API et Web Services

1. API RESTful

1.1 Comprendre les API REST

Les API RESTful (Representational State Transfer) sont des services web qui utilisent les méthodes HTTP pour communiquer et échanger des données entre client et serveur. Voici quelques principes de base :

- **Statelessness** : Chaque requête du client au serveur doit contenir toutes les informations nécessaires pour comprendre la demande. Le serveur ne doit pas stocker l'état de la session client.
- **URI (Uniform Resource Identifier)** : Chaque ressource est identifiée par une URI unique.
- **Méthodes HTTP** : Utilisation des verbes HTTP pour effectuer des actions sur les ressources. Les méthodes courantes sont :
  - **GET** : Récupérer des données.
  - **POST** : Créer une nouvelle ressource.
  - **PUT** : Mettre à jour une ressource existante.
  - **DELETE** : Supprimer une ressource.

1.2 Bonnes Pratiques pour Concevoir une API REST

- **Utiliser des noms de ressources clairs** : Utiliser des noms au pluriel pour les ressources (ex : `/utilisateurs`).
- **Gérer les codes de statut HTTP** : Retourner des codes de statut appropriés (200 OK, 404 Not Found, 500 Internal Server Error, etc.).
- **Pagination** : Lorsque vous renvoyez une liste de ressources, utilisez la pagination pour éviter de surcharger le client. Cela peut être fait via des paramètres de requête.

Exemple d'API REST en Node.js avec Express

```
const express = require('express');
const app = express();
app.use(express.json());

// Simulons une base de données en mémoire
let utilisateurs = [];

// GET - Récupérer tous les utilisateurs
app.get('/utilisateurs', (req, res) => {
    res.status(200).json(utilisateurs);
});

// POST - Créer un nouvel utilisateur
app.post('/utilisateurs', (req, res) => {
```

```
    const { nom, email } = req.body;
    const nouvelUtilisateur = { id: utilisateurs.length + 1, nom, email };
    utilisateurs.push(nouvelUtilisateur);
    res.status(201).json(nouvelUtilisateur);
});

// DELETE - Supprimer un utilisateur par ID
app.delete('/utilisateurs/:id', (req, res) => {
    const { id } = req.params;
    utilisateurs = utilisateurs.filter(u => u.id !== parseInt(id));
    res.status(204).send(); // Pas de contenu à retourner
});

// Démarrer le serveur
app.listen(3000, () => {
    console.log('Serveur à l\'écoute sur le port 3000');
});
```

---

2. GraphQL

2.1 Comprendre GraphQL

GraphQL est un langage de requête pour les APIs, développé par Facebook. Il permet aux clients de demander exactement les données dont ils ont besoin, et rien de plus. Cela contraste avec REST, où les réponses peuvent contenir des données superflues.

2.2 Avantages de GraphQL

- **Flexibilité** : Les clients peuvent spécifier la structure des données qu'ils souhaitent récupérer.
- **Requêtes complexes** : Permet de récupérer des données de plusieurs ressources en une seule requête.

Exemple de Schéma GraphQL

```
type Utilisateur {
    id: ID!
    nom: String!
    email: String!
}

type Query {
    utilisateurs: [Utilisateur!]!
    utilisateur(id: ID!): Utilisateur
}
```

```
type Mutation {
    creerUtilisateur(nom: String!, email: String!): Utilisateur!
}
```

Exemple d'Utilisation de GraphQL avec Apollo Server

```
const { ApolloServer, gql } = require('apollo-server');

// Simulons une base de données en mémoire
let utilisateurs = [];

// Type Definitions (Schéma GraphQL)
const typeDefs = gql`
    type Utilisateur {
        id: ID!
        nom: String!
        email: String!
    }

    type Query {
        utilisateurs: [Utilisateur!]!
    }

    type Mutation {
        creerUtilisateur(nom: String!, email: String!): Utilisateur!
    }
`;

// Résolveurs
const resolvers = {
    Query: {
        utilisateurs: () => utilisateurs,
    },
    Mutation: {
        creerUtilisateur: (_, { nom, email }) => {
            const nouvelUtilisateur = { id: utilisateurs.length + 1, nom, email };
            utilisateurs.push(nouvelUtilisateur);
            return nouvelUtilisateur;
        },
    },
};

// Créer le serveur Apollo
const server = new ApolloServer({ typeDefs, resolvers });

// Démarrer le serveur
server.listen().then(({ url }) => {
    console.log(`Serveur à l'écoute sur ${url}`);
});
```

---

3. WebSockets et gRPC

3.1 WebSockets

WebSockets permettent une communication bidirectionnelle entre le client et le serveur. Contrairement aux requêtes HTTP classiques, les WebSockets permettent au serveur d'envoyer des messages au client à tout moment.

Exemple d'Utilisation de WebSockets avec Socket.io

```
const express = require('express');
const http = require('http');
const { Server } = require('socket.io');

const app = express();
const server = http.createServer(app);
const io = new Server(server);

// Événement de connexion
io.on('connection', (socket) => {
    console.log('Nouvelle connexion');

    // Recevoir un message du client
    socket.on('message', (msg) => {
        console.log('Message reçu:', msg);
        // Émettre le message à tous les clients
        io.emit('message', msg);
    });

    // Événement de déconnexion
    socket.on('disconnect', () => {
        console.log('Client déconnecté');
    });
});

// Démarrer le serveur
server.listen(3000, () => {
    console.log('Serveur à l\'écoute sur le port 3000');
});
```

3.2 gRPC

gRPC est un framework de communication à distance développé par Google. Il utilise HTTP/2 pour les communications et permet de créer des APIs hautes performances.

Avantages de gRPC

- **Performances élevées** : Utilisation de Protocol Buffers pour la sérialisation des données, ce qui réduit la taille des messages.
- **Streaming** : Prise en charge du streaming bidirectionnel, permettant des communications en temps réel.

Exemple de Configuration d'une API gRPC

1. **Définir le schéma dans un fichier `.proto`**

```
syntax = "proto3";

service UtilisateurService {
    rpc CreerUtilisateur (Utilisateur) returns (Utilisateur);
    rpc ListerUtilisateurs (google.protobuf.Empty) returns (UtilisateurList);
}

message Utilisateur {
    int32 id = 1;
    string nom = 2;
    string email = 3;
}

message UtilisateurList {
    repeated Utilisateur utilisateurs = 1;
}
```

2. **Générer le code client et serveur à partir du fichier `.proto`.**
3. **Implémenter le serveur gRPC**

```
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

// Charger le fichier .proto
const packageDefinition = protoLoader.loadSync('utilisateur.proto', {});
const utilisateurProto =
grpc.loadPackageDefinition(packageDefinition).UtilisateurService;

// Simulons une base de données en mémoire
let utilisateurs = [];

// Implémentation des méthodes du service
const creerUtilisateur = (call, callback) => {
    const nouvelUtilisateur = { id: utilisateurs.length + 1, nom: call.request.nom,
email: call.request.email };
    utilisateurs.push(nouvelUtilisateur);
    callback(null, nouvelUtilisateur);
};

const listerUtilisateurs = (call, callback) => {
```

```
    callback(null, { utilisateurs });
};

// Démarrer le serveur gRPC
const server = new grpc.Server();
server.addService(utilisateurProto.UtilisateurService.service, { creerUtilisateur,
listerUtilisateurs });
server.bindAsync('0.0.0.0:50051', grpc.ServerCredentials.createInsecure(), () => {
    console.log('Serveur gRPC à l\'écoute sur le port 50051');
    server.start();
});
```

Ces technologies offrent aux développeurs des outils puissants pour construire des applications web robustes et efficaces. Comprendre ces concepts te permettra de concevoir des systèmes plus flexibles et évolutifs.