FastAPI Real-World Assignment Questions

Scenario (read-only)

You are building an **Orders & Inventory** microservice for a tiny online store with two resources: `Product` and `Order`. Orders reduce product stock; a **payment webhook** marks orders as paid. You must implement CRUD, document the API, test it via multiple tools, deploy it on Render.com, then load test briefly.

Part A Environment & Project Setup

Questions

1. What Python version and minimal dependencies will you choose for a FastAPI CRUD service? Justify each package.
2. How will you structure folders so that your code is readable and testable (e.g., `app/`, `tests/`)? Provide the final tree.

Suggestions

- Keep it small: `fastapi`, `uvicorn`, and one of `sqlmodel`/`sqlalchemy`/in-memory store.
- Add `requests` for black-box tests.
- Include a `requirements.txt` with pinned versions.

Part B Data Modeling & Validation

Questions

1. Define a **Product** model with fields: `id`, `sku`, `name`, `price`, `stock`.
    - Which constraints are necessary (e.g., unique `sku`, `price > 0`, `stock >= 0`)?
    - What indexes would you add and why?
2. Define an **Order** model with fields: `id`, `product_id`, `quantity`, `status`, `created_at`.
    - Which statuses should be allowed?
    - How will you validate `quantity`?

Suggestions

- Decide whether to use **SQLite + SQLModel/SQLAlchemy** or a simple **in-memory dict** for speed. Either is acceptable if constraints are enforced.

- Keep `status` constrained (e.g., `PENDING|PAID|SHIPPED|CANCELED`) and document it.

Part C Endpoints & Behavior

Questions

Implement endpoints for both resources and answer:

1. **Products**
    - `POST /products`: What HTTP code for success? What error for duplicate `sku`?
    - `GET /products`: Will you add pagination? If not, why?
    - `GET /products/{id}`: What should a not-found response look like?
    - `PUT /products/{id}`: Should partial or full update be allowed? How to reject invalid fields?
    - `DELETE /products/{id}`: What is the correct success status code?
2. **Orders**
    - `POST /orders`: How do you atomically reduce stock and handle insufficient stock?
    - `GET /orders/{id}`: What data must be returned for basic tracking?
    - `PUT /orders/{id}`: Which fields can change? How will invalid state transitions be handled?
    - `DELETE /orders/{id}`: When is deletion allowed versus “cancel” semantics?

Suggestions

- Return **201** for resource creation, **404** for not found, **409** for conflicts (e.g., duplicate SKU or insufficient stock).
- For updates, prefer `PUT` with partial payloads using `exclude_unset=True`, but be consistent and document it.

Part D Error Handling & Contracts

Questions

1. List at least five realistic error cases and the exact JSON body you will return for each (include `detail` messages).
2. How will you ensure the API remains consistent if two orders try to buy the last item concurrently?

Suggestions

- Provide deterministic error shapes (e.g., `{"detail": "Insufficient stock"}`).
- If you're using SQLite or in-memory, document limitations around concurrency and how you'd fix them in production.

Part E API Documentation (Swagger UI)

Questions

1. Add OpenAPI metadata (title, version, description). What tags will you use for grouping endpoints?
2. Where will a new developer find the API docs in a running service?
3. Which two endpoints deserve extra description or examples in the OpenAPI schema, and why?

Suggestions

- Use FastAPI docstrings and `response_model`/`responses` to enrich `/docs`.
- Add example request/response bodies to at least one POST and one PUT.

Part F Black-Box Testing (Swagger, curl, Postman, Python)

Questions

1. Using **Swagger UI** only, demonstrate a full flow:
    - Create a product, list it, get it, update it, delete it.
    - Create an order, get/update/cancel it.
    - What response codes and bodies did you observe?
2. Construct **five curl commands** to cover the same flow. Paste the exact commands and responses you received.
3. Build a **Postman collection** with requests for local and deployed URLs. What variables did you use?
4. Write a short **Python (requests)** script that:
    - Creates a product and an order, asserts status codes, and prints the final order JSON.
    - Where would you plug this into a CI pipeline?

Suggestions

- Save screenshots of Swagger UI and Postman runs.
- In the Python script, read `BASE_URL` from an environment variable to reuse locally and in deployment.

Part G Payment Webhook (Security & E2E)

Questions

1. Design a `/webhooks/payment` endpoint that verifies an **HMAC-SHA256** signature header before processing.
    - What header name will you choose?
    - How do you compute the signature on the raw body?
2. On `payment.succeeded`, which order field(s) must change and how will you persist it?
3. How will you protect against **replay attacks** for the same event?
4. Show the exact test steps and proof (commands or code) you used to:
    - Generate a valid signature and call the webhook.
    - Call the webhook with an invalid signature and observe a 401/403.
    - Confirm the order moved to `PAID`.

Suggestions

- Keep the secret in `WEBHOOK_SECRET`. Do not log it or the computed HMAC.
- For local end-to-end tests, expose your app via **ngrok** and hit the public URL with signed payloads.

Part H Deployment on Render.com

Questions

1. Describe your deployment approach on Render: **dashboard** vs **`render.yaml`**. Why did you choose it?
2. What **build** and **start** commands** did you set? Where did `$PORT` come from?
3. Which environment variables did you configure on Render?
4. After deployment, paste the **public base URL** and show:
    - `/docs` opens and lists endpoints.
    - A live curl that creates a product and returns 201.

Suggestions

- Typical start: `uvicorn app.main:app --host 0.0.0.0 --port $PORT`.
- Set `WEBHOOK_SECRET` in Render dashboard or `render.yaml`.

Part I Post-Deployment Re-Testing

Questions

1. Re-run all tests (**Swagger, curl, Postman, Python**) against the **Render URL**.
    - What changed in latency and behavior vs local?
2. Which two errors did you intentionally trigger to confirm your production error handling works?

Suggestions

- Capture output (screenshots or logs) and include it in your submission.
- If you rate-limit yourselves, document Render's plan limits and implications.

Part J Load Testing (Locust) Exploration

Questions

1. Install **Locust** and write a minimal user behavior that mixes `GET /products` with create-order flows.
    - What ratio of reads:writes will you simulate and why?
2. Run two scenarios:
    - Light: ~25 users, spawn rate ~5/s.
    - Heavier: ~200 users, spawn rate ~20/s.
      Record median, 95th, 99th latencies, RPS, and error rates.
3. Where is the first bottleneck you observed? Propose one concrete fix.

Suggestions

- Start with short waits between tasks (e.g., 100–500 ms).
- Avoid test data collisions (randomize SKUs).

Part K Reflection & Hardening

Questions

1. List three production concerns you'd address next (e.g., DB transactions, migrations, auth/rate limits, retries, observability).
2. If this service joined a larger system, what contracts (SLA/SLO), dashboards, and alerts would you set up first?

3. What would you change in your API design after doing the tests?

Suggestions

- Mention idempotency for webhooks, API keys for write operations, and structured logging.

Deliverables (must be submitted)

1. **GitHub repo** with:
    - Code, `requirements.txt`, and a clear **README** describing endpoints, error shapes, and all execution/testing steps.
    - A **Postman collection** (exported JSON) that works both locally and on Render via variables.
    - A small **Python (requests)** script for smoke tests.
2. **Render URL** to the live API and screenshot of `/docs`.
3. **Test evidence**:
    - Swagger screenshots, curl transcripts, Postman runs, and Python script output.
    - Locust dashboard screenshots with a short written summary of results and observations.
4. **Reflection document** answering all questions above.

Grading Rubric (self-check)

- **API correctness (30%)**: CRUD works, proper status codes, validation, clear error JSON.
- **Webhook (20%)**: HMAC verification, correct status update, invalid signature handling, replay protection reasoning.
- **Testing completeness (20%)**: Swagger, curl, Postman, Pythonall performed locally and on Render with evidence.
- **Deployment quality (15%)**: Render runs reliably; `/docs` accessible; environment variables set.
- **Load testing & reflection (15%)**: Locust runs, metrics reported, bottleneck analysis, realistic next steps.