Python Developer Assessment

Data Pusher

Overview

You have to create a Django web application to receive data into the app server for an account and send it across different platforms (destinations) from that particular account using webhook URLs.

Modules

1. **Account Module**
    - Each account should have
        - account id (unique to each account)
        - account name (Mandatory field)
        - App secret token (automatically generated)
        - Website (optional)
        - created_at (Mandatory field)
        - updated_at (Mandatory field)
        - created_by (Mandatory field)
        - updated_by (Mandatory field)
2. **Destination Module**
    - A destination belongs to an account. An account can have multiple destinations.
    - A destination has
        - URL (mandatory field)
        - HTTP method (mandatory field)
        - headers (mandatory field & have multiple values).
        - created_at (Mandatory field)
        - updated_at (Mandatory field)
        - created_by (Mandatory field)
        - updated_by (Mandatory field) Ex for headers:

```
{
    "APP_ID": "1234APPID1234",
    "APP_SECRET": "enwdj3bshwer43bjhjs9ereuinkjcnsiurew8s",
    "ACTION": "user.update",
    "Content-Type": "application/json",
    "Accept": "*"
}
```

3. **User Module**

- User has
  - email (Mandatory field & unique)
  - password (Mandatory field)
  - created_at (Mandatory field)
  - updated_at (Mandatory field)
  - created_by (Mandatory field)
  - updated_by (Mandatory field)

4. **Account Member Module**

- Account Member belongs to account and user
- Account Member has
  - account_id (Mandatory field)
  - user_id (Mandatory field)
  - role_id (Mandatory field)
  - created_at (Mandatory field)
  - updated_at (Mandatory field)
  - created_by (Mandatory field)
  - updated_by (Mandatory field)

5. **Role Module**

- Role has
  - id (unique to each role)
  - role_name (Mandatory field)
  - created_at (Mandatory field)
  - updated_at (Mandatory field)
- There are only two types of roles
  - Admin
  - Normal user
- You can create these roles in the database directly. No need to create a new API for this.

6. **Log Module**

- Log belongs to account and destination
- Log has
  - event_id (unique to each log)
  - account_id (Mandatory field)
  - received_timestamp (Mandatory field, when the data is received from the client)
  - processed_timestamp (Mandatory field, when the data is sent to the destination)
  - destination_id (Mandatory field)
  - received_data (Mandatory field)
  - status (Mandatory field) - (data sent successfully or not - status like success, failed)

7. **Data handler**

- Data handler receives only JSON data in the POST method.
- While receiving the data, it should have an app secret token sent through the header (header key is CL-X-TOKEN).

- While receiving the data, it should have an event_id (some random string but should be unique to each event) sent through the header (header key is CL-X-EVENT-ID).
- Based on the secret token, the account should be identified.
- Implement Async Process to send the data to the destinations.

8. **Authentication & Authorization**

   - Implement user authentication using Django's built-in authentication system or third-party libraries like django-allauth
   - Implement user registration, login, and logout functionality
   - There are two types of users:
     - Admin user: Has access CRUD operations for the accounts and destinations, able to read logs belongs to an account,able to CRUD operations for account members belongs to an account
     - Normal user: Has access Read And Update operations for the accounts and destinations, able to read logs belongs to an account, read account members belongs to an account
   - User who is registering to this app is always an admin user.
   - Admin user can add new users to the account he has access to and assign them as admin or normal user.
   - This is only applicable for CRUD operations for the accounts, destinations, account members, destinations available for account.

9. **Asynchronous Processing**

   - Use Django Channels or Celery to handle asynchronous tasks, such as sending data to destinations in the background.
   - Add logs for each event after the data is sent to the destination.
   - This is only applicable for the incoming data api (Data handler).

10. **Data Validation and Integrity**

    - Validate the data received from the client on all APIs.
    - Implement custom validators for model fields to ensure data integrity
      - ex. for email field, the email should be valid, for website field, the website should be valid etc.
    - If the data is not valid, send a response as "Invalid Data"
    - If the data is valid, send a response as "Data Received"
    - Use Django signals to enforce business rules, such as automatically deleting destinations when an account is deleted.
    - This is only applicable for CRUD operations for the accounts, destinations, account members, destinations available for account, logs.

11. **Advanced Querying and Filtering**

    - Implement advanced querying and filtering capabilities to retrieve specific data based on various modules to search. For exmaple if users want or able to search a specific accounts, destinations and logs based on their fields.
    - Use Django's ORM to query and filter data.

- Implement search functionality to search for data based on specific fields.
- Avoid n+1 queries in the application.
- This is only applicable for CRUD operations for the accounts, destinations, account members, destinations available for account, logs.

12. **Caching and Performance Optimization**

    - Implement caching strategies using Django's caching framework to improve performance for frequently accessed data.
    - Optimize database queries and use indexing to enhance performance. you need to think about fields you are using in the queries and create index for them.
    - This is only applicable for CRUD operations for the accounts, destinations, account members, destinations available for account, logs.

13. **API Rate Limiting and Throttling**

    - Implement rate limiting and throttling mechanisms to prevent abuse of the API.
    - Use Django-limiter or similar libraries to implement rate limiting.
    - This ratelimit is applicable for the incoming data api (/server/incoming_data).
    - Rate limit is 5 requests per second for an account.
    - This is only applicable for the incoming data api (Data handler).

14. **Comprehensive Testing**

    - Implement comprehensive testing for the application.
    - Implement unit tests for all functionalities using Django's test framework.
    - Test edge cases and error scenarios.
    - Ensure API endpoints handle failures gracefully.

15. **Documentation & API Specifications**

    - Document the API endpoints, request/response formats, and usage examples.
    - Provide clear instructions on how to use the API.
    - Use Swagger or similar tools to document the API.
    - Document all the endpoints and the parameters for each endpoint.
    - Document the error codes and the corresponding messages.
    - Document the request and response payloads for each endpoint.
    - This is applicable for all APIs.

Deliverables

1. Create a Django web application.
2. Create User Signup and Login flows.
3. Follow Authentication & Authorization, Data Validation and Integrity, Caching and Performance Optimization, Comprehensive Testing, Documentation & API Specifications For all APIs listed below.
4. Create JSON rest APIs. a. CRUD operations for an Account b. CRUD operations for Destinations. c. CRUD operations for Account Members.
5. Create an API to get destinations available for the account when the account id is given as input.
6. Create an API to get All Logs available for the account, filter logs based on the destination id, status, received_timestamp, processed_timestamp.

7. Create an API for receiving the data. follow Rate Limiting and Throttling, Asynchronous Processing
    - The API path is /server/incoming_data. The data should be received through the post method in the JSON format only.
    - The CL-X-TOKEN and CL-X-EVENT-ID should be received while receiving the data in the header.
    - If the HTTP method is GET and the data is not JSON while receiving the data, send a response as "Invalid Data" in JSON format {"success": false, "message": "Invalid Data"}
    - If the secret key is not received then send a response as "Unauthenticated" in JSON format. {"success": false, "message": "Unauthenticated"}
    - After receiving the valid data, using the app secret token, identify the account and send the data to its destinations using Async Process as a background task.
    - Add logs for each event after the data is sent to the destination.
8. Implementation of Mandatory Features
    - Implementation of proper Authentication and Authorization.
    - Ensure proper Asynchronous Processing for sending data to the destinations.
    - Enforce proper Data Validation and Integrity.
    - Enforce proper Rate Limiting and Throttling.
    - Implementation of proper Caching and Performance Optimization.
    - Implementation of proper Comprehensive Testing.
    - Maintain proper Documentation for APIs.
    - Implementation of Advanced Querying and Filtering.

Code Submission

1. Write the code in the latest version of Python Django web application.
2. Upload the code to Github in private repository
3. Give collaborator access to Github username: clabshr
4. Include all necessary configuration files, documentation, and database migrations in the repository.

Notes

- An account can have multiple destinations. For example, if an account is deleted, the destinations, logs, account members for that account should also be deleted.
- When adding a member to an account, you need to create a user if does not exist else use the existing user.
- If the destination's HTTP method is GET, then the incoming JSON data should be sent as a query parameter. If the method is POST or PUT, send the data as it is (JSON).

Assessment Evaluation Criteria

- Functionalities implemented
- Code Quality including DRY and SOLID principles, naming conventions, etc.
- Implementation of Mandatory Features
- Test Cases
- Error Handling
- Documentation