

Portfolio Tracker API

R1 Identification of the problem you are trying to solve by building this particular app.

The **Portfolio Tracker API** for managing investment portfolios is designed to tackle common issues faced by investors, including:

Scattered Investments:

Many investors have their assets spread out across different platforms or investment types, making it hard to see the big picture of their financial health.

Time-consuming Portfolio Management:

Keeping track of and updating various investments manually can be a hassle and prone to mistakes.

Need for Up-to-date Information:

In the fast-moving financial world, investors require the latest data on their investments to make swift, informed choices.

Security Worries:

Given the sensitive nature of financial data, investors are often concerned about the safety and privacy of using third-party tracking services.

How the App Helps:

Brings Everything Together:

Offers a unified platform for all investment details, simplifying tracking and updates.

Delivers Real-time Insights:

Provides current financial data to help investors make quick, informed decisions.

Keeps track of all transactions, asset classes and overall portfolio wealth:

Includes features for tracking current transactions and grouping asset classes owned together in one convenient place for quick retrieval. Portfolio equity can be also be accessed easily.

Customizable and Scalable:

Can be tailored to fit individual needs and easily handles growing investment portfolios.

Prioritizes Security:

Uses strong security measures to protect users' financial information, offering peace of mind.

This app aims to make investment management more straightforward, informed, and secure, helping investors better achieve their financial goals.

R2 Why is it a problem that needs solving?

The absence of a centralized and automated investment management system presents several significant challenges for investors, highlighting the necessity for a solution like the described app. Without such a system, investors face the following problems:

Scattered Investment Data:

Investors typically have assets across different accounts, platforms, or asset classes like stocks, bonds, and cryptocurrencies. This dispersion makes it difficult to get a unified view of their financial health, leading to disjointed decision-making and potential oversight of critical portfolio diversification or rebalancing opportunities.

Manual Portfolio Updates and Time Consumption:

Tracking and updating portfolio holdings manually is not only tedious but also prone to errors. Each buy or sell transaction requires manual entry, and keeping tabs on the current market value of diverse assets can be overwhelmingly complex, consuming valuable time that could be better spent on strategic investment analysis.

Security and Privacy Concerns:

Given the sensitive nature of financial data, there's a legitimate concern over the security and privacy of using third-party services. Investors need assurance that their data is protected against unauthorized access and breaches.

In summary:

The absence of an efficient investment management system complicates portfolio tracking, exposes investors to the risk of making decisions based on outdated information, and ultimately hampers their ability to achieve optimal financial outcomes. The app addresses these issues by offering a secure, real-time, and user-friendly platform for investors to manage their portfolios more effectively, making it a critical solution for anyone looking to optimize their investment strategy.

R3 Why have you chosen this database system. What are the drawbacks compared to others?

Choosing PostgreSQL for a project comes down to its many advantages and powerful features. Here's a straightforward overview of why PostgreSQL is a standout option and its minor limitations compared to other database systems:

Why Pick PostgreSQL

- **Free and Open-Source:** PostgreSQL is available at no cost, allowing you to use, modify, and distribute it freely. This aspect significantly reduces database management expenses.
- **Packed with Features:** Capable of handling tricky operations and supporting a broad array of advanced data types, PostgreSQL enables extensive data manipulation, making it suitable for diverse projects.
- **Customizable:** It offers the flexibility to introduce custom data types, craft functions, and even employ various programming languages beyond SQL.
- **Follows Standards:** Adhering closely to SQL standards, PostgreSQL ensures your application can seamlessly interface with other database systems as required.
- **Flexible with Data Types:** Supporting both traditional SQL and NoSQL data formats, such as JSON, PostgreSQL is adaptable for various data storage and retrieval needs.
- **Secure:** Equipped with robust built-in security features, including encryption and access control mechanisms, PostgreSQL safeguards your data effectively.

Limitations

- **Might Be Slower for Some Tasks:** In scenarios requiring extensive data reads, other specialized databases might offer superior performance.
- **Complicated Replication:** Establishing data replication, particularly in tricky configurations, can be more challenging in PostgreSQL than in some alternative databases.
- **Uses More Resources:** PostgreSQL's operation may demand greater memory and processing power compared to more lightweight database options.
- **Handling Large Volumes of Unstructured Data:** When dealing with vast amounts of unstructured data, NoSQL databases could potentially provide better efficiency due to their schema-less nature.

Even with a few downsides, PostgreSQL's wide range of features, strong security, and flexibility make it a great pick for projects that need careful data management and reliable performance. Overall, the benefits of using PostgreSQL far outweigh the few limitations it has, making it ideal for handling complicated data tasks.

R4 Identify and discuss the key functionalities and benefits of an ORM

Object-Relational Mapping (ORM) is a way to use your programming language to interact with a database, so you don't have to write SQL code. It's like having a bridge between the way databases store data and how your application wants to use it. Here's a simpler look at what ORM does and why it's useful:

Key Functionalities

- **Makes Coding Easier:** ORM lets you work with database data as if they were objects in your code. No need to write SQL directly.
- **Handles CRUD Easily:** Creating, reading, updating, and deleting database entries is simpler because you use your programming language's syntax, not SQL.
- **Converts Data Types:** Automatically changes data from one type to another, like turning SQL dates into the date types your programming language understands.
- **Manages Transactions Safely:** Keeps your data safe by automatically managing database transactions and rolling back changes if something goes wrong.
- **Improves Performance with Lazy Loading:** Loads data only when it's needed, which can make your application faster and use less memory.
- **Uses Caching to Speed Things Up:** Some ORM tools can remember data that's been used before, reducing the need to go back to the database.

Benefits

- **Saves Time:** Automates routine tasks, so developers can focus on more important stuff, speeding up how fast you can make things.
- **Makes Code Cleaner:** Encourages writing neater, more maintainable code since you're working with high-level concepts, not database specifics.
- **Works Across Different Databases:** Makes your application flexible. You can change databases with minimal fuss because the ORM deals with database-specific details.
- **Keeps Your Data Safe:** Helps prevent common security issues, like SQL injection, by handling data in a safer way.
- **Deals with Complex Data Easily:** Simplifies working with tricky database relationships, turning them into easy-to-understand object properties.

While ORM has many pluses, like making development faster and your code cleaner, there can be downsides, like potentially slower performance for complex queries. However, for many projects, the benefits are worth it, making ORM a favorite for working with databases in web development.

R5 Document all endpoints for your API

User Management (auth_controller)

prefix for all User management API endpoints (auth_controller):

```
http://localhost:5000/auth
```

1. User Registration:

- **Endpoint:** 'POST' /auth/register
- **Description:** Allows new users to register. Accessible by anyone who isn't logged in.
- **Request Methods:** 'POST'
- **Request Body Example:**

```
{  
  "email": "test_account@email.com",  
  "password": "123456"  
}
```

- **Success Response Example:**

```
{  
  "email": "test_account@email.com",  
  "password": "$2b$12$X1H0xFaiMxfPxTWbqv4zQ0zykzuLVCA1hZZhSRFRHPyCW0av4rGXq",  
  "userID": 5  
}
```

- **Error Response Example:**

```
{  
  "error": "Email address already in use"  
}
```

- **Status Codes:**
 - '201 Created' if the registration is successful.
 - '400 Bad Request' if the input validation fails (e.g., missing email or password, email already in use).

2. User Login:

- **Endpoint:** 'POST' /auth/login
- **Description:** Authenticates users and returns a token (JWT). Accessible by anyone who isn't logged in.
- **Request Methods:** 'POST'
- **Request Body Example:**

```
{
  "email": "test_account@email.com",
  "password": "123456"
}
```

- **Success Response Example:**

```
{
  "email": "test_account@email.com",
  "is_admin": false,
  "token":
    "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9.eyJmcmVzaCI6ZmFsc2UsIm1hdCI6MTcxMDI5NDY3MCwianRpIjojOWVjM2RmNzctZDhmMy00YTM2LTk0OWMtMWUyMWY4YWVlYTU2IiwidHlwZSI6ImFjY2VzcyIsInN1YiI6IjUiLCJuYmYiOiJlZ3MTAyOTQ2NzAsImNzcmYiOiI0NzA0NzJjOC1kMGJkLTQ3YmQtODA0Mi05NTAzY2F1YzA2ODkiLCJleHAiOiJlZ3MTAzODEwNzB9.ywr-dRkbhu2vW3K0QnNjj92n1963yiaExvqP1BVS-Q"
}
```

- **Error Response Example:**

```
{
  "error": "Invalid email or password"
}
```

- **Status Codes:**

- **'200 OK'** if the login is successful.
- **'401 Unauthorised'** if the email or password is incorrect.

3. Delete User

- **Endpoint:** 'DELETE' /auth/<int:user_id>
- **Description:** This endpoint allows for the deletion of a specific user by their user ID. It can be accessed by the user themselves (if the user_id in the URL matches the currently logged-in userID) or by an authenticated user with administrative privileges. This setup enables users to delete their own account or allows administrators to remove user accounts and associated data from the system.
- **Request Methods:** 'DELETE'
- **URL Parameters:** 'user_id' - The unique identifier (integer) of the user to be deleted.
- **Requires Headers:**
 - **'Authorization': 'Bearer JWT'** - The JWT token obtained upon authenticating. The token should either belong to the user being deleted or to an administrative user.
- **Success Response Example:**
 - If the user is successfully deleted:

```
{
  "message": "User with userID '<user_id>' was successfully deleted."
}
```

- **Error Response Examples:**

- If the user is not authenticated, the token is invalid, or the user_id does not match, and the user is not an admin:

```
{
  "error": "Unauthorized. You do not have permission to perform this action."
}
```

- If the user to be deleted does not exist:

```
{
  "error": "User with userID '<user_id>' does not exist."
}
```

- **Status Codes:**

- **'200 Ok'** If the user is successfully deleted.
- **'401 Unauthorised'** If the user is not authenticated, the token is invalid, or the user_id does not match, and the user is not an admin.
- **'404 Not Found'** If the user is not found.

Portfolio Management (portfolio_controller)

prefix for all portfolio management API endpoints (portfolio_controller):

```
http://localhost:5000/portfolios
```

1. Create Portfolio

- **Endpoint:** 'POST' /portfolios/create
- **Description:** This endpoint enables authenticated users to create a new portfolio. It requires users to be logged in and provide necessary portfolio details through the request body. The newly created portfolio is associated with the user's account based on their authentication token.
- **Request Methods:** 'POST'

- **Request Body Example:**

```
{
  "name": "My Investment Portfolio",
  "description": "A diversified portfolio of cryptocurrencies"
}
```

- **Requires Headers:**

- **'Authorization': 'Bearer JWT'** - The JWT token obtained upon logging in. This token identifies the authenticated user and links the newly created portfolio to their account.

- **Success Response Example:**

- If the portfolio is successfully created:

```
{
  "message": "Portfolio 'My Investment Portfolio' was successfully
created.",
  "portfolioID": 1,
  "userID": "<user_id>"
}
```

- **Error Response Examples:**

- If the user is not authenticated or the token is invalid:

```
{
  "error": "Unauthorized. You must be logged in to perform this
action."
}
```

- If required data is missing or invalid in the request body:

```
{
  "error": "Bad Request. Missing or invalid data provided."
}
```

- **Status Codes:**

- **'201 Created'** If the portfolio is successfully created.
- **'400 Bad Request'** If there is missing or invalid data in the request body.
- **'401 Unauthorized'** If the user is not authenticated or the token is invalid.

2. Retrieve All Portfolios

- **Endpoint:** 'GET' /portfolios

- **Description:** This endpoint retrieves all portfolios from the database. It is accessible only by users with administrative privileges, allowing for a comprehensive view of all user portfolios.
- **Request Methods:** 'GET'
- **Requires Headers:**
 - **'Authorization': 'Bearer JWT'** - The JWT token obtained upon logging in. This token should belong to an administrative user.
- **Success Response Example:**
 - If the request is successful:

```
[
  {
    "portfolioID": 1,
    "name": "Investment Portfolio A",
    "userID": "2"
  },
  {
    "portfolioID": 2,
    "name": "Investment Portfolio B",
    "userID": "3"
  }
]
```

- **Error Response Examples:**
 - If the user is not an admin:

```
{
  "error": "Unauthorized. Administrative access required."
}
```

- **Status Codes:**
 - **'200 OK'** If the request is successful.
 - **'401 Unauthorized'** If the user is not authenticated as an admin.

3. Search Portfolio

- **Endpoint:** 'GET' /portfolios/<portfolio_id>
- **Description:** Displays details of a specific portfolio. This endpoint is accessible by the owner of the portfolio or an administrator, ensuring privacy and administrative oversight.
- **Request Methods:** 'GET'
- **URL Parameters:** 'id' - The unique identifier of the portfolio to be retrieved.
- **Requires Headers:**
 - **'Authorization': 'Bearer JWT'** - The JWT token obtained upon logging in. The token should either belong to the portfolio's owner or to an administrative user.
- **Success Response Example:**
 - If the portfolio is found:

```
{
  "portfolioID": 1,
  "name": "My Investment Portfolio",
  "userID": "<user_id>"
}
```

- **Error Response Examples:**

- If the portfolio does not exist or the user does not have permission to view it:

```
{
  "error": "Unauthorized or Portfolio not found."
}
```

- **Status Codes:**

- **'200 OK'** If the portfolio is successfully retrieved.
- **'401 Unauthorized'** If the user is not authenticated or lacks access.
- **'404 Not Found'** If the portfolio does not exist.

4. Update Portfolio

- **Endpoint:** 'PATCH' /portfolios/<portfolio_id>
- **Description:** This endpoint allows for the editing of details of a specific portfolio. It ensures that portfolio management is personal and secure by granting access exclusively to the portfolio's owner or to an administrative user. The flexibility to update portfolio details such as its name, description, or other attributes enables users to maintain accurate and up-to-date information.
- **Request Methods:** 'PATCH'
- **URL Parameters:** 'id' - The unique identifier (integer) of the portfolio to be updated.
- **Request Body Example:**

```
{
  "name": "Updated Portfolio Name",
  "description": "New description of the portfolio",
  "holdings": 150000 // Example of an attribute that might be updated
}
```

- **Requires Headers:**

- **'Authorization': 'Bearer JWT'** - The JWT token must be provided in the request header to authenticate. This token verifies the identity of the requester, ensuring that only the portfolio's owner or an administrative user can perform updates.

- **Success Response Example:**

- If the portfolio is successfully updated:

```
{
  "message": "Portfolio with ID '<id>' was successfully updated."
}
```

```
}
```

- **Error Response Examples:**

- If the user is not authenticated, the token is invalid, or the user does not have the necessary permissions:

```
{
  "error": "Unauthorized. You do not have permission to perform this action."
}
```

- If the portfolio to be updated does not exist:

```
{
  "error": "Portfolio with ID '<id>' does not exist."
}
```

- **Status Codes:**

- **'200 OK'** If the portfolio is successfully updated.
- **'400 Bad Request'** If the request body is missing required attributes or contains invalid data.
- **'401 Unauthorized'** If the user is not authenticated or lacks the necessary permissions to update the portfolio.
- **'404 Not Found'** If the specified portfolio does not exist.

5. Delete Portfolio

- **Endpoint:** 'DELETE' `/portfolios/<portfolio_id>`
- **Description:** This endpoint facilitates the removal of a specific portfolio from the system. It is designed to ensure secure portfolio management by allowing only the portfolio's owner or an administrative user to delete a portfolio. This operation permanently deletes the portfolio and its associated data, underscoring the importance of cautious use.
- **Request Methods:** 'DELETE'
- **URL Parameters:** 'id' - The unique identifier (integer) of the portfolio to be deleted.
- **Requires Headers:**
 - **'Authorization': 'Bearer JWT'** - To authenticate the request, a JWT token must be included in the Authorization header. This token serves to identify the requester, confirming whether they are the owner of the portfolio in question or an administrative user with the authority to delete it.
- **Success Response Example:**
 - If the portfolio is successfully deleted:

```
{
  "message": "Portfolio with ID '<id>' was successfully deleted."
}
```

- **Error Response Examples:**

- If the user is not authenticated, the token is invalid, or the user does not have the necessary permissions:

```
{
  "error": "Unauthorized. You do not have permission to perform this action."
}
```

- If the specified portfolio does not exist:

```
{
  "error": "Portfolio with ID '<id>' does not exist."
}
```

- **Status Codes:**

- **'200 OK'** If the portfolio is successfully deleted.
- **'401 Unauthorized'** If the user is not authenticated or lacks the necessary permissions to delete the portfolio.
- **'404 Not Found'** If the specified portfolio does not exist.

Transactions Management (transactions_controller)

1. Create Transaction

- **Endpoint:** 'POST' /transactions/trade

- **Description:** This endpoint allows logged-in users to add a new transaction to their account. It automatically associates the created transaction with the userID of the currently logged-in user, enabling users to track their trades or investments in a personal and secure manner. This functionality is essential for maintaining an accurate and up-to-date record of the user's financial activities.

- **Request Methods:** 'POST'

- **Request Body Example:**

```
{
  "transactionType": "buy",
  "quantity": "25",
  "assetID": "bitcoin"
}
```

- **Requires Headers:**

- **'Authorization': 'Bearer JWT'** - The request must include a JWT token in the Authorization header to authenticate the user. This token identifies the currently logged-in user, ensuring that

the new transaction is correctly associated with their account.

- **Success Response Example:**

- If the transaction is successfully created:

```
{
  "assetID": "bitcoin",
  "date": "2024-03-13",
  "portfolioID": 1,
  "price": 72025.0,
  "quantity": 25,
  "totalCost": 1800625.0,
  "transactionID": 17,
  "transactionType": "buy"
}
```

- **Error Response Examples:**

- If the user is not authenticated or the token is invalid:

```
{
  "error": "Unauthorized. You must be logged in to perform this action."
}
```

- If required data is missing or invalid in the request body:

```
{
  "error": "Bad Request. Missing or invalid data provided."
}
```

- **Status Codes:**

- **'201 Created'** If the transaction is successfully created.
- **'400 Bad Request'** If there is missing or invalid data in the request body.
- **'401 Unauthorized'** If the user is not authenticated or the token is invalid.

2. List Transactions/Assets

- **Endpoint:** 'GET' `/portfolios/<portfolio_id>/transactions`
- **Description:** This endpoint retrieves a list of transactions or assets associated with a specific portfolio. It is designed to be accessible exclusively by the owner of the portfolio, ensuring privacy and security in financial tracking. This functionality enables users to view their investment history and asset allocation within a particular portfolio.

- **Request Methods:** 'GET'
- **URL Parameters:** 'portfolio_id' - The unique identifier (integer) of the portfolio whose transactions or assets are to be retrieved.
- **Requires Headers:**
 - **'Authorization': 'Bearer JWT'** - To access this endpoint, a JWT token must be provided in the Authorization header. This token verifies the identity of the requester, confirming that they are the owner of the specified portfolio.
- **Success Response Example:**
 - If the request is successful, a list of transactions or assets is returned:

```
[
  {
    "assetID": "bitcoin",
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 72093.0,
    "quantity": 10,
    "totalCost": 720930.0,
    "transactionID": 1,
    "transactionType": "buy"
  },
  {
    "assetID": "ethereum",
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 4012.25,
    "quantity": 5,
    "totalCost": 20061.25,
    "transactionID": 2,
    "transactionType": "buy"
  },
  {
    "assetID": "litecoin",
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 98.01,
    "quantity": 30,
    "totalCost": 2940.3,
    "transactionID": 3,
    "transactionType": "buy"
  },
  {
    "assetID": "dogecoin",
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 0.171199,
    "quantity": 1000,
    "totalCost": 171.2,
  }
]
```

```
    "transactionID": 4,  
    "transactionType": "buy"  
  },  
  {  
    "assetID": "bitcoin",  
    "date": "2024-03-13",  
    "portfolioID": 1,  
    "price": 72025.0,  
    "quantity": 25,  
    "totalCost": 1800625.0,  
    "transactionID": 17,  
    "transactionType": "buy"  
  }  
]
```

- **Error Response Examples:**

- If the user is not authenticated or does not own the specified portfolio:

```
{  
  "error": "Unauthorized. Access is restricted to the portfolio's  
owner."  
}
```

- If the specified portfolio does not exist:

```
{  
  "error": "Not Found. Portfolio with ID '<portfolio_id>' does not  
exist."  
}
```

- **Status Codes:**

- **'200 OK'** If the transactions or assets are successfully retrieved.
- **'401 Unauthorized'** If the user is not authenticated or does not own the specified portfolio.
- **'404 Not Found'** If the specified portfolio does not exist.

1. Search for Transaction

- **Endpoint:** 'GET' /transactions/search/<transaction_id>
- **Description:** This endpoint is designed to retrieve a specific transaction. It ensures secure access by allowing only the portfolio's owner or an administrative user to retrieve transaction details. This capability is crucial for reviewing individual transactions, auditing, and comprehensive portfolio management.
- **Request Methods:** 'GET'
- **URL Parameters:** 'transaction_id' - The unique identifier (integer) of the transaction which is to be searched for.

- **Requires Headers:**

- **'Authorization': 'Bearer JWT'** - A JWT token must be included in the Authorization header to authenticate the request. This token identifies the requester, ensuring that only authorized users, such as the portfolio's owner or an admin, can access transaction details.

- **Success Response Example:**

- If the transaction is found:

```
{
  "transactionID": "<transaction_id>",
  "portfolioID": "<portfolio_id>",
  "assetID": "bitcoin",
  "transactionType": "buy",
  "quantity": 1,
  "price": 50000,
  "date": "2024-01-15"
}
```

- **Error Response Examples:**

- If the user is not authenticated, does not own the specified transaction, or is not an admin:

```
{
  "error": "Unauthorized. Access is restricted to the portfolio's owner or an admin."
}
```

- If the specified transaction does not exist:

```
{
  "error": "Not Found. The specified transaction does not exist."
}
```

- **Status Codes:**

- **'200 OK'** If the specific transaction is successfully retrieved.
- **'401 Unauthorized'** If the user is not authenticated or does not have the necessary permissions to access the transaction.
- **'404 Not Found'** If the specified transaction does not exist.

Owned Assets management (ownedAssets_controller)

1. Retrieve all owned Assets

- **Endpoint:** 'GET' /assets/owned

- **Description:** This endpoint is designed to retrieve a comprehensive list of assets owned by the currently authenticated user. For users with administrative privileges, it provides a broader access scope, allowing them to retrieve details of all assets across all users. This functionality is essential for enabling users to overview their investment portfolio and for admins to manage and audit assets across the platform.
- **Request Methods:** 'GET'
- **Requires Headers:**
 - **'Authorization': 'Bearer JWT'** - A JWT token must be provided in the Authorization header for authenticating the user. This token determines whether the requester is a regular user, retrieving their own assets, or an admin, accessing a comprehensive list of all assets.
- **Success Response Example:**
 - If the request is successful, for a regular user, it returns a list of assets owned by them:

```
[
  {
    "ID": 1,
    "assetID": "bitcoin",
    "name": "Bitcoin",
    "portfolioID": 1,
    "price": 74150.85714285714,
    "quantity": 35,
    "symbol": "BTC"
  },
  {
    "ID": 2,
    "assetID": "ethereum",
    "name": "Ethereum",
    "portfolioID": 1,
    "price": 3200,
    "quantity": 5,
    "symbol": "ETH"
  }
]
```

- For an admin, it might return all assets across all users:

```
[
  {
    "assetID": "bitcoin",
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 72093.0,
    "quantity": 10,
    "totalCost": 720930.0,
    "transactionID": 1,
    "transactionType": "buy"
  },
  {
    "assetID": "ethereum",
```

```
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 4012.25,
    "quantity": 5,
    "totalCost": 20061.25,
    "transactionID": 2,
    "transactionType": "buy"
  },
  {
    "assetID": "litecoin",
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 98.01,
    "quantity": 30,
    "totalCost": 2940.3,
    "transactionID": 3,
    "transactionType": "buy"
  },
  {
    "assetID": "dogecoin",
    "date": "2024-03-12",
    "portfolioID": 1,
    "price": 0.171199,
    "quantity": 1000,
    "totalCost": 171.2,
    "transactionID": 4,
    "transactionType": "buy"
  },
  {
    "assetID": "bitcoin",
    "date": "2024-03-13",
    "portfolioID": 1,
    "price": 72025.0,
    "quantity": 25,
    "totalCost": 1800625.0,
    "transactionID": 17,
    "transactionType": "buy"
  },
  {
    "assetID": "bitcoin",
    "date": "2024-03-13",
    "portfolioID": 5,
    "price": 72052.0,
    "quantity": 25,
    "totalCost": 1801300.0,
    "transactionID": 20,
    "transactionType": "buy"
  },
  {
    "assetID": "ethereum",
    "date": "2024-03-13",
    "portfolioID": 5,
    "price": 4043.55,
    "quantity": 5,
```

```
    "totalCost": 20217.75,  
    "transactionID": 21,  
    "transactionType": "buy"  
  },  
  {  
    "assetID": "litecoin",  
    "date": "2024-03-13",  
    "portfolioID": 5,  
    "price": 98.2,  
    "quantity": 100,  
    "totalCost": 9820.0,  
    "transactionID": 22,  
    "transactionType": "buy"  
  }  
]
```

- **Error Response Examples:**

- If the user is not authenticated or the token is invalid:

```
{  
  "error": "Unauthorized. Authentication required to access this  
resource."  
}
```

- **Status Codes:**

- **'200 OK'** If the assets are successfully retrieved.
- **'401 Unauthorized'** If the user is not authenticated or the token is invalid.

Assets management (assets_controller)

1. Retrieve all tradeable assets

- **Endpoint:** 'GET' /assets
- **Description:** This endpoint interfaces with the CoinGecko 3rd party API to retrieve a comprehensive list of cryptocurrency assets available on CoinGecko. It is designed to provide users and administrators with up-to-date information about various cryptocurrencies, including names, symbols, current prices, and market data. This functionality is vital for users looking to make informed investment decisions and for admins to monitor cryptocurrency market trends.
- **Request Methods:** 'GET'
- **No Authentication Required:** This endpoint is designed to be publicly accessible since it retrieves general market data without exposing sensitive user information.
- **Success Response Example:**
 - If the request is successful, it returns a list of cryptocurrency assets:

```
[
  {
    "assetID": "bitcoin",
    "marketCapPos": 1,
    "name": "Bitcoin",
    "price": 72029.0,
    "symbol": "BTC"
  },
  {
    "assetID": "ethereum",
    "marketCapPos": 2,
    "name": "Ethereum",
    "price": 4041.47,
    "symbol": "ETH"
  },
  {
    "assetID": "tether",
    "marketCapPos": 3,
    "name": "Tether",
    "price": 0.999579,
    "symbol": "USDT"
  },
  {
    "assetID": "binancecoin",
    "marketCapPos": 4,
    "name": "BNB",
    "price": 538.45,
    "symbol": "BNB"
  },
  {
    "assetID": "solana",
    "marketCapPos": 5,
    "name": "Solana",
    "price": 149.28,
    "symbol": "SOL"
  },
  {
    "assetID": "staked-ether",
    "marketCapPos": 6,
    "name": "Lido Staked Ether",
    "price": 4038.12,
    "symbol": "STETH"
  },
  {
    "assetID": "ripple",
    "marketCapPos": 7,
    "name": "XRP",
    "price": 0.696329,
    "symbol": "XRP"
  },
  {
    "assetID": "usd-coin",
    "marketCapPos": 8,
```

```
    "name": "USDC",
    "price": 0.999925,
    "symbol": "USDC"
  },
  {
    "assetID": "cardano",
    "marketCapPos": 9,
    "name": "Cardano",
    "price": 0.751814,
    "symbol": "ADA"
  },
  {
    "assetID": "dogecoin",
    "marketCapPos": 10,
    "name": "Dogecoin",
    "price": 0.171544,
    "symbol": "DOGE"
  }
]
```

- **Error Response Examples:**

- If there is an issue connecting to the CoinGecko API or retrieving data:

```
{
  "error": "Service Unavailable. Unable to retrieve data from CoinGecko at this time."
}
```

- **Status Codes:**

- **'200 OK'** If the cryptocurrency asset data is successfully retrieved.
- **'503 Service Unavailable'** If there is an issue connecting to the CoinGecko API or retrieving data.

2. Retrieve Information of a Single Asset

- **Endpoint:** 'GET' `/assets/search/<asset_id>`
- **Description:** This endpoint is specifically designed to retrieve updated general market data about a single cryptocurrency asset.
- **Request Methods:** 'GET'
- **No Authentication Required:** This endpoint is publicly accessible, considering it provides general market data that does not involve sensitive personal information.
- **Success Response Example:**
 - If the request is successful, it returns general information about the asset:

```
{
  "assetID": "litecoin",
```

```
"marketCapPos": 22,  
"name": "Litecoin",  
"price": 98.05,  
"symbol": "LTC"  
}
```

- **Error Response Examples:**

- If the specified asset is not found or there is an issue with the request:

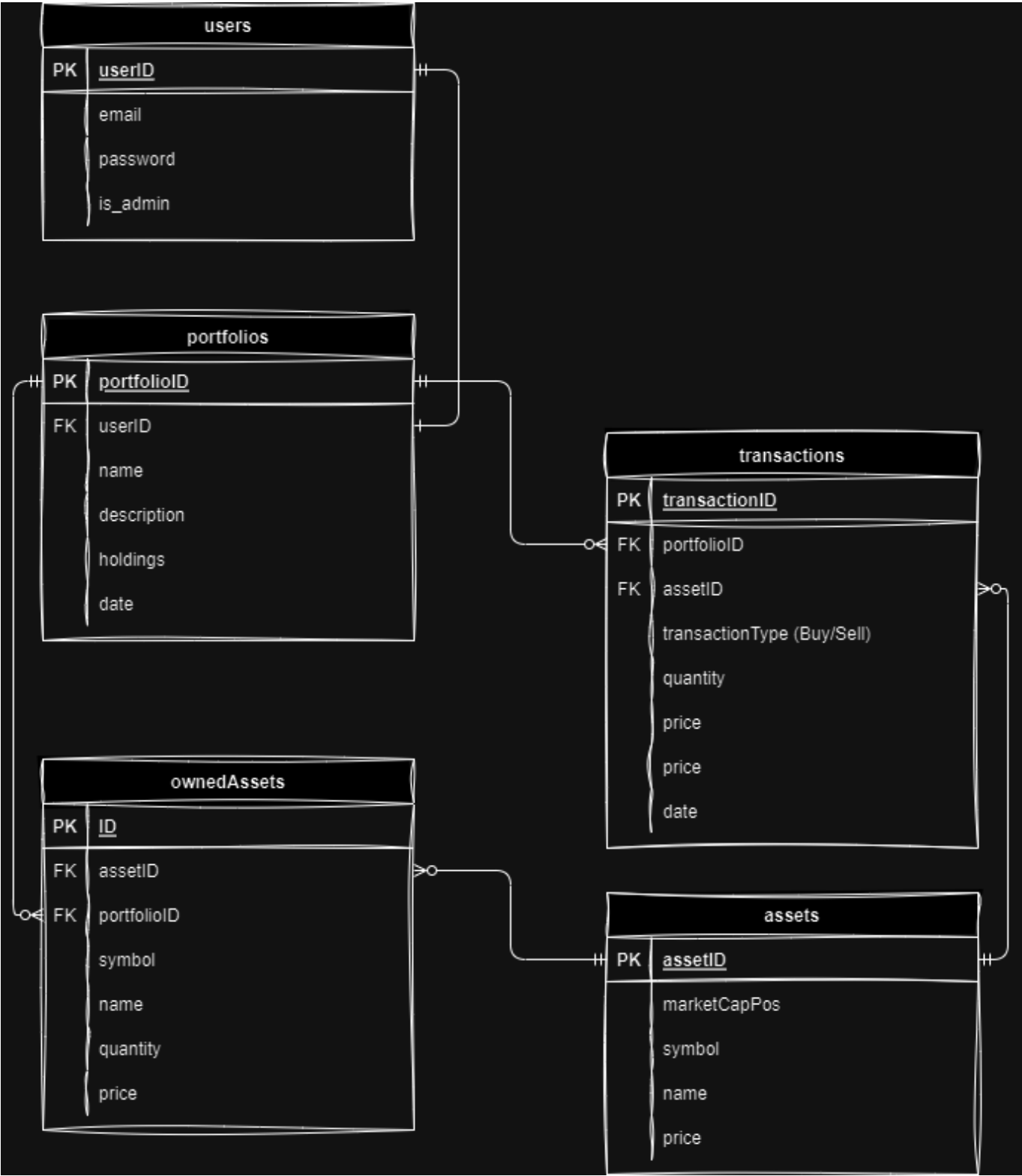
```
{  
  "error": "Not Found. The specified asset could not be found."  
}
```

- **Status Codes:**

- **'200 OK'** If the asset information is successfully retrieved.
- **'404 Not Found'** If the specified asset could not be found or does not exist.

R6 An ERD for your app

The described Entity-Relationship Diagram (ERD) outlines a database schema for managing investment portfolios, describing the relationships between users, their investment portfolio, the assets within those portfolios, and the transactions associated to those assets. Here's a overview:



R7 Detail any third party services that your app will use

PyCoinGecko

Package: pycoingecko==3.1.0

Service: CoinGecko

Description: CoinGecko gives you data about cryptocurrency prices, how much they're being traded, and their market value. The pycoingecko package lets your app get this data straight from CoinGecko.

Use Cases: the portfolio tracker app uses it to show live cryptocurrency prices, how they rank in the market, and their symbols. It also helps retrieve a list of the top 250 cryptocurrencies to update the assets table.

Flask & Flask-Extensions

Package: Flask==3.0.2, Flask-Bcrypt==1.0.1, Flask-JWT-Extended==4.6.0, Flask-SQLAlchemy==3.1.1, flask-marshmallow==1.2.0

Service: Flask

Description: Flask is an easy-to-use web framework for Python. With Flask-Extensions, you can do things like hash passwords (Flask-Bcrypt), manage login tokens (Flask-JWT-Extended), work with databases more easily (Flask-SQLAlchemy), and convert data between formats (flask-marshmallow).

Use Cases: These tools help build and manage portfolio Tracker to handle logins, interact with the database, and work with data easily.

SQLAlchemy & psycopg2-binary

Package: SQLAlchemy==2.0.27, psycopg2-binary==2.9.9

Service: PostgreSQL

Description: SQLAlchemy is a toolkit that makes it easier to talk to databases using Python, and psycopg2-binary lets you connect specifically to PostgreSQL databases.

Use Cases: Together, they help to create database models, run queries, and connect to the PostgreSQL database reliably.

bcrypt & PyJWT

Package: bcrypt==4.1.2, PyJWT==2.8.0

Service: Security and Authentication

Description: bcrypt helps securely hash passwords, so they're safely stored. PyJWT lets you create, check, and use JWT tokens for logging in and keeping information secure.

Use Cases: bcrypt is used to make sure user passwords are safe. PyJWT helps to manage user logins and keep things secure with tokens.

marshmallow

Package: marshmallow==3.21.0

Service: Data Serialization/Deserialization

Description: marshmallow is a tool that helps you convert complex data like objects into simpler Python data types and back.

Use Cases: It's used to check data and convert it to and from JSON format, making it easier to handle API data in the portfolio tracker app.

requests

Package: requests==2.31.0

Service: HTTP Requests

Description: The requests package makes it straightforward to send HTTP requests to websites or APIs.

Use Cases: It is used to talk to web services or APIs, apart from CoinGecko, or for communication within the app if needed.

R8 Describe your projects models in terms of the relationships they have with each other

The project's models represent a financial portfolio tracking system with entities such as users, portfolios, transactions, owned assets, and assets. The relationships between these entities are crucial for organizing how data is linked and managed within the system. Here's an overview of the models and their relationships:

User and Portfolio

- **One-to-One Relationship:** Each **User** is linked to a single **Portfolio**, and vice versa. This connection is made possible by the **portfolio** attribute in the **User** model and the **user** attribute in the **Portfolio** model. A **ForeignKey** on **userID** within the **Portfolio** model ensures this relationship's uniqueness.
- The **cascade="all, delete"** setting on the **User** model's **portfolio** relationship specifies that deleting a **User** will also remove their associated **Portfolio**.

Portfolio and Transaction

- **One-to-Many Relationship:** A **Portfolio** can contain many **Transactions**, yet each **Transaction** is associated with only one **Portfolio**. This relationship is represented by the **transaction** attribute in the **Portfolio** model and the **portfolio** attribute in the **Transaction** model, linked by a **ForeignKey** on **portfolioID** in the **Transaction** model.
- The **cascade="all, delete"** option means that removing a **Portfolio** also deletes all related **Transactions**.

Portfolio and OwnedAsset

- **One-to-Many Relationship:** A **Portfolio** may also contain multiple **OwnedAssets**, but each **OwnedAsset** relates to a single **Portfolio**. This is managed via the **ownedAssets** relationship in the **Portfolio** model and the **portfolio** relationship in the **OwnedAsset** model, connected by a **ForeignKey** on **portfolioID** in the **OwnedAsset** model.
- The **cascade="all, delete"** setting ensures the deletion of a **Portfolio** results in the removal of all its **OwnedAssets**.

Asset, Transaction, and OwnedAsset

- **One-to-Many Relationship:** An **Asset** is linked to multiple **Transactions** and **OwnedAssets**, though each **Transaction** and **OwnedAsset** corresponds to only one **Asset**. These connections are denoted by the **transaction** and **ownedAssets** attributes in the **Asset** model, and the **asset** attributes in both the **Transaction** and **OwnedAsset** models. **ForeignKey** constraints on **assetID** in the **Transaction** and **OwnedAsset** models facilitate these links.
- The **cascade="all, delete"** directives for these relationships indicate that erasing an **Asset** also eliminates all associated **Transactions** and **OwnedAssets**.

The relational structure of these models offers a systematic way to manage portfolios, where users own portfolios that include transactions and assets, and assets may be part of transactions or categorized as owned assets. The inclusion of cascading delete options aids in maintaining data integrity and streamlines the management of interconnected data.

R9 Discuss the database relations to be implemented in your application

In the application, the database relationships are created to support a financial portfolio tracking system. These relationships are key to organizing how data is structured, accessed, and managed, enabling efficient data retrieval and manipulation. Below is a summary of the key relationships and their roles:

User to Portfolio Relationship

- **One-to-One:** This critical relationship ensures each user owns a unique portfolio, streamlining user management and access to financial data. A **ForeignKey** constraint on **userID** within the **Portfolio** model guarantees this relationship's exclusivity.

Portfolio to Transaction Relationship

- **One-to-Many:** Portfolios can contain many transactions. This setup allows for monitoring of buying and selling, offering a view of a user's financial activities.

Portfolio to OwnedAsset Relationship

- **One-to-Many:** This relationship lets a portfolio have many assets, showing a user's diverse investments. It's key for keeping track of how many of each asset a user has and their current value, which is important for understanding how well the portfolio is doing and how investments are spread out.

Asset to Transaction and OwnedAsset Relationships

- **One-to-Many:** Assets play a crucial role in transactions and owned assets, with each asset possibly engaged in more than one transactions and held in multiple portfolios. These connections allow for monitoring an asset's performance over time.

The implementation of **cascade delete** options across these relationships is critical for upholding data integrity. When a user is removed, their associated portfolio and all related data (transactions, owned assets) are likewise deleted. Similarly, eliminating a portfolio results in the deletion of its transactions and owned assets, and removing an asset eradicates all linked transactions and owned asset records. This ensures the database remains current and free from orphaned entries.

By establishing these relationships, the application handles and displays financial data, equipping users with valuable, actionable insights into their investments. The thoughtful organization of these relationships fosters strong data interaction patterns, crucial for a detailed portfolio tracking system.

R10 Describe the way tasks are allocated and tracked in your project

In the **Portfolio tracker API** project, task allocation and tracking are streamlined through the use of Trello, a project management tool that facilitates clear visualization of the project timeline and responsibilities. Here's how it's structured:

Trello Board Setup:

I have set up a Trello board specifically for this project, divided into three lists that correspond to the three weeks of the project timeline. Each list represents a week, ensuring tasks are organized according to the timeframe in which they need to be completed.

Weekly Lists and Daily Cards:

Within each weekly list, I have created cards that represent specific tasks or activities to be undertaken on particular days. These cards detail what needs to be done, and any relevant deadlines. This structure allows for a day-by-day breakdown of the project's progress.

Tracking Progress:

Task's are sorted directly on Trello by using checklist's on cards with status's from "To Do" to "In Progress" and finally to "Done" as I complete their activities. This visual progression allows me to monitor the project's advancement in real-time and identify any areas that may require additional focus.

Adapting to Changes:

Given the tight timeline, flexibility is crucial. The Trello board is reviewed and updated to reflect any changes in priorities or task allocations. This agile approach enables me to respond quickly to unforeseen challenges or adjustments in the project scope.

Summary:

By utilizing Trello in this structured yet flexible manner, I maintain a clear overview of the project's progress, ensure tasks are completed efficiently. This methodical approach to task allocation and tracking is instrumental in keeping the project on track within the constrained three-week timeframe.