Découverte d'un langage Full Stack : Formation complète

Formation complète avec tutoriels pratiques : architecture client/serveur, API REST avec Node.js/Express, authentification JWT, gestion des données Firebase, sécurité applicative. Du Hello World au déploiement.

Développement Web Programmation 35 min de lecture Niveau Intermédiaire

Document généré le 11/05/2026 à 12h47 · nouvy.fr/wiki/decouverte-langage-fullstack-formation

Sommaire

31 section(s) · 35 min de lecture

Objectifs opérationnels

Architecture client/serveur : les fondements

↳ Principe

↳ Pourquoi cette architecture ?

Veille, installation et premier programme

↳ 1.1 Veille technologique : choisir son stack

↳ 1.2 Installation de l'environnement Node.js

↳ 1.3 Créer le projet et le premier serveur Express

↳ 1.4 Structure recommandée d'un projet Full Stack

API REST et connexion aux données

↳ 2.1 Qu'est-ce qu'une API REST ?

↳ 2.2 Implémenter une API REST complète (Express)

↳ 2.3 Tester l'API avec cURL ou Postman

↳ 2.4 Connexion aux données : Firebase (BaaS)

↳ 2.4.1 Adapter les routes CRUD pour utiliser Firebase

↳ 2.5 Alternative : base de données relationnelle (MySQL/PostgreSQL)

Authentification et sécurité

↳ 3.1 Pourquoi l'authentification ?

↳ 3.2 Authentification par JWT (JSON Web Token)

↳ 3.2.1 Authentification avec Firebase Authentication

↳ 3.3 Middleware d'authentification

↳ 3.4 Bonnes pratiques de sécurité

Client : consommer l'API depuis le frontend

↳ 4.1 Appel API depuis JavaScript (fetch)

↳ 4.2 Exemple avec Vue.js (composition API)

Frontend React : inscription, connexion et appel de l'API avec Firebase

↳ 1. Création du frontend React (avec Vite)

↳ 2. Configuration Firebase côté React

↳ 3. Formulaire d'inscription / connexion React

↳ 4. Appeler l'API avec le Bearer Firebase

Ressources pour aller plus loin

Ce wiki vous guide pas à pas pour construire une **application Full Stack** de A à Z. Nous utiliserons **Node.js + Express** côté serveur et **JavaScript/Vue.js** côté client, mais les concepts s'appliquent à tout langage (Python/Django, Kotlin, Flutter, Go). L'objectif : comprendre l'architecture, implémenter une API REST, sécuriser avec l'authentification, et gérer les données.

Objectifs opérationnels

À l'issue de cette formation, vous serez capable de :

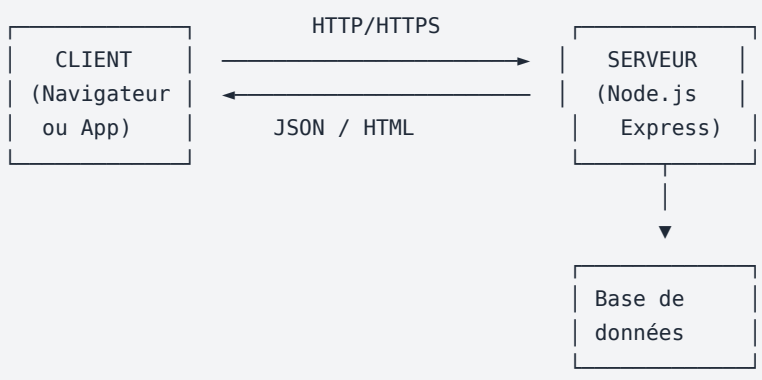
Compétence	Ce que vous saurez faire
Application client/serveur	Créer un serveur qui expose des endpoints et un client qui les consomme
API REST	Concevoir et implémenter une API RESTful (GET, POST, PUT, DELETE)
Authentification	Mettre en place une authentification sécurisée (JWT, sessions)
Gestion des données	Connecter l'app à une base de données ou une plateforme (Firebase, MySQL)

Architecture client/serveur : les fondements

Principe

Une application **client/serveur** sépare deux rôles :

- **Serveur** : héberge la logique métier, la base de données, expose des services via le réseau
- **Client** : interface utilisateur (navigateur, app mobile) qui envoie des requêtes et affiche les réponses



Copier

Pourquoi cette architecture ?

- **Séparation des responsabilités** : le client gère l'UI, le serveur la logique et les données
- **Sécurité** : les données sensibles restent côté serveur, le client ne reçoit que ce qui est nécessaire
- **Scalabilité** : on peut avoir plusieurs clients (web, mobile, desktop) partageant le même backend
- **Maintenabilité** : évolution indépendante du front et du back

---

Veille, installation et premier programme

1.1 Veille technologique : choisir son stack

Avant de coder, il faut **choisir un langage et un framework**. Critères à considérer :

Critère	Questions à se poser
Marché	Quels langages recrutent dans ma région ?
Communauté	Documentation, tutoriels, Stack Overflow ?
Écosystème	Bibliothèques, outils, hébergement ?
Courbe d'apprentissage	Combien de temps pour être productif ?
Actualité / maintenance	Est-ce activement maintenu ? Releases régulières, activité GitHub, doc officielle, compatibilité (Node LTS, navigateurs, etc.)

Exemples de stacks Full Stack populaires :

Stack	Backend	Frontend	Cas d'usage
MERN	Node.js + Express	React	Apps web dynamiques, APIs
MEVN	Node.js + Express	Vue.js	Apps légères, prototypes rapides
Django + Vue	Python Django	Vue.js	Apps métier, admin, data
Flutter + Firebase	Firebase (BaaS)	Flutter	Apps mobile cross-platform
Next.js	API Routes, Server Components	React	Full stack React, SSR, SEO, Vercel
Go + React	Go (Gin, Echo)	React	APIs performantes, microservices

1.2 Installation de l'environnement Node.js

Option recommandée : nvm (Node Version Manager)

nvm permet de gérer plusieurs versions de Node et de facilement en changer selon le projet.

macOS / Linux :

```
# Installer nvm
curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.40.1/install.sh | bash
# Redémarrer le terminal ou : source ~/.zshrc  (ou ~/.bashrc)
```

Copier

Windows :

1. Télécharger nvm-setup.exe sur la page des releases
2. Lancer l'installeur (désinstaller Node.js au préalable si déjà installé)
3. Ouvrir un nouveau terminal (PowerShell ou CMD)

```
**Utilisation (identique sur macOS, Linux et Windows) :**

```bash
# Installer Node 22
nvm install 22

# L'utiliser (pour cette session)
nvm use 22

# Lancer le projet
npm run dev
```

Copier

Pour définir Node 22 par défaut : `nvm alias default 22` (macOS/Linux) ou `nvm use 22` à chaque session (Windows).

1.3 Créer le projet et le premier serveur Express

```
mkdir mon-app-fullstack && cd mon-app-fullstack
npm init -y
npm install express
```

Copier

Créez le fichier `server.js` :

```
const express = require('express');
const app = express();
const PORT = 3000;

app.get("/", (req, res) => {
  res.json({ message: 'Hello World !' });
});

app.listen(PORT, () => {
  console.log(`Serveur démarré sur http://localhost:${PORT}`);
});
```

Copier

Lancez :
```

```
node server.js
```

Copier

Ouvrez `http://localhost:3000` dans le navigateur. Vous devriez voir `{"message":"Hello World !"}`. **Félicitations : votre premier serveur tourne.**

1.4 Structure recommandée d'un projet Full Stack

```
mon-app-fullstack/
├── server.js           # Point d'entrée du serveur
├── package.json
├── config/             # Configuration (DB, env)
├── routes/             # Routes API (users, products...)
├── controllers/        # Logique métier
├── models/             # Modèles de données (si BDD)
├── middleware/         # Auth, validation, logs
├── client/             # Frontend (Vue, React) - ou projet séparé
└── .env                # Variables d'environnement (jamais commité)
```

Copier

API REST et connexion aux données

2.1 Qu'est-ce qu'une API REST ?

REST (Representational State Transfer) est un style d'architecture pour les APIs. Les ressources sont identifiées par des **URL** et manipulées via des **méthodes HTTP** :

Méthode	Action	Exemple
GET	Lire	`GET /api/users` → liste des utilisateurs
POST	Créer	`POST /api/users` → créer un utilisateur
PUT	Remplacer	`PUT /api/users/1` → mettre à jour l'utilisateur 1
PATCH	Modifier partiellement	`PATCH /api/users/1` → modifier un champ
DELETE	Supprimer	`DELETE /api/users/1` → supprimer l'utilisateur 1

2.2 Implémenter une API REST complète (Express)

Exemple : API de gestion de **tâches** (todo list).

Installation des dépendances :

```
npm install express cors
```

Copier

`server.js` – API complète :

```
const express = require('express');
const cors = require('cors');
const app = express();
const PORT = 3000;

app.use(cors());
app.use(express.json());

// Stockage en mémoire (remplacé par une BDD en production)
let tasks = [
  { id: 1, title: 'Apprendre Express', done: false },
  { id: 2, title: 'Créer une API REST', done: false },
];

// GET /api/tasks - Liste toutes les tâches
app.get("/api/tasks", (req, res) => {
  res.json(tasks);
});

// GET /api/tasks/:id - Récupère une tâche par ID
app.get("/api/tasks/:id", (req, res) => {
  const task = tasks.find(t => t.id === parseInt(req.params.id));
  if (!task) return res.status(404).json({ error: 'Tâche non trouvée' });
  res.json(task);
});

// POST /api/tasks - Crée une nouvelle tâche
app.post("/api/tasks", (req, res) => {
  const { title } = req.body;
  if (!title) return res.status(400).json({ error: 'Le titre est requis' });
  const newTask = {
    id: tasks.length + 1,
    title,
    done: false,
  };
  tasks.push(newTask);
  res.status(201).json(newTask);
});

// PUT /api/tasks/:id - Met à jour une tâche
app.put("/api/tasks/:id", (req, res) => {
  const task = tasks.find(t => t.id === parseInt(req.params.id));
  if (!task) return res.status(404).json({ error: 'Tâche non trouvée' });
  task.title = req.body.title ?? task.title;
  task.done = req.body.done ?? task.done;
  res.json(task);
});

// DELETE /api/tasks/:id - Supprime une tâche
app.delete("/api/tasks/:id", (req, res) => {
  const index = tasks.findIndex(t => t.id === parseInt(req.params.id));
  if (index === -1) return res.status(404).json({ error: 'Tâche non trouvée' });
  tasks.splice(index, 1);
  res.status(204).send();
});

app.listen(PORT, () => console.log(`API sur http://localhost:${PORT}`));
```

2.3 Tester l'API avec cURL ou Postman

```
# Lister les tâches
curl http://localhost:3000/api/tasks

# Créer une tâche
curl -X POST http://localhost:3000/api/tasks
  -H "Content-Type: application/json"
  -d '{"title":"Ma nouvelle tâche"}'

# Mettre à jour
curl -X PUT http://localhost:3000/api/tasks/1
  -H "Content-Type: application/json"
  -d '{"done":true}'

# Supprimer
curl -X DELETE http://localhost:3000/api/tasks/1
```

Copier

2.4 Connexion aux données : Firebase (BaaS)

Firebase (Google) propose une base de données NoSQL temps réel et une authentification prête à l'emploi. Idéal pour prototyper rapidement.

Avant toute chose, il faut **créer un projet Firebase** et récupérer les paramètres de configuration :

1. Aller sur console.firebase.google.com et se connecter avec un compte Google
2. Créer un **nouveau projet** (nom du projet, acceptation des conditions)
3. Ajouter une **application Web** au projet (icône `</>`), lui donner un nom puis valider
4. Firebase affiche un bloc de code `const firebaseConfig = { ... }` → copier ces valeurs (apiKey, authDomain, projectId, etc.)
5. Mettre ces valeurs dans des **variables d'environnement** (ex : fichier `.env`) pour éviter de commiter des secrets, puis les lire via `process.env`

Exemple dans `.env` :

```
FIREBASE_API_KEY=xxx
FIREBASE_AUTH_DOMAIN=xxx
FIREBASE_PROJECT_ID=xxx
FIREBASE_STORAGE_BUCKET=xxx
FIREBASE_MESSAGING_SENDER_ID=xxx
FIREBASE_APP_ID=xxx
```

Copier

Installation (SDK Admin côté serveur) :

```
npm install firebase-admin
```

Copier

Télécharger la clé de service Firebase :

1. Aller dans la console Firebase → Paramètres du projet → Comptes de service
2. Générer une **clé privée** pour Firebase Admin SDK

3. Télécharger le fichier JSON (par exemple `service-account.json`)
4. Le placer dans un dossier non versionné, par exemple `config/keys/service-account.json` et **ne jamais le commiter** dans Git

Configuration (`config/firebase.js`) :

Créez un dossier `config/` à la racine de votre projet (au même niveau que `server.js` ou `index.js`), puis le fichier `config/firebase.js` :

```
// config/firebase.js
const admin = require("firebase-admin");
const serviceAccount = require("./ficher-cle.json");

admin.initializeApp({
  credential: admin.credential.cert(serviceAccount)
});

const db = admin.firestore();
module.exports = { admin, db };
```

Copier

Exemple : lire et écrire dans Firestore avec firebase-admin :

```
const { db } = require('./config/firebase');

// Lire les tâches
const snapshot = await db.collection('tasks').get();
const tasks = snapshot.docs.map(doc => ({ id: doc.id, ...doc.data() }));

// Créer une tâche
await db.collection('tasks').add({ title: 'Nouvelle tâche', done: false });
```

Copier

2.4.1 Adapter les routes CRUD pour utiliser Firebase

Dans les exemples précédents, les routes utilisaient un **tableau en mémoire** (`let tasks = [...]`). Pour travailler avec de vraies données persistées dans Firebase, il faut :

1. Importer Firestore dans le fichier de routes (ou `server.js`)
2. Remplacer les opérations sur le tableau par des appels à Firestore

Exemple d'implémentation CRUD complète avec Firestore via firebase-admin (dans `server.js` ou un fichier de routes dédié, par exemple `routes/tasks.js`) :

```
const express = require('express');
const cors = require('cors');
const { db } = require('./config/firebase');

const app = express();
app.use(cors());
app.use(express.json());

// GET /api/tasks - Liste toutes les tâches (READ)
app.get('/api/tasks', async (req, res) => {
  try {
    const snapshot = await db.collection('tasks').get();
```

```
    const tasks = snapshot.docs.map(d => ({ id: d.id, ...d.data() }));
    res.json(tasks);
  } catch (err) {
    res.status(500).json({ error: 'Erreur lors de la récupération des tâches' });
  }
});

// GET /api/tasks/:id - Récupère une tâche par ID (READ)
app.get('/api/tasks/:id', async (req, res) => {
  try {
    const taskSnap = await db.collection('tasks').doc(req.params.id).get();

    if (!taskSnap.exists) {
      return res.status(404).json({ error: 'Tâche non trouvée' });
    }

    res.json({ id: taskSnap.id, ...taskSnap.data() });
  } catch (err) {
    res.status(500).json({ error: 'Erreur lors de la récupération de la tâche' });
  }
});

// POST /api/tasks - Crée une nouvelle tâche (CREATE)
app.post('/api/tasks', async (req, res) => {
  try {
    const { title } = req.body;
    if (!title) {
      return res.status(400).json({ error: 'Le titre est requis' });
    }

    const createdRef = await db.collection('tasks').add({ title, done: false });
    res.status(201).json({ id: createdRef.id, title, done: false });
  } catch (err) {
    res.status(500).json({ error: 'Erreur lors de la création de la tâche' });
  }
});

// PUT /api/tasks/:id - Met à jour une tâche (UPDATE)
app.put('/api/tasks/:id', async (req, res) => {
  try {
    const ref = db.collection('tasks').doc(req.params.id);
    const snap = await ref.get();

    if (!snap.exists()) {
      return res.status(404).json({ error: 'Tâche non trouvée' });
    }

    const current = snap.data();
    const updated = {
      title: req.body.title ?? current.title,
      done: req.body.done ?? current.done,
    };

    await ref.update(updated);
    res.json({ id: snap.id, ...updated });
  } catch (err) {
    res.status(500).json({ error: 'Erreur lors de la mise à jour de la tâche' });
  }
});

// DELETE /api/tasks/:id - Supprime une tâche (DELETE)
app.delete('/api/tasks/:id', async (req, res) => {
  try {
    const ref = db.collection('tasks').doc(req.params.id);
    await ref.delete();
```

```
    res.status(204).send();
  } catch (err) {
    res.status(500).json({ error: 'Erreur lors de la suppression de la tâche' });
  }
});
```

Copier

En résumé, pour intégrer Firestore au CRUD :

- **CREATE** : `addDoc(collection(db, 'tasks'), { ... })`
- **READ (liste)** : `getDocs(collection(db, 'tasks'))`
- **READ (détail)** : `getDoc(doc(db, 'tasks', id))`
- **UPDATE** : `updateDoc(doc(db, 'tasks', id), { ... })`
- **DELETE** : `deleteDoc(doc(db, 'tasks', id))`

2.5 Alternative : base de données relationnelle (MySQL/PostgreSQL)

Avec **MySQL** et le driver `mysql2` :

```
npm install mysql2
```

Copier

```
const mysql = require('mysql2/promise');

const pool = mysql.createPool({
  host: process.env.DB_HOST || 'localhost',
  user: process.env.DB_USER || 'root',
  password: process.env.DB_PASSWORD,
  database: process.env.DB_NAME || 'mon_app',
});

// Récupérer toutes les tâches
const [rows] = await pool.execute('SELECT * FROM tasks');
res.json(rows);

// Créer une tâche
const [result] = await pool.execute(
  'INSERT INTO tasks (title, done) VALUES (?, ?)',
  [req.body.title, false]
);
res.status(201).json({ id: result.insertId, ...req.body });
```

Copier

Authentification et sécurité

3.1 Pourquoi l'authentification ?

Sans authentification, n'importe qui peut appeler votre API et modifier les données. L'authentification permet de :

- **Identifier** l'utilisateur (qui est-il ?)
- **Autoriser** l'accès aux ressources (a-t-il le droit ?)

3.2 Authentification par JWT (JSON Web Token)

Le **JWT** est un token signé contenant des informations (user id, rôle, expiration). Le client l'envoie dans l'en-tête `Authorization` à chaque requête.

Installation :

```
npm install jsonwebtoken bcrypt
```

Copier

Inscription (POST /api/auth/register) :

```
const bcrypt = require('bcrypt');
const jwt = require('jsonwebtoken');

app.post("/api/auth/register", async (req, res) => {
  const { email, password } = req.body;
  if (!email || !password) {
    return res.status(400).json({ error: 'Email et mot de passe requis' });
  }
  // Vérifier si l'utilisateur existe déjà (requête BDD)
  // Hasher le mot de passe
  const hashedPassword = await bcrypt.hash(password, 10);
  // Sauvegarder en BDD (users table)
  // Générer le JWT
  const token = jwt.sign(
    { userId: newUser.id },
    process.env.JWT_SECRET || 'votre-secret-tres-long',
    { expiresIn: '7d' }
  );
  res.status(201).json({ token, user: { id: newUser.id, email } });
});
```

Copier

Connexion (POST /api/auth/login) :

```
app.post("/api/auth/login", async (req, res) => {
  const { email, password } = req.body;
  // Récupérer l'utilisateur en BDD
  const user = await findUserByEmail(email);
  if (!user) return res.status(401).json({ error: 'Identifiants invalides' });
  const valid = await bcrypt.compare(password, user.password);
  if (!valid) return res.status(401).json({ error: 'Identifiants invalides' });
  const token = jwt.sign(
    { userId: user.id },
    process.env.JWT_SECRET,
    { expiresIn: '7d' }
  );
  res.json({ token, user: { id: user.id, email: user.email } });
});
```

Copier

3.2.1 Authentification avec Firebase Authentication

Si vous ne voulez pas gérer vous-même le stockage des utilisateurs et le hachage des mots de passe, vous pouvez déléguer l'authentification à **Firebase Authentication** et ne garder votre API Node.js que pour la logique métier.

Principe :

1. Côté **front** (ou client), l'utilisateur se connecte via Firebase (email/mot de passe, Google, etc.).
2. Firebase renvoie un **ID token** (JWT signé par Google).
3. Le front envoie ce token dans l'en-tête `Authorization: Bearer <idToken>` à votre API.
4. Côté **backend**, vous vérifiez ce token avec le SDK Admin de Firebase avant d'autoriser l'accès.

Installation du SDK Admin dans l'API :

```
npm install firebase-admin
```

Copier

Initialisation (dans un fichier `firebaseAdmin.js` à la racine du projet, au même niveau que `server.js`) :

```
const admin = require('firebase-admin');

admin.initializeApp({
  credential: admin.credential.applicationDefault(),
});

module.exports = { admin };
```

Copier

Middleware d'authentification Firebase :

```
const { admin } = require('./firebaseAdmin');

const firebaseAuth = async (req, res, next) => {
  const authHeader = req.headers.authorization;
  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'Token Firebase manquant' });
  }

  const idToken = authHeader.split()[1];

  try {
    const decoded = await admin.auth().verifyIdToken(idToken);
    req.user = decoded; // contient uid, email, etc.
    next();
  } catch (err) {
    return res.status(401).json({ error: 'Token Firebase invalide ou expiré' });
  }
};

// Exemple de route protégée avec Firebase Auth
app.get("/api/profile", firebaseAuth, (req, res) => {
  res.json({
    uid: req.user.uid,
    email: req.user.email,
  });
});
```

Copier

3.3 Middleware d'authentification

Protéger les routes qui nécessitent une connexion :

```
const authMiddleware = (req, res, next) => {
  const authHeader = req.headers.authorization;
  if (!authHeader || !authHeader.startsWith('Bearer ')) {
    return res.status(401).json({ error: 'Token manquant' });
  }
  const token = authHeader.split()[1];
  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.userId = decoded.userId;
    next();
  } catch (err) {
    return res.status(401).json({ error: 'Token invalide ou expiré' });
  }
};

// Routes protégées
app.get("/api/me", authMiddleware, (req, res) => {
  // req.userId contient l'ID de l'utilisateur connecté
  res.json({ userId: req.userId });
});

app.post("/api/tasks", authMiddleware, (req, res) => {
  // Créer une tâche liée à req.userId
});
```

Copier

3.4 Bonnes pratiques de sécurité

Pratique	Pourquoi
Hasher les mots de passe (bcrypt)	En cas de fuite BDD, les mots de passe restent illisibles
HTTPS en production	Chiffrement des données en transit
Variables d'environnement	Ne jamais mettre de secrets dans le code (JWT_SECRET, DB_PASSWORD)
Validation des entrées	Éviter les injections et les données malformées
CORS configuré	Limiter les origines autorisées (pas * en prod)
Rate limiting	Limiter les tentatives de connexion (brute force)

Client : consommer l'API depuis le frontend

4.1 Appel API depuis JavaScript (fetch)

```
// Récupérer les tâches
const response = await fetch("http://localhost:3000/api/tasks");
const tasks = await response.json();

// Créer une tâche (avec authentification)
const response = await fetch("http://localhost:3000/api/tasks", {
  method: "POST",
  headers: {
    "Content-Type": "application/json",
    "Authorization": `Bearer ${token}`,
  },
  body: JSON.stringify({ title: "Ma tâche" }),
});
const newTask = await response.json();
```

Copier

4.2 Exemple avec Vue.js (composition API)

```
<script setup>
import { ref, onMounted } from "vue";

const tasks = ref([]);
const token = localStorage.getItem("token");

async function loadTasks() {
  const res = await fetch("http://localhost:3000/api/tasks", {
    headers: { Authorization: `Bearer ${token}` },
  });
  tasks.value = await res.json();
}

async function addTask(title) {
  await fetch("http://localhost:3000/api/tasks", {
    method: "POST",
    headers: {
      "Content-Type": "application/json",
      "Authorization": `Bearer ${token}`,
    },
    body: JSON.stringify({ title }),
  });
  loadTasks();
}

onMounted(loadTasks);
</script>

<template>
  <ul>
    <li v-for="task in tasks" :key="task.id">{{ task.title }}</li>
  </ul>
</template>
```

Copier

Frontend React : inscription, connexion et appel de l'API avec Firebase

Pour tester l'API avec authentification Firebase, vous pouvez créer un **frontend React** simple (par exemple avec Vite) qui :

- gère **inscription** et **connexion** via Firebase Auth,
- récupère l'`idToken` de l'utilisateur connecté,
- appelle votre API Node.js en ajoutant `Authorization: Bearer <idToken>` dans les en-têtes.

1. Création du frontend React (avec Vite)

Dans un dossier à côté du backend, créez le projet directement dans le dossier courant :

```
npx create-vite .
```

Copier

Puis, dans l’assistant :

- **Select a framework** : `React`
- **Select a variant** : `TypeScript`
- **Install with npm and start now?** : `Yes`

Installez Firebase côté frontend :

```
npm install firebase
```

Copier

2. Configuration Firebase côté React

Créez `src/config/firebase.ts` :

```
import { initializeApp } from "firebase/app";
import { getAuth } from "firebase/auth";

const firebaseConfig = {
  apiKey: "VOTRE_API_KEY",
  authDomain: "votre-projet.firebaseapp.com",
  projectId: "votre-projet",
  // autres champs si nécessaires
};

const app = initializeApp(firebaseConfig);
export const auth = getAuth(app);
```

Copier

Les valeurs viennent de la console Firebase (même projet que pour le backend).

Important (sinon erreur `auth/configuration-not-found`) : activez l'authentification Email/Password dans Firebase :

1. Console Firebase → **Authentication**
2. Onglet **Sign-in method**
3. Activer **Email/Password**
4. Sauvegarder

3. Formulaire d'inscription / connexion React

Dans `src/AuthForm.tsx` :

```
import { useState } from "react";
import { auth } from "./firebase";
import {
  createUserWithEmailAndPassword,
  signInWithEmailAndPassword,
} from "firebase/auth";

export function AuthForm() {
  const [email, setEmail] = useState("");
  const [password, setPassword] = useState("");
  const [message, setMessage] = useState("");

  async function handleSignup() {
    try {
      await createUserWithEmailAndPassword(auth, email, password);
      setMessage("Inscription réussie, vous pouvez maintenant vous connecter.");
    } catch (err) {
      setMessage("Erreur d'inscription.");
    }
  }

  async function handleLogin() {
    try {
      await signInWithEmailAndPassword(auth, email, password);
      setMessage("Connexion réussie.");
    } catch (err) {
      setMessage("Erreur de connexion.");
    }
  }

  return (
    <div>
      <h2>Auth Firebase</h2>
      <input
        placeholder="Email"
        value={email}
        onChange={(e) => setEmail(e.target.value)}
      />
      <input
        type="password"
        placeholder="Mot de passe"
        value={password}
        onChange={(e) => setPassword(e.target.value)}
      />
      <button onClick={handleSignup}>Inscription</button>
      <button onClick={handleLogin}>Connexion</button>
      <p>{message}</p>
    </div>
  );
}
```

Copier

4. Appeler l'API avec le Bearer Firebase

Toujours côté React, créez un petit composant qui charge les tâches depuis l'API :

```
import { useEffect, useState } from "react";
import { auth } from "./firebase";

export function Tasks() {
  const [tasks, setTasks] = useState([]);
  const [error, setError] = useState("");

  async function loadTasks() {
    try {
      const user = auth.currentUser;
      if (!user) {
        setError("Vous devez être connecté.");
        return;
      }
      const idToken = await user.getIdToken();

      const res = await fetch("http://localhost:3000/api/tasks", {
        headers: {
          Authorization: `Bearer ${idToken}`,
        },
      });

      if (!res.ok) {
        setError("Erreur API");
        return;
      }

      const data = await res.json();
      setTasks(data);
    } catch {
      setError("Erreur lors du chargement des tâches.");
    }
  }

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

  return (
    <div>
      <h2>Liste des tâches</h2>
      {error && <p>{error}</p>}
      <ul>
        {tasks.map((t: any) => (
          <li key={t.id}>{t.title}</li>
        ))}
      </ul>
    </div>
  );
}
```

Copier

Dans `src/App.tsx` :

```
import { AuthForm } from "./AuthForm";
import { Tasks } from "./Tasks";

function App() {
  return (
    <div>
      <AuthForm />
      <Tasks />
    </div>
  );
}

export default App;
```

Copier

Avec cette configuration :

- React gère l'inscription / connexion avec Firebase Auth.
- Après connexion, React récupère l'`idToken` et l'envoie dans `Authorization: Bearer <idToken>`.
- Le backend utilise le middleware `firebaseAuth` basé sur `firebase-admin` pour vérifier le token et autoriser l'accès aux routes CRUD.

Ressources pour aller plus loin

- **Express** : expressjs.com
- **Vue.js** : vuejs.org
- **Firebase** : firebase.google.com/docs
- **JWT** : jwt.io
- **REST API Design** : restfulapi.net