### 1. Project Summary: What the Project Does

The project is a **production-grade task management API** built with FastAPI and Python. Its primary function is to provide a robust, scalable backend for users to create, manage, and track tasks. Unlike a simple CRUD application, this API is designed with enterprise-level features, including secure user authentication, role-based access control for administrative functions, and asynchronous background processing to handle heavy tasks without blocking the main application.

The core technology stack includes:

* **FastAPI** for a high-performance, asynchronous web framework.
* **PostgreSQL** for reliable data persistence.
* **Redis** for high-speed caching and rate limiting.
* **Celery** for handling background jobs.

### 2. Technical Flow & Architecture

The architecture is a classic **microservice-oriented design** with distinct components for different responsibilities. The flow can be explained by following a user's journey.

1. **User Authentication:** A user registers or logs in via the /api/v1/auth/login endpoint. The API validates their credentials and returns a **JWT (JSON Web Token)**. This token is a digitally signed ticket that proves the user's identity.
2. **Protected Endpoint Access:** The user then includes this JWT in the Authorization header of subsequent requests. A dependency function (get\_current\_user) intercepts this token, decodes it, and verifies its authenticity and expiration. This dependency provides the user object to the endpoint, enabling both authentication and **Role-Based Access Control (RBAC)** checks (e.g., if current\_user.role != "admin").
3. **Task Creation:** When a user creates a new task via POST /api/v1/tasks/, the API performs two key actions:
   * It saves the task to the **PostgreSQL** database.
   * It dispatches an asynchronous **Celery** task. This is crucial for performance—the API immediately returns a response to the user, and the heavy work of fetching external data happens in the background.
4. **Background Processing & Caching:** The Celery worker picks up the job, fetches additional metadata for the task from an external API (e.g., a service that provides more context), and updates the task in the database. When the user later retrieves this task, the application first checks the **Redis** cache. If the data is found, it's returned immediately, dramatically improving response time. If not, it fetches from the database and caches the result for future requests.
5. **Rate Limiting & Resilience:** All API requests are protected by a Redis-backed rate limiter to prevent abuse. The system is also resilient, implementing **idempotency** to prevent duplicate actions from a user retrying a failed request.

### 3. Key Files & Their Purpose

* app/main.py: The entry point of the application. It sets up the FastAPI app, includes routers, and registers the custom exception handlers. It’s the orchestrator.
* app/api/v1/routes/user.py: The API router for user-related operations. It handles user creation, fetching user profiles (/me), and includes secure admin-only endpoints for managing all users.
* app/api/v1/routes/auth.py: Manages user authentication, including login and token creation.
* app/api/v1/routes/task.py: The main router for Task CRUD operations. It integrates with Celery for background jobs and Redis for caching and rate limiting. It's the most feature-rich router.
* app/api/v1/routes/comment.py: Provides API endpoints for creating, reading, and deleting comments on tasks.
* app/api/deps.py: Contains **FastAPI dependencies**. This file is a key part of the modular design, centralizing reusable logic like database session management (get\_db\_session) and authentication (get\_current\_user).
* app/core/security.py: Handles all security-related logic, including password hashing with bcrypt and **JWT token encoding and decoding**.
* app/core/cache.py: A singleton class that provides an interface to **Redis**. It handles connection management and serialization (JSON/pickle) for caching data.
* app/core/exceptions.py: Centralizes custom exception handling. It catches common errors (like database or Redis issues) and returns consistent, well-structured API responses.
* app/models/: Defines the **SQLAlchemy models** for the database tables (User, Task, Comment).
* app/schemas/: Defines **Pydantic schemas** for request bodies and response models. This is crucial for automated data validation and documentation.
* app/celery\_app/: Contains the Celery worker configuration and the background tasks (tasks.py, helpers.py) that run independently of the main API.

### 4. How to Talk About Testing & Achievements

#### What did you test?

You should be able to confidently say you followed a test-driven approach and implemented various types of tests to ensure the API's reliability. The provided code gives you specific examples:

* **Authentication Flow:** "I wrote integration tests to ensure that a user can successfully log in and that the generated JWT token can be used to access a protected endpoint, like /users/me." (Refer to app/tests/test\_auth.py).
* **Performance & Resilience:** "I created tests to validate key performance features. For example, the test\_rate\_limit.py file simulates exceeding the rate limit to confirm that the API correctly returns a 429 Too Many Requests status code, and test\_retries.py ensures that a failing external API call is retried."
* **Data Validation:** "I have a test to check that the API rejects invalid date formats for task filters, which shows our data validation works as intended." (Refer to app/tests/test\_task.py).

#### What did you achieve?

Focus on the "why" behind your technical choices.

* "I didn't just build a simple CRUD API; I designed a **scalable, production-ready system**."
* "I integrated a **caching layer with Redis** to drastically reduce database load for frequently accessed data, which is critical for high-concurrency scenarios."
* "I used **Celery for background job processing**, ensuring that the API remains responsive to users even when performing long-running or resource-intensive tasks like fetching external data."
* "I built a comprehensive **role-based access control (RBAC) system** that secures administrative endpoints, which is a non-negotiable feature for any multi-user application."
* "I implemented structured logging with pythonjsonlogger from the start, making the application **observable and ready for production monitoring systems** like ELK or Splunk."

Potential Interview Questions

**General & Project Flow**

* **"Walk me through the lifecycle of a request in your API, from the user's browser to the database."** "It starts with the user's request hitting the FastAPI application. The request goes through middleware for things like rate limiting and request ID generation. If it’s a protected endpoint, a dependency extracts the JWT token from the Authorization header, decodes and validates it, and then passes the user object to the route handler. The handler then uses a database session dependency to interact with the PostgreSQL database via SQLAlchemy, performing the requested operation. Once the data is retrieved or modified, Pydantic handles validation and serialization, and the response is sent back to the user."
* **"Why did you choose FastAPI over Flask or Django?"** "I chose FastAPI for its performance and modern features. Its **asynchronous capabilities** are key for building an I/O-bound application like a task manager, allowing it to handle many concurrent requests without blocking. It also provides automatic data validation and interactive API documentation with Swagger and ReDoc right out of the box, which significantly speeds up development and improves maintainability."
* **"Explain the role of each component in your architecture (FastAPI, Redis, PostgreSQL, Celery)."** "**FastAPI** is the core of the API, handling all HTTP requests and routing. **PostgreSQL** is our primary data store, providing relational integrity and persistence. **Redis** serves as a high-speed, in-memory cache to reduce database load and as a backend for our rate limiting feature. Finally, **Celery** is our asynchronous task queue, which offloads heavy or long-running tasks, like fetching external metadata, from the main API process to a separate worker, keeping the API fast and responsive."
* **"What's the difference between synchronous and asynchronous code, and why is that important for an API?"** "Synchronous code runs one instruction at a time, so it has to wait for an operation to complete before moving on. Asynchronous code, on the other hand, can 'pause' an operation while it waits for an external event—like a database query or an API call—and then handle other tasks. This is crucial for a web API because it allows the server to handle thousands of simultaneous connections without creating a new thread for each, which would consume a lot of memory. It makes the API much more scalable and efficient."

**Security & Authentication**

* **"How does your JWT authentication flow work?"** "A user submits their credentials to the login endpoint. I use passlib to hash their password and compare it against the stored hash in the database. If they match, I generate a JWT using the jose library, encoding the user's ID and role. The token is signed with a secret key. This token is then returned to the user, who must include it in the Authorization header of all future requests. On the server side, a dependency function decodes the token, validates its signature, and ensures it hasn't expired, securely identifying the user for subsequent actions."
* **"Why is it important to hash passwords, and why did you choose bcrypt?"** "It's critical to hash passwords to protect them from theft. If our database were compromised, the attackers would only get the hashes, not the original passwords. I chose **bcrypt** because it's a modern, industry-standard hashing algorithm that is designed to be slow and computationally intensive, which makes it resistant to brute-force attacks and rainbow table attacks."
* **"What is RBAC, and how did you implement it?"** "**RBAC** stands for Role-Based Access Control. It's a method of restricting access to resources based on a user's role, such as 'admin' or 'user.' I implemented it by storing the user's role in the JWT. The get\_current\_user dependency automatically provides the user's role, so I can simply check it at the start of an endpoint function. For example, the /users endpoint has an explicit if current\_user.role != 'admin' check to ensure only administrators can list all users."

**Databases & Models**

* **"Why use an ORM like SQLAlchemy, and what's the difference between AsyncSessionLocal and a standard session?"** "I used SQLAlchemy because it allows me to interact with the database using Python objects, which is far more intuitive and less error-prone than writing raw SQL. It also makes the code database-agnostic. The key difference with AsyncSessionLocal is that it's designed to work with **asynchronous drivers**, ensuring that our database queries don't block the FastAPI event loop, which is essential for our non-blocking API architecture."
* **"How would you handle a database migration to add a new field to the Task model?"**

I'd use a tool like **Alembic** to manage the database schema. The process would be to:

1. **Define the new field** in the Task model within the models/task.py file. For example, completion\_date = Column(DateTime).
2. **Generate a migration script** by running an Alembic command like alembic revision --autogenerate -m "Add completion date to Task model". Alembic would automatically detect the new column and create a Python file with upgrade() and downgrade() functions.
3. **Review the script** to ensure it looks correct. The upgrade() function would contain the SQL to add the new column.
4. **Apply the migration** to the database by running alembic upgrade head.

This approach ensures schema changes are version-controlled, repeatable, and safely applied across all environments.

* **"Explain the purpose of the relationship() function in SQLAlchemy."** "relationship() is the core of SQLAlchemy's ORM. It defines how models are connected to each other. For example, in our project, it connects a Task to a User via the owner\_id. This allows us to write code like task.owner to access the user object directly, without needing to write a separate query to join the tables. It handles all the underlying join logic for us."

**Performance & Caching**

* **"What is the purpose of caching in your application?"** "The main purpose is to **reduce latency and database load**. By caching frequently accessed data in Redis, we can serve requests much faster because reading from an in-memory cache is significantly quicker than a round trip to the database. This is especially useful for paginated lists of tasks or individual task details that are requested often."
* **"How do you handle cache invalidation?"** "I use a **write-through cache strategy**. When a user creates, updates, or deletes a resource, I make sure to invalidate the relevant cache entries. For instance, when a task is updated, I delete the single-task cache key (task:{task\_id}) and use a delete\_pattern command to clear any cached lists of tasks (tasks:user:{user\_id}:\*). This ensures that the cache is never stale and users always see the most up-to-date data."
* **"Can you explain what rate limiting is and how you configured it?"** "Rate limiting is a security measure that restricts the number of requests a client can make to an API within a given time period. I used slowapi, which is a FastAPI extension, with a Redis backend. I configured it to limit the number of requests based on the authenticated user's ID, which is a much more effective strategy than limiting by IP address, as it's more accurate and prevents a single user from consuming all the requests from a shared network."

**Background Jobs**

* **"Why use a separate tool like Celery for background tasks instead of just running them in a thread?"** "Using Celery adds **scalability and reliability**. Unlike a simple thread, Celery provides a task queue and worker architecture. If a task fails, Celery can automatically retry it. If a worker goes down, the task remains in the queue and can be processed by another worker when it comes back online. This makes the system far more resilient and allows me to scale the workers independently from the main API."
* **What happens if a Celery task fails? How would you handle that?**

By default, Celery will not retry a failed task. However, you can configure it to be more resilient. To handle task failures, I would use the @celery\_app.task(bind=True, default\_retry\_delay=300, max\_retries=5) decorator.

* The bind=True argument passes the task instance itself to the function, allowing me to call self.retry().
* The default\_retry\_delay specifies a delay (e.g., 5 minutes).
* The max\_retries sets a maximum number of attempts before giving up.

Inside the task, I would wrap the external API call in a try...except block. If a failure occurs—for example, due to a network error or a 500 status code—I'd log the error and then call self.retry(). This handles transient failures gracefully, ensuring that a task is eventually completed even if a dependency is temporarily unavailable.

**Best Practices**

* **What are FastAPI dependencies, and why are they useful?**

FastAPI dependencies are functions that are run before a request handler. They are a powerful feature for sharing common logic across multiple endpoints.

They are incredibly useful because they:

* **Promote code reuse:** Logic like authenticating a user, getting a database session, or checking user permissions can be written once and reused everywhere.
* **Keep route handlers clean:** By abstracting boilerplate code into dependencies, the main endpoint functions remain focused on their core business logic.
* **Enable testability:** You can easily 'override' a dependency for testing purposes, allowing you to test your endpoint logic in isolation without a real database or authentication.

In our project, get\_current\_user is a perfect example; it handles all authentication logic, allowing every protected route to simply declare current\_user: User = Depends(get\_current\_user).

* **Explain the purpose of Pydantic and how it helps with validation.**

Pydantic is a Python library used for data validation and parsing. It helps us define data schemas, ensuring that the data we receive from an API request and the data we send back in a response is correct and in the right format.

Pydantic helps with validation by:

* **Enforcing a contract:** We define models like UserCreate with specific field types (e.g., str, EmailStr, int). When a request comes in, Pydantic automatically validates that the data conforms to this model.
* **Providing helpful errors:** If the data is invalid, Pydantic raises a detailed ValidationError which FastAPI automatically converts into a clear, concise 422 Unprocessable Entity response, telling the user exactly what went wrong.
* **Automating documentation:** Because FastAPI is built on Pydantic, it can automatically generate interactive API documentation showing the expected request and response bodies. This saves a massive amount of time and reduces manual effort.
* **What are structured logs, and why are they better than plain text logs?**

Structured logs are a format where log messages are machine-readable, typically using a format like **JSON**. Instead of a single line of text, a structured log entry contains key-value pairs that describe the event. For example, a log might have {"timestamp": "...", "level": "INFO", "user\_id": 123, "message": "Task created"}.

They are far better than plain text logs because they:

* **Are easily searchable and filterable:** You can query for specific fields across millions of logs, like finding all log entries for user\_id: 123 or all level: "ERROR" messages, which is nearly impossible with unstructured text.
* **Support automated analysis:** Log management systems like ELK Stack or Datadog can automatically parse and index structured logs, making it easy to create dashboards, alerts, and performance metrics.
* **Provide richer context:** By including key-value pairs for context (e.g., user\_id, request\_id, task\_id), you get a much clearer picture of what was happening at the time of the event, which is invaluable for debugging in production.
* **What is an idempotency key, and why did you implement it?**

An **idempotency key** is a unique value sent with a request. The server uses this key to ensure that the same request, if sent multiple times, only results in a single action. I implemented this on the create\_task endpoint. If a user's network connection is unstable and they retry the same request, the API will recognize the idempotency key and return the cached success response from the first attempt instead of creating a duplicate task. It’s a key part of building a reliable and fault-tolerant API."

\*\*

**. Scaling & Concurrency**

* **Q: How does your API handle high concurrency?**
  + **A:** My API is built on **FastAPI**, which uses **ASGI** (Asynchronous Server Gateway Interface) and an event loop. This allows it to handle many concurrent connections with minimal overhead by asynchronously waiting on I/O-bound operations like database queries or external API calls. I've further optimized this with **Redis caching** to reduce the number of expensive database trips. For long-running tasks, I offload them to a dedicated **Celery worker**, ensuring the API remains fast and responsive even under heavy load.
* **Q: What would be the next step to scale your application?**
  + **A:** The next step is horizontal scaling. I'd containerize the API and the Celery worker using Docker and deploy them on a platform like **AWS ECS Fargate**. I'd place the API behind a **load balancer** to distribute traffic across multiple instances, and configure **auto-scaling** based on CPU usage or request count. The Redis and PostgreSQL services would be managed by platforms like **Amazon ElastiCache** and **Amazon RDS** to ensure high availability and automatic failover.
* **Q: What are the trade-offs of using an asynchronous framework?**
  + **A:** The primary trade-off is the complexity of managing the event loop and understanding coroutines (async/await). It requires all I/O-bound libraries to be asynchronous. Also, a long-running, CPU-bound task will block the event loop, so such tasks must be offloaded to a separate process or worker pool. My project addresses this by using **Celery** for CPU-bound or blocking I/O work.

**2. Security & Data Integrity**

* **Q: How do you handle secrets and sensitive data in your project?**
  + **A:** I use a dedicated **.env file** and the **Pydantic BaseSettings** to manage environment-specific variables like the database URL and secret key. This keeps sensitive information out of the source code. In a production environment, I would use a secrets manager like **AWS Secrets Manager** or **HashiCorp Vault** to securely store and retrieve these credentials at runtime, making them inaccessible to developers and build processes.
* **Q: What is a SQL injection, and how does your project prevent it?**
  + **A:** A SQL injection is a code injection technique where an attacker exploits a web application's database to run malicious queries. My project prevents this by using an **ORM (Object-Relational Mapper)**, **SQLAlchemy**. The ORM automatically sanitizes and parametrizes all queries, meaning it separates user-provided data from the SQL command itself, rendering SQL injection attacks ineffective.
* **Q: Your Idempotency-Key implementation is great. Can you explain the underlying mechanism and potential race conditions?**
  + **A:** The key to idempotency is using a unique identifier and a shared, persistent store. My implementation uses Redis with a key-value pair (idempotency:{key}) that stores the request's response. The potential race condition occurs if two requests with the same key arrive at the exact same moment. To handle this, a distributed lock is often used. A more robust solution would be to use a Redis command like **SETNX** (set if not exists) or a transaction to atomically check for the key's existence and set the response, ensuring only the first request can proceed with the operation.

**3. API Design & Best Practices**

* **Q: How do you version your API, and why is it important?**
  + **A:** My API is versioned using the URL path, specifically /api/v1/. Versioning is crucial for maintaining backwards compatibility. As new features are added or breaking changes are introduced, a new version (/v2/) can be released. This allows older clients to continue using the previous version of the API without being forced to update immediately, providing a smoother transition and preventing service disruptions.
* **Q: What is the purpose of the PaginatedResponse schema?**
  + **A:** The PaginatedResponse schema is a generic Pydantic model that provides a consistent, standardized way to handle list-based responses. It includes not only the data (items) but also essential metadata like total count, page number, and has\_next flags. This prevents inconsistencies across different endpoints and makes it easy for front-end developers to implement pagination logic on their side.
* **Q: What would you do if your Celery worker became a performance bottleneck?**
  + **A:** The great thing about Celery is its scalability. First, I'd check if the bottleneck is due to CPU or I/O. If it’s CPU-bound, I'd simply **increase the number of Celery worker instances** on my cloud platform. If it's I/O-bound (e.g., waiting for an external API), I'd explore using a **concurrent pool of workers** or even look into more efficient task queues designed for I/O-bound tasks. I would also investigate if the external API calls themselves can be parallelized within the Celery task.

\*\*

That's a very solid foundation. To truly excel, you should be prepared for questions that test your depth of experience. Here are more advanced questions and answers to show you're a senior candidate.

**1. Advanced Architecture & Design**

* **Q: How would you implement a simple search API for tasks, and what are the trade-offs of each approach?**
  + **A:** For a simple case, I’d use LIKE queries in PostgreSQL with appropriate database indexes. However, this isn't scalable for large datasets or complex queries. For a more robust solution, I'd integrate a dedicated search engine like **Elasticsearch** or **OpenSearch**. We'd use a background job to index new tasks, comments, and users in real-time. This provides powerful full-text search, faster query times, and features like fuzzy matching, but adds an extra service to maintain.
* **Q: If a user's role changed from 'user' to 'admin', how would you handle the authentication token without forcing them to log out and back in?**
  + **A:** That's a great question about token management. A JWT is a stateless token, so its payload is fixed at creation time. The most direct approach is to force re-authentication. However, for a better user experience, I could implement a **token refresh mechanism**. The server would issue a short-lived access token and a long-lived refresh token. When the access token expires, the client uses the refresh token to request a new one, and at that point, the server can check for any role changes and issue a new JWT with the updated permissions.
* **Q: Your Comment model has task\_id and user\_id. Is that a good design, or is a join table better?**
  + **A:** The current design with foreign keys is correct for this one-to-many relationship (one user can have many comments, one task can have many comments). A join table would be necessary for a **many-to-many relationship**, such as if a task could have multiple assignees. In that case, we would create an AssignedTasks join table with foreign keys to both users and tasks.

**2. Monitoring & DevOps**

* **Q: What is a health check, and what would your health check endpoint (/health) return?**
  + **A:** A health check is an endpoint used by monitoring systems and load balancers to determine if a service is running and healthy. My /health endpoint would return a 200 OK status and a simple JSON response like {"status": "ok"}. For a more detailed check, I would include the status of dependencies like the database and Redis cache, so I can detect failures even if the API process itself is running. This is called a **deep health check**.
* **Q: How would you set up a CI/CD pipeline for this project?**
  + **A:** I'd use **GitHub Actions**. The pipeline would be triggered on every push to the main branch. The steps would be:
    1. **Run tests:** Execute pytest to ensure all tests pass.
    2. **Linting/Static Analysis:** Run tools like Flake8 or Black to check code quality.
    3. **Build Docker images:** Build separate images for the API and the Celery worker.
    4. **Push to a container registry:** Push the images to **Amazon ECR**.
    5. **Deploy:** Update the **Amazon ECS Fargate** service to use the new image. This ensures a consistent and automated path from code to production.
* **Q: Your logger uses a JSON formatter. Why is this better for production, and what's next for observability?**
  + **A:** A JSON formatter makes logs **machine-readable**, which is crucial for modern observability. Log management platforms like Splunk or AWS CloudWatch can easily parse, index, and query JSON logs, enabling me to quickly search for errors or analyze trends. The next step would be to add a **tracing layer**, such as **OpenTelemetry**. This would instrument the code to track a request as it flows through the API, to the database, to the Redis cache, and to the Celery worker, giving me a complete view of a request's journey and pinpointing bottlenecks.

**3. Python & FastAPI Internals**

* **Q: What is the purpose of from\_attributes = True in your Pydantic schemas?**
  + **A:** This is a Pydantic V2 setting that tells the model to allow population from an object's attributes. In our case, it allows us to create a Pydantic TaskRead object directly from an SQLAlchemy Task model instance, like TaskRead.from\_orm(task). It eliminates the need for manual mapping and is a key feature for a seamless **ORM-Pydantic integration**.
* **Q: You use AsyncSessionLocal. Why not a normal SessionLocal?**
  + **A:** Standard SessionLocal creates a blocking session that uses synchronous I/O. If a database query takes 100ms, it would block the entire FastAPI event loop for that duration, preventing it from handling other requests. The AsyncSessionLocal uses an **asynchronous database driver** (like asyncpg), which allows the event loop to pause and handle other tasks while it waits for the database query to complete. This is the correct pattern for writing truly non-blocking, high-performance APIs with FastAPI.
* **Q: What is dependency injection and how does FastAPI handle it?**
  + **A:** Dependency injection is a software design pattern where a component receives its dependencies from an external source rather than creating them itself. FastAPI handles this by using the Depends() function. When a route handler is called, FastAPI inspects its function signature, sees the Depends() keyword, and automatically calls the dependency function to resolve its value (e.g., getting a database session) before passing it to the handler. This makes the code modular, reusable, and much easier to test.

## Final Round: Seniority & Real-World Scenarios

### 1. Architectural Design & Philosophy

* **Q: Your project is structured with different routers and modules. Would you consider this a monolith or a set of microservices? What are the advantages and disadvantages of this approach?**
  + **A:** "I would describe this architecture as a **modular monolith**. It’s a single deployable unit, which simplifies deployment and development initially. The key is that the code is well-structured and decoupled into logical domains like users and tasks. The main advantage is that it’s easy to manage and test. The disadvantage is that it can become a bottleneck if different parts of the application have vastly different scaling requirements. As a next step, I'd consider splitting a part of it into a microservice, for example, a separate 'notifications' service that Celery talks to, to handle high-volume email traffic independently."
* **Q: You used Pydantic for data validation. What if you received a request with an extra field not in your schema? How does Pydantic handle this by default, and how would you configure it for strictness?**
  + **A:** "By default, Pydantic is permissive and will ignore extra fields in the request body, which is a flexible design choice. To make it stricter, I would set the ConfigDict(extra='forbid') in the Pydantic model's Config class. This would raise a validation error and return a clear 422 response if any unexpected fields are present. I would make this decision based on the API's public-facing nature; for a public API, it's often better to be lenient, while for a private API, being strict can prevent data corruption."

### 2. Advanced Testing & Quality

* **Q: How would you write an integration test for a task that relies on a Celery worker and an external API call?**
  + **A:** "This requires a more advanced testing strategy. I would:
    1. Start the FastAPI app, a test database, and a Celery worker in a separate thread or process for the test run.
    2. Use a library like unittest.mock to **mock the external API call** and control its response, so my test doesn't rely on an external service.
    3. Make a POST request to the API to trigger the Celery task.
    4. Wait for a short period (e.g., time.sleep()) to allow the Celery worker to process the task.
    5. Finally, assert that the database record has been updated with the expected data from the mocked external API response. This ensures the entire asynchronous flow works as intended."

### 3. Dependency Management & Project Structure

* **Q: How do you handle project dependencies, and why did you create a separate requirements.txt file for development?**
  + **A:** "I use poetry for dependency management. It’s more modern than pip and handles dependency resolution automatically. All project dependencies are in the pyproject.toml file, which includes versions. For a production deployment, I would use the poetry export command to generate a pinned requirements.txt file. This guarantees that the exact same versions are installed in production, preventing any dependency-related surprises. The separate file is crucial for reproducible builds."

### 4. Database & Caching at Scale

* **Q: You have a single PostgreSQL database. What would be your strategy if you noticed read operations were becoming a bottleneck?**
  + **A:** "I would first check the queries and add **indexing** where needed to speed up searches. If that's not enough, my next step would be to implement **database read replicas**. The primary database would handle all writes, and the replicas would serve all read traffic. The application would have to be configured to send read queries to the replicas and write queries to the primary. This distributes the read load and significantly boosts performance. If both reads and writes become an issue, then sharding or partitioning the database would be a more advanced solution."
* **Q: Why did you choose Redis for caching over just caching in memory within the FastAPI application?**
  + **A:** "In-memory caching is an option, but it has two major drawbacks. First, the cache would be lost every time the application restarts. Second, it wouldn't be shared across multiple instances. If I scaled my application to three instances, each one would have its own empty cache, leading to poor cache hit ratios. Using a separate, external **Redis instance** solves both of these problems. It provides a persistent, centralized cache that all instances can access, which is essential for scaling horizontally."
* **Q: Your project uses two dependencies, sqlalchemy and redis-py. What would you do if a security vulnerability was found in one of them?**
  + **Strategy:** Show a systematic approach to problem-solving.
  + **Answer:** "My first step would be to check the official security advisories for the library and a vulnerability database like the National Vulnerability Database (NVD). If a patch is available, I would update the dependency to the secure version, run all my tests, and then push the change to a staging environment for validation before deploying to production. If no patch is available, I'd assess the risk. If the vulnerability is critical and affects my application's functionality, I might temporarily disable the feature that uses the library or look for a temporary workaround. I would also closely monitor the community and the maintainers for a fix."
* **Q: You’ve implemented an idempotency key. What if a user's request fails, but the API has already started to process the task, so the idempotency key is already stored in Redis?**
  + **Strategy:** This tests your understanding of distributed systems and error states.
  + **Answer:** "This is a key challenge with idempotency. It's a race condition. The correct way to handle this is to use a **status field** in the idempotency cache. When the request first arrives, I'd save the idempotency key with a status of 'processing'. If a subsequent request with the same key arrives, I'd see the 'processing' status and return a 409 Conflict or a 425 Too Early error. If the original request fails, I would update the status to 'failed' in the cache. The client can then retry a new request. Once the original request succeeds, I'd update the status to 'completed' and store the final response."
* **Q: Your User model has a hashed\_password field. How would you handle a password reset for a user?**
  + **Strategy:** This tests your understanding of a common, but non-trivial, security flow.
  + **Answer:** "I'd implement a secure password reset flow that doesn't expose any sensitive information. The process would be:
    1. The user requests a password reset by providing their email address.
    2. The API generates a unique, single-use, **time-limited token** (e.g., a UUID with a 15-minute expiry).
    3. The API saves the token and its expiration time in the database and sends a password reset link containing this token to the user's email address.
    4. The user clicks the link, which brings them to a front-end page where they enter a new password. The request to change the password includes the token.
    5. The API validates the token, checks its expiration, and hashes and updates the user's password in the database.
    6. The token is then immediately invalidated to prevent its reuse. This ensures the process is secure and a malicious actor cannot simply change a user's password."

Here is the one final piece of advice that will help you tie everything together and ace the interview:

Focus on the **Trade-Offs**.

A junior developer knows how to use a library. A senior developer knows when and why to use it, and what alternatives exist. For each of your key components, be ready to discuss a trade-off.

* **FastAPI vs. Django:** You get high performance and async capabilities, but at the cost of less built-in functionality (e.g., no ORM or admin panel out of the box).
* **Redis Caching vs. Database:** You gain speed and reduce database load, but add architectural complexity and a potential single point of failure.
* **Celery vs. In-Process Threads:** You get resilience and horizontal scalability, but add a separate service to manage and monitor.
* **JWT vs. Session-Based Auth:** JWT is stateless and scalable across multiple servers, but you lose the ability to easily revoke a token before it expires.

By speaking in terms of trade-offs, you demonstrate a thoughtful, experienced approach to engineering decisions, proving you can architect a system for the real world, not just for a coding challenge.

## EventBridge Vs Lambda Trigger

## 1. Lambda Triggers (Direct)

* A **Lambda trigger** is anything that directly invokes a Lambda function.
* Common examples:
  + **S3** → when a file is uploaded, trigger Lambda.
  + **DynamoDB Streams** → when data changes, trigger Lambda.
  + **API Gateway** → when an API endpoint is hit, trigger Lambda.
  + **SNS** → when a message is published, trigger Lambda.

👉 In this case, the **event source talks directly to Lambda**.

* Good for **simple, point-to-point integrations**.
* Example: “Whenever a new image is uploaded to S3 → run my image-resize Lambda.”

## 2. EventBridge (Indirect, Event Bus)

* **EventBridge** is an **event bus** (routing system).
* Services (like S3, EC2, custom apps) publish events to the bus.
* You define **rules** on the bus to decide where events should go:
  + To a **Lambda**
  + To an **SQS queue**
  + To **Step Functions**
  + To **another Event Bus**

👉 EventBridge = **decoupled, flexible event routing**.

* Good for **complex, multi-destination event-driven architectures**.
* Example: “Whenever an EC2 instance stops, send notification to Lambda, log to CloudWatch, and also forward to Slack via EventBridge → multiple consumers, no tight coupling.”

## 

## 🔹 Example

* **S3 → Lambda Trigger**: Resize image when uploaded to bucket.
* **S3 → EventBridge → Lambda + SQS + Step Function**: On upload, notify Lambda, send metadata to queue, and start a workflow — all from the same event.

👉 So:

* Use **Lambda trigger** for **one-to-one, simple reactions**.
* Use **EventBridge** when you need **flexible routing, filtering, or fan-out**.