Here I have described the whole development workflow of the Minimal Viable Product (MVP) of this User and Question Management system. We can modify and design it better in the next sprint.

For this Backend test, I have decided to build the API for this system using Django Rest Framework (DRF). Check the requirements.txt to install the required dependencies for this project.

**Data Models:**

* I have created the models.py file according to the specification I was given in the test’s Data Models. (Detailed Model documentation in a separate file)

**API Views:**

* Created five views.
* Implemented API views for managing Users and Questions, including pagination for the count of total favorite and read questions per user.
* Also, implemented filtering for questions by read, unread, favorite, and unfavorite status.
* Implemented views for both inserting and retrieving favorite and read questions.

(Detailed API Views documentation in a separate file)

**Performance Optimization:**

* Optimized database queries to ensure the response time is less than 200ms for the specified views.
* By optimizing the database queries using such as `prefetch\_related`, I tried to reduce the number of database hits and significantly improve the performance of these views, especially when dealing with a large amount of data like 10M users and 1M questions.
* Used pagination to manage large amounts of data efficiently.

**Data Generation:**

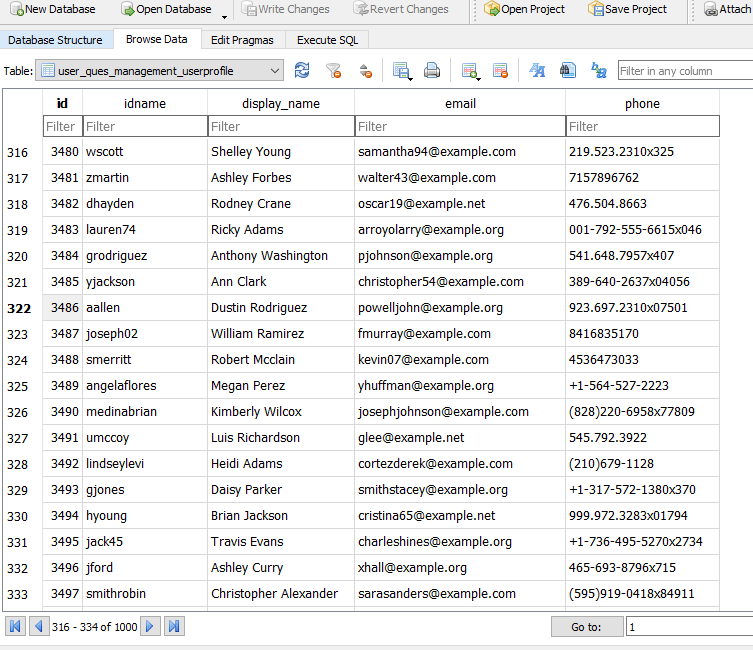
* Generated a significant amount of random data to simulate the intended scale of the application.
* Implemented a script to insert the generated data into the database.
* Here is the complete workflow to generate the necessary data and to populate the database:
* First Generate the data:
  + Created a new Python file (e.g., `data\_generation.py`) and place it in the same directory as your Django project.
  + Imported the necessary modules. Here I am using Faker (install it first).
  + Defined functions to generate user and question data.
  + Generated the user and question data and save it to CSV files.
  + Run the data generation script.



* + This will generate two CSV files: `users\_data.csv` containing user data and `questions\_data.csv` containing question data.
* Next Load the generated data into the database:
  + To use the generated data in my project, I created a custom management command to load the data into the database. I created a new Python file in the `management/commands` directory (following the [Official documentation](https://docs.djangoproject.com/en/4.2/howto/custom-management-commands/)) of the app within my Django project (e.g., `load\_data.py`).
  + In this file, define a command that reads the data from the CSV files and inserts it into the corresponding models.
  + Run the custom management command to load the data.



* + This will insert the generated user and question data into the corresponding models in the database. Here’s a screenshot of userprofile model in the database after loading:

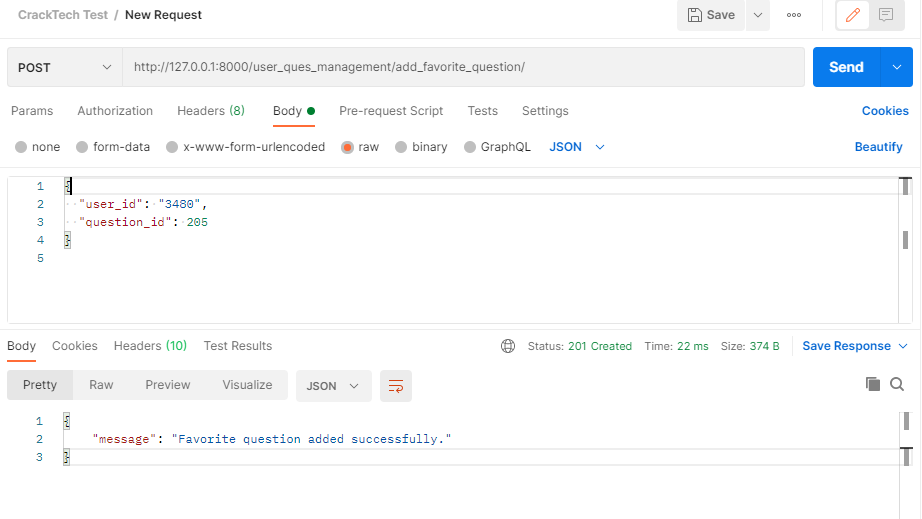


With these steps, I have generated the required data and loaded it into my Django application. We can now use this data to test and verify the functionalities of this Large-Scale User and Question Management System.

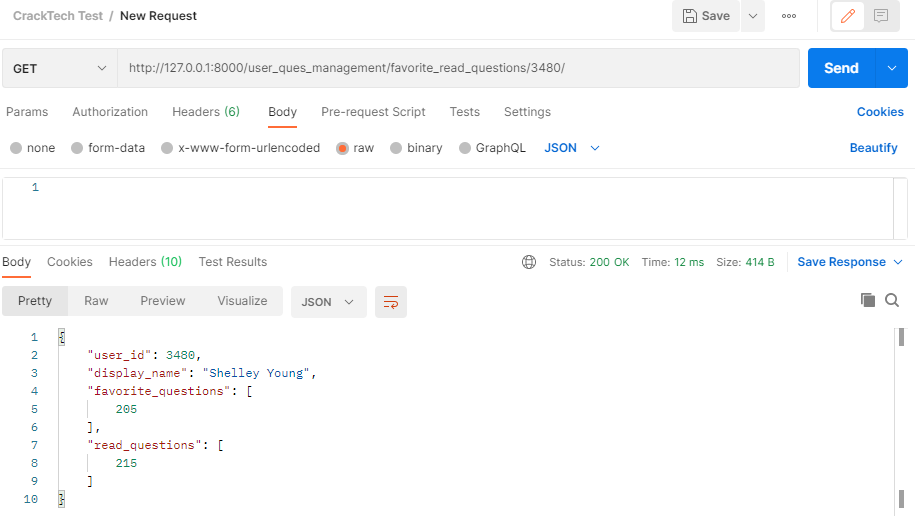
**Test:**

* Wrote comprehensive unit tests for all implemented API views in the **tests.py** file of the user\_question\_manager app. (Detailed tests documentation in a separate file)
* Used Django's built-in testing framework to run the tests.

Also checked the API endpoints using Postman. For example here is a screenshot of a POST request to add\_favorite\_question endpoint:



Another GET request to the 'favorite\_read\_questions/<int:user\_id>/':



**Scalability and Deployment Strategy:**

1. Database Optimization:
   * Database Indexing: We can create appropriate indexes for frequently queried fields in the UserProfile, Question, FavoriteQuestion, and ReadQuestion models to improve query performance.
   * Database Sharding: Implement sharding for the UserProfile and Question models, distributing data across multiple database instances based on user\_id or question\_id to handle large datasets efficiently.
   * Caching: We can utilize Redis for caching frequently accessed data such as user profiles, questions, and favorite/read status to reduce database load.
2. Efficient Querying:
   * Use Selective Fields: We optimized queries to fetch only the required fields instead of retrieving the entire objects when not necessary, reducing network overhead.
   * Query Optimization: Regularly analyze and optimize database queries to identify and eliminate performance bottlenecks.
3. Containerization:
   * Docker: Containerize the Django application using Docker to ensure consistent deployments across different environments.
4. Orchestration:
   * Kubernetes: Deploy the containerized application on a Kubernetes cluster to enable automatic scaling, load balancing, and high availability.
5. CI/CD Pipelines:
   * GitLab CI/CD or Jenkins: We can set up a CI/CD pipeline that triggers automated testing and deployment upon code changes to ensure a streamlined development workflow.
   * Automated Testing: Implement comprehensive automated tests, including unit tests for models and serializers, integration tests for views, and end-to-end tests for API endpoints.
6. Security:
   * Input Validation: Implement input validation and enforce data validation rules to prevent common security vulnerabilities like SQL injection and cross-site scripting (XSS) attacks.
   * Authentication and Authorization: Implement JWT-based authentication for users and role-based access control to restrict access to certain API endpoints.
   * Regular Security Audits: Conduct regular security audits to identify and address potential security vulnerabilities.

Deployment:

* The production environment can be hosted on a cloud platform (e.g., AWS, GCP, Azure) using Kubernetes for container orchestration.