FastAPI Coding Challenges

Interview Questions and Answers

Core Concepts

This section focuses on fundamental principles and advanced concepts that an experienced developer should master.

1. Implement a FastAPI endpoint with proper dependency injection that validates JWT tokens and extracts user claims for authorization. Handle token expiration and invalid signatures gracefully.

JWT Authentication with Dependency Injection

This solution demonstrates **proper separation of concerns** using FastAPI's dependency injection system with JWT validation:

```
from fastapi import Depends, HTTPException, status
from fastapi.security import HTTPBearer, HTTPAuthorizationCredentials
import jwt

security = HTTPBearer()

async def verify_token(credentials: HTTPAuthorizationCredentials = Depends(security)):
    try:
        payload = jwt.decode(credentials.credentials, "SECRET", algorithms=["HS256"])
        return payload
    except jwt.ExpiredSignatureError:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Token expired")
    except jwt.InvalidTokenError:
        raise HTTPException(status_code=status.HTTP_401_UNAUTHORIZED, detail="Invalid token")
```

Key concepts:

- **HTTPBearer** automatically extracts the Authorization header
- **Depends()** creates reusable authentication dependencies
- Proper exception handling for expired and invalid tokens
- Returns decoded payload for downstream use in route handlers
- Follows OAuth2 security standards with bearer token pattern

2. Create a FastAPI background task system that processes uploaded files asynchronously, tracks progress using Redis, and sends webhook notifications upon completion. Include error handling and retry logic.

Async Background Task Processing

This implementation uses **BackgroundTasks** with Redis for progress tracking and robust error handling:

```
from fastapi import BackgroundTasks, UploadFile
import aioredis
import httpx

async def process_file(file_id: str, redis: aioredis.Redis):
    try:
        await redis.set(f"progress:{file_id}", "processing")
        # Simulate processing
        await redis.set(f"progress:{file_id}", "completed")
        async with httpx.AsyncClient() as client:
            await client.post("https://webhook.url", json={"file_id": file_id, "status": "done"})
    except Exception as e:
        await redis.set(f"progress:{file_id}", f"error:{str(e)}")
```

Advanced patterns demonstrated:

- **Async I/O operations** for non-blocking file processing
- **Redis integration** for distributed progress tracking
- **httpx AsyncClient** for webhook notifications
- Graceful error handling with status persistence
- Decoupled task execution from request lifecycle

3. Design a FastAPI middleware that implements rate limiting per API key with sliding window algorithm, stores state in Redis, and returns appropriate headers (X-RateLimit-Remaining, X-RateLimit-Reset).

Custom Rate Limiting Middleware

This middleware implements a **sliding window rate limiter** with proper HTTP headers:

```
from starlette.middleware.base import BaseHTTPMiddleware
import time

class RateLimitMiddleware(BaseHTTPMiddleware):
    async def dispatch(self, request, call_next):
        api_key = request.headers.get("X-API-Key")
        now = int(time.time())
        key = f"ratelimit:{api_key}:{now // 60}"
        count = await redis.incr(key)
        await redis.expire(key, 60)
        if count > 100:
            return JSONResponse({"error": "Rate limit exceeded"}, status_code=429)
        response = await call_next(request)
        response.headers["X-RateLimit-Remaining"] = str(100 - count)
        return response
```

Implementation highlights:

- **Sliding window** using time-bucketed Redis keys
- **Atomic operations** with INCR for race condition safety
- Standard rate limit headers for client awareness
- Automatic key expiration to prevent memory leaks
- Per-API-key isolation for multi-tenant scenarios

4. Implement a FastAPI WebSocket endpoint that supports real-time chat with message broadcasting, user presence tracking, and automatic reconnection handling. Use connection pooling and handle disconnections gracefully.

Production-Grade WebSocket Implementation

This solution demonstrates **connection management** and broadcasting patterns:

```
from fastapi import WebSocket, WebSocketDisconnect
from typing import Dict

class ConnectionManager:
    def __init__(self):
        self.active_connections: Dict[str, WebSocket] = {}

    async def connect(self, user_id: str, websocket: WebSocket):
        await websocket.accept()
        self.active_connections[user_id] = websocket

    async def broadcast(self, message: str, exclude: str = None):
        for user_id, connection in self.active_connections.items():
            if user_id != exclude:
                await connection.send_text(message)
```

Key architectural decisions:

- **ConnectionManager pattern** for centralized connection tracking
- **Dictionary-based storage** for O(1) user lookups
- Selective broadcasting with exclusion support
- Proper connection lifecycle management
- Foundation for scaling with Redis pub/sub or message queues

5. Create a FastAPI endpoint with advanced query parameter validation including regex patterns, custom validators, and dependent fields. Implement pagination with cursor-based navigation for large datasets.

Advanced Query Validation and Pagination

This demonstrates **Pydantic validators** with cursor-based pagination:

```
from fastapi import Query
from pydantic import BaseModel, validator, Field
import re

class SearchParams(BaseModel):
    query: str = Field(..., min_length=3, max_length=100)
    cursor: str = None
    limit: int = Field(default=20, ge=1, le=100)

    @validator('query')
    def validate_query(cls, v):
        if not re.match(r'^[a-zA-Z0-9\s]+$', v):
            raise ValueError('Query contains invalid characters')
        return v
```

Advanced validation features:

- **Field constraints** for min/max length and range validation
- **Custom validators** with regex pattern matching
- **Cursor-based pagination** for consistent large dataset traversal
- Prevents SQL injection through input sanitization
- Type-safe query parameter parsing with automatic documentation

6. Implement a FastAPI dependency that manages database transactions with proper connection pooling, automatic rollback on errors, and context manager support for nested transactions.

Database Transaction Management

This solution uses **async context managers** for transaction safety:

```
from fastapi import Depends
from sqlalchemy.ext.asyncio import AsyncSession
from contextlib import asynccontextmanager

@asynccontextmanager
async def get_db_transaction():
    async with AsyncSession(engine) as session:
        async with session.begin():
            try:
                yield session
                await session.commit()
            except Exception:
                await session.rollback()
                raise

async def db_dependency():
    async with get_db_transaction() as session:
        yield session
```

Transaction management best practices:

- **Async context managers** ensure proper resource cleanup
- **Automatic rollback** on any exception
- **Connection pooling** through SQLAlchemy engine
- Nested transaction support with savepoints
- Prevents connection leaks and deadlocks

7. Design a FastAPI response caching system using Redis with cache invalidation strategies (TTL, tag-based), conditional requests (ETag), and cache warming for frequently accessed endpoints.

Intelligent Response Caching

This implements **multi-layered caching** with ETags and invalidation:

```
from fastapi import Request, Response
import hashlib
import json

async def cache_response(key: str, data: dict, ttl: int = 300):
    etag = hashlib.md5(json.dumps(data).encode()).hexdigest()
    await redis.setex(f"cache:{key}", ttl, json.dumps({"data": data, "etag": etag}))
    return etag

async def get_cached(request: Request, key: str):
    cached = await redis.get(f"cache:{key}")
    if cached:
        cached_data = json.loads(cached)
        if request.headers.get("If-None-Match") == cached_data["etag"]:
            return Response(status_code=304)
        return cached_data
    return None
```

Caching strategies implemented:

- **ETag generation** for conditional GET requests
- **TTL-based expiration** for automatic cache invalidation
- **304 Not Modified** responses to reduce bandwidth
- JSON serialization for complex object caching
- Foundation for tag-based invalidation patterns

8. Implement a FastAPI streaming response endpoint that processes large CSV files chunk-by-chunk, applies transformations, and streams results without loading entire file into memory.

Memory-Efficient Streaming

This demonstrates **generator-based streaming** for large file processing:

```
from fastapi.responses import StreamingResponse
import csv
import io

async def process_csv_stream(file_path: str):
    async with aiofiles.open(file_path, mode='r') as f:
        async for line in f:
            row = line.strip().split(',')
            transformed = ','.join([cell.upper() for cell in row])
            yield f"{transformed}\n"

@app.get("/stream-csv")
async def stream_csv():
    return StreamingResponse(process_csv_stream("data.csv"), media_type="text/csv")
```

Streaming optimization techniques:

- **Async generators** for non-blocking I/O operations
- **Chunk-by-chunk processing** prevents memory overflow
- **StreamingResponse** for incremental data delivery
- Transformations applied in pipeline fashion
- Suitable for GB-sized files with constant memory usage

9. Create a FastAPI service that implements circuit breaker pattern for external API calls, with configurable failure thresholds, timeout handling, and automatic recovery with exponential backoff.

Circuit Breaker Pattern

This implements **resilient external service integration** with failure protection:

```
from datetime import datetime, timedelta
import httpx

class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=30):
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.last_failure_time = None
        self.state = "CLOSED"

    async def call(self, url: str):
        if self.state == "OPEN" and datetime.now() - self.last_failure_time < timedelta(seconds=self.timeout):
            raise Exception("Circuit breaker is OPEN")
        try:
            async with httpx.AsyncClient(timeout=5.0) as client:
                response = await client.get(url)
                self.failure_count = 0
                self.state = "CLOSED"
                return response
        except Exception:
            self.failure_count += 1
            if self.failure_count >= self.failure_threshold:
                self.state = "OPEN"
                self.last_failure_time = datetime.now()
            raise
```

Resilience patterns:

- **State machine** with CLOSED, OPEN, and HALF-OPEN states
- **Failure threshold** prevents cascading failures
- **Timeout handling** for automatic recovery attempts
- Protects downstream services from overload
- Foundation for exponential backoff implementation

10. Implement a FastAPI endpoint with multi-part file upload validation that checks file types, scans for malware signatures, enforces size limits, and generates secure storage paths with collision prevention.

Secure File Upload Handling

This demonstrates **comprehensive upload security** with validation layers:

```
from fastapi import UploadFile, HTTPException
import magic
import uuid
from pathlib import Path

ALLOWED_TYPES = {'image/jpeg', 'image/png', 'application/pdf'}
MAX_SIZE = 10 * 1024 * 1024  # 10MB

async def validate_upload(file: UploadFile):
    content = await file.read()
    if len(content) > MAX_SIZE:
        raise HTTPException(400, "File too large")
    mime = magic.from_buffer(content, mime=True)
    if mime not in ALLOWED_TYPES:
        raise HTTPException(400, "Invalid file type")
    secure_name = f"{uuid.uuid4()}{Path(file.filename).suffix}"
    return secure_name, content
```

Security measures implemented:

- **Magic number validation** prevents MIME type spoofing
- **Size limits** protect against DoS attacks
- **UUID-based naming** prevents path traversal and collisions
- Content inspection before storage
- Foundation for antivirus scanning integration

Data Structures and Algorithms

Questions in this section test your understanding of how to work with and manipulate data efficiently.

1. Implement an LRU (Least Recently Used) cache in FastAPI with GET and PUT endpoints. What's the time complexity?

LRU Cache Implementation

Use **OrderedDict** or a combination of dictionary and doubly linked list for O(1) operations:

```
from collections import OrderedDict
from fastapi import FastAPI

app = FastAPI()
cache = OrderedDict()
CAPACITY = 100

@app.get('/cache/{key}')
def get_cache(key: str):
    if key in cache:
        cache.move_to_end(key)
        return cache[key]
    return None

@app.put('/cache/{key}')
def put_cache(key: str, value: str):
    if key in cache:
        cache.move_to_end(key)
    cache[key] = value
    if len(cache) > CAPACITY:
        cache.popitem(last=False)
    return {'status': 'success'}
```

Time Complexity: O(1) for both GET and PUT operations. **Space Complexity:** O(capacity).

2. Create a FastAPI endpoint that finds all pairs in an array that sum to a target value. Optimize for large datasets.

Two Sum - All Pairs Solution

Use a **hash set** approach for O(n) time complexity:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class ArrayInput(BaseModel):
    numbers: list[int]
    target: int

@app.post('/find-pairs')
def find_pairs(data: ArrayInput):
    seen = set()
    pairs = []
    for num in data.numbers:
        complement = data.target - num
        if complement in seen:
            pairs.append([complement, num])
        seen.add(num)
    return {'pairs': pairs}
```

Time Complexity: O(n), **Space Complexity:** O(n). For large datasets, consider pagination or streaming responses.

3. Implement a sliding window maximum endpoint that returns the maximum value in each window of size k from an array.

Sliding Window Maximum

Use a **deque (double-ended queue)** to maintain indices of potential maximums:

```
from fastapi import FastAPI
from pydantic import BaseModel
from collections import deque

app = FastAPI()

class WindowInput(BaseModel):
    numbers: list[int]
    k: int

@app.post('/sliding-window-max')
def sliding_max(data: WindowInput):
    dq = deque()
    result = []
    for i, num in enumerate(data.numbers):
        while dq and dq[0] < i - data.k + 1:
            dq.popleft()
        while dq and data.numbers[dq[-1]] < num:
            dq.pop()
        dq.append(i)
        if i >= data.k - 1:
            result.append(data.numbers[dq[0]])
    return {'maximums': result}
```

Time Complexity: O(n), **Space Complexity:** O(k). Each element is added and removed at most once.

4. Design a FastAPI endpoint that implements a min-stack supporting push, pop, and getMin operations in O(1) time.

Min Stack Implementation

Maintain two stacks: one for values and one for tracking minimums:

```
from fastapi import FastAPI

app = FastAPI()
stack = []
min_stack = []

@app.post('/stack/push/{value}')
def push(value: int):
    stack.append(value)
    min_val = min(value, min_stack[-1] if min_stack else value)
    min_stack.append(min_val)
    return {'status': 'pushed'}

@app.delete('/stack/pop')
def pop():
    if stack:
        stack.pop()
        min_stack.pop()
    return {'status': 'popped'}

@app.get('/stack/min')
def get_min():
    return {'min': min_stack[-1] if min_stack else None}
```

All operations are O(1). Space complexity is O(n) for storing both stacks.

5. Create an endpoint that detects if a linked list has a cycle and returns the node where the cycle begins.

Cycle Detection with Floyd's Algorithm

Use **two pointers (slow and fast)** to detect cycles, then find the cycle start:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class Node(BaseModel):
    value: int
    next_index: int | None

class ListInput(BaseModel):
    nodes: list[Node]

@app.post('/detect-cycle')
def detect_cycle(data: ListInput):
    if not data.nodes:
        return {'has_cycle': False}
    slow = fast = 0
    while fast is not None and data.nodes[fast].next_index is not None:
        slow = data.nodes[slow].next_index
        fast = data.nodes[data.nodes[fast].next_index].next_index if data.nodes[fast].next_index else None
        if slow == fast:
            start = 0
            while start != slow:
                start = data.nodes[start].next_index
                slow = data.nodes[slow].next_index
            return {'has_cycle': True, 'cycle_start': start}
    return {'has_cycle': False}
```

Time Complexity: O(n), **Space Complexity:** O(1).

6. Implement a trie (prefix tree) with FastAPI endpoints for insert, search, and prefix matching operations.

Trie Data Structure

Use nested dictionaries to represent the trie structure:

```
from fastapi import FastAPI

app = FastAPI()
trie = {}

@app.post('/trie/insert/{word}')
def insert_word(word: str):
    node = trie
    for char in word:
        node = node.setdefault(char, {})
    node['*'] = True
    return {'status': 'inserted'}

@app.get('/trie/search/{word}')
def search_word(word: str):
    node = trie
    for char in word:
        if char not in node:
            return {'found': False}
        node = node[char]
    return {'found': '*' in node}

@app.get('/trie/prefix/{prefix}')
def search_prefix(prefix: str):
    node = trie
```

```
    for char in prefix:
        if char not in node:
            return {'exists': False}
        node = node[char]
    return {'exists': True}
```

Time Complexity: O(m) where m is word/prefix length. **Space:** O(total characters).

7. Design an endpoint that merges k sorted arrays into one sorted array efficiently.

Merge K Sorted Arrays

Use a **min-heap** to efficiently track the smallest element across all arrays:

```
from fastapi import FastAPI
from pydantic import BaseModel
import heapq

app = FastAPI()

class ArraysInput(BaseModel):
    arrays: list[list[int]]

@app.post('/merge-sorted-arrays')
def merge_arrays(data: ArraysInput):
    heap = []
    for i, arr in enumerate(data.arrays):
        if arr:
            heapq.heappush(heap, (arr[0], i, 0))
    result = []
    while heap:
        val, arr_idx, elem_idx = heapq.heappop(heap)
        result.append(val)
        if elem_idx + 1 < len(data.arrays[arr_idx]):
            heapq.heappush(heap, (data.arrays[arr_idx][elem_idx + 1], arr_idx, elem_idx + 1))
    return {'merged': result}
```

Time Complexity: O(n log k) where n is total elements and k is number of arrays.

8. Implement a FastAPI endpoint that finds the longest substring without repeating characters.

Longest Substring Without Repeating Characters

Use **sliding window** technique with a hash map to track character positions:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class StringInput(BaseModel):
    text: str

@app.post('/longest-unique-substring')
def longest_unique(data: StringInput):
    char_index = {}
    max_length = start = 0
    result_start = 0
    for i, char in enumerate(data.text):
        if char in char_index and char_index[char] >= start:
            start = char_index[char] + 1
        char_index[char] = i
        if i - start + 1 > max_length:
            max_length = i - start + 1
            result_start = start
    return {'length': max_length, 'substring': data.text[result_start:result_start + max_length]}
```

Time Complexity: O(n), **Space Complexity:** O(min(n, m)) where m is character set size.

9. Create an endpoint that implements binary search on a rotated sorted array to find a target element.

Search in Rotated Sorted Array

Modified **binary search** that identifies which half is sorted:

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI()

class SearchInput(BaseModel):
    numbers: list[int]
    target: int

@app.post('/search-rotated')
def search_rotated(data: SearchInput):
    left, right = 0, len(data.numbers) - 1
    while left <= right:
        mid = (left + right) // 2
        if data.numbers[mid] == data.target:
            return {'found': True, 'index': mid}
        if data.numbers[left] <= data.numbers[mid]:
            if data.numbers[left] <= data.target < data.numbers[mid]:
                right = mid - 1
            else:
                left = mid + 1
        else:
            if data.numbers[mid] < data.target <= data.numbers[right]:
                left = mid + 1
            else:
                right = mid - 1
    return {'found': False, 'index': -1}
```

Time Complexity: O(log n), **Space Complexity:** O(1).

10. Design a FastAPI endpoint that implements a priority queue with custom priority values and returns top N elements.

Priority Queue with Top N Elements

Use Python's **heapq** module for efficient priority queue operations:

```
from fastapi import FastAPI
from pydantic import BaseModel
import heapq

app = FastAPI()
priority_queue = []

class Task(BaseModel):
    priority: int
    task_id: str
    description: str

@app.post('/queue/add')
def add_task(task: Task):
    heapq.heappush(priority_queue, (-task.priority, task.task_id, task.description))
    return {'status': 'added'}

@app.get('/queue/top/{n}')
def get_top_n(n: int):
    top_items = heapq.nsmallest(n, priority_queue)
    return {'top_tasks': [{'priority': -p, 'id': tid, 'desc': d} for p, tid, d in top_items]}
```

Time Complexity: O(log n) for insertion, O(n log k) for top k elements. Use negative priority for max-heap behavior.

System Design

These questions evaluate your ability to think about the bigger picture, including architecture, scalability, and performance.

1. Design a scalable URL shortener service using FastAPI. What are the key architectural components and how would you handle high traffic?

Architecture Overview

A scalable URL shortener requires careful consideration of **data storage**, **caching**, and **distributed systems**.

Key Components

- **API Layer (FastAPI)**: Handles POST /shorten and GET /{short_code} endpoints
- **Database**: PostgreSQL or Cassandra for URL mappings with indexed short codes
- **Cache Layer**: Redis for hot URLs (80/20 rule applies)
- **Load Balancer**: Distribute traffic across multiple FastAPI instances
- **ID Generation**: Base62 encoding of auto-increment IDs or distributed ID generator like Snowflake

Sample FastAPI Implementation

```
from fastapi import FastAPI, HTTPException
from redis import Redis
import hashlib

app = FastAPI()
redis_client = Redis(host='localhost')

@app.post('/shorten')
async def shorten_url(url: str):
    short_code = hashlib.md5(url.encode()).hexdigest()[:7]
    redis_client.setex(short_code, 3600, url)
    return {'short_url': short_code}
```

Scalability Considerations

- **Horizontal Scaling**: Stateless FastAPI workers behind load balancer
- **Cache Strategy**: Cache-aside pattern with TTL for popular URLs
- **Database Sharding**: Partition by short_code hash for write distribution
- **Rate Limiting**: Implement token bucket algorithm per IP/user
- **Analytics**: Async message queue (Kafka/RabbitMQ) for click tracking

2. How would you design a real-time notification system with FastAPI supporting WebSockets for millions of concurrent users?

System Architecture

A **real-time notification system** requires WebSocket management, message routing, and horizontal scalability.

Core Components

- **WebSocket Manager**: FastAPI WebSocket endpoints with connection pooling
- **Message Broker**: Redis Pub/Sub or Apache Kafka for message distribution
- **Connection Registry**: Redis hash maps tracking user_id to server_id mappings
- **Presence Service**: Track online/offline status with heartbeat mechanism
- **Notification Queue**: Persistent queue for offline message delivery

FastAPI WebSocket Implementation

```
from fastapi import FastAPI, WebSocket
from redis import Redis

app = FastAPI()
redis = Redis()

@app.websocket('/ws/{user_id}')
async def websocket_endpoint(websocket: WebSocket, user_id: str):
    await websocket.accept()
    pubsub = redis.pubsub()
    pubsub.subscribe(f'user:{user_id}')
    async for message in pubsub.listen():
        await websocket.send_json(message)
```

Scalability Strategy

- **Connection Distribution**: Use consistent hashing to route users to specific servers
- **Server Communication**: Redis Pub/Sub allows cross-server message delivery
- **Sticky Sessions**: Load balancer maintains user-to-server affinity
- **Graceful Degradation**: Fallback to long-polling if WebSocket fails
- **Backpressure Handling**: Implement message buffering with max queue size

CAP Theorem Considerations

This system prioritizes **Availability and Partition Tolerance (AP)** . Notifications may be delivered out of order during network partitions, but the system remains operational.

3. Design a rate-limiting middleware for FastAPI that works across distributed instances. Discuss algorithms and implementation strategies.

Rate Limiting Algorithms

For distributed systems, we need **centralized state management** to track request counts across instances.

Algorithm Options

- **Token Bucket**: Allows burst traffic, refills at constant rate
- **Sliding Window Log**: Accurate but memory-intensive
- **Sliding Window Counter**: Balance between accuracy and performance
- **Fixed Window Counter**: Simple but allows double traffic at boundaries

Redis-Based Sliding Window Implementation

```
from fastapi import FastAPI, Request, HTTPException
from redis import Redis
import time

redis_client = Redis()

async def rate_limit(request: Request, limit=100, window=60):
    key = f'rate:{request.client.host}'
    now = time.time()
    redis_client.zremrangebyscore(key, 0, now - window)
    count = redis_client.zcard(key)
    if count >= limit:
        raise HTTPException(429, 'Rate limit exceeded')
```

Distributed Considerations

- **Centralized Store**: Redis cluster for shared rate limit state
- **Atomic Operations**: Use Redis Lua scripts for race condition prevention
- **Granularity**: Support per-user, per-IP, and per-endpoint limits
- **Headers**: Return X-RateLimit-Remaining and X-RateLimit-Reset
- **Bypass Mechanism**: Whitelist for trusted services

Performance Optimization

- **Local Cache**: Cache rate limit checks for 1-2 seconds to reduce Redis load
- **Batch Operations**: Use Redis pipelining for multiple checks
- **Async Processing**: Non-blocking Redis operations with aioredis

4. Design a distributed task queue system using FastAPI and Celery. How would you handle task prioritization, retries, and monitoring?

System Architecture

A robust **distributed task queue** requires message brokers, worker management, and result storage.

Core Components

- **FastAPI API Layer**: Receives task submission requests
- **Message Broker**: RabbitMQ or Redis for task queue management
- **Celery Workers**: Distributed workers processing tasks
- **Result Backend**: Redis or PostgreSQL for task status/results
- **Monitoring**: Flower dashboard for real-time worker monitoring

FastAPI + Celery Integration

```
from fastapi import FastAPI
from celery import Celery

celery_app = Celery('tasks', broker='redis://localhost')
app = FastAPI()

@celery_app.task(bind=True, max_retries=3)
def process_data(self, data):
    try:
        return heavy_computation(data)
    except Exception as exc:
        raise self.retry(exc=exc, countdown=60)
```

Task Prioritization Strategy

- **Multiple Queues**: Separate queues for high/medium/low priority tasks
- **Worker Routing**: Dedicate workers to specific priority queues
- **Queue Weights**: Configure workers to poll high-priority queues more frequently
- **Dynamic Priority**: Use task metadata to adjust priority based on business logic

Retry Mechanism

- **Exponential Backoff**: Increase delay between retries (1s, 2s, 4s, 8s)
- **Max Retries**: Set reasonable limits to prevent infinite loops
- **Dead Letter Queue**: Move failed tasks after max retries for manual inspection
- **Idempotency**: Design tasks to be safely retried without side effects

Monitoring and Observability

- **Task Metrics**: Track success rate, duration, queue length
- **Worker Health**: Monitor CPU, memory, active task count
- **Alerting**: Set up alerts for queue backlog and failure rates

5. How would you implement a multi-tenant SaaS application with FastAPI ensuring data isolation and performance?

Multi-Tenancy Strategies

Choosing the right **data isolation model** depends on security requirements, scale, and cost considerations.

Isolation Approaches

- **Separate Databases**: Highest isolation, complex management, best for enterprise

- **Shared Database, Separate Schemas**: Good isolation, moderate complexity
- **Shared Schema with Tenant Column**: Easiest to implement, requires careful query filtering

FastAPI Tenant Context Implementation

```
from fastapi import FastAPI, Depends, Header
from sqlalchemy.orm import Session

def get_tenant_id(x_tenant_id: str = Header()):
    return x_tenant_id

@app.get('/data')
async def get_data(tenant_id: str = Depends(get_tenant_id), db: Session = Depends(get_db)):
    return db.query(Data).filter(Data.tenant_id == tenant_id).all()
```

Security Considerations

- **Tenant Validation**: Verify tenant ID from JWT token, not just headers
- **Query Filtering**: Use SQLAlchemy filters or Row-Level Security (RLS) in PostgreSQL
- **Connection Pooling**: Separate pools per tenant or dynamic schema switching
- **API Rate Limiting**: Per-tenant quotas to prevent noisy neighbor problem

Performance Optimization

- **Caching Strategy**: Namespace cache keys by tenant_id
- **Database Indexing**: Composite indexes on (tenant_id, other_columns)
- **Query Optimization**: Ensure tenant_id is always in WHERE clause
- **Connection Management**: Use pgbouncer for connection pooling at scale

Scalability Patterns

- **Tenant Sharding**: Distribute large tenants to dedicated database instances
- **Read Replicas**: Route read queries to replicas, writes to primary
- **Microservices**: Separate services for tenant management vs business logic

6. Design a content delivery and caching strategy for a FastAPI-based API serving media files. How would you handle cache invalidation?

Caching Architecture

An effective **CDN and caching strategy** reduces latency and backend load for media-heavy applications.

Multi-Layer Caching

- **CDN Layer**: CloudFront/Cloudflare for edge caching (TTL: hours to days)
- **Application Cache**: Redis for API responses and metadata (TTL: minutes)
- **Database Query Cache**: PostgreSQL query results (TTL: seconds)
- **Browser Cache**: Cache-Control headers for client-side caching

FastAPI with Cache Headers

```
from fastapi import FastAPI, Response
from datetime import datetime, timedelta

@app.get('/media/{file_id}')
async def get_media(file_id: str, response: Response):
    file_url = get_file_url(file_id)
    expires = datetime.utcnow() + timedelta(hours=24)
    response.headers['Cache-Control'] = 'public, max-age=86400'
    response.headers['Expires'] = expires.strftime('%a, %d %b %Y %H:%M:%S GMT')
    return {'url': file_url}
```

Cache Invalidation Strategies

- **Time-Based (TTL)**: Simple but may serve stale data
- **Event-Based**: Invalidate on update/delete operations
- **Versioned URLs**: Include version/hash in URL (e.g., /media/file_id?v=abc123)

- **Purge API**: CDN purge requests for immediate invalidation
- **Cache Tags**: Group related content for bulk invalidation

Implementation Pattern

```
from redis import Redis

redis = Redis()

async def invalidate_cache(pattern: str):
    keys = redis.keys(pattern)
    if keys:
        redis.delete(*keys)

@app.put('/media/{file_id}')
async def update_media(file_id: str, data: dict):
    update_database(file_id, data)
    await invalidate_cache(f'media:{file_id}*')
    return {'status': 'updated'}
```

CDN Integration

- **Signed URLs**: Generate time-limited URLs for private content
- **Origin Shield**: Additional caching layer between CDN and origin
- **Geo-Routing**: Route requests to nearest regional origin
- **Compression**: Enable Brotli/Gzip at CDN edge

7. Design an event-driven microservices architecture using FastAPI. How would you ensure consistency and handle distributed transactions?

Event-Driven Architecture

An **event-driven system** promotes loose coupling and scalability but requires careful handling of consistency.

Core Components

- **Event Bus**: Kafka or RabbitMQ for reliable message delivery
- **Event Store**: Persistent log of all domain events
- **Service Registry**: Consul or Eureka for service discovery
- **API Gateway**: Single entry point routing to microservices
- **Saga Orchestrator**: Manages distributed transactions

FastAPI Event Publisher

```
from fastapi import FastAPI
from kafka import KafkaProducer
import json

app = FastAPI()
producer = KafkaProducer(bootstrap_servers='localhost:9092')

@app.post('/orders')
async def create_order(order: dict):
    event = {'type': 'OrderCreated', 'data': order}
    producer.send('orders', json.dumps(event).encode())
    return {'status': 'processing'}
```

Consistency Patterns

- **Eventual Consistency**: Accept temporary inconsistency, propagate changes via events
- **Saga Pattern**: Sequence of local transactions with compensating actions
- **Event Sourcing**: Store state changes as events, rebuild state by replaying
- **CQRS**: Separate read and write models for optimized queries

Saga Implementation Strategy

- **Choreography**: Services listen to events and react (decentralized)

- **Orchestration**: Central coordinator manages saga workflow (centralized)
- **Compensation**: Define rollback logic for each step
- **Idempotency**: Ensure handlers can process same event multiple times safely

CAP Theorem Trade-offs

Event-driven systems typically favor **Availability and Partition Tolerance (AP)**. Use eventual consistency with conflict resolution strategies like last-write-wins or custom merge logic.

Observability

- **Distributed Tracing**: OpenTelemetry for request flow across services
- **Event Monitoring**: Track event processing lag and failure rates
- **Correlation IDs**: Propagate request IDs through event chain

8. How would you design a search and filtering system for FastAPI with full-text search, faceted filtering, and real-time indexing?

Search Architecture

A robust **search system** requires specialized search engines optimized for full-text queries and aggregations.

Technology Stack

- **Search Engine**: Elasticsearch or Meilisearch for full-text search
- **Primary Database**: PostgreSQL for transactional data
- **Sync Mechanism**: CDC (Change Data Capture) or event-driven indexing
- **Cache Layer**: Redis for popular search queries

FastAPI with Elasticsearch

```
from fastapi import FastAPI, Query
from elasticsearch import AsyncElasticsearch

app = FastAPI()
es = AsyncElasticsearch(['http://localhost:9200'])

@app.get('/search')
async def search(q: str, category: str = None, page: int = 1):
    body = {'query': {'bool': {'must': [{'match': {'title': q}}]}}}
    if category:
        body['query']['bool']['filter'] = [{'term': {'category': category}}]
    return await es.search(index='products', body=body)
```

Indexing Strategies

- **Bulk Indexing**: Initial load using bulk API for performance
- **Real-time Updates**: Index on write using async tasks
- **Debezium CDC**: Stream database changes to Kafka, consume for indexing
- **Periodic Reindexing**: Full reindex scheduled during low traffic

Faceted Search Implementation

- **Aggregations**: Use Elasticsearch aggregations for facet counts
- **Dynamic Facets**: Generate facets based on query results
- **Multi-Select Filters**: Support OR logic within facets, AND across facets
- **Range Facets**: Price ranges, date ranges with histogram aggregations

Performance Optimization

- **Query Caching**: Cache popular searches in Redis with short TTL
- **Index Optimization**: Use appropriate analyzers and field types
- **Pagination**: Use search_after for deep pagination instead of from/size
- **Sharding**: Distribute index across shards based on data volume

Relevance Tuning

- **Boosting**: Increase weight of title matches vs description
- **Synonyms**: Configure synonym filters for better recall
- **Fuzzy Matching**: Handle typos with fuzziness parameter

9. Design a file upload and processing pipeline with FastAPI handling large files, virus scanning, and asynchronous processing.

Upload Pipeline Architecture

Handling **large file uploads** requires streaming, chunking, and background processing to avoid blocking the API.

System Components

- **Upload Endpoint**: FastAPI with streaming support
- **Object Storage**: S3 or MinIO for file storage
- **Task Queue**: Celery for async processing tasks
- **Virus Scanner**: ClamAV integration for security
- **Notification Service**: Inform users of processing completion

FastAPI Chunked Upload

```
from fastapi import FastAPI, UploadFile, BackgroundTasks
import boto3

app = FastAPI()
s3 = boto3.client('s3')

@app.post('/upload')
async def upload_file(file: UploadFile, background_tasks: BackgroundTasks):
    key = f'uploads/{file.filename}'
    await s3.upload_fileobj(file.file, 'bucket', key)
    background_tasks.add_task(process_file, key)
    return {'status': 'uploaded', 'key': key}
```

Processing Pipeline

- **Step 1**: Validate file type and size limits
- **Step 2**: Stream upload to object storage
- **Step 3**: Queue virus scanning task
- **Step 4**: If clean, queue processing tasks (thumbnail, compression, etc.)
- **Step 5**: Update database with file metadata and status
- **Step 6**: Send notification to user

Large File Handling

- **Multipart Upload**: Use S3 multipart for files >5GB
- **Resumable Uploads**: Implement tus protocol for resume capability
- **Direct Upload**: Generate pre-signed URLs for client-to-S3 upload
- **Streaming**: Use async generators to avoid loading entire file in memory

Security Measures

- **Virus Scanning**: Scan files before making them publicly accessible
- **Content-Type Validation**: Verify MIME type matches file extension
- **File Size Limits**: Enforce per-user and per-request limits
- **Quarantine**: Isolate suspicious files for manual review

Scalability Considerations

- **Worker Pools**: Separate workers for CPU-intensive vs I/O tasks
- **CDN Integration**: Serve processed files via CDN
- **Cleanup Jobs**: Scheduled tasks to remove temporary files

10. Design an API versioning strategy for FastAPI that supports multiple versions simultaneously while minimizing code duplication.

Versioning Approaches

Choosing the right **API versioning strategy** balances backward compatibility with maintainability.

Versioning Methods

- **URI Versioning**: /v1/users, /v2/users (most common, explicit)
- **Header Versioning**: Accept: application/vnd.api.v1+json (cleaner URLs)
- **Query Parameter**: /users?version=1 (less common, easy to implement)
- **Content Negotiation**: Use Accept header with custom media types

FastAPI URI Versioning Implementation

```
from fastapi import FastAPI, APIRouter

app = FastAPI()
v1_router = APIRouter(prefix='/v1')
v2_router = APIRouter(prefix='/v2')

@v1_router.get('/users')
async def get_users_v1():
    return {'version': 1, 'users': []}

@v2_router.get('/users')
async def get_users_v2():
    return {'version': 2, 'data': {'users': []}}
```

Code Reusability Strategies

- **Shared Business Logic**: Extract core logic into service layer
- **Version-Specific Serializers**: Use Pydantic models per version
- **Decorator Pattern**: Wrap shared handlers with version-specific transformations
- **Feature Flags**: Toggle features based on version

Migration Strategy

- **Deprecation Warnings**: Include headers like X-API-Deprecated: true
- **Sunset Headers**: Specify end-of-life date (Sunset: Sat, 31 Dec 2024 23:59:59 GMT)
- **Version Support Policy**: Maintain N-1 or N-2 versions
- **Documentation**: Clearly document changes and migration guides

Advanced Patterns

```
from fastapi import Header, HTTPException

async def get_api_version(accept: str = Header('application/json')):
    if 'v2' in accept:
        return 2
    elif 'v1' in accept:
        return 1
    raise HTTPException(400, 'Invalid API version')
```

Testing Considerations

- **Version-Specific Tests**: Separate test suites per version
- **Contract Testing**: Ensure backward compatibility with Pact
- **Automated Checks**: Detect breaking changes in CI/CD

Coding and Debugging

This section presents practical coding challenges and questions about debugging techniques.

1. Write a FastAPI endpoint that accepts a nested list and returns it flattened. How would you handle type validation?

Flattening Nested Lists in FastAPI

Here's a FastAPI endpoint that flattens a nested list with proper type validation:

```
from fastapi import FastAPI
from pydantic import BaseModel
from typing import List, Any

app = FastAPI()

class NestedList(BaseModel):
    data: List[Any]

@app.post("/flatten")
def flatten_list(payload: NestedList):
    def flatten(lst):
        result = []
        for item in lst:
            if isinstance(item, list):
                result.extend(flatten(item))
            else:
                result.append(item)
        return result
    return {"flattened": flatten(payload.data)}
```

Key points:

- Uses Pydantic's **BaseModel** for automatic request validation
- Recursive function handles arbitrary nesting depth
- Type hint **List[Any]** allows mixed-type nested structures
- Returns JSON-serializable response automatically

2. Create a FastAPI middleware that logs the execution time of each request and identify performance bottlenecks. How would you debug slow endpoints?

Request Timing Middleware

```
import time
from fastapi import FastAPI, Request

app = FastAPI()

@app.middleware("http")
async def log_timing(request: Request, call_next):
    start = time.time()
    response = await call_next(request)
    duration = time.time() - start
    response.headers["X-Process-Time"] = str(duration)
    print(f"{request.method} {request.url.path}: {duration:.4f}s")
    return response
```

Debugging slow endpoints:

- **Use cProfile:** Wrap endpoint logic with cProfile to identify bottleneck functions
- **Database query profiling:** Enable SQLAlchemy echo or use database-specific slow query logs

- **Async profiling:** Use py-spy or Austin for production profiling without stopping the app
- **APM tools:** Integrate New Relic, DataDog, or Sentry for distributed tracing
- **Memory profiling:** Use memory_profiler or tracemalloc for memory leak detection

3. Implement a FastAPI endpoint that checks if a string is a palindrome, handling Unicode characters correctly. What edge cases should you consider?

Unicode-Aware Palindrome Checker

```
from fastapi import FastAPI
from pydantic import BaseModel, Field
import unicodedata

app = FastAPI()

class TextInput(BaseModel):
    text: str = Field(..., min_length=1)

@app.post("/palindrome")
def check_palindrome(payload: TextInput):
    normalized = unicodedata.normalize('NFKD', payload.text.lower())
    cleaned = ''.join(c for c in normalized if c.isalnum())
    is_palindrome = cleaned == cleaned[::-1]
    return {"is_palindrome": is_palindrome, "processed": cleaned}
```

Edge cases to consider:

- **Unicode normalization:** 'é' vs 'e' with combining accent (NFKD normalization)
- **Case sensitivity:** 'Racecar' should match 'racecar'
- **Whitespace and punctuation:** 'A man, a plan, a canal: Panama'
- **Empty strings:** Should return true or handle as invalid input
- **Emojis and special characters:** Decide if they should be included or stripped
- **RTL languages:** Arabic/Hebrew text direction considerations

4. How would you implement custom exception handlers in FastAPI to return consistent error responses? Show an example with validation errors.

Custom Exception Handlers

```
from fastapi import FastAPI, Request, status
from fastapi.exceptions import RequestValidationError
from fastapi.responses import JSONResponse

app = FastAPI()

@app.exception_handler(RequestValidationError)
async def validation_exception_handler(request: Request, exc: RequestValidationError):
    errors = [{"field": e["loc"][-1], "message": e["msg"]} for e in exc.errors()]
    return JSONResponse(
        status_code=status.HTTP_422_UNPROCESSABLE_ENTITY,
        content={"success": False, "errors": errors}
    )
```

Best practices for exception handling:

- **Consistent structure:** Always return same JSON schema (success, data/errors, message)
- **Custom exceptions:** Create domain-specific exceptions (UserNotFound, InvalidCredentials)
- **Status codes:** Use appropriate HTTP status codes (400, 401, 403, 404, 422, 500)
- **Error logging:** Log exceptions with context (user_id, request_id) before returning response
- **Security:** Never expose stack traces or internal details in production
- **Internationalization:** Support multiple languages for error messages

5. Write a FastAPI dependency that implements request rate limiting using Redis. How would you test this dependency?

Redis-Based Rate Limiting Dependency

```
from fastapi import FastAPI, Depends, HTTPException, Request
import redis.asyncio as redis
```

```
app = FastAPI()
redis_client = redis.Redis(host='localhost', decode_responses=True)

async def rate_limit(request: Request, limit: int = 10):
    key = f"rate:{request.client.host}:{request.url.path}"
    count = await redis_client.incr(key)
    if count == 1:
        await redis_client.expire(key, 60)
    if count > limit:
        raise HTTPException(429, "Rate limit exceeded")

@app.get("/api/data", dependencies=[Depends(rate_limit)])
async def get_data():
    return {"data": "success"}
```

Testing strategies:

- **Mock Redis:** Use fakeredis or pytest-mock to mock redis_client
- **Integration tests:** Use TestClient with real Redis (docker-compose for CI/CD)
- **Parametrized tests:** Test different rate limits and time windows
- **Concurrent requests:** Use asyncio.gather to simulate simultaneous requests
- **Edge cases:** Test Redis connection failures, expired keys, race conditions

6. Demonstrate how to use Python's memory profiler to identify memory leaks in a FastAPI application. What tools would you use in production?

Memory Profiling in FastAPI

Development profiling with memory_profiler:

```
from memory_profiler import profile
from fastapi import FastAPI

app = FastAPI()

@profile
@app.get("/memory-test")
async def memory_intensive():
    data = [i for i in range(1000000)]
    result = sum(data)
    return {"result": result}

# Run with: python -m memory_profiler app.py
```

Using tracemalloc (built-in):

```
import tracemalloc
tracemalloc.start()

@app.get("/snapshot")
async def memory_snapshot():
    snapshot = tracemalloc.take_snapshot()
    top_stats = snapshot.statistics('lineno')[:5]
    return {"top_memory": [str(stat) for stat in top_stats]}
```

Production tools:

- **py-spy:** Sampling profiler that doesn't require code changes
- **Memray:** Memory profiler from Bloomberg, tracks allocations
- **Prometheus + Grafana:** Monitor process memory over time
- **objgraph:** Visualize object references to find circular references
- **guppy3:** Heap analysis and memory consumption tracking

7. Implement monkey patching in FastAPI to modify third-party library behavior. When is this appropriate and what are the risks?

Monkey Patching Example

```
from fastapi import FastAPI
import httpx

app = FastAPI()

# Original method backup
original_request = httpx.AsyncClient.request

async def patched_request(self, method, url, **kwargs):
    print(f"Intercepted: {method} {url}")
    kwargs.setdefault('timeout', 5.0)
    return await original_request(self, method, url, **kwargs)

httpx.AsyncClient.request = patched_request

@app.get("/external")
async def call_external():
    async with httpx.AsyncClient() as client:
        response = await client.get("https://api.example.com")
        return response.json()
```

When to use monkey patching:

- **Testing:** Mock external dependencies without dependency injection
- **Hot fixes:** Temporary fixes for third-party bugs until official patch
- **Adding logging/metrics:** Instrument libraries that don't support it natively
- **Compatibility layers:** Bridge API differences between library versions

Risks and alternatives:

- **Fragility:** Breaks when library internals change - prefer composition/inheritance
- **Debugging difficulty:** Hard to trace modified behavior
- **Testing issues:** Can cause test pollution if not properly isolated
- **Better alternatives:** Use dependency injection, wrapper classes, or contribute to upstream

8. Create a FastAPI endpoint that reverses a string efficiently while preserving Unicode grapheme clusters (e.g., emojis with modifiers). How does this differ from simple string reversal?

Unicode-Aware String Reversal

```
from fastapi import FastAPI
from pydantic import BaseModel
import regex

app = FastAPI()

class TextPayload(BaseModel):
    text: str

@app.post("/reverse")
def reverse_string(payload: TextPayload):
    graphemes = regex.findall(r'\X', payload.text)
    reversed_text = ''.join(reversed(graphemes))
    simple_reverse = payload.text[::-1]
    return {"unicode_aware": reversed_text, "simple": simple_reverse}
```

Why grapheme clusters matter:

- **Emojis with modifiers:** '□□□□' is one grapheme but multiple codepoints
- **Combining characters:** 'é' can be 'e' + combining acute accent
- **Regional indicators:** Flag emojis like '□□' are two codepoints
- **Skin tone modifiers:** '□□' should stay together when reversed

Performance considerations:

- Simple reversal [::-1] is O(n) but breaks Unicode
- Grapheme-aware reversal using regex library is O(n) but slower
- For ASCII-only text, use simple reversal for better performance
- Consider caching grapheme boundaries for repeated operations

9. How would you debug a FastAPI application that's experiencing intermittent 500 errors in production? Walk through your debugging process.

Systematic Debugging Approach

Step 1: Implement comprehensive logging

```
import logging
from fastapi import FastAPI, Request
import uuid

app = FastAPI()
logger = logging.getLogger(__name__)

@app.middleware("http")
async def log_requests(request: Request, call_next):
    request_id = str(uuid.uuid4())
    logger.info(f"[{request_id}] {request.method} {request.url}")
    try:
        response = await call_next(request)
        return response
    except Exception as e:
        logger.error(f"[{request_id}] Error: {e}", exc_info=True)
        raise
```

Step 2: Add error tracking

- **Integrate Sentry:** Captures stack traces, request context, and user info
- **Structured logging:** Use JSON logs with correlation IDs for distributed tracing
- **Health checks:** Implement /health endpoint monitoring dependencies (DB, Redis, external APIs)

Step 3: Analyze patterns

- Check if errors correlate with specific endpoints, users, or time periods
- Review resource usage (CPU, memory, connections) during error spikes
- Examine database connection pools for exhaustion
- Check for race conditions in async code

Step 4: Reproduce and fix

- Use production logs to recreate exact request conditions
- Add unit tests covering the error scenario
- Implement circuit breakers for external service failures
- Add retries with exponential backoff for transient errors

10. Write a FastAPI background task that processes items from a queue with error handling and retry logic. How would you ensure tasks don't get lost?

Robust Background Task Processing

```
from fastapi import FastAPI, BackgroundTasks
import asyncio
import logging

app = FastAPI()
logger = logging.getLogger(__name__)

async def process_item(item_id: str, retry_count: int = 0):
    max_retries = 3
    try:
        # Simulate processing
        await asyncio.sleep(1)
        if item_id == "fail":
            raise ValueError("Processing failed")
        logger.info(f"Processed {item_id}")
    except Exception as e:
        if retry_count < max_retries:
            await asyncio.sleep(2 ** retry_count)
            await process_item(item_id, retry_count + 1)
```

```
        else:
            logger.error(f"Failed after {max_retries} retries: {e}")

@app.post("/queue/{item_id}")
async def queue_item(item_id: str, bg: BackgroundTasks):
    bg.add_task(process_item, item_id)
    return {"status": "queued"}
```

Ensuring task reliability:

- **Use message queues:** Replace BackgroundTasks with Celery, RQ, or Dramatiq for persistence
- **Dead letter queues:** Move failed tasks to DLQ for manual inspection
- **Idempotency:** Design tasks to be safely retried without side effects
- **Task status tracking:** Store task state in database (pending, processing, completed, failed)
- **Monitoring:** Track queue depth, processing time, and failure rates
- **Graceful shutdown:** Handle SIGTERM to finish in-flight tasks before shutdown