

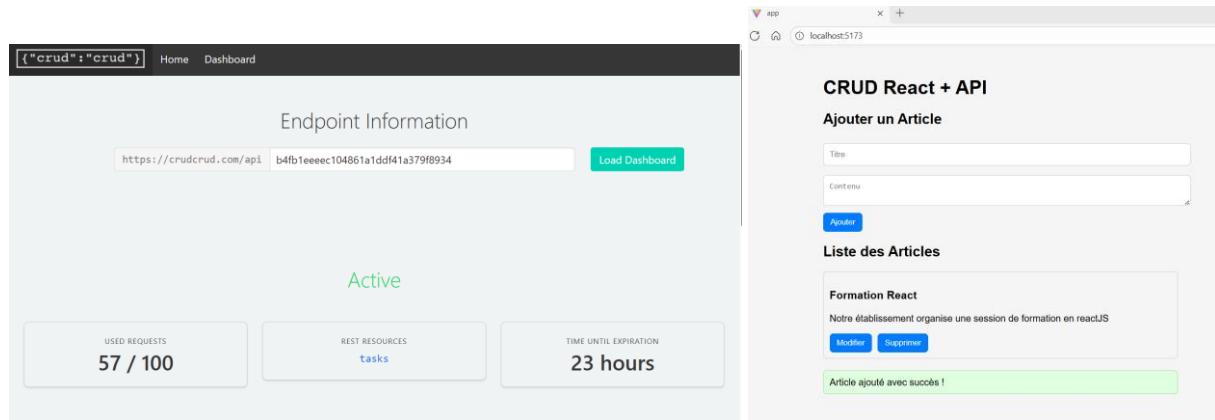
Atelier 06 APICrud

Objectifs pédagogiques

- Organiser le CRUD dans un dossier src/crud/
- Comprendre chaque étape avec des instructions claires
- Utiliser une API réellement compatible CRUD : <https://crudcrud.com/> (gratuite & complète)
- Afficher les messages de retour directement **en bas de la page**, pas dans la console
- Déplacer l'appel principal (Router + Provider) dans **src/main.jsx**
- Ajouter une section CSS finale pour styliser l'atelier

API utilisée :

👉 <https://crudcrud.com/> (GET, POST, PUT, DELETE 100% gratuits et fonctionnels)



Étape 1 : Préparer l'environnement React

Objectif pédagogique :

Savoir créer un projet React fonctionnel et structurer les fichiers.

Instructions :

1. Dans le dossier Ressources de cet atelier, copier le dossier **APICrudApp** dans le dossier **C:\ReactProjects**
2. Ouvrir le dossier **C:\ReactProjects\ APICrudApp** avec votre éditeur de code
3. Démarrer docker et vérifier que les containers ne sont pas en exécution

4. Avec l'invite de commande, entre dans le dossier C:\ReactProjects\APICrudApp et lancer la commande suivante :

docker-compose up -d --build

5. Créez un projet React avec Vite ou Create React App.

1. Accéder au APICrudApp_container avec la commande suivante :

docker exec -it APICrudApp_container sh

2. npm create vite@latest . --template

Deux traits avant
template

```
/usr/src/app # npm create vite@latest . --template
> npx
> create-vite .

Select a framework:
  React

Select a variant:
  JavaScript

Use rollup-vite (Experimental)?:
  No

Install with npm and start now?
  Yes / No
```

3. Si le serveur Vite lancé, arrête-le (CTR + C)

4. Quitter le container

```
/usr/src/app # exit
```

5. Modifier le fichier vite.config.js

```
import { defineConfig } from 'vite'
import react from '@vitejs/plugin-react'

// https://vitejs.dev/config/
export default defineConfig({
  plugins: [react()],
  server: {
    host: true,           // équivaut à --host
    watch: {
      usePolling: true, // <== active le mode "polling"
      interval: 1000,   // <== vérifie les changements toutes les 1s
    },
  },
})
```

6. Puis arrêter et démarrer le container, directement avec le docker ou bien exécuter ces deux commandes

```
docker-compose down
docker-compose up -d --build
```

7. Entre dans le container

```
docker exec -it APICrudApp_container sh
```

8. Lancer cette commande :

```
npm run dev -- --host
```

9. Lancer l'application sur l'url <http://localhost:5173/>

Structure du Projet

```
src/
  |- crud/
    |   |- PostList.jsx
    |   |- PostItem.jsx
    |   |- PostForm.jsx
    |   |- MessageBox.jsx
    |- App.jsx
    |- main.jsx
```

Étape 1 : Configurer main.jsx (appel de l'API ici)

Objectif : centraliser le routeur ou la logique globale

```
import React from 'react'
import ReactDOM from 'react-dom/client'
import App from './App'
import './crud/style.css'

ReactDOM.createRoot(document.getElementById('root')).render(
  <React.StrictMode>
    <App />
  </React.StrictMode>
)
```

Étape 2 : App.jsx

Objectif : charger les composants CRUD

```
import PostList from './crud/PostList'
import PostForm from './crud/PostForm'
import MessageBox from './crud	MessageBox'
import { createContext, useState } from 'react'

export const ApiContext = createContext("")
```

```

export default function App() {
  const [message, setMessage] = useState("")

  const API_URL = "https://crudcrud.com/api/b4fb1eeeeec104861a1ddf41a379f8934/tasks"

  return (
    <ApiContext.Provider value={API_URL}>
      <div style={{ width: "600px", margin: "auto" }}>
        <h1>CRUD React + API</h1>
        <PostForm setMessage={setMessage} />
        <PostList setMessage={setMessage} message={message} />
        <MessageBox message={message} />
      </div>
    </ApiContext.Provider>
  )
}

```

Étape 3 : MessageBox.jsx

Objectif : afficher les messages en bas de page

```

export default function MessageBox({ message }) {
  if (!message) return null

  return (
    <div style={{
      marginTop: "20px",
      padding: "10px",
      background: "#e0ffe0",
      borderRadius: "6px",
      border: "1px solid #9cd89c"
    }}>
      {message}
    </div>
  )
}

```

Étape 4 : Liste des Articles — PostList.jsx

Objectif : afficher + supprimer + bouton modifier

```

import { useState, useEffect, useContext } from "react"
import PostItem from "./PostItem"
import { ApiContext } from "../App"

export default function PostList({ setMessage, message }) {
  const [posts, setPosts] = useState([])
  const API_URL = useContext(ApiContext)

  useEffect(() => {
    const loadPosts = async () => {
      const res = await fetch(API_URL)
      const data = await res.json()
      // crudcrud returns an array for the collection; guard safely
      const items = Array.isArray(data) ? data.slice(0, 10) : []
      setPosts(items)
    }
    loadPosts()
  }, [API_URL, message])
}

```

```

const deletePost = async (id) => {
  await fetch(`/${API_URL}/${id}`, { method: "DELETE" })
  setPosts(posts.filter(p => (p._id || p.id) !== id))
  setMessage("Article supprimé avec succès !")
}

return (
  <div>
    <h2>Liste des Articles</h2>
    {posts.map(post => (
      <PostItem
        key={post._id || post.id}
        post={post}
        deletePost={deletePost}
        setPosts={setPosts}
        setMessage={setMessage}
      />
    ))}
  </div>
)
}

```

Étape 5 : PostItem.jsx (Afficher + Modifier + Supprimer)

Objectif : ajouter boutons edit/delete

```

import { useState, useContext } from "react"
import { ApiContext } from "../App"

export default function PostItem({ post, deletePost, setPosts, setMessage }) {
  const [editing, setEditing] = useState(false)
  const [title, setTitle] = useState(post.title)
  const [body, setBody] = useState(post.body || "")
  const API_URL = useContext(ApiContext)

  const updatePost = async () => {
    const id = post._id || post.id
    if (!API_URL) {
      console.error("API_URL is not defined")
      setMessage("Erreur: URL API non configurée")
      return
    }

    try {
      // Build payload without id fields – some backends reject changing the id
      const { _id, id: legacyId, ...rest } = post
      const payload = { ...rest, title, body }
      const url = `${API_URL}/${id}`
      console.log("Updating post", { url, payload })

      // Try PUT first
      let res
      try {
        res = await fetch(url, {
          method: "PUT",
          headers: { "Content-Type": "application/json" },
          body: JSON.stringify(payload),
          mode: "cors"
        })
      } catch (networkErr) {
        console.error("Network error on PUT:", networkErr)
        // fallback to PATCH attempt
        res = await fetch(url, {
          method: "PATCH",
          headers: { "Content-Type": "application/json" },
        })
      }
      if (!res.ok) {
        const error = new Error(`Error ${res.status}: ${res.statusText}`)
        error.response = res
        throw error
      }
      const updatedPost = await res.json()
      setPosts([
        ...posts.slice(0, id),
        { ...updatedPost, _id: legacyId },
        ...posts.slice(id + 1),
      ])
      setMessage(`Post updated successfully`)
    } catch (err) {
      console.error("Error updating post", err)
      setMessage("An error occurred while updating the post")
    }
  }
}

```

```

        body: JSON.stringify(payload),
        mode: "cors"
    })
}

if (!res.ok) {
    const text = await res.text().catch(() => "")
    throw new Error(`Update failed: ${res.status} ${text}`)
}

// Some APIs (including crudcrud) may return an empty response body on successful
PUT.
// Guard against calling res.json() on an empty body.
const text = await res.text().catch(() => "")
let updated
if (text) {
    try {
        updated = JSON.parse(text)
    } catch (e) {
        console.warn("Failed to parse JSON response, falling back to payload", e)
        updated = { ...payload, _id: id, id }
    }
} else {
    // No body returned; assume server accepted the payload and use it as the updated
resource.
    updated = { ...payload, _id: id, id }
}

const updatedId = updated._id || updated.id
setPosts(prev => prev.map(p => (p._id === updatedId || p.id === updatedId) ? updated
: p))
setEditing(false)
setMessage && setMessage("Article modifié !")
} catch (err) {
    console.error(err)
    setMessage && setMessage("Erreur lors de la modification de l'article")
}
}

return (
    <div style={{ border: "1px solid #ccc", padding: "10px", margin: "10px 0", borderRadius:
"6px" }}>

        {editing ? (
            <div>
                <input value={title} onChange={(e) => setTitle(e.target.value)} />
                <textarea value={body} onChange={(e) => setBody(e.target.value)} />
            </div>
        ) : (
            <>
                <h3>{post.title}</h3>
                <p>{post.body}</p>
            </>
        )}
    {editing ? (
        <button onClick={updatePost}>Enregistrer</button>
    ) : (
        <button onClick={() => setEditing(true)}>Modifier</button>
    )}

    <button onClick={() => deletePost(post._id || post.id)}>Supprimer</button>
</div>
)
}
}

```

Étape 6 : Ajouter un Article — PostForm.jsx

Objectif : formulaire POST + message visuel

```
import { useState, useContext } from "react"
import { ApiContext } from "../App"

export default function PostForm({ setMessage }) {
  const [title, setTitle] = useState("")
  const [body, setBody] = useState("")
  const [loading, setLoading] = useState(false)

  const API_URL = useContext(ApiContext)

  const handleSubmit = async (e) => {
    e.preventDefault()

    if (!title.trim() || !body.trim()) {
      if (setMessage) setMessage("Veuillez remplir le titre et le contenu")
      return
    }

    setLoading(true)
    try {
      const res = await fetch(API_URL, {
        method: "POST",
        headers: { "Content-Type": "application/json" },
        body: JSON.stringify({ title, body, userId: 1 })
      })

      if (!res.ok) {
        throw new Error(`Request failed with status ${res.status}`)
      }

      await res.json()
      if (setMessage) setMessage("Article ajouté avec succès !")
      setTitle("")
      setBody("")
    } catch (err) {
      console.error(err)
      if (setMessage) setMessage("Erreur lors de l'ajout de l'article")
    } finally {
      setLoading(false)
    }
  }

  return (
    <form onSubmit={handleSubmit} style={{ marginBottom: "20px" }}>
      <h2>Ajouter un Article</h2>

      <input
        placeholder="Titre"
        value={title}
        onChange={(e) => setTitle(e.target.value)}
      />
      <textarea
        placeholder="Contenu"
        value={body}
        onChange={(e) => setBody(e.target.value)}
      />

      <button type="submit" disabled={loading}>
        {loading ? "Envoi..." : "Ajouter"}
      </button>
    </form>
  )
}
```

🎨 Étape 7 : Style CSS (simple et propre)

Crée un fichier :

src/crud/style.css

```
body {  
  font-family: Arial, sans-serif;  
  background: #f5f5f5;  
  padding: 20px;  
}  
  
input, textarea {  
  width: 100%;  
  padding: 10px;  
  margin: 8px 0;  
  border: 1px solid #ccc;  
  border-radius: 6px;  
}  
  
button {  
  padding: 8px 12px;  
  margin-right: 10px;  
  background: #007bff;  
  color: white;  
  border: none;  
  border-radius: 6px;  
  cursor: pointer;  
}  
  
button:hover {  
  background: #0056b3;  
}
```

Puis importe ce CSS dans **main.jsx** :

```
import './crud/style.css'
```

🎯 Atelier Complet : CRUD avec React + API Gratuite

Objectif général :

Apprendre à créer une application CRUD complète avec React, en consommant une API REST, en séparant correctement les composants, en gérant les formulaires, les états, les messages d'erreur/succès et le style.

Nous allons utiliser **l'API CRUD complète gratuite suivante :**

👉 <https://crudcrud.com> (génère une URL privée valable 24h)

Elle permet :

- ✓ GET
- ✓ POST
- ✓ PUT
- ✓ DELETE

👉 Si jamais crudcrud bloque, alternative : <https://jsonplaceholder.typicode.com/> (MAIS ne supporte pas bien PUT/DELETE).

📁 Structure finale du projet

```
src/
  ├── main.jsx ← appels API ici !
  ├── App.jsx
  └── crud/
    ├── CrudList.jsx
    ├── CrudForm.jsx
    ├── CrudItem.jsx
    └── crud.css
```

✿ Étape 1 — Préparer la structure

🎓 Objectif : créer les fichiers nécessaires

Créer le dossier et les fichiers :

```
src/crud/CrudList.jsx
src/crud/CrudItem.jsx
src/crud/CrudForm.jsx
src/crud/crud.css
```

✳ Étape 2 — Configurer l'API dans main.jsx

🎓 Objectif :

- Centraliser les appels API
- Fournir les fonctions CRUD à App.jsx

👉 Va sur <https://crudcrud.com> → copie ta clé (ex :
https://crudcrud.com/api/XXXXXXXXXX/tasks)

📌 src/main.jsx

```
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";
import App from "./App.jsx";

const API_URL = "https://crudcrud.com/api/XXXXXXXXXX/tasks";

function Main() {
  const [tasks, setTasks] = useState([]);
  const [message, setMessage] = useState("");

  // -----
  // READ
  // -----
  const fetchTasks = async () => {
    try {
      const res = await fetch(API_URL);
      const data = await res.json();
      setTasks(data);
    } catch (error) {
      setMessage(error.message);
    }
  };

  useEffect(() => {
    fetchTasks();
  }, []);

  return (
    <div>
      <h1>React API Crud</h1>
      <p>{message}</p>
      <table>
        <thead>
          <tr>
            <th>Task ID</th>
            <th>Task Description</th>
            <th>Actions</th>
          </tr>
        </thead>
        <tbody>
          {tasks.map((task) => (
            <tr>
              <td>{task._id}</td>
              <td>{task.description}</td>
              <td>
                <a href="#">Edit</a>
                <a href="#">Delete</a>
              </td>
            </tr>
          ))}
        </tbody>
      </table>
    </div>
  );
}

const root = ReactDOM.createRoot(document.getElementById("root"));
root.render(<Main />);
```

```
        setMessage("Erreur lors de la récupération des données");

    }

};

// -----
// CREATE
// -----



const createTask = async (text) => {

    try {

        await fetch(API_URL, {

            method: "POST",

            headers: { "Content-Type": "application/json" },

            body: JSON.stringify({ text }),

        });

        setMessage("Tâche ajoutée !");

        fetchTasks();

    } catch {

        setMessage("Erreur lors de l'ajout");

    }

};

// -----
// UPDATE
// -----



const updateTask = async (id, newText) => {

    try {

        await fetch(` ${API_URL}/${id}` , {

            method: "PUT",
```

```
headers: { "Content-Type": "application/json" },  
body: JSON.stringify({ text: newText }),  
});  
  
setMessage("Tâche modifiée !");  
fetchTasks();  
} catch {  
setMessage("Erreur lors de la modification");  
}  
};  
  
// -----  
// DELETE  
// -----  
const deleteTask = async (id) => {  
try {  
await fetch(` ${API_URL}/${id}` , { method: "DELETE" });  
setMessage("Tâche supprimée !");  
fetchTasks();  
} catch {  
setMessage("Erreur lors de la suppression");  
}  
};  
  
useEffect(() => {  
fetchTasks();  
}, []);
```

```
return (
  <App
    tasks={tasks}
    createTask={createTask}
    updateTask={updateTask}
    deleteTask={deleteTask}
    message={message}
  />
);
```

```
ReactDOM.createRoot(document.getElementById("root")).render(<Main />);
```

✳️ Étape 3 — App.jsx

🎓 Objectif :

- Recevoir les fonctions API
- Gérer l'affichage global

📌 src/App.jsx

```
import CrudList from "./crud/CrudList";
import CrudForm from "./crud/CrudForm";
import "./crud/crud.css";

export default function App({
  tasks,
  createTask,
  updateTask,
  deleteTask,
  message,
```

```
) {  
  return (  
    <div className="container">  
      <h1>CRUD avec React + API</h1>  
  
      <CrudForm createTask={createTask} />  
  
      <CrudList  
        tasks={tasks}  
        updateTask={updateTask}  
        deleteTask={deleteTask}  
      />  
  
      <div className="message">{message}</div>  
    </div>  
  );  
}
```

✳️ Étape 4 — Formulaire d'ajout

🎓 Objectif :

- Ajouter une tâche
- Nettoyer le champ après ajout

📌 src/crud/CrudForm.jsx

```
import { useState } from "react";  
  
export default function CrudForm({ createTask }) {  
  const [text, setText] = useState("");
```

```
const handleSubmit = (e) => {
  e.preventDefault();
  if (!text.trim()) return;
  createTask(text);
  setText("");
};

return (
  <form onSubmit={handleSubmit} className="crud-form">
    <input
      type="text"
      value={text}
      placeholder="Nouvelle tâche..."
      onChange={(e) => setText(e.target.value)}
    />
    <button>+ Ajouter</button>
  </form>
);

}
```

✳ Étape 5 — Liste

🎓 Objectif :

- Afficher toutes les tâches

📌 src/crud/CrudList.jsx

```
import CrudItem from "./CrudItem";

export default function CrudList({ tasks, updateTask, deleteTask }) {
  return (
    <ul>
      {tasks.map((task) => (
        <li>
          <CrudItem task={task} updateTask={updateTask} deleteTask={deleteTask} />
        </li>
      ))}
    </ul>
  );
}
```

```
<div className="crud-list">  
  {tasks.map((t) => (  
    <CrudItem  
      key={t._id}  
      task={t}  
      updateTask={updateTask}  
      deleteTask={deleteTask}  
    />  
  ))}  
</div>  
);  
}
```

✳ Étape 6 — Item avec modification + suppression

🎓 Objectif :

- Modifier le texte d'une tâche
- Supprimer une tâche

📌 src/crud/CrudItem.jsx

```
import { useState } from "react";  
  
export default function CrudItem({ task, updateTask, deleteTask }) {  
  const [editMode, setEditMode] = useState(false);  
  const [newText, setNewText] = useState(task.text);  
  
  const saveEdit = () => {  
    updateTask(task._id, newText);  
    setEditMode(false);  
  };
```

```
return (  
  <div className="crud-item">  
    {editMode ? (  
      <>  
      <input  
        value={newText}  
        onChange={(e) => setNewText(e.target.value)}  
      />  
      <button onClick={saveEdit}>💾 </button>  
    ) : (  
      <>  
      <span>{task.text}</span>  
      <button onClick={() => setEditMode(true)}>📝 </button>  
    )  
  )}  
  
<button onClick={() => deleteTask(task._id)}>🗑 </button>  
</div>  
);  
}
```

✿ Étape 7 — Style CSS final

🎓 Objectif :

- Donner un look moderne
- Uniformiser l'affichage

📌 [src/crud/crud.css](#)

```
.container {  
    max-width: 600px;  
    margin: auto;  
    padding: 20px;  
    font-family: Arial;  
}
```

```
h1 {  
    text-align: center;  
}
```

```
.crud-form,  
.crud-item {  
    display: flex;  
    gap: 10px;  
    margin-bottom: 15px;  
}
```

```
.crud-form input,  
.crud-item input {  
    flex: 1;  
    padding: 8px;  
}
```

```
button {  
    padding: 8px 12px;  
    cursor: pointer;  
    border: none;
```

```
background: #0077ff;  
color: white;  
border-radius: 5px;  
transition: 0.2s;  
}
```

```
button:hover {  
background: #005fc;  
}
```

```
.crud-item {  
padding: 10px;  
background: #f4f4f4;  
border-radius: 6px;  
align-items: center;  
}
```

```
.message {  
margin-top: 20px;  
padding: 10px;  
background: #e3ffe3;  
text-align: center;  
border-radius: 6px;  
font-weight: bold;  
}
```

Si tu veux, je peux :

- Générer un **PDF pédagogique**
- Ajouter une **version GitHub**
- Ajouter une **version avec Axios**
- Ajouter une **version avec Zustand ou Redux Toolkit**

Voici la **version complète Axios**, propre, claire, prête à intégrer dans ton atelier.

Je modifie uniquement les parties nécessaires : **main.jsx**, en laissant les composants CRUD tels quels.

Version CRUD avec Axios

Installation

Dans ton projet :

```
npm install axios
```

1 — Configuration Axios dans main.jsx

-  Nous allons centraliser Axios, ajouter une instance configurée
 -  Les opérations CRUD deviennent plus propres
-

src/main.jsx (VERSION AXIOS COMPLÈTE)

```
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom/client";
import axios from "axios";
import App from "./App.jsx";

// 1 Créer une instance Axios
const api = axios.create({
  baseURL: "https://crudcrud.com/api/XXXXXXXXXX", // Mets ta clé ici
```

```
});  
  
function Main() {  
  const [tasks, setTasks] = useState([]);  
  const [message, setMessage] = useState("");  
  
  // -----  
  // READ  
  // -----  
  
  const fetchTasks = async () => {  
    try {  
      const res = await api.get("/tasks");  
      setTasks(res.data);  
    } catch (error) {  
      setMessage("Erreur de récupération des données");  
    }  
  };  
  
  // -----  
  // CREATE  
  // -----  
  
  const createTask = async (text) => {  
    try {  
      await api.post("/tasks", { text });  
      setMessage("Tâche ajoutée !");  
      fetchTasks();  
    } catch (error) {  
      setMessage("Erreur lors de l'ajout");  
    }  
  };  
}
```

```
}

};

// -----
// UPDATE

// -----

const updateTask = async (id, newText) => {
  try {
    await api.put(` /tasks/${id}` , { text: newText });
    setMessage("Tâche modifiée !");
    fetchTasks();
  } catch (error) {
    setMessage("Erreur lors de la modification");
  }
};

// -----
// DELETE

// -----

const deleteTask = async (id) => {
  try {
    await api.delete(` /tasks/${id}` );
    setMessage("Tâche supprimée !");
    fetchTasks();
  } catch (error) {
    setMessage("Erreur lors de la suppression");
  }
};
```

```
// Charger les données au démarrage

useEffect(() => {
  fetchTasks();
}, []);

return (
  <App
    tasks={tasks}
    createTask={createTask}
    updateTask={updateTask}
    deleteTask={deleteTask}
    message={message}
  />
);
}
```

```
ReactDOM.createRoot(document.getElementById("root")).render(<Main />);
```

✳️ 2 — Explication rapide pour les apprenants (optionnel pour ton cours)

✓ Pourquoi Axios ?

- Meilleure gestion des erreurs
- Plus simple pour configurer les headers
- api.get(), api.post() → plus lisible
- Possibilité d'utiliser des **intercepteurs** (token JWT)

✓ Pourquoi créer une instance Axios ?

Parce qu'après tu peux facilement ajouter :

```
api.defaults.headers.common["Authorization"] = "Bearer ...";
```

ou

```
api.interceptors.response.use(...)
```

✳️ 3 — Rien à changer dans tes composants CRUD

CrudList, CrudItem, CrudForm, crud.css, App.jsx restent identiques.

🎁 Bonus disponibles (choisis !)

Je peux t'ajouter :

- 🔥 **Version Axios + Intercepteurs (tokens JWT)**
- 🔥 **Version avec Redux Toolkit Query**
- 🔥 **Version avec Zustand**
- 🔥 **Version avec React Query (TanStack Query)**
- 🔥 **Version avec un backend Django API que tes étudiants consomment en React**

Crud Redux Version· typescript

```
// Version CRUD avec Redux Toolkit
```

```
useEffect(() => {
```

```
    dispatch(fetchTasks());
```

```
}, []);
```

```
const handleCreate = () => {
```

```
    dispatch(createTask(text)).then(() => dispatch(fetchTasks()));
```

```
    setText("");
```

```
};
```

```
const handleUpdate = () => {
```

```
    dispatch(updateTask({ id: editId, text })).then(() => dispatch(fetchTasks()));
```

```
    setEditId(null);
```

```
    setText("");
```

```
};
```

```
return (  
<div style={{ padding: '20px' }}>  
  <h2>CRUD avec Redux Toolkit</h2>  
  
  {message && <p style={{ color: 'green' }}>{message}</p>}
```

```
<input  
  type="text"  
  value={text}  
  placeholder="Nouvelle tâche"  
  onChange={(e) => setText(e.target.value)}  
/>
```

```
{editId ? (  
  <button onClick={handleUpdate}>Modifier</button>  
) : (  
  <button onClick={handleCreate}>Ajouter</button>  
)}
```

```
<ul>  
  {items.map((t) => (  
    <li key={t._id}>  
      {t.text}  
      <button onClick={() => { setEditId(t._id); setText(t.text); }}>  </button>  
      <button onClick={() => dispatch(deleteTask(t._id)).then(() =>  
        dispatch(fetchTasks()))}>  </button>
```

```
</li>
))>
</ul>
</div>
);

}

// -----
// src/crud/crud.css
ul {
list-style: none;
padding: 0;
}

li {
padding: 10px;
margin-bottom: 5px;
background: #f7f7f7;
border-radius: 6px;
display: flex;
justify-content: space-between;
align-items: center;
}

button {
margin-left: 10px;
cursor: pointer;
padding: 5px 10px;
```

```
border-radius: 6px;  
border: 1px solid #ccc;  
}
```

La version Redux est maintenant ajoutée dans le document.

Si tu veux, je peux aussi intégrer :

- Une version **Redux Toolkit Query (RTK Query)**
- Une architecture encore plus propre (features/, services/, hooks/)
- La version **Axios + Redux + Intercepteurs JWT**
- Un mini cours théorique sur Redux à ajouter au début du document
- Un schéma PNG explicatif (flow Redux → React)