Python/Django Developer Assessment

Overview

This assessment consists of two tasks designed to evaluate your skills as a Python/Django developer. You may choose to attempt either Task 1 or Task 2 based on your expertise and preference. Each task is structured to simulate real-world scenarios and assesses your ability to write clean, maintainable, and efficient code.

- Task 1: API Development

- Task 2: Middleware Development

You may choose the task you feel more confident in or complete both for a comprehensive evaluation.

Please do not use ChatGPT for the purpose of this interview. While we provide GitHub Copilot Business plan and encourage all our employees to use ChatGPT; to ensure a fair interview process, we request you to refrain from using ChatGPT for this interview.

Submission guidelines

In your response email body, copy/paste below questions and provide your response:

1. Task Attempted: Which task did you attempt? (Task 1: API Development / Task 2: Middleware Development)
2. Completed Functionality: Have you completed the functionality as per the task requirements? (Yes/No/Partial)
3. Testing: Have you included unit test code for the functionality? (Yes/No). Please attach a screenshot or a copy of the test results for your unit tests.
4. Documentation Resources Used: Which documentation or resources did you reference while completing this task? (e.g., online documentation, ChatGPT, etc.)

Also include the attachments as requested for the chosen task.

Task: Step 1 - API Development

Objective

Develop a Django REST Framework (DRF) API endpoint that processes user data from a CSV file. This task evaluates your ability to build APIs, handle file uploads, validate data, and interact with a database effectively.

Requirements

1. API Functionality:
- Build a POST endpoint that allows uploading a CSV file.
- Parse the uploaded CSV file and validate its content based on the following rules:
  - `name`: Must be a non-empty string.
  - `email`: Must be a valid email address.
  - `age`: Must be an integer between 0 and 120.
- Save valid records into a `User` model.
- Respond with a JSON object summarizing:
  - The total number of records successfully saved.
  - The total number of records rejected.
  - Detailed validation errors for rejected records.

2. Constraints:
- The API must accept only files with a `.csv` extension.
- Duplicate email addresses should be gracefully skipped without causing errors.

3. Bonus Points:
- Use Django REST Framework serializers for validation.
- Add unit tests to validate the functionality of your API.

Submission Requirements

Provide the following in a GitHub repository or as a ZIP file:
1. Code: The complete source code for the API.
2. Sample Input: The CSV file you used for testing.
3. Sample Output: A JSON file containing the response generated by your API.
4. Documentation: README file on how to run and test your solution locally.
5. Unit tests: If you have developed unit tests, include screenshots or logs of the execution.
6. Ensure to follow the submission guidelines mentioned on first page as well.

Evaluation Criteria

1. Functionality: Does the API meet all the stated requirements?
2. Code Quality: Is the code readable, maintainable, and adherent to Django best practices?
3. Error Handling: Are edge cases handled effectively (e.g., invalid files, duplicate emails)?
4. Performance: Is the solution efficient when handling large CSV files?
5. Testing: Are there unit tests, and do they cover critical cases?

Task: Step 2 - Middleware Development

Objective

Develop a custom Django middleware to implement request rate limiting based on IP addresses. This task evaluates your ability to work with Django's middleware, caching systems, and efficient request handling.

Requirements

1. Middleware Functionality:
- Implement a middleware that tracks the number of requests made by a user (identified by their IP address).
- Block requests from an IP if the number exceeds 100 requests in a rolling 5-minute window.
- Return an HTTP 429 (Too Many Requests) status code for blocked IPs.

2. Implementation Details:
- Use Django's middleware format (`MiddlewareMixin` or equivalent).
- Store request data in a caching mechanism (e.g., Redis or Django cache framework).
- Include headers in the response to indicate the remaining allowed requests.

3. Constraints:
- Middleware must handle high-traffic scenarios efficiently.
- Ensure thread safety when handling shared data.

Submission Requirements

Provide the following in a GitHub repository or as a ZIP file:
1. Code: Complete source code for the middleware.
2. Documentation: A brief explanation of
  - How the rolling window rate limiting is implemented.
  - How to test your solution locally.
3. Example Input/Output: Provide example logs or debugging outputs showing the middleware in action.
4. Unit tests: If you have developed unit tests, include screenshots.
5. Ensure to follow the submission guidelines mentioned on first page as well.

Evaluation Criteria

1. Functionality: Does the middleware meet all stated requirements?
2. Code Quality: Is the code readable, maintainable, and adherent to Django best practices?
3. Error Handling: Are edge cases handled effectively (e.g., high traffic, missing headers)?
4. Performance: Is the middleware efficient in tracking and blocking requests under high traffic?
5. Testing: Are there sufficient unit tests or manual testing results provided?