Temporal Training Session 3: Activities, Retries, and Idempotency

Table of Contents

Introduction & Recap of Day 2

Topic: Activities, Retries, and Idempotency: Building Robust Execution Logic
Estimated Duration: 60-75 minutes

Good morning, everyone, and welcome to Day 3. Yesterday, we took a deep dive into the heart of Temporal: the Workflow. We established that a Workflow's primary role is to **orchestrate** business logic. We learned that to achieve the reliability and durability Temporal promises, Workflow code must be **deterministic**. This means no direct network calls, no file I/O, no random numbers, and no relying on system clocks.

So, how do we perform these essential, real-world actions? That is the entire focus of today. We move from the orchestrator—the Workflow—to the executor: the **Activity**.

Today's agenda, as outlined in our syllabus, is to cover:

- Defining and registering Activities in Python
- How to call Activities from our Workflows
- Making our Activities resilient with Retry Policies and Timeouts
- Handling long-running tasks using Heartbeats
- The critical concept of Idempotency in distributed systems
- Tying it all together with a real-world example

Defining and Calling Activities

As we covered on Day 1, an Activity is a function that performs a single, well-defined action. It's where you interact with the outside world. Let's formalize this.

What is an Activity in Temporal?

Aspect	Workflow Functions (Recap from Day 2)	Activities
Purpose	To **orchestrate** the business logic and state.	To **execute** a single, well-defined task or side effect.
Determinism	**Must be deterministic**.	**Can and should be non-deterministic**.
Interaction	**Cannot** make external calls (APIs, DBs, files).	**Should** perform all I/O and external interactions.
State	State is durably managed by Temporal's event history.	Considered stateless from the Workflow's perspective.

The golden rule remains: **Workflows orchestrate, Activities execute**.

Defining and Registering Activities in Python

Defining an Activity in Python is straightforward. It's a standard Python function that you mark with a decorator.

1. Definition (`@activity.defn`)

You create a function and decorate it with `@activity.defn`. This decorator tells the Temporal SDK that this function is intended to be run as an Activity.

```
# activities.py
from temporalio import activity
import requests

@activity.defn
async def call_api(url: str) -> dict:
    """An activity that makes a network call to a given URL."""
    try:
        activity.logger.info(f"Calling external API at: {url}")
        response = requests.get(url)
        response.raise_for_status()  # Raises an exception for bad status
codes
        return response.json()
    except requests.RequestException as e:
        activity.logger.error(f"API call failed: {e}")
        # It's often better to let the exception propagate so Temporal can
retry it.
        raise
```

2. Registration (On the Worker)

Defining the Activity isn't enough. The Worker that will execute the task needs to be aware of it. When you initialize your Worker, you provide it with a list of the Activity functions it is responsible for.

```
# from your worker runner script
from .activities import call_api
from .workflows import YourWorkflow

# ... client and worker setup ...
worker = Worker(
    client,
    task_queue="my-task-queue",
    workflows=[YourWorkflow],
    activities=[call_api],  # <-- The activity is registered here
)
```

This registration tells the Worker: "When you get a task from the my-task-queue to run an activity named call_api, this is the function you should execute."

Calling Activities from Workflows

Inside your deterministic Workflow code, you use `workflow.execute_activity()` to schedule an Activity for execution. This call is a command that gets sent to the Temporal Server, which then puts a task on the appropriate Task Queue.

```
# workflows.py
from datetime import timedelta
from temporalio import workflow
from .activities import call_api

@workflow.defn
class MyDataWorkflow:
    @workflow.run
    async def run(self, api_url: str) -> dict:
        workflow.logger.info("Workflow starting, will call an activity.")

        # This sends a command to the Temporal Server
        result = await workflow.execute_activity(
            call_api,  # The activity function to call
            api_url,   # The arguments to pass to the activity
            start_to_close_timeout=timedelta(seconds=30),
        )

        workflow.logger.info("Activity completed successfully.")
        return result
```

The `await` here is crucial. The Workflow execution effectively pauses at this point, its state preserved, until the `call_api` Activity either completes, fails, or times out. The result of the activity is then returned to the workflow to continue its logic.

Building Resilient Activities

Interacting with the outside world is messy. Networks fail, APIs are slow, and services crash. A key reason to use Temporal is to manage this messiness gracefully. You do this by configuring your Activities with timeouts and retry policies.

Activity Timeouts and Failure Handling

A timeout is a guarantee that your Workflow won't be stuck forever waiting for an unresponsive Activity. If a timeout is exceeded, the Activity execution is marked as failed, and Temporal will then consult the retry policy.

There are several key timeouts you can configure:

- **start_to_close_timeout:** This is the most common timeout. It defines the maximum time an Activity is allowed to run after a Worker has picked it up. If the Activity function itself hangs or runs too long, this timeout will trigger.
- **schedule_to_close_timeout:** This is the total time from when the Activity was scheduled by the Workflow to when it must be completed. It includes time spent waiting in the Task Queue and the execution time. This is your overall deadline for the activity.
- **schedule_to_start_timeout:** The maximum time an Activity can wait in a Task Queue before a Worker picks it up. This is useful for detecting if your Workers are overloaded or not running.
- **heartbeat_timeout:** We'll discuss this one separately in a moment.

These options are passed when you call `execute_activity`. If a timeout is breached, it raises an `ActivityError` in your Workflow, which you can handle with a `try...except` block.

Retry Policies and Backoff Strategies

When an Activity fails (either due to an unhandled exception or a timeout), you rarely want to give up immediately. You want to retry. Temporal's `RetryPolicy` gives you fine-grained control over this behavior.

```
# workflows.py
# ... imports
from temporalio.common import RetryPolicy

@workflow.defn
class MyDataWorkflow:
    @workflow.run
    async def run(self, api_url: str) -> dict:
        retry_policy = RetryPolicy(
            backoff_coefficient=2.0,     # Double the wait time after each
failure
            maximum_attempts=5,          # Try a total of 5 times
            initial_interval=timedelta(seconds=1), # Wait 1s after the
```

```
first failure
            maximum_interval=timedelta(seconds=60),# Cap the wait time at
60s
        )

        result = await workflow.execute_activity(
            call_api,
            api_url,
            start_to_close_timeout=timedelta(seconds=30),
            retry_policy=retry_policy,
        )
        return result
```

Exponential Backoff (backoff_coefficient): This is critical. It prevents you from overwhelming a struggling downstream service by increasing the delay between each retry. A coefficient of 2.0 is a common and effective choice.

Heartbeats for Long-Running Activities

What if you have an Activity that is expected to run for a long time, like processing a large video file or running a machine learning model? A `start_to_close_timeout` of 10 minutes might be too short for the task, but a timeout of 4 hours might be too long to detect if the worker has crashed.

Heartbeating is the solution. The Activity can periodically report back to the Temporal Server to say, "I'm still alive and making progress."

Configure heartbeat_timeout: You set a short timeout (e.g., 2 minutes) in your Activity options.

Call activity.heartbeat(): Inside your long-running Activity loop, you periodically call `activity.heartbeat()`.

If the Temporal Server doesn't receive a heartbeat within the `heartbeat_timeout` window, it assumes the Worker has crashed. It will then fail the Activity and schedule a retry on another Worker. This allows you to have very long-running, fault-tolerant tasks.

```python
# activities.py
import time
from temporalio import activity

@activity.defn
async def process_large_file(file_path: str):
    activity.logger.info(f"Starting to process {file_path}")
    for i in range(100):
        # Simulate doing a piece of work
        time.sleep(1)  # This would be actual processing in real code

        # Send a heartbeat every 10 iterations
        if i % 10 == 0:
            activity.heartbeat()
            activity.logger.info(f"Processed {i}% of the file")
```

```
    activity.logger.info("File processing completed")
    return {"status": "completed", "processed_chunks": 100}
```

Idempotency in Distributed Systems

Definition: An operation is idempotent if performing it multiple times has the same effect as performing it once.

Why it matters in Temporal: Because Activities can be retried, they might execute multiple times for the same logical operation. If your Activity is not idempotent, you could end up with duplicate side effects.

Examples of Non-Idempotent Operations:

- Sending an email (sends multiple emails)
- Creating a database record (creates duplicate records)
- Processing a payment (charges the customer multiple times)

Examples of Idempotent Operations:

- Setting a value (setting it to X multiple times is the same as setting it once)
- Deleting a resource (deleting it multiple times has the same effect)
- Checking a condition (reading doesn't change state)

Making Activities Idempotent:

1. **Use unique identifiers:** Pass a unique ID to your Activity and use it to prevent duplicates
2. **Check before acting:** Verify the operation hasn't already been performed
3. **Use upsert operations:** Use database operations that create or update based on existence

```
@activity.defn
async def send_notification(user_id: str, message: str, notification_id:
str):
    """Idempotent notification sending using a unique notification_id."""

    # Check if we've already sent this notification
    if await check_notification_sent(notification_id):
        activity.logger.info(f"Notification {notification_id} already sent,
skipping")
        return {"status": "already_sent"}

    # Send the notification
    await send_email(user_id, message)

    # Mark as sent
    await mark_notification_sent(notification_id)

    return {"status": "sent", "notification_id": notification_id}
```

Real-World Example

Let's tie everything together with a practical example: a loan approval workflow that demonstrates all the concepts we've covered.

```
# activities.py
from temporalio import activity
import requests
from datetime import datetime

@activity.defn
async def check_credit_score(applicant_id: str, request_id: str) -> dict:
    """Check credit score with idempotency."""

    # Check if we've already processed this request
    if await is_credit_check_completed(request_id):
        return await get_credit_check_result(request_id)

    try:
        response = requests.post(
            "https://credit-bureau-api.com/check",
            json={"applicant_id": applicant_id},
            timeout=30
        )
        response.raise_for_status()
        result = response.json()

        # Store the result with the request_id for idempotency
        await store_credit_check_result(request_id, result)
        return result

    except requests.RequestException as e:
        activity.logger.error(f"Credit check failed: {e}")
        raise

@activity.defn
async def verify_documents(applicant_id: str, document_ids: list) -> dict:
    """Verify uploaded documents."""
    activity.logger.info(f"Verifying {len(document_ids)} documents for
{applicant_id}")

    verified_count = 0
    for i, doc_id in enumerate(document_ids):
        # Simulate document verification
        await asyncio.sleep(2)

        # Send heartbeat every 5 documents
        if i % 5 == 0:
            activity.heartbeat()

        verified_count += 1
```

```
    return {"verified_documents": verified_count, "total_documents":
len(document_ids)}

@activity.defn
async def approve_loan(applicant_id: str, amount: float, request_id: str) -
> dict:
    """Approve loan with idempotency."""

    # Check if already approved
    if await is_loan_approved(request_id):
        return await get_loan_approval_result(request_id)

    # Perform approval logic
    approval_result = {
        "approved": True,
        "amount": amount,
        "interest_rate": 5.5,
        "approved_at": datetime.utcnow().isoformat()
    }

    # Store result for idempotency
    await store_loan_approval(request_id, approval_result)

    return approval_result
```

```
# workflows.py
from datetime import timedelta
from temporalio import workflow
from temporalio.common import RetryPolicy
from .activities import check_credit_score, verify_documents, approve_loan

@workflow.defn
class LoanApprovalWorkflow:
    @workflow.run
    async def run(self, applicant_id: str, amount: float, document_ids:
list) -> dict:
        workflow.logger.info(f"Starting loan approval for {applicant_id}")

        # Generate unique request ID for idempotency
        request_id =
f"loan_approval_{applicant_id}_{workflow.now().timestamp()}"

        # Configure retry policy for external API calls
        api_retry_policy = RetryPolicy(
            backoff_coefficient=2.0,
            maximum_attempts=3,
            initial_interval=timedelta(seconds=1),
            maximum_interval=timedelta(seconds=30),
        )

        # Step 1: Credit Check
```

```
        credit_result = await workflow.execute_activity(
            check_credit_score,
            applicant_id,
            request_id,
            start_to_close_timeout=timedelta(seconds=60),
            retry_policy=api_retry_policy,
        )

        if credit_result["score"] < 650:
            return {"approved": False, "reason": "Insufficient credit
score"}

        # Step 2: Document Verification
        doc_result = await workflow.execute_activity(
            verify_documents,
            applicant_id,
            document_ids,
            start_to_close_timeout=timedelta(minutes=10),
            heartbeat_timeout=timedelta(minutes=2),
        )

        if doc_result["verified_documents"] < len(document_ids):
            return {"approved": False, "reason": "Document verification
failed"}

        # Step 3: Final Approval
        approval_result = await workflow.execute_activity(
            approve_loan,
            applicant_id,
            amount,
            request_id,
            start_to_close_timeout=timedelta(seconds=30),
            retry_policy=api_retry_policy,
        )

        workflow.logger.info(f"Loan approval completed for {applicant_id}")
        return approval_result
```

This example demonstrates:

- **Idempotency:** Each activity uses a unique request_id to prevent duplicate operations
- **Retry Policies:** External API calls have retry policies with exponential backoff
- **Heartbeats:** Long-running document verification uses heartbeats
- **Timeout Configuration:** Appropriate timeouts for different types of operations
- **Error Handling:** Graceful handling of failures with proper logging

---

Summary

Today we've covered the essential building blocks for creating robust, production-ready Activities in Temporal:

1. **Activity Definition and Registration:** How to define activities and register them with workers
2. **Timeout Configuration:** Different types of timeouts and when to use them
3. **Retry Policies:** Implementing exponential backoff for resilience
4. **Heartbeats:** Managing long-running activities
5. **Idempotency:** Ensuring activities can be safely retried
6. **Real-World Integration:** Putting it all together in a practical example

These concepts form the foundation for building reliable, scalable workflows that can handle the complexities of real-world distributed systems.

Tomorrow, on Day 4, we will explore Task Queues and Workers in more detail. We'll learn how to separate different types of work, how to scale our workers to handle more load, and what the worker lifecycle looks like.