Nest

A progressive Node.js framework for building efficient and scalable server-side applications.

Installation Manual for TuneIn Application

This manual provides step-by-step instructions to set up and run the TuneIn application locally. The application consists of a frontend built with React Native and a backend built with NestJS.

Prerequisites

1. **Node.js**: Ensure you have Node.js installed (version 20.0.6).
2. **npm**: npm is included with Node.js.
3. **Docker**: Install Docker for managing local databases (optional).
4. **PostgreSQL**: If not using Docker, install PostgreSQL on your machine.
5. **AWS Account**: Necessary for S3 and Cognito configurations.
6. **Expo CLI**: Install Expo CLI globally using:

   ```
   npm install -g expo-cli
   ```

Clone the Repository

1. Clone the repository from GitHub:

   ```
   git clone https://github.com/COS301-SE-2024/TuneIn
   cd TuneIn
   ```

Setup Backend

Step 1: Navigate to Backend Folder

```
cd backend
```

Step 2: Install Dependencies

```
npm install
```

Step 3: Configure Environment Variables

Create a `.env` file in the `backend` folder with the following format:

```
AWS_S3_BUCKET_NAME="<your-s3-bucket-name>"
AWS_S3_REGION="<your-s3-region>"
```

```
# Auth settings
JWT_SECRET_KEY="<your-jwt-secret>"
JWT_EXPIRATION_TIME="2h"
EXPO_USER_POOL_ID="<your-expo-user-pool-id>"
EXPO_CLIENT_ID="<your-expo-client-id>"
PERSONAL_EMAIL="<your-email>"
PERSONAL_PASSWORD="<your-password>"
SOCKET_ROOM_ID="<your-socket-room-id>"
SOCKET_SENDER="<your-socket-sender-id>"

# Spotify
SPOTIFY_CLIENT_ID="<your-spotify-client-id>"
SPOTIFY_CLIENT_SECRET="<your-spotify-client-secret>"
SPOTIFY_REDIRECT_URI="http://localhost:3000/auth/spotify/callback"

AUTH_STATE_SECRET_KEY="<your-auth-state-secret>"
SALT="<your-salt>"
TUNEIN_USER_ID="<your-tunein-user-id>"
```

Step 4: Setup Database

If using PostgreSQL, ensure your database is running and create the necessary tables:

1. **Using Docker**:

   ```
   docker run --name tunein-db -e POSTGRES_USER=<your-username> -e POSTGRES_PASSWORD=<your
   ```

2. **Using Local PostgreSQL**:

   - Create a database named `initial_db` and configure the user and password as specified in your `.env` file.

Step 5: Run Database Migrations

Run the following command to set up the database schema:

```
npx prisma migrate dev
```

Step 6: Start the Backend Server

Run the backend server:

```
npm run start
```

Setting Up a Spotify App

To enable Spotify-related functionality in the TuneIn app, you need to create and configure a Spotify Developer application. Follow the steps below:

Step 1: Create a Spotify Developer Account

1. Visit the Spotify Developer Dashboard and log in with your Spotify account. If you don't have an account, create one first.
2. Once logged in, click on the **Create an App** button.

Step 2: Create a Spotify App

1. Give your application a **name** (e.g., "TuneIn App") and a **description** (optional).
2. Accept the terms and conditions, then click **Create**.

Step 3: Configure Redirect URIs

1. After creating the app, you'll be redirected to the app's settings page.
2. Under the **Redirect URIs** section, click **Edit Settings** and add the following URI:

http://localhost:3000/auth/spotify/callback

Make sure to click **Save** after adding the URI.

Step 4: Obtain Client ID and Client Secret

1. On the app's settings page, you will see a **Client ID** and **Client Secret**. Copy these values as they will be needed for the `.env` configuration files.

Step 5: Set Environment Variables

Add the following environment variables in both your backend and frontend `.env` files:

```
SPOTIFY_CLIENT_ID="<your-spotify-client-id>"
SPOTIFY_CLIENT_SECRET="<your-spotify-client-secret>"
SPOTIFY_REDIRECT_URI="http://localhost:3000/auth/spotify/callback"
```

Replace and with the actual values from your Spotify app.

Step 6: Set Up Permissions (Scopes)

In the Spotify Dashboard, click on Edit Settings for your app.

In the OAuth section, make sure to request the necessary scopes for your application, such as:

- `user-read-playback-state`
- `user-modify-playback-state`
- `user-read-currently-playing`
- `user-read-recently-played`

- `playlist-read-private`
- `playlist-modify-private`

These permissions will allow the app to access Spotify data and control playback.

Setup Frontend

Step 1: Navigate to Frontend Folder

Open a new terminal and run:

```
cd frontend
```

Step 2: Install Dependencies

```
npm install
```

Step 3: Configure Environment Variables

Create a `.env` file in the `frontend` folder with the following format:

```
DATABASE_URL="postgresql://<your-username>:<your-password>@<your-db-host>:5432/initial_db?sc

AWS_COGNITO_CLIENT_ID="<your-cognito-client-id>"
AWS_COGNITO_USER_POOL_ID="<your-cognito-user-pool-id>"
AWS_ACCESS_KEY_ID="<your-aws-access-key-id>"
AWS_SECRET_ACCESS_KEY="<your-aws-secret-access-key>"
USE_PRODUCTION_SERVER="false"
JWT_SECRET_KEY="<your-jwt-secret>"
JWT_EXPIRATION_TIME="60m"
EXPO_USER_POOL_ID="<your-expo-user-pool-id>"
EXPO_CLIENT_ID="<your-expo-client-id>"
AWS_S3_BUCKET_NAME="<your-s3-bucket-name>"
AWS_S3_REGION="<your-s3-region>"
AWS_S3_ENDPOINT="<your-s3-endpoint>"

# Spotify
SPOTIFY_CLIENT_ID="<your-spotify-client-id>"
SPOTIFY_CLIENT_SECRET="<your-spotify-client-secret>"
SPOTIFY_REDIRECT_URI="http://localhost:3000/auth/spotify/callback"
SPOTIFY_REDIRECT_TARGET="http://localhost:3000/auth/spotify/callback"
```

Step 4: Start the Frontend Application

You can run the app on a browser or an Android emulator:

- **For Browser**:

  ```
  npm start
  ```

- **For Android Emulator**:

  ```
  npm run android
  ```

Conclusion

You should now have the TuneIn application up and running locally. If you encounter any issues, please refer to the console for error messages and ensure that all configurations are correctly set in the `.env` files.

Test

```
# unit tests
$ npm run test

# e2e tests
$ npm run test:e2e

# test coverage
$ npm run test:cov
```

Welcome to Backend

This is a NestJS project with LOTS of plugins to support good programming.

Navigating the files

Files

- **API_SCHEMA.md**
  Contains the schema documentation for the project's API.
- **check-versions.js**
  A script to enforce the correct & consistent version of NodeJS.
- **jest.setup.js**
  Jest configuration and setup script for testing.
- **nest-cli.json**
  Configuration file for the Nest CLI.
- **package.json**
  Lists project dependencies, scripts, and metadata. (for Node & NPM)
- **README.md**
  Overview and documentation of the project.

- **tsconfig.json**
  TypeScript compiler options and configuration for the project.
- **package-lock.json**
  Records the exact version of each installed package to ensure consistent installs.
- **tsconfig.build.json**
  TypeScript configuration for the build process.
- **webpack-hmr.config.js**
  Webpack configuration for Hot Module Replacement (HMR).

Directories

- **archive**
  Archived files & folders.
- **coverage**
  Contains code coverage reports generated by Jest.
- **dist**
  Compiled output files from the TypeScript source code.
- **documentation**
  Auto-generated documentation for the project.
- **jest_mocking**
  Contains mock files for Jest testing.
- **node_modules**
  Dependency modules & addons installed by npm.
- **prisma**
  Contains Prisma-related files, including schema and migration configurations.
- **src**
  Source code of the application.
- **test**
  Contains unit and integration tests for the application.
- **tools**
  Extra tools & scripts for development.
- **uploads**
  Directory for uploaded files.

Source files

Files

- **app.controller.spec.ts**
  Unit tests for the `AppController` class.
- **app.service.ts**
  Contains the business logic for the `AppService` class.
- **app.controller.ts**
  The main controller handling incoming requests and returning responses.
- **app.module.ts**
  Main application module that imports and configures other modules and providers.
- **main.ts**
  The entry point of the NestJS application, bootstrapping the app.

Directories

- **auth**
  Handles authentication-related functionality, including login, registration, and token management.
- **bull-config**
  Configuration files for Bull queue management.
- **bull-board**
  Provides a UI for monitoring Bull queues.
- **chat**
  Manages chat functionalities, including messaging and chat rooms.
- **common**
  Contains shared utilities, guards, and pipes used throughout the application.
- **config**
  Configuration files and settings for different environments.
- **modules**
  Contains the core modules of the application, organizing related features and components.
- **s3**
  Integrations and services related to Amazon S3 for file storage.
- **spotify**
  Integration with the Spotify API for handling music-related features. (Excluding Spotify Auth)

- **tasks**
  Scheduled tasks and background jobs for the application.

How things work in NestJS

NestJS is built for an MVC architecture out of the box. It merges the backend and API layers into 1 server-side app.

Controllers

All endpoints / accessible paths to the application are defined in controllers. Eg. `GET /user` and `POST /user/settings` could be in `UserController.ts`

Services

Controllers are best left to handle how input and output data flows through the application. Thus, the complex logic expected from the endpoints & paths are best kept in services.

Business logic, such as checking if data is valid, managing complex operations on the data, database (and other external service) interactions, etc are stored in services.

Eg, `getUserInfo()` or `getUserRooms()` would be stored in `UserService.ts`

Modules

Given that controllers and services can be closely related and can also make use of other services, dependencies for them are stored in modules.

Modules in NestJS are used to organize the application into cohesive blocks of related functionality. They encapsulate controllers, providers (services), and other configuration elements, ensuring modularity and maintainability of the codebase.

Eg, `UserModule.ts` would link `UserService` and `UserController` together as well as manage their imports (shared or otherwise), make `UserController`'s endpoints to the rest of the application, allow for `UserService` to be able to be imported by other services/controllers, etc

Defining a Module A module definition looks like this:

```
@Module({
    imports: [PrismaModule],
    controllers: [UsersController],
    providers: [
        UsersService,
        PrismaService,
        DtoGenService,
```

```
        DbUtilsService,
        AuthService,
    ],
    exports: [UsersService],
})
export class UsersModule {}
```

1. **Imports** When a module uses another module's functionalities and dependencies, they should be imported. This establishes a hierarchical relationship where modules can depend on one another.

Anything that's external to the module must have its accompanying Module imported here.

2. **Controllers** By default, all modules are linked to the main `AppModule` and thus, any controllers added to the `controllers` list of a module makes them accessible when the server app is running.

3. **Providers** Any services created and used within the module should be added to the providers list as that they can be injected into controllers or other services for loose coupling.

This is for any services & providers defined internally / within the module. No external services (eg. `PrismaService`) should be placed here.

4. **Exports** Any components (like services/providers) used within the module and that should be made available for use in other modules should be added to the exports array. This allows them to be included in another module's providers without being concerned about the service's module structure.

Coding Standards

(TBA)

Conclusion

For any further assistance or questions, refer to the resources linked above or consult the official documentation of the respective libraries and frameworks.

Happy coding!