Backend Engineer Case Study

Backend Engineer Case Study: Smart Task Queue System

Build a production-ready task queue system using FastAPI that handles job scheduling, prioritization, and execution with real-world constraints.

Suggested Time: 2-3 hours of focused work

The Challenge

Your company needs a task queue system that processes background jobs for a real time fintech platform. Jobs range from sending emails to processing large data exports. The system must handle priorities, dependencies, and resource constraints intelligently.

Core Requirements

1. API Endpoints

Create a FastAPI application with the following endpoints:

```
POST   /jobs                 # Submit a new job
GET    /jobs/{job_id}        # Get job status and details
GET    /jobs                 # List jobs with filtering
PATCH  /jobs/{job_id}/cancel # Cancel a job if possible
GET    /jobs/{job_id}/logs   # Get job execution logs
WS     /jobs/stream          # WebSocket for real-time updates
```

2. Job Model

Jobs should support: - Different types (email, data*export, report*generation, etc.) - Priority levels (critical, high, normal, low) - Dependencies on other jobs - Resource requirements (CPU units, memory MB) - Retry configuration - Timeout configuration (timeout_seconds) - Status tracking with timestamps

3. Core Features

- **Smart Scheduling**: Jobs execute based on priority, dependencies, and available resources
- **Dependency Management**: Jobs can depend on other jobs completing successfully
- **Resource Allocation**: System tracks resource usage and prevents overload
- **Failure Handling**: Failed jobs retry with exponential backoff
- **Idempotency**: Same job submitted twice doesn't execute twice

4. Production Considerations

- Concurrent job execution (simulate with asyncio.sleep)
- Graceful shutdown (complete running jobs)
- Job timeout handling
- Proper error messages and status codes
- Basic monitoring metrics

Example Input/Output

Submit a Job

```
POST /jobs
{
    "type": "data_export",
    "priority": "high",
    "payload": {
        "user_id": 123,
        "format": "csv"
    },
    "resource_requirements": {
        "cpu_units": 2,
        "memory_mb": 512
    },
    "depends_on": ["job_abc123"],
    "retry_config": {
        "max_attempts": 3,
        "backoff_multiplier": 2
    }
}
```

```
Response:
{
    "job_id": "job_xyz789",
    "status": "pending",
    "created_at": "2024-01-15T10:00:00Z",
    "priority": "high",
    "position_in_queue": 5
}
```

Implementation Guidelines

Database Schema

Design efficient PostgreSQL tables considering: - Job metadata and status - Dependency relationships (graph structure) - Execution history and logs - Resource allocation tracking

Key Technical Decisions

You'll need to make decisions about: - How to handle job dependencies
- How to allocate resources fairly - When to retry vs permanently fail a job - How to handle concurrent updates to job status

Performance Considerations

- Jobs table will grow to millions of rows
- Status queries need to be fast
- Queue operations should be O(log n) or better
- Consider indexes carefully

Deliverables

1. **Code**
    - Complete FastAPI application
    - Database migrations/schema
    - Basic tests for critical paths
    - docker-compose.yml for easy setup
2. **Documentation**
    - README with setup instructions
    - Architecture decisions document

- API documentation
- **Development Journey** (Required) - Include a section describing:
  - Your implementation approach
  - Any challenges you encountered and how you solved them
  - What you would improve with more time
  - How you used AI/documentation and what you had to figure out yourself
- **AI_USAGE.md** - If you used any AI assistants (ChatGPT, Claude, Copilot, etc.), include your conversation history. We want to see how you leveraged tools to solve problems.

3. **Evaluation Results**

- Performance metrics from your test runs
- How the system handles the provided test scenarios
- Discussion of bottlenecks and scaling limits

Evaluation Criteria

We're looking for: - **System design thinking** - not just feature implementation - **Production readiness** - error handling, monitoring, graceful degradation - **Database design** - efficient schema for queue operations - **Code quality** - clean, maintainable, well-tested - **Performance awareness** - understanding of bottlenecks - **Problem-solving approach** - how you handle edge cases

Test Scenarios

We'll provide 5 test scenarios that progressively reveal system complexity: 1. Basic job submission and execution 2. Jobs with simple dependencies 3. Complex dependency graphs with multiple paths 4. Resource contention scenarios 5. Failure and retry scenarios

See test scenarios in the next section of this document.

Important Notes

- **Use any resources** - We encourage using documentation, AI, Stack Overflow
- **Make pragmatic choices** - we prefer working solutions over perfect ones
- **Document trade-offs** - explain what you chose not to implement and why
- **Keep it runnable** - we should be able to docker-compose up and test

Bonus Considerations (if time permits)

- Job scheduling (run at specific times)
- Priority boost for long-waiting jobs
- Dead letter queue for permanently failed jobs
- Basic admin UI for queue monitoring
- Distributed locking for multi-instance deployment

Submission Structure

```
your-name-task-queue/
├── app/
│   ├── main.py
│   ├── models/
│   ├── routes/
│   ├── services/
│   └── workers/
├── tests/
├── migrations/
├── docker-compose.yml
├── requirements.txt
├── README.md
├── ARCHITECTURE.md
├── AI_USAGE.md  # Your AI conversation history if you used AI assistants
└── evaluation_results.md
```

Good luck! We're excited to see how you approach this challenge.

Test Scenarios

Task Queue Test Scenarios

These scenarios progressively test different aspects of your task queue implementation. Each scenario includes the expected behavior and edge cases to consider.

Scenario 1: Basic Job Flow

Description: Test fundamental job submission and execution

```
test_jobs = [
    {
        "type": "send_email",
        "priority": "normal",
        "payload": {"to": "user@example.com", "template": "welcome"},
        "resource_requirements": {"cpu_units": 1, "memory_mb": 128}
    },
    {
        "type": "send_email",
        "priority": "critical",
        "payload": {"to": "vip@example.com", "template": "alert"},
        "resource_requirements": {"cpu_units": 1, "memory_mb": 128}
    },
    {
        "type": "generate_report",
        "priority": "low",
        "payload": {"report_type": "daily_summary", "date": "2024-01-15"},
        "resource_requirements": {"cpu_units": 4, "memory_mb": 2048}
    }
]
```

Expected Behavior: - Critical priority job executes first - Jobs complete in priority order - Resource usage is tracked correctly

Scenario 2: Simple Dependencies

Description: Test job dependency handling

```
dependency_chain = [
    {
        "job_id": "fetch_data_001",
        "type": "data_fetch",
        "priority": "high",
        "payload": {"source": "market_api", "symbols": ["AAPL", "GOOGL"]},
        "resource_requirements": {"cpu_units": 2, "memory_mb": 512}
    },
    {
        "job_id": "process_data_001",
        "type": "data_processing",
        "priority": "high",
        "payload": {"operation": "calculate_indicators"},
        "depends_on": ["fetch_data_001"],
        "resource_requirements": {"cpu_units": 4, "memory_mb": 1024}
    },
    {
        "job_id": "generate_report_001",
        "type": "report_generation",
        "priority": "normal",
        "payload": {"format": "pdf"},
```

```
        "depends_on": ["process_data_001"],
        "resource_requirements": {"cpu_units": 2, "memory_mb": 512}
    }
]
```

Expected Behavior: - Jobs execute in dependency order regardless of priority - If parent job fails, dependent jobs don't run - Status correctly reflects blocked vs ready state

Scenario 3: Complex Dependency Graph

Description: Test handling of complex DAG structures

```
complex_dag = [
    // Two parallel data fetches
    {"job_id": "fetch_prices", "type": "data_fetch", "priority": "high"},
    {"job_id": "fetch_volumes", "type": "data_fetch", "priority": "high"},

    // Processing depends on both fetches
    {
        "job_id": "analyze_market",
        "type": "analysis",
        "depends_on": ["fetch_prices", "fetch_volumes"],
        "priority": "critical"
    },

    // Two parallel reports depend on analysis
    {
        "job_id": "trader_report",
        "type": "report",
        "depends_on": ["analyze_market"],
        "priority": "high"
    },
    {
        "job_id": "risk_report",
        "type": "report",
        "depends_on": ["analyze_market"],
        "priority": "normal"
    },

    // Final job depends on both reports
    {
        "job_id": "send_notifications",
        "type": "notification",
        "depends_on": ["trader_report", "risk_report"],
        "priority": "high"
    }
]
```

Scenario 4: Resource Contention

Description: Test resource allocation under constraints

```
// Assume system has 8 CPU units and 4096 MB memory total
resource_stress_test = [
    // Heavy jobs - only 2 can run simultaneously
    {
        "job_id": "heavy_0",
        "type": "data_processing",
        "priority": "high",
        "payload": {"batch_size": 10000},
        "resource_requirements": {
            "cpu_units": 4,
            "memory_mb": 2048
        }
    },
    {
        "job_id": "heavy_1",
        "type": "data_processing",
        "priority": "high",
        "payload": {"batch_size": 10000},
        "resource_requirements": {
            "cpu_units": 4,
            "memory_mb": 2048
        }
    },
    {
        "job_id": "heavy_2",
        "type": "data_processing",
        "priority": "normal",
        "payload": {"batch_size": 10000},
        "resource_requirements": {
            "cpu_units": 4,
            "memory_mb": 2048
        }
    },
    {
        "job_id": "heavy_3",
        "type": "data_processing",
        "priority": "normal",
        "payload": {"batch_size": 10000},
        "resource_requirements": {
            "cpu_units": 4,
            "memory_mb": 2048
        }
    },
    {
        "job_id": "heavy_4",
        "type": "data_processing",
        "priority": "normal",
        "payload": {"batch_size": 10000},
```

```
        "resource_requirements": {
            "cpu_units": 4,
            "memory_mb": 2048
        }
    },
    // Light jobs that could fill gaps
    {
        "job_id": "light_0",
        "type": "quick_task",
        "priority": "normal",
        "payload": {"task_id": 0},
        "resource_requirements": {
            "cpu_units": 1,
            "memory_mb": 256
        }
    },
    {
        "job_id": "light_1",
        "type": "quick_task",
        "priority": "normal",
        "payload": {"task_id": 1},
        "resource_requirements": {
            "cpu_units": 1,
            "memory_mb": 256
        }
    },
    {
        "job_id": "light_2",
        "type": "quick_task",
        "priority": "normal",
        "payload": {"task_id": 2},
        "resource_requirements": {
            "cpu_units": 1,
            "memory_mb": 256
        }
    },
    {
        "job_id": "light_3",
        "type": "quick_task",
        "priority": "normal",
        "payload": {"task_id": 3},
        "resource_requirements": {
            "cpu_units": 1,
            "memory_mb": 256
        }
    },
    {
        "job_id": "light_4",
        "type": "quick_task",
        "priority": "normal",
        "payload": {"task_id": 4},
        "resource_requirements": {
            "cpu_units": 1,
```

```
            "memory_mb": 256
        }
    }
]
```

Expected Behavior: - System never exceeds resource limits - Higher priority jobs get resources first - Light jobs fill resource gaps efficiently - Fair scheduling prevents starvation

Scenario 5: Failure and Recovery

Description: Test retry logic and failure handling

```
failure_scenarios = [
    {
        "job_id": "unreliable_api_call",
        "type": "external_api",
        "payload": {"endpoint": "flaky_service", "fail_times": 2},
        "retry_config": {
            "max_attempts": 3,
            "backoff_multiplier": 2,
            "initial_delay_seconds": 1
        }
    },
    {
        "job_id": "dependent_on_flaky",
        "type": "processing",
        "depends_on": ["unreliable_api_call"],
        "retry_config": {"max_attempts": 1}
    },
    {
        "job_id": "will_timeout",
        "type": "long_running",
        "payload": {"duration_seconds": 300},
        "timeout_seconds": 60,
        "retry_config": {"max_attempts": 2}
    },
    {
        "job_id": "will_fail_permanently",
        "type": "invalid_operation",
        "payload": {"error": "division_by_zero"},
        "retry_config": {"max_attempts": 3}
    }
]
```

Test Cases: - Job fails twice then succeeds on third attempt - Timeout triggers retry with backoff - Permanent failures don't retry forever - Dependent jobs handle parent failures gracefully

Bonus Scenario: Circular Dependencies

Description: What happens with invalid dependency graphs?

```
circular_deps = [
    {"job_id": "job_a", "depends_on": ["job_c"]},
    {"job_id": "job_b", "depends_on": ["job_a"]},
    {"job_id": "job_c", "depends_on": ["job_b"]}
]
```

Performance Test

Submit 1000 jobs with various priorities and measure: - Time to accept all submissions - Queue operation performance - Memory usage growth - Query performance as queue grows

Evaluation Metrics to Track

1. **Correctness**: Do jobs execute in the right order?
2. **Performance**: How fast are queue operations?
3. **Resource Safety**: Are limits always respected?
4. **Failure Handling**: Do retries work correctly?
5. **Observability**: Can you track what's happening?