Travaux Diriges

TD3 : Requetes HTTP avec Vue.js

GET, POST, PATCH, DELETE

Ibrahim ALAME

Formation Vue.js - 2025

Objectifs

Duree : 2h30

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

- Effectuer des requetes HTTP avec `fetch` dans Vue.js
- Recuperer des donnees depuis une API REST (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
- Utiliser les hooks de cycle de vie (`onMounted`)

Prerequis :

- Bases de Vue.js 3 (reactivite, composants)
- Connaissance de `async/await`
- Notions de base sur les API REST

Table des matières

1 Introduction aux requetes HTTP

1.1 Les methodes HTTP

Les API REST utilisent differentes methodes HTTP pour effectuer des operations CRUD (Create, Read, Update, Delete) :

Methode	Action	Corps	Exemple d'URL
GET	Lire	Non	`/users` ou `/users/1`
POST	Creer	Oui	`/users`
PATCH	Modifier partiellement	Oui	`/users/1`
PUT	Remplacer completement	Oui	`/users/1`
DELETE	Supprimer	Non	`/users/1`

Table 1: Methodes HTTP et leurs usages

1.2 L'API de test

Pour ce TD, nous utiliserons **JSONPlaceholder**, une API REST gratuite pour les tests :

`https://jsonplaceholder.typicode.com`

Endpoints disponibles :

- `/users` – Liste des utilisateurs
- `/posts` – Liste des articles
- `/todos` – Liste des taches
- `/comments` – Liste des commentaires

Attention

JSONPlaceholder est une **fake API**. Les requetes POST, PATCH et DELETE sont simulees : elles retournent une reponse valide mais ne modifient pas reellement les donnees sur le serveur.

1.3 La fonction fetch

Rappel

La fonction `fetch` est l'API native JavaScript pour effectuer des requetes HTTP.

Syntaxe de base :

```
const response = await fetch(url, options)
const data = await response.json()
```

Structure des options :

```
{
  method: 'POST',            // GET, POST, PATCH, DELETE
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(data) // Pour POST et PATCH
}
```

1.4 Pattern de gestion des etats

Pour une bonne UX, il faut toujours gerer 3 etats :

1. **Chargement** (`loading`) – Afficher un indicateur
2. **Erreur** (`error`) – Afficher le message d'erreur
3. **Succes** (`data`) – Afficher les donnees

Pattern recommande

```
<template>
  <div v-if="loading">Chargement...</div>
  <div v-else-if="error">{{ error }}</div>
  <div v-else>
    <!-- Afficher les donnees -->
  </div>
</template>
```

2 Mise en place du projet

2.1 Creation du projet

1. Creez un nouveau projet Vue.js avec Vite :

   ```
   npm create vue@latest td3-http
   ```

2. Selectionnez les options :

   - TypeScript : **Yes**
   - JSX : No
   - Vue Router : No
   - Pinia : No
   - Vitest : No
   - ESLint : Yes
   - Prettier : Yes

3. Installez les dependances et lancez le serveur :

   ```
   cd td3-http
   npm install
   npm run dev
   ```

2.2 Structure des fichiers

Creez la structure suivante dans le dossier `src/` :

```
src/
+-- components/
|   +-- UserList.vue        (Exercice 1 - GET)
|   +-- CreateUser.vue      (Exercice 2 - POST)
|   +-- UserCard.vue        (Exercice 3 - DELETE)
|   +-- EditUser.vue        (Exercice 4 - PATCH)
|   +-- UserManager.vue     (Exercice 5 - CRUD complet)
+-- App.vue
+-- main.ts
```

2.3 Configuration de App.vue

Modifiez `App.vue` pour tester vos composants :

```
<script setup lang="ts">
// Importez le composant a tester
import UserList from './components/UserList.vue'
</script>

<template>
```

```
  <div class="app">
    <h1>TD3 - Requetes HTTP</h1>
    <UserList />
  </div>
</template>

<style>
.app {
  max-width: 800px;
  margin: 0 auto;
  padding: 20px;
  font-family: Arial, sans-serif;
}
</style>
```

3 Exercice 1 : Requete GET - Afficher des utilisateurs

Exercice 1 - GET (30 min)

Objectif : Recuperer et afficher une liste d'utilisateurs depuis l'API.

Specifications :

- Charger les utilisateurs au montage du composant
- Afficher un message de chargement pendant la requete
- Afficher un message d'erreur si la requete echoue
- Afficher la liste des utilisateurs (nom, email, ville)
- Ajouter un bouton pour recharger les donnees

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

3.1 Structure de depart

components/UserList.vue

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

// TODO 1: Definir l'interface User
// Proprietes: id, name, email, phone, address (avec city)
interface User {
  // A completer...
}

// TODO 2: Declarer les variables reactives
// - users: tableau d'utilisateurs (initialement vide)
// - loading: booleen (initialement false)
// - error: string ou null (initialement null)


// TODO 3: Implementer la fonction fetchUsers
// - Mettre loading a true
// - Reinitialiser error a null
// - Utiliser try/catch/finally
// - Appeler fetch sur l'URL de l'API
// - Verifier response.ok
// - Parser le JSON et mettre a jour users
// - Gerer les erreurs
// - Mettre loading a false dans finally
async function fetchUsers() {
  // A completer...
}
```

```
28
29 // TODO 4: Appeler fetchUsers au montage du composant
30
31 </script>
32
33 <template>
34   <div class="user-list">
35     <h2>Liste des Utilisateurs</h2>
36
37     <!-- TODO 5: Bouton pour recharger -->
38
39     <!-- TODO 6: Affichage conditionnel -->
40     <!-- Si loading: afficher "Chargement..." -->
41     <!-- Sinon si error: afficher le message d'erreur -->
42     <!-- Sinon: afficher la liste des utilisateurs -->
43
44   </div>
45 </template>
46
47 <style scoped>
48 .user-list {
49   padding: 20px;
50 }
51
52 /* TODO 7: Ajouter vos styles */
53 </style>
```

3.2 Indications

Indication

TODO 1 - Interface User :
Consultez la structure retournee par l'API en visitant directement l'URL dans votre navigateur. L'adresse (`address`) est un objet imbrique contenant `city`.

TODO 2 - Variables reactives :
Utilisez `ref<Type>(valeurInitiale)` pour chaque variable :

```
const maVariable = ref<MonType>(valeurInitiale)
```

TODO 3 - Fonction fetchUsers :
Structure recommandee :

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

  try {
    const response = await fetch('URL_ICI')

    if (!response.ok) {
      throw new Error('Erreur HTTP: ' + response.status)
    }

    users.value = await response.json()
  } catch (err) {
    // Gerer l'erreur...
  } finally {
    loading.value = false
  }
}
```

TODO 4 - Hook onMounted :

```
onMounted(() => {
  fetchUsers()
})
```

TODO 6 - Template :
Utilisez `v-if`, `v-else-if`, `v-else` et `v-for` pour afficher les donnees.

Astuce

Pour typer l'erreur dans le catch :

```
catch (err) {
  error.value = err instanceof Error
    ? err.message
    : 'Une erreur est survenue'
}
```

4 Exercice 2 : Requete POST - Creer un utilisateur

Exercice 2 - POST (30 min)

Objectif : Creer un formulaire pour ajouter un nouvel utilisateur.

Specifications :

- Formulaire avec champs : nom, email, telephone
- Validation : tous les champs obligatoires
- Bouton "Creer" desactive si formulaire invalide
- Afficher un indicateur pendant l'envoi
- Afficher un message de succes avec les donnees retournees
- Reinitialiser le formulaire apres succes

API : `POST https://jsonplaceholder.typicode.com/users`
Corps de la requete :

```
{
  "name": "John Doe",
  "email": "john@example.com",
  "phone": "0612345678"
}
```

4.1 Structure de depart

components/CreateUser.vue

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

// TODO 1: Definir l'interface pour le formulaire
interface UserForm {
  name: string
  email: string
  phone: string
}

// TODO 2: Declarer les variables reactives
// - form: objet UserForm avec valeurs vides
// - loading: booleen
// - error: string | null
// - success: booleen
// - createdUser: objet ou null (utilisateur cree)


```

```
// TODO 3: Creer une propriete computed isFormValid
// Retourne true si tous les champs sont remplis


// TODO 4: Implementer la fonction createUser
// - Verifier que le formulaire est valide
// - Mettre loading a true, reinitialiser error et success
// - Utiliser fetch avec method: 'POST'
// - Ajouter les headers Content-Type
// - Envoyer le body en JSON.stringify
// - Gerer la reponse et les erreurs
// - Reinitialiser le formulaire en cas de succes
async function createUser() {
  // A completer...
}

// TODO 5: Fonction pour reinitialiser le formulaire
function resetForm() {
  // A completer...
}
</script>

<template>
  <div class="create-user">
    <h2>Creer un Utilisateur</h2>

    <!-- TODO 6: Message de succes (si success est true) -->

    <!-- TODO 7: Formulaire -->
    <!-- Utiliser @submit.prevent pour eviter le rechargement -->
    <form>
      <!-- Champ Nom -->
      <div class="field">
        <label for="name">Nom</label>
        <!-- TODO: input avec v-model -->
      </div>

      <!-- Champ Email -->
      <div class="field">
        <label for="email">Email</label>
        <!-- TODO: input type="email" avec v-model -->
      </div>

      <!-- Champ Telephone -->
      <div class="field">
        <label for="phone">Telephone</label>
        <!-- TODO: input avec v-model -->
      </div>

      <!-- TODO 8: Message d'erreur (si error) -->

```

```
      <!-- TODO 9: Bouton submit -->
      <!-- Desactive si !isFormValid ou loading -->
      <!-- Texte: "Creation..." si loading, sinon "Creer" -->

    </form>
  </div>
</template>

<style scoped>
/* TODO 10: Ajouter vos styles */
.create-user {
  padding: 20px;
}

.field {
  margin-bottom: 15px;
}

.field label {
  display: block;
  margin-bottom: 5px;
}

.field input {
  width: 100%;
  padding: 8px;
  border: 1px solid #ddd;
  border-radius: 4px;
}
</style>
```

4.2 Indications

Indication

TODO 4 - Requete POST :
Structure d'une requete POST avec fetch :

```
const response = await fetch(URL, {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify(donnees)
})
```

TODO 3 - Validation :
Utilisez `computed` pour verifier que tous les champs ne sont pas vides :

```
const isFormValid = computed(() => {
  return form.value.name.trim() !== '' &&
         form.value.email.trim() !== '' &&
         // ...
})
```

TODO 9 - Bouton avec etats :

```
<button
  type="submit"
  :disabled="!isFormValid || loading"
>
  {{ loading ? 'Creation...' : 'Creer' }}
</button>
```

Attention

N'oubliez pas `@submit.prevent` sur le formulaire pour empecher le rechargement de la page !

5 Exercice 3 : Requete DELETE - Supprimer un utilisateur

Exercice 3 - DELETE (25 min)

Objectif : Afficher une liste d'utilisateurs avec possibilite de suppression.

Specifications :

- Charger et afficher les utilisateurs (reutiliser l'exercice 1)
- Ajouter un bouton "Supprimer" sur chaque carte
- Desactiver le bouton pendant la suppression
- Retirer l'utilisateur de la liste apres suppression reussie
- Afficher un message d'erreur si la suppression echoue

API : `DELETE https://jsonplaceholder.typicode.com/users/{id}`

5.1 Structure de depart

components/UserCard.vue

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

interface User {
  id: number
  name: string
  email: string
}

const users = ref<User[]>([])
const loading = ref(false)
const error = ref<string | null>(null)

// TODO 1: Variable pour suivre quel utilisateur est en cours de suppression
// Astuce: utilisez l'ID de l'utilisateur (number | null)


// TODO 2: Fonction fetchUsers (copier de l'exercice 1)


// TODO 3: Implementer la fonction deleteUser(id: number)
// - Mettre deletingId a l'ID en cours
// - Appeler fetch avec method: 'DELETE'
// - Verifier response.ok
// - Filtrer users pour retirer l'utilisateur supprime
```

```
// - Gerer les erreurs
// - Remettre deletingId a null
async function deleteUser(id: number) {
  // A completer...
}

onMounted(() => {
  fetchUsers()
})
</script>

<template>
  <div class="user-cards">
    <h2>Utilisateurs</h2>

    <div v-if="loading">Chargement...</div>
    <div v-else-if="error" class="error">{{ error }}</div>

    <div v-else class="cards-grid">
      <!-- TODO 4: Boucle sur les utilisateurs -->
      <!-- Pour chaque utilisateur, afficher une carte avec :
           - Nom et email
           - Bouton Supprimer
           - Le bouton est desactive si deletingId === user.id
           - Texte du bouton: "Suppression..." ou "Supprimer"
      -->
    </div>

    <p v-if="!loading && users.length === 0">
      Aucun utilisateur
    </p>
  </div>
</template>

<style scoped>
.cards-grid {
  display: grid;
  grid-template-columns: repeat(auto-fill, minmax(250px, 1fr));
  gap: 15px;
}

/* TODO 5: Styles pour les cartes */
</style>
```

5.2 Indications

Indication

TODO 1 - Suivi de la suppression :
Pour savoir quel utilisateur est en cours de suppression :

```
const deletingId = ref<number | null>(null)
```

TODO 3 - Requete DELETE :
La requete DELETE ne necessite pas de body :

```
const response = await fetch(`${URL}/${id}`, {
  method: 'DELETE'
})
```

Filtrer la liste apres suppression :

```
users.value = users.value.filter(user => user.id !== id)
```

TODO 4 - Bouton conditionnel :

```
<button
  @click="deleteUser(user.id)"
  :disabled="deletingId === user.id"
>
  {{ deletingId === user.id ? 'Suppression...' : 'Supprimer'
     }}
</button>
```

Astuce

Utilisez les template literals (backticks) pour construire l'URL avec l'ID :

```
const url = `https://jsonplaceholder.typicode.com/users/${id}`
```

6 Exercice 4 : Requete PATCH - Modifier un utilisateur

Exercice 4 - PATCH (35 min)

Objectif : Permettre l'edition en ligne d'un utilisateur.

Specifications :

- Afficher les utilisateurs avec un bouton "Modifier"
- Au clic, passer en mode edition (afficher des inputs)
- Boutons "Sauvegarder" et "Annuler" en mode edition
- Envoyer uniquement les champs modifies (PATCH)
- Mettre a jour l'affichage apres sauvegarde

API : `PATCH https://jsonplaceholder.typicode.com/users/{id}`
Corps de la requete :

```
{
  "name": "Nouveau nom",
  "email": "nouveau@email.com"
}
```

6.1 Structure de depart

components/EditUser.vue

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

interface User {
  id: number
  name: string
  email: string
  phone: string
}

const users = ref<User[]>([])
const loading = ref(false)
const error = ref<string | null>(null)

// TODO 1: Variables pour le mode edition
// - editingId: ID de l'utilisateur en cours d'edition (number | null)
// - editForm: objet contenant les valeurs du formulaire d'edition
// - saving: booleen pour indiquer la sauvegarde en cours


```

```
// TODO 2: Fonction fetchUsers (copier des exercices precedents)


// TODO 3: Fonction startEdit(user: User)
// - Mettre editingId a user.id
// - Copier les valeurs de user dans editForm
function startEdit(user: User) {
  // A completer...
}

// TODO 4: Fonction cancelEdit()
// - Remettre editingId a null
// - Reinitialiser editForm
function cancelEdit() {
  // A completer...
}

// TODO 5: Fonction saveEdit(id: number)
// - Mettre saving a true
// - Appeler fetch avec method: 'PATCH'
// - Envoyer editForm dans le body
// - Mettre a jour l'utilisateur dans la liste users
// - Quitter le mode edition
async function saveEdit(id: number) {
  // A completer...
}

onMounted(() => {
  fetchUsers()
})
</script>

<template>
  <div class="edit-users">
    <h2>Modifier les Utilisateurs</h2>

    <div v-if="loading">Chargement...</div>
    <div v-else-if="error" class="error">{{ error }}</div>

    <ul v-else class="user-list">
      <li v-for="user in users" :key="user.id" class="user-item">

        <!-- TODO 6: Mode edition (si editingId === user.id) -->
        <!-- Afficher des inputs pour name, email, phone -->
        <!-- Boutons Sauvegarder et Annuler -->
        <template v-if="editingId === user.id">
          <!-- A completer... -->
        </template>

        <!-- TODO 7: Mode affichage (sinon) -->
        <!-- Afficher les infos et un bouton Modifier -->
```

```
72          <template v-else>
73            <!-- A completer... -->
74          </template>
75
76        </li>
77      </ul>
78    </div>
79  </template>
80
81  <style scoped>
82  .user-list {
83    list-style: none;
84    padding: 0;
85  }
86
87  .user-item {
88    padding: 15px;
89    border: 1px solid #ddd;
90    border-radius: 8px;
91    margin-bottom: 10px;
92  }
93
94  /* TODO 8: Styles supplementaires */
95  </style>
```

6.2 Indications

Indication

TODO 1 - Variables d'edition :

```
const editingId = ref<number | null>(null)
const editForm = ref({
  name: '',
  email: '',
  phone: ''
})
const saving = ref(false)
```

TODO 3 - Demarrer l'edition :
Copiez les valeurs actuelles pour pouvoir les modifier :

```
function startEdit(user: User) {
  editingId.value = user.id
  editForm.value = {
    name: user.name,
    email: user.email,
    phone: user.phone
  }
}
```

TODO 5 - Requete PATCH :
Structure similaire a POST, mais avec `method: 'PATCH'` :

```
const response = await fetch(`${URL}/${id}`, {
  method: 'PATCH',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(editForm.value)
})
```

Mettre a jour la liste locale :

```
const index = users.value.findIndex(u => u.id === id)
if (index !== -1) {
  users.value[index] = {
    ...users.value[index],
    ...editForm.value
  }
}
```

Attention

La difference entre PUT et PATCH :

- **PUT** : Remplace **toute** la ressource
- **PATCH** : Modifie **partiellement** la ressource

Avec PATCH, vous n'envoyez que les champs a modifier.

7 Exercice 5 : Application CRUD complete

Exercice 5 - CRUD complet (30 min)

Objectif : Combiner tous les concepts dans une application complete.

Specifications :

- Afficher la liste des utilisateurs (GET)
- Formulaire pour ajouter un utilisateur (POST)
- Bouton pour supprimer chaque utilisateur (DELETE)
- Edition en ligne de chaque utilisateur (PATCH)
- Compteur d'utilisateurs
- Gestion propre des etats de chargement et d'erreur

7.1 Structure recommandee

components/UserManager.vue - Structure

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

// Interface
interface User {
  id: number
  name: string
  email: string
  phone: string
}

// === ETATS ===
const users = ref<User[]>([])
const loading = ref(false)
const error = ref<string | null>(null)

// Etats pour la creation
const newUser = ref({ name: '', email: '', phone: '' })
const creating = ref(false)

// Etats pour l'edition
const editingId = ref<number | null>(null)
const editForm = ref({ name: '', email: '', phone: '' })
const saving = ref(false)

// Etat pour la suppression
const deletingId = ref<number | null>(null)

```

```
// === COMPUTED ===
// TODO: Compteur d'utilisateurs
// TODO: Validation du formulaire de creation

// === FONCTIONS ===
// TODO: fetchUsers() - GET
// TODO: createUser() - POST
// TODO: deleteUser(id) - DELETE
// TODO: startEdit(user), cancelEdit(), saveEdit(id) - PATCH

onMounted(() => {
  fetchUsers()
})
</script>
```

7.2 Indications pour le template

Indication

Structure du template recommandee :

1. **En-tete** avec titre et compteur
2. **Formulaire d'ajout** (toujours visible)
3. **Zone de chargement/erreur**
4. **Liste des utilisateurs** avec pour chaque :
   - Mode affichage OU mode edition
   - Boutons d'action (Modifier, Supprimer)

Conseil : Decomposez votre template en sections claires avec des commentaires :

```
<template>
  <div class="user-manager">
    <!-- En-tete -->
    <header>...</header>

    <!-- Formulaire d'ajout -->
    <section class="add-section">...</section>

    <!-- Liste des utilisateurs -->
    <section class="list-section">
      <div v-if="loading">...</div>
      <div v-else-if="error">...</div>
      <ul v-else>...</ul>
    </section>
  </div>
</template>
```

7.3 Grille d'auto-evaluation

Verifiez que votre application respecte ces criteres :

Critere	OK ?
Les utilisateurs s'affichent au chargement	
Un indicateur de chargement est affiche	
Les erreurs sont affichees proprement	
Le formulaire d'ajout fonctionne	
Le formulaire est valide avant soumission	
L'utilisateur cree apparait dans la liste	
La suppression retire l'utilisateur de la liste	
Le bouton est desactive pendant la suppression	
L'edition en ligne fonctionne	
Les modifications sont sauvegardees	
Le compteur se met a jour automatiquement	
Le code est propre et commente	

Table 2: Grille d'auto-evaluation

8 Pour aller plus loin

8.1 Bonus 1 : Confirmation de suppression

Ajoutez une demande de confirmation avant de supprimer :

```
async function deleteUser(id: number) {
  // TODO: Utiliser window.confirm() ou une modal
  if (!confirm('Voulez-vous vraiment supprimer cet utilisateur ?'))
      {
    return
  }
  // ... reste du code
}
```

8.2 Bonus 2 : Recherche et filtrage

Ajoutez un champ de recherche pour filtrer les utilisateurs :

```
const searchQuery = ref('')

const filteredUsers = computed(() => {
  if (!searchQuery.value) return users.value

  const query = searchQuery.value.toLowerCase()
  return users.value.filter(user =>
    user.name.toLowerCase().includes(query) ||
    user.email.toLowerCase().includes(query)
  )
})
```

8.3 Bonus 3 : Gestion des erreurs par requete

Affichez les erreurs specifiques a chaque operation :

```
const errors = ref({
  fetch: null as string | null,
  create: null as string | null,
  delete: null as string | null,
  update: null as string | null
})
```

8.4 Bonus 4 : Composable reutilisable

Creez un composable pour centraliser la logique HTTP :

composables/useUsers.ts

```
import { ref } from 'vue'

export function useUsers() {
  const users = ref([])
  const loading = ref(false)
  const error = ref(null)

  async function fetchUsers() { /* ... */ }
  async function createUser(data) { /* ... */ }
  async function updateUser(id, data) { /* ... */ }
  async function deleteUser(id) { /* ... */ }

  return {
    users,
    loading,
    error,
    fetchUsers,
    createUser,
    updateUser,
    deleteUser
  }
}
```

9 Recapitulatif

Rappel

Resume des requetes HTTP avec fetch :

GET (lecture) :

```
const response = await fetch(URL)
const data = await response.json()
```

POST (creation) :

```
const response = await fetch(URL, {
  method: 'POST',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(data)
})
```

PATCH (modification) :

```
const response = await fetch(`${URL}/${id}`, {
  method: 'PATCH',
  headers: { 'Content-Type': 'application/json' },
  body: JSON.stringify(data)
})
```

DELETE (suppression) :

```
const response = await fetch(`${URL}/${id}`, {
  method: 'DELETE'
})
```

Pattern de gestion des etats :

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

Bon travail !

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