ChatGPT

Social Media Backend – Brandie Assignment

Introduction

This project is a small-scale **social media backend API** built for the Brandie Backend Engineer assignment. It provides core social networking features such as user following, follower/following lists, creating posts with text and media, and retrieving user feeds and posts [1]. The goal is to demonstrate a clean and well-structured implementation of these features using **Node.js 18+** and **TypeScript**, along with a lightweight authentication mechanism and appropriate data modeling.

Key highlights of this project include a RESTful API (Express.js) for all endpoints, a **JWT-based authentication** system, and the use of a robust database (either **PostgreSQL** or **Neo4j**) for managing user relationships and posts. The repository is containerized with Docker for easy setup, and the backend is deployed live for testing.

Live API Base URL: *<Deployment URL placeholder>*
Postman Collection: *<Postman link placeholder>*
GitHub Repository: *<Repo link placeholder>*

Features

This backend supports the following core features (as required by the assignment) [1]:

- **Follow/Unfollow Users:** Users can follow other users to create a social graph. A follow action adds the target user to the follower's "following" list and vice-versa. An unfollow action will remove that relationship. The system prevents invalid operations (e.g. following the same user twice or following oneself) by validating requests. These actions update the follower/following counts accordingly.

- **Followers & Following Lists:** For a given user, the API can retrieve the list of users who follow them (followers) and the list of users they are following. Results can be filtered or paginated as needed for large lists. This allows clients to display social connections (e.g., showing a user's followers or the people they follow).

- **Create Posts (with Text & Media):** Authenticated users can create new posts containing textual content and optional media input. Media could be an image or video; in this implementation, we accept an image URL (or a file upload path) as the media attachment for simplicity. Each post is associated with its author and timestamp. The API ensures the post is stored with all necessary details (e.g. content, media URL/path, author reference, creation time).

- **User Timeline (Feed):** Users can retrieve a personalized timeline/feed consisting of recent posts from all the users they follow. This feed is typically sorted by recency (most recent posts first). The timeline endpoint merges posts from followed users into a single feed, allowing the client to show a social media "home" feed. (For the scope of this project, all followed posts are retrieved on request; caching or pagination can be added if needed.)

- **User's Posts:** The API provides an endpoint to fetch all posts made by a specific user. This allows viewing a particular user's profile timeline or post history. It returns that user's posts (and only their posts), sorted by recency. This feature is useful for implementing user profiles where one can see everything a user has shared.

Each feature above has been implemented with attention to correctness and performance. For example, follow relationships are stored efficiently to allow quick lookups of follower/following lists, and appropriate database queries or graph traversals are used to assemble a user's feed.

Tech Stack

Language & Runtime: Node.js 18+ with TypeScript – providing modern JavaScript features and type safety.

Framework: Express.js (REST API) for routing and request handling, making it straightforward to define RESTful endpoints. *(Alternatively, the assignment allowed using GraphQL with Apollo Server, but this implementation uses REST for simplicity [2].)*

Database: PostgreSQL 15 or **Neo4j 4.x** for data persistence [3]. The project supports either a relational model or a graph model for the social data: - If using **Postgres** (relational database), the data is modeled with tables (for Users, Posts, and Follows) and relationships via foreign keys and join tables. - If using **Neo4j** (graph database), the data is modeled as a graph with nodes and relationships (User and Post nodes, connected by FOLLOWS and POSTED relationships).

(Note: Choose one of the above databases when running the project – by default, the project is configured for the one specified in environment variables. Both options are considered valid for this assignment.)

ORM/Database Library: Depending on the database choice, either an ORM or driver is used: - For Postgres, this could use a query builder/ORM (e.g. Knex, TypeORM, or Prisma) or the native `pg` client for executing SQL queries. - For Neo4j, the official Neo4j JavaScript driver is used to run Cypher queries and handle graph results.

Authentication: JSON Web Tokens (**JWT**) for stateless auth. The JWT approach was chosen as a lightweight scheme [4]. It involves issuing a signed token on user login and verifying it on each protected request.

Other Libraries: - **bcrypt** (or similar) for password hashing to securely store user credentials. - **dotenv** for loading environment variables. - **Joi/Celebrate** (optional) for validating request payloads (ensuring required fields like post content are present, email format for registration, etc.). - **cors** middleware to enable Cross-Origin Resource Sharing, allowing the API to be accessed by frontends or tools from different domains (in development, it's often set to `*` for ease of testing). - **Jest/Supertest** for testing (if automated tests were to be written, though not required, these would be used).

Dev Tools: - **Nodemon** or `ts-node-dev` for a hot-reloading development server. - **Docker** for containerization (Dockerfile and Docker Compose configuration included for containerized deployment). - **Postman** (for API testing via the provided collection) or GraphQL Playground if using GraphQL.

Architecture and Data Model

Data Model

The data model is designed to accurately represent a social networking domain while optimizing for common queries:

- **User:** Each user has a unique identifier (e.g., `id` or `username`), and fields such as name, email, password (hashed), etc. In a relational model, users are stored in a `Users` table (with `id` as primary key). In a graph model, users are nodes labeled `User` with unique properties (like `userId` or `email`). We enforce uniqueness for user identity (e.g., unique username or email) via database constraints or indexes.

- **Post:** Each post has an `id`, text content, optional media URL/path, timestamp, and a reference to the author (the user who created it). In Postgres, posts reside in a `Posts` table with a foreign key `author_id` referencing `Users(id)`. In Neo4j, posts are nodes labeled `Post` with properties (content, media, timestamp) and an incoming `POSTED` relationship from the authoring `User` node. We index relevant fields (e.g., post `id` and author reference) to speed up lookups.

- **Follow Relationship:** In Postgres, follows are modeled with a join table (e.g., `Follows`) that has a composite key of (`follower_id`, `followee_id`). Each entry represents a user (follower) following another user (followee). We ensure referential integrity with foreign keys, and often add an index or unique constraint on (`follower_id`, `followee_id`) to prevent duplicate follows and to efficiently query relationships. In Neo4j, a `FOLLOWS` relationship connects one User node to another User node. We add an index on the user nodes (like an index on `userId`) to quickly find start points for traversals. The follow relationship is directed and its inverse can be derived (if A FOLLOWS B, then B has A as a follower).

These tables/relationships enable the key queries: - Fetching a user's followers or following (a simple query on the Follows table or an outgoing/incoming relationship traversal in Neo4j). - Checking if one user follows another (one-row lookup or relationship existence check). - Retrieving a user's posts (query Posts by author_id, or traverse `POSTED` relationships from a User node). - Retrieving a feed (in SQL: join Follows with Posts to get posts from all followees; in Cypher: follow outgoing FOLLOWS from user to Post nodes).

The data model is kept minimal but can be extended (e.g., adding profile info, comments, likes, etc.). Proper **indexes** are used for performance, such as indexing user IDs and post timestamps (for sorting feeds by date), and ensuring fast relationship lookups (indexes on join table columns or Neo4j node properties) [5].

Project Structure

The project follows a modular structure for clarity and maintainability (all code resides in the `src/` directory):

```
src/
├── models/             # Schema definitions and database models (ORM models
or schema interfaces)
```

```
├── controllers/        # Request handlers (business logic for each route or
GraphQL resolver)
├── routes/             # Express route definitions mapping URLs to
controllers (ignored if using GraphQL)
├── services/           # Reusable services or helpers (e.g., authentication,
validation, etc.)
├── middleware/         # Express middleware (e.g., auth token verification,
request validation)
├── utils/              # Utility functions (e.g., error formatting,
pagination helpers)
└── index.ts / server.ts  # Entry point to start the server
```

(If using GraphQL, the structure would instead include schema definition files and resolver mappings rather than Express routes and controllers.)

This layered architecture ensures a clear separation of concerns: - **Routes** define the API interface (endpoints and HTTP methods). - **Controllers** handle the incoming requests, interact with models/services, and formulate responses. - **Models/Database** layer interacts with the database (running queries or calling ORM methods). - **Middleware** (like auth check or input validation) is applied to routes for cross-cutting concerns. - **Services/Utils** encapsulate logic that might be reused across controllers (for example, a function to generate JWT tokens or to handle password hashing).

This structure makes the codebase easier to navigate and expand. New features (like comments or likes) could be added by creating new models, controllers, and routes without affecting existing functionality. The folder layout was chosen to be intuitive and align with common Node.js best practices [6].

Design Decisions & Trade-offs

During development, several key decisions and assumptions were made to balance simplicity with functionality:

- **API Style (REST vs GraphQL):** The assignment allowed either REST or GraphQL [7]. We chose **RESTful API** with Express.js for this implementation because it's straightforward to set up and test with tools like Postman. REST endpoints clearly map to the required features and make it easy to demonstrate each capability. (Alternatively, a GraphQL schema could model users, posts, and follow relationships with queries like `feed` or `userPosts` and mutations like `followUser`, `createPost`. GraphQL would reduce round-trips by fetching complex data in one query, but given the small scope of this project, REST was sufficient and more familiar.)

- **Database Choice (Relational vs Graph DB):** We evaluated using PostgreSQL versus Neo4j for storing the social graph. **PostgreSQL** was chosen for the implementation due to its reliability, familiarity, and simplicity in a quick project setup. Modeling follows in a relational DB (via a join table) is straightforward and indexing ensures efficient lookups. On the other hand, **Neo4j** is naturally suited for graph relations like follows; it could simplify some queries (e.g., getting a feed via graph traversals). The trade-off was that introducing a new technology (Neo4j) might add setup overhead. Our design, however, keeps the data access layer abstract enough that switching to Neo4j in the future or supporting both would be feasible with moderate changes. We documented the data model for both options, and ensured our queries (SQL or Cypher) are optimized with indexes for performance [5].

- **Authentication Method:** We opted for **JWT-based authentication** as a lightweight scheme [4]. This choice allows the backend to remain stateless (no server-side session storage). The trade-off is that we must ensure the JWTs are properly secured (signed with a strong secret, have an expiration, and are used over HTTPS to prevent interception). We considered session cookies, but that would require managing sessions and is less convenient for a pure API service. With JWT, any server instance can authenticate requests as long as it has the secret, which fits cloud deployment. We decided to implement a simple login to issue tokens, and all protected routes (follow, unfollow, create post) require a valid `Authorization: Bearer <token>` header. For development, the token expiration might be set long, but in production we'd consider shorter expiry and refresh tokens (not implemented here due to scope).

- **User Registration & Seeding:** The assignment scope didn't explicitly mention user sign-up, but to test follow/post features, we needed a way to create users. We chose to implement a **register and login** endpoint to allow dynamic testing. Alternatively, a set of sample users could have been pre-seeded in the database for demonstration. We decided that including basic auth endpoints makes the project more self-contained and easier to evaluate. The registration process hashes passwords (using bcrypt) before storing, and login verifies the hash, returning a JWT on success. This approach follows security best practices (never storing plain passwords). For quick testing, one can also manually insert users in the database if needed, but the endpoints provide a cleaner solution.

- **Media Handling:** The assignment requires posts to support media content (images/videos). Implementing full file upload/storage in 48 hours can be complex, so we made a simplifying assumption: **media files are provided as URLs** (or base64 strings) in the post request. The `POST /posts` endpoint accepts a `mediaUrl` field (or similar) which could be a link to an image. This avoids the need for storing binary data on the server or integrating a storage service. In a real-world scenario, we would handle file uploads (to S3 or a static server) and store only references in the database. This is noted as a possible extension, but for now the approach covers the requirement in a basic form. The trade-off is that the client must upload the image elsewhere (or encode it) and provide a URL, but it keeps the backend simpler and within scope.

- **Feed Generation Approach:** To retrieve a user's timeline feed, our implementation queries the posts of all users that the current user follows, sorted by date. In SQL, this is done with a join between the Follows and Posts tables (filtering by follower and ordering by post timestamp). In Neo4j, this would be a traversal from the user to followed users to their posts. We chose an **on-the-fly feed assembly** for simplicity. The trade-off is that this can be less efficient if the following list or posts grow very large (e.g., pulling many records each time). In a production scenario, one might introduce caching or a feed generation service (fan-out on write) to optimize this. However, given the scope and expected data volume for an assignment, the straightforward approach is acceptable and clear in demonstrating functionality. We documented this decision and acknowledge that scalability could be improved with more time.

- **Error Handling & Validation:** We added basic error handling to the API. If a resource is not found (e.g., a user ID that doesn't exist when trying to follow), the API returns a 404 Not Found or a relevant error message. If a request is invalid (e.g., missing required fields, or trying to follow already followed user), it returns a 400 Bad Request with explanation. We considered using a validation library (like Joi) to enforce input schema, but in the interest of time, simple manual checks are in place for critical paths. All database operations are wrapped in try/catch to handle and log errors gracefully. This ensures the API fails gracefully and provides useful feedback for debugging and testing.

- **Security Considerations:** In addition to authentication and password hashing, we implemented other security best practices where possible. All SQL queries (if using Postgres) are parameterized or use an ORM to avoid SQL injection. We enabled CORS in a controlled manner, and set the JWT secret via environment variable (not hard-coded) to keep it confidential. If deployed, the service should be run behind HTTPS for encryption of tokens in transit. We also ensure that protected routes check the JWT and that we only allow actions that the user is authorized for (for example, you can only create a post as yourself, follow as yourself, etc., and the token's user ID is used for these operations server-side, ignoring any user IDs given in the request to prevent spoofing). Further security enhancements (rate limiting, audit logging, etc.) are noted for future improvements but were outside the immediate scope.

- **Ambiguities & Assumptions:** Where the requirements were open-ended, we made reasonable assumptions and documented them. For instance, we assume that all users can see each other's posts (no privacy settings) since none were specified. We assume a simple follower relationship (no additional states like follow requests). We also assume that deleting users or posts is out of scope – so, for example, if a user were removed, we have not implemented cascading deletes of their posts or follow relations (though referential integrity in SQL or optional Cypher scripts could handle it). Any such scenario would be handled with manual cleanup for now. These assumptions allowed us to focus on the core features within the time limit [8] [9].

Overall, our design choices were guided by simplicity, clarity, and meeting the assignment criteria. We have noted where trade-offs were made and how one might address them in a full production system.

Authentication

Authentication is handled via **JSON Web Tokens (JWT)** in this project. A user must first register an account or log in with valid credentials to obtain a token. The token is a signed JWT (using an HMAC SHA256 signature, with a secret key defined in environment config). This token contains the user's identifier and an expiration timestamp.

Key points of the JWT auth setup:

- **Registration & Login:** We provide `POST /api/auth/register` and `POST /api/auth/login` endpoints. On registration, the user provides details (e.g., username, email, password). The password is hashed using bcrypt before storing in the database, and the new user record is created. On login, the user provides their credentials; the API verifies the email/username and password against the stored hash. If valid, a JWT is generated for that user. For simplicity, the JWT payload includes the user's ID (and optionally username/email) and a short expiration (e.g., a few hours).

- **Using the Token:** The client must include the JWT in the `Authorization` header for any API calls that require authentication. The format is:

  ```
  Authorization: Bearer <token>
  ```

  On each such request, a middleware checks for the presence of the header and verifies the token's signature and expiry (using the JWT secret). If the token is missing or invalid/expired, the request is rejected with a **401 Unauthorized** error.

- **Protected Endpoints:** Follow/unfollow actions and creating posts are protected; they can only be performed by authenticated users. The server trusts the JWT to identify the user – for example, when calling `POST /api/posts`, the server uses the token's user ID as the author of the new post (ignoring any user id in the request body to prevent malicious override). Similarly, `POST /api/users/:id/follow` doesn't literally take the follower from the path; instead, it uses the logged-in user from JWT as the follower and the `:id` in the path as the followee.

- **Token Secrets & Config:** The JWT signing secret is defined in an environment variable (e.g., `JWT_SECRET`). It's crucial to keep this secret safe (not commit it to code). The token expiration duration may also be configured (e.g., `JWT_EXPIRES_IN`). In development, you might use a long-lived token for convenience, but in production a shorter expiry (with refresh mechanism) would be recommended.

- **Session Management:** We chose not to use server-side sessions or cookies, meaning the backend does not store session data. This makes scaling easier (no need for sticky sessions or session storage) and aligns with a stateless microservice design. The trade-off is that we rely entirely on the token – if a token is compromised or if we want to revoke a token, we'd need additional infrastructure (like a token blacklist or using short expiration tokens). For this assignment scope, JWT provides a simple and effective auth solution.

- **Testing the Auth:** To test protected routes via Postman, first call the login or register endpoint to get a token, then set the `Authorization` header for subsequent requests. The Postman collection (link provided above) includes this step. The API returns descriptive errors if authentication fails (e.g., "Invalid token" or "Token expired"), which helps in debugging auth issues.

Overall, the JWT authentication setup ensures that only verified users can perform sensitive actions, and it fits the "lightweight auth scheme" requirement of the assignment [4].

Setup Instructions (Local Development)

Follow these steps to set up and run the project on your local machine:

1. **Prerequisites:** Make sure you have the following installed:
2. **Node.js 18+** and npm (Node Package Manager).
3. Either **PostgreSQL 15** or **Neo4j 4.x** database installed and running locally. (You can also use Docker to run the database if preferred.)

4. (Optional) **Docker** if you plan to use the Docker setup instead of running directly on your host.

5. **Clone the Repository:**

```
git clone <repository-url.git>
cd <repository-folder>
```

6. **Install Dependencies:**
   Run npm to install all required Node.js packages.

```
npm install
```

This will install Express, database drivers/ORM, and other dependencies listed in `package.json`.

7. **Environment Configuration:**
   Create a `.env` file in the project root (you can copy `.env.example` if provided). This file will hold environment variables for configuration. At minimum, set the following variables:

8. **App settings:**
   `PORT=3000` (or any port number on which the server will run)
   `JWT_SECRET=<your_jwt_secret>` (a random secret key for signing JWTs)
9. **Database (PostgreSQL) settings:** (if using Postgres)
   `DB_HOST=localhost`
   `DB_PORT=5432`
   `DB_NAME=<your_db_name>`
   `DB_USER=<your_db_username>`
   `DB_PASSWORD=<your_db_password>`
   (Alternatively, you can set a single `DATABASE_URL=postgres://user:pass@host:5432/dbname` *)*
10. **Database (Neo4j) settings:** (if using Neo4j)
    `NEO4J_URI=bolt://localhost:7687`
    `NEO4J_USERNAME=<your_neo4j_user>` (default is usually `neo4j`)
    `NEO4J_PASSWORD=<your_neo4j_password>`

11. **Other settings:**
    `NODE_ENV=development` (for local development)
    (Add any other configuration, such as cloud storage keys if relevant, or debug flags as needed.)

12. **Database Setup:**
    Set up the database according to your choice:

13. **If using PostgreSQL:** Create a new database schema for the project (for example, via `createdb brandie_social`). Update the `.env` with the DB name and credentials. Run any migration or initialization script if provided (e.g., there might be an SQL script or an npm command like `npm run migrate` to create tables). If no explicit migration tool is used, the application code will create tables on the first run (depending on implementation). Ensure that the Postgres server is running and accepting connections on the host/port you configured.

14. **If using Neo4j:** Start your Neo4j server (community or enterprise edition). Make sure it's accessible at the `NEO4J_URI` you set (for local default, `bolt://localhost:7687`). No SQL migration is needed, but you may want to create a unique constraint on user nodes for `userId` or `username` to prevent duplicates (this can be done via the Neo4j Browser or an init script). For example:

```
CREATE CONSTRAINT unique_username IF NOT EXISTS
FOR (u:User) REQUIRE u.username IS UNIQUE;
```

(Adjust for your property names.) The application might also create needed indexes in code on startup if implemented to do so.

15. **Start the Server:**
    You can start the application in development mode or production mode:

16. **Development:** For example, if a dev script is set up with Nodemon, run:

    ```
    npm run dev
    ```

    This will start the server with hot-reloading (auto restarting on code changes). By default, it listens on port `3000` (or the `PORT` you set). You should see log output indicating the server is running and connected to the database.

17. **Production build:** To run without dev dependencies, first transpile the TypeScript to JavaScript:

    ```
    npm run build
    ```

    This will create a `dist/` folder with compiled code. Then start the server using Node:

    ```
    npm run start
    ```

    (Ensure you have set `NODE_ENV=production` and appropriate environment vars for production if needed.)

18. **Verify Running Service:**
    Once the server is running, you can test the health endpoint (if implemented) or simply try hitting the base URL. For example, open a browser or use curl/Postman to GET `http://localhost:3000/` (if the app has a base route or returns 404 that's fine). The console logs should indicate that the server received a request. You can then proceed to test specific API endpoints as described below in the **API Overview** section.

If everything is set up correctly, you should now have the backend running locally. Use the provided Postman collection (or your own requests) to create a user, log in, follow other users, create posts, and retrieve feeds, verifying that each feature works as expected.

Troubleshooting: If you encounter errors on startup, check the following: - All environment variables are correctly set (especially DB connection info and JWT secret). - The database is running and accessible (for Postgres, try connecting with `psql` or a GUI; for Neo4j, open the Neo4j Browser to confirm it's up). - The port is not already in use (if 3000 is busy, either stop the other service or change the `PORT`). - Dependencies are installed without errors (if `npm install` failed, there might be a network issue or a compatibility issue to resolve). - If using Neo4j and you see authentication errors, ensure the username/password in `.env` match those of your Neo4j instance (the default password for Neo4j is typically `neo4j` on first run, which you should have changed as prompted by Neo4j).

With a correct setup, the local server should work and be ready for development or testing.

Docker Setup

For convenience and to ensure consistency across environments, the project includes a Docker setup. This allows you to run the application and the database in containers without manually installing Node or the database locally. There are two main ways to use Docker here:

1. Using Docker Compose (Recommended)

The repository provides a `docker-compose.yml` file that defines both the application service and a database service (PostgreSQL or Neo4j). By default, the compose file is configured for the database that the project is primarily using (you can switch the service as needed):

- **App Service:** Uses the Dockerfile in the repo to build a lightweight Node.js container running the application.
- **Database Service:** Either a Postgres 15 image or a Neo4j 4.x image. The compose file sets up default credentials and ports for these.

Steps to run with Docker Compose:

1. Ensure Docker is installed and running on your system. (You can verify by running `docker --version` and `docker-compose --version`.)

2. Build and start the services using compose:

   ```
   docker-compose up --build
   ```

   The `--build` flag ensures the image is rebuilt (you might omit it on subsequent runs if no changes). This command will:

3. Build the custom image for the Node app (installing dependencies and compiling the code inside the container).
4. Pull the database image (if not already pulled) and create a container for it.

5. Start both containers and network them together.

6. Wait for the services to come up. Docker Compose will stream the logs of both containers. Give the database a few seconds to initialize. The Node app might attempt to connect immediately; if it logs a database connection error at first, it may retry or you might need to restart the app container once the DB is ready.

7. Once running, the API should be accessible at the host machine on the port configured (e.g., `http://localhost:3000`). The database will be running in its container (Postgres default port 5432, Neo4j default 7687 & 7474 for browser). The compose file likely maps the database port to the host as well, for easy access.

8. **Environment Variables:** The Docker Compose setup is configured to pass environment variables to the containers. This may be done via a `.env` file or environment section in the compose YAML. Ensure that the environment values (DB password, JWT secret, etc.) in the compose config are correct. For Postgres, the compose file will set `POSTGRES_USER`,

`POSTGRES_PASSWORD`, and `POSTGRES_DB` which initialize the database. The app container needs corresponding variables (DB connection string) to connect to that DB service. In the provided compose, these should be aligned, but double-check if you make changes.

9. To stop the containers, press `Ctrl+C` in the terminal running compose, then run:

```
docker-compose down
```

This will stop and remove the containers. The database data may be persisted in a Docker volume (depending on the compose setup) so that data isn't lost between runs. If you want to start fresh, you might remove the volumes via `docker-compose down -v` (be cautious, as this deletes data).

Switching Database in Docker Compose: If you want to use Neo4j instead of Postgres (or vice versa), you might have separate compose files or you can modify the existing one. For example, if the default is Postgres, to use Neo4j you would: - Comment out or remove the Postgres service and replace it with a Neo4j service in the YAML (using the official `neo4j` image, setting `NEO4J_AUTH` env for creds, etc.). - Adjust the app service environment variables to point to the Neo4j URI and creds. - Then run `docker-compose up` with this configuration. Alternatively, use a different compose file (e.g., `docker-compose.neo4j.yml`) and run `docker-compose -f docker-compose.neo4j.yml up`.

2. Using Docker Manually (Docker CLI)

If you prefer or need to run without docker-compose, you can manually use Docker to run the app:

- **Build the App Image:**

```
docker build -t brandie-backend:latest .
```

This uses the Dockerfile to create an image named "brandie-backend".

- **Run the Database Container:** If using Postgres, for example:

```
docker run -d --name brandie-postgres -e POSTGRES_USER=postgres -e
POSTGRES_PASSWORD=postgres -e POSTGRES_DB=brandie_db -p 5432:5432
postgres:15
```

This will start a Postgres container with a user/password "postgres" and a database "brandie_db", exposing it on localhost:5432. For Neo4j, a similar command can be used:

```
docker run -d --name brandie-neo4j -e NEO4J_AUTH=neo4j/test123 -p 7474:
7474 -p 7687:7687 neo4j:4.4
```

(This sets a neo4j password to "test123" and exposes the bolt and web ports.)

- **Run the App Container:**

```
docker run -d --name brandie-app -p 3000:3000 --env-file .env --link
brandie-postgres brandie-backend:latest
```

In this command, we assume you're using the .env file for configuration. The `--link` flag is one way to connect containers (deprecated in favor of networks, but quick for a test). Alternatively, connect both containers to a common Docker network and use that for communication. Make sure the DB host in your `.env` is set to the service name (`brandie-postgres` if using link, or the network alias if using networks) instead of `localhost`, because inside the container, localhost refers to the container itself.

- The app container should start and connect to the database container. Check logs using `docker logs -f brandie-app` if needed to debug.

This manual approach is more involved, but it is useful if you want to run the components separately or in environments where docker-compose is not available.

Notes on Docker configuration:

- The provided **Dockerfile** is configured to:
- Use a Node 18 base image (alpine variant for smaller size if possible).
- Set the working directory, copy package files, install dependencies, copy source code, build the TypeScript (if we do build inside Docker), and then start the server.
- It exposes port 3000 (as defined by the app).

- It uses a multi-stage build (if configured) to reduce image size by not including dev dependencies or unneeded files in the final image.

- **Docker Compose** handles setting up the environment. The compose file may mount volumes for live reload (in dev) or for database persistence. In a production scenario, you might not mount the source code (you would just run the built image), but for dev you can mount code and use nodemon inside the container for hot reload.

- If you encounter issues with Docker (e.g., permission issues, or the app cannot connect to DB), ensure that:

- The environment variables in Docker are correct (you might need to adjust `HOST` for DB as noted).
- The containers are on the same network (compose does this automatically).
- No other service is occupying the ports on the host.
- For Neo4j, the container needs to set `NEO4J_AUTH` to avoid it expecting interactive password change. Also, the Neo4j browser UI will be available on localhost:7474 when running via Docker if you want to inspect the data.

Using Docker is optional but recommended for consistency. It ensures that the app runs in the same environment everywhere. It also makes deployment to cloud services easier (just deploy the container image). The provided Docker setup aims to earn the bonus points for Dockerization and can be used in the deployment as described next.

API Overview

Below is an overview of the API endpoints and their functionality. All endpoints are prefixed with a base path (e.g., `/api` or similar) as configured in the server. The examples assume the base URL is simply shown without `/api` for brevity. Adjust accordingly if a prefix is used.

Authentication & User Management

- **POST** `/auth/register` – Register a new user account.
  Request: JSON body with user details, for example:

  ```
  { "username": "alice", "email": "alice@example.com", "password":
  "securePass123" }
  ```

  The server hashes the password and creates a user.
  Response: On success, returns 201 Created with the created user's info (often excluding the password) and possibly a JWT token (or you might require a separate login). In this implementation, it returns a JWT upon registration for convenience.
  Errors: 400 if missing fields or username/email already taken.

- **POST** `/auth/login` – Authenticate an existing user.
  Request: JSON body with credentials:

  ```
  { "email": "alice@example.com", "password": "securePass123" }
  ```

  (Or it could accept username instead of email depending on implementation.)
  Response: 200 OK with a JSON containing the JWT token (and possibly user info) if credentials are valid. Example:

  ```
  { "token": "<jwt_token>", "user": { "id": 1, "username":
  "alice", ... } }
  ```

  Errors: 401 Unauthorized if credentials are wrong, 400 if input is invalid.

(Note: There might not be other user management endpoints like update profile or delete user within this assignment scope.)

Social Graph (Follow/Unfollow)

- **POST** `/users/{id}/follow` – Follow a user.
  Description: Authenticated user (let's call them A) sends a request to follow user with id `{id}` (call that user B). This creates a follow relationship where A follows B.
  Authentication: Required (JWT token).
  Response: 200 OK if successful (or 201 Created if treating it as a new resource). The response may include a message or the updated follow status. For example:

```
{ "message": "Now following user 5." }
```

or return the follow object/relationship created.
Errors: 401 if not authenticated. 404 if the target user doesn't exist. 400 if trying to follow an invalid target (e.g., themselves or duplicate follow). If already following, the API might either return an error or simply ignore (idempotent follow). Our implementation treats duplicate follow as a no-op (or returns a 200 with no change).

- **DELETE** `/users/{id}/follow` – Unfollow a user.
  Description: Authenticated user (A) requests to unfollow user B (with id `{id}`). This removes the follow relationship if it exists.
  Authentication: Required.
  Response: 200 OK on success (or 204 No Content). We might return a message:

```
{ "message": "Unfollowed user 5." }
```

Errors: 401 if unauthorized, 404 if the target user doesn't exist. If A wasn't following B to begin with, we may return 400 or 404 (or also treat as no-op success). In our implementation, we check the relationship and return 200 even if there was nothing to unfollow (idempotent behavior).

- **GET** `/users/{id}/followers` – Get followers of a user.
  Description: Returns the list of users who are following the user with id `{id}`. This can be accessed by anyone (it's usually public data).
  Response: 200 OK with JSON array of user profiles (or IDs) who follow the target. For example:

```
{ "userId": 5, "followers": [ { "id": 1, "username": "alice" }, { "id":
2, "username": "bob" } ] }
```

The exact fields returned for each follower might be limited (could just be username and id). If the user has no followers, the list will be empty. We might include a count as well.
Errors: 404 if the user `{id}` is not found. No auth required (or if we require auth, any logged-in user can view others' followers, we didn't implement privacy levels).

- **GET** `/users/{id}/following` – Get following list of a user.
  Description: Returns the list of users that the user with id `{id}` is following. Similar behavior to followers endpoint.
  Response: 200 OK with JSON array of users that the target user follows.
  Example:

```
{ "userId": 5, "following": [ { "id": 10, "username":
"charlie" }, ... ] }
```

Errors: 404 if user not found. Publicly accessible in our implementation.

(Note: If user IDs are not ideal for public exposure, these endpoints could use usernames in the path instead. E.g., `/users/alice/followers`. Our implementation uses IDs for simplicity, assuming IDs are not sensitive and perhaps not easily guessable if they were GUIDs. This can be adjusted based on preference.)

Posting and Feeds

- **POST** `/posts` – Create a new post.
  Description: Creates a post for the authenticated user.
  Authentication: Required.
  Request: JSON body with post content, for example:

```
{ "text": "Hello, world!", "mediaUrl": "http://example.com/image.png" }
```

  The `text` field is required (non-empty content), and `mediaUrl` is optional (if provided, the backend will associate it with the post). Alternatively, if a file upload was supported, this would be a multipart form request, but as noted, we are using URL for media.
  Response: 201 Created on success. Returns the created post's data, e.g.:

```
{
  "post": {
     "id": 123,
     "text": "Hello, world!",
     "mediaUrl": "http://example.com/image.png",
     "authorId": 5,
     "createdAt": "2025-06-17T09:12:34Z"
  }
}
```

  The exact response may vary; it could just return the post ID or the whole post object with populated fields as shown.
  Errors: 401 if not authenticated (must have JWT). 400 if validation fails (e.g., text is missing or too long, or media URL is malformed). There might also be a payload size limit if not handled (but since we're mostly text and URLs, not an issue here).

- **GET** `/users/{id}/posts` – Get all posts by a specific user.
  Description: Retrieves the list of posts authored by the user `{id}`. This is typically used to display a user's profile with all their posts.
  Authentication: Not strictly required (one user can view another's posts) unless we wanted to restrict it. In this project, it's open or at least available to any logged-in user.
  Response: 200 OK with JSON array of posts. For example:

```
{
  "userId": 5,
  "posts": [
     { "id": 123, "text": "Hello, world!", "mediaUrl": "...",
"createdAt": "...", "authorId": 5 },
     { "id": 124, "text": "Another post", "mediaUrl": null,
"createdAt": "...", "authorId": 5 }
```

```
  ]
}
```

The posts are usually sorted by `createdAt` descending (newest first). If the user has no posts, the array will be empty.
Errors: 404 if the user `{id}` doesn't exist. (If the user exists but no posts, it's 200 with empty list.)

- **GET** `/feed` (or `/users/{id}/feed`) – Get the timeline feed for the current user.
Description: Returns a combined feed of recent posts from all users that the authenticated user is following. This endpoint uses the current logged-in user's ID (from JWT) to determine whose posts to fetch, so it may not need an explicit `{id}` in path (we assume the feed is always for the requesting user). For clarity, we use `/feed` for the logged-in user's feed.
Authentication: Required (because feed is personalized).
Response: 200 OK with a JSON array of posts from followed users, typically sorted by time (most recent first). Example:

```
{
  "userId": 5,
  "feed": [
    { "id": 130, "text": "Post by someone I follow", "authorId": 8,
"authorUsername": "bob", "createdAt": "2025-06-17T08:00:00Z" },
    { "id": 129, "text": "Another followed post", "authorId": 2,
"authorUsername": "charlie", "createdAt": "2025-06-17T07:45:00Z" }
  ]
}
```

We include each post along with info about its author (perhaps username or id) so the client can display who made the post. This is effectively the "home timeline" of the user with id 5 in this example.
Errors: 401 if not authenticated. (No 404 here because it's not user-specific beyond the token; if the user isn't following anyone or no posts are available, it returns 200 with an empty feed list.)

Other Considerations

- We might have a **GET** `/users/{id}` to fetch a user's profile info (like username, follower count, following count, post count, etc.). This wasn't explicitly required, but could be useful. If implemented, it would return basic public profile data and maybe some stats (some data can also be derived by counting posts or followers via other endpoints).

- All timestamps are in ISO 8601 UTC format by default (as is common in JSON APIs).

- The API follows conventional HTTP status codes for success and error conditions to be easily understood by clients.

- If GraphQL were used, the equivalent would be:

- Queries: `feed: [Post]`, `userPosts(userId: ID!): [Post]`, `followers(userId: ID!): [User]`, `following(userId: ID!): [User]`, `user(id: ID!): User`.

- Mutations: `followUser(targetUserId: ID!): FollowStatus`, `unfollowUser(targetUserId: ID!): FollowStatus`, `createPost(text: String!, mediaUrl: String): Post`, `register(...): AuthPayload`, `login(...): AuthPayload`.
- And types like `User`, `Post`, etc., with relations (if using Neo4j with GraphQL, could even auto-resolve relations). Authentication in GraphQL would be handled via context (reading the Authorization header and verifying JWT for protected mutations/queries).

The above endpoints give a comprehensive interface to the social media backend. Clients (like a frontend app or Postman for testing) can use them to create accounts, follow others, make posts, and retrieve the necessary data to display a social feed and user profiles.

(For detailed request/response examples and to test the API, refer to the Postman collection provided. It contains predefined requests for each endpoint with example data.)

Test Cases

Below is a list of functional test cases that cover the critical features of the system. These test scenarios outline what is being tested and the expected outcomes, ensuring the backend meets the requirements and handles edge cases properly. *(Note: These are not automated tests, but rather scenarios that one could manually test or write tests for. They demonstrate understanding of the expected behavior.)*

1. **User Registration and Login:**
   Test: Register a new user and then log in with the same credentials.
   Expected Outcome: Registration returns a success response (201 Created) with a JWT for the new user. Logging in with the correct credentials returns 200 OK and a JWT. Logging in with an incorrect password for that user should return 401 Unauthorized.

2. **Follow another user:**
   Test: User A (authenticated) attempts to follow User B.
   Expected Outcome: The follow action succeeds (response 200 or 201). After this, User B should appear in User A's following list, and User A should appear in User B's followers list. Both users' follow counts (if accessible via profile or separate endpoint) should reflect the new connection. The API should not allow A to follow B if A is not authenticated (that should return 401).

3. **Unfollow a user:**
   Test: User A, who is currently following User B, initiates an unfollow action.
   Expected Outcome: The unfollow succeeds (200 OK). User B is removed from A's following list, and A is removed from B's followers list. Follow counts decrement appropriately. If A tries to unfollow B again (when already not following), the system should handle gracefully (either no change with a success status or an informative error, but no crash or new follow entry created).

4. **Prevent self-follow and duplicate follow:**
   Test: (a) User A attempts to follow themselves. (b) User A attempts to follow User B when A is already following B.
   Expected Outcome: (a) The API should reject the self-follow request (400 Bad Request, with a message like "Cannot follow self"). (b) The API should prevent duplicate follows; either by returning a 400 stating "Already following" or by silently ignoring and returning a success without duplicating the relationship. In either case, after the operation, there should still be only one follow relationship from A to B.

5. **Followers/Following list retrieval:**
   Test: After some follow relationships are established (e.g., A follows B and C; D follows A), retrieve A's followers and following lists via the API.
   Expected Outcome: The `GET /users/A/followers` should list D (and anyone else who follows A). The `GET /users/A/following` should list B and C. The counts of these lists should match the number of relationships. If A has no followers or is following no one, the respective endpoints should return an empty list (with 200 OK).

6. **Create Post:**
   Test: An authenticated user A creates a new post with text (and optionally a media URL).
   Expected Outcome: The post is successfully created (201 Created). The response contains the new post's data (id, content, media URL, author, timestamp). The post should be retrievable afterward via `GET /users/A/posts` (it should appear in A's own posts list) and also via any followers' feed. If A is not authenticated (missing or invalid token), the post creation request should be denied with 401 Unauthorized.

7. **User's Posts Listing:**
   Test: User A has created several posts. Retrieve the list of A's posts via `GET /users/A/posts`.
   Expected Outcome: The response returns 200 OK with all posts authored by A, sorted from newest to oldest. The number of posts matches what was created. Each post item should correctly show A as the author. If A has no posts, the returned list is empty. If the requested user ID does not exist, expect a 404 Not Found.

8. **Timeline Feed Content:**
   Test: User A follows users B and C. B and C have each created some posts. User A retrieves their feed (`GET /feed` as A).
   Expected Outcome: The feed includes posts from **both** B and C, combined and sorted by time (recent first). No posts from users other than B or C should appear. If B or C make a new post, a subsequent fetch of the feed should include that new post. If A is not following anyone, the feed should be empty (or possibly could include A's own posts depending on implementation, but in our design feed is purely others' posts). The feed endpoint should require A to be authenticated; if not, return 401.

9. **Timeline Feed Refresh after Unfollow:**
   Test: Continuing from the above scenario, User A unfollows User B, then fetches the feed again.
   Expected Outcome: Any posts from B should no longer appear in A's feed after the unfollow. Posts from C remain. This ensures the feed is dynamically reflecting the current follow graph. (If a post from B still appears, it indicates the unfollow didn't properly update A's follow list in the system.)

10. **Unauthorized Access Attempts:**
    Test: Attempt to access protected endpoints without a valid token – for example, calling `POST /posts` or `POST /users/XYZ/follow` without providing JWT, or with a malformed/expired token.
    Expected Outcome: The API should return 401 Unauthorized for each of these requests. The response may include an error message like "Authentication required" or "Invalid token". No action (post creation or follow) should occur without login. This confirms that the middleware security is in place.

11. **Invalid Data Handling:**
    Test: (a) Try to create a post with an empty text field. (b) Try to register with an invalid email

format or too short of a password. (c) Use an invalid user ID in a follow or unfollow request (e.g., an ID that doesn't exist in the database).
Expected Outcome: (a) The post creation should be rejected with 400 Bad Request, e.g., "Post content cannot be empty." (b) Registration should validate inputs and return 400 with messages like "Email format is invalid" or "Password must be X characters long" as applicable. (c) Following a non-existent user should result in a 404 Not Found error; the system should not create any follow relationship if the target user isn't real. These tests ensure robust input validation and error responses.

12. **Data Consistency Checks:**
Test: After a series of operations (multiple users registering, following each other, creating posts), manually verify the data consistency via multiple endpoints. For example, if A follows B and B follows C, ensure that A's following list shows B, B's followers list shows A; B's following shows C, C's followers shows B. If B created 2 posts and C created 1 post, and A follows B and C, A's feed should show 3 posts total.
Expected Outcome: All the relationships and posts cross-check correctly. There are no duplicate follows, no missing entries, and no mismatches (like a follower count not matching actual list length). This test is more of a comprehensive integration scenario to verify everything works together as expected.

Each of these test cases can be executed with tools like Postman or automated in a testing framework. They cover positive paths (expected use) and negative paths (error conditions), ensuring the backend handles each situation gracefully and in line with the requirements. These tests demonstrate the system's correctness and robustness in managing users, relationships, and posts.

Deployment Instructions

The backend is deployed and hosted so that it can be accessed remotely (per the assignment's requirement for a hosted API). This section provides details on how the deployment was done and how you can deploy the application yourself.

Current Live Deployment: The API is currently hosted at *<Live API URL>* (for example, this could be on a service like Heroku, Render, AWS, etc.). This live instance is connected to a cloud database and is running the latest build of the code from this repository. You can use the live base URL along with the endpoints in the API Overview to interact with the system (e.g., register a user, then follow, etc., on the live environment). The Postman collection is configured to use this base URL for convenience when testing.

Deployment Steps (Generalized):

The application can be deployed to any platform that supports Node.js or Docker. Here are general steps that were followed (assuming a Docker-based deployment on a cloud service for reproducibility):

1. **Set up a Production Database:** Decide whether to use Postgres or Neo4j in production, and provision the database:
2. For Postgres: Use a managed PostgreSQL service (like Heroku Postgres addon, AWS RDS, ElephantSQL, etc.) or run a Postgres Docker container on your server. Ensure you have the database URL and credentials.

3. For Neo4j: Use a hosted Neo4j service (Neo4j Aura) or run a Neo4j Docker container. Ensure connectivity and credentials. In the current deployment, we used *<Postgres/Neo4j>* as the

database. The connection string and credentials were configured as environment variables on the server.

4. **Configure Environment Variables:** On the hosting platform, set the necessary env vars (similar to the `.env` used locally). This includes `JWT_SECRET` (use a secure, long random string in production), `PORT` (if needed, though many platforms set this automatically), and all the DB connection settings. Also set `NODE_ENV=production`. Double-check that no sensitive credentials are left at defaults (e.g., don't use "postgres/postgres" in production, use a strong password or the managed DB's credentials).

5. **Build and Bundle the Application:** If using Docker:

6. Build the Docker image for the app (as described in Docker Setup). If using a platform like Heroku which auto-builds from source, it will detect the Dockerfile or use buildpacks. For a platform like AWS ECS or DigitalOcean, you might build the image and push to a registry (like Docker Hub or ECR).
7. The provided Dockerfile is production-ready (it runs `npm run build` and then `npm run start` in a slim Node runtime). Ensure you've set it to use a smaller base image for production (alpine variant) to reduce image size.

If not using Docker: - Install Node.js on the server. - Pull the code (e.g., via git clone or an artifact). - Run `npm install` and then `npm run build`. - Start the app with `npm run start` (or better, use a process manager like PM2 or the platform's node start mechanism). - Make sure to handle restarts on crashes and serve logs.

1. **Deployment Platform:** Choose a hosting method:
2. **Heroku:** Create a Heroku app, add a Postgres addon (if using Postgres), set config vars (in Heroku dashboard or CLI), and push the code. If using Docker, you can use Heroku Container Registry to push the image.
3. **Render.com or Railway.app:** These platforms can auto-deploy from GitHub. Provide the Dockerfile or use their Node environment. Set environment variables in their dashboard. They often provide managed Postgres as well.
4. **AWS ECS/Fargate:** Use Docker to containerize, push to ECR, then create an ECS service. Or use AWS Elastic Beanstalk for a simpler Node deployment (Beanstalk can handle Docker or just Node).
5. **DigitalOcean App Platform / Droplet:** If using App Platform, similar to Heroku/Render. If using a Droplet (VM), you'd manually set up Docker or Node and run the app.

6. **Others:** The app is fairly standard Node.js; it can run on Azure App Service, Google Cloud Run, Kubernetes cluster, etc. The key is ensuring environment configs and connecting to the DB.

7. **Domain and HTTPS:** If required, configure a custom domain and HTTPS. Many platforms (like Heroku, Render) provide a default domain (e.g., `myapp.herokuapp.com`) which is already HTTPS. For a custom domain, follow the platform instructions to point DNS and get an SSL certificate.

8. **Testing the Deployed API:** After deployment, use Postman or curl to test the live API. Go through the critical flows: create a user, log in, follow another user, create posts, fetch feed. Ensure that the responses are correct. Monitor logs on the server for any errors (e.g., database connection issues or unhandled exceptions). In our case, we tested the live endpoint *<Live API URL>* to verify everything working as expected.

9. **Scaling & Monitoring:** Though not necessarily needed for the assignment, note that this backend can be scaled horizontally if stateless (which it is, thanks to JWT). If using Docker, you can run multiple containers behind a load balancer. Ensure the database can handle the connections or use a connection pooler if needed. Monitor performance metrics and set up basic alerts if this were a long-running service.

Deployment-specific notes for this project:
- The backend is currently running on <*Platform X*> with <*N*> dynos/instances. We chose this platform for its simplicity in deploying Dockerized applications and free tier availability (important for an assignment demo).
- We used <*Service Y*> for the database, with the following configuration: <*e.g., hobby tier Postgres with 1GB storage*>. The connection string is stored securely and not exposed.
- Environment variables on the host were configured through the platform's interface to match those in development. No development secrets (like dummy JWT secret) were used in production; everything was appropriately set to production values.

- We have not implemented CI/CD for this assignment due to time, but ideally, a GitHub Actions pipeline could build the Docker image and deploy to the host on new commits, ensuring a smooth delivery process.

By following the above steps, you can replicate the deployment or set up the backend on your preferred hosting service. The key is to ensure configuration parity between local and production (especially database and JWT secret) and to use the Docker setup for consistency. With the application running in the cloud, evaluators and team members can interact with the API using the documented endpoints and verify the functionality in real-time.

Feel free to reach out or create an issue if you have any trouble setting up or running the project. The README covers the essential information, but additional comments are provided in the code for clarity on implementation details. Happy testing!

---

1 2 3 4 5 6 7 8 9 BrandieBackendEngineerAssignment2025.pdf
file://file-Jx4zrqrwbYAUJXhKDzRH8R