Temporal Training Session 5: Signals and Queries

Table of Contents

Introduction

On Day 5, we focus on advanced workflow interaction patterns in Temporal: **Signals** and **Queries**. These features allow external systems and users to interact with running workflows, enabling dynamic, real-time business processes. We'll go beyond the basics to cover advanced usage, best practices, and real-world scenarios.

Signals and Queries: Conceptual Overview

What are Signals?

- **Signals** are asynchronous, durable messages sent to a running workflow.
- They allow external systems, users, or other workflows to inject information, trigger state changes, or prompt workflows to take specific actions.
- Signals are persisted by the Temporal server and delivered even if the workflow is not currently running.

Signal Lifecycle:

1. Signal is sent from a client or another workflow.
2. Temporal persists the signal event in the workflow's event history.
3. When the workflow is scheduled, the signal handler is invoked.
4. The workflow can react immediately or buffer signals for later processing.

Signal Use Cases:

- Human-in-the-loop approval (e.g., waiting for a manager's decision)
- Webhook triggers (e.g., payment confirmation, shipment update)
- IoT device control (e.g., send command to a device)
- SLA monitoring (e.g., escalate if not approved in time)

What are Queries?

- **Queries** are synchronous, read-only requests to a running workflow.
- They allow external systems to inspect the current state of a workflow without causing any side effects.
- Queries are not persisted in the workflow history and are only available while the workflow is running.

Query Lifecycle:

1. Query is sent from a client.
2. Temporal routes the query to a worker running the workflow.
3. The query handler is invoked and returns the result.

Query Use Cases:

- Polling for workflow status (e.g., order status, progress)
- Fetching workflow variables for dashboards
- SLA monitoring (e.g., how long in current state)

Signals: Advanced Usage and Patterns

Defining and Handling Signals

```
from temporalio import workflow

@workflow.defn
class ApprovalWorkflow:
    def __init__(self):
        self.approval_status = None
        self.comments = []
        self.signal_buffer = []

    @workflow.run
    async def run(self, item_id: str):
        workflow.logger.info(f"Started workflow for {item_id}")
        # Example: batch process signals every 10 seconds
        while self.approval_status is None:
            await workflow.sleep(10)
            if self.signal_buffer:
                for status, comment in self.signal_buffer:
                    self._process_signal(status, comment)
                self.signal_buffer.clear()
        if self.approval_status == "approved":
            workflow.logger.info("Approved!")
        else:
            workflow.logger.info("Rejected.")

    @workflow.signal
    async def approve(self, status: str, comment: str = ""):
        # Buffer signals for batch processing
        self.signal_buffer.append((status, comment))

    def _process_signal(self, status, comment):
        self.approval_status = status
        if comment:
            self.comments.append(comment)
```

Sending Signals

- From a client:

```
await handle.signal(ApprovalWorkflow.approve, "approved", "Looks
good!")
```

- From another workflow:

```
await workflow.signal_external_workflow(
    workflow_id="target-workflow-id",
    signal=ApprovalWorkflow.approve,
```

```
        args=["approved", "Auto-approved by system"]
    )
```

Signal Patterns

- **Broadcast Signals:** Send the same signal to multiple workflows (e.g., system-wide pause).
- **Signal Fan-In:** Aggregate multiple signals before proceeding (e.g., wait for N approvals).
- **Signal Batching:** Buffer signals and process them in batches for efficiency.
- **Signal with Payload:** Send complex data structures as signal arguments (ensure they are serializable).
- **Deduplication:** Use unique IDs in signals to ignore duplicates (e.g., for idempotency).

Signal Batching and Deduplication

- Buffer incoming signals and process them periodically to reduce workflow task load.
- Use a set or dict to deduplicate signals by unique key (e.g., user ID, event ID).

Error Handling in Signal Handlers

- Always validate signal payloads (e.g., check types, required fields).
- Use try/except blocks to catch and log errors in signal handlers.
- If a signal is invalid, log and ignore or send a compensating signal.

Queries: Advanced Usage and Patterns

Defining and Handling Queries

```
@workflow.defn
class ApprovalWorkflow:
    def __init__(self):
        self.status = "pending"
        self.comments = []
        self.last_query_time = None

    @workflow.run
    async def run(self, item_id: str):
        self.status = "awaiting_approval"
        await workflow.wait_condition(lambda: self.status !=
"awaiting_approval")
        # ... rest of workflow

    @workflow.query
    def get_status(self) -> str:
        self.last_query_time = workflow.now()
        return self.status

    @workflow.query
    def get_comments(self) -> list:
        return self.comments
```

Query Consistency and Staleness

- Queries are **strongly consistent**: they reflect the latest state as of the last completed workflow task.
- If a workflow is busy, queries may be delayed until the workflow task completes.
- For high-frequency queries, consider returning cached/snapshotted state to reduce workflow task load.

Error Handling in Query Handlers

- Validate query parameters (if any).
- Catch and log exceptions; return a default or error message if needed.
- Never perform side effects (e.g., writes, external calls) in query handlers.

Query Patterns

- **Polling:** Periodically query workflow state for progress or status.
- **Snapshot Queries:** Return a snapshot of workflow state for dashboards or monitoring.
- **Custom Queries:** Expose multiple query handlers for different aspects of workflow state.

Query Limitations

- Queries must be **side-effect free** and **deterministic**.
- Queries are not persisted in workflow history.
- If no worker is available to serve the query, the query will fail.

Best Practices and Anti-Patterns

Best Practices

- **Idempotency:** Signal handlers should be idempotent to handle duplicate delivery.
- **Validation:** Validate signal/query payloads to prevent invalid state changes.
- **Timeouts:** Use timeouts when waiting for signals to avoid indefinite waits.
- **Buffering:** Buffer signals if you expect bursts or high frequency.
- **Documentation:** Document all available signals and queries for each workflow.
- **Monitoring:** Log all received signals and queries for auditing and debugging.

Anti-Patterns

- **Side Effects in Queries:** Never perform side effects (e.g., database writes) in query handlers.
- **Blocking in Signal Handlers:** Avoid long-running or blocking operations in signal handlers.
- **Overusing Signals:** Don't use signals for high-frequency, low-latency communication (use activities or external queues).
- **Ignoring Signal Ordering:** Don't assume signals will always arrive in the order you expect if sent from distributed systems.
- **Ignoring Security:** Never expose signal/query endpoints without authentication/authorization.

Security and Compliance

- **Authentication:** Ensure only authorized clients can send signals or queries (enforce at the application layer).
- **Validation:** Validate all incoming data to prevent injection or corruption.
- **Auditability:** Signals are persisted in workflow history, providing an audit trail.
- **Sensitive Data:** Avoid sending sensitive data in signals/queries unless encrypted and access-controlled.

Authentication and Authorization Patterns

- Use API gateways or middleware to authenticate clients before allowing them to send signals/queries.
- Implement role-based access control (RBAC) for sensitive workflows.
- Log all access attempts for compliance and auditing.

Performance, Scaling, and Monitoring

Performance

- Signals are lightweight but can be delayed if the workflow is busy or not scheduled.
- Queries are fast but require a worker to be available and the workflow to be loaded.
- For high-throughput scenarios, batch signals and cache query results.

Scaling

- For high signal/query volume, scale workers horizontally.
- Use **sticky queues** to keep workflows loaded on the same worker for faster query response.
- Partition workflows by business key (e.g., customer ID) to distribute load.

Monitoring Signals and Queries

- Monitor signal delivery latency and query response times.
- Use Temporal's Web UI and metrics to track signal/query activity.
- Log all received signals and queries for debugging and auditing.
- Set up alerts for failed signal deliveries or query timeouts.

Troubleshooting and Debugging

- **Missed Signals:** Check workflow history for signal events; ensure workflow is running and not completed.
- **Query Failures:** Ensure a worker is available and the workflow is not completed.
- **Debugging:** Use Temporal Web UI to inspect signal and query events, and workflow state.
- **Retries:** If a signal fails to deliver, Temporal will retry until the workflow completes.
- **Common Issues:**
    - Signal handler not registered: Ensure the signal method is decorated and present in the workflow class.
    - Query returns stale data: Check if workflow is busy or overloaded.
    - Security errors: Check authentication/authorization middleware and logs.

Real-World Scenarios and Extended Examples

Example 1: Multi-Stage Approval Workflow

- Multiple users send approval signals.
- Workflow waits for a quorum (e.g., 3 out of 5 approvals).
- Exposes queries for current approval count and list of approvers.

```
@workflow.defn
class MultiApprovalWorkflow:
    def __init__(self):
        self.approvals = set()
        self.required = 3

    @workflow.run
    async def run(self, item_id: str):
        while len(self.approvals) < self.required:
            await workflow.wait_condition(lambda: len(self.approvals) >=
self.required)
        # Proceed with next steps

    @workflow.signal
    async def approve(self, user_id: str):
        self.approvals.add(user_id)

    @workflow.query
    def approval_count(self) -> int:
        return len(self.approvals)

    @workflow.query
    def approvers(self) -> list:
        return list(self.approvals)
```

Example 2: Real-Time Order Status Polling

- E-commerce order workflow receives signals for payment, shipment, delivery.
- Exposes queries for order status, shipment tracking, and history.

Example 3: Webhook-Driven Workflows

- External systems (e.g., payment gateways) send signals to workflows via webhooks.
- Workflow reacts to signals for payment confirmation, fraud alerts, etc.

Example 4: IoT Device Control

- IoT devices send signals to a workflow to report status or receive commands.
- Workflow queries expose device state, last communication time, and error logs.

Example 5: SLA Monitoring

- Workflow receives signals for events (e.g., task started, completed).
- Queries expose time spent in each state and whether SLA is breached.

FAQ: Signals and Queries

Q: Can signals be lost?
A: No, signals are persisted and delivered at-least-once as long as the workflow is running or not completed.

Q: Can queries change workflow state?
A: No, queries must be side-effect free and deterministic.

Q: What happens if a signal arrives after the workflow completes?
A: The signal is dropped and not delivered.

Q: How do I secure signals and queries?
A: Use authentication/authorization at the application or API gateway layer.

Q: Can I send signals to multiple workflows at once?
A: Yes, by iterating over workflow handles and sending the signal to each.

Q: How do I debug signal or query issues?
A: Use Temporal Web UI to inspect workflow history, signal/query events, and logs.

Summary

- **Signals** and **Queries** are powerful tools for interacting with running workflows.
- Use advanced patterns (fan-in, batching, broadcast) for complex scenarios.
- Follow best practices for idempotency, validation, and security.
- Monitor and scale your workers to handle high signal/query volume.
- Use Temporal's observability tools for troubleshooting and auditing.
