Guide

BAHIJ OUMAIMA,CHAOU ABDERRAHMANE, ELKAOURI AISSAME

March 2025

1 Guide détaillé d'utilisation de l'API HBase avec Node.js et React

1.1 Installation et démarrage de HBase avec Docker

Dans cette section, nous allons expliquer en détail comment télécharger, configurer et démarrer un conteneur Docker contenant HBase. L'utilisation de Docker permet de déployer rapidement un environnement HBase sans avoir à installer manuellement toutes les dépendances et configurations.

1.1.1 Télécharger l'image Docker de HBase

La commande suivante télécharge l'image `harisekhon/hbase` depuis Docker Hub. Cette image est configurée pour lancer HBase avec une configuration de base.

```
docker pull harisekhon/hbase
```

Explication : *docker pull* récupère l'image depuis le registre distant (ici Docker Hub) et la stocke localement. Vous pourrez ensuite l'utiliser pour lancer un ou plusieurs conteneurs.

1.1.2 Lancer le conteneur HBase

Pour démarrer un conteneur HBase, vous pouvez utiliser la commande suivante :

```
docker run -d --name hbase_container -p 8080:8080 -p 9091:9091 harisekhon/hbase
```

Explications :

- `-d` lance le conteneur en arrière-plan (mode détaché).
- `--name hbase_container` donne un nom facile à retenir au conteneur.
- `-p 8080:8080` publie le port 8080 du conteneur (par exemple pour l'interface web de HBase) sur le port 8080 de la machine hôte.

- `-p 9091:9091` publie le port 9091 qui sera utilisé pour le service Thrift (qui permet aux clients de se connecter à HBase via des API REST ou des clients compatibles Thrift).
- Enfin, `harisekhon/hbase` est le nom de l'image que nous avons téléchargée.

1.1.3 Vérifier l'état du conteneur

Après avoir lancé le conteneur, vous pouvez vérifier qu'il est bien en cours d'exécution avec la commande suivante :

```
docker ps
```

Cette commande liste tous les conteneurs actifs et vous permet de vérifier que votre conteneur HBase est opérationnel (il doit apparaître dans la liste avec le nom que vous lui avez attribué, par exemple `hbase_container`).

1.1.4 Démarrer le service Thrift de HBase

Pour permettre aux applications externes d'interagir avec HBase via des protocoles comme Thrift, il est nécessaire de démarrer le service Thrift dans le conteneur. Vous pouvez le faire avec la commande suivante :

```
docker exec -it hbase_container hbase thrift start -p 9091
```

Explications :

- `docker exec` permet d'exécuter une commande dans un conteneur déjà en cours d'exécution.
- `-it` ouvre un terminal interactif pour pouvoir suivre les éventuels messages.
- `hbase thrift start -p 9091` lance le service Thrift de HBase sur le port 9091. Ce service facilite la communication avec HBase depuis divers clients (par exemple, via une API REST).

Grâce à ces étapes, vous disposez désormais d'un environnement HBase fonctionnel dans un conteneur Docker, accessible localement via les ports 8080 (pour l'interface web) et 9091 (pour Thrift). Cette configuration est idéale pour des tests ou pour un développement local rapide.
<

1.2 Manipulation de données dans HBase via le Shell

Dans cette section, nous allons voir comment interagir directement avec HBase en utilisant son shell. Le shell HBase permet d'exécuter des commandes pour insérer, mettre à jour, consulter ou supprimer des données dans vos tables. Cela est particulièrement utile pour tester vos requêtes ou pour administrer la base en mode interactif.

1.2.1 Création de la table et insertion d'un utilisateur

Avant de manipuler les données, assurez-vous que la table existe. Par exemple, pour créer une table nommée `users` avec une colonne famille `info` :

```
create 'users', 'info'
```

Ensuite, pour insérer un utilisateur, on utilise la commande `put`. Voici comment insérer un enregistrement pour l'utilisateur avec l'identifiant `1` :

```
echo -e "put 'users', '1', 'info:name', 'Alice'" | hbase shell
echo -e "put 'users', '1', 'info:email', 'alice@example.com'" | hbase shell
echo -e "put 'users', '1', 'info:age', '25'" | hbase shell
```

Explications :

- `put 'users', '1', 'info:name', 'Alice'` insère la valeur "Alice" dans la colonne `name` de la famille `info` pour la ligne dont la clé est `1`.
- Chaque commande est envoyée via le pipe (|) au shell HBase, ce qui permet d'exécuter les commandes non-interactivement.

1.2.2 Insertion de plusieurs utilisateurs en une seule commande

Il est possible de combiner plusieurs commandes `put` dans une seule exécution. Par exemple, pour insérer les données de l'utilisateur avec l'identifiant `2` :

```
echo -e "put 'users', '2', 'info:name', 'Bob'
put 'users', '2', 'info:email', 'bob@example.com'
put 'users', '2', 'info:age', '30'" | hbase shell
```

Conseil : L'utilisation d'un retour à la ligne (`\n`) permet d'exécuter plusieurs commandes dans une seule invocation, ce qui peut simplifier le déploiement de scripts batch.

1.2.3 Mise à jour d'un utilisateur

Pour mettre à jour une valeur existante, on utilise également la commande `put`. Par exemple, pour mettre à jour l'âge de l'utilisateur `1` à `26` :

```
echo -e "put 'users', '1', 'info:age', '26'" | hbase shell
```

HBase écrase la valeur existante dans la cellule spécifiée par cette commande. Ainsi, l'ancien âge est remplacé par le nouveau.

1.2.4 Vérification et consultation des données

Pour vérifier que les données ont bien été insérées ou mises à jour, vous pouvez utiliser :

```
# Affiche l'ensemble des lignes de la table "users"
echo -e "scan 'users'" | hbase shell

# Récupère la ligne avec la clé "1" pour vérifier les informations spécifiques d'un utilisat
echo -e "get 'users', '1'" | hbase shell
```

Explications :

- `scan` parcourt toute la table et affiche chaque ligne ainsi que ses colonnes et valeurs.
- `get` permet de récupérer une seule ligne (identifiée par la clé) et d'afficher ses cellules.

1.2.5 2.5 Suppression d'un utilisateur

Pour supprimer un enregistrement entier (toutes les cellules d'une ligne), on utilise la commande `deleteall`. Par exemple, pour supprimer toutes les informations de l'utilisateur avec la clé `1` :

```
echo -e "deleteall 'users', '1'" | hbase shell
```

Cette commande supprime toutes les colonnes associées à la ligne `1` dans la table `users`.

En résumé, le shell HBase offre une interface simple et puissante pour interagir avec la base de données. Vous pouvez automatiser ces commandes en les regroupant dans des scripts pour effectuer des opérations de maintenance, de migration ou pour tester vos applications avant de passer en production.

1.3 Création d'une API REST avec Node.js

Dans cette section, nous allons créer un serveur Node.js à l'aide d'Express afin d'exposer une API REST. Cette API servira d'intermédiaire pour communiquer avec HBase via son API REST. Nous verrons notamment comment encoder les données en Base64 (exigence de HBase), gérer les erreurs et définir des endpoints pour ajouter et récupérer des utilisateurs.

1.3.1 Installation et configuration

Assurez-vous d'installer les dépendances nécessaires dans votre dossier `server` :

```
// Dans le dossier server, exécutez ces commandes :
npm init -y
npm install express axios cors
npm install nodemon --save-dev
```

Ici, nous installons :

- `express` : pour créer notre serveur web.
- `axios` : pour effectuer des requêtes HTTP vers l'API REST de HBase.
- `cors` : pour autoriser les requêtes cross-origin depuis notre application React.
- `nodemon` : pour redémarrer automatiquement le serveur lors de modifications du code (en mode développement).

1.3.2 Code complet du serveur Node.js

Voici le code complet, avec des commentaires pour expliquer chaque étape :

```
// Importation des modules nécessaires
const express = require("express");
const axios = require("axios");  // Pour effectuer des requêtes HTTP vers l'API REST de HBas
const cors = require("cors");

// Création de l'application Express
const app = express();

// Configuration des middlewares
// cors() autorise les requêtes cross-origin, utile si votre frontend (ex. React) tourne sur
app.use(cors());
// express.json() permet de parser automatiquement les requêtes avec un body en JSON
app.use(express.json());

/**
 * Endpoint POST /users
 * Permet d'ajouter ou de mettre à jour un utilisateur dans HBase.
 * Les données (id, name, email, age) sont attendues dans le corps de la requête.
 * Chaque valeur est encodée en Base64, conformément aux exigences de HBase.
 */
app.post("/users", async (req, res) => {
  try {
    // Extraction des données de la requête
    const { id, name, email, age } = req.body;

    // Construction de l'objet de données dans le format attendu par l'API REST de HBase
    const data = {
```

```
      Row: [
        {
          key: Buffer.from(id).toString("base64"),
          Cell: [
            {
              column: Buffer.from("info:name").toString("base64"),
              $: Buffer.from(name).toString("base64"),
            },
            {
              column: Buffer.from("info:email").toString("base64"),
              $: Buffer.from(email).toString("base64"),
            },
            {
              column: Buffer.from("info:age").toString("base64"),
              $: Buffer.from(age.toString()).toString("base64"),
            },
          ],
        },
      ],
    };

    // Envoi d'une requête PUT à l'API REST de HBase pour insérer ou mettre à jour l'utilisa
    await axios.put(`http://127.0.0.1:8080/users/${id}`, data, {
      headers: { "Content-Type": "application/json" }
    });

    // Réponse en cas de succès
    res.status(201).send(`Utilisateur ${id} ajouté.`);
  } catch (err) {
    // Gestion de l'erreur : log dans la console et envoi d'une réponse d'erreur au client
    console.error("Erreur lors de l'ajout de l'utilisateur :", err);
    res.status(500).send(`Erreur : ${err.message}`);
  }
});

/**
 * Endpoint GET /users
 * Permet de récupérer la liste des utilisateurs présents dans HBase.
 * Utilise un scanner HBase pour lire plusieurs lignes.
 */
app.get("/users", async (req, res) => {
  try {
    // Création d'un scanner en envoyant une requête PUT à l'API REST de HBase
    const scannerResponse = await axios.put(
      "http://127.0.0.1:8080/users/scanner",
      { batch: 10 }, // Spécifie le nombre maximal de lignes à récupérer par lot
```

```
      { headers: { Accept: "application/json" } }
    );
    // Extraction de l'ID du scanner à partir de l'en-tête 'location'
    const scannerId = scannerResponse.headers.location.split("/").pop();

    // Récupération des données via le scanner avec une requête GET
    const dataResponse = await axios.get(`http://127.0.0.1:8080/users/scanner/${scannerId}`,
      headers: { Accept: "application/json" }
    });

    // Envoi de la réponse JSON contenant les données récupérées
    res.json(dataResponse.data);
  } catch (err) {
    console.error("Erreur lors de la récupération des utilisateurs :", err);
    res.status(500).send(`Erreur : ${err.message}`);
  }
});

/**
 * Démarrage du serveur
 * Le serveur écoute sur le port 7070.
 */
app.listen(7070, () => {
  console.log("Serveur Node.js en cours d'exécution sur le port 7070.");
});
```

1.3.3 Explications détaillées

Importation et initialisation :
Nous importons les modules `express`, `axios` et `cors` pour créer un serveur web, effectuer des requêtes HTTP vers HBase et gérer les problèmes de CORS.

Middlewares :
- `app.use(cors())` permet aux requêtes provenant d'autres origines (comme votre application React sur le port 3000) d'accéder à l'API.
- `app.use(express.json())` convertit automatiquement le corps des requêtes au format JSON en objet JavaScript.

Endpoint POST /users :
- Les informations de l'utilisateur sont extraites du corps de la requête.
- Chaque donnée (identifiant, nom, email, âge) est encodée en Base64 à l'aide de `Buffer.from(...).toString("base64")`. HBase exige cet encodage pour le stockage.
- Une requête PUT est envoyée à l'URL de HBase pour insérer ou mettre à jour la ligne correspondant à l'utilisateur dans la table `users`.
- En cas de succès, le serveur renvoie un message de confirmation ; en cas d'erreur, il renvoie un message d'erreur avec le code 500.

Endpoint GET /users :

- Un scanner est créé en envoyant une requête PUT à l'endpoint `/users/scanner` avec un paramètre `batch` indiquant le nombre de lignes à récupérer.
- L'ID du scanner est récupéré depuis l'en-tête `location` de la réponse.
- Une requête GET utilisant cet ID permet de lire les données du scanner et de renvoyer le résultat en JSON.

Démarrage du serveur :
Le serveur écoute sur le port 7070, ce qui permet aux clients (comme une application React) d'interagir avec l'API en accédant à `http://localhost:7070`.

1.4 Création d'une interface utilisateur avec React

Dans cette section, nous allons créer une application React qui consomme l'API REST de notre serveur Node.js. L'application se connectera à l'API, récupérera des données et les affichera dynamiquement dans l'interface. Vous pourrez étendre cette base pour ajouter, modifier ou supprimer des utilisateurs.

1.4.1 Initialisation de l'application React

Dans le dossier `client`, nous avons généré l'application avec `create-react-app`. La structure du projet comprend notamment le fichier `src/App.js`, où nous allons écrire notre composant principal.

1.4.2 Code complet du composant App.js

[] // Importation des hooks React et du module Axios pour les requêtes HTTP import React, { useState, useEffect } from "react"; import axios from "axios"; import "./App.css";

/** * Composant principal de l'application. * Il récupère un message depuis le backend et l'affiche dans un élément h1. */ function App() { // Déclaration d'un état 'message' pour stocker le message récupéré const [message, setMessage] = useState("");

/** * useEffect se déclenche lors du montage du composant. * Il effectue une requête GET vers le backend pour récupérer le message. */ useEffect(() => { // Utilisation d'Axios pour appeler l'API REST sur le port 7070 axios.get("http://localhost:7070/message") .then((response) => { // Mettre à jour l'état avec le message reçu setMessage(response.data.message); }) .catch((error) => { // En cas d'erreur, afficher l'erreur dans la console console.error("Erreur lors de la récupération du message :", error); }); }, []); // Le tableau vide [] assure que l'effet se déclenche uniquement lors du montage

// Rendu du composant : affichage du message dans une balise h1 return ({message}); }

// Exportation du composant pour être utilisé dans d'autres parties de l'application export default App;

1.4.3 4.3 Explications détaillées du code

Importation : Nous importons `useState` et `useEffect` depuis React pour gérer l'état et les effets de bord, ainsi que `axios` pour effectuer des requêtes HTTP. Le fichier `App.css` est également importé pour la mise en forme.

useState : La fonction `useState` initialise une variable d'état `message` avec une chaîne vide. Cette variable sera mise à jour lorsque nous recevrons la réponse du backend.

useEffect : Le hook `useEffect` exécute une requête HTTP dès le montage du composant (grâce au tableau de dépendances vide `[]`). Il utilise Axios pour envoyer une requête GET à `http://localhost:7070/message`. Si la requête réussit, le message est stocké dans l'état grâce à `setMessage`; en cas d'erreur, celle-ci est affichée dans la console.

Rendu du composant : La fonction `App` retourne un élément JSX qui affiche le contenu de la variable `message` dans une balise `<h1>`. Vous pouvez étendre cette structure en ajoutant des formulaires ou d'autres composants pour interagir davantage avec votre API.

Exportation : Enfin, le composant est exporté afin qu'il puisse être utilisé dans d'autres parties de l'application React.

1.4.4 Structure du projet React

```
client/
 public/
 src/
    App.css       // Styles de l'application
    App.js        // Composant principal (décrit ci-dessus)
    index.js      // Point d'entrée de l'application React
    ...           // Autres composants et fichiers
 package.json      // Dépendances et scripts pour le projet
```

En résumé, cette section vous guide dans la création d'une interface utilisateur avec React. Elle met en œuvre une logique de base pour récupérer des données via une API REST et les afficher, tout en laissant la possibilité d'étendre l'application pour des opérations CRUD supplémentaires.

1.5 Structure du projet

Le projet est organisé en deux parties distinctes pour séparer les responsabilités du frontend (interface utilisateur) et du backend (serveur API). Cette séparation facilite le développement, le déploiement et la maintenance.

1.5.1 Arborescence du projet

```
project-root/
 client/         # Application React (frontend)
    public/     # Fichiers statiques et index.html
```

```
    src/        # Code source de l'application React
       App.js  # Composant principal de l'interface
       App.css # Fichier de style principal
       index.js# Point d'entrée de l'application React
    package.json# Dépendances et scripts pour le frontend
 server/         # Serveur Node.js (backend)
     server.js   # Fichier principal du serveur Express
     package.json# Dépendances et scripts pour le backend
```

Chaque dossier contient son propre fichier `package.json`, ce qui permet de gérer indépendamment les dépendances et scripts spécifiques au frontend et au backend.

1.5.2 Scripts de démarrage et commandes

Pour lancer l'application, il est nécessaire de démarrer séparément le serveur backend et l'application frontend.

- **Backend :** Dans le dossier `server`, utilisez la commande suivante pour démarrer le serveur en mode développement (avec `nodemon`) :
  `npm run dev`
- **Frontend :** Dans le dossier `client`, lancez l'application React avec la commande :
  `npm start`

Une fois ces commandes exécutées, le serveur backend sera accessible sur le port 7070 et l'application React sera lancée sur le port 3000, permettant ainsi d'interconnecter les deux parties via l'API REST.

1.5.3 Gestion des dépendances et variables d'environnement

Chaque partie du projet (frontend et backend) possède son propre fichier `package.json` pour définir les dépendances et les scripts utiles. Vous pouvez également utiliser un fichier `.env` dans chaque dossier pour configurer des paramètres tels que les ports ou les URLs de connexion.

Par exemple, vous pourriez définir dans le dossier `server` un fichier `.env` contenant :

```
PORT=7070
HBASE_API_URL=http://127.0.0.1:8080
```

Et dans votre code Node.js, utiliser ces variables avec `process.env` pour une meilleure portabilité.

1.5.4 Points clés et avantages

- **Séparation claire des responsabilités :** Le frontend gère l'interface utilisateur tandis que le backend se charge de la logique métier et de l'interaction avec HBase.
- **Modularité :** Une structure distincte facilite la maintenance et l'évolution du projet.
- **Scripts dédiés :** Les commandes `npm run dev` et `npm start` simplifient le démarrage et la surveillance de chaque partie de l'application.
- **Flexibilité :** La possibilité d'utiliser des fichiers `.env` permet de configurer facilement l'environnement sans modifier le code source.

Cette organisation modulaire est idéale pour des applications web modernes où le frontend et le backend peuvent évoluer indépendamment tout en communiquant efficacement via des API REST.

1.6 Démarrage et tests

Dans cette section, nous allons expliquer comment démarrer le serveur backend et l'application frontend, puis effectuer quelques tests pour vérifier que tout fonctionne correctement.

1.6.1 Démarrage du serveur backend

Pour démarrer le serveur Node.js (backend), ouvrez un terminal, accédez au dossier `server` et exécutez la commande suivante :

```
cd server
npm run dev
```

Explications :

- `cd server` vous positionne dans le répertoire du backend.
- `npm run dev` lance le serveur en mode développement grâce à `nodemon`, qui surveille automatiquement les modifications dans le fichier `server.js` et redémarre le serveur au besoin.
- Le serveur écoute sur le port 7070, comme défini dans le code.

1.6.2 Démarrage de l'application frontend

Pour démarrer l'application React (frontend), ouvrez un autre terminal, accédez au dossier `client` et lancez la commande suivante :

```
cd client
npm start
```

Explications :

- `npm start` démarre l'application React en mode développement.
- L'application se lance par défaut sur le port 3000 et s'ouvre automatiquement dans votre navigateur.

1.6.3 Vérification du bon fonctionnement

Test du backend :
Ouvrez votre navigateur et accédez à `http://localhost:7070/message`. Vous devriez voir une réponse JSON ressemblant à :

```
{
  "message": "Hello from server!"
}
```

Test du frontend :
L'application React récupère automatiquement ce message depuis le backend et l'affiche dans une balise <h1> sur la page d'accueil.

Test de l'API REST :
Vous pouvez également tester l'API en utilisant Postman, cURL ou un autre outil similaire. Par exemple, pour ajouter un utilisateur, envoyez une requête POST à `http://localhost:7070/users` avec un corps JSON comme celui-ci :

```
{
  "id": "3",
  "name": "Charlie",
  "email": "charlie@example.com",
  "age": "28"
}
```

La réponse devrait indiquer que l'utilisateur a été ajouté avec succès. Ensuite, vous pouvez envoyer une requête GET à `http://localhost:7070/users` pour récupérer la liste des utilisateurs.

1.6.4 Conseils supplémentaires

Configuration des URL :
Vérifiez que l'URL du backend utilisée dans votre application React (ou dans vos tests) pointe bien vers `http://localhost:7070` afin d'assurer une communication correcte entre le frontend et le backend.

Gestion des erreurs :
Si une requête échoue, consultez la console du serveur ou du navigateur pour obtenir des détails sur l'erreur. Ces informations vous aideront à identifier et corriger les problèmes éventuels.

Une fois ces tests validés, vous pouvez être certain que la communication entre l'application React et le serveur Node.js est correctement établie, vous permettant d'ajouter de nouvelles fonctionnalités ou d'envisager un déploiement en production.