

ARCHITECTURE LOGICIELLE

GROUPE_8 : TP1_DOCKERFILE



docker.

MEMBRES DU GROUPE

NOMS ET PRÉNOMS

MATRICULES

KEMTHO PAUL ZIDANE

19M2662

NDZANA NNONO JEANNE D'ARC

18T2860

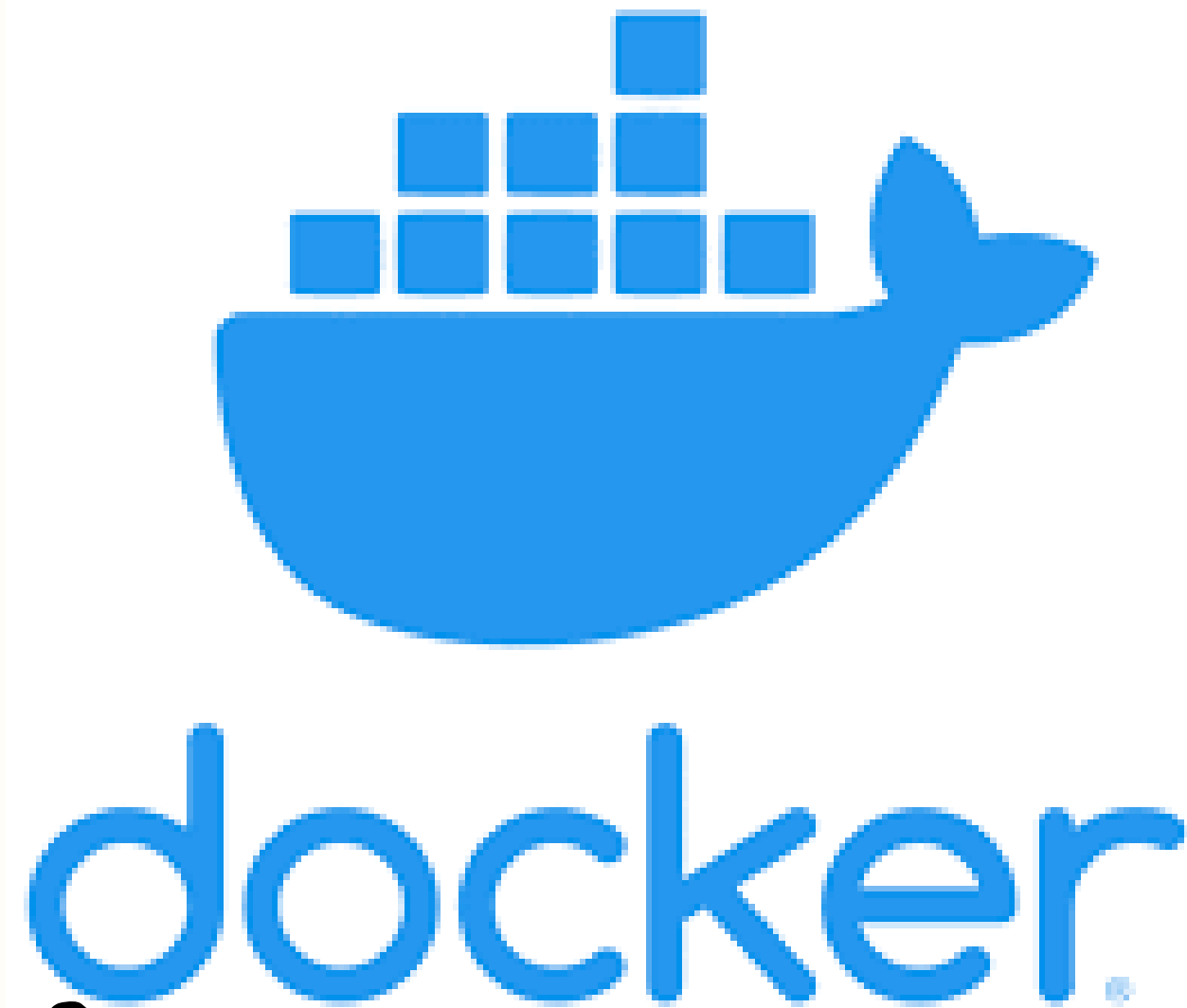
NJIKOUFON MFOCHIVE OUSMANOU

19M2876

KAMGA MOTADJE FRANCK DONALD

19M2493

INTRODUCTION



Lors du développement du mini blog en utilisant React et Node le déploiement de l'application reposait sur des hypothèses importantes concernant votre environnement de travail. En effet, le fait de devoir lancer manuellement chaque service (par exemple avec la commande `npm start`) rend le déploiement complexe et peu reproductible.

Pour résoudre ce problème, nous allons dans ce TP mettre en place une solution de déploiement basée sur les conteneurs Docker.

1. CREATION DES DOCKER-FILE POUR CHAQUE APPLICATION

NB: IL N'EST PAS NÉCESSAIRE D'EXÉCUTER LES COMMANDES APRÈS LES DOCKERFILES, CAR CES OPÉRATIONS SERONT EXÉCUTÉES PAR LA COMMANDE DE **DOCKER-COMPOSE.**

A-DOCKERFILE POUR L'APPLICATION "CLIENT"

```
1  # Use a Node.js image as the base image
2  FROM node:14
3
4  # Set the working directory to /app
5  WORKDIR /app
6
7  # Copy the package.json and package-lock.json files to the container
8  COPY package*.json ./
9
10 # Install the dependencies
11 RUN npm install
12
13 # Copy the rest of the application code to the container
14 COPY . .
15
16 # Build the React application
17 RUN npm run build
18
19 # Serve the React application using a simple HTTP server
20 CMD ["npx", "serve", "-s", "build", "-p", "3000"]
21
```

B-DOCKERFILE POUR L'APPLICATION "COMMENT"

```
1  # Use the Node.js 14 image as the base image
2  FROM node:14
3
4  # Set the working directory to /app
5  WORKDIR /app
6
7  # Copy the package.json and package-lock.json files to the container
8  COPY package*.json ./
9
10 # Install the dependencies
11 RUN npm install
12
13 # Copy the rest of the application code to the container
14 COPY . .
15
16 # Build the application
17 #RUN npm run build
18
19 # Expose the port that the application will run on (in this case, port 4001)
20 EXPOSE 4001
21
22 # Start the application
23 CMD ["npm", "start"]
```

C-DOCKERFILE POUR L'APPLICATION "EVENT-BUS"



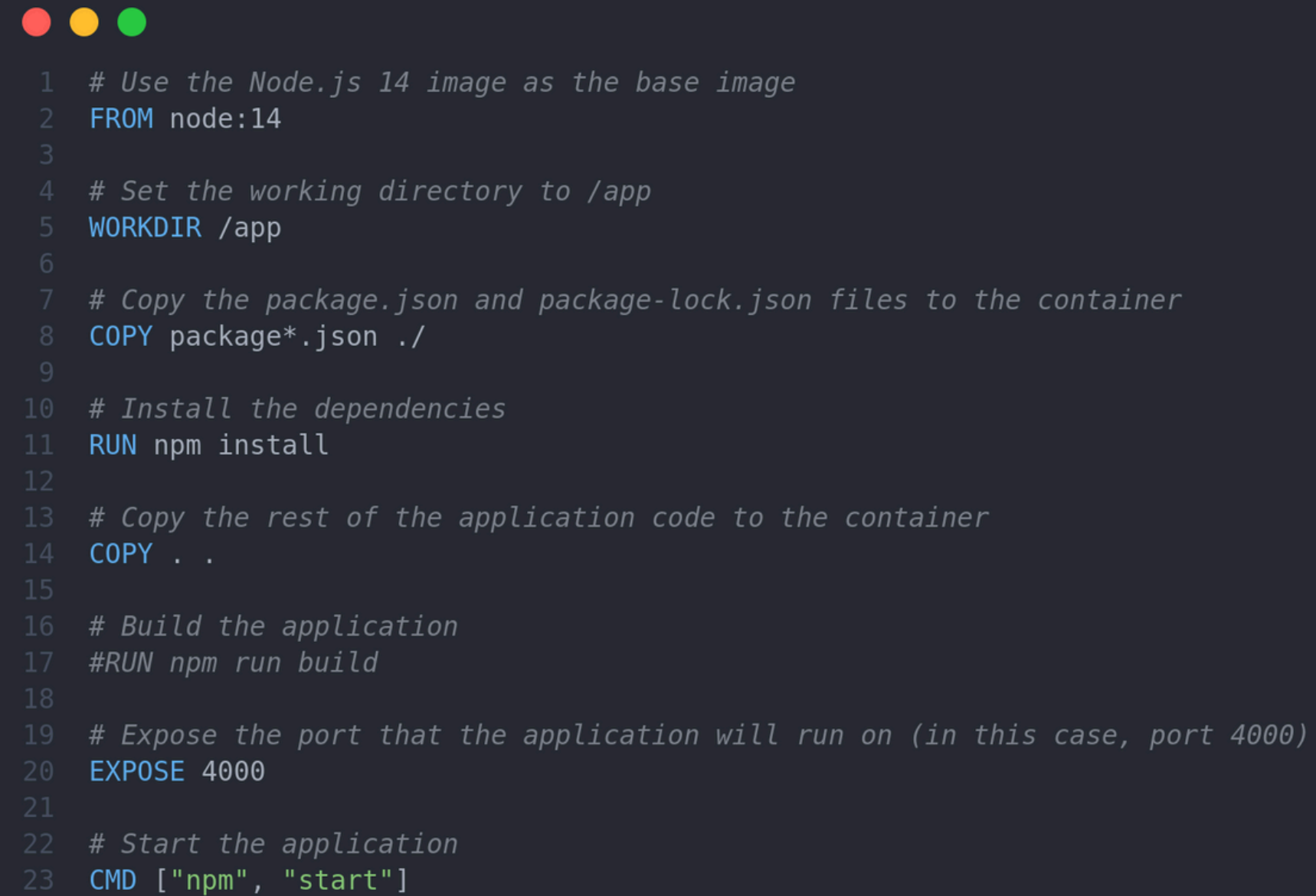
```
1  # Use the Node.js 14 image as the base image
2  FROM node:14
3
4  # Set the working directory to /app
5  WORKDIR /app
6
7  # Copy the package.json and package-lock.json files to the container
8  COPY package*.json ./
9
10 # Install the dependencies
11 RUN npm install
12
13 # Copy the rest of the application code to the container
14 COPY . .
15
16 # Build the application
17 #RUN npm run build
18
19 # Expose the port that the application will run on (in this case, port 4005)
20 EXPOSE 4005
21
22 # Start the application
23 CMD ["npm", "start"]
```

D-DOCKERFILE POUR L'APPLICATION "MODERATION"



```
1  # Use the Node.js 14 image as the base image
2  FROM node:14
3
4  # Set the working directory to /app
5  WORKDIR /app
6
7  # Copy the package.json and package-lock.json files to the container
8  COPY package*.json ./
9
10 # Install the dependencies
11 RUN npm install
12
13 # Copy the rest of the application code to the container
14 COPY . .
15
16 # Build the application
17 #RUN npm run build
18
19 # Expose the port that the application will run on (in this case, port 4003)
20 EXPOSE 4003
21
22 # Start the application
23 CMD ["npm", "start"]
```


D-DOCKERFILE POUR L'APPLICATION "POSTE"



```
1  # Use the Node.js 14 image as the base image
2  FROM node:14
3
4  # Set the working directory to /app
5  WORKDIR /app
6
7  # Copy the package.json and package-lock.json files to the container
8  COPY package*.json ./
9
10 # Install the dependencies
11 RUN npm install
12
13 # Copy the rest of the application code to the container
14 COPY . .
15
16 # Build the application
17 #RUN npm run build
18
19 # Expose the port that the application will run on (in this case, port 4000)
20 EXPOSE 4000
21
22 # Start the application
23 CMD ["npm", "start"]
```

E-DOCKERFILE POUR L'APPLICATION "QUERY"



```
1  # Use the Node.js 14 image as the base image
2  FROM node:14
3
4  # Set the working directory to /app
5  WORKDIR /app
6
7  # Copy the package.json and package-lock.json files to the container
8  COPY package*.json ./
9
10 # Install the dependencies
11 RUN npm install
12
13 # Copy the rest of the application code to the container
14 COPY . .
15
16 # Build the application
17 #RUN npm run build
18
19 # Expose the port that the application will run on (in this case, port 4002)
20 EXPOSE 4002
21
22 # Start the application
23 CMD ["npm", "start"]
```

2-A) CREATION ET CONFIGURATION DU DOCKER COMPOSE 1

```
1  version: '3'
2  services:
3
4    comments:
5      build: ./comments
6      ports:
7        - "4001:4001"
8      environment:
9        - EVENTS_SERVICE_URL=http://events-service:4005
10
11   posts:
12     build: ./posts
13     ports:
14       - "4000:4000"
15     environment:
16       - EVENTS_SERVICE_URL=http://events-service:4005
17
18   events-service:
19     build: ./event-bus
20     ports:
21       - "4005:4005"
22     environment:
23       - POSTS_SERVICE_URL=http://posts:4000
24       - COMMENTS_SERVICE_URL=http://comments:4001
25       - QUERY_SERVICE_URL=http://query:4002
26       - MODERATION_SERVICE_URL=http://moderation:4003
27
```

2-A) CREATION ET CONFIGURATION DU DOCKER COMPOSE 2

```
1
2  query:
3    build: ./query
4    ports:
5      - "4002:4002"
6    environment:
7      - POSTS_SERVICE_URL=http://posts:4000
8      - COMMENTS_SERVICE_URL=http://comments:4001
9      - EVENTS_SERVICE_URL=http://events-service:4005
10
11  moderation:
12    build: ./moderation
13    ports:
14      - "4003:4003"
15    environment:
16      - EVENTS_SERVICE_URL=http://events-service:4005
17
18  client:
19    build: ./client
20    ports:
21      - "3000:3000"
```

Chaque application est définie comme un service dans le fichier **docker-compose.yml**.
Pour chaque service, le **Dockerfile** correspondant est spécifié dans le champ build.dockerfile.
Les ports exposés par chaque application sont mappés aux ports hôtes.
Des variables d'environnement sont définies pour permettre aux applications de communiquer entre elles (par exemple, **EVENTS_HOST**, **COMMENTS_HOST**, **POSTS_HOST**).
Pour construire et exécuter cette configuration, vous pouvez utiliser les commandes suivantes :

Construire les images Docker
docker-compose build

Démarrer les conteneurs
docker-compose up -d

Une fois que tous les conteneurs sont lancés, vous pourrez accéder à chaque application aux adresses suivantes :

Comments: <http://localhost:4001>

Posts: <http://localhost:4000>

Events: <http://localhost:4005>

Query: <http://localhost:4002>

Moderation: Pas d'interface web, communique avec les autres services

Avec cette configuration Docker Compose, vous pouvez facilement déployer l'ensemble de vos applications en une seule commande, en gérant leurs dépendances et leurs communications.

Quelques commandes pour la gestion des microservices

→ Pour lancer un seul service par exemple posts on peut executer cette commande :

`docker-compose start posts`

→ pour arrêter sans arrêter les autres services : `docker-compose stop posts`

→ pour relancer une nouvelle version sans arreter les conteneurs en cours d'execution :

`docker-compose up -d --force-recreate`

→ pour arreter tous les conteneurs : `docker-compose down`

Quelques modifications à effectuer dans les fichiers pour l'adressage des applications:

Lorsque les applications sont déployées dans des conteneurs, l'utilisation de "localhost" dans les liens d'API n'est plus appropriée. En effet, chaque conteneur a sa propre adresse IP et son propre environnement, ce qui signifie que les liens "localhost" ne fonctionneront pas de manière fiable entre les différents composants de l'application.

Pour résoudre ce problème, il est recommandé de découpler la communication entre les services en utilisant les variables d'environnement. Voici comment cela fonctionne :

→ Définir les liens d'accès aux API dans des variables d'environnement :

Au lieu d'avoir des liens codés en dur dans le code, on les stocke dans des variables d'environnement.

Par exemple, on peut avoir une variable d'environnement appelée "POSTS_API_URL" qui contient le lien d'accès à l'API "posts".

→ **Utiliser ces variables d'environnement dans le code :**

Dans le fichier "index.js" du service "posts", on utilise la valeur de la variable d'environnement "POSTS_API_URL" pour accéder à l'API.

Cela permet de découpler le code du lien d'accès spécifique.

→ **Avantages de cette approche :**

Si le lien d'accès à un service change, il suffit de mettre à jour la variable d'environnement correspondante, sans avoir à modifier le code.

Cela rend l'application plus flexible et facilite la maintenance, car les changements de liens d'API n'impactent pas le code source.

Cette méthode est également plus sécurisée, car les liens d'API ne sont pas exposés directement dans le code.

Fichiers indexe.js de post modifié:

```
const express = require("express");
const bodyParser = require("body-parser");
const { randomBytes } = require("crypto");
const cors = require("cors");
const axios = require("axios");

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

const posts = {};

app.get("/posts", (req, res) => {
  res.send(posts);
});

app.post("/posts", async (req, res) => {
  const id = randomBytes(4).toString("hex");
  const { title } = req.body;

  posts[id] = {
    id,
    title,
  };
});
```

```
await axios.post(process.env.EVENTS_SERVICE_URL + "/events", {  
  type: "PostCreated",  
  data: {  
    id,  
    title,  
  },  
});
```

```
res.status(201).send(posts[id]);  
});
```

```
app.post("/events", (req, res) => {  
  console.log("Received Event", req.body.type);
```

```
  res.send({});  
});
```

```
app.listen(4000, () => {  
  console.log("Listening on 4000");  
});
```

`process.env.EVENTS_SERVICE_URL` représente la variable d'environnement définie lors de la construction de l'application post voir cette section de `docker-compose.yml`:

```
posts:
build: ./posts
ports:
- "4000:4000"
environment:
- EVENTS_SERVICE_URL=http://events-service:4005
```

on constate que sur la dernière ligne localhost est remplacé par `events-service`

il faut donc faire de même pour les autres services selon les variables d'environnement définies depuis `docker-compose.yml`

problème rencontré et résolu

Tous les services communiquent maintenant normalement, mais le service `Query` renvoie toujours une liste vide au lieu **des posts et commentaires** qui ont été créés.

Code du fichier indexe.js de query:

```
const express = require("express");  
const bodyParser = require("body-parser");  
const cors = require("cors");  
const axios = require("axios");  
  
const app = express();  
app.use(bodyParser.json());  
app.use(cors());  
  
const posts = {};  
  
const handleEvent = (type, data) => {  
  if (type === "PostCreated") {  
    const { id, title } = data;  
  
    posts[id] = { id, title, comments: [] };  
  }  
  
  if (type === "CommentCreated") {  
    const { id, content, postId, status } = data;  
  
    const post = posts[postId];
```

```
|post.comments.push({ id, content, status });
}

if (type === "CommentUpdated") {
  const { id, content, postId, status } = data;

  const post = posts[postId];
  const comment = post.comments.find((comment) => {
    return comment.id === id;
  });

  comment.status = status;
  comment.content = content;
}

app.get("/posts", (req, res) => {
  res.send(posts);
});

app.post("/events", (req, res) => {
  const { type, data } = req.body;
```

```
handleEvent(type, data);

res.send({});
});

app.listen(4002, async () => {
  console.log("Listening on 4002");
  try {
    const res = await axios.get(process.env.EVENTS_SERVICE_URL + "/events");
    console.log("Received events:", res.data.length);
    for (let event of res.data) {
      console.log("Processing event:", event.type);
      handleEvent(event.type, event.data);
    }
  } catch (error) {
    console.log(error.message);
  }
});
```

D'APRÈS CE CODE, ON CONSTATE QUE LES ÉVÉNEMENTS SONT CHARGÉS UNIQUEMENT LORS DU DÉMARRAGE DE CE SERVICE. CELA SIGNIFIE QUE TOUT ÉVÉNEMENT CRÉÉ APRÈS LE DÉMARRAGE DE CE SERVICE NE SERA PAS CHARGÉ DANS CE SERVICE SI L'ON NE RELANCE PAS LE SERVICE.

ON CONSTATE AUSSI SUR CETTE SECTION :

```
const posts = {};  
  
const handleEvent = (type, data) => {  
  if (type === "PostCreated") {  
    const { id, title } = data;  
  
    posts[id] = { id, title, comments: [] };  
  }  
  
  if (type === "CommentCreated") {  
    const { id, content, postId, status } = data;  
  
    const post = posts[postId];  
    post.comments.push({ id, content, status });  
  }  
}
```

```
  if (type === "CommentUpdated") {  
    const { id, content, postId, status } = data;  
  
    const post = posts[postId];  
    const comment = post.comments.find((comment) => {  
      return comment.id === id;  
    });  
  
    comment.status = status;  
    comment.content = content;  
  }  
};  
  
app.get("/posts", (req, res) => {  
  res.send(posts);  
});
```

que l'objet `posts` est initialement vide, et ne peut contenir une valeur que par la méthode `handleEvent`.

Si on fait donc une demande des posts par la méthode

```
app.get("/posts", (req, res) => {  
  res.send(posts);  
});
```

Elle ne pourra renvoyer qu'un tableau vide car pour le moment, l'objet `posts` est encore vide, alors que les données de l'interface client proviennent de là.

SOLUTION

Une solution serait de recharger les événements d'abord, avant de renvoyer les **posts**, car cette variable aurait pris une valeur grâce à la méthode **handleEvent**.

Voici donc le nouveau code de **index.js** du service Query avec le changement de la méthode **app.get** :

```
const express = require("express");
const bodyParser = require("body-parser");
const cors = require("cors");
const axios = require("axios");

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

const posts = {};

const handleEvent = (type, data) => {
  if (type === "PostCreated") {
    const { id, title } = data;

    posts[id] = { id, title, comments: [] };
  }
}
```

```
if (type === "CommentCreated") {  
  const { id, content, postId, status } = data;  
  
  const post = posts[postId];  
  post.comments.push({ id, content, status });  
}  
  
if (type === "CommentUpdated") {  
  const { id, content, postId, status } = data;  
  
  const post = posts[postId];  
  const comment = post.comments.find((comment) => {  
    return comment.id === id;  
  }));  
  
  comment.status = status;  
  comment.content = content;  
}  
};  
  
app.get("/posts", async (req, res) => {  
  try {  
    const res = await axios.get(process.env.EVENTS_SERVICE_URL + "/events");  
    console.log("Received events:", res.data.length);  
    for (let event of res.data) {
```

```
console.log("Processing event:", event.type);  
handleEvent(event.type, event.data);  
}  
} catch (error) {  
console.log(error.message);  
}  
res.send(posts);  
});
```

```
app.post("/events", (req, res) => {  
const { type, data } = req.body;  
  
handleEvent(type, data);  
  
res.send({});  
});
```

```
app.listen(4002, async () => {  
  console.log("Listening on 4002");  
  try {  
    const res = await axios.get(process.env.EVENTS_SERVICE_URL + "/events");  
    console.log("Received events:", res.data.length);  
    for (let event of res.data) {  
      console.log("Processing event:", event.type);  
      handleEvent(event.type, event.data);  
    }  
  } catch (error) {  
    console.log(error.message);  
  }  
});
```

MERCI