INFORMATICS INSTITUTE OF TECHNOLOGY

6COSC022C.2 Advanced Server-Side Web Programming

Course Work 1

Documentation

Student Name: - G.H.S.A Hettiarachchi

Student ID: - 20200296

UOW ID: - W1867415

Contents

1. Initialization Phase 4
    1. Database Setup (config/db.js) 4
    2. Model Definitions 4
    User Model (models/User.js) 4
    ApiKey Model (models/ApiKey.js) 5
2. Authentication Flow 5
    Signup Process 5
    Login Process 6
3. API Key Lifecycle 7
    Key Generation 7
Checking API Key for User 7
4. Country Data Service 8
    Process Country Data: 9
5. Admin Operations 10
    1. Delete User and Associated API Key (deleteUserAndApiKey) 10
    2. Bulk Data Retrieval (getAllUsersWithApiData) 11
    3. Delete Specific API Key (deletekey) 12
    4. Regenerate API Key (regenkey) 12
    Router Setup: 13
7. Data Relationships 13
    Data Relationships - User ↔ API Key Binding 13
8. JWT Verification Middleware (verifyToken) 14
    Process Overview 14
    JWT Verification: 15
    Email Validation: 15
    Proceeding with Valid Request: 15
    Error Handling 16

Security Considerations 16
Usage in Routes 16
9. Frontend 16

1. Initialization Phase

1. Database Setup (config/db.js)

Database Connection:

This code configures Sequelize to connect to an SQLite database (./database.sqlite). It establishes the database connection using Sequelize's ORM.

```
const sequelize = new Sequelize({
  dialect: 'sqlite',
  storage: './database.sqlite'
});
```

Foreign Key Constraints:

SQLite doesn't enforce foreign key constraints by default. To ensure these constraints are applied, PRAGMA foreign_keys = ON; is executed after the connection is authenticated. This ensures relationships (like the one between User and ApiKey) are maintained properly.

```
sequelize.authenticate()
  .then(() => {
    sequelize.query('PRAGMA foreign_keys = ON;');
    console.log('Connection has been established successfully.');
  })
  .catch((err) => {
    console.error('Unable to connect to the database:', err);
  });
```

---

2. Model Definitions

User Model (models/User.js)

Password Hashing:

The beforeCreate hook ensures the user's password is hashed before it's saved to the database. The bcrypt library is used for password hashing, enhancing security.

```
User.beforeCreate(async (user) => {
  user.password = await bcrypt.hash(user.password, 10);
});
```

Associations:

The User model has a one-to-one relationship with the ApiKey model, where the userId foreign key in ApiKey links the two models together.

```
User.hasOne(models.ApiKey, { foreignKey: 'userId' });
```

ApiKey Model (models/ApiKey.js)

Foreign Key:
The ApiKey model includes a userId field that acts as a foreign key linking the API key to the corresponding user. This ensures the API key is associated with a specific user.

```
ApiKey.init({
 userId: {
  type: DataTypes.INTEGER,
  allowNull: false,
  references: {
   model: 'users',
   key: 'id'
  }
 },
 apiKey: {
  type: DataTypes.STRING,
  allowNull: false,
  unique: true
 }
});
```

Associations:
The ApiKey model uses a belongsTo association with the User model, establishing the one-to-one relationship between the models.

```
ApiKey.belongsTo(models.User, { foreignKey: 'userId' });
```

2. Authentication Flow

Signup Process

Route: POST /signup

Controller Steps:

- **Validates Email Format**: Ensures the email format is valid, using built-in validation functions (e.g., isEmail).
- **Checks for Duplicate Email/Username**: Uses User.findOne to check if the provided email or username already exists in the database.

- **Hashes Password**: Uses bcrypt.hash with 10 rounds to securely hash the user's password before storing it in the database.
- **Creates User Record**: If the email/username is not taken, a new user is created with User.create using the provided data.
- **Issues JWT**: A JWT (JSON Web Token) is issued with a one-hour expiry time to authenticate the user in future requests.

```
const newUser = await User.create({
  username,
  email,
  password
});

const token = jwt.sign({ userId: newUser.id, email: email }, 'jwt', {
  expiresIn: '1h'
});
res.status(201).json({ message: 'User created successfully', user: newUser, token: token });
```

Login Process

Route: POST /login

Controller Steps:

- **Finds User by Email**: Searches for the user by their email using User.findOne.
- **Compares Password Hashes**: Uses bcrypt.compare to compare the hashed password in the database with the password the user provided.
- **Generates JWT**: If the password matches, a JWT is generated for the user, valid for 1 hour.
- **Checks for Existing API Key**: Calls APIkeyController.ChekApiKeyForUser to check if the user already has an API key. If they do, it returns the existing key; otherwise, it returns 0.

  Code Snippet:

```
const user = await User.findOne({ where: { email } });
const isMatch = await bcrypt.compare(password, user.password);
const token = jwt.sign({ userId: user.id, email: email }, 'jwt', {
  expiresIn: '1h'
});
```

3. API Key Lifecycle

Key Generation

Route: POST /generate-api-key

Controller Steps:

- **Uses Node's Crypto Module**: The crypto.randomBytes function is used to generate a secure 32-byte random string, which is then converted to a 64-character hexadecimal string for the API key.
- **Stores with Reference to User ID**: The generated API key is stored in the ApiKey table with a reference to the userId to link the key to the respective user.
- **Returns 64-Character Hex String**: The generated API key is returned in the response.

  Code Snippet:

```
const apiKey = crypto.randomBytes(32).toString('hex');


const newApiKey = await ApiKey.create({
  userId: userId,
  apiKey: apiKey,
});

res.status(200).json({ message: 'API key generated successfully', apiKey: newApiKey.apiKey
});
```

Checking API Key for User

The `ChekApiKeyForUser` method checks if a user already has an API key. If the user ID is valid and the key exists, it returns the API key. If not, it returns an error message.

```
const userApiKey = await ApiKey.findOne({ where: { userId: userId } });

if (!userApiKey) {
  return { status: 'error', message: 'No API key found for this user' };
}
return { status: 'success', apiKey: userApiKey.apiKey };
```

4. Country Data Service

Request Handling:

- Route: `GET /api/countries?country=name`
- The route accepts an optional query parameter `country`, which specifies the country to search for.

Execution Flow:

- JWT Verification: The `verifyToken` middleware ensures the user is authenticated before processing the request.
- Forwarding to External API: The request is forwarded to the REST Countries API (https://restcountries.com) to retrieve country data. If a specific country is specified, it fetches details for that country, otherwise, it fetches details for all countries.
- Decompression: The response from the external API can be compressed using `gzip`, `deflate`, or `br` compression formats. These are handled and decompressed using the `zlib` library to ensure the data is correctly processed.
- Data Transformation: After receiving the response, the data is processed via the `processCountryData` function. This function filters and formats the data to include only relevant information (e.g., country name, flag, capital, etc.).
- API Call Count: After a successful response, the API count for the authenticated user is incremented. This helps track the usage of the API.

Error Handling:

- 404: If no countries are found (empty response from external API).
- 502: If the external API response is invalid or corrupt.
- 504: If a request times out (more than 60 seconds without a response from the external API).

Code Breakdown:

Router: Handles the route and connects to the controller.

```
router.get('/api/countries', verifyToken, countryController.getCountries);
```

Controller (`getCountries`):

- The `axios` library is used to fetch data from the external API with a 60-second timeout.

- The response is streamed and decompressed based on the encoding type (gzip, deflate, or br).

Handling Decompression:

- If the response is compressed, it is decompressed using `zlib` streams (either `createGunzip`, `createInflate`, or `createBrotliDecompress`).
- Data Processing: The raw response is transformed into a more usable format by calling `processCountryData`.

```
const response = await axios({
  method: 'get',
  url: url,
  responseType: 'stream',
  timeout: 60000,
  headers: { 'Accept-Encoding': 'gzip, deflate, br' }
});
```

- **Increment API Call Count**: The API call count for the user is incremented to track usage.

```
const incrementApiCount = async (userId) => {
 // logic for updating API call count
};
```

- **Error Handling in Controller**: Handles different types of errors (e.g., connection timeout, bad API response, etc.) and responds accordingly.

```
if (error.code === 'ECONNABORTED') {
  res.status(504).json({ error: 'Gateway Timeout' });
} else if (error.response) {
  res.status(error.response.status).json({ error: 'API Error' });
} else {
  res.status(502).json({ error: 'Bad Gateway', details: error.message });
}
```

Process Country Data:

The `processCountryData` function extracts and formats the relevant information from the country data:

```
exports.processCountryData = (data) => {
  return data.map((country) => ({
    name: country.name.common,
    currency: country.currencies,
    capital: country.capital,
    languages: country.languages,
    flag: country.flags.png
  }));
};
```
Why Decompression and Streams?

Sometimes the REST Countries API response might not return immediately or might be compressed. Handling streams allows for the proper processing of large responses that could be compressed, ensuring that data is correctly received and decompressed even for larger payloads.

---

5. Admin Operations

1. Delete User and Associated API Key (deleteUserAndApiKey)

This function is responsible for deleting a user and their associated API key from the database. It performs the following actions:

- **Find the User**: It first looks up the user based on the `userId` parameter in the request.
- **Cascade Delete**: Once the user is found, their associated `ApiKey` is also deleted (if it exists), ensuring that both the user and the API key are removed from the system.

```
async function deleteUserAndApiKey(req, res) {
  const userId = req.params.userId;

  try {
    const user = await User.findOne({ where: { id: userId } });

    if (!user) {
      return res.status(404).json({ message: 'User not found' });
    }

    const apiKey = await ApiKey.findOne({ where: { userId } });

    if (apiKey) {
      await ApiKey.destroy({ where: { userId } });
      console.log('API key deleted');
    }

    await user.destroy();
    console.log('User deleted');
    return res.status(200).json({ message: 'User and associated data deleted successfully'
});
  } catch (error) {
    console.error('Error:', error);
    return res.status(500).json({ message: 'Failed to delete user and API key', error:
error.message });
  }
}
```

2. Bulk Data Retrieval (getAllUsersWithApiData)

This function retrieves all users along with their associated API key information. The main logic involves:

- **LEFT JOIN**: It performs a `LEFT JOIN` between the `User` and `ApiKey` models. This ensures that even users without an API key are included in the result.
- **Attributes**: The response includes both user information (e.g., `username`, `email`) and API key details (e.g., `apiKey`, `apiCount`).

Code:

```
async function getAllUsersWithApiData(req, res) {
  try {
    const usersWithApiKeys = await User.findAll({
      include: {
        model: ApiKey,
        required: false,  // Optional join, includes users without API keys
        where: {
          userId: { [Sequelize.Op.eq]: Sequelize.col('User.id') }  // Ensures the join is based
on userId
        },
        attributes: ['apiKey', 'apiCount', 'createdAt', 'updatedAt'],
      },
      attributes: ['id', 'username', 'email', 'createdAt', 'updatedAt'],
    });

    if (!usersWithApiKeys || usersWithApiKeys.length === 0) {
      return res.status(404).json({ message: 'No users found' });
    }

    res.json({ users: usersWithApiKeys });
  } catch (err) {
    res.status(500).json({ message: 'Failed to retrieve users and API key data', error:
err.message });
```

```
  }
}
```

3. Delete Specific API Key (deletekey)

This function deletes a specific API key based on the `apikey` parameter in the request. It performs:

- Find the API Key: Looks up the `ApiKey` model by the provided API key.
- Delete the API Key: If found, the API key is removed from the database.

```
async function deletekey(req, res) {
  const apiKey = req.params.apikey;

  try {
    const key = await ApiKey.findOne({ where: { apiKey } });

    if (!key) {
      return res.status(404).json({ error: 'API key not found' });
    }

    await ApiKey.destroy({ where: { apiKey } });

    res.status(200).json({ message: 'API key revoked successfully' });
  } catch (error) {
    console.error(error);
    res.status(500).json({ error: 'Something went wrong' });
  }
}
```

4. Regenerate API Key (regenkey)

- This function regenerates a new API key for a user. It follows these steps:
- Generate a New API Key: It generates a new 32-byte random key using the `crypto` library and stores it in the `ApiKey` model.
- Return the New Key: The newly generated API key is sent back in the response.

```
async function regenkey(req, res) {
  const userId = req.params.userId;

  try {
    const apiKey = crypto.randomBytes(32).toString('hex');

    const newApiKey = await ApiKey.create({
          userId: userId,
          apiKey: apiKey,
        });

        res.status(200).json({ message: 'API key generated successfully', apiKey:
newApiKey.apiKey });
    } catch (error) {
        console.error(error);
        res.status(500).json({ error: 'Something went wrong while generating the API key' });
    }
}
```

Router Setup:

In the router configuration, different routes are set up for the admin operations like deleting users, retrieving all users, regenerating API keys, etc.

```
const express = require('express');
const router = express.Router();
const { deleteUserAndApiKey, getAllUsersWithApiData, deletekey, regenkey } =
require('../controllers/admincontroller');
const verifyToken = require('../middlewares/authMiddleware');

router.post('/getAllAdmin', verifyToken, getAllUsersWithApiData);
router.delete('/deleteAdmin/:userId', verifyToken, deleteUserAndApiKey);
router.delete('/deleteApiKey/:apikey', verifyToken, deletekey);
router.post('/regenApiKey/:userId', verifyToken, regenkey);

module.exports = router;
```

7. Data Relationships

Data Relationships - User ↔ API Key Binding

Purpose

The relationship between the User and ApiKey models establishes a one-to-one relationship, which means each user is linked to a single API key, and each API key is tied to a single user. This relationship ensures data integrity, facilitates efficient data retrieval, and supports cascade deletion, maintaining the consistency of related data.

Data Relationship Overview

User ↔ ApiKey Binding:

- One-to-One Relationship: A user can have at most one API key, and each API key is assigned to a single user. This is expressed in the database schema as a one-to-one relationship between the `User` and `ApiKey` models.

Model Definitions:

- In the User model:

```
User.hasOne(ApiKey, { foreignKey: 'userId' });
```

This line establishes a one-to-one relationship between the `User` and `ApiKey` models. The `User` model can reference an `ApiKey` via the `userId` foreign key.

- In the ApiKey model:

```
ApiKey.belongsTo(User, { foreignKey: 'userId' });
```

This line indicates that the `ApiKey` model references the `User` model through the `userId` foreign key.

8. JWT Verification Middleware (verifyToken)

Purpose

The `verifyToken` middleware function is designed to authenticate users by validating their JSON Web Token (JWT) and ensuring that the user's email matches the one encoded within the token. This middleware secures specific routes, preventing unauthorized access by verifying both the presence of a valid token and the email consistency.

Process Overview

Token Retrieval:

- The function checks for the presence of a JWT in the request's `Authorization` header. The token should be provided in the format: `Authorization: Bearer <token>`. The middleware extracts the token by splitting the string and retrieving the second part (after "Bearer").
- If the token is missing, a `403 Forbidden` status is returned, along with an error message indicating that no token was provided.

```
const token = req.headers['authorization'];
```

```
if (!token) {
  return res.status(403).json({ error: 'No token provided' });
}
```

JWT Verification:

- The middleware attempts to **verify** the extracted token using the `jsonwebtoken` library's `verify` method. It decodes the token using a predefined secret (`'jwt'` in this case). The secret key should ideally be stored in an environment variable for enhanced security, especially in production environments.
- **If the token is invalid or expired**, the middleware catches the error and responds with a `401 Unauthorized` status, indicating that the token is either invalid or has expired.

```
const decoded = jwt.verify(token.split(' ')[1], 'jwt');
```

Email Validation:

- The middleware compares the **email** provided in the request (either from `req.query.email` or `req.body.email`) with the email stored in the decoded JWT (`decoded.email`). This step ensures that the token matches the expected user.
- **If the emails don't match**, the request is rejected with a `401 Unauthorized` status, indicating an email mismatch. This check ensures that the user cannot tamper with their email in the request body to access data belonging to another user.

```
var requestEmail = req.query.email;
if (!requestEmail) {
  requestEmail = req.body.email;
}

if (requestEmail !== decoded.email) {
  return res.status(401).json({ error: 'Invalid email' });
}
```

Proceeding with Valid Request:

- **If both the token is valid and the emails match**, the decoded user information is attached to the `req.user` object, making it available for the rest of the route handler or other middleware functions. The request then proceeds to the next middleware or controller action via `next()`.

```
req.user = decoded;
next();
```

Error Handling

- No token provided: Returns a `403 Forbidden` response if no JWT is found in the request header.
- Invalid or expired token: If the JWT is invalid or has expired, the middleware sends a `401 Unauthorized` response.
- Invalid email: If the email in the request doesn't match the email encoded in the token, the middleware responds with a `401 Unauthorized` status.

```
return res.status(401).json({ error: 'Invalid or expired token' });
```

Security Considerations

- JWT Secret: The secret key used for token verification (`'jwt'`) should be stored securely in an environment variable rather than hardcoded in the source code.
- Logging: While logging the email of the user is helpful for debugging, it should be carefully managed to avoid logging sensitive data in production environments.

Usage in Routes

The `verifyToken` middleware is used to secure routes by attaching it as a middleware function before the route handler. This ensures that only authenticated users can access protected routes. Here's an example of using the middleware in a route:

```
const express = require('express');
const router = express.Router();
const verifyToken = require('../middlewares/authMiddleware');

router.get('/protected-route', verifyToken, (req, res) => {
  res.status(200).json({ message: 'Access granted to protected route', user: req.user });
});

module.exports = router;
```

In this example, any request to `/protected-route` will first pass through the `verifyToken` middleware. If the token is valid and the email matches, the route handler will execute. The decoded user data will be available in `req.user`.

9. Frontend

The frontend of the application is fully developed using React.js, providing a smooth and responsive interface for both administrative and user tasks. It integrates with the REST Country API backend to handle various functionalities. For user-related tasks, the frontend interacts with the backend to fetch data such as country information, process requests like country search, and display details like flags and currencies. Admin tasks such as user management, API key generation, and deletion are also handled efficiently. The frontend makes API calls to perform these tasks, ensuring seamless communication with the backend, which processes the data and returns responses for display on the user interface.

The React.js-based frontend is designed to offer an intuitive and straightforward navigation experience. Admins can easily manage users and API keys, while users can access country information from the backend. Whether performing administrative functions or retrieving country data, the frontend ensures smooth and efficient interactions with the backend, enhancing overall usability and performance.