# **Documentation for Project Savorly: A Java Spring Boot REST API + React/Vite/TypeScript Frontend Application for Recipes**

**Members: Sean Darbyson, Amine Mankai, Fei Yang**

[**Project Savorly Documentation: Java Spring Boot REST API + React/Vite/TypeScript Frontend Application** 1](#_Toc189332950)

[**1. Overview** 3](#_Toc189332951)

[**2. Tech Stack & Project Setup** 3](#_Toc189332952)

[**3. System Architecture** 7](#_Toc189332953)

[**4. Dependencies & Implementation Details** 7](#_Toc189332954)

[**4.1 Dependencies** 7](#_Toc189332955)

[**4.2 Implementation Overview** 7](#_Toc189332956)

[**4.3 Layers Details** 8](#_Toc189332957)

[**5. API Design** 12](#_Toc189332958)

[**5.1 User API** 12](#_Toc189332959)

[**5.2 Recipe API** 13](#_Toc189332960)

[**5.3 Favourite API** 15](#_Toc189332961)

[**5.4 Rating API** 15](#_Toc189332962)

[**5.5 Review API** 16](#_Toc189332963)

[**5.6 Unit Test classes** 17](#_Toc189332964)

[**6. Database Design** 18](#_Toc189332965)

[**6.1 Users Table** 18](#_Toc189332966)

[**6.2 Recipes Table** 18](#_Toc189332967)

[**6.3 Favourites Table** 19](#_Toc189332968)

[**6.4 Ratings Table** 19](#_Toc189332969)

[**6.5 Reviews Table** 19](#_Toc189332970)

[**6.6 Tags Table** 20](#_Toc189332971)

[**7. Frontend Structure (React)** 20](#_Toc189332972)

[**8. Vector Search Implementation** 20](#_Toc189332973)

[**8.1 Ollama Installation** 21](#_Toc189332974)

[**8.2 Python scripts for Embedding vectors** 22](#_Toc189332975)

[**9. Hosting Solutions** 23](#_Toc189332976)

[**10. Conclusion** 23](#_Toc189332977)

## **1. Overview**

This project is a full-stack web application using **Spring Boot** as the backend and **React.js** as the frontend. It implements a **Recipe Management System** that supports user registration, login, and recipes view/creation/updating/deletion/collecting/rating. Users can also search for similar recipes.

## **2. Tech Stack & Project Setup**

* **Backend (REST API)**: Spring Boot, Spring Security, JPA (Hibernate), JWT, Junit5
* **Frontend**: React.js, React Router, Bootstrap
* **Database**: PostgreSQL (with Pgvector),
* **Deployment**: AWS S3 Bucket and RDS, Railway Frontend/Backend

#### 2.1 Project Setup

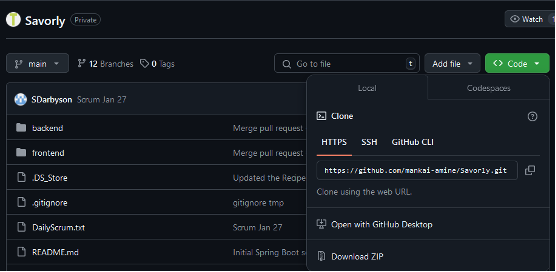
Before you begin, we recommend installing **Visual Studio Code** for Typescript development, **IntelliJ IDEA** for Java Spring Boot development, and **GitHub Desktop** for keeping track of branches and resolving any merge conflicts.

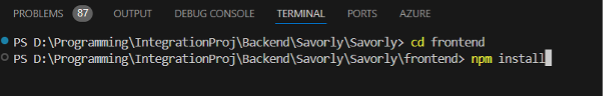
You’ll also need **NPM (Node Package Manager)** to install packages for React.

To set up the project, you can start by heading over to our [GitHub Repository](https://github.com/mankai-amine/Savorly/tree/main) and cloning it.

After you’ve cloned the repository, open the project is VSCode and open the terminal by pressing “Ctrl” and the tilde “ ` ” key at the same time. In the terminal, “cd” into the frontend folder. (as in pic.1)

Then run “npm install” to get the required packages for running the project. (as in pic.2)





The frontend should now be set up.

For the backend, open the project in IntelliJ and let it automatically install the packages it needs to run the software. Once that is done, you’ll need to add a “.env” file to the root of the project, and a “.env” inside the frontend folder, and fill them with the following variables (please reach out to us if you need assistance with any of these. Also note that the database will be set up in the next step.):

**ROOT .env:**

**For authentication:**

JWT\_EXPIRATION (int),

JWS\_SECRET(length should be longer than 32 for SHA-256)

**For PostgreSQL (with Pgvector plug-in):**

POSTGRES\_HOST=

POSTGRES\_PORT=

POSTGRES\_DB=

DB\_USER=YOUR\_DB\_USER

DB\_PASSWORD=YOUR\_DB\_PASSWORD

**For AWS S3 storage service:**

AWS\_ACCESS\_KEY\_ID,

AWS\_SECRET\_ACCESS\_KEY,

S3\_BUCKET\_NAME

**For OpenAi Embedding model accessing:**

spring.ai.openai.embedding.api-key(mandatory)

spring.ai.openai.api-key(optional)

**For email sending:**

spring\_mail\_host

spring\_mail\_name

spring\_mail\_password

spring\_mail\_port

spring\_mail\_properties\_mail\_smtp\_auth (default value true)

spring\_mail\_properties\_mail\_smtp\_starttls\_enable (default value true)

**For CORS:**

ALLOWED\_CORS\_URL

**FRONTEND .env:**

VITE\_API\_URL

VITE\_UPLOAD\_URL

#### 2.2 Database Setup

Now that these aspects are set up, let’s set up the Postgres database. This can take a bit of extra configuration due to the Vector plugin, which we’ll outline below.

Most of us already have Visual Studio installed from a previous course, but you’ll need to make sure to install some C++ packages on it for the Vector plugin.

**Select “Change” from the Programs and Features menu when right clicking on Visual Studio.**

**A screenshot of a computer

Description automatically generated**

**Add the packages for “Desktop development with C++” and confirm the installation details.**

**A screenshot of a computer

Description automatically generated**

Installing all these packages may take 15 minutes or more. After they complete, you’ll want to install PostgreSQL 16 if you have not already. Note that PGSQL 17 is not as compatible with the Vector plugin, so please make sure to grab 16!

Once you have PGSQL 16 installed, create a database for the project and then run the following command on it in the query client: CREATE EXTENSION vector;

When using the OpenAI embedding, ensure the dimensions of the vector table is 1536. If that’s not set properly, the Spring Boot app will encounter an error.

## **3. System Architecture**

┌───────────────┐ ┌───────────┐ ┌─────────────────┐  
│ React Frontend│ → │ REST API │ → │ PostgreSQL DB │  
└───────────────┘ └───────────┘ └─────────────────┘

## **4. Dependencies & Implementation Details**

This section describes the main dependencies used in this project and key implementation details.

### **4.1 Dependencies**

The project is built using Spring Boot, and the key dependencies are:

* Spring Boot Framework - spring-boot-starter-web, spring-boot-starter-data-jpa
* Database – PostgreSQL – For PgVector - hibernate-vector
* Security - Spring Security, JWT authentication - jjwt-api, jjwt-impl, jjwt-jackson
* Embedding Model Accessing   
  – For Ollama - spring-ai-ollama-spring-boot-starter  
  – For OpenAi - spring-ai-openai-spring-boot-starter
* Logging - spring-boot-starter-logging
* PDF Generation - itextpdf
* Cloud Storage - AWS S3 SDK for Java - aws-java-sdk-s3, s3, sdk-core, apache-client, sts
* SMTP - spring-boot-starter-mail

The full list of dependencies can be found in the pom.xml file.

### **4.2 Implementation Overview**

The project follows a **layered architecture**:

* **Controller Layer** (org.styd.intproj.savorly.controller)

Handles **HTTP requests** and serves as the entry point for clients interacting with the application.

* **Service Layer** (org.styd.intproj.savorly.service)

Contains **business logic** and orchestrates interactions between controllers and repositories.

* **Repository Layer** (org.styd.intproj.savorly.repository)

Provides **direct access to the database** using JPA and manages data persistence.

* **Configuration Layer** (org.styd.intproj.savorly.config)

Defines **application-wide settings** for **cloud storage, SMTP email services, external APIs (Ollama API), security configurations, and other global settings**.

* **SecuringWeb Layer** (org.styd.intproj.savorly.securingweb)

Implements **authentication and authorization** using **Spring Security and JWT**, ensuring secure access control for API endpoints.

### **4.3 Layers Details**

#### 4.3.1 Controller Layer

The **Controller Layer** contains all the RESTful API endpoints. These endpoints allow users to manage the **entire recipe lifecycle**, from **creation and modification to sharing and discovery**. Each controller is responsible for handling specific entities and user interactions. And it contains a part of business logic.

#### **Key Controller Classes**

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| UserController.java | Provides complete user registration, login and editing functions with verification |
| RecipeController.java | Presents complete recipe creation, modification, deletion, fuzzy query, and vector query functions. |
| FavoriteController.java | Manages the favorite recipes with complete creation, deletion and query functions. |
| ReviewController.java | Manages the review of recipes with complete creation, deletion and query functions. |
| FileUploadController.java | Provides a function for uploading a recipe picture. |
| MailController.java | Provides a function for sharing a recipe by Gmail SMTP serve. |

#### 4.3.2 Service Layer

The **Service Layer** contains the most **business logic** and ensures proper interactions between controllers and repositories. It enforces data validation, applies business rules, and integrates with external services.

#### **Key Service Classes**

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| JwtTokenProvider.java | Be responsible for generating, parsing, and validating JSON Web Tokens (JWTs) to facilitate authentication and authorization processes. |
| RecipeService.java | Manages CRUD operations for recipes, including creating, reading, updating, and deleting recipes, as well as handling associated tags and embeddings, and performing search functionalities. |
| S3Service.java | Generates pre-signed URLs for objects stored in an Amazon S3 bucket, allowing clients to perform time-limited operations (such as uploads or downloads) directly to and from S3 without requiring AWS credentials. |
| OpenAiEmbeddingService.java | Generates text embeddings by interacting with OpenAI's API, for facilitating tasks vector search. |
| EmbeddingService.java | Generates embeddings for provided text inputs by interacting with the OpenAiEmbeddingService, for vector query in vector database |
| TagService.java | Manages operations related to tags, including creating, reading, updating, and deleting tags, as well as performing fuzzy searches and finding nearest tags if wihout embeddings. |
| PdfWithS3Service.java | Generates PDF documents that incorporate images stored in Amazon S3 by utilizing pre-signed URLs to securely access and embed these images into the PDFs. |
| MailService.java | Sends emails, including support for HTML content, by utilizing the JavaMailSender to construct and dispatch MimeMessage instances. |

#### 4.3.3 Repository Layer

The **Repository Layer** serves as an abstraction over the data access layer, providing a convenient interface for CRUD operations and custom queries on the underlying database. By extending interfaces like JpaRepository . Developers can leverage Spring Data JPA to handle data persistence with minimal boilerplate code.

#### **Key Service Classes**

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| UserRepository.java | Extends JpaRepository to provide CRUD operations for User entities. |
| RecipeRepository.java | Extends JpaRepository to provide CRUD operations for Recipe entities including custom query methods, to facilitate complex data retrieval operations. |
| TagRepository.java | Extends JpaRepository to provide CRUD operations for Tag entities and native SQL queries for PostgreSQL with Pgvector like finding nearest tags based on embeddings and performing insert or update operations with embeddings. |
| FavouriteRepository.java | Extends JpaRepository to provide CRUD operations for Favourite entities and defines custom to retrieve a list of Recipe entities favorited by a specific user and finding a Favourite entity based on both recipeId and userId. |
| ReviewRepository.java | Extends JpaRepository to provide CRUD operations for Review entities and defines a custom query method to retrieve all reviews associated with a specific recipe. |

#### 4.3.4 Configuration Layer

The Configuration Layer contains essential application-wide configuration files to set up security, database connections, API documentation, and other global settings.

#### **Key Configuration Classes**

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| SecurityConfig.java | Defines JWT authentication, role-based access control, and session policies. |
| CorsConfig.java | Configures Cross-Origin Resource Sharing (CORS) for frontend integration. |
| DatabaseConfig.java | Manages database connections, transaction management, and JPA settings. |
| LoggingConfig.java | Configures application-wide logging using Logback. |
| PdfConfig.java | Handles PDF generation using iText. |

#### 4.3.5 Securingweb Layer

The **Securingweb Layer** integrates multiple components to provide comprehensive security. We use 4 components work together to safeguard the application against unauthorized access collectively.

#### **Key Configuration Classes**

|  |  |
| --- | --- |
| **Class** | **Purpose** |
| CustomUserDetails.java | Provides user-specific information such as authorities, ID, and user details. |
| CustomUserDetailsService.java | Includes methods like loadUserByUsername to retrieve user data and getCurrentUserId to fetch the current user's ID. |
| JwtAuthenticationFilter.java | Processes JWT tokens through methods like doFilterInternal and getJwtFromRequest. |
| WebSecurityConfig.java | Configures security settings, including defining the authenticationManager, setting up CORS configurations with corsConfigurationSource, and establishing the security filter chain via securityFilterChain. |

## **5. API Design**

### **5.1 User API**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| POST | /api/user/register | User registration | ❌ |
| POST | /api/user/login | User login | ❌ |
| GET | /api/user/ | Get current user info | ✅ |
| POST | /api/user/edit | Edit current user info | ✅ |
|  |  |  |  |

**Example: User Register**

POST /api/user/register  
{  
 "id": ,

"username": "tester1",

"password": "password123",

"password2": "password123"  
}

**Example: User Login**

POST /api/user/login  
{  
 "username": "tester1",

"password": "password123"  
}

**Example: Get Current User Information**

GET /api/user

Headers:

Authorization: Bearer <token>

**Example: User Upation**

POST /api/user/edit  
{  
 "id":,

"username": "tester1",

"password": "password1234",

"password2": "password1234"  
}

### **5.2 Recipe API**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| GET | /api/recipe | Get all recipes | ✅ |
| GET | /api/recipe/{id} | Get a recipe by id | ✅ |
| GET | /api/recipe/mine | Get recipes created by current user | ✅ |
| POST | /api/recipe/create | Create a recipe | ✅ |
| PUT | /api/recipe/edit/{id} | Update a recipe | ✅ |
| DELETE | /api/recipe/delete/{id} | Delete a recipe | ✅ |

**Example: Get all Recipes**

GET /api/recipe

Headers:

Authorization: Bearer <token>

Content-Type: application/json

**Example: Get Recipe by id**

GET /api/recipe/{id}

Headers:

Authorization: Bearer <token>

**Example: Get Recipes created by current user**

GET /api/recipe/mine

Headers:

Authorization: Bearer <token>

**Example: Create Recipe**

POST /api/recipe/create

Headers:

Authorization: Bearer <token>

Body:

```json

{

"id":,

"name": "Spaghetti Carbonara",

"ingredients": "Spaghetti, Eggs, Parmesan Cheese, Pancetta, Pepper",

"instructions": "Boil pasta, cook pancetta, mix eggs and cheese, combine everything",

"picture": "IMG\_202501230001.png",

"authorId": 1L,

}

### **5.3 Favourite API**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| GET | /api/favourites | Get all favourite recipes of current user | ✅ |
| POST | /api/favourites/{recipeId} | Create a favourite recipe by recipe id | ✅ |
| DELETE | /api/favourites/{recipeId} | Delete a favourite recipe by recipe id | ✅ |

**Example: Get Favourite Recipes of current user**

GET /api/favourites

Headers:

Authorization: Bearer <token>

Content-Type: application/json

**Example: Create a Favourite Recipe by Recipe id**

POST /api/favourites/{recipeId}

Headers:

Authorization: Bearer <token>

Body:

```json

{

"recipeId": 1L,

}

### **5.4 Rating API**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| GET | /api/rating/user/{recipeId} | Get a rating of a recipe given by current user by recipe id | ✅ |
| GET | /api/rating/recipe/{recipeId} | Get an average rating of a recipe by recipe id | ✅ |
| POST | /api/rating/{recipeId} | Create a rating by recipe id | ✅ |

**Example: Get a Rating of a Recipe given by current user by Recipe id**

GET /api/rating/user/{recipeId}

Headers:

Authorization: Bearer <token>

Content-Type: application/json

**Example: Create a Rating by Recipe id**

POST /api/rating/{recipeId}

Headers:

Authorization: Bearer <token>

Body:

```json

{

"rating": 4,

}

### **5.5 Review API**

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Endpoint** | **Description** | **Auth Required** |
| GET | /api/reviews/{recipeId} | Get all reviews of a recipe by recipe id | ✅ |
| POST | /api/reviews/{recipeId} | Create a review of a recipe by recipe id | ✅ |
| DELETE | /api/reviews/{recipeid} | Delete a review of a recipe by recipe id | ✅ |

**Example: Get all Reviews of a Recipe by Recipe id**

GET /api/reviews/{recipeId}

Headers:

Authorization: Bearer <token>

Content-Type: application/json

**Example: Create a Review of a Recipe by Recipe id**

POST /api/reviews/{recipeId}

Headers:

Authorization: Bearer <token>

Body:

```json

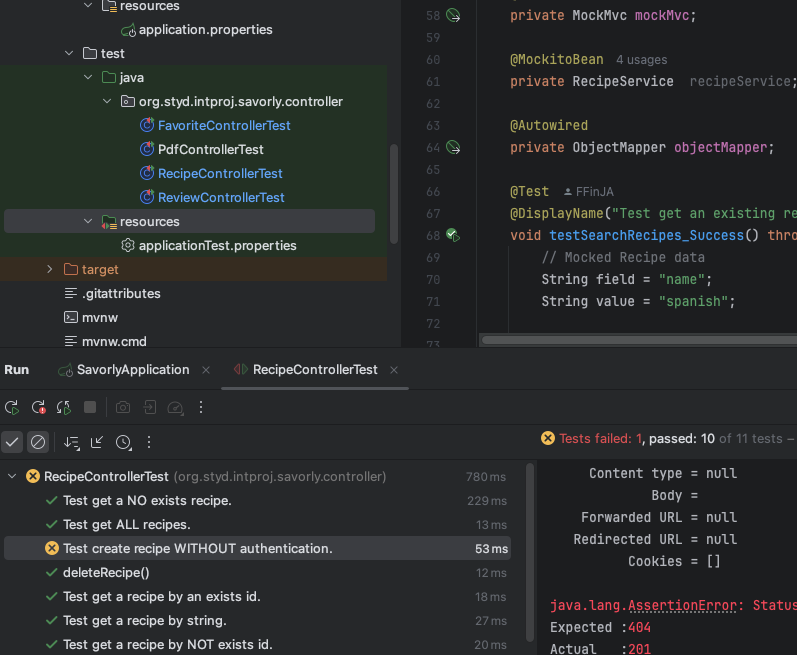
{

"text": "A great Spanish style recipe.",

}

### **5.6 Unit Test classes**

We Used the junit5 for unit test. We brought 4 Test classes to cover about 30% of our endpoints.



## **6. Database Design**

### **6.1 Users Table**

|  |  |  |
| --- | --- | --- |
| **Column** | **Type** | **Description** |
| id | LONG | Primary Key |
| username | VARCHAR(100) | Unique UserName |
| password | VARCHAR(255) | Hashed Password |
| favourites | List<Favourites> | One-to-Many relationship with Favourite |
| ratings | List<Rating> | One-to-Many relationship with Rating |

### **6.2 Recipes Table**

|  |  |  |
| --- | --- | --- |
| **Column** | **Type** | **Description** |
| id | Long | Primary Key |
| name | VARCHAR(255) | Unique Name of Recipe |
| ingredients | TEXT | Ingredients of Recipe |
| instructions | TEXT | Instructions of Recipe |
| picture | VARCHAR(255) | Picture of Recipe |
| authorid | Long | Author of Recipe |
| favourites | List<Favourites> | One-to-Many relationship with Favourite |
| ratings | List<Rating> | One-to-Many relationship with Rating |

### **6.3 Favourites Table**

|  |  |  |
| --- | --- | --- |
| **Column** | **Type** | **Description** |
| id | Long | Primary Key |
| recipeid | Long | Foreign Key referencing Recipe(id) |
| userid | Long | Foreign Key referencing User(id) |

### **6.4 Ratings Table**

|  |  |  |
| --- | --- | --- |
| **Column** | **Type** | **Description** |
| id | Long | Primary Key |
| recipeid | Long | Foreign Key referencing Recipe(id) |
| authorid | Long | Foreign Key referencing User(id) |
| rating | Integer | Rating value between 1 to 5. |

### **6.5 Reviews Table**

|  |  |  |
| --- | --- | --- |
| **Column** | **Type** | **Description** |
| id | Long | Primary Key |
| recipeid | Long | Foreign Key referencing Recipe(id) |
| userid | Long | Foreign Key referencing User(id) |
| text | TEXT | The content of review |
| date | DATE | The date when the review was written, in YYYY-MM-DD format. |

### **6.6 Tags Table**

|  |  |  |
| --- | --- | --- |
| **Column** | **Type** | **Description** |
| id | Long | Primary Key |
| recipeid | Long | Foreign Key referencing Recipe(id) |
| title | Long | Foreign Key referencing User(id) |
| ingredients | TEXT | The content of review |
| description | DATE | The date when the review was written, in YYYY-MM-DD format. |
| embedding | vector | Vector embedding generated by AI model |

## **7. Frontend Structure (React)**

/src

├── assets # Assets  
 ├── components # Components

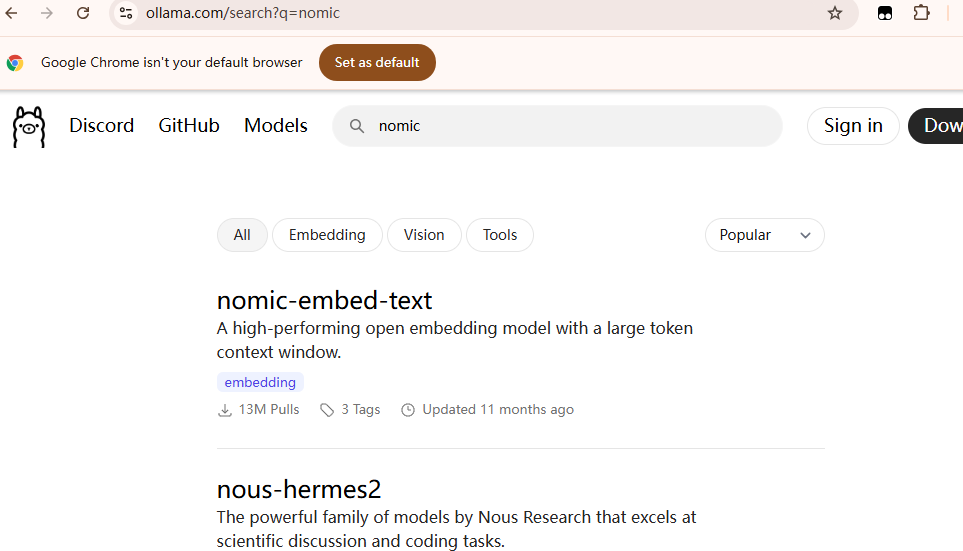
├── helpers # API and FlashMessage settings  
 ├── pages # Pages  
 ├── App.js # Main Entry Point

## **8. Vector Search Implementation**

In the early stage of our program design, we called the nomic-embed-text model hosted on Ollama to generate embedding data. Later, in the production environment, for ease of configuration and testing, we used OpenAi's text-embedding3-small model as the embedding model. We will show you how to install the ollama and embedding models locally.

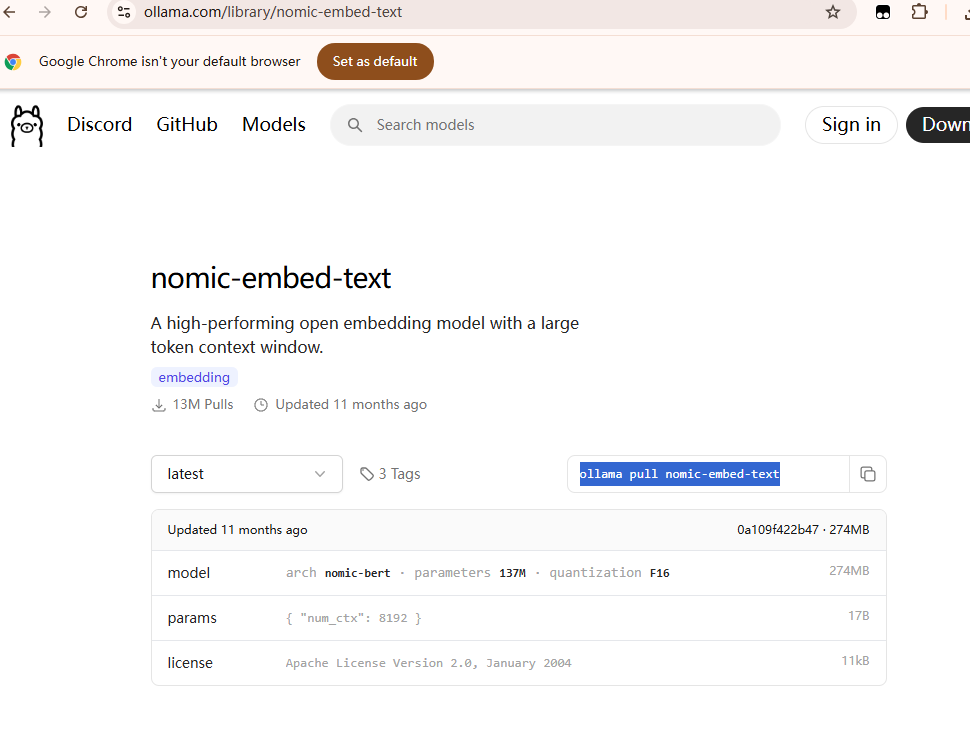
### **8.1 Ollama Installation**

Ollama is a locally-run LLM (Large Language Model) management platform, mainly used to run and manage open source AI models on local machines, focused on local deployment and lightweight operation.



Of course, the best case scenario is that your machine is equipped with an nVidia graphics card with CUDA, and the corresponding drivers and frameworks are installed.

The easiest way for Ollama installation is in Docker, use docker run --rm -it --gpus all -p 11434:11434 ollama/ollama to install Ollama in exists local docker server. Instead of downloading the installer directly from the Ollama official website and installing it.

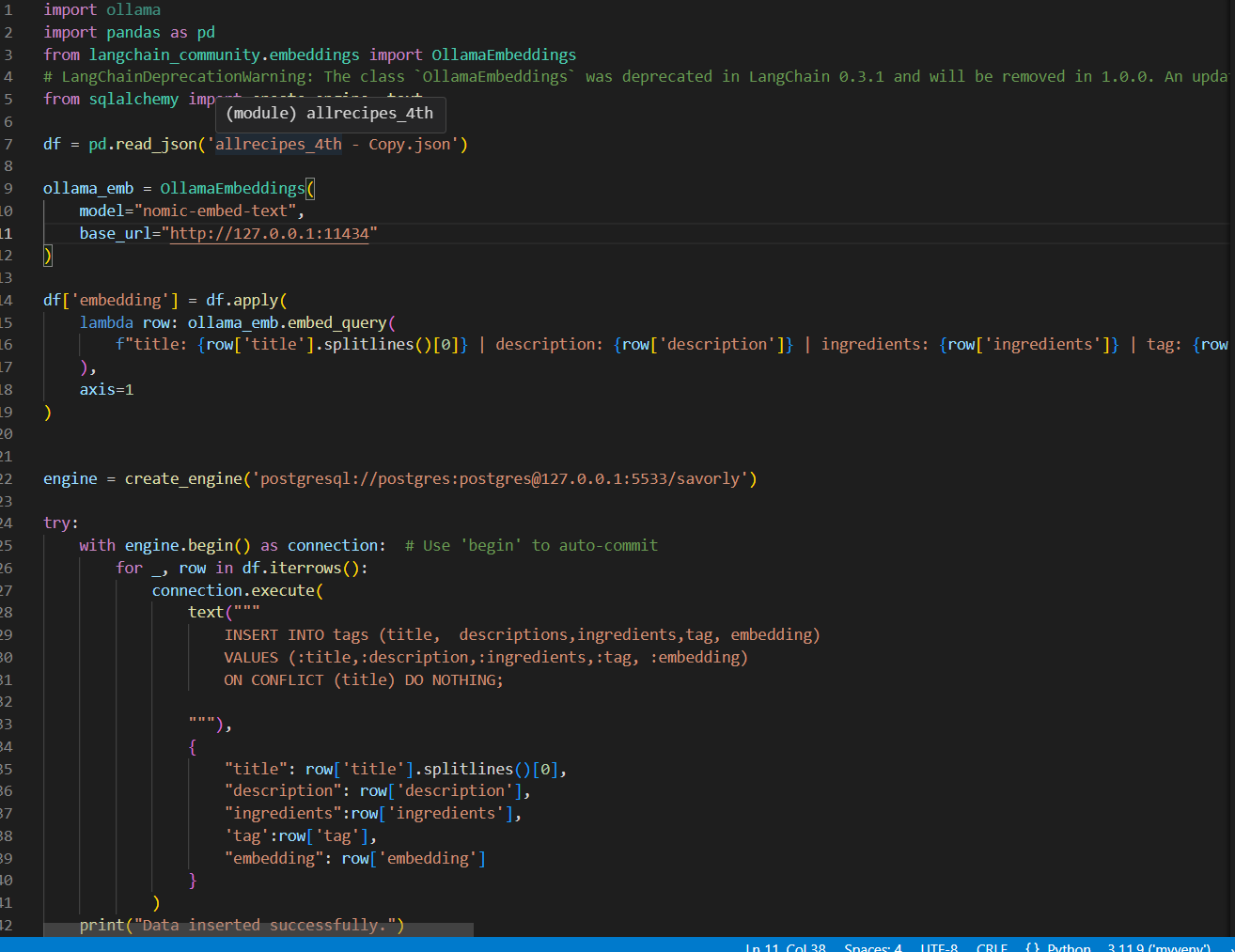


Use ollama pull nomic-embed-text in CLI/Terminal, to install the embedding model. Please notice that this model will return 768 dimensions vectors (while 1536 are returned for text-embedding-3-small, or 3072 for text-embedding-3-large, both by OpenAI).

The default port of Ollama is 11434, and Ollama API for LLMs is on <http://127.0.0.1:11434/v1> for local docker installation, and API for embedding is on <http://127.0.0.1:11434/api/embed>.

### **8.2 Python scripts for Embedding vectors**

This Python script is for demonstration purposes only, because according to our database design now, the Tags table can only be inserted after a record has been inserted into the Recipes table first. But it explains how to get the embedding vector.



## **9. Hosting Solutions**

For hosting our frontend and backend projects, we opted to use a service called “Railway”. Because we had a “monorepo” setup on GitHub where both projects were in respective “frontend” and “backend” folders, we had to ensure the root directory for each project pointed towards the correct folder. Our environment variables (from the .env files) were securely stored alongside each project deployment on the Railway platform.

For our database, we opted to use an Amazon RDS instance that we connect to using JPA and Spring Boot. As for picture uploading, we used an Amazon S3 bucket.

You can access our [hosted application on Railway](https://savorly-frontend-production.up.railway.app/).

## **10. Conclusion**

* **Decoupled frontend & backend**, using REST API
* **JWT Authentication**, ensuring security
* **Backend Deployment**, cloud-ready
* **Frontend Deployment,** available and hosted

This documentation provides a complete project structure for the easy development and maintenance of this project.