ClickUp Backend Interview
Document Event Processing

Complete Preparation Guide - November 30, 2024

November 30, 2025

Contents

1 Interview Overview

1.1 What to Expect

Interview Type: Backend Live Coding (60 minutes)

Format:

- CodeSignal platform (browser-based IDE)
- No pre-written test suite
- You must test your own code
- Recommended languages: Python, JavaScript, TypeScript
- Focus on correctness, edge cases, and code quality

Problem Domain:

- Event processing in memory
- Document state management (similar to Google Docs)
- Real-time collaborative editing
- ClickUp's core functionality

1.2 Success Criteria

1. **Correctness**: Handle all test cases including edge cases
2. **Code Quality**: Clean, readable, well-structured code
3. **Communication**: Think out loud, explain your approach
4. **Testing**: Demonstrate testing strategy
5. **Problem-Solving**: Handle follow-up questions and variations

2 Core Problem: Document Event Processor

Problem

Context: ClickUp has a feature that allows users to create documents similar to Google Docs. Changes are tracked through batches of events.

Task: Process events on a document and return the correctly updated document.

Data Structures:
Document:

```
{
    title: string,
    content: string,          // Lines separated by \n
    lastUpdated: timestamp,
    createdOn: timestamp
}
```

Event:

```
{
    event_id: number,
    event_name: string,       // Case-insensitive
    payload: object,
    timestamp: timestamp
}
```

Event Types:
1. **append** - Add content to a specific line

```
payload: {
    newContent: string,
    startLine: number        // 1-indexed
}
```

If content exists at that line, append newContent to the end of that line.
2. **delete** - Delete all content from document

```
payload: {}  // Empty
```

Requirements:

- Process events in timestamp order
- Update document.lastUpdated to latest event timestamp
- Handle case-insensitive event names
- Return updated document object

Test Cases

Example 1: Multiple Appends
Input:

```
events = [
    {
        "event_id": 1,
        "event_name": "append",
        "payload": {"newContent": "Line 1 ", "startLine": 1},
        "timestamp": 1641024000001
    },
    {
        "event_id": 2,
        "event_name": "APPEND",
        "payload": {"newContent": "Line 2 ", "startLine": 2},
        "timestamp": 1641024000002
    },
    {
        "event_id": 3,
        "event_name": "APPEND",
        "payload": {"newContent": "Line 3 ", "startLine": 3},
        "timestamp": 1641024000003
    }
]

document = {
    "title": "Lorem Ipsum",
    "lastUpdated": 123456789,
    "createdOn": 123456789
}
```

Output:

```
{
    "title": "Lorem Ipsum",
    "content": "Line 1 \nLine 2 \nLine 3 ",
    "lastUpdated": 1641024000003,
    "createdOn": 123456789
}
```

Example 2: Delete Event
Input:

```
events = [
    {
        "event_id": 1,
        "event_name": "delete",
        "timestamp": 1641024000000
    }
]

document = {
    "title": "Lorem Ipsum",
    "content": "This is Lorem ipsum",
    "lastUpdated": 123456789,
    "createdOn": 123456789
}
```

Output:

```
{
    "title": "Lorem Ipsum",
    "content": "",
    "lastUpdated": 1641024000000,
```

3 Solution Strategy

3.1 Approach

1. **Sort events by timestamp** (handle out-of-order)
2. **Initialize content** as list of lines (if document has content)
3. **Process each event** in order:
   - For `append`: Update/create line at specified position
   - For `delete`: Clear all content
4. **Join lines** back into string with \n
5. **Update lastUpdated** to latest timestamp

3.2 Key Insights

- Use **list for lines** - efficient indexing and modification
- Handle **1-indexed lines** (convert to 0-indexed internally)
- **Extend list** if startLine exceeds current length
- Event names are **case-insensitive** (use .lower())
- **Preserve existing line content** when appending

4 Complete Solution

Solution

Python Implementation

```python
def execute(events, document):
    """
    Process document events and return updated document.

    Args:
        events: List of event dictionaries
        document: Document dictionary

    Returns:
        Updated document dictionary
    """
    # Handle empty events
    if not events:
        return document

    # Sort events by timestamp (handle out-of-order)
    sorted_events = sorted(events, key=lambda e: e["timestamp"])

    # Initialize content as list of lines
    content = document.get("content", "")
    lines = content.split("\n") if content else []

    # Track latest timestamp
    latest_timestamp = document["lastUpdated"]

    # Process each event
    for event in sorted_events:
        event_name = event["event_name"].lower()
        timestamp = event["timestamp"]

        if event_name == "append":
            payload = event["payload"]
            new_content = payload["newContent"]
            start_line = payload["startLine"]  # 1-indexed

            # Convert to 0-indexed
            line_idx = start_line - 1

            # Extend lines list if necessary
            while len(lines) <= line_idx:
                lines.append("")

            # Append to existing line content
            lines[line_idx] += new_content

        elif event_name == "delete":
            # Clear all content
            lines = []

        # Update latest timestamp
        latest_timestamp = max(latest_timestamp, timestamp)

    # Join lines back to string
    document["content"] = "\n".join(lines)
    document["lastUpdated"] = latest_timestamp

    return document
```

5 Edge Cases to Handle

Important Edge Cases

Critical Edge Cases:

1. **Empty Events List**
   - Return original document unchanged
   - Don't crash on empty input
2. **Out-of-Order Events**
   - Events may not arrive sorted by timestamp
   - MUST sort before processing
3. **Case-Insensitive Event Names**
   - "append", "APPEND", "Append" all valid
   - Use .lower() for comparison
4. **Gaps in Line Numbers**
   - Append to line 5 when only 2 lines exist
   - Fill with empty lines in between
5. **Appending to Same Line**
   - Multiple appends to line 1: concatenate all
   - Don't replace, append to existing content
6. **Delete Then Append**
   - Delete clears all lines
   - Subsequent appends start fresh
7. **Document Without Content Field**
   - Initialize as empty string
   - Use document.get("content", "")
8. **Empty Payload**
   - Delete event has empty payload
   - Handle missing keys gracefully
9. **Line Number Edge Cases**
   - startLine = 1 (first line, 1-indexed)
   - startLine = 0 or negative (invalid - skip or error)
   - startLine = 1000 (fill gaps with empty lines)
10. **Trailing Newline Handling**
    - Be careful with split/join creating extra empty lines
    - Test with content ending in \n

6 Comprehensive Test Cases

Test Cases

```
def test_document_processor():
    """Comprehensive test suite."""

    # Test 1: Basic append
    events = [{
        "event_id": 1,
        "event_name": "append",
        "payload": {"newContent": "Hello", "startLine": 1},
        "timestamp": 100
    }]
    doc = {"title": "Test", "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == "Hello"
    assert result["lastUpdated"] == 100
    print("Test 1 passed: Basic append")

    # Test 2: Multiple appends to different lines
    events = [
        {"event_id": 1, "event_name": "append",
         "payload": {"newContent": "Line1 ", "startLine": 1},
         "timestamp": 100},
        {"event_id": 2, "event_name": "append",
         "payload": {"newContent": "Line2 ", "startLine": 2},
         "timestamp": 101}
    ]
    doc = {"title": "Test", "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == "Line1 \nLine2 "
    assert result["lastUpdated"] == 101
    print("Test 2 passed: Multiple appends")

    # Test 3: Multiple appends to SAME line
    events = [
        {"event_id": 1, "event_name": "append",
         "payload": {"newContent": "Hello ", "startLine": 1},
         "timestamp": 100},
        {"event_id": 2, "event_name": "append",
         "payload": {"newContent": "World", "startLine": 1},
         "timestamp": 101}
    ]
    doc = {"title": "Test", "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == "Hello World"
    print("Test 3 passed: Same line appends")

    # Test 4: Delete event
    events = [{
        "event_id": 1,
        "event_name": "delete",
        "timestamp": 100
    }]
    doc = {"title": "Test", "content": "Some content",
           "lastUpdated": 0, "createdOn": 0}
    result = execute(events, doc)
    assert result["content"] == ""
    assert result["lastUpdated"] == 100
    print("Test 4 passed: Delete event")

    # Test 5: Delete then append
    events = [
```

7 Follow-Up Questions & Variations

7.1 Expected Follow-Ups

1. **What if events can arrive out of order?**
   - Already handled by sorting
   - Could use priority queue for streaming
2. **What if we need to support undo/redo?**
   - Store event history
   - Apply events up to specific point in time
   - Implement reverse operations
3. **What about concurrent editing by multiple users?**
   - Operational Transform (OT)
   - Conflict-free Replicated Data Types (CRDTs)
   - Last-write-wins with timestamps
4. **How would you optimize for very large documents?**
   - Lazy loading of content
   - Chunk-based storage
   - Only load/process visible lines
5. **What if we add more event types?**
   - Insert at position within line
   - Delete specific line
   - Replace content
   - Format events (bold, italic)

7.2 Variation: Insert Event

New Event Type: insert

Instead of appending to end of line, insert at specific character position:

```
{
    "event_name": "insert",
    "payload": {
        "newContent": "text",
        "startLine": 2,
        "position": 5  # Character position in line
    },
    "timestamp": 100
}
```

Implementation:

```
elif event_name == "insert":
    new_content = payload["newContent"]
    start_line = payload["startLine"]
    position = payload["position"]

```

```
    line_idx = start_line - 1
    while len(lines) <= line_idx:
        lines.append("")

    line = lines[line_idx]
    # Insert at position
    lines[line_idx] = line[:position] + new_content + line[position:]
```

7.3 Variation: Delete Specific Lines

Delete Range Event:

```
{
    "event_name": "delete_range",
    "payload": {
        "startLine": 2,
        "endLine": 4    # Inclusive
    },
    "timestamp": 100
}
```

Implementation:

```
elif event_name == "delete_range":
    start_line = payload["startLine"]
    end_line = payload["endLine"]

    start_idx = start_line - 1
    end_idx = end_line - 1

    # Delete lines in range
    if start_idx < len(lines):
        del lines[start_idx:min(end_idx + 1, len(lines))]
```

7.4 Variation: Event Validation

Add validation and return errors:

```
def execute_with_validation(events, document):
    """
    Version that validates events and returns errors.
    """
    errors = []
    valid_events = []

    for event in events:
        # Validate event structure
        if "event_name" not in event:
            errors.append(f"Event {event.get('event_id')} missing
                event_name")
            continue

        if "timestamp" not in event:
            errors.append(f"Event {event.get('event_id')} missing
                timestamp")
            continue

        event_name = event["event_name"].lower()
```

```
19
20          if event_name == "append":
21              payload = event.get("payload", {})
22              if "newContent" not in payload or "startLine" not in
                    payload:
23                  errors.append(f"Event {event['event_id']} invalid
                        append payload")
24                  continue
25
26              if payload["startLine"] < 1:
27                  errors.append(f"Event {event['event_id']} invalid line
                        number")
28                  continue
29
30          valid_events.append(event)
31
32      # Process valid events
33      result = execute(valid_events, document)
34
35      return result, errors
```

8 Alternative Implementations

8.1 JavaScript/TypeScript Version

```javascript
function execute(events, document) {
    // Handle empty events
    if (!events || events.length === 0) {
        return document;
    }

    // Sort by timestamp
    const sortedEvents = [...events].sort((a, b) =>
        a.timestamp - b.timestamp
    );

    // Initialize content
    let content = document.content || "";
    let lines = content ? content.split("\n") : [];

    let latestTimestamp = document.lastUpdated;

    // Process events
    for (const event of sortedEvents) {
        const eventName = event.event_name.toLowerCase();
        const timestamp = event.timestamp;

        if (eventName === "append") {
            const { newContent, startLine } = event.payload;
            const lineIdx = startLine - 1;

            // Extend lines array if needed
            while (lines.length <= lineIdx) {
                lines.push("");
            }

            // Append to line
            lines[lineIdx] += newContent;

        } else if (eventName === "delete") {
            lines = [];
        }

        latestTimestamp = Math.max(latestTimestamp, timestamp);
    }

    // Update document
    return {
        ...document,
        content: lines.join("\n"),
        lastUpdated: latestTimestamp
    };
}
```

8.2 Object-Oriented Approach

```python
class DocumentProcessor:
    """OOP approach for document event processing."""
```

```python

    def __init__(self, document):
        self.document = document
        self.lines = self._init_lines()

    def _init_lines(self):
        """Initialize lines from document content."""
        content = self.document.get("content", "")
        return content.split("\n") if content else []

    def process_events(self, events):
        """Process all events and return updated document."""
        if not events:
            return self.document

        sorted_events = sorted(events, key=lambda e: e["timestamp"])

        for event in sorted_events:
            self._process_event(event)

        return self._finalize()

    def _process_event(self, event):
        """Process single event."""
        event_name = event["event_name"].lower()

        if event_name == "append":
            self._handle_append(event)
        elif event_name == "delete":
            self._handle_delete(event)

    def _handle_append(self, event):
        """Handle append event."""
        payload = event["payload"]
        new_content = payload["newContent"]
        start_line = payload["startLine"]

        line_idx = start_line - 1

        while len(self.lines) <= line_idx:
            self.lines.append("")

        self.lines[line_idx] += new_content

        self.document["lastUpdated"] = event["timestamp"]

    def _handle_delete(self, event):
        """Handle delete event."""
        self.lines = []
        self.document["lastUpdated"] = event["timestamp"]

    def _finalize(self):
        """Finalize and return document."""
        self.document["content"] = "\n".join(self.lines)
        return self.document


# Usage
```

```
processor = DocumentProcessor(document)
result = processor.process_events(events)
```

9 System Design Considerations

9.1 Real-World ClickUp Architecture

In production, ClickUp likely uses:

1. **Event Store**
   - Kafka or similar for event streaming
   - Permanent event log (source of truth)
   - Can replay events to rebuild state
2. **CRDT (Conflict-free Replicated Data Types)**
   - Handle concurrent edits from multiple users
   - Eventual consistency
   - Examples: Yjs, Automerge
3. **Operational Transform**
   - Transform conflicting operations
   - Maintain causal ordering
   - Used by Google Docs
4. **Snapshot + Events**
   - Store periodic snapshots
   - Apply only recent events
   - Faster recovery

9.2 Scalability Challenges

- **Large Documents**: Millions of characters
- **High Event Rate**: Hundreds of events per second
- **Many Concurrent Users**: Real-time collaboration
- **Offline Support**: Sync when reconnected
- **Undo/Redo**: Maintain operation history

9.3 Follow-Up Discussion Points

Be ready to discuss:

1. How would you handle conflicts in real-time collaboration?
2. What data structures optimize for large documents?
3. How do you ensure consistency across multiple clients?
4. What's your strategy for event ordering with distributed systems?
5. How would you implement autosave with event batching?

10 Interview Tips & Strategy

10.1 Before You Code

1. **Clarify Requirements** (2-3 minutes)
   - Ask about edge cases
   - Confirm input/output format
   - Understand constraints
2. **Discuss Approach** (3-5 minutes)
   - Explain your strategy
   - Mention data structures
   - Discuss time/space complexity
3. **Consider Edge Cases**
   - Empty inputs
   - Out-of-order events
   - Invalid data
   - Boundary conditions

10.2 While Coding

1. **Think Out Loud**
   - Explain your thought process
   - Voice concerns and trade-offs
   - Ask questions if stuck
2. **Write Clean Code**
   - Clear variable names
   - Logical structure
   - Add comments for complex logic
3. **Handle Errors Gracefully**
   - Check for None/null
   - Validate inputs
   - Use .get() for dictionaries

10.3 After Coding

1. **Test Your Code**
   - Run provided examples
   - Test edge cases
   - Fix any bugs found
2. **Discuss Improvements**

- Optimization opportunities
- Alternative approaches
- Production considerations

3. **Handle Follow-Ups**
   - Be ready for variations
   - Explain how to extend solution
   - Discuss system design implications

10.4 Communication Checklist

Clarified requirements upfront

Explained approach before coding

Thought out loud while implementing

Handled edge cases explicitly

Tested code with examples

Discussed complexity analysis

Proposed optimizations

Asked intelligent follow-up questions

11 Quick Reference

11.1 Key Points to Remember

1. **Sort events by timestamp first**
2. **Use list for lines** (not string manipulation)
3. **Handle 1-indexed lines** (subtract 1 for array index)
4. **Case-insensitive event names** (use .lower())
5. **Append to existing content** (don't replace)
6. **Fill gaps with empty lines**
7. **Update lastUpdated to latest event**

11.2 Common Mistakes to Avoid

- Forgetting to sort events
- Not handling case-insensitive names
- Replacing line content instead of appending
- Off-by-one errors with line numbers
- Not handling empty/missing fields
- Creating extra empty lines with split/join

11.3 Time Allocation (60 minutes)

- **5 min**: Clarify requirements, ask questions
- **5 min**: Discuss approach and edge cases
- **30 min**: Implement solution
- **10 min**: Test with examples and edge cases
- **10 min**: Follow-up questions and discussion

11.4 Python Quick Reference

```
# Sort by timestamp
sorted_events = sorted(events, key=lambda e: e["timestamp"])

# Case-insensitive comparison
event_name = event["event_name"].lower()

# Safe dictionary access
content = document.get("content", "")
payload = event.get("payload", {})

# List operations
lines = []
lines.append("new line")
```

```
lines[idx] += "append"
del lines[start:end]

# String operations
lines = content.split("\n")
content = "\n".join(lines)
```

12 Additional Practice Problems

12.1 Problem 1: Event Replay with Time Window

Modify the solution to replay events only within a time window:

```
def execute_with_window(events, document, start_time, end_time):
    """
    Process only events within time window [start_time, end_time].
    """
    # Filter events by time window
    filtered = [e for e in events
                if start_time <= e["timestamp"] <= end_time]

    return execute(filtered, document)
```

12.2 Problem 2: Event History

Return both the final document and the complete history:

```
def execute_with_history(events, document):
    """
    Return final document plus history of all intermediate states.
    """
    history = [document.copy()]

    sorted_events = sorted(events, key=lambda e: e["timestamp"])
    lines = []

    for event in sorted_events:
        # Process event (similar to original)
        # ...

        # Snapshot current state
        snapshot = document.copy()
        snapshot["content"] = "\n".join(lines)
        snapshot["lastUpdated"] = event["timestamp"]
        history.append(snapshot)

    return history[-1], history
```

12.3 Problem 3: Collaborative Conflict Resolution

Handle concurrent edits from multiple users:

```
def execute_with_conflicts(events, document):
    """
    Detect and resolve conflicts in concurrent edits.

    Conflict: Same line modified at same timestamp by different users.
    Resolution: Last event_id wins.
    """
    sorted_events = sorted(events,
                           key=lambda e: (e["timestamp"], e["event_id"])
                              )

    # Rest similar to original execute()
    return execute(sorted_events, document)
```

13 Final Checklist

13.1 Before Interview

- [ ] Practiced core problem multiple times
- [ ] Can implement solution in under 30 minutes
- [ ] Memorized edge cases
- [ ] Tested with all provided examples
- [ ] Understand time/space complexity
- [ ] Reviewed follow-up variations
- [ ] Comfortable with both Python and JavaScript
- [ ] Practiced explaining thought process out loud

13.2 During Interview

- [ ] Listen carefully to requirements
- [ ] Ask clarifying questions
- [ ] Explain approach before coding
- [ ] Write clean, readable code
- [ ] Think out loud
- [ ] Test with examples
- [ ] Handle edge cases explicitly
- [ ] Be open to feedback and hints

13.3 Key Success Factors

1. **Correctness** - Solution works for all cases
2. **Code Quality** - Clean, maintainable code
3. **Communication** - Clear explanation of thinking
4. **Problem-Solving** - Handle unexpected challenges
5. **Collaboration** - Work well with interviewer

Good luck with your ClickUp interview!

Remember: The interviewer wants you to succeed. They're evaluating not just your coding skills, but how you think, communicate, and collaborate. Show your problem-solving process, ask good questions, and demonstrate your passion for building great products.