FastAPI Developer

Interview Questions and Answers

Core Concepts

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

1. Explain FastAPI's dependency injection system and how it differs from other frameworks

FastAPI's dependency injection system is based on Python's type hints and Pydantic models. Key features include:

- Automatic dependency resolution based on type annotations
- Support for nested dependencies
- Built-in caching mechanism for performance optimization

Example dependency:

```
async def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

@app.get('/items/')
async def read_items(db: Session = Depends(get_db)):
    return db.query(Item).all()
```

2. How would you implement rate limiting in FastAPI?

Rate limiting in FastAPI can be implemented using dependencies and third-party libraries like slowapi:

```
from fastapi import FastAPI, Depends
from slowapi import Limiter
from slowapi.util import get_remote_address

limiter = Limiter(key_func=get_remote_address)
@app.get('/api/resource')
@limiter.limit('5/minute')
async def rate_limited_endpoint():
    return {'status': 'success'}
```

3. Explain FastAPI's background tasks and when to use them versus async endpoints

Background tasks in FastAPI are useful for operations that:

- Don't need immediate response
- Should continue even if client disconnects
- Are resource-intensive but non-critical

```
from fastapi import BackgroundTasks

@app.post('/send-notification/')
async def send_notification(background_tasks: BackgroundTasks):
    background_tasks.add_task(send_email_notification)
    return {'message': 'Notification scheduled'}
```

4. How do you handle database transactions and rollbacks in FastAPI?

Database transactions in FastAPI are typically handled using SQLAlchemy's session management and context managers:

```
from fastapi import HTTPException

async def create_item(db: Session, item: ItemCreate):
    try:
        db_item = Item(**item.dict())
        db.add(db_item)
        db.commit()
        return db_item
    except Exception:
        db.rollback()
        raise HTTPException(500)
```

5. Explain FastAPI's security and authentication mechanisms

FastAPI provides several security mechanisms:

- OAuth2 with Password (and hashing)
- JWT tokens
- HTTP Basic auth
- API Key authentication

```
from fastapi.security import OAuth2PasswordBearer

oauth2_scheme = OAuth2PasswordBearer(tokenUrl='token')

@app.get('/users/me')
async def read_users_me(token: str = Depends(oauth2_scheme)):
    return decode_token(token)
```

6. How would you implement custom middleware in FastAPI?

Custom middleware in FastAPI can be implemented using the middleware decorator or ASGI middleware:

```
@app.middleware('http')
async def add_process_time_header(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    process_time = time.time() - start_time
    response.headers['X-Process-Time'] = str(process_time)
    return response
```

7. Explain FastAPI's approach to API versioning and how to implement it

API versioning in FastAPI can be implemented through:

- Path versioning (/v1/users)
- Header versioning (X-API-Version)
- Parameter versioning (?version=1)

```
app_v1 = FastAPI(root_path='/v1')
app_v2 = FastAPI(root_path='/v2')

@app_v1.get('/users')
def get_users_v1(): return {'version': 'v1'}

@app_v2.get('/users')
def get_users_v2(): return {'version': 'v2'}
```

8. How do you handle file uploads and streaming responses in FastAPI?

File handling in FastAPI supports both regular uploads and streaming:

```
from fastapi import File, UploadFile
from fastapi.responses import StreamingResponse

@app.post('/uploadfile/')
async def create_upload_file(file: UploadFile = File(...)):
    return {'filename': file.filename}

@app.get('/stream')
def stream_video():
    return StreamingResponse(video_generator())
```

9. Explain FastAPI's testing approach and best practices

Testing in FastAPI typically involves:

- TestClient for HTTP testing
- Pytest for async testing
- Dependency overrides for mocking

```
from fastapi.testclient import TestClient

client = TestClient(app)

def test_read_main():
    response = client.get('/')
    assert response.status_code == 200
    assert response.json() == {'msg': 'Hello'}
```

10. How would you optimize FastAPI application performance?

FastAPI performance optimization strategies include:

- Using async database drivers
- Implementing caching mechanisms
- Response compression
- Database query optimization

```
from fastapi_cache import FastAPICache
from fastapi_cache.decorator import cache

@app.get('/expensive-operation')
@cache(expire=60)
async def get_expensive_data():
    return await compute_expensive_operation()
```

Data Structures and Algorithms

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

1. How would you implement an LRU Cache in Python with FastAPI?

Implementation Approach:

An LRU Cache can be implemented using a combination of dictionary and doubly-linked list for O(1) operations:

```
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity: int):
        self.cache = OrderedDict()
        self.capacity = capacity
    def get(self, key: int) -> int:
        if key not in self.cache: return -1
        self.cache.move_to_end(key)
```

Time Complexity: O(1) for both get and put operations

2. Explain how you would implement rate limiting in FastAPI using a sliding window algorithm

Sliding Window Implementation:

```
from collections import deque
from time import time

class RateLimiter:
    def __init__(self, window_size: int, max_requests: int):
        self.window = deque()
        self.window_size = window_size
        self.max_requests = max_requests
```

Key points:

- Uses deque for efficient sliding window operations
- Time complexity: O(1) for both adding and removing requests
- Space complexity: O(N) where N is max_requests

3. How would you implement an efficient caching mechanism for FastAPI endpoints using a custom data structure?

Custom Cache Implementation:

```
class EndpointCache:
    def __init__(self, max_size: int):
        self.cache = {}
        self.max_size = max_size
        self.access_count = defaultdict(int)
    def get(self, key: str) -> Any:
        self.access_count[key] += 1
```

Features:

- O(1) lookup time
- Frequency-based eviction policy
- Thread-safe operations

4. Explain how you would implement a custom priority queue for background task scheduling in FastAPI

Priority Queue Implementation:

```
from heapq import heappush, heappop

class TaskScheduler:
    def __init__(self):
        self._queue = []
        self._entry_count = 0
    def add_task(self, priority: int, task: Callable):
        heappush(self._queue, (-priority, self._entry_count, task))
```

Characteristics:

- O(log n) insertion and deletion
- Maintains FIFO order for equal priorities
- Thread-safe implementation needed for production

5. How would you implement an efficient trie data structure for route matching in FastAPI?

Trie Implementation:

```
class TrieNode:
    def __init__(self):
        self.children = {}
        self.is_endpoint = False
        self.handler = None
    def insert(self, path: str, handler: Callable):
        node = self
```

Benefits:

- O(m) lookup time where m is path length
- Space-efficient for common prefixes
- Supports dynamic route parameters

6. Implement a thread-safe circular buffer for managing WebSocket connections in FastAPI

Circular Buffer Implementation:

```
from threading import Lock

class WebSocketBuffer:
    def __init__(self, size: int):
        self.buffer = [None] * size
        self.head = self.tail = 0
        self.lock = Lock()
```

Key Features:

- Thread-safe operations with Lock
- O(1) insertion and removal
- Efficient memory usage with fixed size

7. How would you implement an efficient bloom filter for request deduplication in FastAPI?

Bloom Filter Implementation:

```
class BloomFilter:
    def __init__(self, size: int, hash_count: int):
        self.size = size
        self.hash_count = hash_count
        self.bit_array = [0] * size
```

Advantages:

- O(k) insertion and lookup where k is hash_count
- Space-efficient for large datasets
- No false negatives possible
- Configurable false positive rate

8. Implement a custom consistent hashing mechanism for distributed caching in FastAPI

Consistent Hashing Implementation:

```
class ConsistentHashing:
    def __init__(self, nodes: int, replicas: int):
        self.ring = {}
        self.sorted_keys = []
        self._add_nodes(nodes, replicas)
```

Benefits:

- Minimal redistribution on node changes
- O(log n) lookup time
- Even distribution of load
- Scalable for distributed systems

9. How would you implement a custom B-tree index for efficient database query caching in FastAPI?

B-tree Implementation:

```
class BTreeNode:
    def __init__(self, leaf=True):
        self.leaf = leaf
        self.keys = []
        self.child = []
        self.n = 0
```

Advantages:

- O(log n) search, insert, delete operations
- Self-balancing structure
- Efficient for disk-based operations
- Supports range queries efficiently

10. Implement an efficient skip list data structure for maintaining sorted FastAPI route parameters

Skip List Implementation:

```
class SkipListNode:
    def __init__(self, key, level):
        self.key = key
        self.forward = [None] * (level + 1)
    def insert(self, key: Any) -> None:
        update = [None] * self.max_level
```

Characteristics:

- O(log n) average case for search/insert/delete
- Probabilistic alternative to balanced trees
- Simple implementation compared to Red-Black trees

System Design

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

1. How would you design a scalable URL shortener service using FastAPI?

Key Components:

- **API Layer:** FastAPI service handling URL creation/redirection
- **Database:** Primary store (PostgreSQL) for URL mappings
- **Cache Layer:** Redis for frequently accessed URLs
- **ID Generation:** Base62 encoding or distributed sequence

Sample Code:

```
@app.post('/shorten')
async def create_short_url(url: str):
    url_hash = generate_unique_hash()
    await redis.set(f'url:{url_hash}', url, ex=3600)
    await db.urls.insert_one({'hash': url_hash, 'long_url': url})
    return {'short_url': f'domain.com/{url_hash}'}
```

Scale Considerations:

- Load balancing across multiple API instances
- Database sharding for URL distribution
- CDN for global access

2. Design a real-time chat system using FastAPI and WebSockets. How would you handle scaling?

Architecture Components:

- **WebSocket Server:** FastAPI with WebSocket support
- **Message Queue:** Redis Pub/Sub or RabbitMQ
- **State Management:** Redis for user sessions
- **Database:** MongoDB for message persistence

```
@app.websocket('/ws/{room_id}')
async def websocket_endpoint(websocket: WebSocket, room_id: str):
    await manager.connect(websocket, room_id)
    try:
        while True:
            data = await websocket.receive_text()
            await manager.broadcast(room_id, data)
```

Scaling Strategy:

- Multiple WebSocket servers behind load balancer
- Sticky sessions for connection persistence
- Distributed message queue for cross-server communication

3. How would you implement rate limiting in a FastAPI application serving millions of requests?

Implementation Approach:

- **Storage:** Redis for token bucket implementation
- **Middleware:** Custom rate limiting middleware

- **Distribution:** Consistent hashing for multiple Redis instances

```
class RateLimiter:
    async def check_rate_limit(self, key: str) -> bool:
        current = await redis.incr(key)
        if current == 1:
            await redis.expire(key, 60)
        return current <= MAX_REQUESTS
```

Scale Considerations:

- Distributed rate limiting across multiple instances
- Sliding window algorithm for accuracy
- Fallback mechanisms for Redis failures
- Client IP-based and API key-based limiting

4. Design a social media feed system using FastAPI. How would you handle post creation and retrieval efficiently?

Core Components:

- **Post Service:** FastAPI endpoints for CRUD operations
- **Cache Layer:** Redis for feed caching
- **Database:** PostgreSQL with materialized views
- **Queue:** Celery for async processing

```
@app.get('/feed/{user_id}')
async def get_feed(user_id: int, skip: int = 0, limit: int = 20):
    cache_key = f'feed:{user_id}:{skip}:{limit}'
    feed = await redis.get(cache_key)
    if not feed:
        feed = await generate_feed(user_id, skip, limit)
```

Performance Optimizations:

- Feed pre-computation
- Cursor-based pagination
- Content denormalization
- Read replicas for scaling

5. How would you design a distributed task processing system using FastAPI and Celery?

Architecture Components:

- **API Layer:** FastAPI for task submission
- **Queue:** RabbitMQ or Redis
- **Workers:** Celery workers for processing
- **Monitoring:** Flower for task tracking

```
@app.post('/tasks')
async def create_task(task_data: TaskModel):
    task_id = await process_task.delay(task_data.dict())
    return {'task_id': task_id, 'status': 'pending'}
```

Scale Considerations:

- Dynamic worker scaling
- Task prioritization
- Error handling and retries
- Result backend optimization

6. Design a notification system using FastAPI that can handle multiple channels (email, push, SMS). How would you ensure reliability?

System Components:

- **API Gateway:** FastAPI service
- **Queue System:** Apache Kafka for message streaming

- **Channel Services:** Separate microservices per channel
- **State Store:** Redis for notification status

```
@app.post('/notify')
async def send_notification(notification: NotificationModel):
    channels = determine_channels(notification.user_id)
    for channel in channels:
        await kafka_producer.send(f'{channel}_topic', notification)
```

Reliability Features:

- Dead letter queues
- Retry mechanisms with exponential backoff
- Circuit breakers for external services
- Message deduplication

7. How would you implement a search service using FastAPI and Elasticsearch?

Architecture Components:

- **API Layer:** FastAPI endpoints
- **Search Engine:** Elasticsearch cluster
- **Cache:** Redis for frequent queries
- **Queue:** Kafka for indexing pipeline

```
@app.get('/search')
async def search(query: str, filters: dict = None):
    cache_key = f'search:{hash(query + str(filters))}'
    if cached := await redis.get(cache_key):
        return cached
    results = await es.search(index='products', query=query)
```

Optimization Strategies:

- Async indexing
- Query optimization
- Result caching
- Shard allocation

8. Design a file upload service using FastAPI that can handle large files efficiently

System Components:

- **API Layer:** FastAPI with streaming support
- **Storage:** S3 or similar object storage
- **Database:** PostgreSQL for metadata
- **Queue:** Celery for post-processing

```
@app.post('/upload')
async def upload_file(file: UploadFile):
    async with aiofiles.open(temp_path, 'wb') as out_file:
        content = await file.read(1024 * 1024)
        while content:
            await out_file.write(content)
```

Features:

- Chunked upload support
- Resume capability
- Progress tracking
- Async processing pipeline

9. How would you design a caching strategy for a FastAPI application handling high-traffic endpoints?

Caching Layers:

- **Application Cache:** In-memory caching

- **Distributed Cache:** Redis cluster
- **CDN:** Edge caching for static content
- **Browser Cache:** ETags and cache headers

```
@cache(ttl=3600)
async def get_product(product_id: int):
    cache_key = f'product:{product_id}'
    if cached := await redis.get(cache_key):
        return cached
    product = await db.get_product(product_id)
```

Optimization Strategies:

- Cache invalidation patterns
- Write-through caching
- Cache warming
- Stale-while-revalidate

10. Design a real-time analytics system using FastAPI that can process and visualize streaming data

System Components:

- **Ingestion API:** FastAPI endpoints
- **Stream Processing:** Apache Kafka Streams
- **Time-series DB:** InfluxDB or TimescaleDB
- **WebSocket:** Real-time updates

```
@app.websocket('/metrics')
async def metrics_websocket(websocket: WebSocket):
    await websocket.accept()
    while True:
        metrics = await get_latest_metrics()
        await websocket.send_json(metrics)
```

Features:

- Time-window aggregations
- Multiple visualization types
- Data downsampling
- Historical data access

Coding and Debugging

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

1. How would you implement rate limiting in FastAPI?

Rate Limiting Implementation

You can implement rate limiting using FastAPI dependencies and a cache backend like Redis:

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

redis = Redis(host='localhost', port=6379)
RATE_LIMIT = 100  # requests
RATE_TIME = 3600  # seconds

async def rate_limit(client_id: str):
    pipe = redis.pipeline()
    now = datetime.now().timestamp()
    pipe.zremrangebyscore(client_id, 0, now - RATE_TIME)
    pipe.zcard(client_id)
    pipe.zadd(client_id, {now: now})
    results = pipe.execute()

    if results[1] >= RATE_LIMIT:
        raise HTTPException(status_code=429)
```

2. Explain how you would implement custom middleware for logging all requests in FastAPI

Custom Middleware Implementation

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

app = FastAPI()

@app.middleware('http')
async def log_requests(request: Request, call_next):
    start_time = time.time()
    response = await call_next(request)
    duration = time.time() - start_time
    logging.info(f'{request.method} {request.url} {duration:.2f}s')
    return response
```

Key points:

- Middleware runs before and after each request
- Can access request/response objects
- Useful for logging, authentication, CORS

3. How would you handle database migrations in a FastAPI application?

Database Migration Strategy

Using Alembic with FastAPI and SQLAlchemy:

```
from alembic import context
```

```
from sqlalchemy import engine_from_config

def run_migrations():
    config = context.config
    target_metadata = None
    connectable = engine_from_config(
        config.get_section(config.config_ini_section),
        prefix='sqlalchemy.',
        poolclass=pool.NullPool)
    with connectable.connect() as connection:
        context.configure(connection=connection,
                          target_metadata=target_metadata)
```

- Use Alembic for version control of database schema
- Define upgrade and downgrade functions
- Integrate with FastAPI startup events

4. How would you implement websocket authentication in FastAPI?

WebSocket Authentication

```
from fastapi import WebSocket, Depends
from typing import Optional

async def get_token(websocket: WebSocket):
    token = websocket.query_params.get('token')
    if not token:
        await websocket.close(code=4000)
        return None
    return token

@app.websocket('/ws')
async def websocket_endpoint(websocket: WebSocket,
                             token: Optional[str] = Depends(get_token)):
    await websocket.accept()
    while True:
        data = await websocket.receive_text()
```

5. How would you implement background tasks with proper error handling in FastAPI?

Background Task Implementation

```
from fastapi import BackgroundTasks
from functools import partial
import logging

async def handle_error(task_id: str, error: Exception):
    logging.error(f'Task {task_id} failed: {str(error)}')

async def process_task(task_id: str, data: dict):
    try:
        # Process task
        result = await heavy_computation(data)
        await save_result(task_id, result)
    except Exception as e:
        await handle_error(task_id, e)
```

6. Explain how you would implement custom exception handlers in FastAPI

Custom Exception Handling

```
from fastapi import FastAPI, Request
from fastapi.responses import JSONResponse

class CustomException(Exception):
    def __init__(self, name: str):
        self.name = name
```

```
@app.exception_handler(CustomException)
async def custom_exception_handler(request: Request, exc: CustomException):
    return JSONResponse(
        status_code=418,
        content={'message': f'Error for {exc.name}'}
```

7. How would you implement caching in FastAPI using Redis?

Redis Caching Implementation

```
from fastapi import FastAPI
from redis import Redis
import pickle

redis_client = Redis(host='localhost', port=6379)

async def get_cached_data(key: str, ttl: int = 3600):
    cached = redis_client.get(key)
    if cached:
        return pickle.loads(cached)
    data = await fetch_data()
    redis_client.setex(key, ttl, pickle.dumps(data))
    return data
```

8. How would you implement pagination with cursor-based navigation in FastAPI?

Cursor-Based Pagination

```
from fastapi import FastAPI
from typing import Optional

async def get_items(cursor: Optional[str] = None, limit: int = 10):
    query = db.select([items])
    if cursor:
        query = query.where(items.c.id > cursor)
    query = query.limit(limit + 1)
    results = await database.fetch_all(query)

    next_cursor = results[limit].id if len(results) > limit else None
    return {'items': results[:limit], 'next_cursor': next_cursor}
```

9. How would you implement request validation with custom validators in FastAPI?

Custom Request Validation

```
from pydantic import BaseModel, validator
from typing import List

class Item(BaseModel):
    name: str
    tags: List[str]

    @validator('name')
    def name_must_be_valid(cls, v):
        if len(v) < 3:
            raise ValueError('name too short')
        return v.title()
```

Key aspects:

- Use Pydantic models for validation
- Custom validators with decorators
- Pre and post-processing of data

10. How would you implement API versioning in FastAPI?

API Versioning Strategy

```
from fastapi import FastAPI, APIRouter

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

@v1_router.get('/items')
async def get_items_v1():
    return {'version': 'v1', 'items': []}

@v2_router.get('/items')
async def get_items_v2():
    return {'version': 'v2', 'items': []}
```

Behavioral Questions

These questions assess your soft skills, problem-solving approach, and how you work in a team.

1. Tell me about a time when you had to optimize a FastAPI endpoint that was performing poorly.

Situation: At my previous role, we noticed one of our critical API endpoints was taking 3+ seconds to respond and causing timeout issues during peak loads.

Task: I needed to identify the bottleneck and optimize the endpoint to respond within 500ms.

Action: I:

- Implemented query optimization using FastAPI's async features
- Added proper database indexing
- Implemented response caching using Redis
- Used FastAPI's background tasks for non-critical operations

Result: The endpoint response time improved to 200ms on average, eliminating timeout issues and improving overall system stability.

2. Describe a situation where you had to implement complex authentication in a FastAPI application.

Situation: Our team needed to implement multi-tenant authentication with role-based access control for a B2B SaaS platform.

Task: Design and implement a secure authentication system supporting multiple authentication methods and fine-grained permissions.

Action: I:

- Implemented JWT-based authentication with FastAPI's security dependencies
- Created custom middleware for tenant isolation
- Designed role-based decorators for endpoint protection
- Added OAuth2 support for third-party integrations

Result: Successfully deployed a secure authentication system that handled 50+ enterprise clients with different security requirements.

3. Share an experience where you had to handle API versioning in FastAPI.

Situation: Our API was evolving rapidly, and we needed to maintain backward compatibility for existing clients.

Task: Implement a versioning strategy that would allow us to evolve the API without breaking existing integrations.

Action: I:

- Implemented URL-based versioning using FastAPI's router prefixes
- Created version-specific Pydantic models
- Set up automated testing for all API versions
- Documented migration paths for clients

Result: Successfully maintained multiple API versions with 100% backward compatibility while allowing new features to be added.

4. Tell me about a time when you had to debug a complex production issue in a FastAPI application.

Situation: Users reported intermittent 503 errors during high-traffic periods.

Task: Identify and resolve the root cause of the service disruptions.

Action: I:

- Added detailed logging using FastAPI's logging middleware
- Implemented distributed tracing with OpenTelemetry
- Analyzed connection pooling configurations
- Discovered and fixed a connection leak in async database calls

Result: Eliminated the 503 errors and improved application stability, leading to 99.99% uptime.

5. Describe a situation where you had to integrate FastAPI with legacy systems.

Situation: Needed to integrate a new FastAPI service with a legacy SOAP-based system.

Task: Create a reliable integration while maintaining modern API standards.

Action: I:

- Built a service adapter layer using FastAPI's dependency injection
- Implemented request/response transformation middleware
- Created retry mechanisms for unreliable legacy endpoints
- Added comprehensive error handling

Result: Successfully integrated with the legacy system while maintaining modern REST principles and 99.9% service reliability.

6. Share an experience where you had to scale a FastAPI application.

Situation: Our FastAPI application needed to handle 10x more traffic after a successful product launch.

Task: Scale the application to handle increased load while maintaining performance.

Action: I:

- Implemented horizontal scaling with Kubernetes
- Optimized database queries and added connection pooling
- Set up load balancing with nginx
- Implemented caching strategies

Result: Successfully handled 10x traffic increase with no performance degradation and 99.95% uptime.

7. Tell me about a time when you had to implement real-time features in FastAPI.

Situation: Client requested real-time updates for their dashboard application.

Task: Implement real-time data streaming while maintaining system performance.

Action: I:

- Implemented WebSocket endpoints using FastAPI's WebSocket support
- Set up Redis pub/sub for event distribution
- Created connection management system
- Implemented heartbeat mechanism

Result: Delivered real-time updates with less than 100ms latency and supported 10,000+ concurrent connections.

8. Describe a situation where you had to improve API documentation.

Situation: External developers were struggling with API integration due to poor documentation.

Task: Enhance API documentation to reduce support tickets and improve developer experience.

Action: I:

- Enhanced FastAPI's automatic documentation with detailed descriptions
- Added example requests and responses
- Created interactive tutorials using FastAPI's test client
- Implemented documentation testing

Result: Reduced integration-related support tickets by 70% and improved developer onboarding time by 50%.

9. Share an experience where you had to implement complex data validation in FastAPI.

Situation: Need to validate complex nested JSON payloads with conditional validation rules.

Task: Implement robust validation while maintaining code maintainability.

Action: I:

- Created custom Pydantic validators and types
- Implemented dependency validation chains
- Added custom error handling for validation failures
- Created reusable validation components

Result: Achieved 100% validation accuracy while keeping code maintainable and reducing bug reports by 80%.

10. Tell me about a time when you had to mentor junior developers in FastAPI.

Situation: Team expanded with three junior developers new to FastAPI.

Task: Get the new team members productive with FastAPI while maintaining code quality.

Action: I:

- Created a FastAPI best practices guide
- Conducted weekly training sessions
- Set up pair programming sessions
- Implemented code review guidelines

Result: Junior developers became productive within one month, contributing high-quality code that met our standards.