing);
    }

    public void delete(Long id) {
        if (!repository.existsById(id)) throw new TodoNotFoundException(id);
        repository.deleteById(id);
    }

    public Todo findById(Long id) {
        return repository.findById(id)
                .orElseThrow(() -> new TodoNotFoundException(id));
    }

    public List<Todo> findByCompleted(boolean completed) {
        return repository.findByCompleted(completed);
    }

    public List<Todo> findAllByPriority() {
        return repository.findAllByOrderByPriorityDesc();
    }

    public List<Todo> completeAll() {
        List<Todo> todos = repository.findAll();
        todos.forEach(todo -> todo.setCompleted(true));
        return repository.saveAll(todos);
```

Mot-clé / Expression	Description courte
`@Service`	Indique une classe métier (logique applicative).
`TodoRepository`	Couche d'accès aux données (DAO).
`findAll()`	Retourne toutes les tâches depuis la base.
`create(todo)`	Ajoute une nouvelle tâche et l'enregistre.
`update(id, todo)`	Met à jour une tâche existante.
`repository.findById(id)`	Recherche une tâche par ID.
`orElseThrow()`	Lève une exception si la tâche n'existe pas.
`TodoNotFoundException`	Exception personnalisée pour les IDs inexistants.
`existsById(id)`	Vérifie si une tâche existe avant suppression.
`deleteById(id)`	Supprime une tâche.
`findByCompleted(completed)`	Filtre par statut complété / non complété.
`findAllByOrderByPriorityDesc()`	Récupère les tâches triées par priorité décroissante.
`completeAll()`	Marque toutes les tâches comme complétées.
`todos.forEach(...)`	Parcourt les todos pour modifier chaque élément.
`save(todo)`	Enregistre / met à jour une tâche.

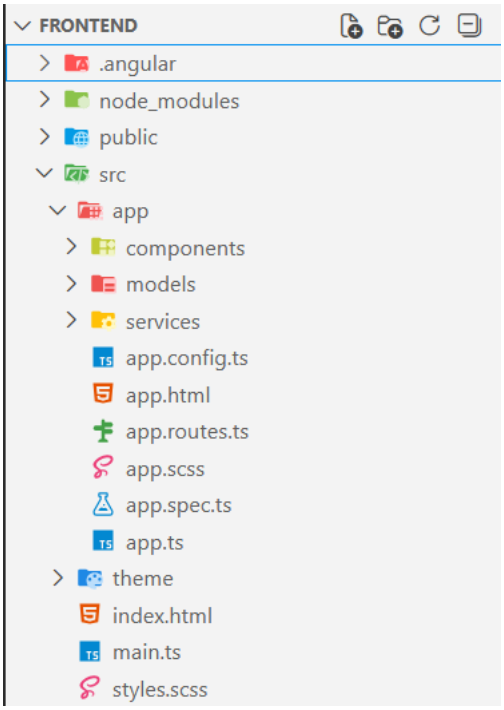
5. Application.properties :

Ce fichier configure l'application Spring Boot : son nom, le port du serveur et la connexion à la base de données **PostgreSQL**.Il active Hibernate (création/mise à jour des tables, affichage SQL formaté) et règle le niveau de logs pour faciliter le débogage.

```
spring.application.name=todo

server.port=8080

spring.datasource.url=jdbc:postgresql://localhost:5432/todoapp
spring.datasource.username=postgres
spring.datasource.password=1234
spring.datasource.driver-class-name=org.postgresql.Driver

spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

spring.sql.init.mode=never

# Logging Hibernate
logging.level.org.hibernate.type.descriptor.sql=trace
```



III. Structure du projet Angular :



L'application suit une architecture Component-Based avec une séparation claire des responsabilités :

- **Composant Principal (App)** : Gère l'UI et la logique de présentation
- **Service (TodoService)** : Gère la communication avec l'API backend
- **Modèle (Todo)** : Définit la structure des données

L'application utilise **Angular 20** , Angular **Material**, le **CDK** (Drag & Drop), avec un design moderne basé sur des variables CSS personnalisées.La communication avec le backend se fait via HttpClient, tandis que les **Signals** et la programmation réactive **RxJS** améliorent la gestion des états et des flux.

➢ Modèle Todo :

Cette interface définit la structure d'un objet Todo côté frontend avec ses champs : titre, description, priorité, statut et date de création.Elle permet au TypeScript d'assurer un typage strict et d'éviter les erreurs lors des appels API et de l'affichage dans Angular.

src > app > models > todo.model.ts > ...

```ts
export interface Todo {
  id?: number;
  title: string;
  completed: boolean;
  description?: string;
  priority?: 'LOW' | 'MEDIUM' | 'HIGH';
  createdAt?: Date;
}

```

1. Composant Principal (App.ts) :

Le composant App en Angular gère entièrement l'interface Todo, incluant l'affichage des tâches, l'ajout via un formulaire, la suppression et le basculement des tâches entre les listes "à faire" et "terminées".

Il utilise **le CDK Drag & Drop** pour permettre le déplacement interactif des tâches, en synchronisant automatiquement l'état completed avec le backend.

La réactivité est assurée par **les signals** d'Angular, qui stockent les listes de tâches, les champs du formulaire et l'état du thème sombre ou clair, tandis que les effects permettent de mettre à jour le DOM en fonction des changements de signaux.

La communication avec le serveur est gérée par le **TodoService** utilisant **HttpClient**, garantissant le **CRUD** complet, la récupération filtrée des tâches complétées, et le tri par priorité.

Le composant initialise également automatiquement **le thème** en fonction des préférences du système et offre la possibilité de basculer manuellement entre le mode clair et sombre. L'ensemble fournit une interface utilisateur réactive, moderne et fluide, tout en maintenant une cohérence avec la logique métier du backend Spring Boot.

➢ Système de Signals (State Management)

```ts
export class App {
  todos = signal<Todo[]>([]);
  isLoading = signal(true);
  isDark = signal(window.matchMedia('(prefers-color-scheme: dark)').matches);

  // Listes pour le drag and drop
  todo = signal<Todo[]>([]);
  done = signal<Todo[]>([]);

  // Champs pour le formulaire
  newTodoTitle = signal('');
  newTodoDescription = signal('');
  newTodoPriority = signal<'LOW' | 'MEDIUM' | 'HIGH'>('MEDIUM');
```

Tableau 1: Fonctions de Gestion d'État (Signals) :

Fonction	Type	Description	Valeur Initiale	Mise à Jour	Impact UI
todos()	Signal<Todo[]>	Liste complète de toutes les tâches	[]	Via set() ou update()	Actualise toutes les listes
isLoading()	Signal<boolean>	État de chargement des données	true	set(true/false)	Affiche/cache le skeleton loader
isDark()	Signal<boolean>	Thème sombre/clair	window.matchMedia()	set(!isDark())	Bascule toutes les variables CSS
todo()	Signal<Todo[]>	Tâches non complétées	[]	Filtrage de todos()	Liste "À faire" drag & drop
done()	Signal<Todo[]>	Tâches complétées	[]	Filtrage de todos()	Liste "Terminé" drag & drop
newTodoTitle()	Signal<string>	Titre de la nouvelle tâche	''	set(valeurInput)	Champ formulaire
newTodoDescription()	Signal<string>	Description nouvelle tâche	''	set(valeurInput)	Champ formulaire
newTodoPrioity()	Signal<'LOW'|'MEDIUM'|'HIGH'>	Priorité nouvelle tâche	'MEDIM'	set(valeurSelec)	Sélecteur formulaire

➢ Avantages de cette approche Signals :

- **Réactivité fine** : Seules les parties affectées sont mises à jour
- **Type Safety** : Typage TypeScript strict
- **Performance** : Évite les change detection cycles inutiles

➢ Cycle de Vie et Initialisation

```
constructor(private todoService: TodoService) {
  this.loadTodos();

  // Dark mode auto
  const media = window.matchMedia('(prefers-color-scheme: dark)');
  media.addEventListener('change', e => this.isDark.set(e.matches));
  effect(() => document.documentElement.dataset['theme'] = this.isDark() ? 'dark' : 'light');
}
```

Tableau 2: Méthodes de Cycle de Vie

Méthode	Appel	Paramètres	Actions	Effets
constructor()	Initialisation composant	TodoService	1. Appel loadTodos() 2. Configuration dark mode 3. Écouteur préférences système	Chargement données, thème appliqué
effect() (thème)	Réactivité Signal	Aucun	Met à jour data-theme sur documentElement	Bascule CSS variables light/dark
Media Query Listener	Changement préférence système	MediaQueryListEvent	isDark.set(e.matches)	Synchronisation thème OS/app

➢ **Points clés :**

- **Chargement immédiat** des données au constructeur
- **Écouteur système** pour le thème sombre
- **Effect réactif** qui met à jour les attributs CSS

Tableau 3: Méthodes de Gestion des Données :

Méthode	Déclencheur	Paramètres	Actions Internes	Appels API	Mise à jour State
loadTodos()	Constructor	Aucun	1. isLoading.set(true) 2. Subscribe API 3. Filtrage données	getTodos()	todos.set() todo.set() done.set() isLoading.set(false)
addTodo()	Click button / Enter	Aucun	1. Validation titre 2. Construction objet 3. Appel API 4. Reset form	addTodo(newTodo)	todos.update() todo.update() Reset signaux form
toggle(todo)	Click checkbox	Todo objet	1.Inverse completed 2. Appel API 3. Transfert entre listes	updateTodo()	todos.update() todo.update() done.update()
deleteTodo(id)	Click delete button	number id	1.Validation ID 2. Appel API 3. Suppression listes	deleteTodo(id)	todos.update() todo.update() done.update()

```
44
45  >  loadTodos() {···
56     }
57
58  >  addTodo() {···
76     }
77
78  >  toggle(todo: Todo) {···
94     }
95
96  >  deleteTodo(id?: number) {···
103    }
104
```

Tableau 4: Méthodes d'Interface Utilisateur :

Méthode	Événement	Paramètres	Comportement	Feedback Visuel
toggleTheme()	Click icon theme	Aucun	Inverse isDark() signal	Changement icône ⚙/🌙, bascule thème
drop(event)	Drag & Drop	CdkDragDrop<Todo[]>	1. Déplacement liste 2. Transfert liste 3. Mise à jour API	Animation drag, mise à jour compteurs

```
105 >   toggleTheme() {···
107     }
108
109 >   drop(event: CdkDragDrop<Todo[]>) {···
138     }
139 }
```

Tableau 5: Fonctions de Drag & Drop

Fonction	Contexte	Paramètres	Logique Conditionnelle	Actions
drop(event)	cdkDropListDropped	CdkDragDrop<Todo[]>	**Même liste**: event.previousContainer === event.container	moveItemInArray()
			Listes différentes: else	transferArrayItem() + updateTodo() API
Post-drop	Après transfert	movedTodo, completed	**Liste destination**: event.container.id	Mise à jour completed status

1. Service TodoService - Communication Backend

Ce service Angular communique avec ton backend Spring Boot via HttpClient pour récupérer, créer, modifier et supprimer des tâches (Todos).Toutes les méthodes renvoient des **Observables** afin d'assurer un fonctionnement réactif et asynchrone dans l'application Angular.

src > app > services > todo.service.ts > ...

```typescript
import { HttpClient } from '@angular/common/http';
import { inject, Injectable } from '@angular/core';
import { Observable } from 'rxjs';
import { Todo } from '../models/todo.model';

@Injectable({
  providedIn: 'root',
})
export class TodoService {
  private http = inject(HttpClient);
  private apiUrl = 'http://localhost:8080/api/todos';

  // constructor(private http: HttpClient) { }

  getTodos(): Observable<Todo[]> {
    return this.http.get<Todo[]>(this.apiUrl);
  }

  addTodo(todo: Todo): Observable<Todo> {
    return this.http.post<Todo>(this.apiUrl, todo);
  }

  updateTodo(todo: Todo): Observable<Todo> {
    return this.http.put<Todo>(`${this.apiUrl}/${todo.id}`, todo);
  }

  deleteTodo(id: number): Observable<void> {
    return this.http.delete<void>(`${this.apiUrl}/${id}`);
  }

}
```

Méthode	HTTP Method	Endpoint	Paramètres	Body	Retour
getTodos()	GET	/api/todos	Aucun	Aucun	Observable<Todo[]>
addTodo(todo)	POST	/api/todos	Todo (sans id)	Todo object	Observable<Todo> (avec id)
updateTodo(todo)	PUT	/api/todos/{id}	Todo (avec id)	Todo object	Observable<Todo>
deleteTodo(id)	DELETE	/api/todos/{id}	number id	Aucun	Observable<void>

Résultat final :