Travaux Diriges

Requetes HTTP avec Vue.js 3

GET - POST - DELETE - PATCH

Ibrahim ALAME – Formation Vue.js 2025

Objectifs du TD

Duree : 2h30

A la fin de ce TD, vous serez capable de :

- Effectuer des requetes HTTP avec `fetch`
- Recuperer des donnees depuis une API (GET)
- Envoyer des donnees pour creer une ressource (POST)
- Modifier une ressource existante (PATCH)
- Supprimer une ressource (DELETE)
- Gerer les etats de chargement et d'erreur

API utilisee : `https://jsonplaceholder.typicode.com/users`

Table des matières

1 Preparation du projet

Mise en place (10 min)

1. Creez un nouveau projet Vue.js :

   ```
   npm create vue@latest td-http
   ```

   Selectionnez : TypeScript = **Yes**, autres options = No

2. Installez et lancez :

   ```
   cd td-http
   npm install
   npm run dev
   ```

3. Remplacez le contenu de `src/App.vue` par le code de demarrage ci-dessous.

1.1 Code de demarrage

src/App.vue

```vue
<script setup lang="ts">
import { ref, computed, onMounted } from 'vue'

// URL de l'API
const API_URL = 'https://jsonplaceholder.typicode.com/users'

// TODO: Definir l'interface User (id, name, email)


// TODO: Declarer les variables reactives pour la liste


// TODO: Implementer les fonctions dans les exercices suivants

</script>

<template>
  <div class="app">
    <h1>Gestion des Utilisateurs</h1>

    <!-- Le contenu sera ajoute au fur et a mesure -->

  </div>
</template>

<style>
.app {
  max-width: 800px;
```

```
  margin: 0 auto;
  padding: 20px;
  font-family: Arial, sans-serif;
}
h2 {
  margin-top: 30px;
  color: #42B883;
  border-bottom: 2px solid #42B883;
  padding-bottom: 5px;
}
input {
  padding: 8px;
  margin-right: 10px;
  border: 1px solid #ddd;
  border-radius: 4px;
}
button {
  padding: 8px 16px;
  margin: 2px;
  cursor: pointer;
  border: none;
  border-radius: 4px;
  background: #42B883;
  color: white;
}
button:hover { background: #35495E; }
button:disabled { background: #ccc; cursor: not-allowed; }
ul { list-style: none; padding: 0; }
li {
  padding: 15px;
  border-bottom: 1px solid #eee;
  display: flex;
  justify-content: space-between;
  align-items: center;
}
.error { color: red; padding: 10px; background: #fee; border-radius
    : 4px; }
.loading { color: gray; font-style: italic; }
.btn-danger { background: #e74c3c; }
.btn-secondary { background: #95a5a6; }
</style>
```

2 Exercice 1 : GET - Recuperer les utilisateurs

Exercice 1 (30 min)

Objectif : Charger et afficher la liste des utilisateurs depuis l'API.

Fonctionnalites a implementer :

1. Definir une interface `User` avec les proprietes : `id`, `name`, `email`
2. Declarer les variables reactives : `users`, `loading`, `error`
3. Implementer la fonction `fetchUsers()` qui recupere les donnees
4. Appeler `fetchUsers()` au montage du composant
5. Afficher la liste avec gestion des etats loading/error

2.1 Etape 1.1 : Interface et variables

Completez le code suivant dans la partie `<script setup>` :

```
// Interface pour typer les utilisateurs
interface User {
  // TODO: Ajouter les proprietes id, name, email avec leurs types
}

// Variables reactives
const users = ref</* TODO: type du tableau */>([])
const loading = ref(/* TODO: valeur initiale */)
const error = ref</* TODO: type */>(null)
```

Indications

- `id` est un nombre, `name` et `email` sont des chaines
- Un tableau d'utilisateurs se type : `User[]`
- `loading` est un booleen, initialement `false`
- `error` peut etre une chaine ou `null`

2.2 Etape 1.2 : Fonction fetchUsers

Completez la fonction pour recuperer les utilisateurs :

```
async function fetchUsers() {
  // TODO 1: Mettre loading a true
  // TODO 2: Reinitialiser error a null

  try {
    // TODO 3: Appeler fetch() sur API_URL
    // TODO 4: Verifier si response.ok, sinon throw Error
```

```
    // TODO 5: Parser le JSON et stocker dans users.value

  } catch (err) {
    // TODO 6: Gerer l'erreur (verifier le type avec instanceof)

  } finally {
    // TODO 7: Remettre loading a false
  }
}
```

Indications

- `fetch()` retourne une Promise, utilisez `await`
- La reponse a une propriete `.ok` qui est `true` si le statut est 2xx
- `response.json()` retourne aussi une Promise
- Dans le catch : `err instanceof Error ? err.message : 'Erreur'`

2.3 Etape 1.3 : Chargement au demarrage

```
// TODO: Utiliser onMounted pour appeler fetchUsers au chargement
```

Astuce

```
onMounted(() => {
  // Code execute quand le composant est monte
})
```

2.4 Etape 1.4 : Template

Completez le template pour afficher les donnees :

```
<template>
  <div class="app">
    <h1>Gestion des Utilisateurs</h1>

    <h2>Liste des utilisateurs</h2>

    <!-- TODO: Bouton pour recharger (appelle fetchUsers) -->

    <!-- TODO: Afficher "Chargement..." si loading est true -->

    <!-- TODO: Afficher l'erreur si error n'est pas null -->

    <!-- TODO: Sinon, afficher la liste avec v-for -->
    <!-- Chaque li doit afficher : user.name - user.email -->

  </div>
```

```
</template>
```

Indications

Utilisez les directives :

- `v-if="loading"` pour le chargement
- `v-else-if="error"` pour l'erreur
- `v-else` pour la liste
- `v-for="user in users" :key="user.id"` pour la boucle

Validation

Testez votre code :

La liste des 10 utilisateurs s'affiche au chargement

"Chargement..." apparait brievement

Le bouton "Recharger" fonctionne

Testez l'erreur en modifiant l'URL (ajoutez "xxx")

3 Exercice 2 : POST - Creer un utilisateur

Exercice 2 (30 min)

Objectif : Creer un formulaire pour ajouter un nouvel utilisateur.

Fonctionnalites a implementer :

1. Formulaire avec champs nom et email (v-model)
2. Validation : les deux champs doivent etre remplis
3. Bouton desactive si formulaire invalide ou envoi en cours
4. Ajouter l'utilisateur cree a la liste locale
5. Vider le formulaire apres creation

3.1 Etape 2.1 : Variables du formulaire

Ajoutez les variables necessaires :

```
// Donnees du formulaire
const newUser = ref({
  // TODO: Proprietes name et email (chaines vides)
})

// Etat d'envoi
const creating = ref(/* TODO */)

// Validation du formulaire
const isFormValid = computed(() => {
  // TODO: Retourner true si name ET email ne sont pas vides
  // Astuce: utilisez .trim() pour ignorer les espaces
})
```

Indications

Pour `computed`, la syntaxe est :

```
const maVariable = computed(() => {
  return /* expression booleenne */
})
```

3.2 Etape 2.2 : Fonction createUser

```
async function createUser() {
  // TODO 1: Verifier isFormValid, sinon return

  // TODO 2: Mettre creating a true
```

```
  try {
    // TODO 3: Appeler fetch avec les options suivantes :
    //   - method: 'POST'
    //   - headers: { 'Content-Type': 'application/json' }
    //   - body: JSON.stringify(newUser.value)

    // TODO 4: Verifier response.ok

    // TODO 5: Parser la reponse JSON

    // TODO 6: Ajouter l'utilisateur a la liste avec unshift()
    //   Attention: utiliser un ID unique (Date.now())

    // TODO 7: Reinitialiser newUser a { name: '', email: '' }

  } catch (err) {
    // TODO 8: Gerer l'erreur

  } finally {
    // TODO 9: Remettre creating a false
  }
}
```

Attention

Erreur frequente avec body :
FAUX : `body: JSON.stringify({ newUser: newUser.value })`
CORRECT : `body: JSON.stringify(newUser.value)`
La premiere version envoie `{"newUser": {...}}` au lieu de `{"name": ..., "email": ...}`

3.3 Etape 2.3 : Template du formulaire

Ajoutez le formulaire **avant** la section liste :

```
<h2>Ajouter un utilisateur</h2>

<form @submit.prevent="/* TODO: appeler createUser */">
  <input
    v-model="/* TODO */"
    placeholder="Nom"
  />
  <input
    v-model="/* TODO */"
    placeholder="Email"
    type="email"
  />
  <button
    type="submit"
    :disabled="/* TODO: desactiver si invalide OU creating */"
  >
```

```
    <!-- TODO: Afficher "Envoi..." si creating, sinon "Ajouter" -->
  </button>
</form>
```

Indications

- `@submit.prevent` empeche le rechargement de la page
- `:disabled` accepte une expression booleenne
- Utilisez une expression ternaire pour le texte : `{{ condition ? 'texte1' : 'texte2' }}`

Validation

Testez votre code :

Le bouton est desactive si un champ est vide

Le bouton affiche "Envoi..." pendant la creation

L'utilisateur apparait en haut de la liste

Le formulaire se vide apres creation

4 Exercice 3 : DELETE - Supprimer un utilisateur

Exercice 3 (20 min)

Objectif : Ajouter la possibilite de supprimer un utilisateur.

Fonctionnalites a implementer :

1. Bouton "Supprimer" sur chaque ligne
2. Requete DELETE vers l'API
3. Retirer l'utilisateur de la liste locale
4. Desactiver le bouton pendant la suppression

4.1 Etape 3.1 : Variable de suivi

```
// Pour savoir quel utilisateur est en cours de suppression
const deletingId = ref</* TODO: quel type? number ou null */>(null)
```

4.2 Etape 3.2 : Fonction deleteUser

```
async function deleteUser(id: number) {
  // TODO 1: Mettre deletingId a l'id en cours

  try {
    // TODO 2: Appeler fetch sur `${API_URL}/${id}`
    //   avec method: 'DELETE'
    //   (pas de headers ni body necessaires)

    // TODO 3: Verifier response.ok

    // TODO 4: Filtrer users pour retirer l'utilisateur supprime
    //   users.value = users.value.filter(u => ...)

  } catch (err) {
    // TODO 5: Gerer l'erreur

  } finally {
    // TODO 6: Remettre deletingId a null
  }
}
```

Indications

- L'URL avec l'id : utilisez les template literals `` `${API_URL}/${id}` ``
- Pour filtrer : garder les utilisateurs dont l'id est **different** de celui supprime

4.3 Etape 3.3 : Bouton dans le template

Modifiez chaque `<li>` pour ajouter le bouton :

```
<li v-for="user in users" :key="user.id">
  <span>{{ user.name }} - {{ user.email }}</span>

  <button
    class="btn-danger"
    @click="/* TODO: appeler deleteUser avec user.id */"
    :disabled="/* TODO: desactiver si deletingId === user.id */"
  >
    <!-- TODO: Afficher "..." si en suppression, sinon "Supprimer"
       -->
  </button>
</li>
```

Validation

Testez votre code :

- Le clic sur "Supprimer" retire l'utilisateur
- Seul le bouton concerne affiche "..."
- Les autres boutons restent actifs

5 Exercice 4 : PATCH - Modifier un utilisateur

Exercice 4 (30 min)

Objectif : Permettre la modification d'un utilisateur en ligne.

Fonctionnalites a implementer :

1. Bouton "Modifier" pour passer en mode edition
2. Afficher des inputs a la place du texte
3. Boutons "Sauver" et "Annuler"
4. Requete PATCH pour sauvegarder
5. Mettre a jour la liste locale

5.1 Etape 4.1 : Variables d'edition

```
// ID de l'utilisateur en cours d'edition (null si aucun)
const editingId = ref<number | null>(null)

// Copie des donnees pour l'edition
const editForm = ref({
  name: '',
  email: ''
})

// Indicateur de sauvegarde en cours
const saving = ref(false)
```

5.2 Etape 4.2 : Fonctions startEdit et cancelEdit

```
function startEdit(user: User) {
  // TODO 1: Mettre editingId a user.id
  // TODO 2: Copier les donnees de user dans editForm
  //   editForm.value = { name: ..., email: ... }
}

function cancelEdit() {
  // TODO: Remettre editingId a null
}
```

5.3 Etape 4.3 : Fonction saveEdit

```
async function saveEdit(id: number) {
  // TODO 1: Mettre saving a true
```

```
  try {
    // TODO 2: Appeler fetch sur `${API_URL}/${id}` avec :
    //   - method: 'PATCH'
    //   - headers: { 'Content-Type': 'application/json' }
    //   - body: JSON.stringify(editForm.value)

    // TODO 3: Verifier response.ok

    // TODO 4: Trouver l'index de l'utilisateur dans users
    //   const index = users.value.findIndex(u => u.id === id)

    // TODO 5: Si trouve (index !== -1), mettre a jour :
    //   users.value[index] = { ...users.value[index], ...editForm.
       value }

    // TODO 6: Quitter le mode edition (editingId = null)

  } catch (err) {
    // TODO 7: Gerer l'erreur

  } finally {
    // TODO 8: Remettre saving a false
  }
}
```

Rappel

Spread operator pour fusionner des objets :

```
const original = { id: 1, name: 'Jean', email: 'jean@mail.com'
    }
const modifications = { name: 'Pierre' }
const resultat = { ...original, ...modifications }
// resultat = { id: 1, name: 'Pierre', email: 'jean@mail.com'
   }
```

5.4 Etape 4.4 : Template avec mode edition

Remplacez le contenu du <li> :

```
<li v-for="user in users" :key="user.id">

  <!-- MODE EDITION -->
  <template v-if="/* TODO: editingId === user.id */">
    <div>
      <input v-model="/* TODO: editForm.name */" />
      <input v-model="/* TODO: editForm.email */" />
    </div>
    <div>
      <button
        @click="/* TODO: saveEdit(user.id) */"
```

```
        :disabled="/* TODO: saving */"
      >
        <!-- TODO: "..." si saving, sinon "Sauver" -->
      </button>
      <button
        class="btn-secondary"
        @click="/* TODO: cancelEdit */"
      >
        Annuler
      </button>
    </div>
  </template>

  <!-- MODE AFFICHAGE -->
  <template v-else>
    <span>{{ user.name }} - {{ user.email }}</span>
    <div>
      <button @click="/* TODO: startEdit(user) */">
        Modifier
      </button>
      <button
        class="btn-danger"
        @click="deleteUser(user.id)"
        :disabled="deletingId === user.id"
      >
        {{ deletingId === user.id ? '...' : 'Supprimer' }}
      </button>
    </div>
  </template>

</li>
```

Validation

Testez votre code :

"Modifier" affiche les inputs avec les valeurs actuelles

"Annuler" revient au mode affichage sans sauvegarder

"Sauver" met a jour l'affichage

Un seul utilisateur peut etre en edition a la fois

6 Recapitulatif

Rappel

Les 4 methodes HTTP :

Methode	Usage	Specificites
GET	Lire	Pas de body
POST	Creer	Body obligatoire + Content-Type
PATCH	Modifier	URL avec ID + Body + Content-Type
DELETE	Supprimer	URL avec ID, pas de body

Rappel

Pattern standard pour une requete :

```
async function maRequete() {
  loading.value = true
  error.value = null

  try {
    const response = await fetch(URL, { /* options */ })
    if (!response.ok) throw new Error('Erreur')
    const data = await response.json()
    // Traiter data...
  } catch (err) {
    error.value = err instanceof Error ? err.message : 'Erreur
       '
  } finally {
    loading.value = false
  }
}
```

Rappel

Template avec les 3 etats :

```
<div v-if="loading">Chargement...</div>
<div v-else-if="error">{{ error }}</div>
<div v-else>
  <!-- Contenu principal -->
</div>
```

7 Bonus (si temps disponible)

Bonus 1 : Confirmation de suppression

Ajoutez une confirmation avant de supprimer :

```
async function deleteUser(id: number) {
  // TODO: Utiliser window.confirm() pour demander
     confirmation
  // Si l'utilisateur annule, ne pas continuer

  // ... reste du code
}
```

Bonus 2 : Recherche

Ajoutez un champ de recherche pour filtrer les utilisateurs :

```
const searchQuery = ref('')

const filteredUsers = computed(() => {
  // TODO: Si searchQuery est vide, retourner tous les users
  // Sinon, filtrer par name OU email (insensible a la casse)
})
```

Puis utilisez `filteredUsers` dans le `v-for` au lieu de `users`.

Bonus 3 : Compteur

Ajoutez un compteur d'utilisateurs :

```
const userCount = computed(() => {
  // TODO: Retourner le nombre d'utilisateurs
})
```

Affichez : "X utilisateur(s)" dans le template.

Bon travail !

N'hesitez pas a experimenter avec d'autres endpoints :
`/posts`, `/todos`, `/comments`

A Annexe : Backend Spring Boot

Cette annexe vous guide pour creer votre propre API REST avec Spring Boot, afin de remplacer JSONPlaceholder par votre backend.

A.1 Creation du projet

Etape A1 : Initialisation

1. Allez sur **Spring Initializr** : https://start.spring.io
2. Configurez le projet :
   - Project : **Maven**
   - Language : **Java**
   - Spring Boot : **3.2.x** (derniere version stable)
   - Group : `com.example`
   - Artifact : `user-api`
   - Packaging : **Jar**
   - Java : **17** ou **21**
3. Ajoutez les **dependances** :
   - **Spring Web** – Pour creer l'API REST
   - **Spring Data JPA** – Pour l'acces aux donnees
   - **H2 Database** – Base de donnees en memoire (dev)
   - **Lombok** – Pour reduire le code repetitif (optionnel)
4. Cliquez sur **Generate** et decompressez l'archive
5. Ouvrez le projet dans votre IDE (IntelliJ, VS Code, Eclipse)

A.2 Structure du projet

Creez la structure de packages suivante dans `src/main/java/com/example/userapi/` :

```
src/main/java/com/example/userapi/
+-- UserApiApplication.java      (existe deja)
+-- model/
|   +-- User.java
+-- repository/
|   +-- UserRepository.java
+-- service/
|   +-- UserService.java
+-- controller/
|   +-- UserController.java
+-- config/
```

```
    +-- CorsConfig.java
```

A.3 Configuration de la base de donnees

Etape A2 : Configuration H2

Modifiez le fichier `src/main/resources/application.properties` :

application.properties

```
# Configuration H2 (base de donnees en memoire)
spring.datasource.url=jdbc:h2:mem:userdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=

# JPA / Hibernate
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# Console H2 (pour debug)
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# Port du serveur
server.port=8080
```

Astuce

La console H2 sera accessible sur `http://localhost:8080/h2-console` pour visualiser les donnees.

A.4 Entite User

Etape A3 : Creer l'entite

Creez la classe `User.java` dans le package `model/` :

model/User.java

```
package com.example.userapi.model;

import jakarta.persistence.*;

@Entity
@Table(name = "users")
public class User {

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

```
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String email;

    private String phone;

    // Constructeur vide (requis par JPA)
    public User() {}

    // Constructeur avec parametres
    public User(String name, String email, String phone) {
        this.name = name;
        this.email = email;
        this.phone = phone;
    }

    // Getters et Setters
    public Long getId() { return id; }
    public void setId(Long id) { this.id = id; }

    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    public String getEmail() { return email; }
    public void setEmail(String email) { this.email = email; }

    public String getPhone() { return phone; }
    public void setPhone(String phone) { this.phone = phone; }
}
```

Indications

Annotations importantes :

- `@Entity` : Marque la classe comme entite JPA
- `@Table` : Nom de la table en base de donnees
- `@Id` : Cle primaire
- `@GeneratedValue` : Auto-increment
- `@Column` : Configuration de la colonne

A.5 Repository

Etape A4 : Creer le repository

Creez l'interface `UserRepository.java` dans le package `repository/` :

repository/UserRepository.java

```
package com.example.userapi.repository;

import com.example.userapi.model.User;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.Optional;

@Repository
public interface UserRepository extends JpaRepository<User, Long> {

    // Spring Data genere automatiquement les methodes CRUD :
    // - findAll()
    // - findById(Long id)
    // - save(User user)
    // - deleteById(Long id)

    // Methode personnalisee (optionnelle)
    Optional<User> findByEmail(String email);
}
```

Astuce

En etendant `JpaRepository<User, Long>`, vous obtenez automatiquement toutes les methodes CRUD sans ecrire de code !

A.6 Service

Etape A5 : Creer le service

Creez la classe `UserService.java` dans le package `service/` :

service/UserService.java

```
package com.example.userapi.service;

import com.example.userapi.model.User;
import com.example.userapi.repository.UserRepository;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.Optional;
```

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;

    // GET - Tous les utilisateurs
    public List<User> findAll() {
        return userRepository.findAll();
    }

    // GET - Un utilisateur par ID
    public Optional<User> findById(Long id) {
        return userRepository.findById(id);
    }

    // POST - Creer un utilisateur
    public User create(User user) {
        return userRepository.save(user);
    }

    // PATCH/PUT - Modifier un utilisateur
    public User update(Long id, User userData) {
        return userRepository.findById(id)
            .map(user -> {
                if (userData.getName() != null) {
                    user.setName(userData.getName());
                }
                if (userData.getEmail() != null) {
                    user.setEmail(userData.getEmail());
                }
                if (userData.getPhone() != null) {
                    user.setPhone(userData.getPhone());
                }
                return userRepository.save(user);
            })
            .orElseThrow(() ->
                new RuntimeException("User not found with id " + id
                    )
            );
    }

    // DELETE - Supprimer un utilisateur
    public void delete(Long id) {
        userRepository.deleteById(id);
    }
}
```

A.7 Controller REST

Etape A6 : Creer le controller

Creez la classe `UserController.java` dans le package `controller/` :

controller/UserController.java

```
package com.example.userapi.controller;

import com.example.userapi.model.User;
import com.example.userapi.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.List;

@RestController
@RequestMapping("/api/users")
public class UserController {

    @Autowired
    private UserService userService;

    // GET /api/users
    @GetMapping
    public List<User> getAllUsers() {
        return userService.findAll();
    }

    // GET /api/users/{id}
    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable Long id)
       {
        return userService.findById(id)
            .map(ResponseEntity::ok)
            .orElse(ResponseEntity.notFound().build());
    }

    // POST /api/users
    @PostMapping
    public ResponseEntity<User> createUser(@RequestBody User user)
       {
        User created = userService.create(user);
        return ResponseEntity
            .status(HttpStatus.CREATED)
            .body(created);
    }

    // PATCH /api/users/{id}
```

```
    @PatchMapping("/{id}")
    public ResponseEntity<User> updateUser(
            @PathVariable Long id,
            @RequestBody User user) {
        try {
            User updated = userService.update(id, user);
            return ResponseEntity.ok(updated);
        } catch (RuntimeException e) {
            return ResponseEntity.notFound().build();
        }
    }

    // DELETE /api/users/{id}
    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteUser(@PathVariable Long id) {
        userService.delete(id);
        return ResponseEntity.noContent().build();
    }
}
```

Indications

Annotations du Controller :

- `@RestController` : Combine `@Controller` + `@ResponseBody`
- `@RequestMapping` : Prefixe de toutes les routes
- `@GetMapping`, `@PostMapping`, etc. : Methodes HTTP
- `@PathVariable` : Recupere l'ID depuis l'URL
- `@RequestBody` : Parse le JSON du corps de la requete

A.8 Configuration CORS

Attention

Sans configuration CORS, votre frontend Vue.js (port 5173) ne pourra pas communiquer avec le backend Spring Boot (port 8080).

Etape A7 : Configurer CORS

Creez la classe `CorsConfig.java` dans le package `config/` :

config/CorsConfig.java

```
package com.example.userapi.config;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
```

```
import org.springframework.web.servlet.config.annotation.
    CorsRegistry;
import org.springframework.web.servlet.config.annotation.
    WebMvcConfigurer;

@Configuration
public class CorsConfig implements WebMvcConfigurer {

    @Override
    public void addCorsMappings(CorsRegistry registry) {
        registry.addMapping("/api/**")
            .allowedOrigins(
                "http://localhost:5173",  // Vite dev server
                "http://localhost:3000"   // Autre frontend
                    possible
            )
            .allowedMethods("GET", "POST", "PUT", "PATCH", "DELETE
                ", "OPTIONS")
            .allowedHeaders("*")
            .allowCredentials(true)
            .maxAge(3600);
    }
}
```

Astuce

Alternative : annotation par controller

Vous pouvez aussi ajouter `@CrossOrigin` directement sur le controller :

```
@RestController
@RequestMapping("/api/users")
@CrossOrigin(origins = "http://localhost:5173")
public class UserController {
    // ...
}
```

A.9 Donnees initiales (optionnel)

Etape A8 : Ajouter des donnees de test

Pour avoir des donnees au demarrage, creez un fichier `data.sql` :

src/main/resources/data.sql

```
INSERT INTO users (name, email, phone) VALUES
('Jean Dupont', 'jean@example.com', '0601020304'),
('Marie Martin', 'marie@example.com', '0605060708'),
('Pierre Durand', 'pierre@example.com', '0609101112'),
('Sophie Bernard', 'sophie@example.com', '0613141516'),
('Lucas Petit', 'lucas@example.com', '0617181920');
```

Ajoutez dans `application.properties` :

```
spring.jpa.defer-datasource-initialization=true
spring.sql.init.mode=always
```

A.10 Lancer et tester

Etape A9 : Demarrer le serveur

1. Dans le terminal, a la racine du projet Spring Boot :

   `./mvnw spring-boot:run`

   Ou sous Windows : `mvnw.cmd spring-boot:run`

2. Le serveur demarre sur `http://localhost:8080`

3. Testez les endpoints dans le navigateur ou avec curl :

   `curl http://localhost:8080/api/users`

Validation

Testez chaque endpoint :

`GET /api/users` – Liste des utilisateurs

`GET /api/users/1` – Un utilisateur

`POST /api/users` – Creer (avec body JSON)

`PATCH /api/users/1` – Modifier

`DELETE /api/users/1` – Supprimer

Console H2 : `http://localhost:8080/h2-console`

A.11 Connecter le frontend Vue.js

Etape A10 : Modifier le frontend

Dans votre projet Vue.js, modifiez simplement l'URL de l'API :

App.vue

```
// Remplacer :
const API_URL = 'https://jsonplaceholder.typicode.com/users'

// Par :
const API_URL = 'http://localhost:8080/api/users'
```

Attention

Ordre de demarrage :

1. Demarrez d'abord le backend Spring Boot (port 8080)
2. Puis demarrez le frontend Vue.js (port 5173)

A.12 Resume de l'architecture

Couche	Role	Annotation
Controller	Expose l'API REST	`@RestController`
Service	Logique metier	`@Service`
Repository	Acces base de donnees	`@Repository`
Entity	Modele de donnees	`@Entity`

Rappel

Flux d'une requete :

`Vue.js` $HTTP$ `Controller` $\rightarrow$ `Service` $\rightarrow$ `Repository` $\rightarrow$ `H2 DB`

Votre stack full-stack est prete !