Temporal Training Session 7: Comprehensive Error Handling and Retry Strategies

Table of Contents

Introduction and Recap

Welcome to Day 7 of our Temporal training. Yesterday, we explored timers, sleep, and external waits—crucial for building workflows that interact with the real world and handle time-based logic. Today, we focus on a topic that is at the heart of building robust, production-grade systems: **error handling and retry strategies**.

Learning Objectives:

- Understand how errors propagate in Temporal
- Learn to configure and use custom retry policies
- Distinguish between specific and generic error handling
- Implement compensation and cleanup logic for failed workflows
- Apply best practices for error handling in distributed workflows

---

Error Propagation in Temporal

How Errors Flow

In Temporal, errors can occur in both workflows and activities. Understanding how these errors propagate is essential for designing resilient systems.

- **Activity Errors:** If an activity fails (raises an exception, times out, or is cancelled), the error is reported to the workflow. The workflow can catch and handle the error, or let it propagate up.
- **Workflow Errors:** If a workflow raises an unhandled exception, the workflow execution fails. The error is recorded in the workflow history and can be inspected via the Temporal Web UI or CLI.

Error Types

- **Application Errors:** Raised by your code (e.g., `raise Exception("Invalid input")`)
- **Timeouts:** Activity or workflow exceeds its configured timeout
- **Cancellations:** Workflow or activity is cancelled (e.g., by a signal or external request)
- **Child Workflow Failures:** Errors in child workflows propagate to the parent

---

Types of Errors in Workflows and Activities

ActivityError

When an activity fails, Temporal raises an `ActivityError` in the workflow. This error contains information about the cause, including the original exception, timeout, or cancellation.

```
from temporalio import workflow
from temporalio.exceptions import ActivityError

@workflow.defn
class ErrorHandlingWorkflow:
    @workflow.run
    async def run(self, input_data: str) -> str:
        try:
            result = await workflow.execute_activity(
                risky_activity,
                input_data,
                start_to_close_timeout=timedelta(seconds=30)
            )
            return result
        except ActivityError as e:
            workflow.logger.error(f"Activity failed: {e}")
            return "Activity failed"
```

ApplicationError

You can raise custom errors in your activities or workflows to signal specific failure conditions.

```
from temporalio import activity

@activity.defn
async def validate_input(input_data: str):
    if not input_data:
        raise ValueError("Input data cannot be empty")
    return True
```

TimeoutError

Temporal raises a `TimeoutError` if an activity or workflow exceeds its timeout.

```
from temporalio.exceptions import TimeoutError

try:
    await workflow.execute_activity(
        slow_activity,
        start_to_close_timeout=timedelta(seconds=5)
    )
except TimeoutError:
    workflow.logger.warning("Activity timed out")
```

CancelledError

If a workflow or activity is cancelled, a `CancelledError` is raised.

```
from temporalio.exceptions import CancelledError

try:
    await workflow.execute_activity(
        cancellable_activity,
        start_to_close_timeout=timedelta(seconds=30)
    )
except CancelledError:
    workflow.logger.info("Activity was cancelled")
```

Custom Retry Logic with RetryPolicy

Temporal's retry mechanism is a powerful tool for handling transient failures. You can configure retry policies for activities and child workflows.

RetryPolicy Parameters

- **initial_interval:** Time to wait before the first retry
- **backoff_coefficient:** Multiplier for exponential backoff
- **maximum_interval:** Maximum wait time between retries
- **maximum_attempts:** Maximum number of attempts (including the first)
- **non_retryable_error_types:** List of error types that should not be retried

Example: Configuring RetryPolicy

```
from temporalio.common import RetryPolicy

retry_policy = RetryPolicy(
    initial_interval=timedelta(seconds=2),
    backoff_coefficient=2.0,
    maximum_interval=timedelta(seconds=30),
    maximum_attempts=5,
```

```
    non_retryable_error_types=["ValueError"]
)

result = await workflow.execute_activity(
    flaky_activity,
    start_to_close_timeout=timedelta(seconds=10),
    retry_policy=retry_policy
)
```

Disabling Retries

Set `maximum_attempts=1` to disable retries for an activity.

```
retry_policy = RetryPolicy(maximum_attempts=1)
```

Handling Specific vs. Generic Errors

Catching Specific Exceptions

Catching specific exceptions allows you to handle different error conditions appropriately.

```
try:
    await workflow.execute_activity(
        validate_payment_activity,
        start_to_close_timeout=timedelta(seconds=30)
    )
except ValueError as e:
    workflow.logger.error(f"Validation error: {e}")
    # Handle validation error
except TimeoutError:
    workflow.logger.warning("Payment validation timed out")
    # Handle timeout
except Exception as e:
    workflow.logger.error(f"Unexpected error: {e}")
    # Handle generic error
```

Handling Non-Retryable Errors

Use `non_retryable_error_types` in your retry policy to prevent retries for certain errors.

```
retry_policy = RetryPolicy(
    maximum_attempts=3,
    non_retryable_error_types=["ValueError"]
)
```

try/except Patterns in Workflows

Basic try/except

```
try:
    result = await workflow.execute_activity(
        process_order_activity,
        start_to_close_timeout=timedelta(seconds=60)
    )
    return result
except Exception as e:
    workflow.logger.error(f"Order processing failed: {e}")
    # Perform compensation or cleanup
    await workflow.execute_activity(
        rollback_order_activity,
        start_to_close_timeout=timedelta(seconds=30)
    )
    return "Order rolled back due to failure"
```

Nested try/except for Multiple Steps

```
try:
    await workflow.execute_activity(step_one_activity)
    try:
        await workflow.execute_activity(step_two_activity)
    except Exception as e2:
        workflow.logger.error(f"Step two failed: {e2}")
        # Handle step two failure
except Exception as e1:
    workflow.logger.error(f"Step one failed: {e1}")
    # Handle step one failure
```

Compensation and Cleanup Logic

In distributed systems, it's often necessary to compensate for failed operations. This is known as the **Saga Pattern**.

Example: Saga Pattern

```
@workflow.defn
class PaymentSagaWorkflow:
    @workflow.run
    async def run(self, order_id: str, amount: float) -> str:
        try:
            await workflow.execute_activity(
```

```
                reserve_inventory_activity,
                order_id,
                start_to_close_timeout=timedelta(seconds=30)
            )
            await workflow.execute_activity(
                charge_payment_activity,
                order_id,
                amount,
                start_to_close_timeout=timedelta(seconds=30)
            )
            await workflow.execute_activity(
                ship_order_activity,
                order_id,
                start_to_close_timeout=timedelta(seconds=60)
            )
            return "Order completed successfully"
        except Exception as e:
            workflow.logger.error(f"Workflow failed: {e}")
            # Compensation logic
            await workflow.execute_activity(
                release_inventory_activity,
                order_id,
                start_to_close_timeout=timedelta(seconds=30)
            )
            await workflow.execute_activity(
                refund_payment_activity,
                order_id,
                amount,
                start_to_close_timeout=timedelta(seconds=30)
            )
            return "Order failed and compensation completed"
```

Real-World Example: Payment Failure and Refund

Scenario

A customer places an order. The workflow reserves inventory, charges the payment, and ships the order. If payment fails, the workflow must release the inventory and issue a refund.

```
@workflow.defn
class OrderWorkflow:
    @workflow.run
    async def run(self, order_id: str, amount: float) -> str:
        try:
            await workflow.execute_activity(
                reserve_inventory_activity,
                order_id,
                start_to_close_timeout=timedelta(seconds=30)
            )
            await workflow.execute_activity(
```

```
                charge_payment_activity,
                order_id,
                amount,
                start_to_close_timeout=timedelta(seconds=30)
            )
            await workflow.execute_activity(
                ship_order_activity,
                order_id,
                start_to_close_timeout=timedelta(seconds=60)
            )
            return "Order completed successfully"
        except Exception as e:
            workflow.logger.error(f"Order failed: {e}")
            # Compensation logic
            await workflow.execute_activity(
                release_inventory_activity,
                order_id,
                start_to_close_timeout=timedelta(seconds=30)
            )
            await workflow.execute_activity(
                refund_payment_activity,
                order_id,
                amount,
                start_to_close_timeout=timedelta(seconds=30)
            )
            return "Order failed and compensation completed"
```

Advanced Error Handling Patterns

Pattern 1: Retry with Fallback

```
try:
    result = await workflow.execute_activity(
        primary_service_activity,
        start_to_close_timeout=timedelta(seconds=10)
    )
    return result
except Exception:
    workflow.logger.warning("Primary service failed, using fallback")
    result = await workflow.execute_activity(
        fallback_service_activity,
        start_to_close_timeout=timedelta(seconds=30)
    )
    return result
```

Pattern 2: Error Aggregation

Aggregate multiple errors and report them at the end.

```
errors = []
for item in items:
    try:
        await workflow.execute_activity(process_item_activity, item)
    except Exception as e:
        errors.append((item, str(e)))
if errors:
    workflow.logger.error(f"Errors occurred: {errors}")
    # Handle aggregated errors
```

Pattern 3: Escalation on Repeated Failure

```
retry_count = 0
max_retries = 3
while retry_count < max_retries:
    try:
        await workflow.execute_activity(
            flaky_activity,
            start_to_close_timeout=timedelta(seconds=10)
        )
        break
    except Exception as e:
        retry_count += 1
        if retry_count == max_retries:
            await workflow.execute_activity(
                escalate_issue_activity,
                start_to_close_timeout=timedelta(seconds=30)
            )
            raise e
        await workflow.sleep(2 ** retry_count)
```

Best Practices and Anti-Patterns

Best Practices

1. **Use Specific Exceptions:** Catch and handle specific exceptions where possible
2. **Configure Retry Policies:** Use `RetryPolicy` for transient errors
3. **Implement Compensation:** Always provide compensation logic for critical operations
4. **Log All Errors:** Use structured logging for all error cases
5. **Test Error Scenarios:** Write tests for failure and compensation paths

Anti-Patterns

1. **Catching All Exceptions Without Handling:**

```
try:
    ...
except Exception:
    pass  # Bad: Swallows errors silently
```

2. **Infinite Retry Loops:**

```
while True:
    try:
        ...
        break
    except Exception:
        pass  # Bad: Can cause infinite retries
```

3. **No Compensation Logic:**
    - Failing to clean up after errors can leave systems in inconsistent states

---

Testing Error Handling

Unit Testing with Mocks

Use mocking to simulate activity failures and test workflow error handling.

```
from unittest.mock import patch

@patch('my_module.risky_activity', side_effect=Exception("Simulated
failure"))
def test_workflow_handles_activity_failure(mock_activity):
    # Start workflow and assert compensation logic is triggered
    ...
```

Integration Testing

Run workflows against a local Temporal server and inject failures to test end-to-end error handling.

Troubleshooting Common Issues

1. **Retries Not Working:**
    - Check `RetryPolicy` configuration
    - Ensure exceptions are not in `non_retryable_error_types`
2. **Compensation Not Triggered:**
    - Verify compensation logic is in the correct except block
3. **Silent Failures:**
    - Ensure all exceptions are logged

    - Avoid bare except blocks
4. **Timeouts Not Firing:**
    - Check activity and workflow timeout settings

Summary

Today, we covered comprehensive error handling and retry strategies in Temporal:

- How errors propagate in workflows and activities
- Configuring and using custom retry policies
- Handling specific and generic errors
- Implementing compensation and cleanup logic
- Advanced error handling patterns and best practices
- Testing and troubleshooting error scenarios

Key Takeaways:

- Always handle errors explicitly and log them
- Use retry policies for transient failures
- Implement compensation for critical operations
- Test error handling paths thoroughly

Tomorrow, we'll explore child workflows, parallel processing, and fan-out/fan-in patterns for scalable distributed systems.