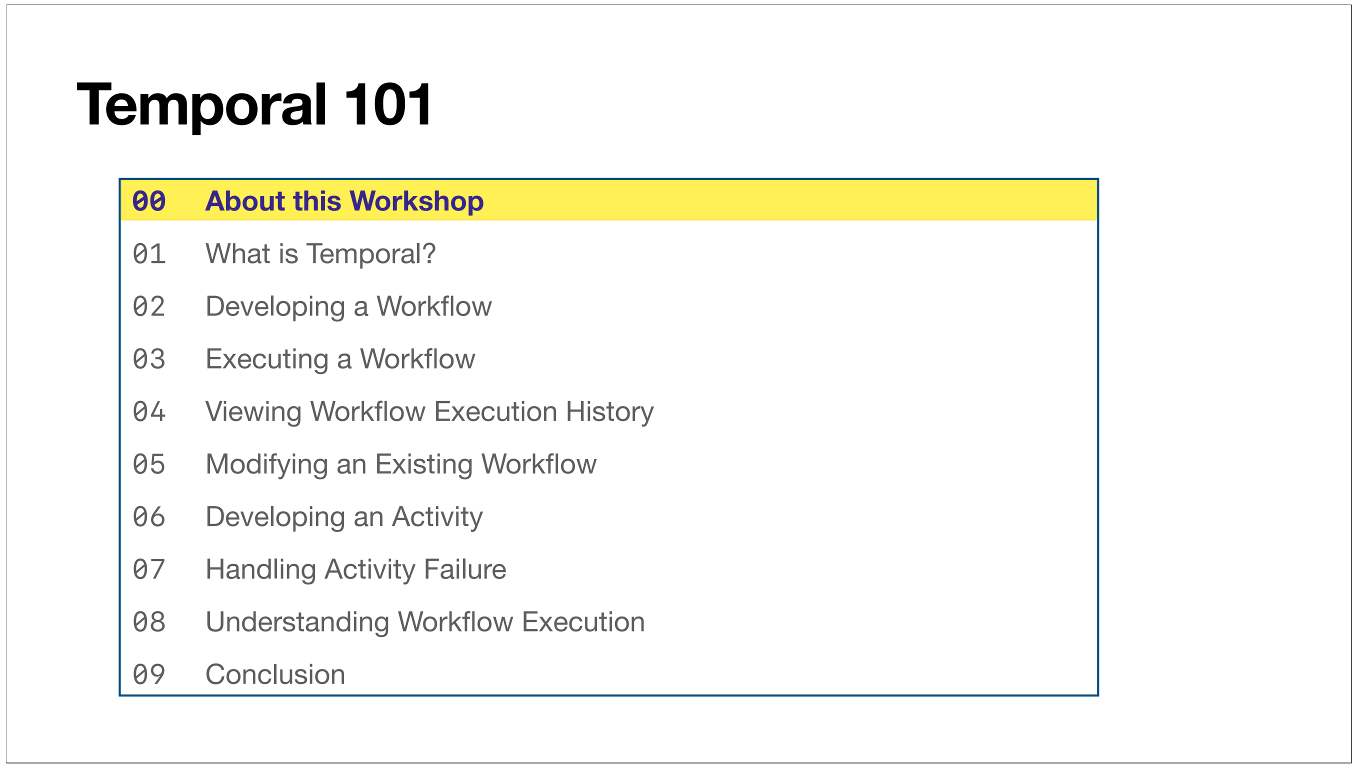
Temporal 101

Java

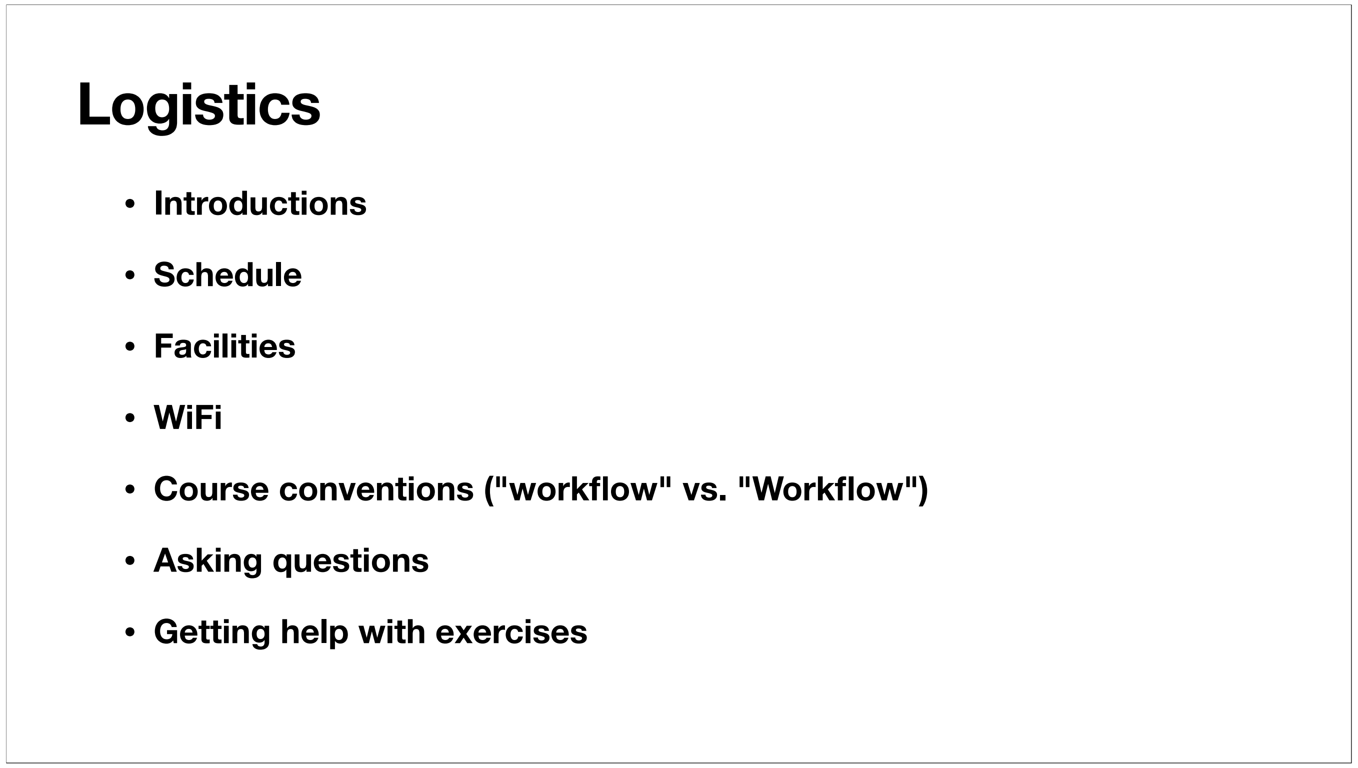
Version - TW20230831a

The "TW20230831a" update to this slide deck adds two very minor changes:
1. Slide 3 adds new bullet point for us to mention our convention of capitalizing Temporal-specific things
2. Slide 16 clarifies (via the speakers notes) that Elasticsearch is no longer required for Advanced Visibility in recent versions of Temporal Server software.



Temporal 101

--



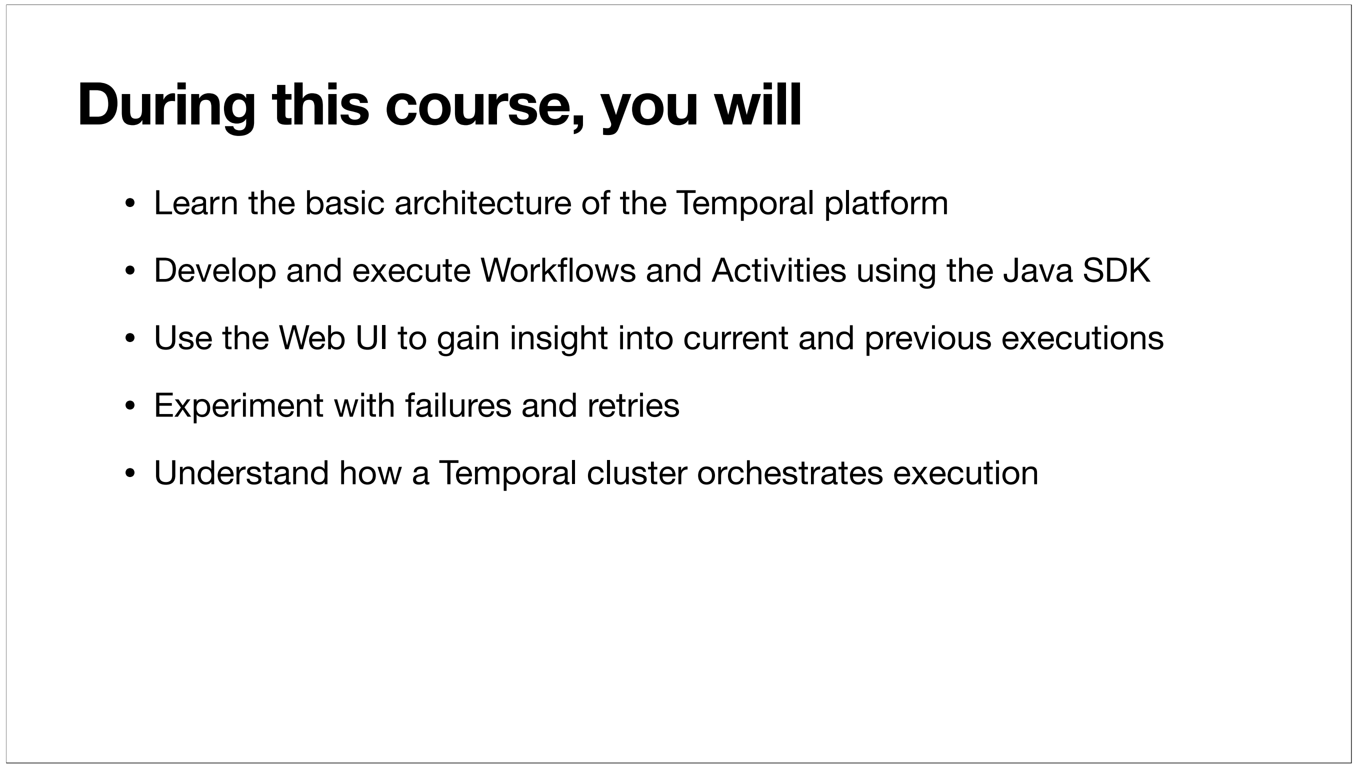
Logistics

- Introductions
- Schedule
- Facilities
- WiFi
- Course conventions ("workflow" vs. "Workflow")
- Asking questions
- Getting help with exercises

* Intro
* 3 hour course with 2 10 minute breaks

NOTE: The instructor should do an intro here, if not already done before now. After that, mention how long this session will take and when there will be breaks. Also mention that you'll need an internet connection for the hands-on exercises during this session, so please connect to the WiFi if you haven't already done so (provide the SSID and password, if applicable). Finally, explain that if they have questions during the workshop or need help with exercises, they should feel free to ask at any time -- no need to wait until the end. Finally, if this is an in-person session, mention where to find things (such as refreshments, cafeteria, restrooms, etc.).

You may notice that certain terms used in the course, such as Workflow, Activity, and Event History, are capitalized. This is a convention that we use at Temporal to help distinguish between a general concept (shown in lowercase, by convention) and a Temporal-specific implementation (shown in uppercase, by convention). For example, we would use "workflow" to describe a business process, but would use "Workflow" to describe an implementation of that business process that uses Temporal's APIs.



During this course, you will

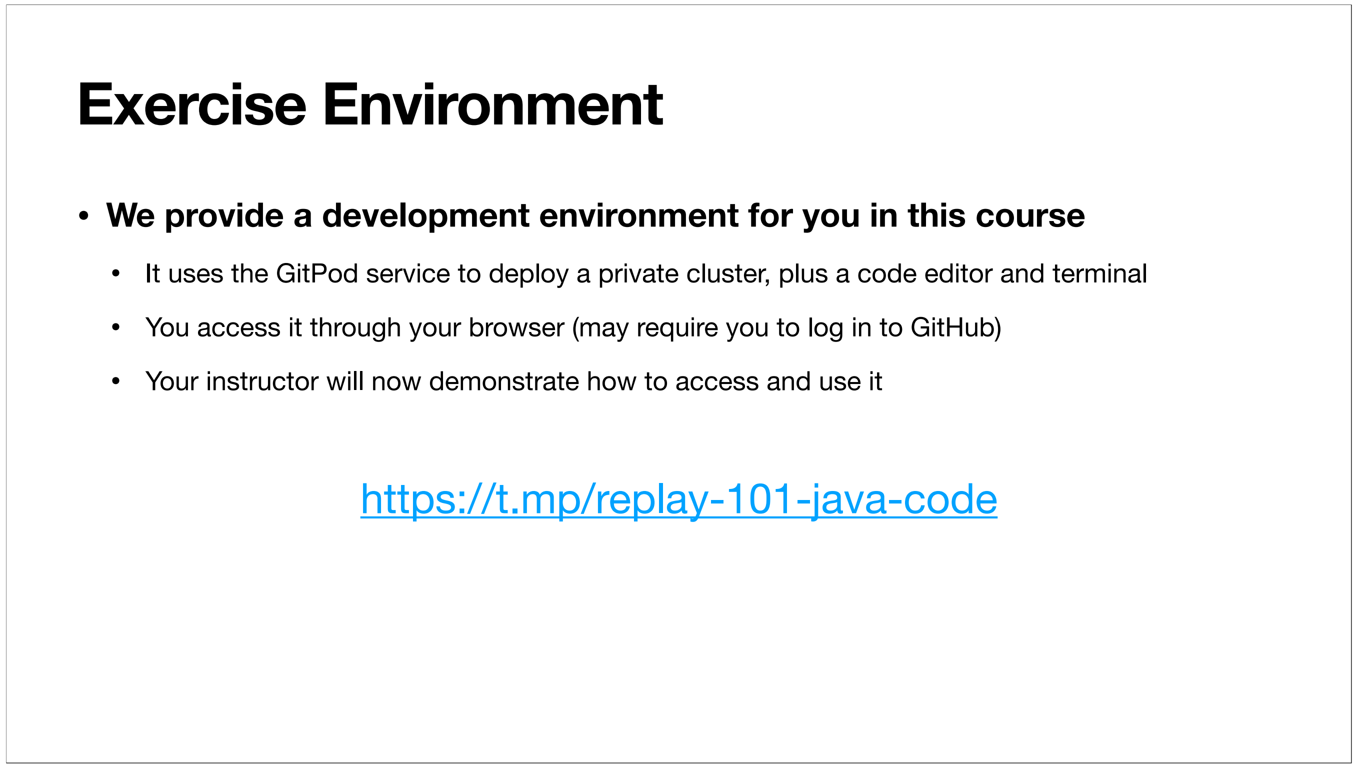
- Learn the basic architecture of the Temporal platform
- Develop and execute Workflows and Activities using the Java SDK
- Use the Web UI to gain insight into current and previous executions
- Experiment with failures and retries
- Understand how a Temporal cluster orchestrates execution

- Target audience: Experienced developers new to Temporal
- Programming language: Examples and exercises in Java
  - No need to be a Java expert
- Course competencies:
  1. Learn Temporal platform architecture
     - Set up a development cluster
  2. Develop, execute, and modify Temporal applications
  3. Find and interpret execution details
     - Troubleshoot application issues
  4. Use Temporal's command-line and web-based tools for application management

- Note on capitalization:
  - Capitalized words like "Workflow Execution" and "Event History" refer to Temporal-specific implementations.
  - Lowercase words like "workflow" denote generic concepts, e.g., business processes.

This workshop is intended for experienced developers who are relatively new to Temporal. All of the examples and exercises are in Java, but pretty much stick to the basics of that language, so you won't need to be a Java expert. The four course competencies are listed here, but I can summarize them. You'll learn the basic architecture of the Temporal platform, including how you can set up your own development cluster. You'll learn how to develop, execute, and modify a Temporal application. You'll also learn how to find and interpret details related to those executions, which will help you to find and fix problems in your application. And finally, you'll learn how to use Temporal's command-line and web-based tools for working with your applications.

I'll mention one important point before I continue. You may notice the strange capitalization of certain words, such as Workflow Execution and Event History on this slide. It's our convention to use the capitalized form when talking about the Temporal-specific implementation, rather than just a generic concept. For example, if you see "workflow" in the middle of a sentence and it has a lowercase "w," then it's discussing workflows in a general sense; for example, business processes. On the other hand, if you see "Workflow" with a capital "W" in the middle of a sentence, it means that we're talking specifically about a Temporal Workflow. This convention comes from the documentation, but we've also adopted it.



Exercise Environment

- **We provide a development environment for you in this course**
  - It uses the GitPod service to deploy a private cluster, plus a code editor and terminal
  - You access it through your browser (may require you to log in to GitHub)
  - Your instructor will now demonstrate how to access and use it

https://t.mp/replay-101-java-code

- Minimize learning barriers for Temporal
- Browser-based exercise environment via GitPod
  - Access to a private Temporal Cluster
  - Terminal windows for executing commands
  - Code editor for writing code
- Requirements:
  - Up-to-date web browser
  - GitHub account for logging in
  - Enables GitPod to set up exercise code

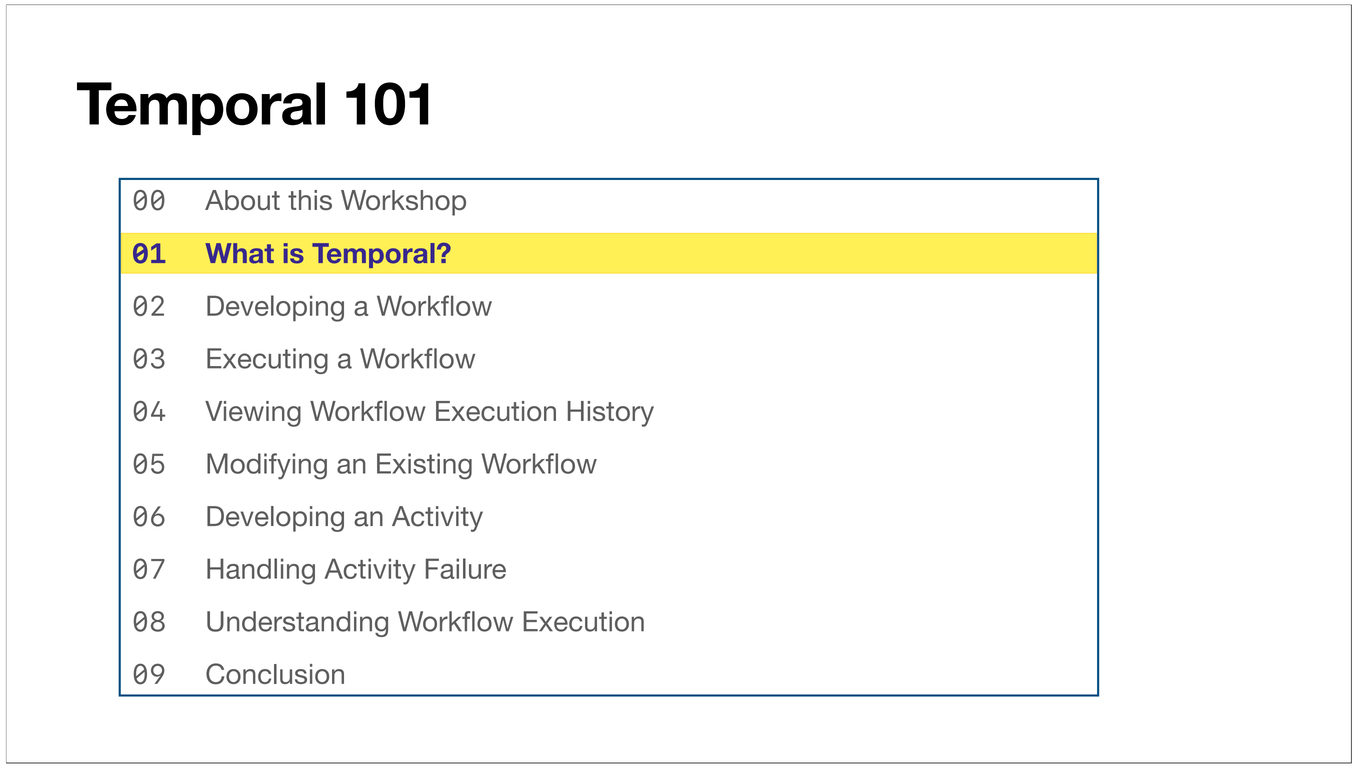
We want to minimize the barriers for learning Temporal, and we know that not everyone is able or willing to install software on their computers for a training course. That's why this course uses a browser-based exercise environment provided through GitPod. It gives you access to a private Temporal Cluster, as well as terminal windows where you can type commands and an editor you can use to write your code.

The only thing you'll need besides a up-to-date Web browser is a GitHub account that you can log into from that browser, which allows the GitPod service to set up the exercise code you'll use during the workshop.

I'll demonstrate this now.

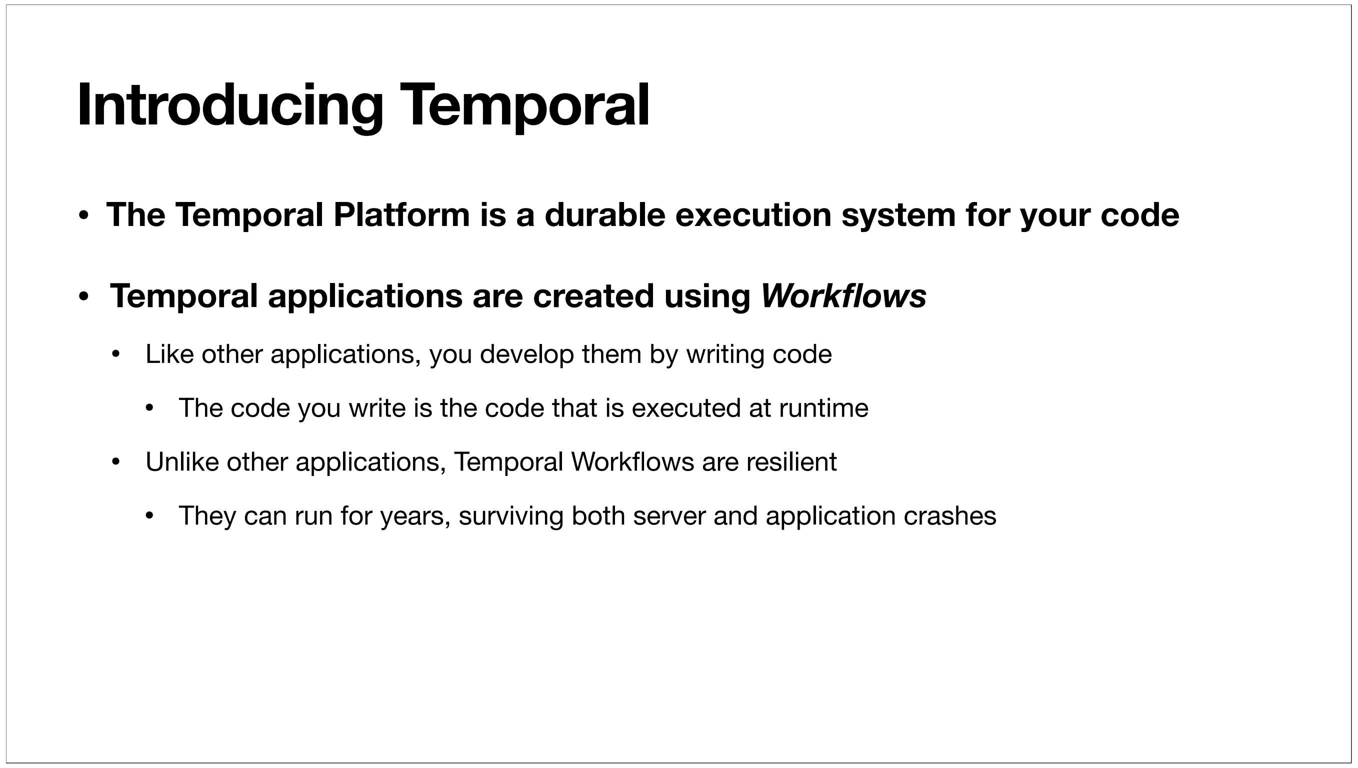
--

1. Everything that happens after clicking the link (maybe hide or obscure the address bar so we don't show the actual URL here; we might need/want to change it later)
2. Logging into GitHub (when prompted to do so)
3. Seeing GitPod do its set up (show the first 5 seconds, then speed it up until 2 seconds before the end; and mention that it takes up to two minutes for everything to come up, and if the Web UI shows an error, refresh it and all should be OK)
4. Point out the File Browser, Code Editor, Terminals (and how to create new terminals / switch between them), embedded browser, and refresh button in web UI -- warn NOT to hit the browser's refresh button)
5. Finally, mention that GitPod is free for users -- you won't be billed for using the exercise environment, but mention that it will "sleep" after being idle for ~ 30 minutes and demo what that looks like and how to wake it up again. Also mention that it is subject to deletion after a few days of non-use (best to be vague and point to GitPod's policy page that details the specific time frames)
6. Also mention how to re-open the browser window if closed (show the graphical approach). Also mention that it is port 8080 -- the 8088 is an older version of the Web UI that is now deprecated, so we won't use it in during any exercises.
7. Finally, mention that Temporal regularly releases new versions of the software, such as the web UI, and while we make every effort to update the exercise environment to use the latest software, it may happen that you see a banner in the Web UI (show demo of this) telling you that a new version is available. You can ignore this...it's just a matter of timing, as we test all exercises in the new version after it's released.



Temporal 101

--



Introducing Temporal

- **The Temporal Platform is a durable execution system for your code**
- **Temporal applications are created using *Workflows***
  - Like other applications, you develop them by writing code
    - The code you write is the code that is executed at runtime
  - Unlike other applications, Temporal Workflows are resilient
    - They can run for years, surviving both server and application crashes

- Temporal platform overview:
  - Guarantees durable execution of application code
  - Allows development without worrying about failures
  - Handles catastrophic problems (e.g., network outages, server crashes)
  - Focus on business logic, not failure recovery

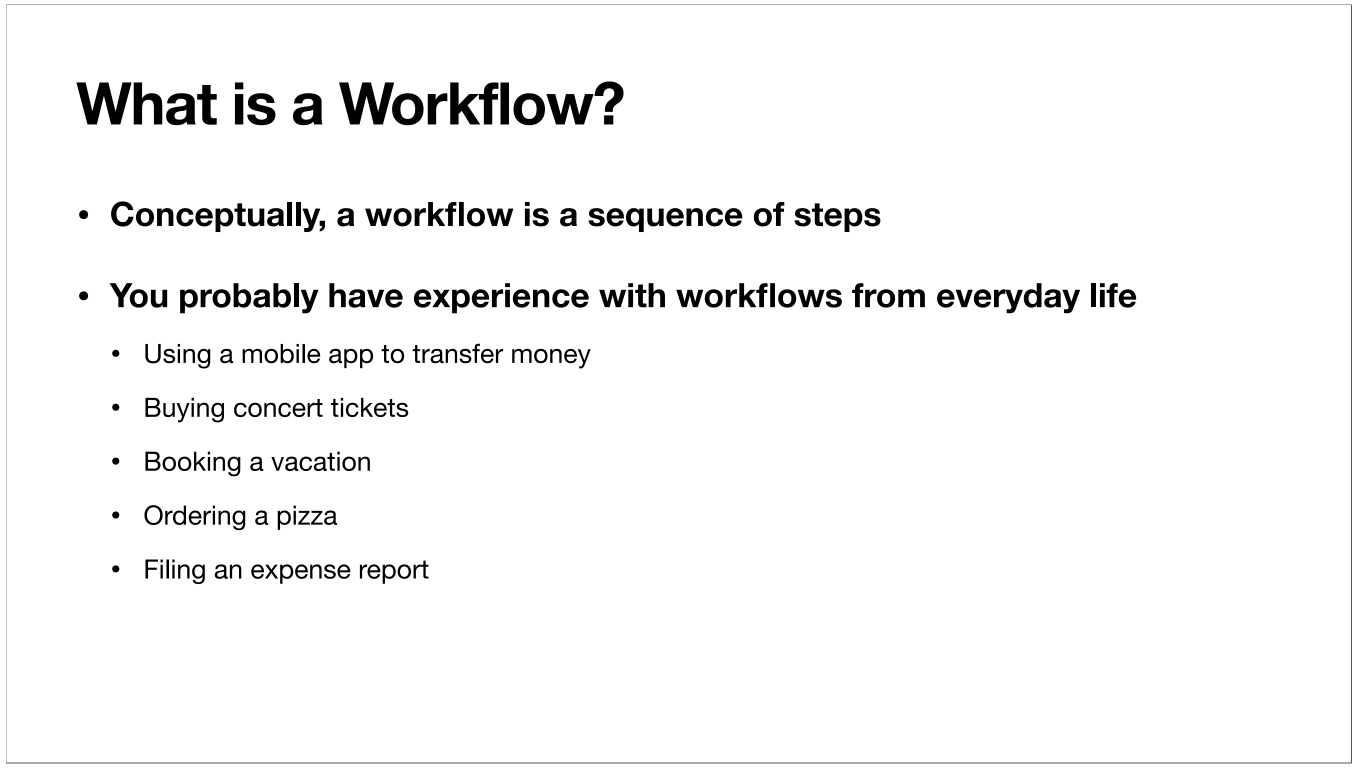
- Temporal application development:
  - Uses an abstraction called Workflows
  - Develop Workflows in programming languages like Go, Java, TypeScript, or Python
  - Same code executed at runtime
  - Utilize favorite tools and libraries for Temporal Workflows development

- Resilience of Temporal Workflows:
  - Workflows can run for years
  - Survives underlying infrastructure failures
  - Automatic recreation of pre-failure state
  - Continues execution seamlessly after application crashes

In short, Temporal is a platform that guarantees the durable execution of your application code. It allows you to develop as if failures don't even exist. Your application will run reliably even if it encounters problems that would be catastrophic for a typical application, such as network outages or server crashes. The Temporal platform handles these types of problems, allowing you to focus on the business logic, instead of writing application code to detect and recover from failures.

Temporal applications are built using an abstraction called Workflows. You'll develop those Workflows by writing code in a general purpose programming language such as Go, Java, TypeScript, or Python. This is the same code that is executed at runtime, so you can use your favorite tools and libraries to develop Temporal Workflows.

Temporal Workflows are resilient. They can run—and keeping running—for years, even if the underlying infrastructure fails. If the application itself crashes, Temporal will automatically recreate its pre-failure state so it can continue right where it left off.



What is a Workflow?

- **Conceptually, a workflow is a sequence of steps**
- **You probably have experience with workflows from everyday life**
  - Using a mobile app to transfer money
  - Buying concert tickets
  - Booking a vacation
  - Ordering a pizza
  - Filing an expense report

- Use cases for understanding workflows:
  - Mapping concepts to real-world applications
  - Workflow defines a sequence of steps

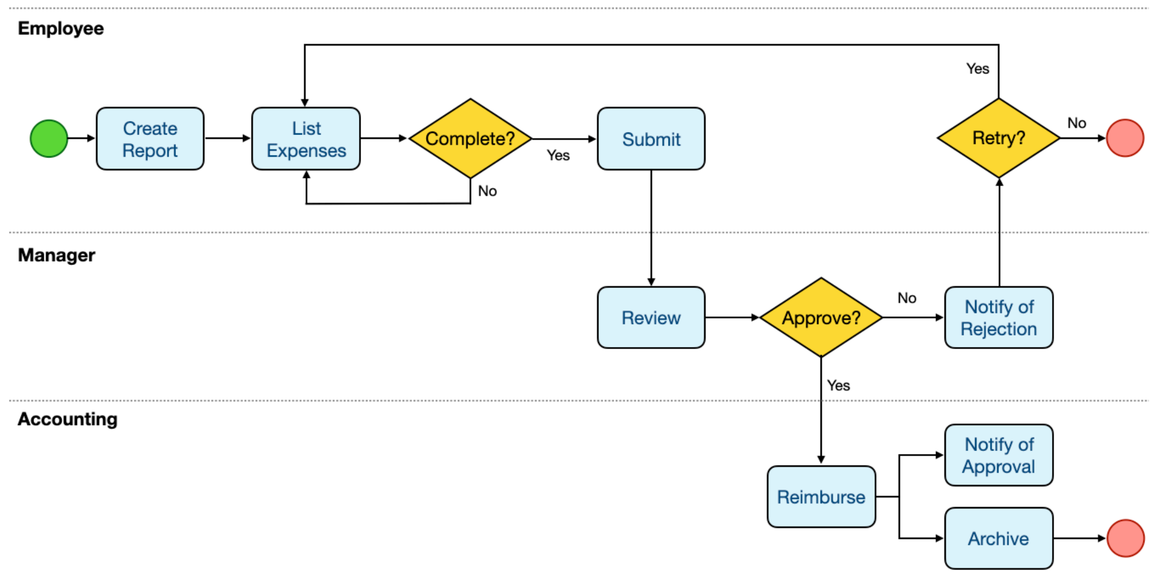
- Everyday workflow examples:
  - Subscribing to an entertainment service
  - Buying concert tickets
  - Booking a vacation
  - Ordering a pizza
  - Filing an expense report

Looking at a few potential use cases will help us map concepts to real-world applications. Conceptually, a workflow defines a sequence of steps.

You probably encounter workflows every day. Some examples might include subscribing to an entertainment service, buying concert tickets, booking a vacation, ordering a pizza, or filing an expense report.

PARTICIPATION: Ask the audience to give an example of another workflow they've encountered, ideally, one that they've been involved in implementing.

Workflow Example: Expense Report



- Examining the "filing an expense report" workflow:
  - Sequence of steps similar to other workflows
  - Steps in the workflow:
    1. Create the report
    2. Describe purchased items
    3. Attach receipts if necessary
    4. Submit the report
    5. Manager's review (reject or approve)
    6. Accounting department processing
    7. Reimbursement via check or direct deposit
    8. Notification of reimbursement
    9. Archiving the report for audit purposes

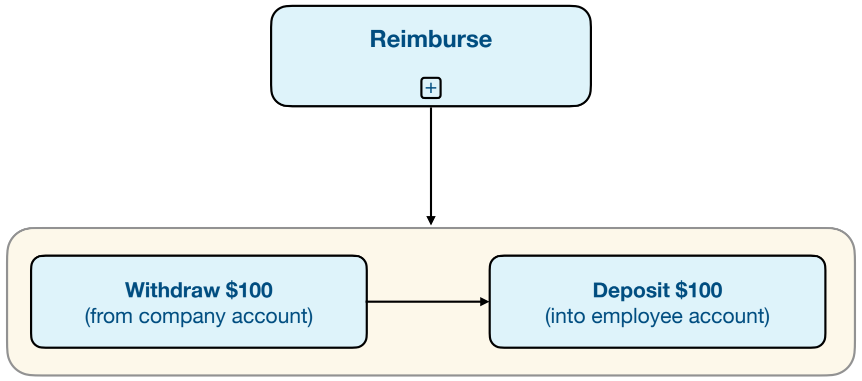
- Notable characteristics of this workflow:
  - Potentially long-running process (days, weeks, or longer)
  - Conditional logic with decision points
  - Divergent execution paths based on outcomes (acceptance or rejection)
  - Possibility of cycles (e.g., correction, resubmission, re-review)
  - Involvement of multiple human interactions (employee, manager, accounting department)
  - Integration with external systems (company's bank for reimbursement, employee's bank for funds transfer)

Let’s have a closer look at the last one: filing an expense report.

Like the other workflows, this is a sequence of steps. First, you create the report, describe the items you purchased, attach receipts if necessary, and then you submit it. The manager then reviews it, either rejecting it (in which case you'll be notified so you can fix the problem and resubmit if necessary) or approving it. If approved, the accounting department will process it, reimburse you by sending a check or direct deposit, and then let you know they've done so. They will also archive this report so it's available in case of an audit.

There are some interesting characteristics of this workflow, many of which are also present in other workflows in different domains. One of them is that it's potentially a long-running process. Depending on the organization and the number of approvals required, it may take days, weeks, or longer from start to finish. Another is that there's conditional logic; just like a computer program, there are decision points and execution paths that diverge based on the outcome: if your expense report is accepted, reimbursement is the next step, but if it's rejected, the next step is sending a notification requesting that you modify and resubmit the report. And that's another characteristic: the workflow can contain cycles; for example, because a rejected report may lead to correction, re-submission, and another review. It's also worth noting that the workflow involves multiple points of human interaction, from the employee, the manager, and the accounting department. It also involves external systems, notably the company's bank, which is the source of the reimbursement, and the employee's bank, which is the target of those funds.
--

Workflow Example: Reimbursement



Correctness requires exactly-once execution

- Composition of workflows:
  - Workflows can be composed of other workflows
  - Examining an example related to the expense report scenario

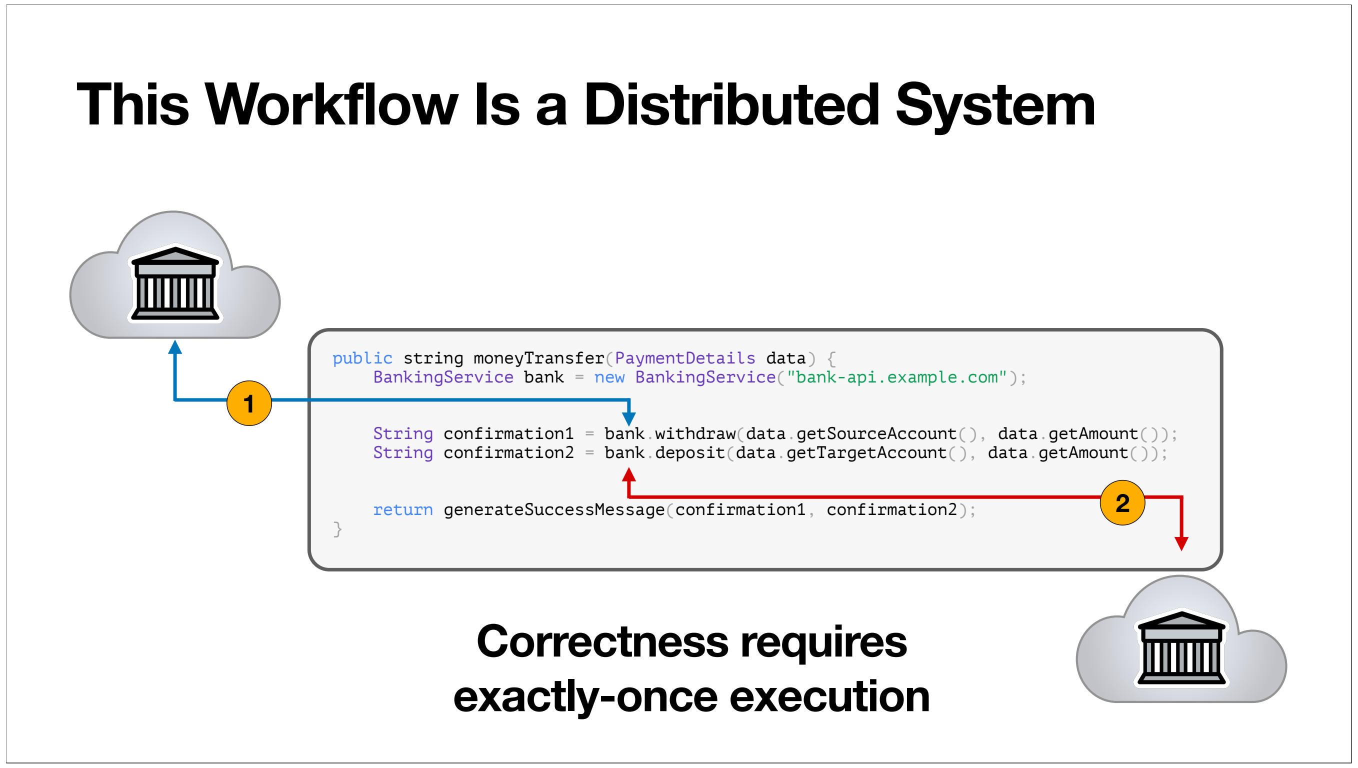
- Expense report reimbursement breakdown:
  - Reimbursement is not a single step but two distinct operations
  - Operations involved:
    1. Withdraw $100 from the employer's bank account
    2. Deposit the same amount into the employee's bank account

- Key constraints for correct execution:
  - Execution of both withdrawal and deposit steps
  - Each step must be performed exactly once

Another interesting thing to observe about a workflow is that it can be composed of other workflows. I want to explore one of those examples now.

In the case of the expense report scenario, you might think of the reimbursement as a single step, but it's actually two distinct operations. Let’s say that employee will be reimbursed $100. The first step is to withdraw the $100 from the employer’s bank account and the second is to deposit the same amount into the employee’s bank account. There are two important constraints for doing this correctly. First, we must execute *both* the withdrawal and the deposit. Second, we must do each of them *exactly once*.

--



This Workflow Is a Distributed System

```
public string moneyTransfer(PaymentDetails data) {
    BankingService bank = new BankingService("bank-api.example.com");

    String confirmation1 = bank.withdraw(data.getSourceAccount(), data.getAmount());
    String confirmation2 = bank.deposit(data.getTargetAccount(), data.getAmount());

    return generateSuccessMessage(confirmation1, confirmation2);
}
```

Correctness requires exactly-once execution

- Broad perspective on reimbursement workflow:
  - In essence, it's a money transfer between two accounts
  - Widely used in services like Square, Stripe, Western Union, PayPal, Venmo, and Apple Pay

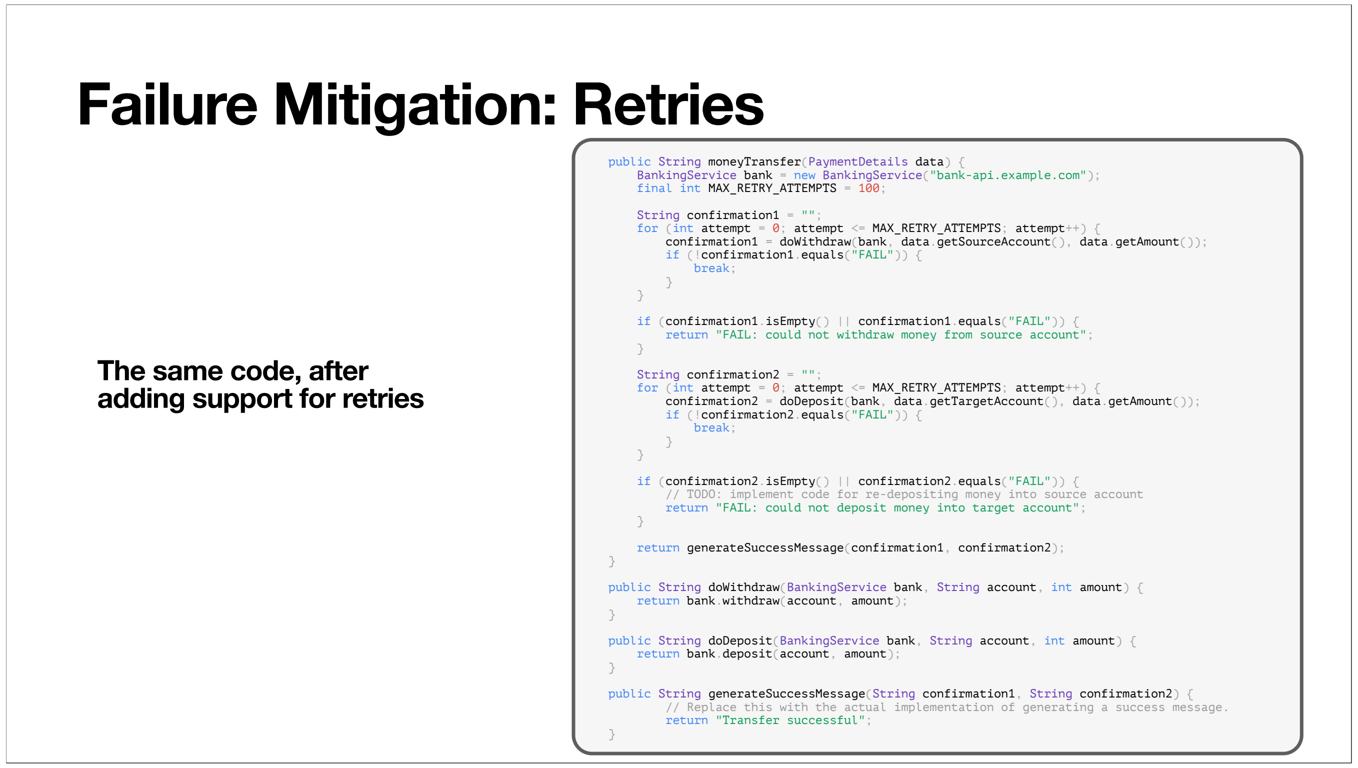
- Core business logic:
  - Withdraw money from one account
  - Deposit it into another account
  - Provide confirmation of successful completion

- Distributed system nature:
  - Involves multiple accounts accessed through remote procedure calls
  - Potential for various failure scenarios

- Developer's responsibility:
  - Detect and mitigate failures, especially in cases like a failure between withdrawal and deposit operations
---
More broadly, the reimbursement is just a transfer of money between two accounts. There are plenty of other use cases for this same workflow. In fact, millions of people every day depend on it when they use services like Square, Stripe, Western Union, PayPal, Venmo, and Apple Pay.

At its core, the business logic involves withdrawing money from one account, depositing it into another account, and providing some confirmation that it was successfully

completed. And because it involves multiple accounts that are accessed through some type of remote procedure calls, this is a distributed system. Although this example only involves a few lines of code, it can fail in a variety of ways. Imagine, for example, how much of a problem it would be if there was a failure right in the middle of this method, between the withdrawal and deposit. As a developer, it's up to you to detect and mitigate those failures.
--



Failure Mitigation: Retries

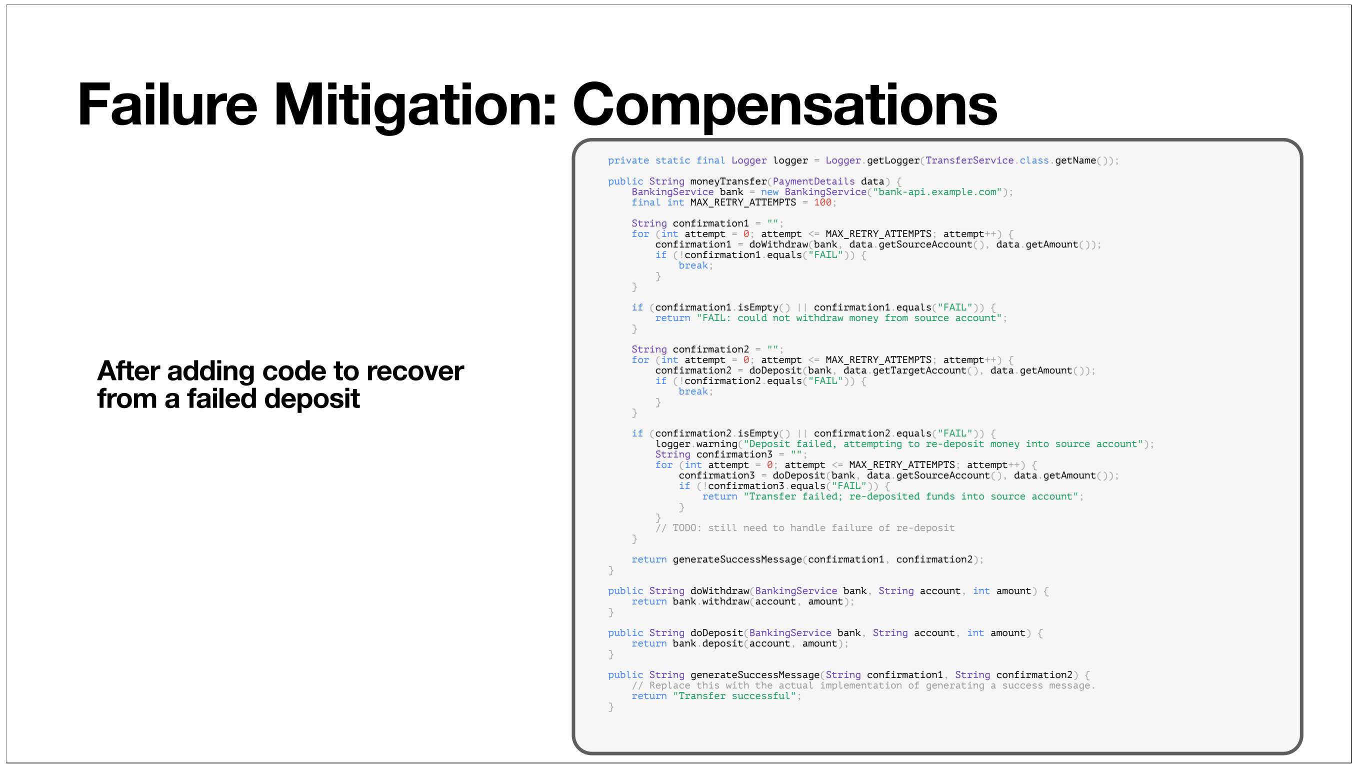
The same code, after adding support for retries

```java
public String moneyTransfer(PaymentDetails data) {
    BankingService bank = new BankingService("bank-api.example.com");
    final int MAX_RETRY_ATTEMPTS = 100;

    String confirmation1 = "";
    for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
        confirmation1 = doWithdraw(bank, data.getSourceAccount(), data.getAmount());
        if (!confirmation1.equals("FAIL")) {
            break;
        }
    }

    if (confirmation1.isEmpty() || confirmation1.equals("FAIL")) {
        return "FAIL: could not withdraw money from source account";
    }

    String confirmation2 = "";
    for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
        confirmation2 = doDeposit(bank, data.getTargetAccount(), data.getAmount());
        if (!confirmation2.equals("FAIL")) {
            break;
        }
    }

    if (confirmation2.isEmpty() || confirmation2.equals("FAIL")) {
        // TODO: implement code for re-depositing money into source account
        return "FAIL: could not deposit money into target account";
    }

    return generateSuccessMessage(confirmation1, confirmation2);
}

public String doWithdraw(BankingService bank, String account, int amount) {
    return bank.withdraw(account, amount);
}

public String doDeposit(BankingService bank, String account, int amount) {
    return bank.deposit(account, amount);
}

public String generateSuccessMessage(String confirmation1, String confirmation2) {
        // Replace this with the actual implementation of generating a success message.
        return "Transfer successful";
}
```

You might begin by adding support for retries, so you can survive an intermittent failure with the remote service. You may have written code like this before.

--

NOTE: The font is intentionally small on this slide and the two that follow. It's not important to read the code, but simply to show the relative increase in the amount of code we've written as we try to address each problem.



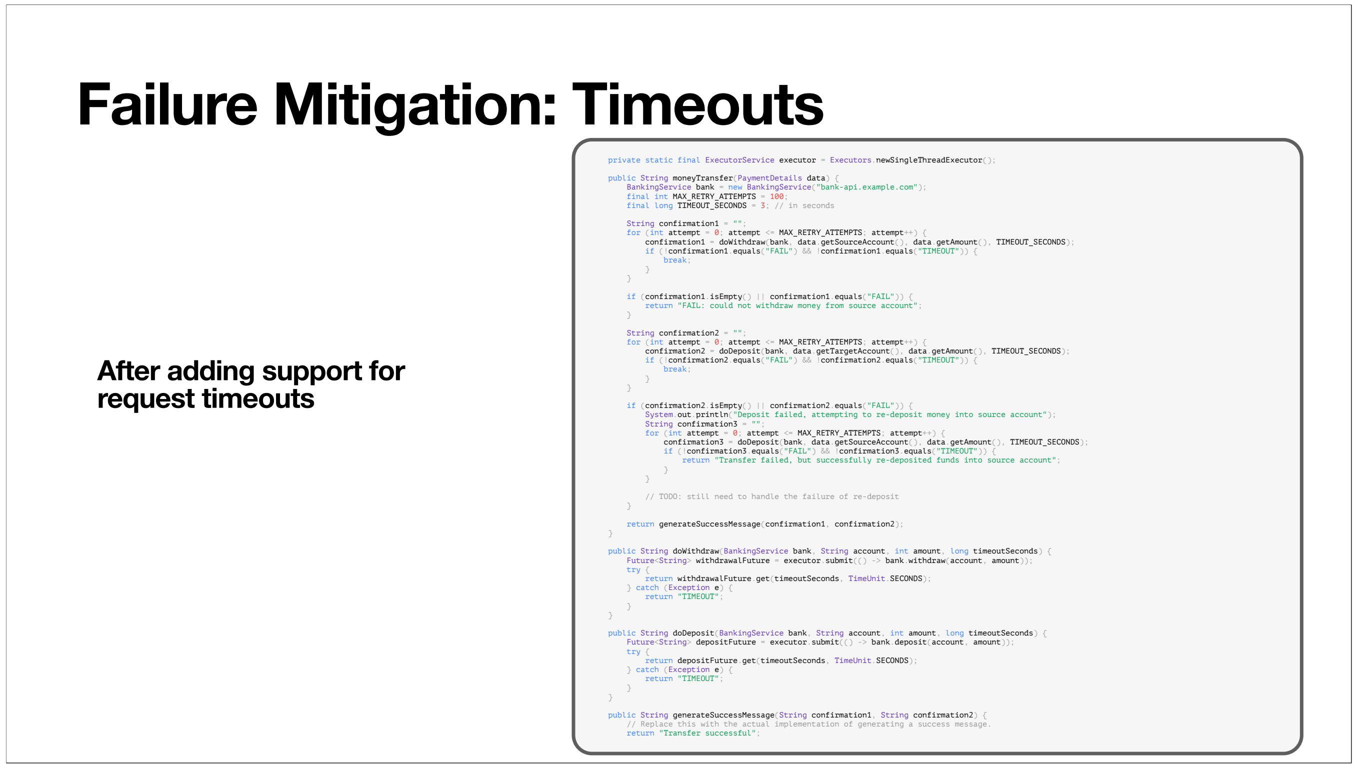
Failure Mitigation: Compensations

After adding code to recover from a failed deposit

```java
private static final Logger logger = Logger.getLogger(TransferService.class.getName());

public String moneyTransfer(PaymentDetails data) {
    BankingService bank = new BankingService("bank-api.example.com");
    final int MAX_RETRY_ATTEMPTS = 100;

    String confirmation1 = "";
    for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
        confirmation1 = doWithdraw(bank, data.getSourceAccount(), data.getAmount());
        if (!confirmation1.equals("FAIL")) {
            break;
        }
    }

    if (confirmation1.isEmpty() || confirmation1.equals("FAIL")) {
        return "FAIL: could not withdraw money from source account";
    }

    String confirmation2 = "";
    for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
        confirmation2 = doDeposit(bank, data.getTargetAccount(), data.getAmount());
        if (!confirmation2.equals("FAIL")) {
            break;
        }
    }

    if (confirmation2.isEmpty() || confirmation2.equals("FAIL")) {
        logger.warning("Deposit failed, attempting to re-deposit money into source account");
        String confirmation3 = "";
        for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
            confirmation3 = doDeposit(bank, data.getSourceAccount(), data.getAmount());
            if (!confirmation3.equals("FAIL")) {
                return "Transfer failed; re-deposited funds into source account";
            }
        }
        // TODO: still need to handle failure of re-deposit
    }

    return generateSuccessMessage(confirmation1, confirmation2);
}

public String doWithdraw(BankingService bank, String account, int amount) {
    return bank.withdraw(account, amount);
}

public String doDeposit(BankingService bank, String account, int amount) {
    return bank.deposit(account, amount);
}

public String generateSuccessMessage(String confirmation1, String confirmation2) {
    // Replace this with the actual implementation of generating a success message.
    return "Transfer successful";
}
```

And if you reach the maximum number of retries and the deposit still doesn't go through, you might compensate by writing more code to returns the withdrawn amount back to the original account.

--



Failure Mitigation: Timeouts

After adding support for request timeouts

```
private static final ExecutorService executor = Executors.newSingleThreadExecutor();

public String moneyTransfer(PaymentDetails data) {
    BankingService bank = new BankingService("bank-api.example.com");
    final int MAX_RETRY_ATTEMPTS = 100;
    final long TIMEOUT_SECONDS = 3; // in seconds

    String confirmation1 = "";
    for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
        confirmation1 = doWithdraw(bank, data.getSourceAccount(), data.getAmount(), TIMEOUT_SECONDS);
        if (!confirmation1.equals("FAIL") && !confirmation1.equals("TIMEOUT")) {
            break;
        }
    }

    if (confirmation1.isEmpty() || confirmation1.equals("FAIL")) {
        return "FAIL: could not withdraw money from source account";
    }

    String confirmation2 = "";
    for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
        confirmation2 = doDeposit(bank, data.getTargetAccount(), data.getAmount(), TIMEOUT_SECONDS);
        if (!confirmation2.equals("FAIL") && !confirmation2.equals("TIMEOUT")) {
            break;
        }
    }

    if (confirmation2.isEmpty() || confirmation2.equals("FAIL")) {
        System.out.println("Deposit failed, attempting to re-deposit money into source account");
        String confirmation3 = "";
        for (int attempt = 0; attempt <= MAX_RETRY_ATTEMPTS; attempt++) {
            confirmation3 = doDeposit(bank, data.getSourceAccount(), data.getAmount(), TIMEOUT_SECONDS);
            if (!confirmation3.equals("FAIL") && !confirmation3.equals("TIMEOUT")) {
                return "Transfer failed, but successfully re-deposited funds into source account";
            }
        }

        // TODO: still need to handle the failure of re-deposit
    }

    return generateSuccessMessage(confirmation1, confirmation2);
}

public String doWithdraw(BankingService bank, String account, int amount, long timeoutSeconds) {
    Future<String> withdrawalFuture = executor.submit(() -> bank.withdraw(account, amount));
    try {
        return withdrawalFuture.get(timeoutSeconds, TimeUnit.SECONDS);
    } catch (Exception e) {
        return "TIMEOUT";
    }
}

public String doDeposit(BankingService bank, String account, int amount, long timeoutSeconds) {
    Future<String> depositFuture = executor.submit(() -> bank.deposit(account, amount));
    try {
        return depositFuture.get(timeoutSeconds, TimeUnit.SECONDS);
    } catch (Exception e) {
        return "TIMEOUT";
    }
}

public String generateSuccessMessage(String confirmation1, String confirmation2) {
    // Replace this with the actual implementation of generating a success message.
    return "Transfer successful";
```

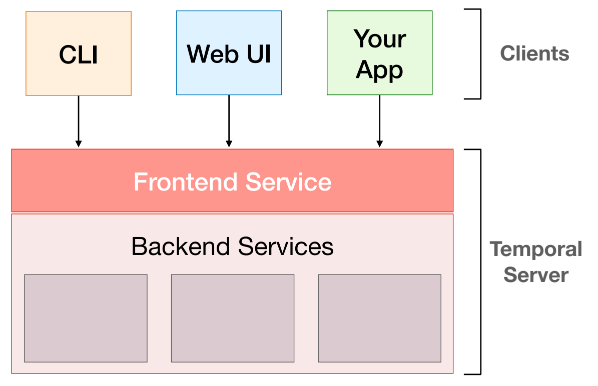
Eventually, you'll probably find out that the program gets stuck waiting for a response, so you add support for timing out requests that take too long.

Compared to where we started, the code is now harder to understand and more difficult to maintain. As a developer, maybe this feels familiar to you. It certainly feels familiar to me! Fortunately, there is a solution...

--

Architectural Overview: Temporal Server

- **Consists of multiple services**
  - Each service is horizontally scalable
  - The frontend service is an API gateway
  - Clients are external to the server and interact only with the frontend service



- Understanding Temporal's architecture basics:
  - Recommended for developers to grasp its fundamentals
  - Temporal can refer to both the software and the company

- Two parts of the Temporal Platform:
  1. Server:
     - Comprises frontend and several backend services
     - Manages execution of application code
     - Horizontally scalable, independent services
     - Production environment deploys multiple instances on multiple machines for performance and availability

  2. Clients:
     - External to the Temporal Server but communicate with it
     - Three main clients used in this course:
       - Command-line interface for server interaction
       - Web-based user interface (Web UI) for monitoring running workflows
       - Application itself, acting as a client
     - All clients communicate with the cluster via the frontend service
     - The frontend service serves as an API gateway and is distinct from the Web UI.

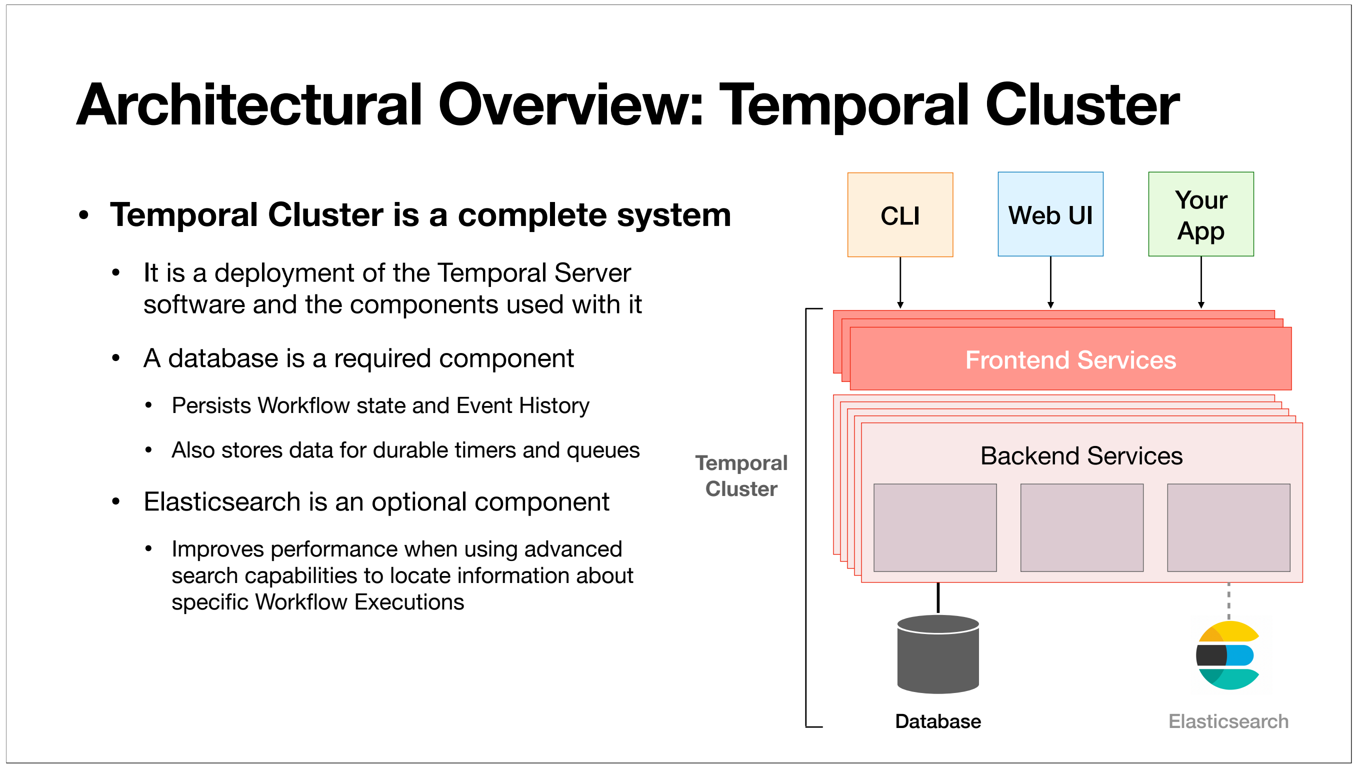
Although this training is meant for developers, I think it's helpful to have at least a basic understanding of Temporal's architecture. The name "Temporal" can be ambiguous because it's the name of the software and also the name of the company that was founded by its creators and which employs many of the engineers who work on it. However, when someone mentions Temporal without additional context, they're probably referring to the *Temporal Platform*.

I think it's best to think of the Temporal Platform as having two parts, much like the World Wide Web has two parts.

On one side, you have the server. The Temporal Server consists of a frontend service, plus several backend services that work together to manage the execution of your application code. All of those services are horizontally scalable, independently from one another, and a production environment will typically run multiple instances of each, deployed across multiple machines, to increase performance and availability.

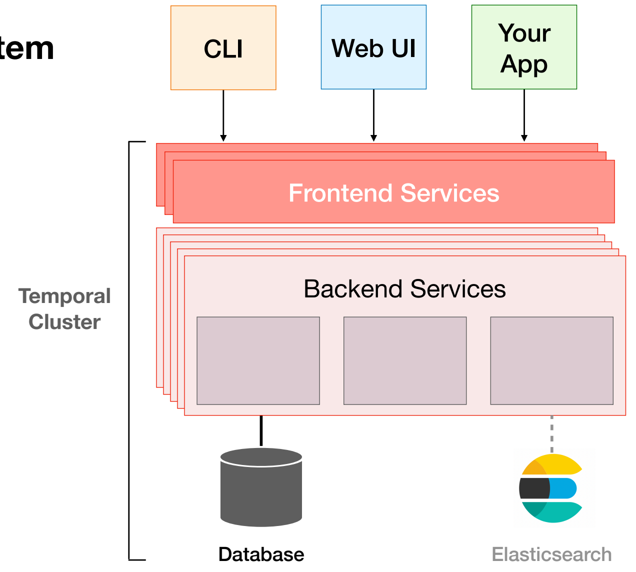
On the other side, you have the clients, which communicate with the Temporal Server, but are external to it. We'll work with at least three clients in this course. One is a command-line interface that you'll use to interact with the server; for example, to start a workflow. Another is a web-based user interface, which we call the Web UI, that displays information about workflows that are currently running as well as those that have recently completed. During this course, you'll see firsthand why this is such an amazing tool for gaining insight into your applications. Finally, your application is also a client, or more precisely, your application will *contain a client* and will make API calls that ultimately result in communication with the cluster. Regardless of which type of client we're talking about -- the command-line tool, the Web UI, or your application, they all communicate with the cluster through that frontend service. In other words, the frontend service is an API gateway, and is completely distinct from the Web UI.
--

NOTE: I have intentionally omitted the names of the services, except for the frontend, from both the diagram and the explanation. They are not relevant to this audience at this point and I think that even mentioning them could cause confusion (in particular, they may confuse the Worker Service with Workers in the application and then conclude that Temporal Server runs their code).



Architectural Overview: Temporal Cluster

- **Temporal Cluster is a complete system**
  - It is a deployment of the Temporal Server software and the components used with it
  - A database is a required component
    - Persists Workflow state and Event History
    - Also stores data for durable timers and queues
  - Elasticsearch is an optional component
    - Improves performance when using advanced search capabilities to locate information about specific Workflow Executions



- Mainly of interest to engineers working on the platform itself
  - Analogous to the CPU in a computer or the engine in a car
  - Essential part of the system but not inherently useful on its own

- The complete Temporal system:
  - Known as the Temporal Cluster
  - Deployment of Temporal Server software on multiple machines
  - Includes additional external components
  - Components:
    1. Required: Database (e.g., Cassandra, PostgreSQL, MySQL)
       - Tracks workflow states and event history
       - Persists information, including durable timers and queues
    2. Optional: Elasticsearch
       - No longer mandatory for advanced visibility
       - Recommended for performance in self-hosted production clusters
       - Enables advanced searching, sorting, and filtering capabilities
       - Eases the database load for specific searches

- Additional tools used with Temporal:
  - Prometheus: Collects metrics from Temporal

  - Grafana: Creates dashboards based on collected metrics
  - Aid operations teams in monitoring clusters and applications.

The Temporal Server is mainly of interest to the engineers who are working on the platform itself. Like the CPU in a computer or the engine in a car, it's an essential part of the overall system, but isn't necessarily that useful by itself. The complete system in Temporal is known as the Temporal Cluster, which is a deployment of the Temporal Server software on some number of machines, plus the additional components used with it. There are currently two such components.

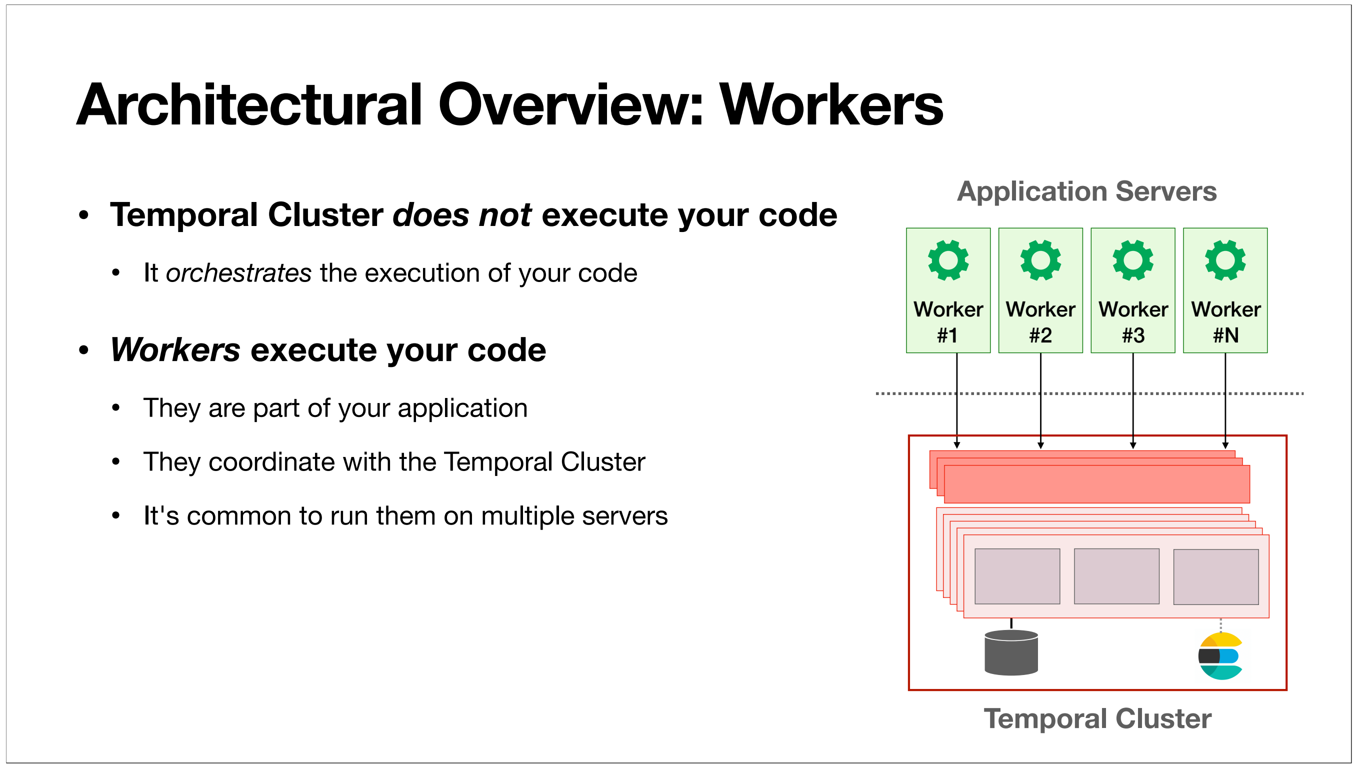
The required one is a database, which could be Cassandra, PostgreSQL, or MySQL. The Temporal Cluster tracks the current state of every workflow you run, and also maintains a history of all events that occur during the Execution of each Workflow, which it uses to reconstruct the current state in case of failure. It persists this and other information, such as details related to durable timers and queues, to the database.

The optional one is Elasticsearch. This used to be required for enabling advanced visibility, but this dependency was eliminated as a requirement and recent versions of Temporal Server software support advanced visibility without Elasticsearch. However, Elasticsearch is still recommended for self-hosted clusters in production because it will improve performance (i.e., because it handles searches for Workflow Executions that would otherwise be performed by querying the database).

It's not necessary for basic operation, but adding it will give you advanced searching, sorting, and filtering capabilities for information about current and recent Workflow Executions. This is helpful when you run Workflows millions of times and want to locate a specific one; for example, based on when it started, how long it took to run, or what is final status was. Adding Elasticsearch to the cluster can also improve performance because it frees the database from having to handle these types of searches.

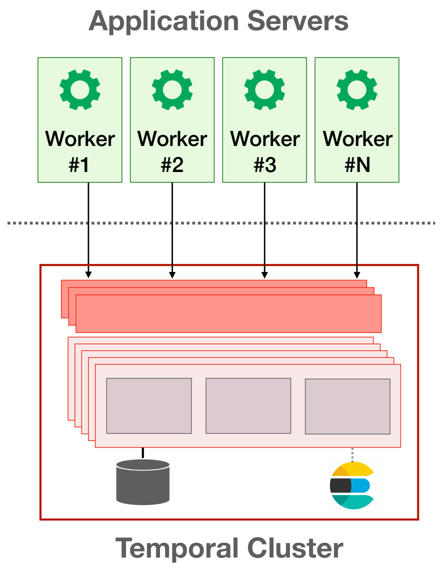
There are two other tools often used with Temporal that are worth mentioning: Prometheus, for collecting metrics from Temporal, and Grafana, for creating dashboards based on those metrics. Together, these tools help operations teams monitor cluster and applications.

--



Architectural Overview: Workers

- **Temporal Cluster *does not* execute your code**
  - It *orchestrates* the execution of your code
- ***Workers* execute your code**
  - They are part of your application
  - They coordinate with the Temporal Cluster
  - It's common to run them on multiple servers



- Execution of application code in the Temporal Cluster:
  - Contrary to expectations, the Temporal Cluster doesn't directly execute your code
  - Achieves code reliability through orchestration
  - Application code execution is external to the cluster
  - Typically occurs on separate servers, possibly in different data centers

- Entity responsible for code execution:
  - The Worker
  - Common to run Workers on multiple servers for scalability and availability

- Role of the application:
  - Contains code, usually a few dozen lines, to initialize the Worker
  - Worker contains a Temporal Client to communicate with the Temporal Cluster and coordinate Workflows

- Runtime requirements:
  - Application includes Worker initialization code, business logic methods, and potentially Workflow-related code
  - Dependencies required for application execution
  - All needed on machines running at least one Worker process
  - Temporal uses gRPC for communication, so each machine with a Worker needs connectivity to the frontend service on the Temporal Cluster

- Temporal frontend service specifics:
  - Listens on TCP port 7233 by default

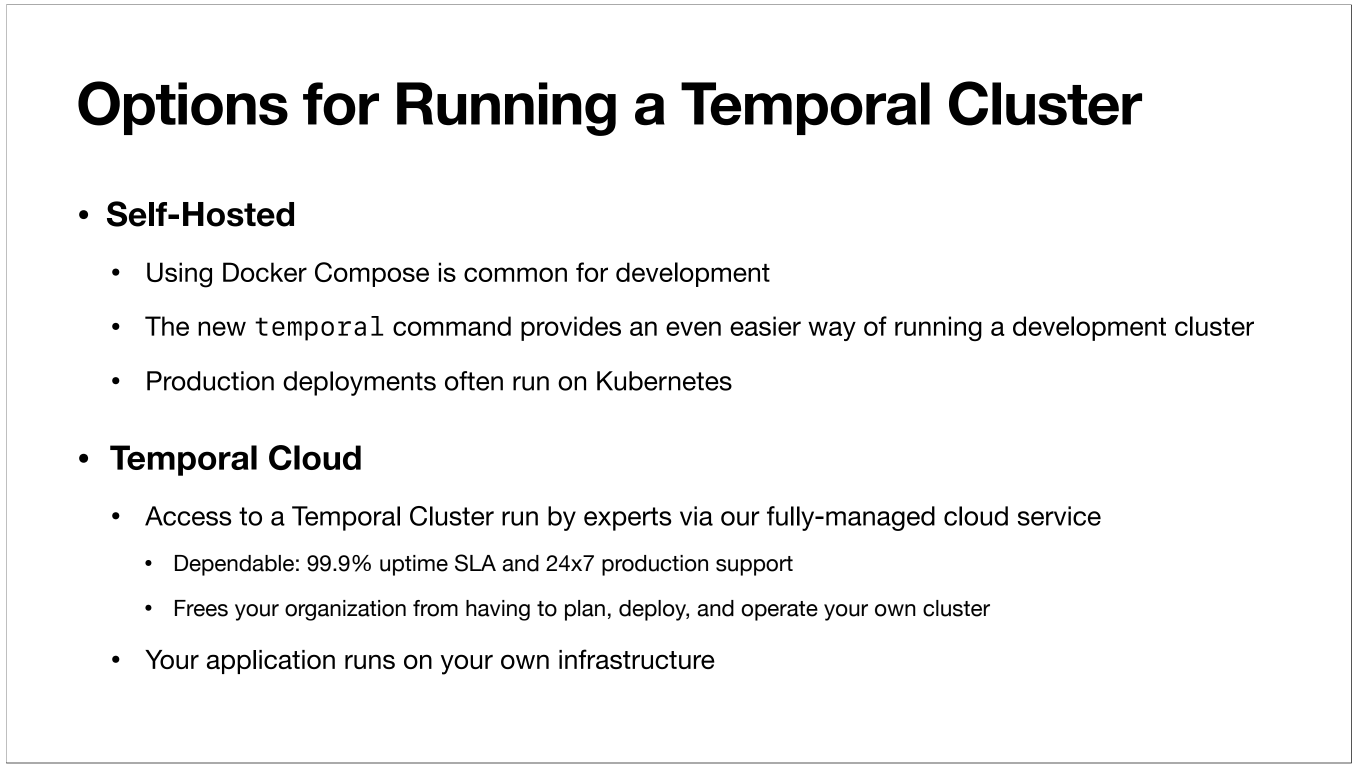
One thing that people new to Temporal may find surprising is that the Temporal Cluster does not execute your code. That may sound strange, given that the platform guarantees that your code executes reliably, but it achieves this through *orchestration*. The execution of your application code is external to the cluster, and in typical deployments, happens on a separate set of servers, potentially even in a different data center than the Temporal Cluster.

The entity responsible for executing your application code is known as a Worker, and it's common to run Workers on multiple servers, since this increases both the scalability and availability of your application.

Your application will contain some code, usually no more than a few dozen lines, that initialize the Worker. Remember a moment ago when I said that your application will contain a Temporal Client? This is what I was talking about. The Worker contains a Temporal Client, which it uses to communicate with the Temporal Cluster and coordinate the execution of your Workflows.

Since this diagram depicts both the application and the cluster that orchestrates its execution, it provides the opportunity to mention what's required at runtime. The application will contain the code used to initialize the Worker, the methods that comprise your business logic, and possibly also code used to start or check the status of the Workflow. At runtime, you'll need everything needed to execute the application, which will include any dependencies, on each machine where at least one Worker process will run. Temporal uses gRPC for communication, so each machine running a Worker will require connectivity to the frontend service on the Temporal cluster.

--
The Temporal frontend service listens on TCP port 7233, by default.



Options for Running a Temporal Cluster

- **Self-Hosted**
  - Using Docker Compose is common for development
  - The new `temporal` command provides an even easier way of running a development cluster
  - Production deployments often run on Kubernetes
- **Temporal Cloud**
  - Access to a Temporal Cluster run by experts via our fully-managed cloud service
    - Dependable: 99.9% uptime SLA and 24x7 production support
    - Frees your organization from having to plan, deploy, and operate your own cluster
  - Your application runs on your own infrastructure

- Temporal Cluster deployment options:
  - Categorized into two main groups: self-hosted or Temporal-managed

- Self-hosted options:
  1. Docker Compose:
     - Easily run all necessary components, including the database and optional Elasticsearch
     - Convenient for development, eliminates manual installation and configuration
  2. Kubernetes:
     - Popular for self-hosted production clusters
     - Documentation provides insights into various deployment scenarios

- Temporal CLI option:
  - Provides a small Temporal Cluster running in a single process
  - No external runtime dependencies
  - Simpler and less resource-intensive than Docker Compose

- Alternative: Temporal Cloud
  - Fully-managed cloud service operated by Temporal experts
  - Access to a production-ready Temporal Cluster with support
  - Removes operational workload (planning, deploying, updating, monitoring, scaling) for your organization

- Key point:
  - Regardless of deployment choice (self-hosted or Temporal Cloud), your application, including code and executing Workers, runs on infrastructure under your control
  - Infrastructure may be servers in your datacenter or virtual machines hosted by a cloud provider
  - Temporal does not execute or access your code.

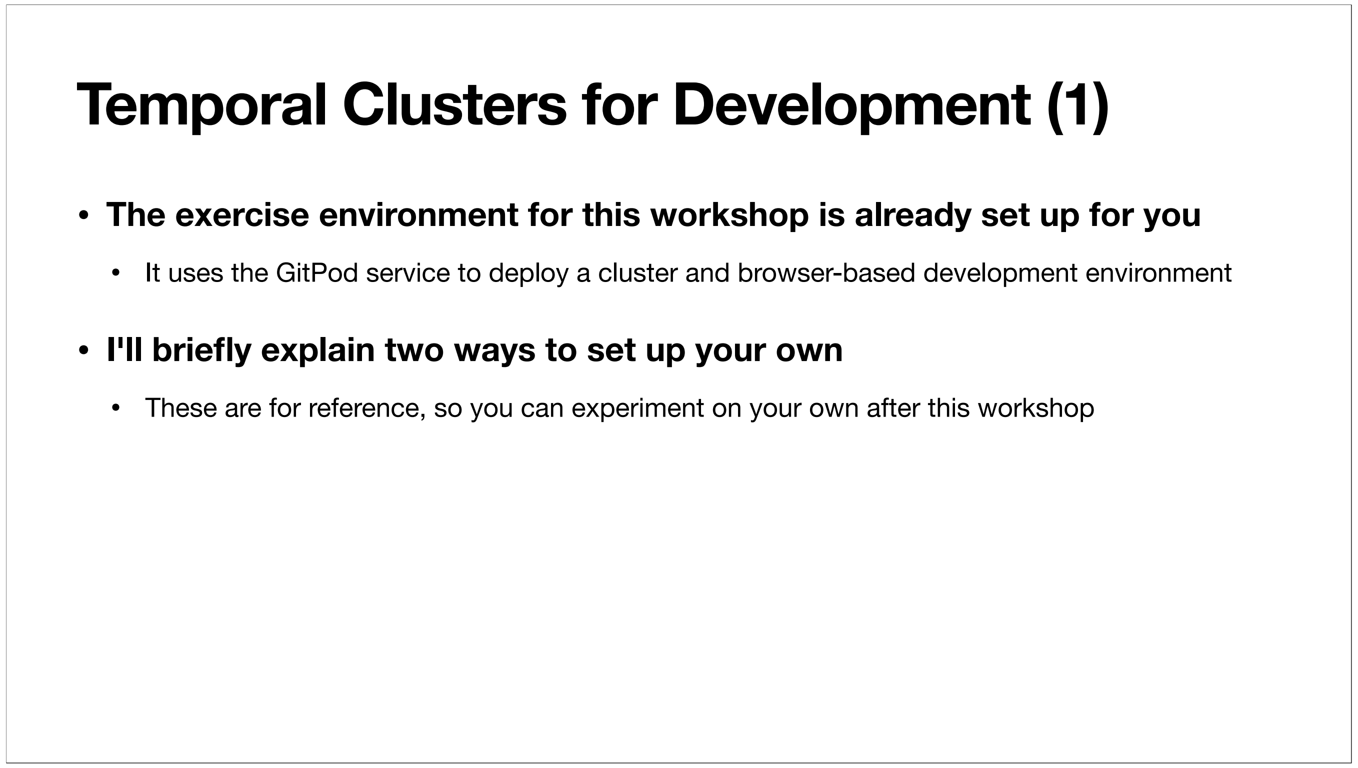
There are lots of ways to run a Temporal Cluster, but we can group them all into two categories: host it yourself or let Temporal do it for you.

One self-hosted option is Docker Compose, which lets you easily run everything you need -- including the database and, optionally, Elasticsearch. It's extremely convenient for development because it avoids the need to manually install and configure individual components. Using Kubernetes is a popular way of running self-hosted production clusters, and our documentation provides some information about these and other deployment scenarios.

The Temporal cli provides a small Temporal Cluster that runs in a single process and doesn't have any external runtime dependencies, so it is simpler and less resource-intensive than the Docker Compose approach.

The alternative to hosting your own Temporal Cluster is to use Temporal Cloud, a fully-managed cloud service operated and staffed by the experts at Temporal. This gives you access to a production-ready Temporal Cluster, with support, and frees your organization from the operational workload of running your own cluster: planning, deploying, updating, monitoring, scaling, and so on.

However, I want to reiterate a point that I made earlier. Whether you host your own Temporal Cluster or use Temporal Cloud, your application -- that is, your code and the Workers that execute it -- runs on infrastructure that you control. Those might be servers in your own datacenter or virtual machines hosted by your favorite cloud provider, but Temporal does not run your code or even have access to it.
--



Temporal Clusters for Development (1)

- **The exercise environment for this workshop is already set up for you**
  - It uses the GitPod service to deploy a cluster and browser-based development environment
- **I'll briefly explain two ways to set up your own**
  - These are for reference, so you can experiment on your own after this workshop

- In this course we minimize learning barriers in the course:
  - Acknowledging challenges of software installation on personal computers

- Provided exercise environment through GitPod:
  - Requires only a standard web browser (e.g., Google Chrome, Safari, Firefox)
  - Includes:
    - A small Temporal Cluster for practice
    - Terminal windows for command input
    - Visual Studio Code editor for coding exercises

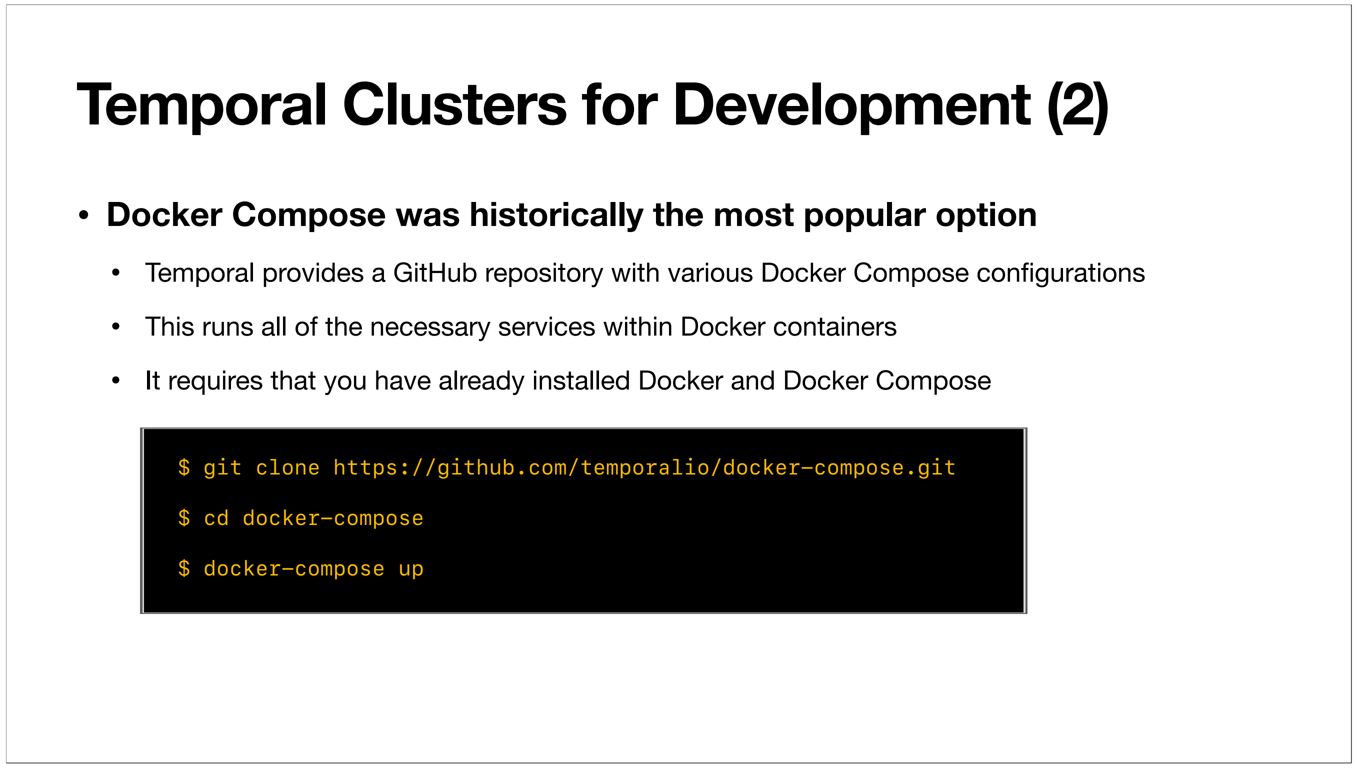
- Encouraging further exploration post-course:
  - Future instructions for setting up a personal development cluster
  - Not necessary to run these commands during the course
  - Provided for future reference when you want to create your own environment

---

In this course, we want to minimize the barriers for people to learn Temporal, and we know that not everyone is able or willing to install software on their computers. For that reason, we've provided an exercise environment for you through a service called GitPod, and the only thing you need to use it is a standard Web browser such as Google Chrome, Safari, or Firefox. That environment contains not only a small Temporal Cluster we've set up just for you, but also terminal windows where you can type commands and the Visual Studio Code editor where you can write your programs.

However, we also hope that you'll be excited to explore more on your own after this course is over, so I've included the instructions here for setting up your own development cluster later.

Just to be clear, **you don't need to run any of these commands now** -- this is for future reference, when you want to set up your own environment.
--



Temporal Clusters for Development (2)

- **Docker Compose was historically the most popular option**
  - Temporal provides a GitHub repository with various Docker Compose configurations
  - This runs all of the necessary services within Docker containers
  - It requires that you have already installed Docker and Docker Compose

```
$ git clone https://github.com/temporalio/docker-compose.git

$ cd docker-compose

$ docker-compose up
```

- Using Docker Compose for local Temporal development:
  - Popular choice for desktop or laptop development environments

- Prerequisites:
  - Docker and Docker Compose already installed and configured on your machine
  - Instructions for Docker setup available online

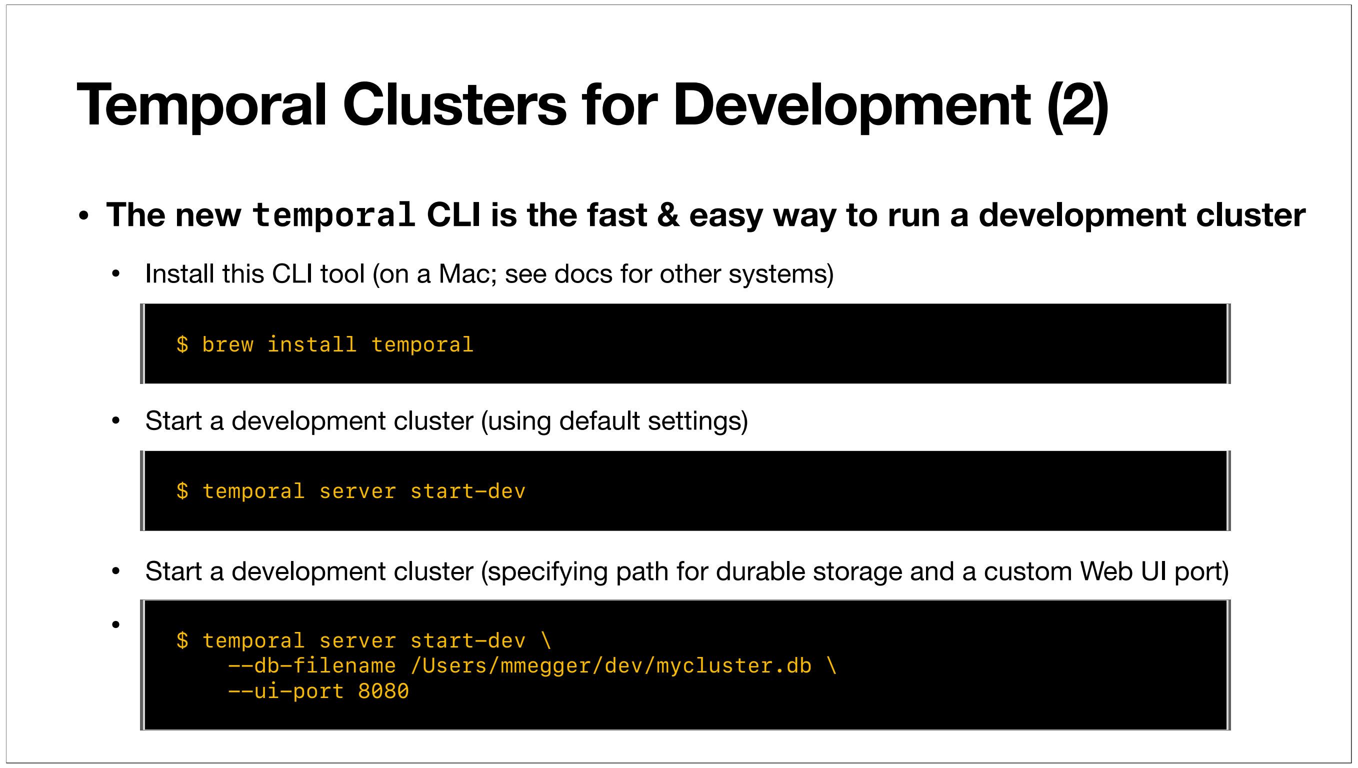
- Temporal setup in three simple steps:
  1. Clone the Temporal docker-compose repository.
  2. Navigate to the directory created by the cloning command.
  3. Run "docker-compose up" to launch the Temporal Cluster.

- Quick setup:
  - Within a minute, you'll have a functioning Temporal Cluster with the Web UI.
  - Default configuration includes PostgreSQL and Elasticsearch.
  - Multiple alternative configurations available in the docker-compose repository, accommodating different databases and memory constraints for development purposes.

As I mentioned, Docker Compose is a popular way to run Temporal on your desktop or laptop for development purposes.

This requires that you already have Docker and Docker Compose installed and configured on the machine you're using, but there are plenty of instructions online for that. There are only three steps for the Temporal part: 1) clone our docker-compose repository, 2) change into the directory created by the first command, and 3) run "docker-compose up" to launch the cluster. You should have a Temporal Cluster -- complete with the Web UI -- running less than a minute later. By the way, the default configuration in our docker-compose repository uses PostgreSQL and includes Elasticsearch, but also provides several other configurations with other databases, both with and without Elasticsearch. These other configurations can be a helpful alternative for development on a memory-constrained laptop.

--



Temporal Clusters for Development (2)

- **The new `temporal` CLI is the fast & easy way to run a development cluster**
  - Install this CLI tool (on a Mac; see docs for other systems)

```
$ brew install temporal
```

  - Start a development cluster (using default settings)

```
$ temporal server start-dev
```

  - Start a development cluster (specifying path for durable storage and a custom Web UI port)
  -

```
$ temporal server start-dev \
    --db-filename /Users/mmegger/dev/mycluster.db \
    --ui-port 8080
```

- An alternative to Docker Compose for local Temporal development:
  - New temporal command-line tool, rapidly replacing Docker Compose

- Advantages of using the temporal command-line tool:
  - Simpler setup than Docker Compose
  - No external dependencies required
  - Lower memory usage compared to Docker Compose (single-process execution)
  - No licensing concerns

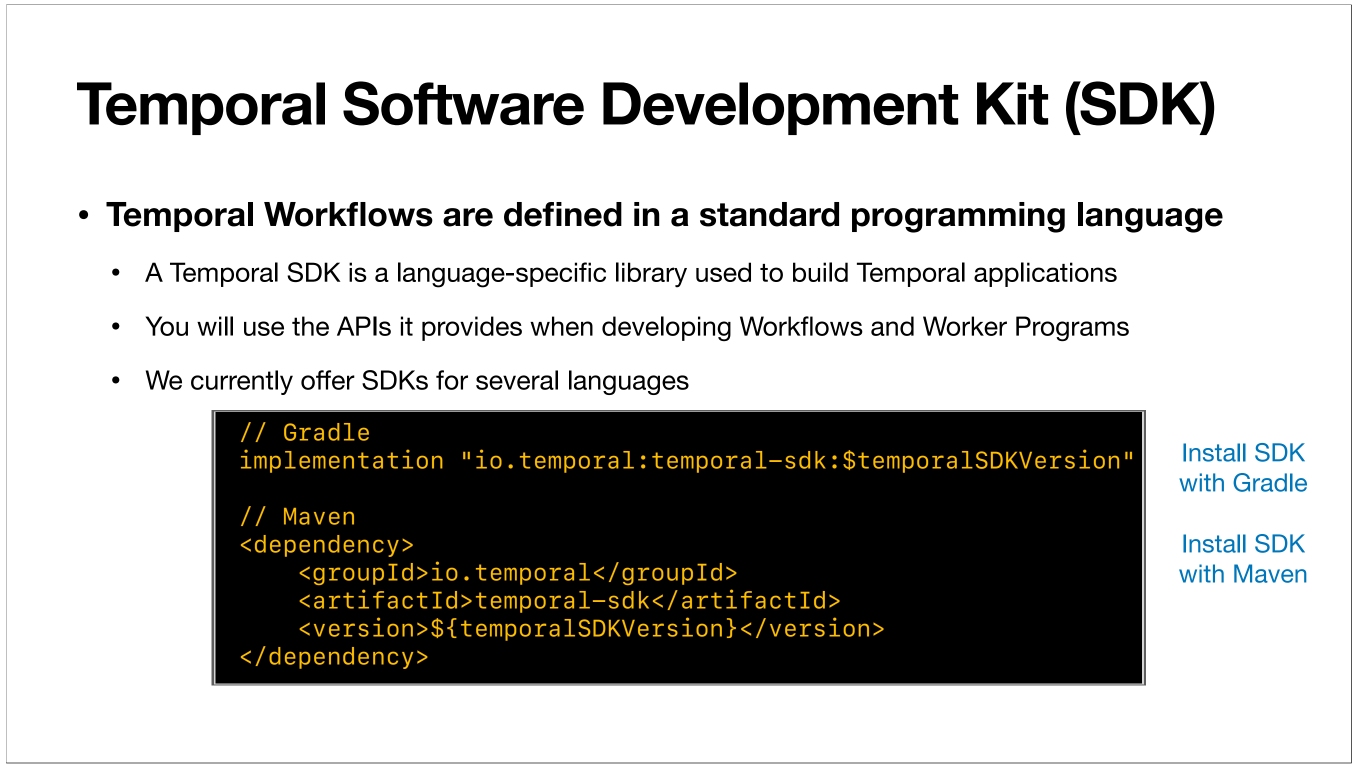
- Installation options:
  - Homebrew for macOS users
  - Pre-built binaries for various systems available on the GitHub repository
  - Option to check out the code and build your own binary

- Additional capabilities of the temporal command:
  - Beyond starting a development cluster, it can:
    - Initiate Workflows
    - Check Workflow status
    - Retrieve Workflow results
  - Expected to replace the older "tctl" command, currently recommended for production use in this workshop.

Another, much newer, option—and one that's quickly replacing Docker Compose—is to use the new temporal command-line tool to run a development cluster. This is easier than Docker Compose, has no external dependencies, and uses less memory than Docker Compose because it runs everything in a single process. Also, unlike Docker Compose, there are no licensing concerns.

You can install this using Homebrew, which is common on a Mac. You can also download pre-built binaries for other systems from our GitHub repository or check out the code and build your own.

The temporal command has many capabilities beyond starting a development cluster—it can also start Workflows, check their status, and retrieve their results. This will soon supersede the older "tctl" command, which is used in this workshop since that's currently what we recommend in production.



Temporal Software Development Kit (SDK)

- **Temporal Workflows are defined in a standard programming language**
  - A Temporal SDK is a language-specific library used to build Temporal applications
  - You will use the APIs it provides when developing Workflows and Worker Programs
  - We currently offer SDKs for several languages

```
// Gradle
implementation "io.temporal:temporal-sdk:$temporalSDKVersion"

// Maven
<dependency>
    <groupId>io.temporal</groupId>
    <artifactId>temporal-sdk</artifactId>
    <version>${temporalSDKVersion}</version>
</dependency>
```

Install SDK with Gradle

Install SDK with Maven

- Developing Temporal Workflows:
  - Done by writing code, not JSON or XML representations of business logic
  - Utilizes standard programming languages, similar to any other application development
  - Code interfaces with Temporal APIs and uses a Temporal Client for cluster communication

- Supporting library: Software Development Kit (SDK)
  - SDKs are language-specific
  - Available for several programming languages, with more in development
  - This course focuses on Java developers, using the Java SDK
  - Commands provided are for Java and applicable to a project management tool (e.g., Maven)

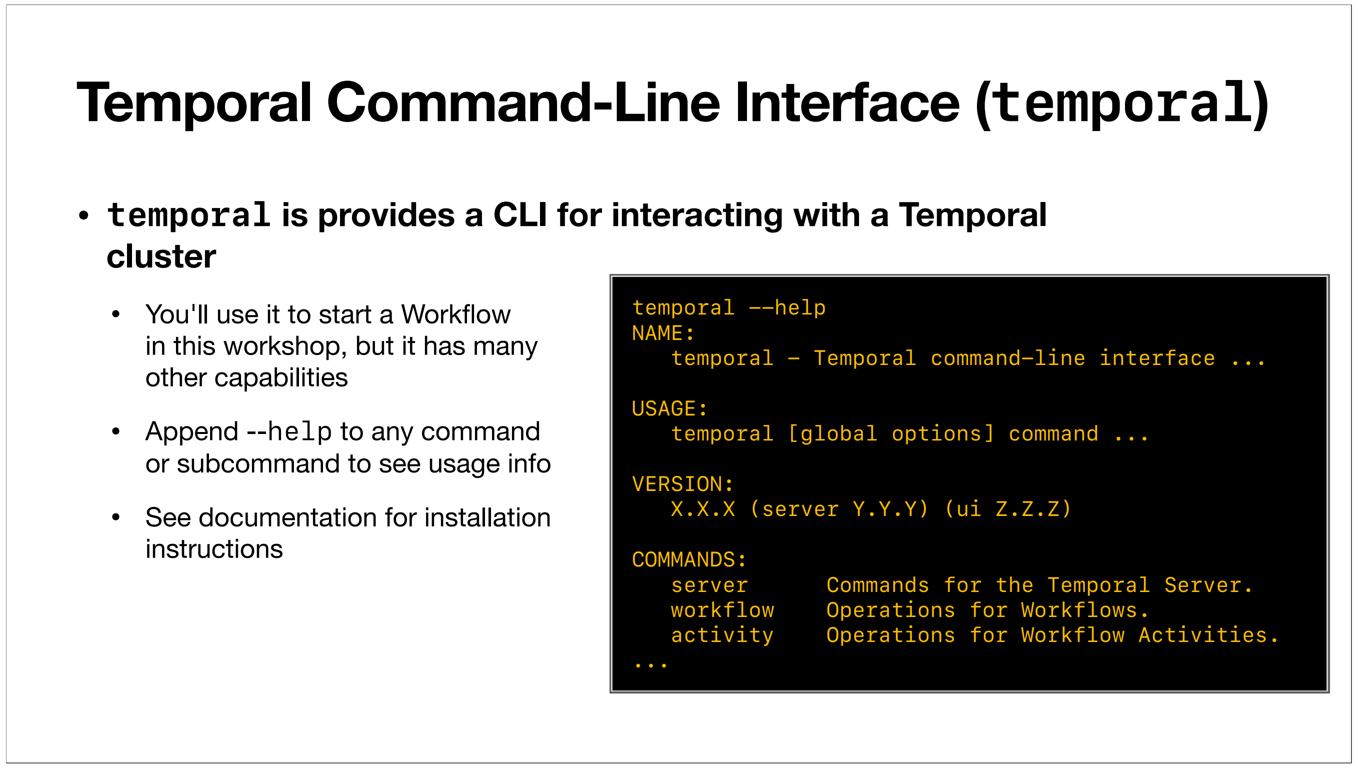
- Configuration and setup:
  - Provided fully-configured exercise environment for the course, so no SDK installation needed
  - Command examples are for setting up your own development environment post-workshop, applicable to Maven projects.

As I mentioned earlier, you develop Temporal Workflows by writing code. Not a JSON or XML representation of your business logic, but standard code using a standard programming language, similar to how you'd write any other type of application. The code you write will use Temporal APIs and use a Temporal Client to communicate with the cluster. The library that provides support for these things is called a Software Development Kit, or SDK. They're specific to the programming language you use and we currently offer them for several languages, with a few others in development. Since this course is for Java developers, we'll be using the Java SDK, and the commands shown here is the one you would type into your project management tool. For this course we will be using maven. Again, we've provided a fully-configured

exercise environment for you -- you won't need to install the SDK there -- but you'll use this command to set up your own development environment once this workshop is over.
--

NOTE: Your project management tool will resolve the dependencies in your pom.xml and install them on first compile

NOTE: I say "language-specific library" but that is not technically correct. It's more like "specific to the runtime environment" because you can write Kotlin (or presumably Scala) with the Java SDK or JavaScript with the TypeScript SDK. I'm not sure that this detail matters at this stage, but it would be best if I could come up with a technically correct way of saying it, even if I don't explain why I chose that wording.



Temporal Command-Line Interface (`temporal`)

- **`temporal` is provides a CLI for interacting with a Temporal cluster**
  - You'll use it to start a Workflow in this workshop, but it has many other capabilities
  - Append `--help` to any command or subcommand to see usage info
  - See documentation for installation instructions

```
temporal --help
NAME:
   temporal - Temporal command-line interface ...

USAGE:
   temporal [global options] command ...

VERSION:
   X.X.X (server Y.Y.Y) (ui Z.Z.Z)

COMMANDS:
   server      Commands for the Temporal Server.
   workflow    Operations for Workflows.
   activity    Operations for Workflow Activities.
...
```

- Installation of the tctl tool:
  - Command-line interface for Temporal operations
  - Useful for starting workflows, monitoring execution status and history, and offering advanced capabilities

- Installation methods:
  - Already installed in the provided exercise environment
  - For macOS users with Homebrew, the first command installs tctl
  - Various installation methods available, depending on your system and preferences
  - Documentation contains detailed installation instructions

- tctl functionality:
  - Consists of 10 commands, each with multiple subcommands
  - "help" command provides descriptions of available options
  - "temporal --help" provides basic usage information, including version number and option list

---
Another tool you'll probably want to install in your development environment, although it's technically optional, is called tctl. It's a command-line interface for working with Temporal. We'll use it during this course to start workflows and view the status and history of their execution, although it has many more advanced capabilities as well.

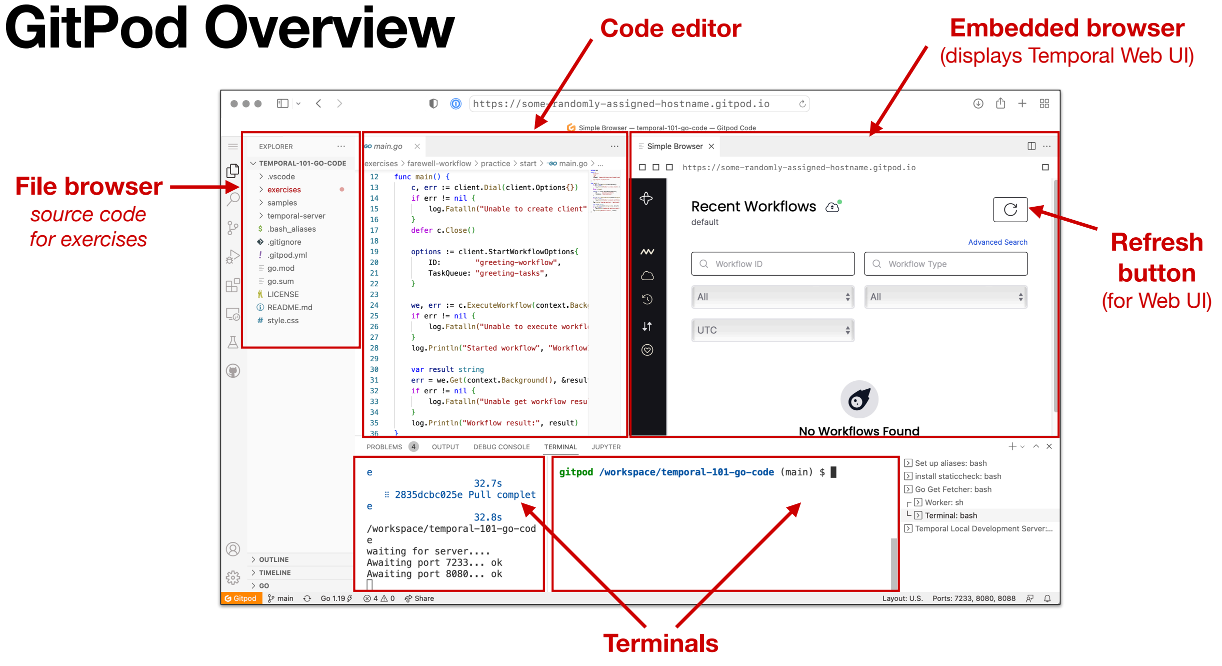
Although it's already installed in the exercise environment we've set up for you, the first command shown here is the one you'd use to install it on a if you have Homebrew installed, which is popular on Macs. There are also other methods for installing it; for example, if you don't have Homebrew installed or if you're using a different operating system, and they are described in the documentation.

The tctl program has 10 commands, each of which has several subcommands. If you type "help" after a command or partial command, it will describe the options to you. If you type "temporal --help" like I've shown in the second example, it will show basic usage information, including the version number and a list of options.
--

Exercise Environment

- **We provide a development environment for you in this course**
  - It uses the GitPod service to deploy a private cluster, plus a code editor and terminal
  - You access it through your browser (may require you to log in to GitHub)
  - Your instructor will provide the URL you can use to launch the environment
    - Please launch the exercise environment now so it will be ready for your first exercise
    - Your instructor will demonstrate how to do this (and how to open and select terminals in the environment)

GitPod Overview



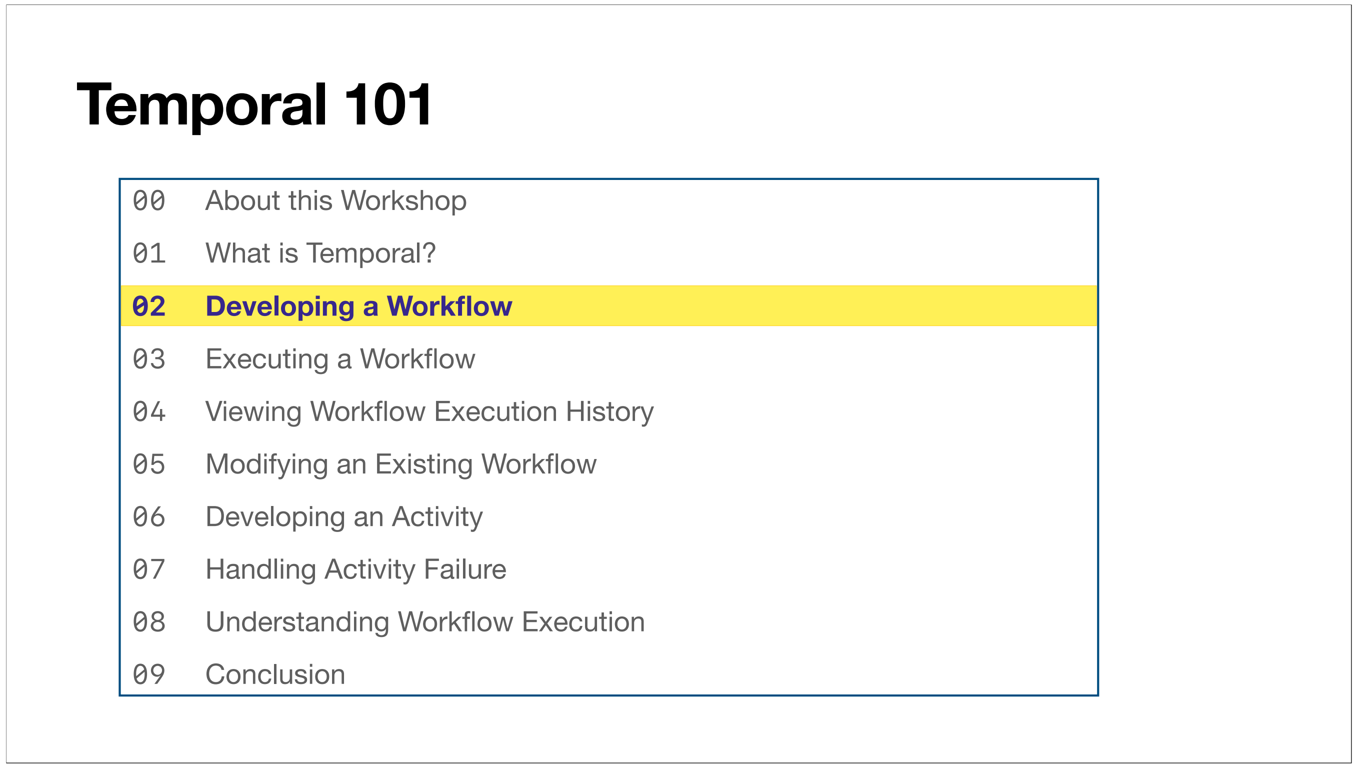
```
func main() {
    c, err := client.Dial(client.Options{})
    if err != nil {
        log.Fatalln("Unable to create client"
    }
    defer c.Close()

    options := client.StartWorkflowOptions{
        ID:        "greeting-workflow",
        TaskQueue: "greeting-tasks",
    }

    we, err := c.ExecuteWorkflow(context.Back
    if err != nil {
        log.Fatalln("Unable to execute workfl
    }
    log.Println("Started workflow", "Workflow

    var result string
    err = we.Get(context.Background(), &resul
    if err != nil {
        log.Fatalln("Unable get workflow resu
    }
    log.Println("Workflow result:", result)
}
```

```
e
                   32.7s
   ⠿ 2835dcbc025e Pull complet
e
                   32.8s
/workspace/temporal-101-go-cod
e
waiting for server....
Awaiting port 7233... ok
Awaiting port 8080... ok
```

```
gitpod /workspace/temporal-101-go-code (main) $
```

--

It's better to demo this live, if possible, than to show the picture (which is really intended for reference)

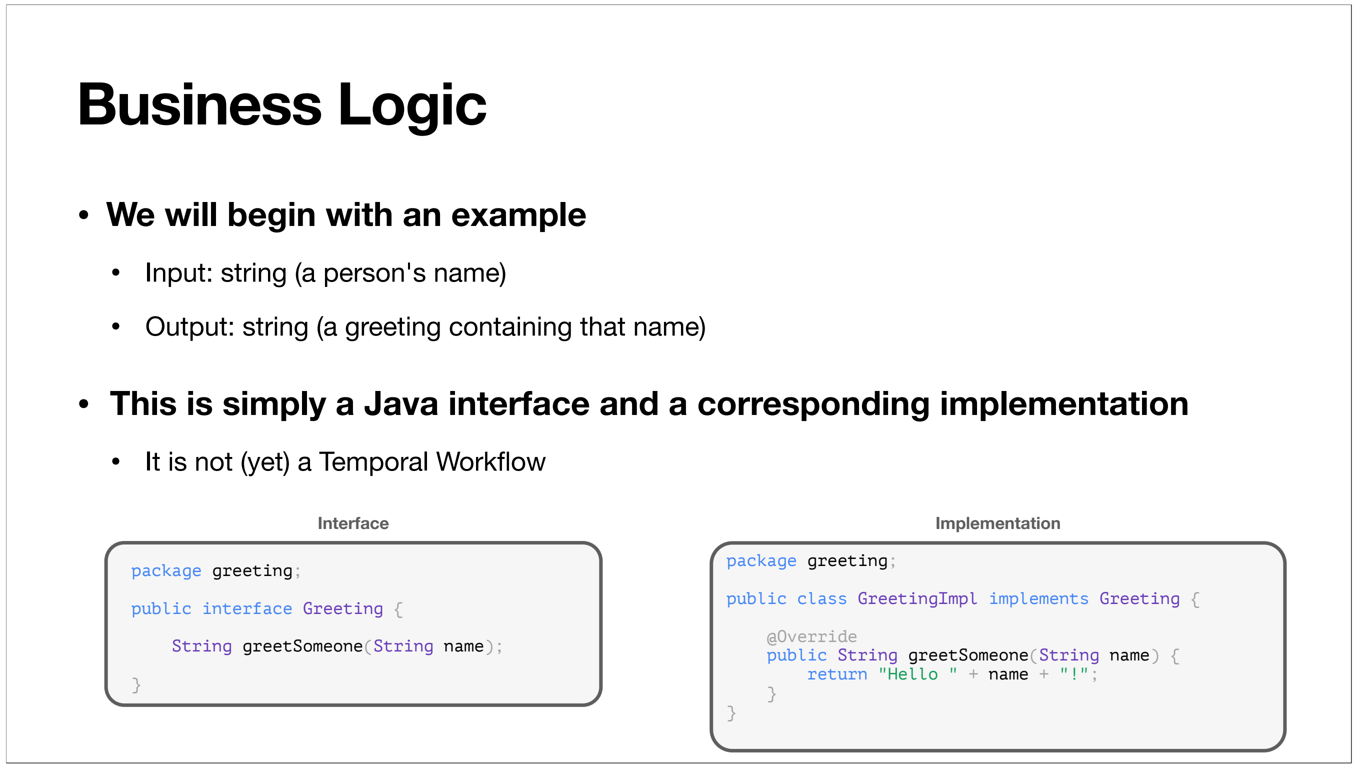


Temporal 101

--

AUDIENCE PARTICIPATION
Before we continue, can anyone tell me what we call the part of an application that executes your code?

Answer: The Worker is what executes your code. The Temporal Cluster communicates with the Worker to orchestrate this, and you'll learn more about their interaction a little later in this session.



Business Logic

- **We will begin with an example**
  - Input: string (a person's name)
  - Output: string (a greeting containing that name)
- **This is simply a Java interface and a corresponding implementation**
  - It is not (yet) a Temporal Workflow

Interface

```java
package greeting;

public interface Greeting {

    String greetSomeone(String name);

}
```

Implementation

```java
package greeting;

public class GreetingImpl implements Greeting {

    @Override
    public String greetSomeone(String name) {
        return "Hello " + name + "!";
    }
}
```

- Creating a Workflow in Temporal:
  - Commencing with a basic example and progressing incrementally
  - Temporal enables concentration on the business logic

- Initial steps in Java application development:
  1. Begin by creating a package, for example, "greeting" (for simplicity).
  2. Proceed with writing the code itself.
  3. Define an interface containing a method:
     - Input: String (representing a person's name)
     - Output: String (a personalized greeting including the person's name)
  4. Create a class that implements this interface and provide the method implementation.
  - This code represents the business logic for our initial Workflow, although it's not a Workflow yet; it's an interface and its implementation.

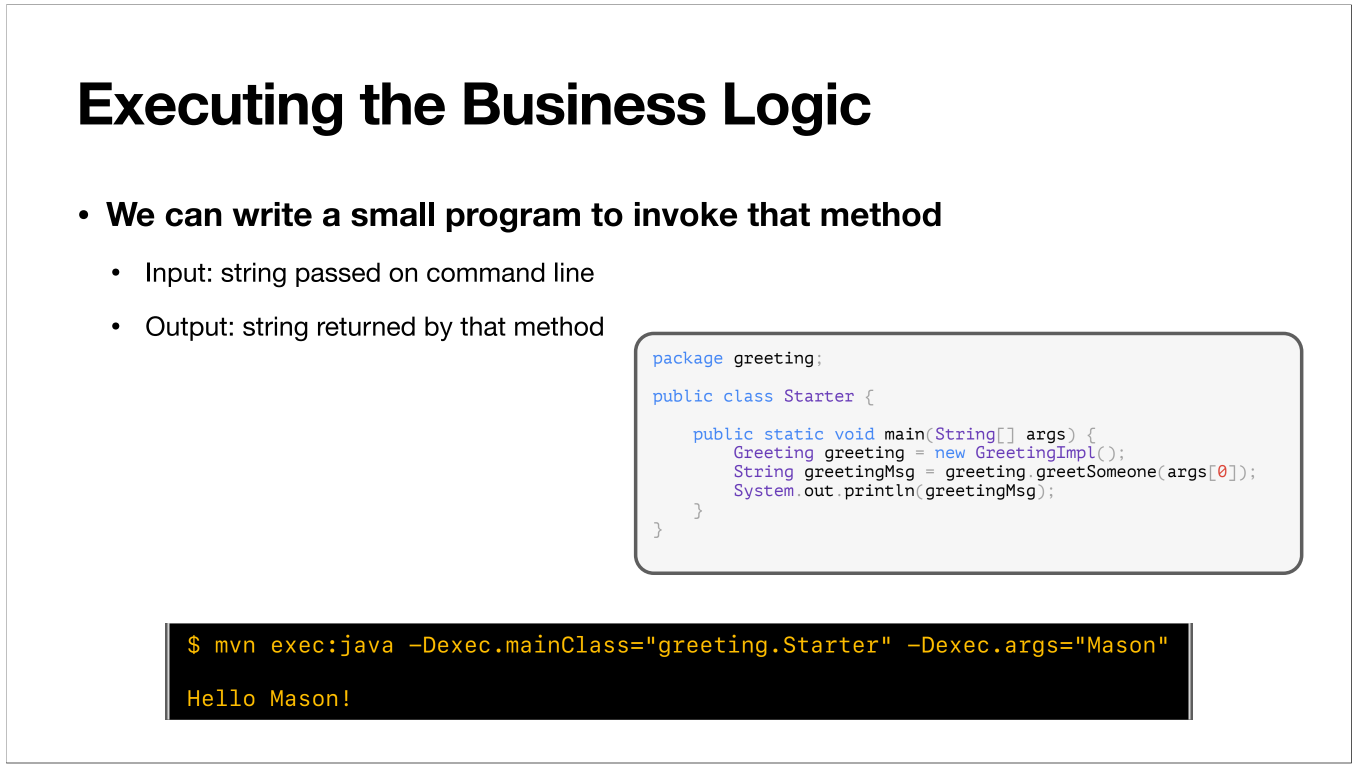
- Note: Future code samples will include syntax highlighting and line numbering.

Now that I've explained the basic architecture and tools you'll use with Temporal, I'll explain how to create a Workflow. In this course, we're going to start with a very basic example and build on it as we go. Since Temporal lets you focus on the business logic, we'll begin with that.

When developing a Java application, our first step is often to create a package, and I've named it "greeting" for the sake of simplicity. The next step is writing the code itself, so I've defined an interface that defines a method that takes as string -- a person's name -- as an input parameter and returns another string -- a greeting

customized with that person's name -- as output. I've then created a class that implements this interface and implemented the methodality for the interface.That's the business logic for our first Workflow, but it's not a Workflow yet, it's just a Interface/Implementation.
--

NOTE: Starting with this slide, I should show all code samples in a different box, one with syntax highlighting and line numbering. The command box can still be above it, if relevant, although we could leave out the "cat" command since it should be obvious from context that the code sample will be what we're editing.



Executing the Business Logic

- **We can write a small program to invoke that method**
  - Input: string passed on command line
  - Output: string returned by that method

```java
package greeting;

public class Starter {

    public static void main(String[] args) {
        Greeting greeting = new GreetingImpl();
        String greetingMsg = greeting.greetSomeone(args[0]);
        System.out.println(greetingMsg);
    }
}
```

```
$ mvn exec:java -Dexec.mainClass="greeting.Starter" -Dexec.args="Mason"

Hello Mason!
```

- Running Java Code:
  - Entry point for a Java application is typically a "main" method defined in a separate class within the package.
  - This approach separates business logic from the details of running the program.

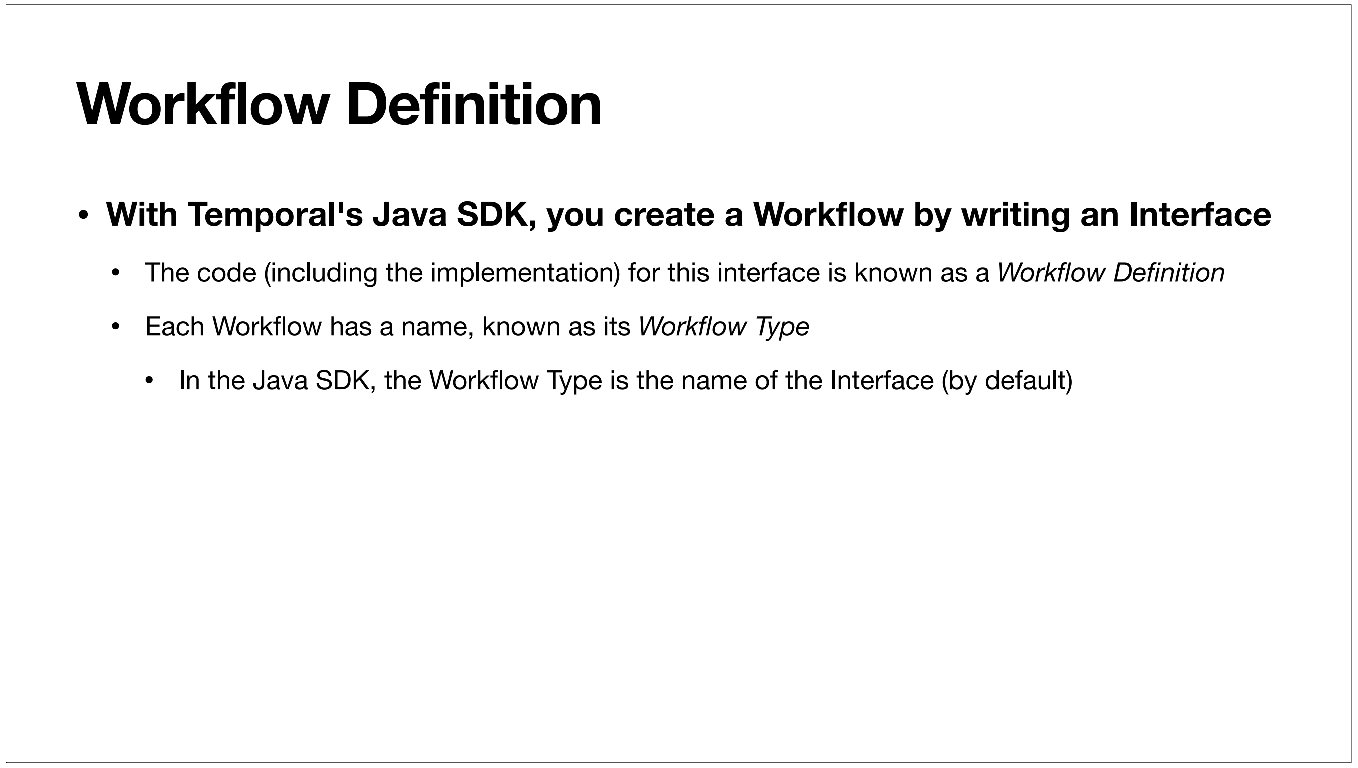
- Example Java program:
  - Contains a "main" method in the "main" package
  - Accepts input from a command-line argument
  - Invokes the business logic method with the input
  - Assigns the returned string to a variable named "greeting"
  - Prints the "greeting" value to standard output
  - Below the code, you can see the command used to run it and the resulting output.

- Note: The code shown here is not a Temporal Workflow; it's standard Java code representing business logic and a method for execution. This demonstrates how you can incorporate existing business logic into Temporal Workflows.

Although the method I just showed is perfectly valid, there's no way to run it. That's because the entry point for a Java application is a method called "main" that's defined in a separate class within the package. This is a typical way of invoking a Java program, and we keep the business logic separate from all the details related to running it.

The code shown here is a Java program, which we can run because it has a "main" method defined in the "main" package. It takes input from an argument specified on the command line, invokes the business logic method by passing in that input, assigns the string returned from the business logic method to a value named "greeting" and then prints that value to standard output. Just below the code, you can see the command I used to run this, along with the output it produced.

Again, what I've shown so far is not a Temporal Workflow; it's just standard Go code that represents our business logic and a means of executing it. However, as a developer, you've probably written lots of business logic before today, and you might use some of that existing code when creating Temporal Workflows, so I think it's important to show you how to do that.



Workflow Definition

- **With Temporal's Java SDK, you create a Workflow by writing an Interface**
  - The code (including the implementation) for this interface is known as a *Workflow Definition*
  - Each Workflow has a name, known as its *Workflow Type*
    - In the Java SDK, the Workflow Type is the name of the Interface (by default)

- Defining a Workflow in Temporal:
  - Create an interface to define the Workflow.
  - The code constituting the Workflow is known as the "Workflow Definition."

- Introducing the term "Workflow Type", as seen in UI:
  - Each Workflow has a name, referred to as the Workflow Type in Temporal.
  - "Workflow Type" may seem confusing but think of it like a "type" in a programming language, used to refer to an entity in your code (e.g., "Customer" or "Product" type).
  - In the Java SDK, a Workflow's type defaults to the name of the interface used to define it.
  - You can override this default to provide a more user-friendly value since the Web UI displays Workflow Executions by their type.

- Naming conventions for Workflow methods:
  - Temporal does not impose any rules on how you name your Workflow methods.
  - You can use the naming convention you prefer.

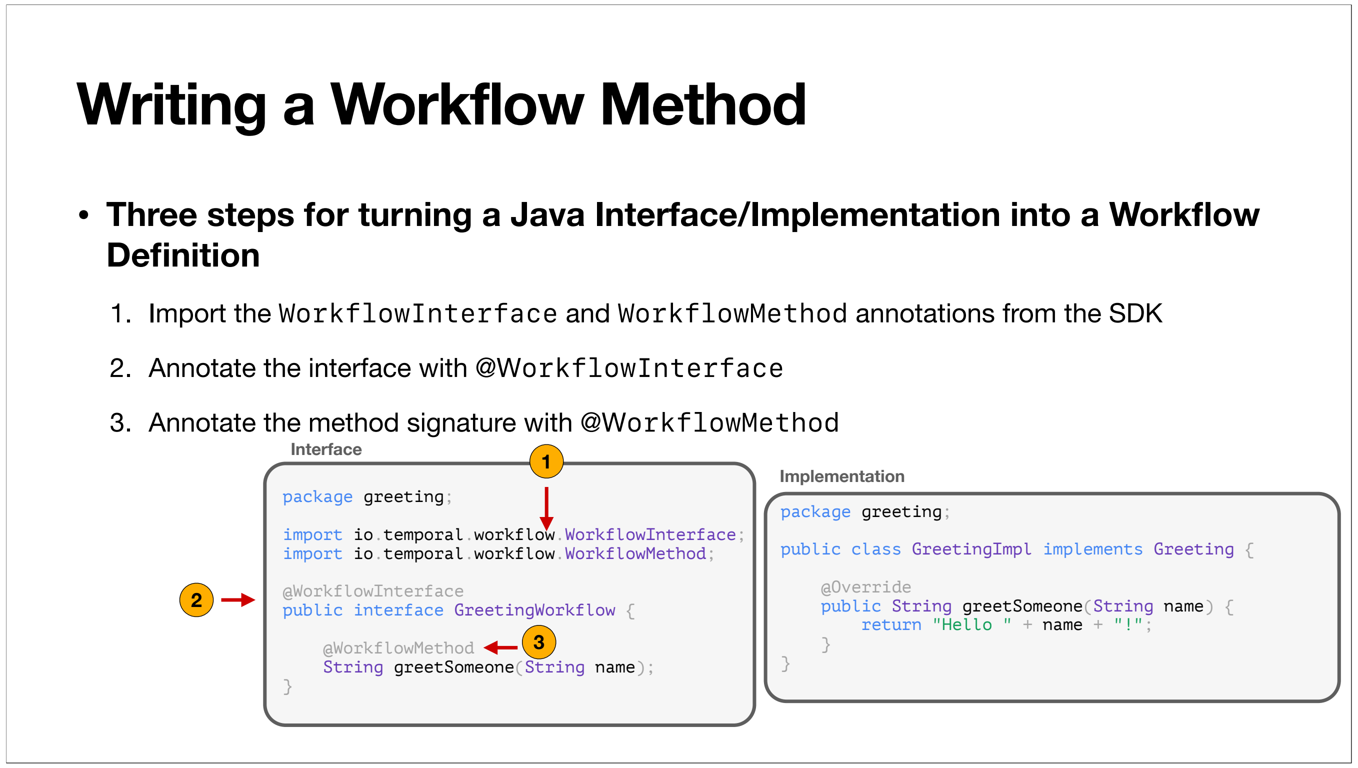
- Note: While the Workflow Type defaults to the interface name in the Java SDK, this behavior may differ with other SDKs. For example, in the Go, TypeScript, and Python SDKs, the Workflow Type defaults to the method name.

In Temporal, you define a Workflow by creating an interface. The code that makes up the Workflow is known as the "Workflow Definition."

I want to quickly introduce another term, because you'll encounter it in the Web UI. Every Workflow has a name, which Temporal refers to as the *Workflow Type*. It's perhaps a confusing term because "type" can mean so many things in English, but think of it like a "type" in a programming language, which is a way to refer to an entity in your code. For example, you might have a "Customer" type or a "Product" type. In theJava SDK, by default, a Workflow's type is the name of the interface used to define that Workflow, but it is possible to override it and provide a more user-friendly value since the Web UI displays Workflow Executions by their type.

Also, Temporal doesn't impose any rules about how you name your Workflow method, so you can use whatever convention you prefer.
--

Although the Workflow Type defaults to the name of the interface in the Java SDK, this behavior may differ with other SDKs. For example, in the Go, TypeScript and Python SDKs, the Workflow Type defaults to the method name



Writing a Workflow Method

- **Three steps for turning a Java Interface/Implementation into a Workflow Definition**
  1. Import the `WorkflowInterface` and `WorkflowMethod` annotations from the SDK
  2. Annotate the interface with `@WorkflowInterface`
  3. Annotate the method signature with `@WorkflowMethod`

Interface

1

```
package greeting;

import io.temporal.workflow.WorkflowInterface;
import io.temporal.workflow.WorkflowMethod;

@WorkflowInterface
public interface GreetingWorkflow {

    @WorkflowMethod
    String greetSomeone(String name);
}
```

2

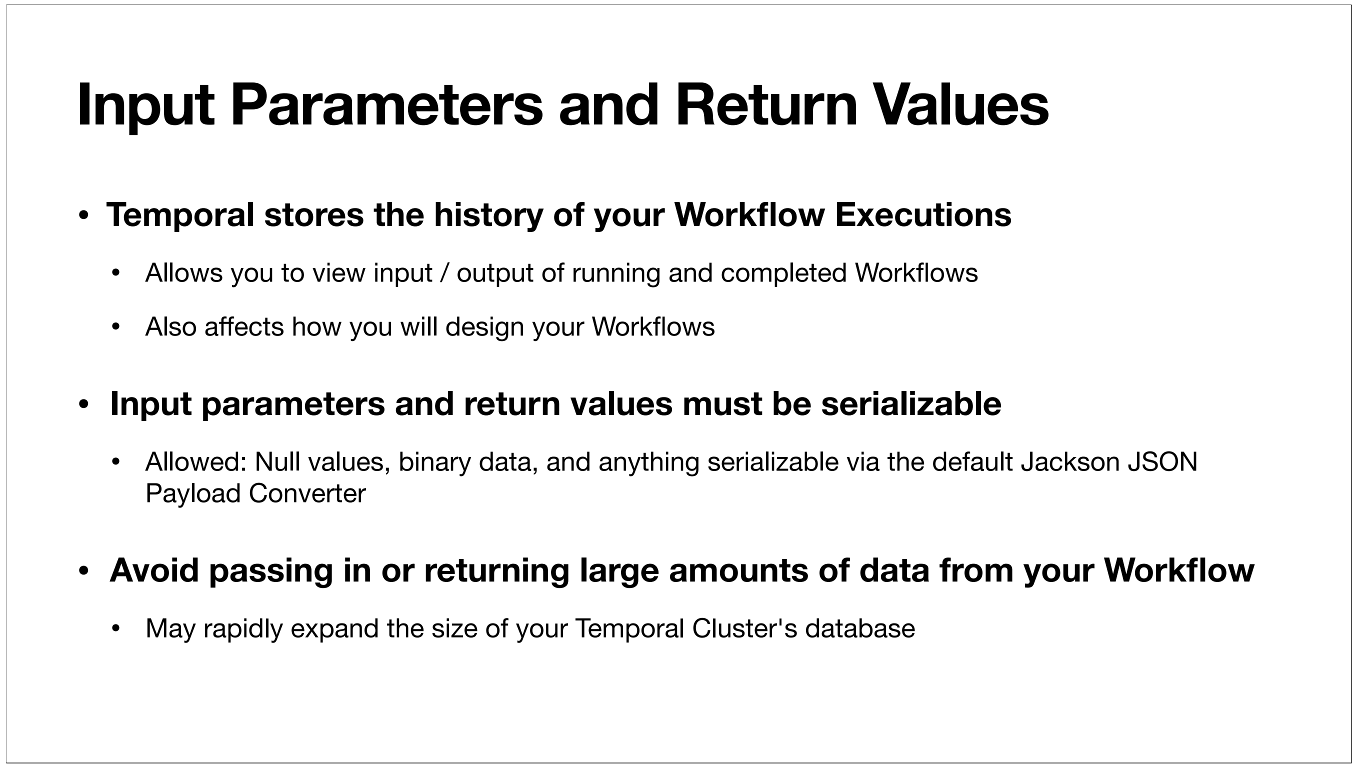
3

Implementation

```
package greeting;

public class GreetingImpl implements Greeting {

    @Override
    public String greetSomeone(String name) {
        return "Hello " + name + "!";
    }
}
```

You could write a Workflow Definition from scratch or you could create one by modifying an existing method. In either case, you'll need to do three things. First, you'll need to import the "WorkflowInterface" and "WorkflowMethod" annotations from the Temporal Java SDK. This is required because your interface and workflow method must be annotated respectfully.
--

NOTES: this is not an exhaustive list of changes we might need to make, since we haven't covered determinism yet (this is why I wrote "our Java interface" since it is already deterministic).



Input Parameters and Return Values

- **Temporal stores the history of your Workflow Executions**
  - Allows you to view input / output of running and completed Workflows
  - Also affects how you will design your Workflows
- **Input parameters and return values must be serializable**
  - Allowed: Null values, binary data, and anything serializable via the default Jackson JSON Payload Converter
- **Avoid passing in or returning large amounts of data from your Workflow**
  - May rapidly expand the size of your Temporal Cluster's database

There are a few important things to consider when it comes to the values passed as input and returned as output from your - Considerations for Workflow Input and Output Values:
  - Temporal maintains information about current and past Workflow Executions.
  - This data retention is helpful for problem investigation, even for issues that occurred in the past.
  - Designing Workflow Definitions requires consideration of input and output values.

- Serialization Requirement:
  - Temporal stores Workflow input and output data, necessitating serializability.
  - By default, Temporal handles null or binary values and data serializable using the Java Jackson JSON Payload Converter.
  - Most commonly used data types, such as integers, floating-point numbers, booleans, and strings, are automatically handled.
  - Complex types composed of these basic types are also supported.

- Data Confidentiality:
  - While input parameters and return values are stored as part of the Event History, you can set up encryption to ensure confidentiality during transmission and storage.

- Note: Ensure your Workflow input and output values are serializable to work effectively with Temporal.

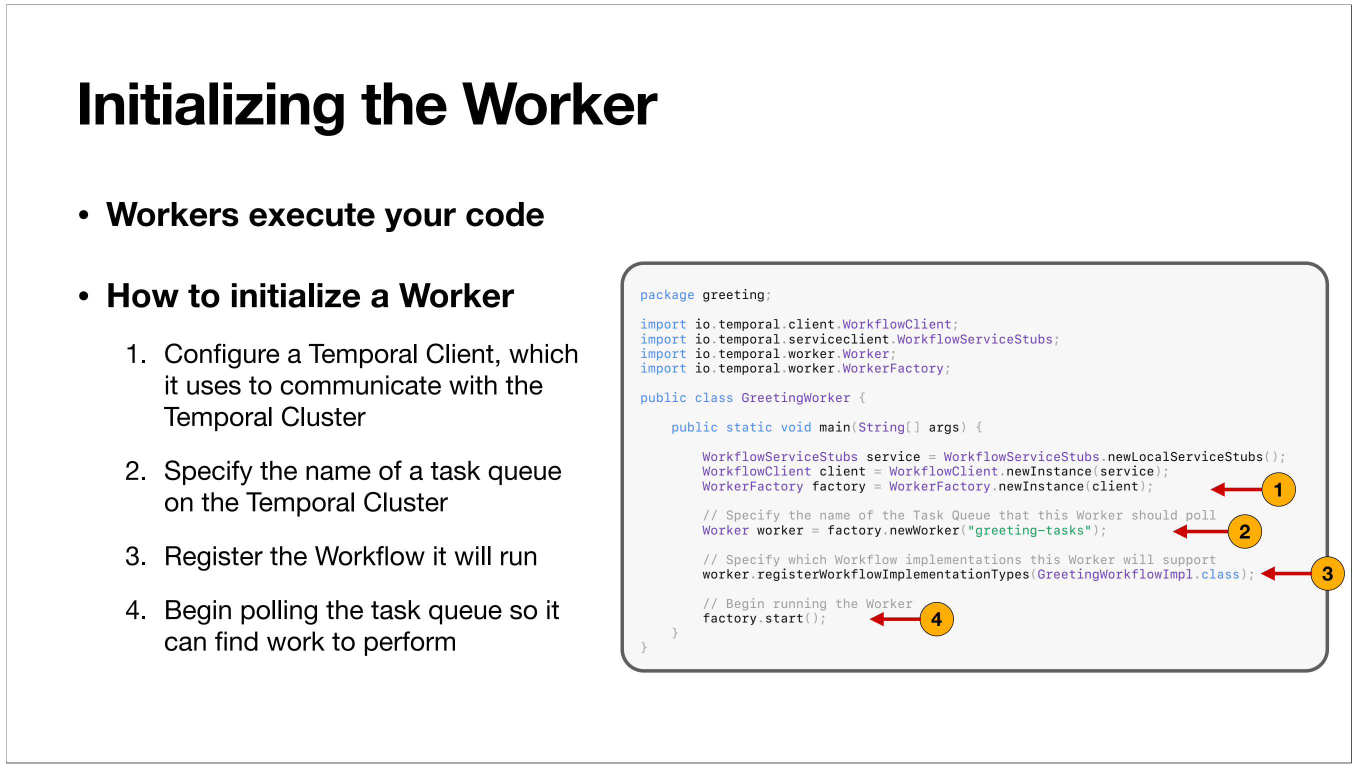
Workflows. As I mentioned earlier, Temporal maintains information about current and past Workflow Executions. This is very helpful, because as you'll see later, you can use the Web UI to explore these details when investigating a problem, even one that may have occurred days or weeks earlier. However, the fact that it does this will

affect how you design Workflow Definitions.

In order for Temporal to store the Workflow's input and output means that this data must be serializable. By default, Temporal can handle null or binary values, as well as any data that can be serialized using the Java Jackson JSON Payload Converter. This means that most of the types you'd typically use in a method, such as integers and floating point numbers, boolean values, and strings are all handled automatically, as are classes composed from these things.

By the way, although the input parameters and return values are stored as part of the Event History of your Workflow Executions, you can set up encryption to ensure its confidentiality during both transmission and storage.
--

NOTE: The serialization mechanism described is known as the DataConverter and it is also possible to develop a custom one, using it instead of the default. This is commonly done for security (e.g. to encrypt data before it's stored in the cluster's database) or performance (e.g., applying data compression).



Initializing the Worker

- **Workers execute your code**
- **How to initialize a Worker**
  1. Configure a Temporal Client, which it uses to communicate with the Temporal Cluster
  2. Specify the name of a task queue on the Temporal Cluster
  3. Register the Workflow it will run
  4. Begin polling the task queue so it can find work to perform

```java
package greeting;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);              ← 1

        // Specify the name of the Task Queue that this Worker should poll
        Worker worker = factory.newWorker("greeting-tasks");                ← 2

        // Specify which Workflow implementations this Worker will support
        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);  ← 3

        // Begin running the Worker
        factory.start();       ← 4
    }
}
```

- Worker Initialization and Workflow Execution:
  - Worker initialization is part of your application, alongside the Workflow Definition code.
  - Workers execute Workflow code, similar to invoking a method in a regular program.
  - Workers run the code contained within the Workflow method.
  - Unlike the basic program example shown earlier, the Worker program doesn't directly call the Workflow method.

- Code Explanation:
  - The Worker Program resides in the "greeting" package and has a "main" method.
  - Necessary classes from the Temporal SDK are imported.
  - In the "main" method:
    - A Temporal client is created, including gRPC stubs, a client, and a factory of clients.
    - Default options for creating a local development client are used.
    - A Worker entity is created, specifying the task queue name and configuration options.
    - Workers coordinate with the Temporal Cluster through Task Queues.
    - The Worker registers the Workflow using a reference to the fully-qualified Workflow method.
    - In a production environment, there may be multiple Workers and Workflow Types.
    - The "factory.start()" method is called on the Worker, initiating a "long poll" with the cluster to continuously poll the task queue for work.
    - The Factory manages worker creation and lifecycle.
    - This is a blocking method and continues until an error occurs.
    - A single Worker instance can process numerous short-lived Workflows during its lifetime.

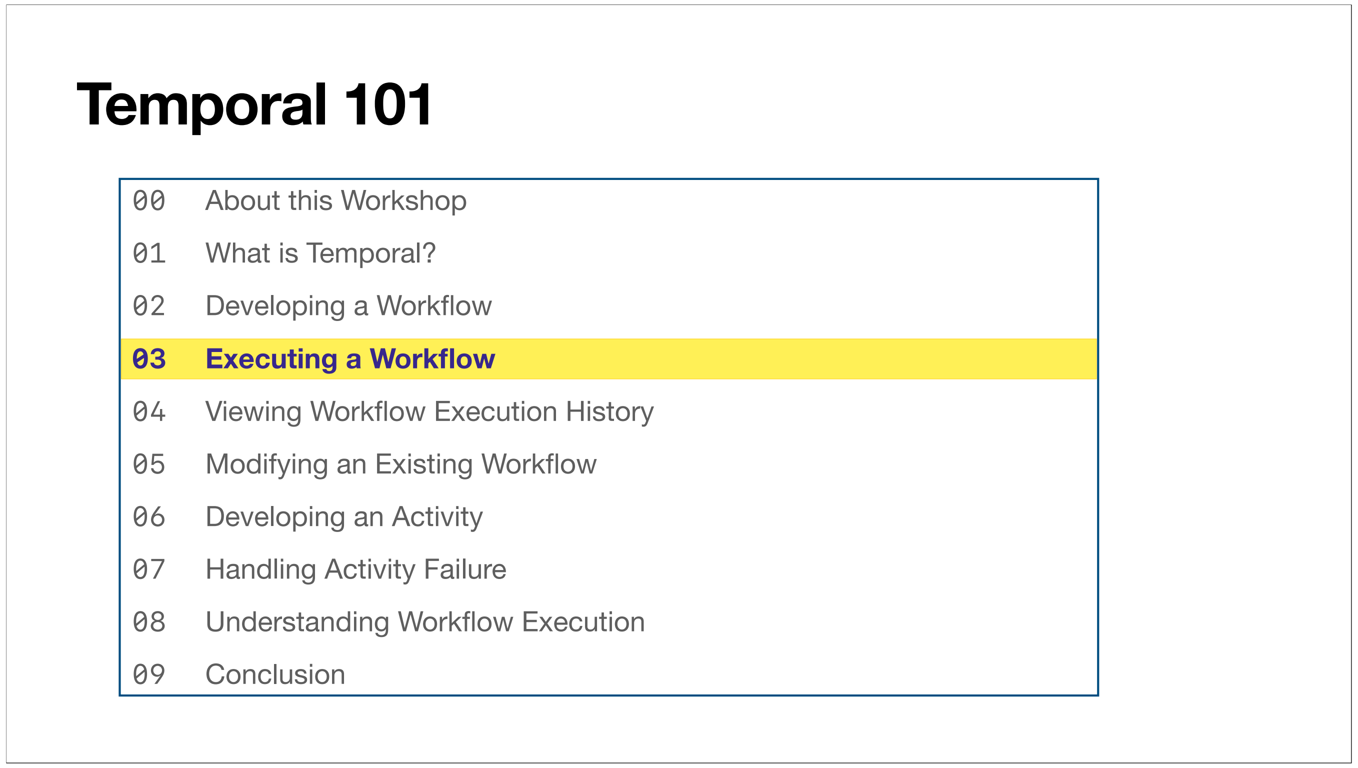
- Note: Workers execute Workflow code and are initialized within your application. They continuously poll the task queue for work in a long-polling fashion.

As I mentioned before, the Worker initialization is part of your application, right alongside the code that makes up your Workflow Definition. In fact, you may recall that Workers execute your Workflow code. Remember the basic program I showed earlier that ran our business logic by invoking the method? The same concept applies here -- the Worker invokes the Workflow method and runs all the code it contains. There is one slight difference, which we'll explore in just a moment, and that is that the Worker program doesn't contain any code that *directly calls* the Workflow method.

Let me explain the code, starting at the top. First, the Worker Program is a program we'll run, so it needs to be in the "greeting" package and have a "main" method. We need to import the necessary classes from the Temporal SDK. The first three lines of the "main" method create a Temporal client that will communicate with the cluster, first by creating the gRPC stubs, then a client, then a factory of clients. By the way, we're using the default options for creating a local development client here, as you can see with "newLocalServiceStubs". To connect to a remote Cluster we'd use "newServiceStubs" and set options to specificy things like IP address/fqdn and port. Once we've created a client, we use it to create a new Worker entity, also specifying the name of a task queue, and some configuration options. Workers coordinate with the Temporal Cluster through Task Queues, and they're created dynamically, so the Temporal Cluster will create a task queue called "greeting-tasks" if it doesn't already exist. In the next line of code, the Worker registers the Workflow using a reference to the fully-qualified Workflow method.

In this example, there's only one Worker and only one Workflow Type, but you should understand that in a production environment we would typically have multiple of each. In other words, a single Worker can support more than one Workflow Type, and a single Workflow Type might have many concurrent executions across multiple Workers. Towards the end of the "main" method, we call the "factory.start()" method on the Worker, which begins a "long poll" with the cluster -- in other words, it will continuously poll the task queue on the Temporal Cluster's, looking for work to perform. The Factory maintains worker creation and lifecycle. This is a blocking method, so it doesn't stop unless there's an error. Assuming that you have short-lived Workflows like this one, a single Worker instance might process thousands or more of them during its lifetime.

--

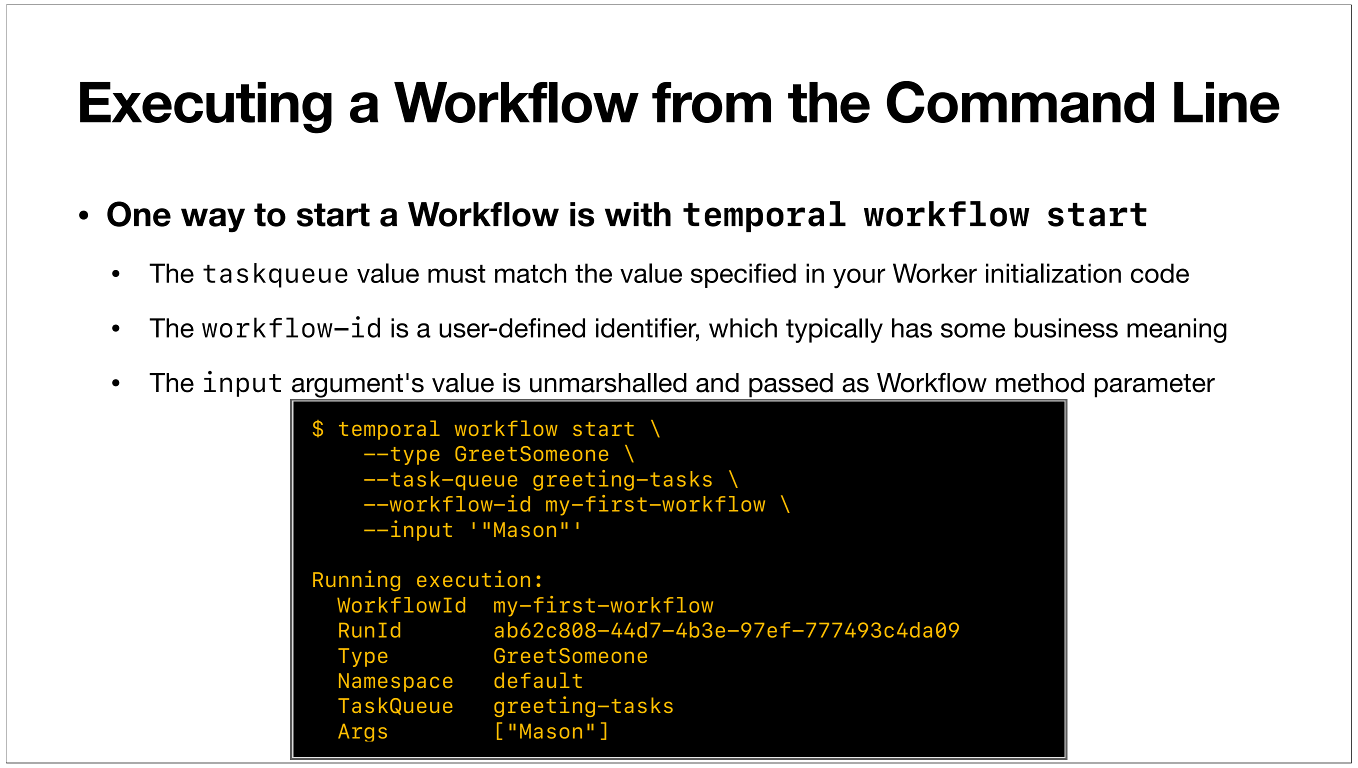


Temporal 101

--

AUDIENCE PARTICIPATION:
Before we go one, can someone name one of the changes we need to make to a method for it to be a valid Workflow Definition using the Go SDK?

1) add "workflow.Context" as first input parameter to the method,
2) return an error from the method (potentially in addition to some other return value)



Executing a Workflow from the Command Line

- **One way to start a Workflow is with `temporal workflow start`**
  - The `taskqueue` value must match the value specified in your Worker initialization code
  - The `workflow-id` is a user-defined identifier, which typically has some business meaning
  - The `input` argument's value is unmarshalled and passed as Workflow method parameter

```
$ temporal workflow start \
    --type GreetSomeone \
    --task-queue greeting-tasks \
    --workflow-id my-first-workflow \
    --input '"Mason"'

Running execution:
  WorkflowId  my-first-workflow
  RunId       ab62c808-44d7-4b3e-97ef-777493c4da09
  Type        GreetSomeone
  Namespace   default
  TaskQueue   greeting-tasks
  Args        ["Mason"]
```

- Executing a Temporal Workflow:
  - Running a Temporal Workflow is conceptually similar to executing a regular Java program.
  - Two ways to execute a Workflow:
    - Using the "temporal" command-line tool.
    - Using the Temporal Client within your application.

- Executing with the "temporal" CLI:
  - Command format: "temporal workflow start"
  - Required options:
    - Workflow Type: Defaults to the name of your Workflow method (e.g., "GreetSomeone").
    - Task Queue Name: Must match the name specified in the Worker code.
    - Input Data: Passed in JSON format, either inline or via an input file.
  - The command submits the execution request to the cluster and returns Workflow ID and Run ID.
  - Workflow ID uniquely identifies a Workflow, while Run ID uniquely identifies this specific execution of that Workflow.
  - Note: The command doesn't display the Workflow result but provides a means to fetch it, as we'll cover later.

You now have a Temporal Workflow and a Worker capable of executing it. The next step is to run your application, so I am going to explain how to do that. This is conceptually similar to that basic Java program I showed in the beginning, which accepts some input and executes your business logic by passing the input to your method.

I'm going to cover two ways that you can execute a Workflow and the first is by using the "temporal" command-line tool. You'll type "temporal", which is the name of the program, "workflow", which is the command, and "start", which is the subcommand.

When executing a Workflow, you'll need to specify a few options on the command line. The first is the Workflow Type. By default, this is the the name of your Workflow method, so I type "GreetSomeone" here. Next, I specify the name of the task queue that the Worker and Temporal Cluster will use for coordination, so the value specified here must exactly match the one specified in the Worker code. Since task queues are dynamically created, typing the task queue name incorrectly wouldn't cause an error, it would just result in two different task queues being created, so no coordination would take place and the Workflow would never make any progress. Finally, if the Workflow takes input, you'll need to pass it in. When submitting a Workflow for execution through the command line, the input is always in JSON format, which is why the input in this command shows double quotes inside of single quotes. Typing JSON directly on the command line is fine for a simple case like this, where there's just one parameter and a single value, but it would be a clumsy way of passing more complex data. Luckily, you can save the input to a file, in JSON format, and specify its path to the "input_file" option, rather than using the "input" option to specify the data inline as shown here.

When you run the command, it submits your execution request to the cluster, which responds with the Workflow ID, a value that uniquely identifies the Workflow, and a Run ID, which uniquely identifies this specific execution of that Workflow. It does *not* display the result returned by the Workflow -- remember that Workflows can run for months or years -- but you can use a different temporal command to fetch the result, and I'll cover it later.
--

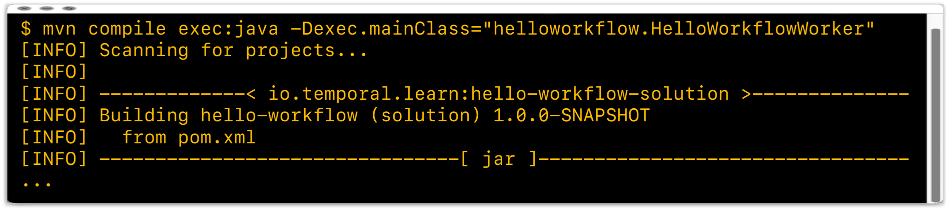
IG: Why is it necessary to have both a Workflow ID and a run ID, given that a Workflow ID uniquely identifies a Workflow Execution (not just a Workflow)? Well, Temporal guarantees (by default) that there will only be one Workflow Execution with a given Workflow ID open at a time, not that there won't ever be another Workflow Execution with that same Workflow ID (i.e., after the previous one reaches the Closed state). Since you can specify the Workflow ID, rather than using a system-assigned value, then this is indeed a likely scenario. It also happens when you have a Temporal Cron job; for example, one that runs every 10 minutes; all will have the same Workflow ID but will have different run IDs.

As for limitations on Workflow ID, this came up in a thread on Community Slack and Roey pointed to Temporal Server code that imposes a 1,000-character limit. Yimin later replied that the string must be valid UTF-8, but the actual limit depends on which database you're using (1,000 for Cassandra and 255 for PostgreSQL). Therefore, it's probably a good idea to use strings shorter than that. Messages in a different thread suggest that the Workflow ID value is not case sensitive, but I have not seen that officially documented, so it's probably best to standardize on using lowercase values.

UNANSWERED QUESTION: why is workflow type not fully-qualified here but was in the worker? IN the worker, we specified a reference to the method; here we specify the workflow type, which defaults to the *name* of the method (i.e., just the name...could be some other string if we overrode the default value).

Starting the Worker Program

- **Since Workers runs your code, there is no progress unless one is running**
  - After starting it, the Worker program outputs a few lines and then appears to do nothing
  - This is expected behavior, as it is busy polling the task queue and executing your code
  - The Worker will keep running after this Workflow completes, because it then waits for more work to appear in the task queue

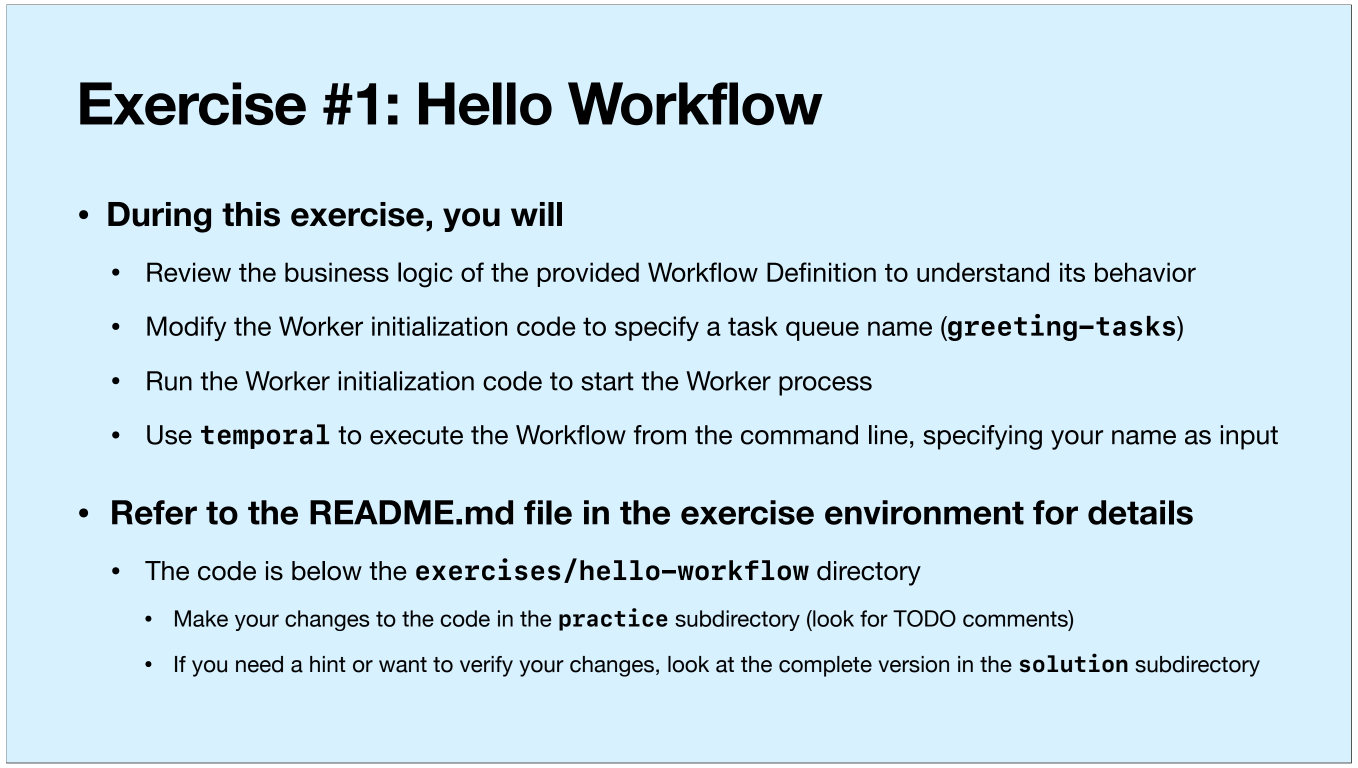


```
$ mvn compile exec:java -Dexec.mainClass="helloworkflow.HelloWorkflowWorker"
[INFO] Scanning for projects...
[INFO]
[INFO] -------------< io.temporal.learn:hello-workflow-solution >--------------
[INFO] Building hello-workflow (solution) 1.0.0-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
...
```

- Running the Worker:
  - The Worker must be running for your Workflow to execute.
  - When you start the Worker Program, it will enter a loop, polling the task queue for work.
  - The Worker may appear to be "stuck," but this is expected behavior.
  - It's important to note that the Worker's lifespan is unrelated to the Workflow's lifespan.
  - The Worker can process many Workflow Executions, while a Workflow may run for a long time.

Since the Worker runs your code, not much will happen unless it's running. Once you start the Worker Program, it will probably output a few lines and appear to be stuck. That's expected behavior, because it's running in a loop, polling the task queue looking for work to perform. Another thing to understand is that the lifespan of your Worker is unrelated to the lifespan of your Workflow. The Worker can process many Workflow Executions, particularly if they run as quickly as ours did. On the other hand, a Workflow can run for several years and it's unlikely that you'd have a single Worker process that runs for that long, due to things like operating system reboots, hardware upgrades, or power outages.

--



Exercise #1: Hello Workflow

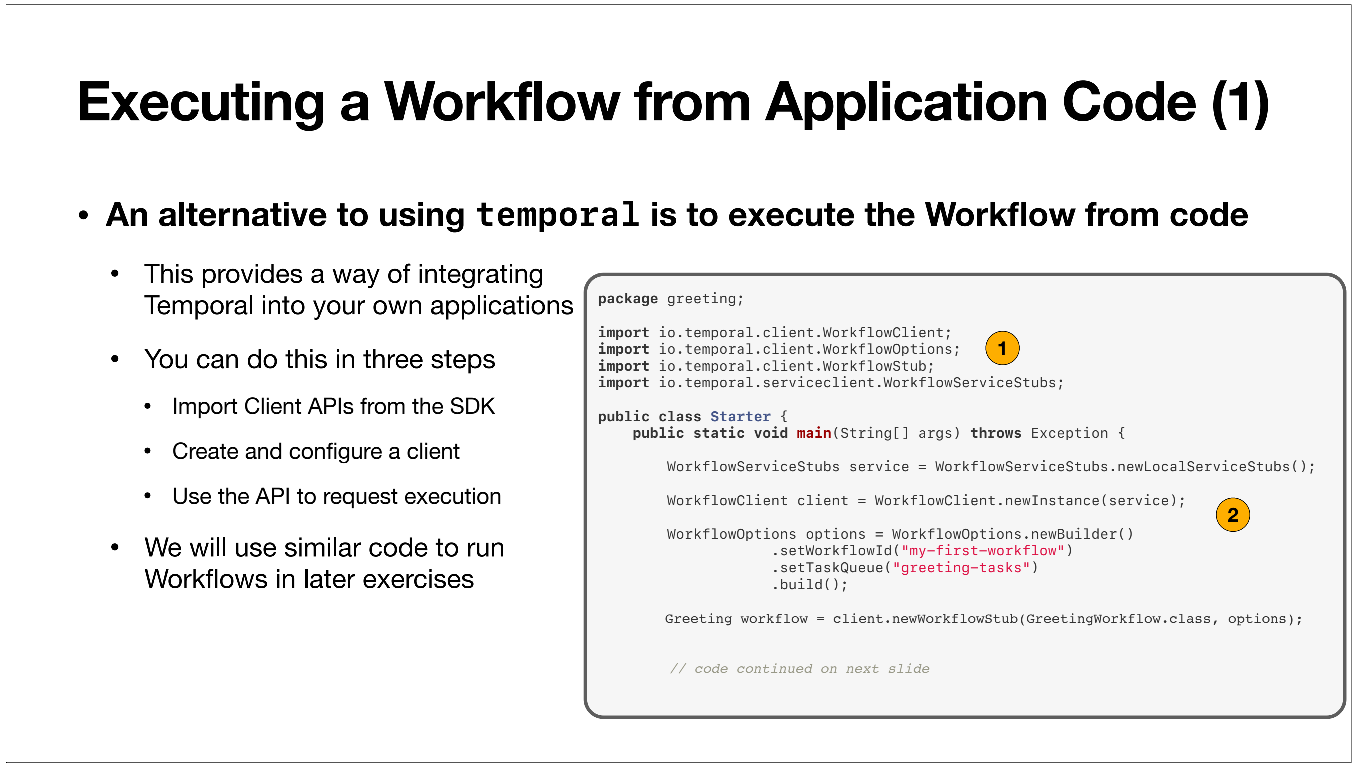
- **During this exercise, you will**
  - Review the business logic of the provided Workflow Definition to understand its behavior
  - Modify the Worker initialization code to specify a task queue name (**`greeting-tasks`**)
  - Run the Worker initialization code to start the Worker process
  - Use **`temporal`** to execute the Workflow from the command line, specifying your name as input
- **Refer to the README.md file in the exercise environment for details**
  - The code is below the **`exercises/hello-workflow`** directory
    - Make your changes to the code in the **`practice`** subdirectory (look for TODO comments)
    - If you need a hint or want to verify your changes, look at the complete version in the **`solution`** subdirectory

--
Allow 7 minutes for this exercise, but check in with students after 5 minutes just to see if they're already finished or need help.

Mention that the tctl command submits the workflow for execution, but does not wait for it to complete and therefore doesn't show the result. There's a different command for that (`temporal workflow observe`), which I mention in the exercise's `README.md` file in case the students have time, but if not, you'll get to do this using the much more convenient Web UI during the next exercise. Also, mention that later exercises will use code to start the Workflow, so they'll display the result in the terminal.

Exercise #1: Hello Workflow SOLUTION

10 minute break



Executing a Workflow from Application Code (1)

- **An alternative to using `temporal` is to execute the Workflow from code**
  - This provides a way of integrating Temporal into your own applications
  - You can do this in three steps
    - Import Client APIs from the SDK
    - Create and configure a client
    - Use the API to request execution
  - We will use similar code to run Workflows in later exercises

```
package greeting;

import io.temporal.client.WorkflowClient;
import io.temporal.client.WorkflowOptions;         1
import io.temporal.client.WorkflowStub;
import io.temporal.serviceclient.WorkflowServiceStubs;

public class Starter {
    public static void main(String[] args) throws Exception {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();

        WorkflowClient client = WorkflowClient.newInstance(service);          2

        WorkflowOptions options = WorkflowOptions.newBuilder()
                    .setWorkflowId("my-first-workflow")
                    .setTaskQueue("greeting-tasks")
                    .build();

        Greeting workflow = client.newWorkflowStub(GreetingWorkflow.class, options);


        // code continued on next slide
```

Here's the information converted into a bulleted list outline:

Executing a Workflow Using Temporal SDK

* Two ways to execute a Workflow:
  * Using the temporal command-line tool.
  * Using the APIs provided by the Temporal SDK to start it from code.

* Advantages of executing from code:
  * Integration into your own applications.
  * Allows executing or terminating a Workflow in response to user activity.
  * Enables complex interactions.

* Example code for executing a Workflow provided.
  * Code included in the "samples" subdirectory for reference.
  * Similar code used to start Workflows in future exercises.
  * Eliminates the need to type out long tctl commands each time.

* "main" method created within the "greeting" package to make the example runnable.

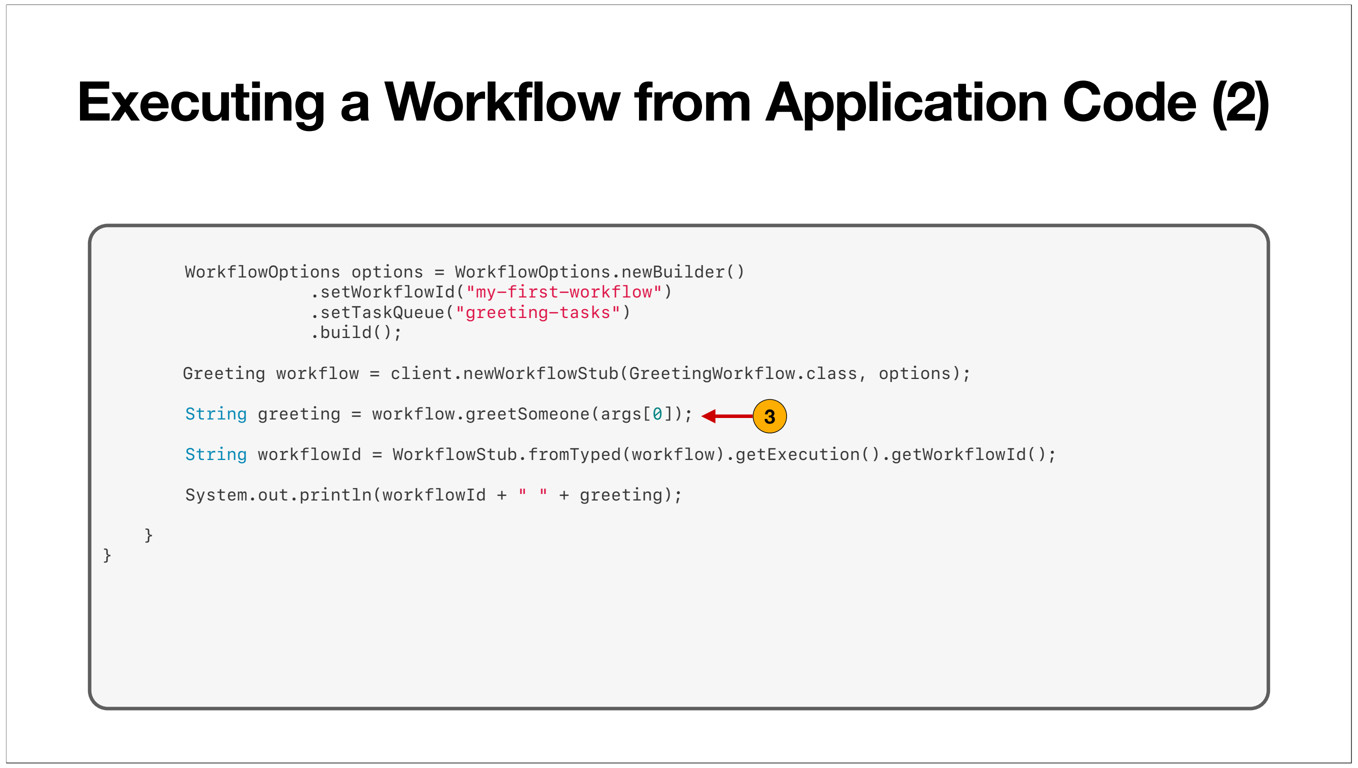
* Importance of the "WorkflowClient" package from the Temporal SDK.
  * Used to interact with Temporal from code.

* Steps for executing a Workflow:
  1. Setting Workflow Execution options.
     * Specifies Workflow ID and Task Queue.
  2. Registering the Workflow with the client.
     * Includes execution options and a reference to the Workflow Definition Interface.

I mentioned a moment ago that I was going to cover two ways of executing a Workflow. You've already learned about -- and practiced -- the first one, which is to use the temporal command-line tool. The second is to use the APIs provided by the Temporal SDK to start it from code. Although both approaches accomplish the same result, doing it from code provides a way of integrating Temporal into your own applications. For example, you might execute or terminate a Workflow in response to some user activity, such as clicking a button in a Web or mobile app.

I've written some example code for executing a Workflow and I'll explain how it works. It's a little too big to fit on a single slide, so what you see here is the first half. By the way, I've included this code in the "samples" subdirectory (samples/greeting/src/main/java/greeting/Starter.java) on the exercise environment, for reference, and you'll use similar code to start Workflows in future exercises to save you from having to type out long tctl commands each time.

In order to make this example runnable, I created a "main" method within the "greeting" package. I imported a few things, but the important one is the "WorkflowClient" package from the Temporal SDK. Then I create the client. The code on the next slide is somewhat more interesting...
--



Executing a Workflow from Application Code (2)

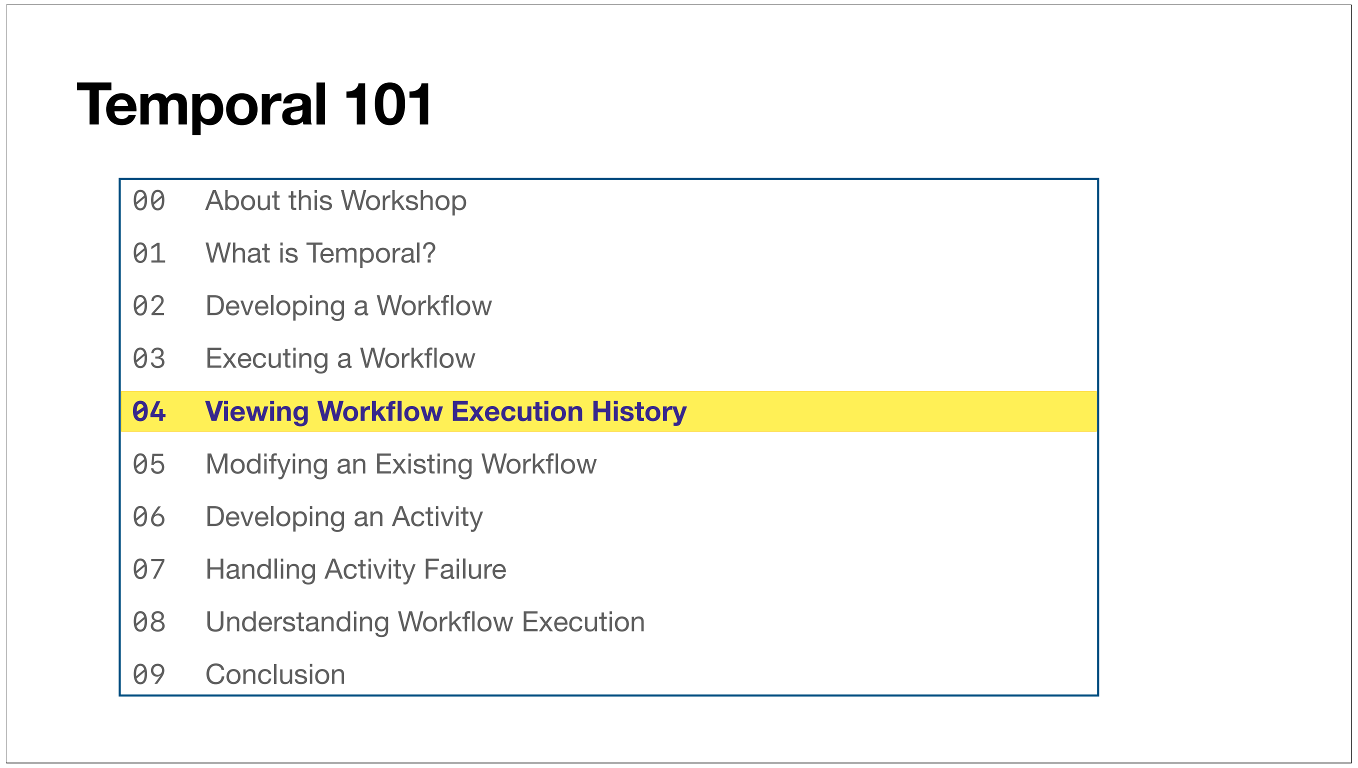
```
        WorkflowOptions options = WorkflowOptions.newBuilder()
                    .setWorkflowId("my-first-workflow")
                    .setTaskQueue("greeting-tasks")
                    .build();

        Greeting workflow = client.newWorkflowStub(GreetingWorkflow.class, options);

        String greeting = workflow.greetSomeone(args[0]); ← 3

        String workflowId = WorkflowStub.fromTyped(workflow).getExecution().getWorkflowId();

        System.out.println(workflowId + " " + greeting);

    }
}
```

3. Calling the Workflow Method.
 * Uses the WorkflowClient to call the Workflow method.
 * Passes input data.
4. Handling input for the Workflow.
 * Input can be provided from various sources (e.g., command-line argument, database, user input).
 * No need to specify input in JSON format; allowed types are automatically converted to JSON.
5. Blocking call nature of Workflow Method.
 * Program execution halts until the Workflow completes.
 * Workflow Execution can also be asynchronous, covered later in the course.
6. Result of Workflow Method call.
 * If Workflow Execution completes successfully, the "greeting" variable is assigned its output.

Here is the rest of the code. The first thing we do is to set our Workflow Execution options, which specifies the name of the Workflow ID and the name of the Task Queue. Next we register our Workflow with the client, including the options that we set and a reference to the Workflow Definition Interface.

Next, we use the client to call the Workflow Method, greetSomeone, passing in a context object, our options, a reference to our Workflow method, and the input. In this example, we get that input -- a person's name -- as a command-line argument to this program, although you could use any mechanism you like for getting the input; for example, from a database or configuration file or user input in your application. By the way, when we start a Workflow from code, we don't have to specify the input in JSON format like we did on the command line. We can just use any of the allowed types in Go, such as numbers or strings or structs, and the SDK will convert it into JSON for us automatically.

I want to emphasize something here. The Workflow Method call here is a blocking call, meaning the program execution will halt until the Workflow completes. Workflow Execution can be called asynchronously, which we will cover later in the course. If the Workflow Execution completes successfully, the "greeting" variable will be assigned its output.
--



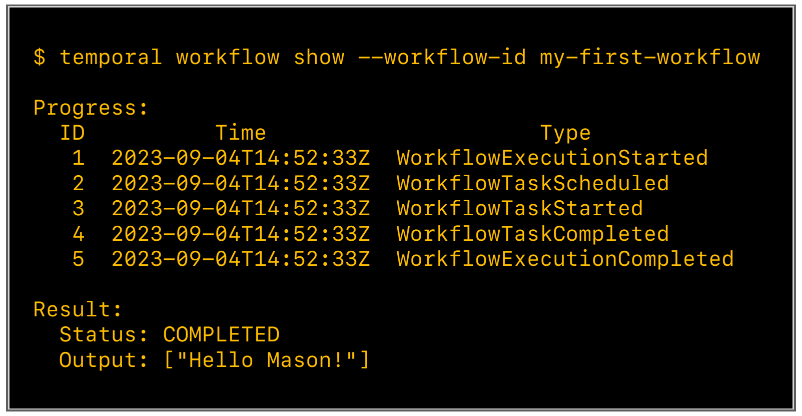
Temporal 101

--

AUDIENCE PARTICIPATION
Before we continue, can anyone name the two approaches I explained for executing a Workflow?

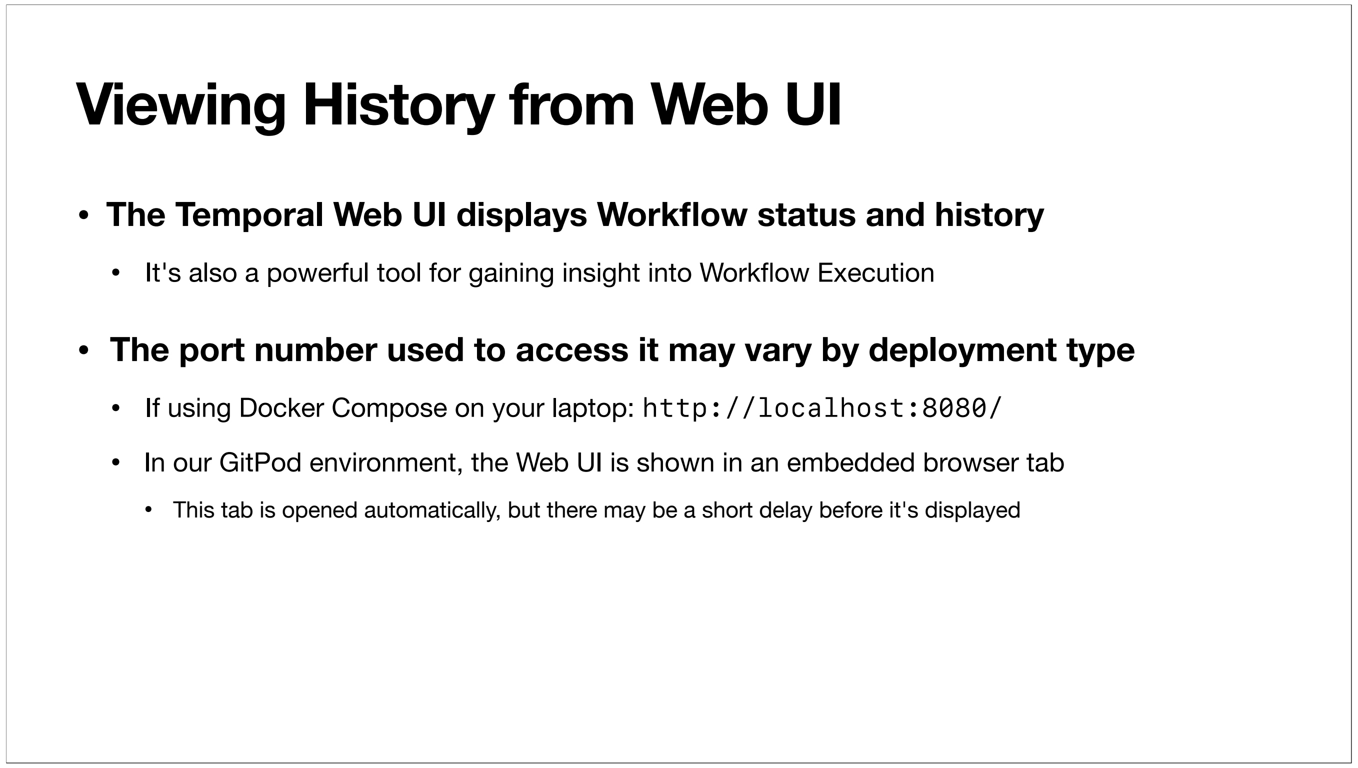
Answer: From the command-line (using tctl) and from code. They both accomplish the same thing, but doing it through code allows you to integrate it into your application; for example, getting input from a user, passing it to a Workflow, and doing something with the result.

Viewing Workflow History with `temporal`



```
$ temporal workflow show --workflow-id my-first-workflow

Progress:
  ID          Time                     Type
   1  2023-09-04T14:52:33Z  WorkflowExecutionStarted
   2  2023-09-04T14:52:33Z  WorkflowTaskScheduled
   3  2023-09-04T14:52:33Z  WorkflowTaskStarted
   4  2023-09-04T14:52:33Z  WorkflowTaskCompleted
   5  2023-09-04T14:52:33Z  WorkflowExecutionCompleted

Result:
  Status: COMPLETED
  Output: ["Hello Mason!"]
```

You can use temporal to display the detailed history of a Workflow Execution with the command shown here. However, once you see the Temporal Web UI, I suspect you'll probably prefer using it instead.



Viewing History from Web UI

- **The Temporal Web UI displays Workflow status and history**
  - It's also a powerful tool for gaining insight into Workflow Execution
- **The port number used to access it may vary by deployment type**
  - If using Docker Compose on your laptop: `http://localhost:8080/`
  - In our GitPod environment, the Web UI is shown in an embedded browser tab
    - This tab is opened automatically, but there may be a short delay before it's displayed

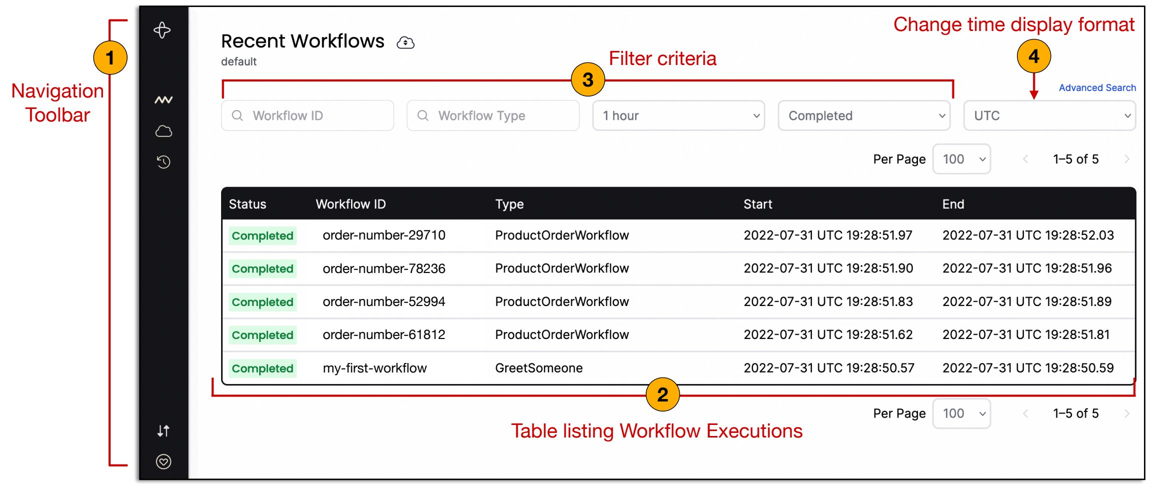
When I explained the basic architecture of Temporal, I mentioned the Web UI. It not only displays the status and history of your Workflow Executions, it's also a powerful tool for understanding what's happening during your Workflow Executions.

The method you used to access it for your *own* Temporal instance will vary based on the type of deployment. For example, if you're using Temporal Cloud, you'll access it through a secured connection to the standard HTTPS port on temporal.cloud.ui, which will require you to log in.

If you used Docker Compose to deploy a development cluster to your laptop, you can access it through localhost on port 8080. If you're using a self-hosted production cluster, ask your administrator for the hostname and port number.

In the exercise environment for this course, the Web UI is shown automatically, although as I mentioned during the exercise environment orientation, it won't become visible until the environment is fully initialized.

Web UI: Main Page



Recent Workflows
default

Workflow ID | Workflow Type | 1 hour | Completed | UTC

Per Page 100 1–5 of 5

Status	Workflow ID	Type	Start	End
Completed	order-number-29710	ProductOrderWorkflow	2022-07-31 UTC 19:28:51.97	2022-07-31 UTC 19:28:52.03
Completed	order-number-78236	ProductOrderWorkflow	2022-07-31 UTC 19:28:51.90	2022-07-31 UTC 19:28:51.96
Completed	order-number-52994	ProductOrderWorkflow	2022-07-31 UTC 19:28:51.83	2022-07-31 UTC 19:28:51.89
Completed	order-number-61812	ProductOrderWorkflow	2022-07-31 UTC 19:28:51.62	2022-07-31 UTC 19:28:51.81
Completed	my-first-workflow	GreetSomeone	2022-07-31 UTC 19:28:50.57	2022-07-31 UTC 19:28:50.59

Per Page 100 1–5 of 5

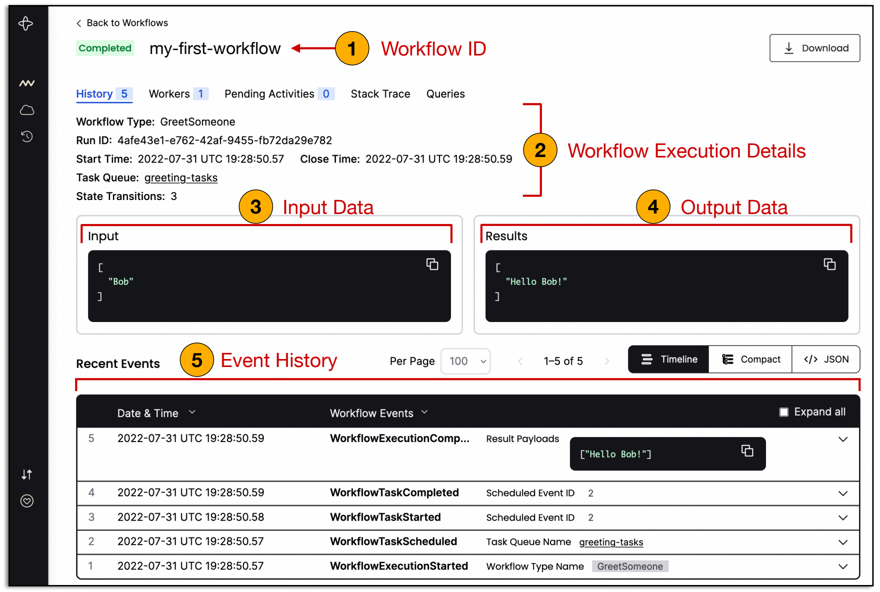
The main page of the Web UI shows recent Workflow Executions. On the left, you have a toolbar that lets you navigate to other pages, although we'll mainly use this page in the Temporal 101 course.

Most the page is consumed by a table that lists Workflow Executions. Above that table, across the top, there are some fields we can use to filter the table; by specifying any combination of a Workflow ID, Workflow Type, time window, or Execution Status. Next to that, on the right, you can change the display format for the timestamps in the table, switching from the default UTC timezone to your local timezone or even a relative time, such as 10 hours ago.

In addition to displaying details about Workflow Executions that are currently running, the Temporal Cluster also displays the history of past Workflow Executions, up to a configurable retention period.

--

Web UI: Workflow Execution Detail Page



my-first-workflow

Workflow Type: GreetSomeone
Run ID: 4afe43e1-e762-42af-9455-fb72da29e782
Start Time: 2022-07-31 UTC 19:28:50.57 Close Time: 2022-07-31 UTC 19:28:50.59
Task Queue: greeting-tasks
State Transitions: 3

Input

```
[
  "Bob"
]
```

Results

```
[
  "Hello Bob!"
]
```

Recent Events

Per Page 100 1–5 of 5

	Date & Time	Workflow Events		
5	2022-07-31 UTC 19:28:50.59	WorkflowExecutionComp...	Result Payloads	["Hello Bob!"]
4	2022-07-31 UTC 19:28:50.59	WorkflowTaskCompleted	Scheduled Event ID	2
3	2022-07-31 UTC 19:28:50.58	WorkflowTaskStarted	Scheduled Event ID	2
2	2022-07-31 UTC 19:28:50.57	WorkflowTaskScheduled	Task Queue Name	greeting-tasks
1	2022-07-31 UTC 19:28:50.57	WorkflowExecutionStarted	Workflow Type Name	GreetSomeone

When you click one of the items in that table, the Web UI will display the detail page for the Workflow Execution you selected.

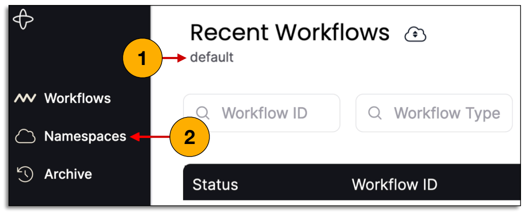
The Workflow ID is shown at the top, and below that, you'll see some other details, such as the Workflow Type and Task Queue.

What you see below that will help you realize what a useful tool this can be. On the left, it shows the input data for the workflow, and on the right, the output data. At the bottom of the page, it shows the Event History, so you can see exactly what took place at each step in the Workflow Execution. This is something you'll learn more about during this course.

--

Namespaces

- **The Web UI lists recent Workflow Executions within a given *namespace***
  - You can see the selected namespace (1) and switch among available namespaces (2)
- **Namespaces are a means of isolation within a Temporal cluster**
  - Used to logically separate Workflows according to your needs
    - For example, by lifecycle (development vs. production) or department (Marketing vs. Accounting)
  - Some settings are applied at a per-namespace level
  - The default namespace is named `default`



* Namespace in Temporal serves as a means of isolation and organization.
* Similar to namespaces or packages in programming languages, namespaces provide logical separation within Temporal.
* Use namespaces to isolate Workflows based on various criteria, such as status, teams, or departments.
* Some configuration options and concepts are applied on a per-namespace level, not globally across the entire cluster.
* For example, Temporal ensures that there is only one active Workflow Execution with a specific Workflow ID within a given namespace.

* In the Web UI:
  * The namespace is displayed at the top of the main page (labeled "1").
  * Clicking on "namespaces" in the left navigation allows you to view and switch between available namespaces.
  * The default namespace is named "default."

* To work with namespaces:
  * Use `tctl` to register additional namespaces.
  * Your code specifies the namespace it wants to use using a configuration option when creating a client.

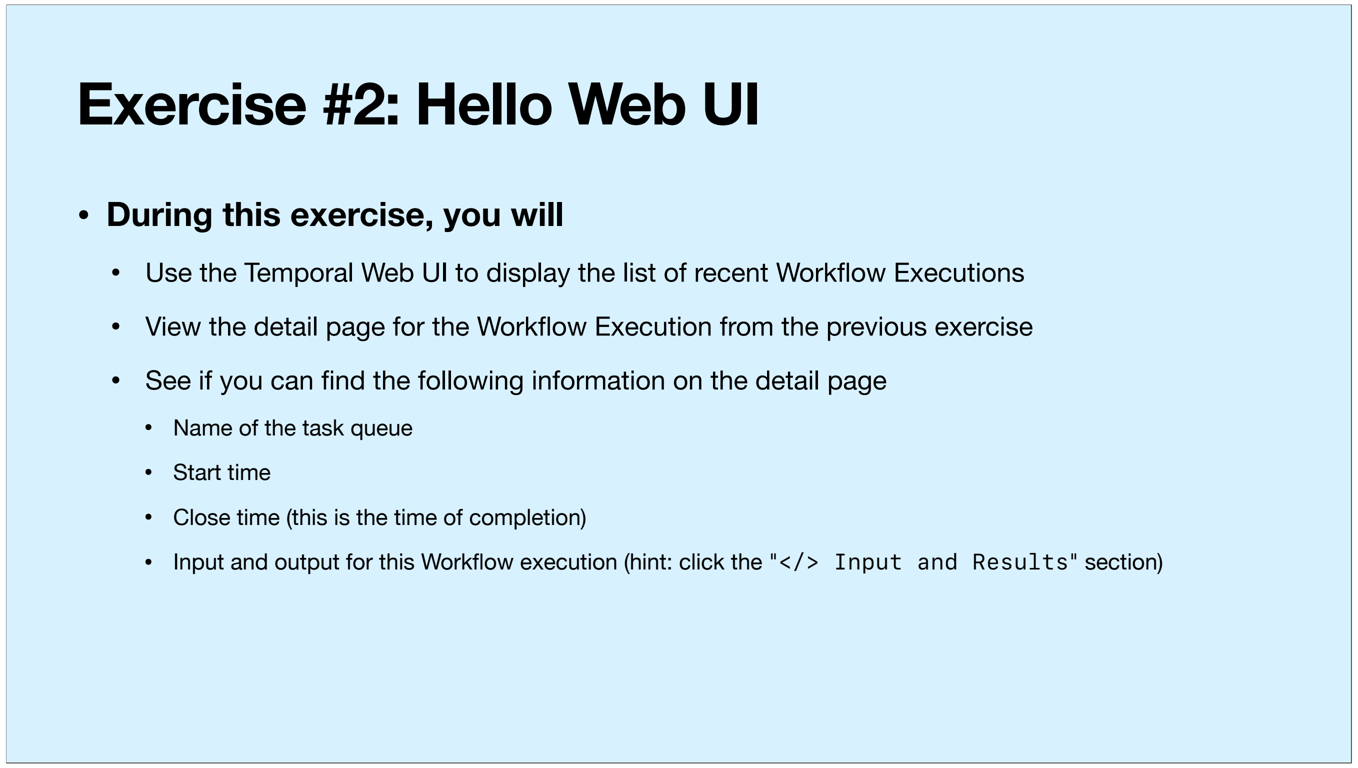
I skipped over one thing you'll see on the main page, because it's important enough to call out separately. I said earlier that the Web UI shows a table of Workflow Executions, but it's actually showing a table of Workflow Executions *within a given namespace*.

A namespace provides a means of isolation with Temporal, much like namespaces or packages in some programming languages provide isolation between different parts of the code. They allow you to logically separate things, using whatever meets your needs. For example, you might isolate Workflows based on their status by having a "development" namespace and a "production" namespace. Or you might separate them by team or department, in which case you might have one namespace for Marketing and another for Accounting.

One aspect of this isolation is that some configuration options or concepts are applied on a per-namespace level, rather than for the entire cluster. For example, remember the Workflow ID specified when starting a Workflow? Temporal guarantees that there is only one Workflow Execution with a given Workflow ID currently running within any given namespace. If you tried to use the same Workflow ID to start a Workflow while another was already running in that namespace, it would fail with a "Workflow Execution already started" error.

The Web UI shows the namespace near the top of the main page, which I've labeled with a "1" here. Clicking on "namespaces" in the left navigation will show the available namespaces and let you switch between them. As you might guess from the Web UI, the default namespace is named "default." You can use tctl to register additional namespaces and your code will specify the namespace it wants to use using a configuration option provided when creating a client.

--

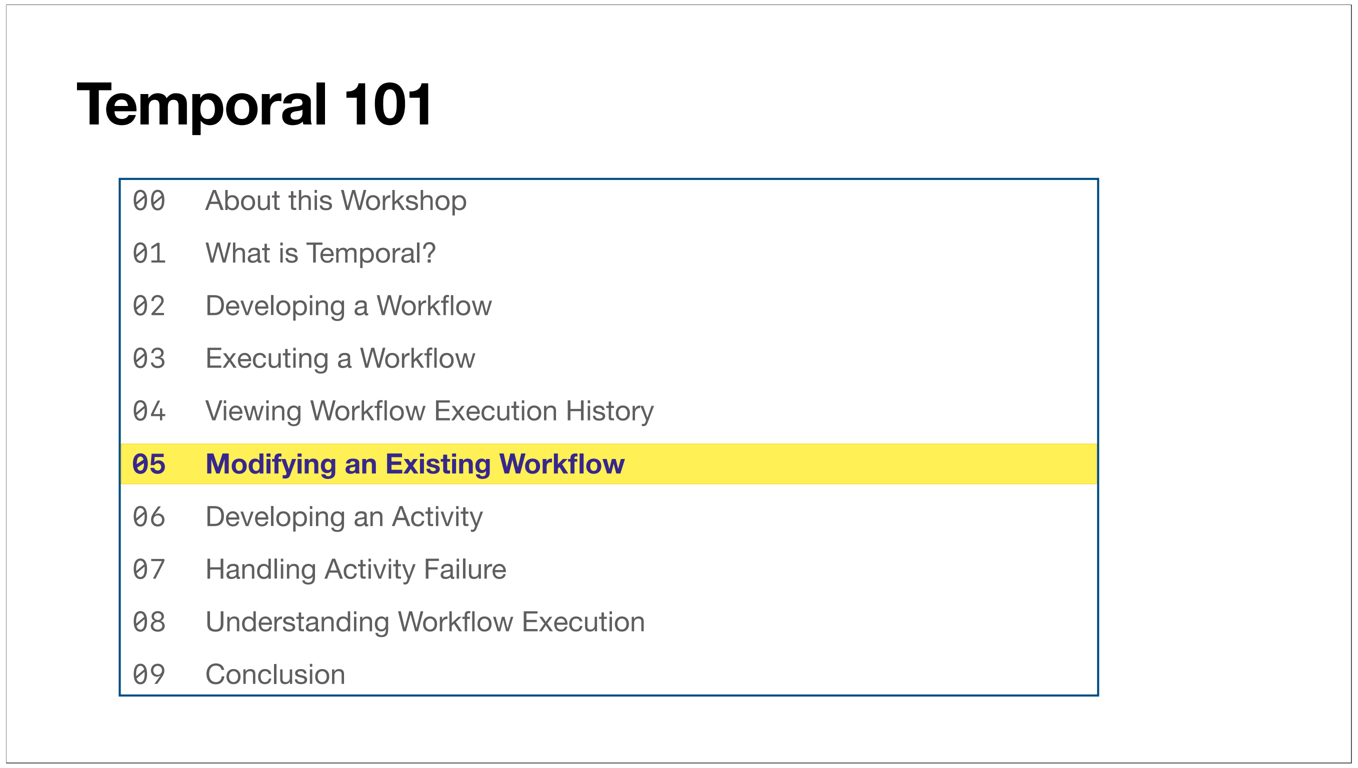


Exercise #2: Hello Web UI

- **During this exercise, you will**
  - Use the Temporal Web UI to display the list of recent Workflow Executions
  - View the detail page for the Workflow Execution from the previous exercise
  - See if you can find the following information on the detail page
    - Name of the task queue
    - Start time
    - Close time (this is the time of completion)
    - Input and output for this Workflow execution (hint: click the "`</> Input and Results`" section)

NOTE: Allow 5 minutes for this exercise. Also mention that there is no code for this exercise; there's just a separate exercise directory so we have a place to put the README file, although its content is the same as what you see here.

Exercise #2: Hello Web UI SOLUTION

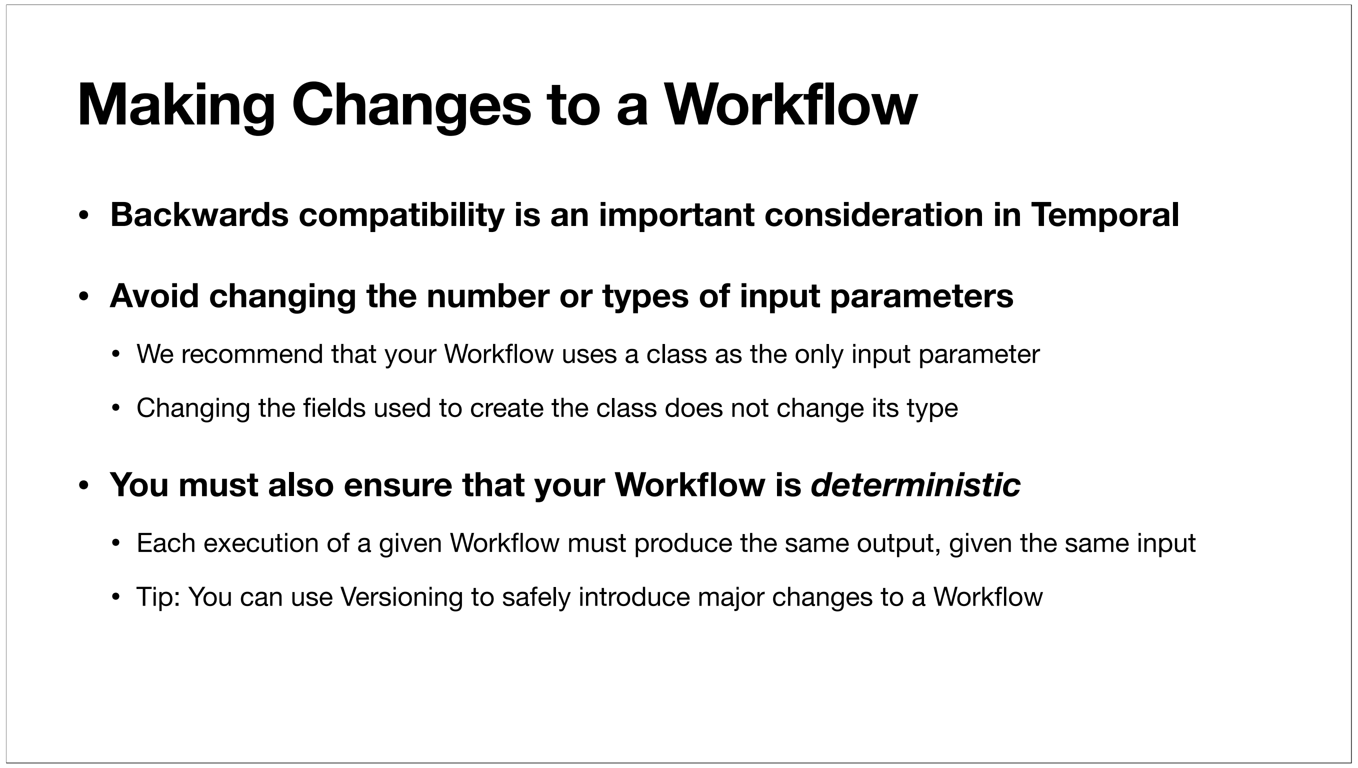


Temporal 101

--

AUDIENCE PARTICIPATION
Before we continue, can anyone speculate about what would happen if you execute a Workflow but don't have a Worker running?

Answer: Not much, since the Worker is responsible for executing your code. This used to be a common problem for people getting started with Temporal: You'd run the Workflow and be confused why you didn't see any progress, only to realize that there wasn't a Worker running. Fortunately, that's no longer much of a problem, thanks to the incredible people who designed our Web UI, because it now shows a very obvious warning when you look at the detail page for a Workflow that doesn't have any Workers running.



Making Changes to a Workflow

- **Backwards compatibility is an important consideration in Temporal**
- **Avoid changing the number or types of input parameters**
  - We recommend that your Workflow uses a class as the only input parameter
  - Changing the fields used to create the class does not change its type
- **You must also ensure that your Workflow is *deterministic***
  - Each execution of a given Workflow must produce the same output, given the same input
  - Tip: You can use Versioning to safely introduce major changes to a Workflow

* Ensure Workflow executions produce the same output for the same input, avoiding changes to input parameters.
* Use a single input parameter (e.g., a class) to allow future changes without altering the parameter structure.
* Workflows must be deterministic, guaranteeing consistent output for the same input.
* Avoid non-deterministic actions like random number generation; use SDK-provided alternatives.
* Handle non-deterministic operations (e.g., file access, network services) using Temporal features (explained later).
* Plan for major Workflow Definition changes during ongoing executions.
* Deploy changes that don't affect determinism.
* Utilize the SDK's "Versioning" feature for non-deterministic changes.
* Consider backward compatibility with determinism, Activities, and Timers in future courses.
* Restart workers after making changes for consistency in Workflow executions.

This is a topic that we'll explore more deeply in advanced training, but it's important, so I want to briefly mention it here, too.

In Temporal, you can execute a given Workflow Definition hundreds, thousands, or millions of times. If the execution fails, the Temporal will reconstruct the Workflow's state before the failure, and then continue with the execution. It's premature to cover the details of how that works, but it has some implications on how you develop and maintain the Workflow Definitions.

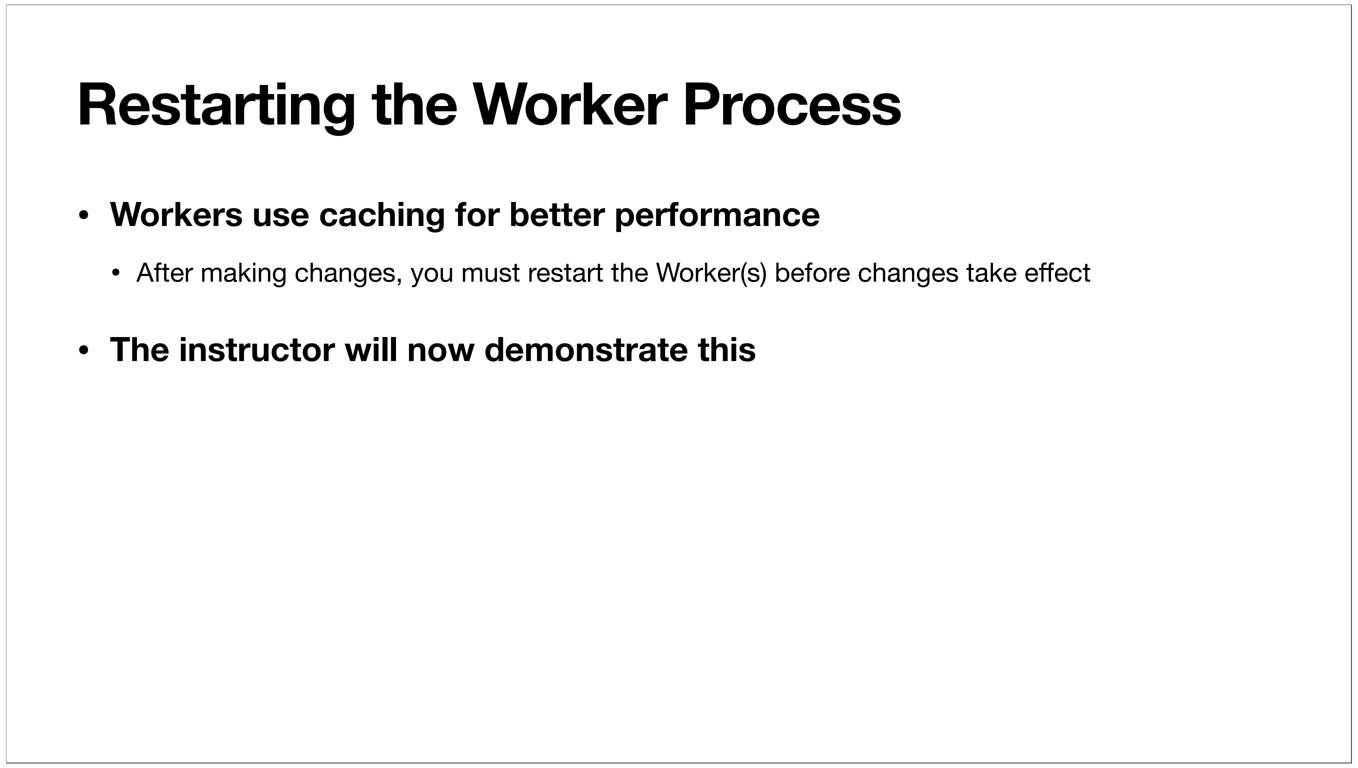
The general rule is that, for a given input, the Workflow must produce the same output. This means that you should avoid changing the number or types of input parameters. Although we don't follow the advice in this course for the sake of simplicity, Temporal recommends that your Workflow Method takes a single input parameter -- a class -- rather than multiple input parameters. This is because changing which fields are part of the struct doesn't change the type of the struct itself, so this is sort of a backdoor way that lets you make changes over time.

Also, your Workflow must be deterministic. Temporal has a specific definition for this, but it requires some detailed knowledge of Workflow Execution, so I'll generalize for now, as I'll cover this topic in much greater depth in a subsequent training course. For now, we can view determinism as a requirement that each execution of a given Workflow must produce the same output, given the same input. This means that you shouldn't do things like work with random numbers in your Workflow code. If you need to do things such as working with random numbers, the SDK provides safe alternatives. There is also a way to handle *operations* that aren't deterministic, such as executing that accesses files or network services, and we'll cover that in the next section.

Since Workflow Executions might run for months or years, it's possible that you'll need to make major changes to a Workflow Definition while there are already executions running based on the current definition of that Workflow. If these changes do not affect the deterministic nature of the Workflow, you can simply deploy them. However, it is possible to use the SDK's "Versioning" feature to identify when a non-deterministic change is introduced, which allows older executions to use the original code and new executions to use the modified version. This is a topic we'll explore more in a future course, but in the meantime I recommend

--

NOTE: There is probably not much we can say about backwards compatibility at this point, at least not in terms of changes related to determinism, since that requires coverage of Activities and ideally also Timers. Focus here and now is on getting them to see that you must restart workers after making a change



Restarting the Worker Process

- **Workers use caching for better performance**
  - After making changes, you must restart the Worker(s) before changes take effect
- **The instructor will now demonstrate this**

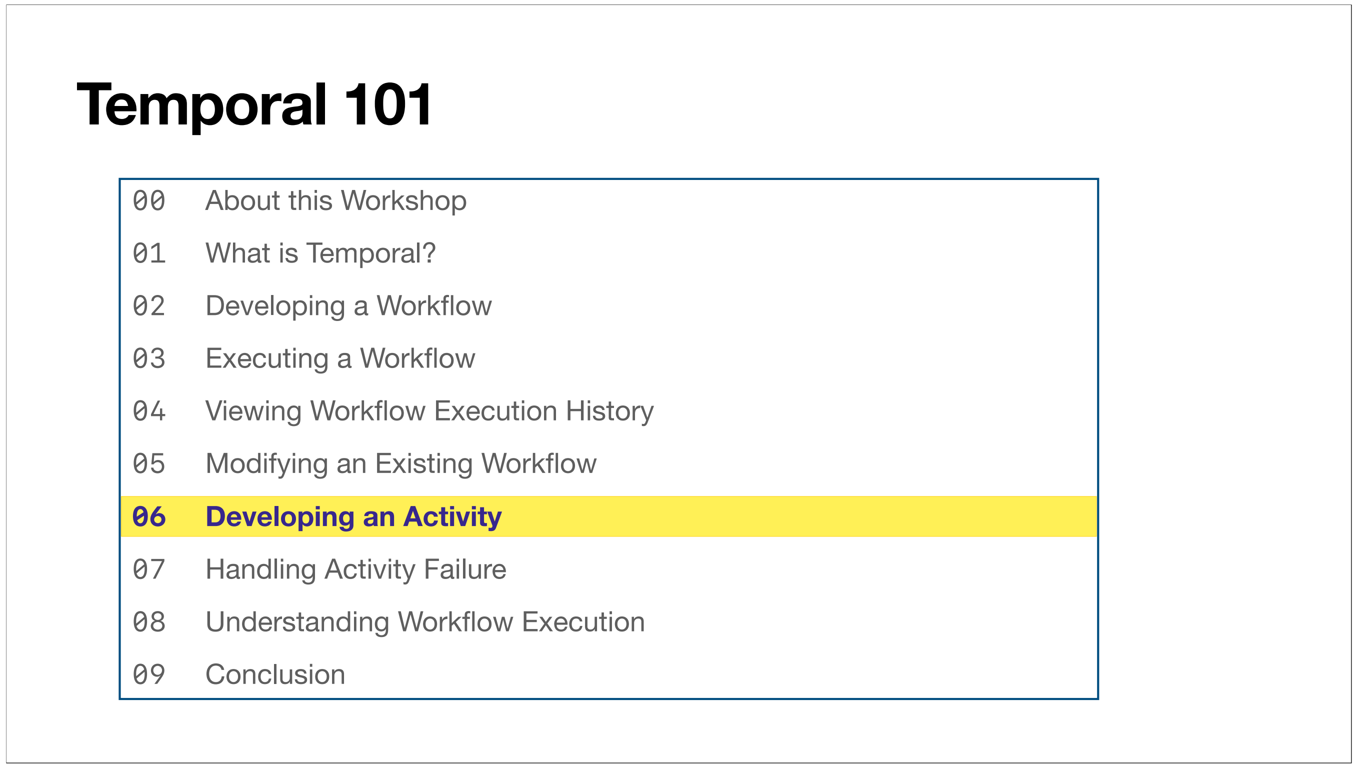
In order to provide the best possible performance, Temporal Workers cache the application state. A consequence of this is that the modifications you make to the code won't take effect until you restart the Workers that are running your application.

Let me demonstrate this now.

--

DEMO: Run hello-workflow solution code, then modify it to change the greeting from English to Spanish. Use option + 1 to type upside-down exclamation mark (¡), specify a different name to distinguish from past WFEs (NOTE: to type accented á in Tomás, press Option + e, then type an "a")

The Worker caches the code, so the changes you made didn't take effect. To make sure the Workers have the latest version of the code, you have to restart them after making a change.

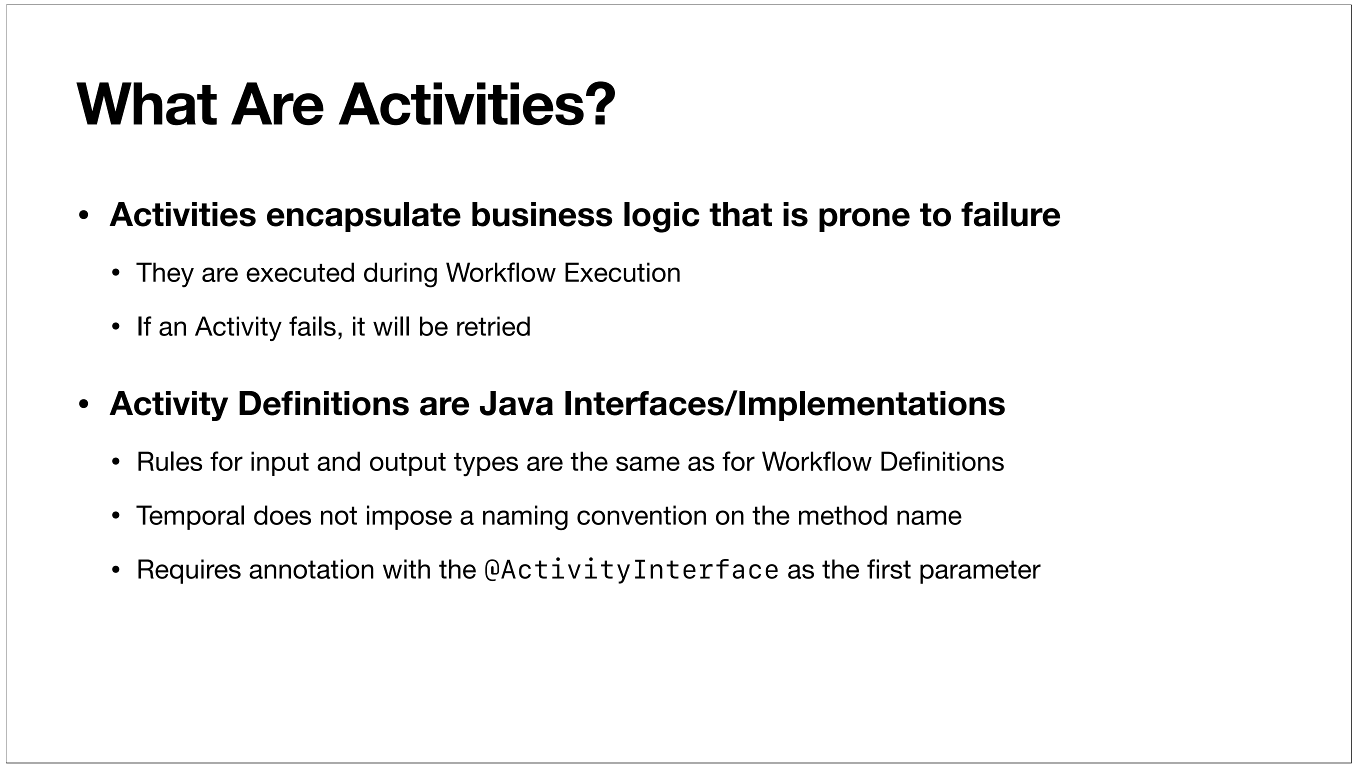


Temporal 101

--

AUDIENCE PARTICIPATION
Before we continue, what must you do after changing your application code; for example, modifying your Workflow or Activity method?

Answer: You must restart the Worker for your change to takes effect.



What Are Activities?

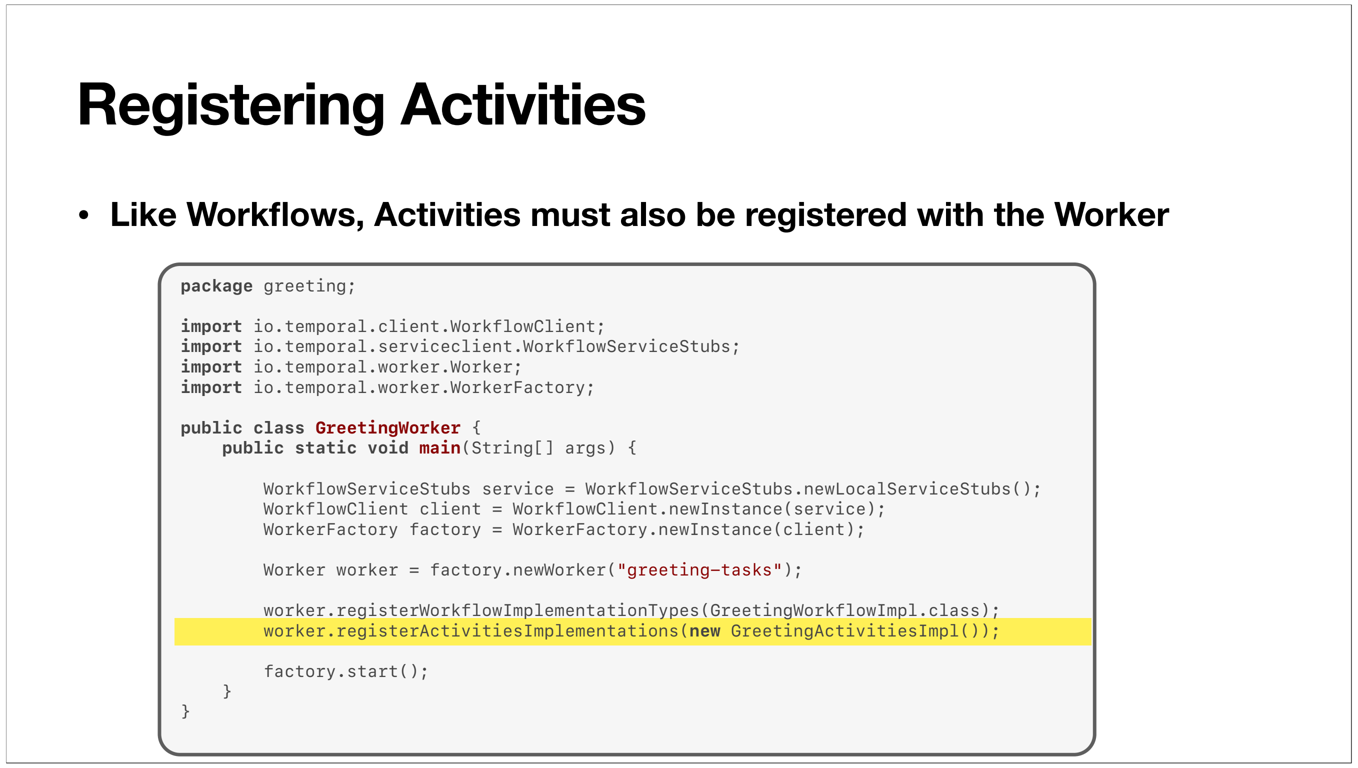
- **Activities encapsulate business logic that is prone to failure**
  - They are executed during Workflow Execution
  - If an Activity fails, it will be retried
- **Activity Definitions are Java Interfaces/Implementations**
  - Rules for input and output types are the same as for Workflow Definitions
  - Temporal does not impose a naming convention on the method name
  - Requires annotation with the `@ActivityInterface` as the first parameter

- Workflow code must be deterministic and produce consistent output given the same input.
- Interaction with the external world, such as accessing files or network resources, is prohibited due to potential unavailability.
- The need for external interactions arises in business logic.
- Solution: Utilize Activities to address non-deterministic operations.
- Activities are executed as part of Workflow Execution and are retried upon failure.
- Activities allow encapsulation of code requiring external interactions or potential failures.
- Activity Definitions consist of an Interface and Implementation.
- Activity input/output follows the same type rules as Workflows (e.g., JSON-compatible types).
- The naming of Activity methods is flexible.
- Activities must be available to the Worker at runtime, similar to Workflow definitions.

I explained earlier that Workflow code must be deterministic, and must produce the same output each time, given the same input. This also implies that it can't interact with the outside world, such as accessing files or network resources, because those might not be available at a given point in time. However, our business logic may require that we do such things. So, how do we reconcile this?

The answer is Activities. In general, any operation that introduces the possibility of failure should be done as part of an Activity, rather than as part of the Workflow directly. While Activities are executed as part of Workflow Execution, they have an important characteristic: they're retried when they fail. So if we have an extensive Workflow that needs to access a service, and that service happens to become unavailable, we don't want to re-run the entire Workflow. Instead, we just want to retry the part that failed, so we encapsulate that code into an Activity and Temporal will execute it, retry if necessary, and then continue with the rest of the Workflow once the Activity completes successfully.

Just like a Workflow Definition is an Interface and its Implementation, an Activity Definition is also an Interface and its Implementation, and has the same rules about types used as input and output as the Workflow does; for example, anything that converts to JSON is fine. Temporal doesn't care what you name the method. Naming doesn't matter, but the Activity needs to be available to the Worker at runtime, just like your Workflow definition.
--



Registering Activities

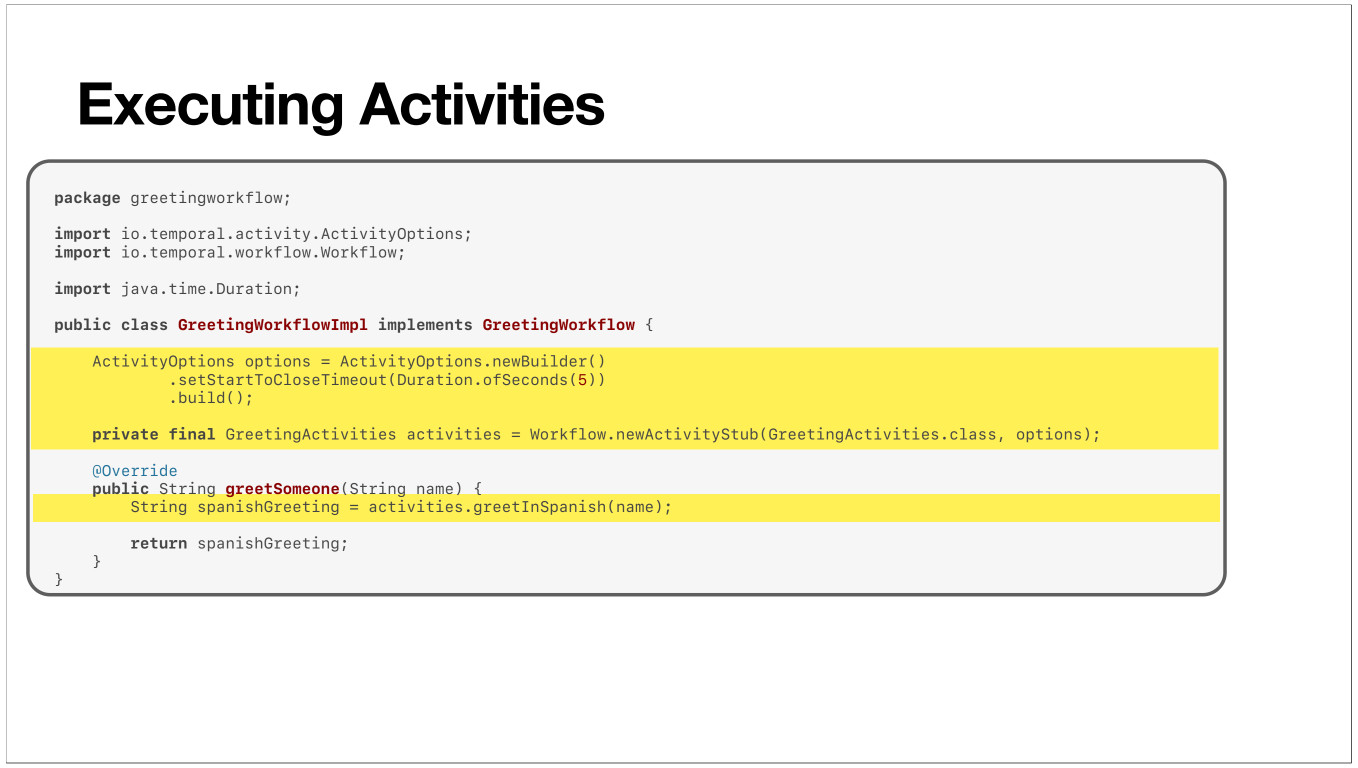
- **Like Workflows, Activities must also be registered with the Worker**

```
package greeting;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {
    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);
        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

You may recall that we had to register our Workflow Type with the Worker. We have to do the same thing for Activities, and the process is almost identical. The value we specify here is a reference to the Activity method.

By the way, the example I'll show on the next few slides comes from the upcoming exercise. Rather than hardcode a greeting in Spanish into our Workflow, we use a microservice to access it.

--



Executing Activities

```java
package greetingworkflow;

import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

import java.time.Duration;

public class GreetingWorkflowImpl implements GreetingWorkflow {

    ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities = Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);

        return spanishGreeting;
    }
}
```

- Workflow code for incorporating Activities execution
  - Defines options for ActivityExecution
    - Includes a "StartToClose" timeout
      - Recommended to set longer than the expected activity execution time
      - Used to detect Worker crashes and handle failed attempts
    - Other timeout values can be set but less frequently used
  - Creates an instance variable "activities"
    - Passes ActivityClass and execution options
- Calls the Activity Method
  - Requests execution of "greetInSpanish" Activity
  - Blocks until completion
  - Note: Workflow does not execute the activity directly
    - Requests the cluster to schedule activity execution
    - Cluster creates a task for activity execution
    - A Worker subsequently picks up the task and invokes the Activity method

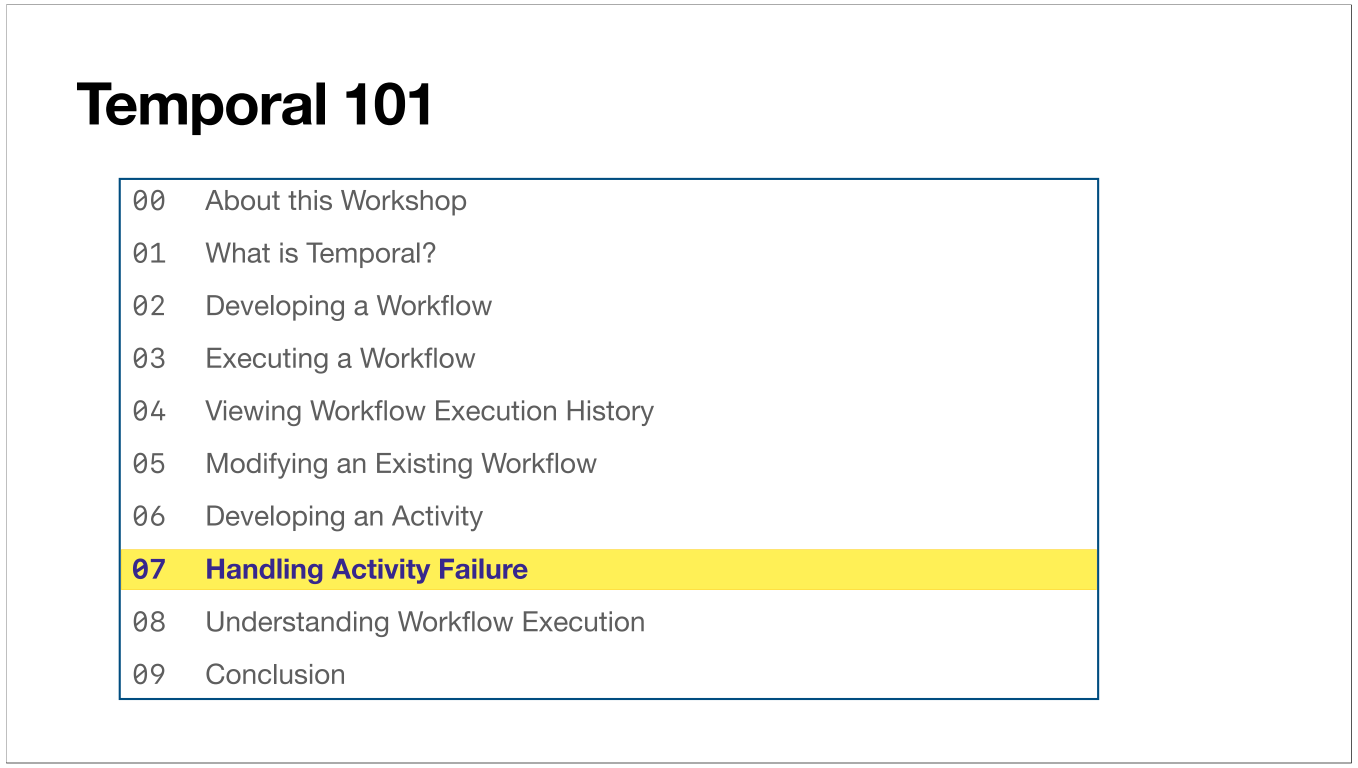
---

Here's the Workflow code, which illustrates how we incorporate the execution of Activities. I've highlighted two sections, both of which are new.

The first one defines some options for ActivityExecution, crucially, this includes a timeout called "StartToClose" timeout, which we always recommend setting. Its value should be longer than the maximum amount of time you think the execution of the Activity should take. The reason we do this is so we can detect to a Worker that crashed, in which case the Temporal Cluster will consider that attempt failed, and will create another task that a different Worker could pick up. There are other timeout values we could set here, but they're less frequently used and not relevant to this course. It then creates an instance variable named "activities", passing in the ActivityClass and options that should dicate its execution.

The second one calls the Activity Method. This requests an execution of the "greetInSpanish" Activity and will block until completion. By the way, I want to point out that the Workflow does not execute the activity, in other words, it doesn't invoke the Activity Method. Instead, it's making a request to the cluster to *schedule execution* of the activity. The cluster will respond by creating a task representing activity execution, which a Worker will subsequently pick up, and that Worker will then invoke the Activity method.

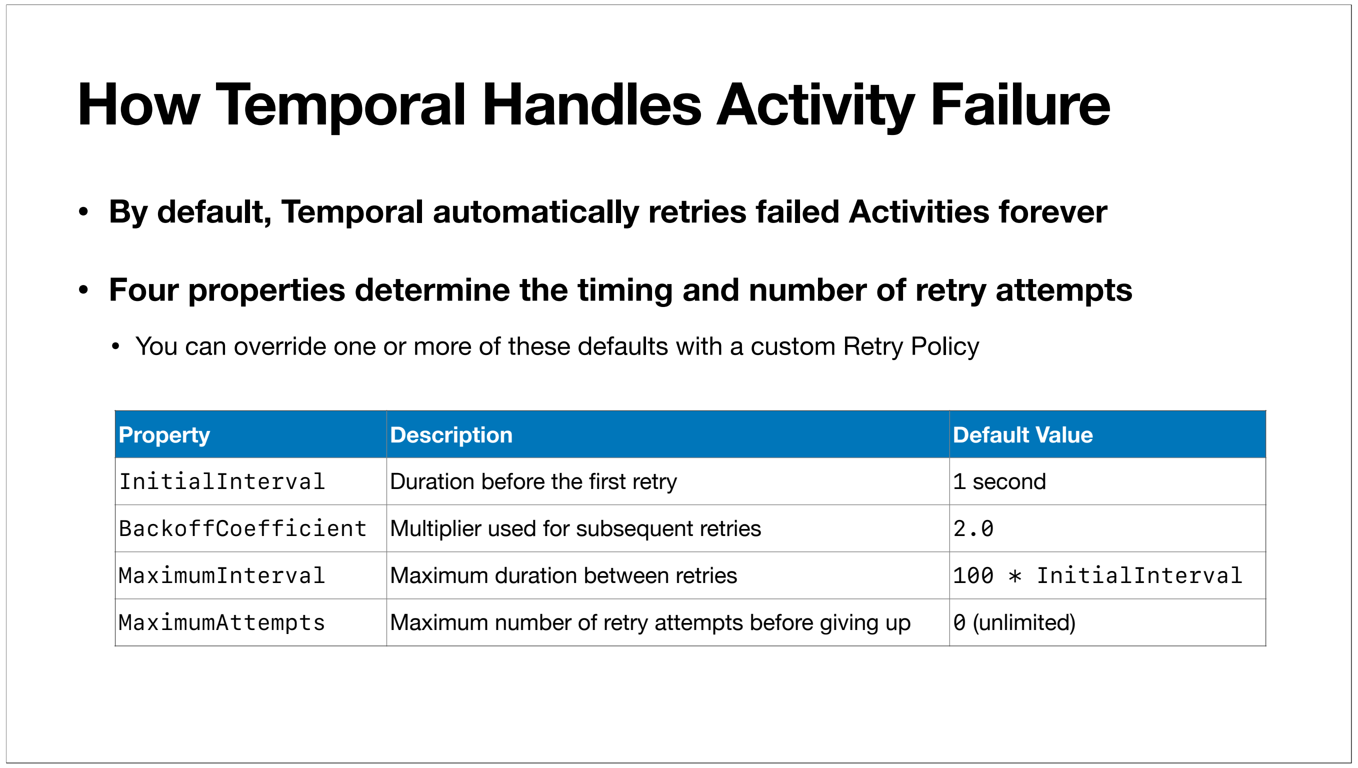
--



Temporal 101

--

AUDIENCE PARTICIPATION
Before we continue, can anyone tell me why you might need to execute part of your business logic as an Activity, as opposed to just putting that code directly in the Workflow itself?



How Temporal Handles Activity Failure

- **By default, Temporal automatically retries failed Activities forever**
- **Four properties determine the timing and number of retry attempts**
  - You can override one or more of these defaults with a custom Retry Policy

Property	Description	Default Value
`InitialInterval`	Duration before the first retry	1 second
`BackoffCoefficient`	Multiplier used for subsequent retries	`2.0`
`MaximumInterval`	Maximum duration between retries	`100 * InitialInterval`
`MaximumAttempts`	Maximum number of retry attempts before giving up	`0` (unlimited)

- Temporal's default behavior for Activity retries
  - Automatically retries with short delays until success or cancellation
  - No action needed for intermittent failures
  - Code resumes as if the failure never occurred

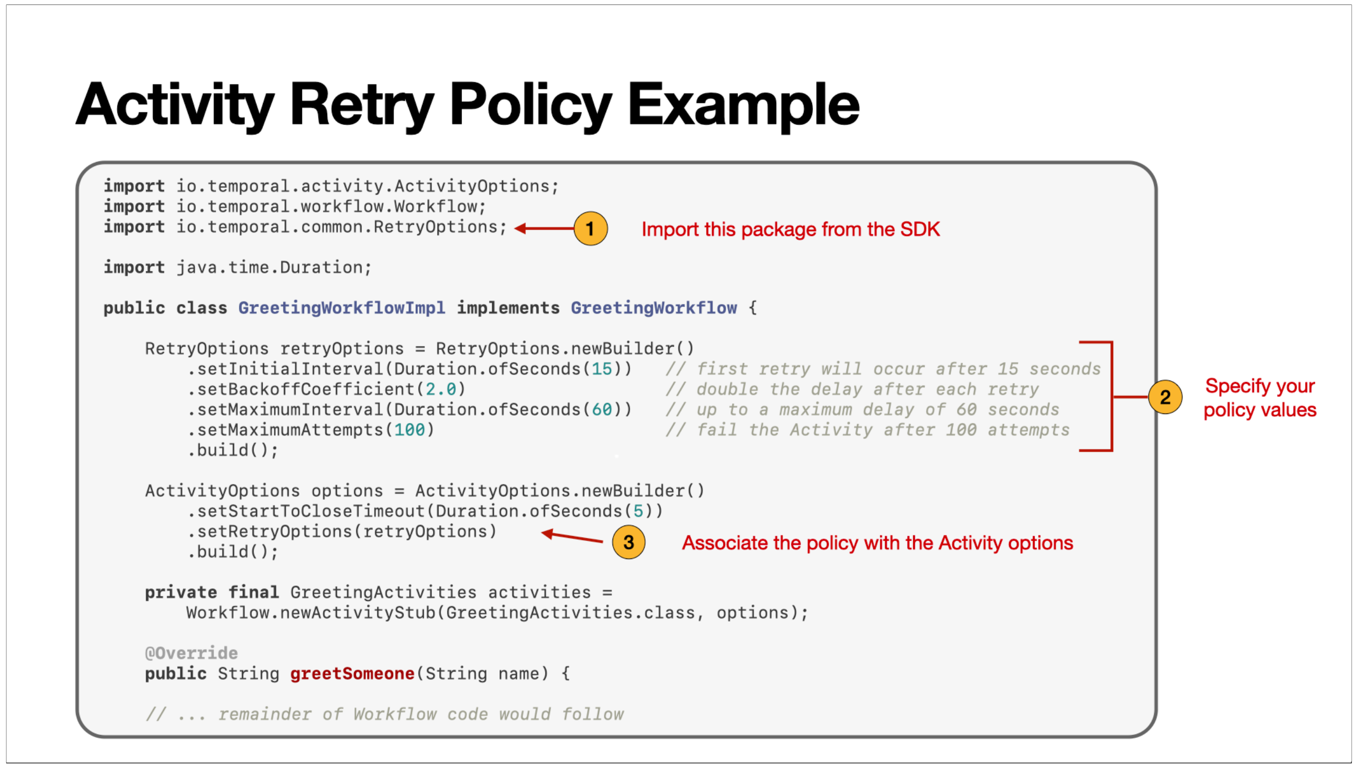
- Customizing retry behavior with a custom Retry Policy
  - Four properties determine timing and retries:
    - "InitialInterval"
      - Default: 1 second
      - Defines delay for the first retry after initial failure
    - "BackoffCoefficient"
      - Default: Multiplier for subsequent retry intervals
      - Defaults result in increasing intervals (1s, 2s, 4s, 8s, ...)
    - "MaximumInterval"
      - Default: 100 times the initial interval
      - Limits delay, preventing it from exceeding 100 seconds
    - "MaximumAttempts"
      - Specifies maximum retry count
      - Marks Activity as failed if exceeded, causing the workflow to fail

---
Temporal's default behavior is to automatically retry an Activity, with a short delay between each attempt, until it either succeeds or is canceled. That means that intermittent failures *require no action* on our part---when a subsequent request succeeds, our code will resume as if the failure never occurred. However, that behavior may not always be desirable, so Temporal allows you to customize it through a custom Retry Policy.

There are four properties that determine the timing and number of retries:

The "InitialInterval" property defines how long after the initial failure the first retry will occur. By default, that's one second.
The "BackoffCoefficient" is a multiplier applied to the "InitialInterval" value that's used to calculate the delay between each subsequent attempt. Assuming you're using the defaults for both properties, that means there will be a retry after 1 second, another after 2 seconds, then 4 seconds, 8 seconds and so on.
The "MaximumInterval" puts a limit on that delay, and by default it's 100 times the initial interval, which means that the delays would keep doubling as described, but would never exceed 100 seconds.
Finally, the "MaximumAttempts" specified the maximum count of retries allowed before marking the Activity as failed, which in turn means the workflow trying to execute it will fail.

--



Activity Retry Policy Example

```java
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;
import io.temporal.common.RetryOptions;

import java.time.Duration;

public class GreetingWorkflowImpl implements GreetingWorkflow {

    RetryOptions retryOptions = RetryOptions.newBuilder()
        .setInitialInterval(Duration.ofSeconds(15))   // first retry will occur after 15 seconds
        .setBackoffCoefficient(2.0)                   // double the delay after each retry
        .setMaximumInterval(Duration.ofSeconds(60))   // up to a maximum delay of 60 seconds
        .setMaximumAttempts(100)                      // fail the Activity after 100 attempts
        .build();

    ActivityOptions options = ActivityOptions.newBuilder()
        .setStartToCloseTimeout(Duration.ofSeconds(5))
        .setRetryOptions(retryOptions)
        .build();

    private final GreetingActivities activities =
        Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {

    // ... remainder of Workflow code would follow
```

1 Import this package from the SDK

2 Specify your policy values

3 Associate the policy with the Activity options

So here's what a custom Retry Policy looks like in practice. There are just three steps to defining one.

First, you import the "RetryOptions" class
Second, you override one or more of those values, such as the InitialInterval or BackoffCoefficient, that I just described.
Finally, you associate your policy with the ActivityOptions used when executing your Activity.

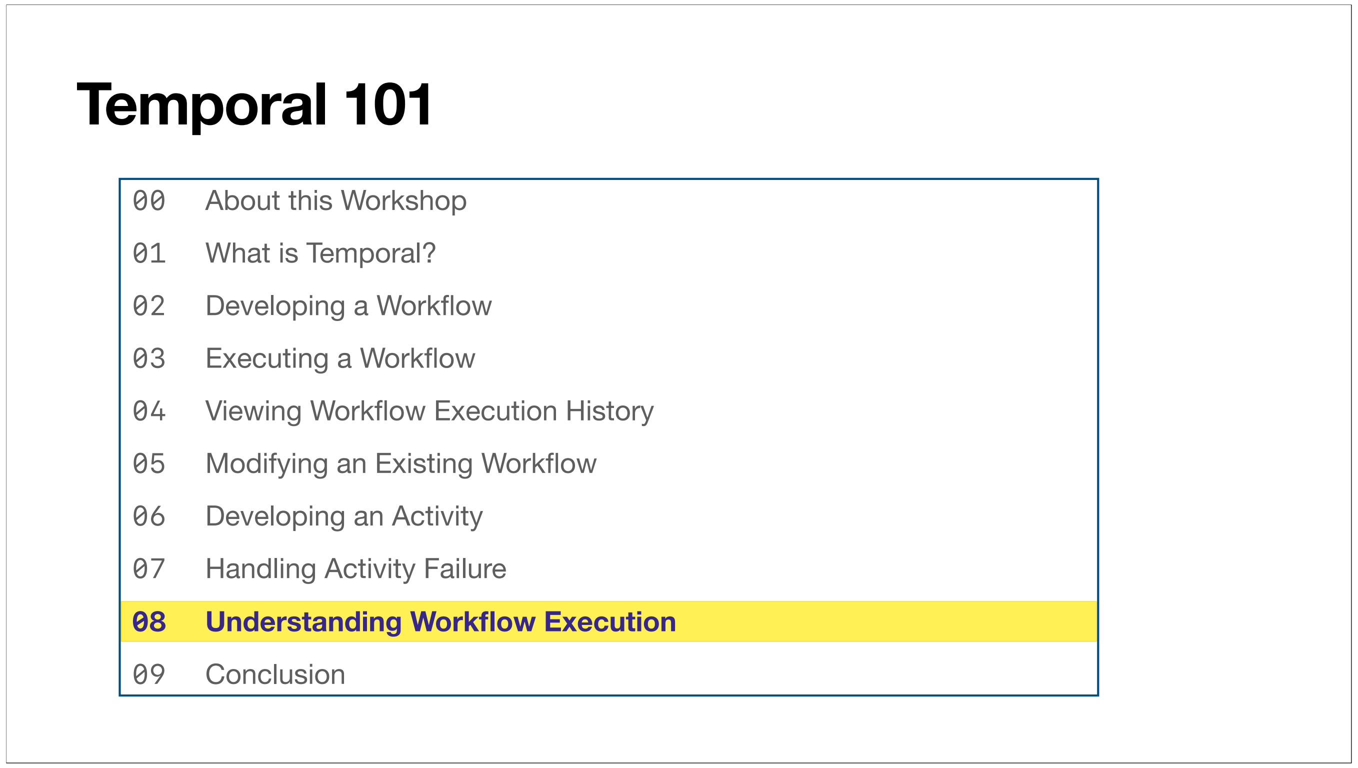
--

Exercise #3: Farewell Workflow

- **During this exercise, you will**
  - Write an Activity method
  - Register the Activity method
  - Modify the Workflow to execute your new Activity
  - Run the Workflow
- **Refer to the README.md file in the exercise environment for details**
  - The code is below the **`exercises/farewell-workflow`** directory
    - Make your changes to the code in the **`practice`** subdirectory (look for TODO comments)
    - If you need a hint or want to verify your changes, look at the complete version in the **`solution`** subdirectory

Exercise #3: Farewell Workflow SOLUTION

10 minute break



Temporal 101

--

AUDIENCE PARTICIPATION
Before we continue, can anyone tell me what the limit is on the number of attempts for an Activity, given the default retry policy?

Answer: Unlimited

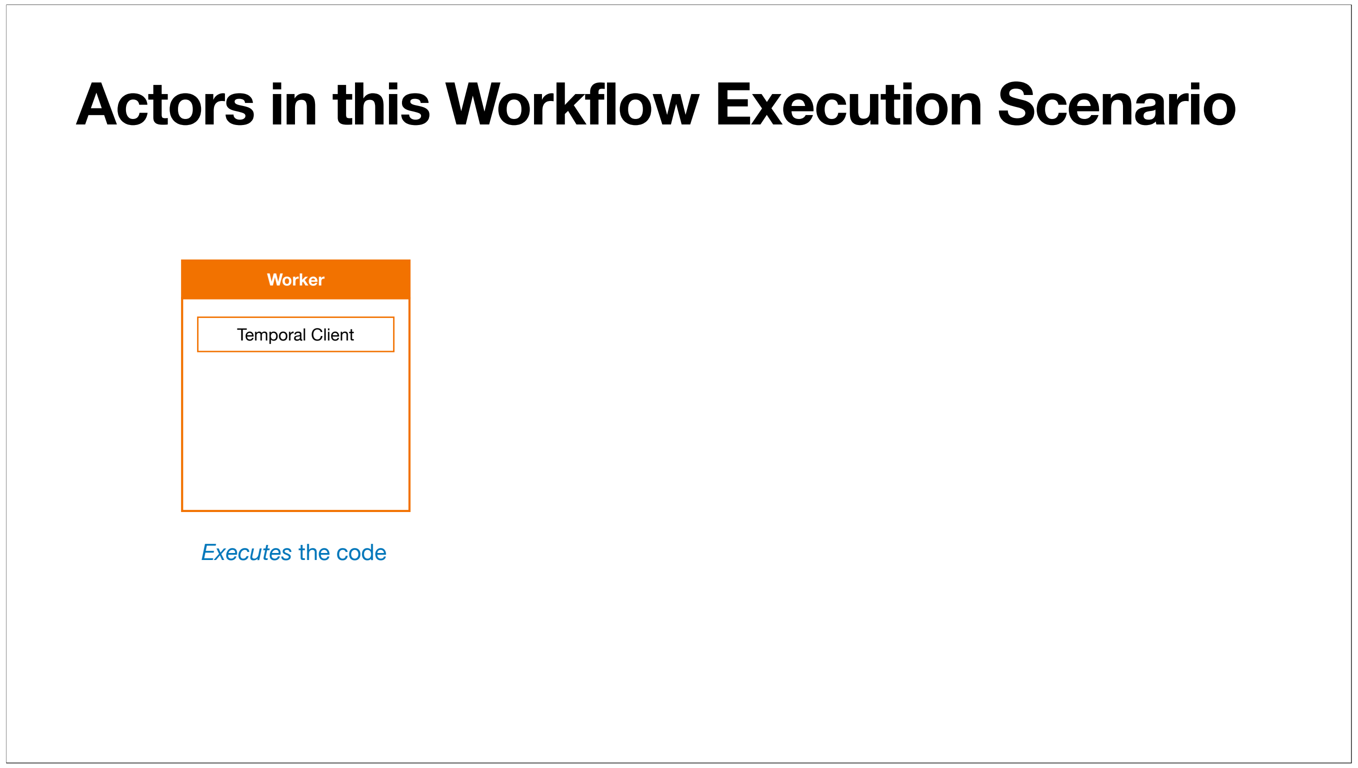
Actors in this Workflow Execution Scenario

During the previous exercise, you executed a Workflow that included two Activities, both of which made a call to a microservice that provided a customized message in Spanish. That exercise demonstrates many of the key concepts you've learned during this course. Although you now have first-hand experience with developing and running applications on the Temporal Platform, you'll gain a deeper understanding of how Temporal works by looking at what happens during Workflow Execution.

I'll begin by identifying the actors in this scenario, which will help to reiterate some important concepts.

First, we have the Worker, which executes the Workflow and Activity code, and uses a client to communicate with the cluster. Next, we have the Temporal Cluster, which orchestrates the execution of that code by coordinating with the Worker, using a shared task queue. Finally, we have a program that starts the Workflow, which I'll refer to as a client application because it requests Workflow Execution as well as the result from the Temporal Cluster, and it uses a client to do this.

--
IG: Although I have shown a single task queue here for the sake of simplicity, the Workflow and Activity Task queues are actually separate.



Actors in this Workflow Execution Scenario

Worker
Temporal Client
Executes the code

During the previous exercise, you executed a Workflow that included two Activities, both of which made a call to a microservice that provided a customized message in Spanish.

- Workflow with two Activities and microservice calls
  - Demonstrates key concepts in the course
  - Provides insight into Temporal Platform operation during Workflow Execution

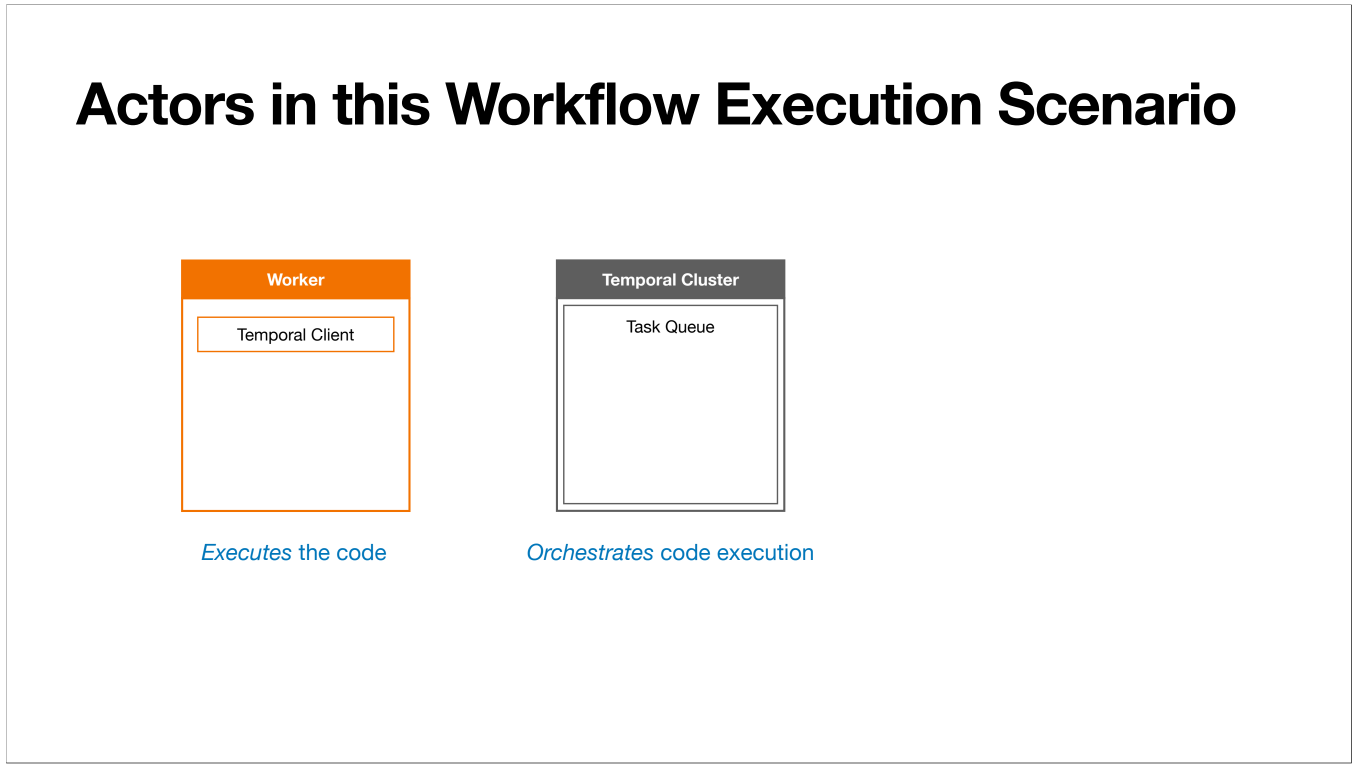
- Actors in the scenario:
  - Worker
    - Executes Workflow and Activity code
    - Uses a client to communicate with the cluster

- Note: In the diagram, a single task queue is shown for simplicity, but Workflow and Activity Task queues are separate.

That exercise demonstrates many of the key concepts you've learned during this course. Although you now have first-hand experience with developing and running applications on the Temporal Platform, you'll gain a deeper understanding of how Temporal works by looking at what happens during Workflow Execution.

I'll begin by identifying the actors in this scenario, which will help to reiterate some important concepts.

First, we have the Worker, which executes the Workflow and Activity code, and uses a client to communicate with the cluster. Next, we have the Temporal Cluster, which orchestrates the execution of that code by coordinating with the Worker, using a shared task queue. Finally, we have a program that starts the Workflow, which I'll refer to as a client application because it requests Workflow Execution as well as the result from the Temporal Cluster, and it uses a client to do this.

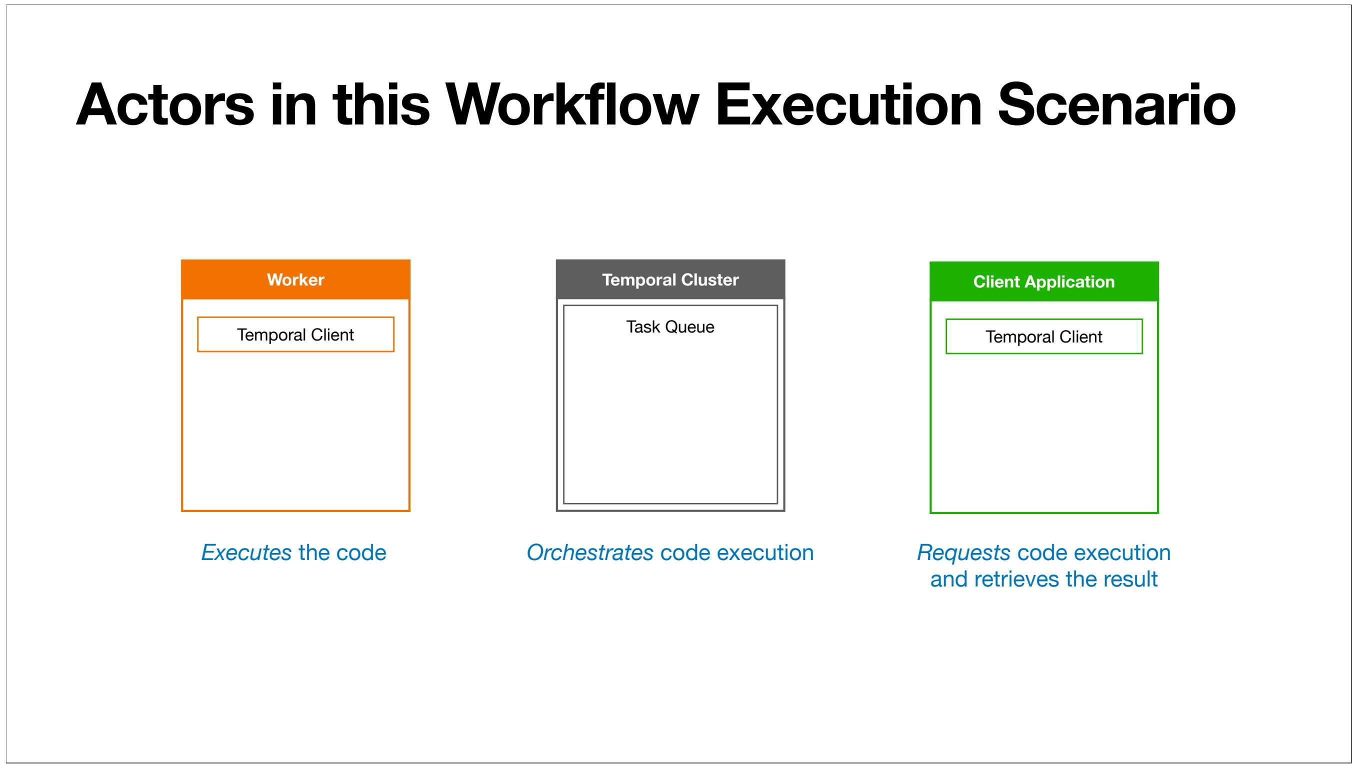

Actors in this Workflow Execution Scenario

- Temporal Cluster
  - Orchestrates code execution
  - Coordinates with the Worker
  - Utilizes a shared task queue

During the previous exercise, you executed a Workflow that included two Activities, both of which made a call to a microservice that provided a customized message in Spanish. That exercise demonstrates many of the key concepts you've learned during this course. Although you now have first-hand experience with developing and running applications on the Temporal Platform, you'll gain a deeper understanding of how Temporal works by looking at what happens during Workflow Execution.

I'll begin by identifying the actors in this scenario, which will help to reiterate some important concepts.

First, we have the Worker, which executes the Workflow and Activity code, and uses a client to communicate with the cluster. Next, we have the Temporal Cluster, which orchestrates the execution of that code by coordinating with the Worker, using a shared task queue. Finally, we have a program that starts the Workflow, which I'll refer to as a client application because it requests Workflow Execution as well as the result from the Temporal Cluster, and it uses a client to do this.



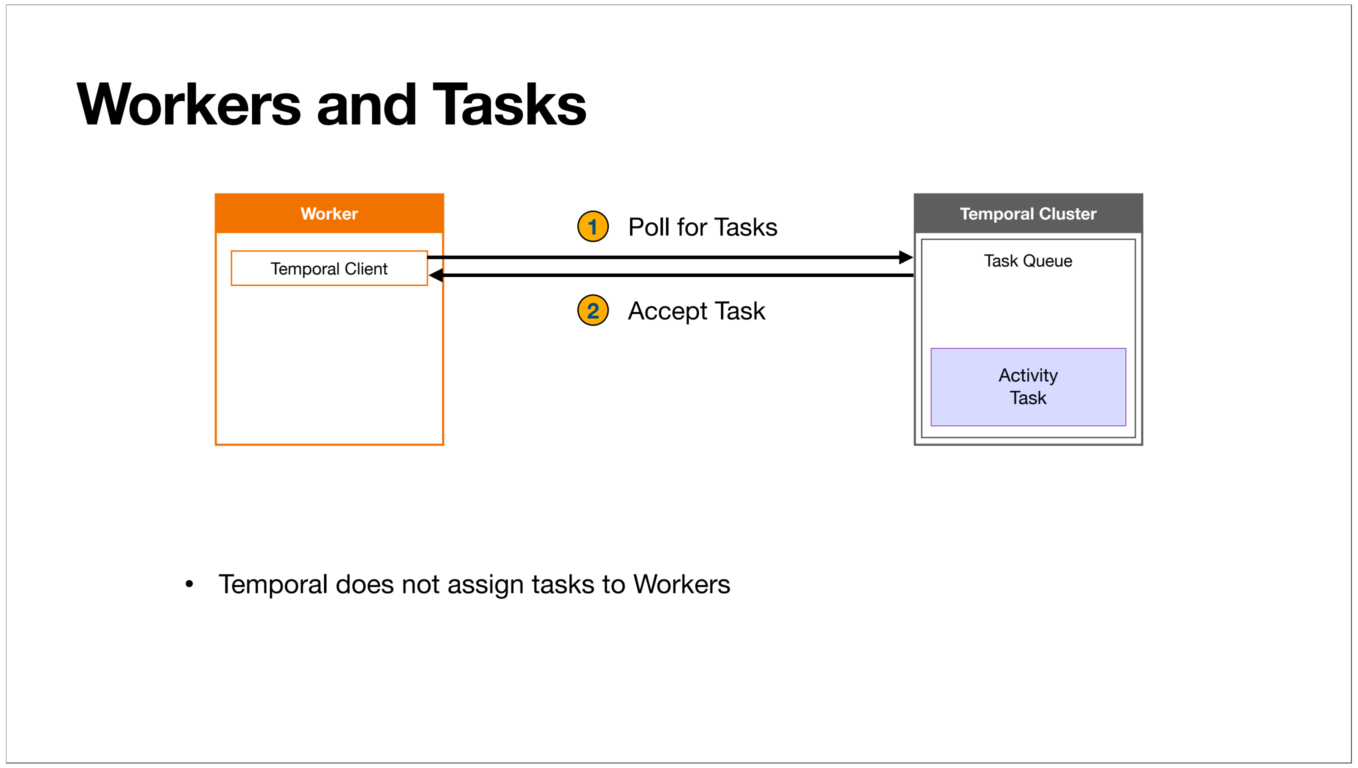
Actors in this Workflow Execution Scenario

- Client Application
  - Initiates Workflow
  - Requests Workflow Execution
  - Receives results from the Temporal Cluster
  - Uses a client for communication

During the previous exercise, you executed a Workflow that included two Activities, both of which made a call to a microservice that provided a customized message in Spanish. That exercise demonstrates many of the key concepts you've learned during this course. Although you now have first-hand experience with developing and running applications on the Temporal Platform, you'll gain a deeper understanding of how Temporal works by looking at what happens during Workflow Execution.

I'll begin by identifying the actors in this scenario, which will help to reiterate some important concepts.

First, we have the Worker, which executes the Workflow and Activity code, and uses a client to communicate with the cluster. Next, we have the Temporal Cluster, which orchestrates the execution of that code by coordinating with the Worker, using a shared task queue. Finally, we have a program that starts the Workflow, which I'll refer to as a client application because it requests Workflow Execution as well as the result from the Temporal Cluster, and it uses a client to do this.



Workers and Tasks

- Temporal does not assign tasks to Workers

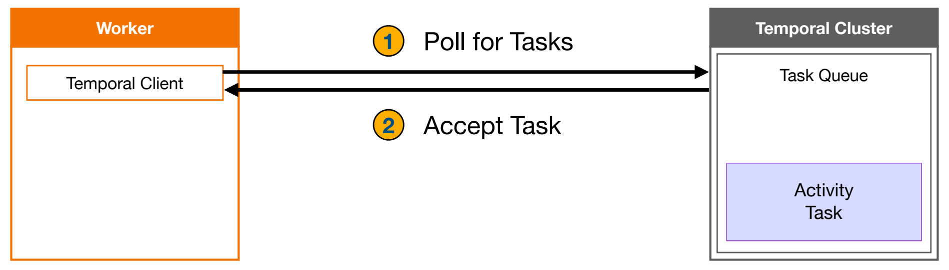
- Work assignment in Temporal is indirect
  - Temporal Cluster does not assign tasks to Workers
  - Temporal Cluster doesn't maintain a list of Workers

- Note: Typically, production Temporal applications have multiple Workers.
- Note: Example uses a single Worker for simplicity.
- Note: While a single task queue is shown for simplicity, Workflow and Activity Task queues are separate.

The assignment of work is indirect. The Temporal Cluster does not assign tasks to a Worker (in fact, the Temporal cluster does not maintain a list of Workers). Instead, the Workers continually poll the Temporal Cluster's task queue and accept tasks when they have spare capacity to process them. There are several benefits to this approach, but one of them is that tasks will just sit in the queue if there aren't enough Workers, which means that you can increase throughput and scalability by adding more Workers.

As you learned earlier, Temporal applications in production will typically have multiple Workers, but I'm using a single Worker in this example for simplicity.

Workers and Tasks



- Temporal does not assign tasks to Workers
- Workers continuously poll, accepting tasks when they have spare capacity

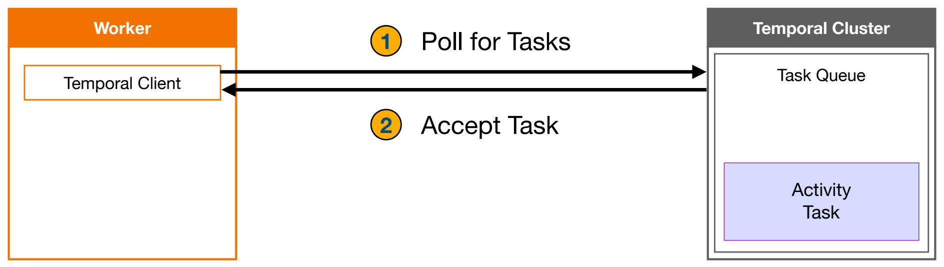
- Workers poll the Temporal Cluster's task queue
- Workers accept tasks when they have capacity
- Tasks remain in the queue if there aren't enough Workers

- Note: Typically, production Temporal applications have multiple Workers.
- Note: Example uses a single Worker for simplicity.
- Note: While a single task queue is shown for simplicity, Workflow and Activity Task queues are separate.

The assignment of work is indirect. The Temporal Cluster does not assign tasks to a Worker (in fact, the Temporal cluster does not maintain a list of Workers). Instead, the Workers continually poll the Temporal Cluster's task queue and accept tasks when they have spare capacity to process them. There are several benefits to this approach, but one of them is that tasks will just sit in the queue if there aren't enough Workers, which means that you can increase throughput and scalability by adding more Workers.

As you learned earlier, Temporal applications in production will typically have multiple Workers, but I'm using a single Worker in this example for simplicity.

Workers and Tasks



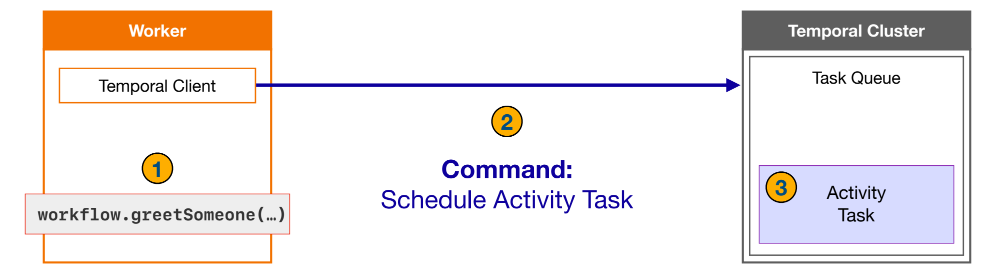
- Temporal does not assign tasks to Workers
- Workers continuously poll, accepting tasks when they have spare capacity
- You can increase throughput and scalability by adding Workers

- Scalability by adding more Workers
- Benefits of this approach include increased throughput
- Note: Typically, production Temporal applications have multiple Workers.
- Note: Example uses a single Worker for simplicity.
- Note: While a single task queue is shown for simplicity, Workflow and Activity Task queues are separate.

The assignment of work is indirect. The Temporal Cluster does not assign tasks to a Worker (in fact, the Temporal cluster does not maintain a list of Workers). Instead, the Workers continually poll the Temporal Cluster's task queue and accept tasks when they have spare capacity to process them. There are several benefits to this approach, but one of them is that tasks will just sit in the queue if there aren't enough Workers, which means that you can increase throughput and scalability by adding more Workers.

As you learned earlier, Temporal applications in production will typically have multiple Workers, but I'm using a single Worker in this example for simplicity.

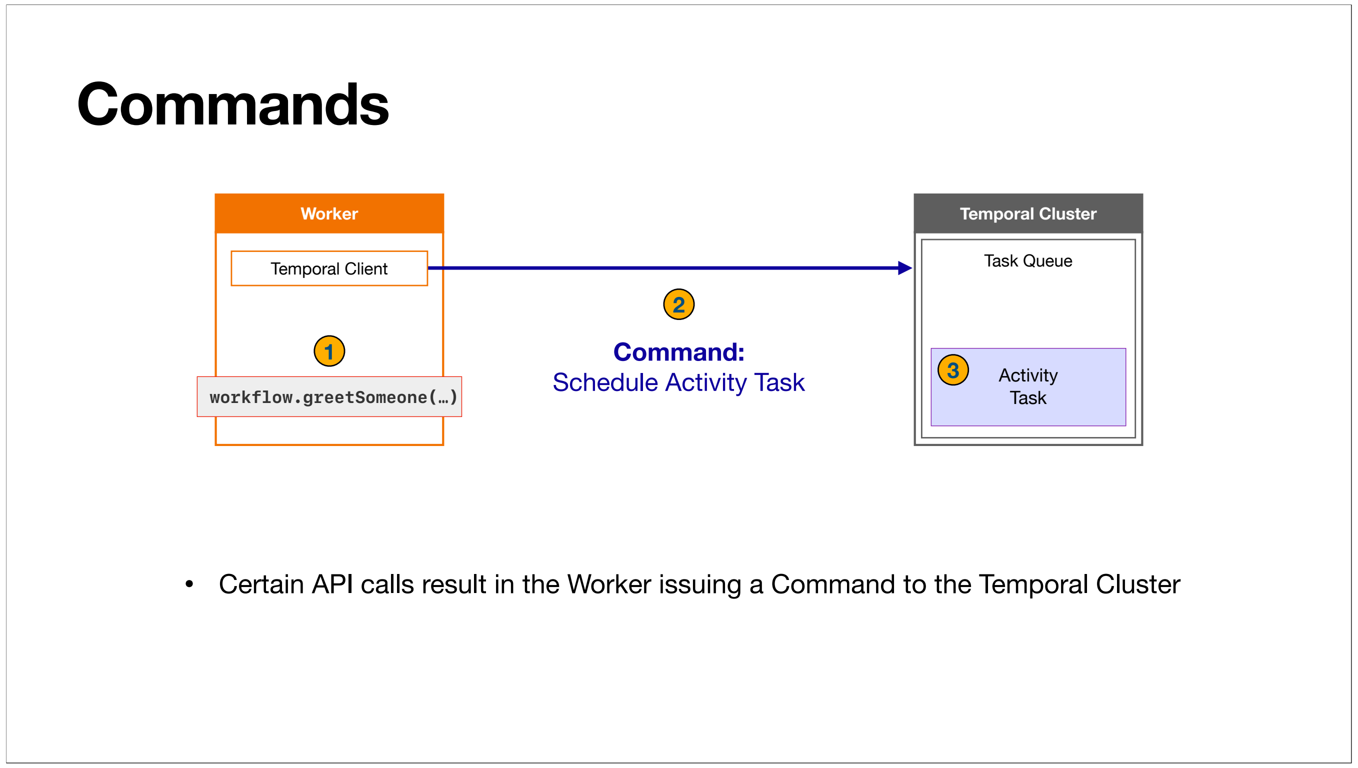
Commands



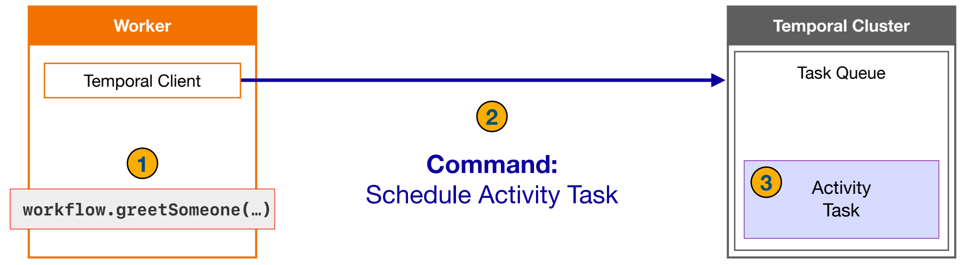
Another thing that will help you understand Temporal is the role of Commands.

When the Worker encounters certain API calls during Workflow Execution, such as a call to the Workflow's ExecuteActivity function, it sends a Command to the Temporal Cluster. The cluster acts on these Commands, for example, by creating an Activity Task, but also stores them in case of failure. For example, if the Worker crashes, the Temporal cluster uses this information to recreate the state of the Workflow to what it was immediately before the crash and then resumes progress from that point. This allows you, as a developer, to code as if this type of failure wasn't even a possibility.

--



Commands



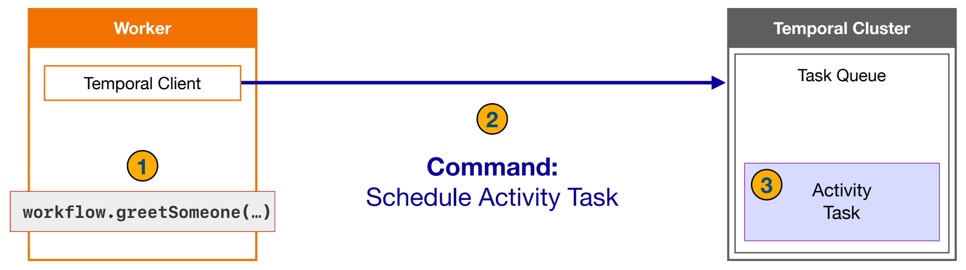
- Certain API calls result in the Worker issuing a Command to the Temporal Cluster

- Role of Commands in Temporal
  - During Workflow Execution, Worker encounters certain API calls
  - Example: Worker calls Workflow's ExecuteActivity function
  - Worker sends a Command to the Temporal Cluster

Another thing that will help you understand Temporal is the role of Commands. When the Worker encounters certain API calls during Workflow Execution, such as a call to the Workflow's ExecuteActivity function, it sends a Command to the Temporal Cluster. The cluster acts on these Commands, for example, by creating an Activity Task, but also stores them in case of failure. For example, if the Worker crashes, the Temporal cluster uses this information to recreate the state of the Workflow to what it was immediately before the crash and then resumes progress from that point. This allows you, as a developer, to code as if this type of failure wasn't even a possibility.

--

Commands



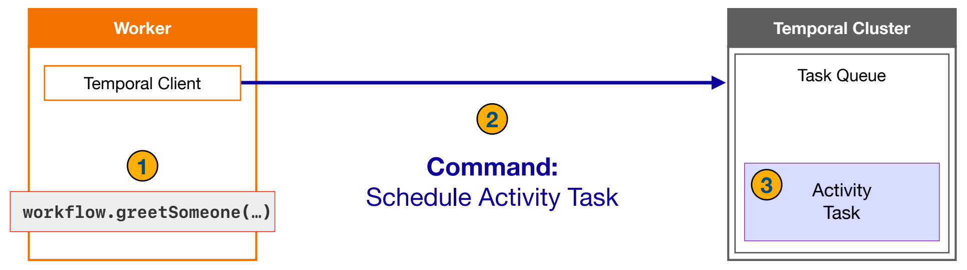
- Certain API calls result in the Worker issuing a Command to the Temporal Cluster
- The Cluster acts on these commands, but also stores them

- Temporal Cluster acts on Commands
  - Creates an Activity Task and performs related actions
  - Stores Commands for recovery in case of failure

Another thing that will help you understand Temporal is the role of Commands. When the Worker encounters certain API calls during Workflow Execution, such as a call to the Workflow's ExecuteActivity function, it sends a Command to the Temporal Cluster. The cluster acts on these Commands, for example, by creating an Activity Task, but also stores them in case of failure. For example, if the Worker crashes, the Temporal cluster uses this information to recreate the state of the Workflow to what it was immediately before the crash and then resumes progress from that point. This allows you, as a developer, to code as if this type of failure wasn't even a possibility.

--

Commands



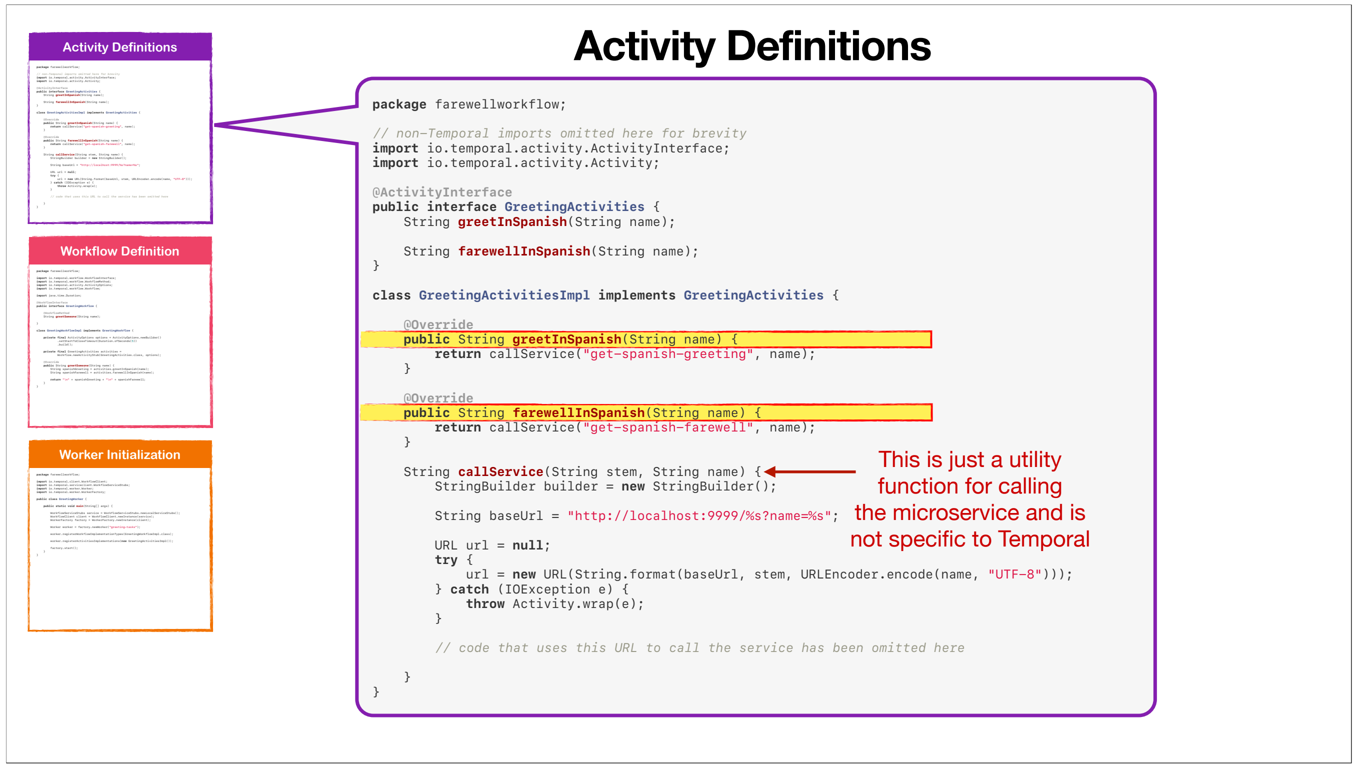
- Certain API calls result in the Worker issuing a Command to the Temporal Cluster
- The Cluster acts on these commands, but also stores them
- This allows the Worker to recreate the state of a Workflow Execution following a crash

- In case of Worker crash, Temporal Cluster uses Commands to recreate Workflow state
  - Resumes progress from the point before the crash
- Allows developers to code as if such failures were not a concern

---

Another thing that will help you understand Temporal is the role of Commands. When the Worker encounters certain API calls during Workflow Execution, such as a call to the Workflow's ExecuteActivity function, it sends a Command to the Temporal Cluster. The cluster acts on these Commands, for example, by creating an Activity Task, but also stores them in case of failure. For example, if the Worker crashes, the Temporal cluster uses this information to recreate the state of the Workflow to what it was immediately before the crash and then resumes progress from that point. This allows you, as a developer, to code as if this type of failure wasn't even a possibility.

--



Activity Definitions

Workflow Definition

Worker Initialization

Activity Definitions

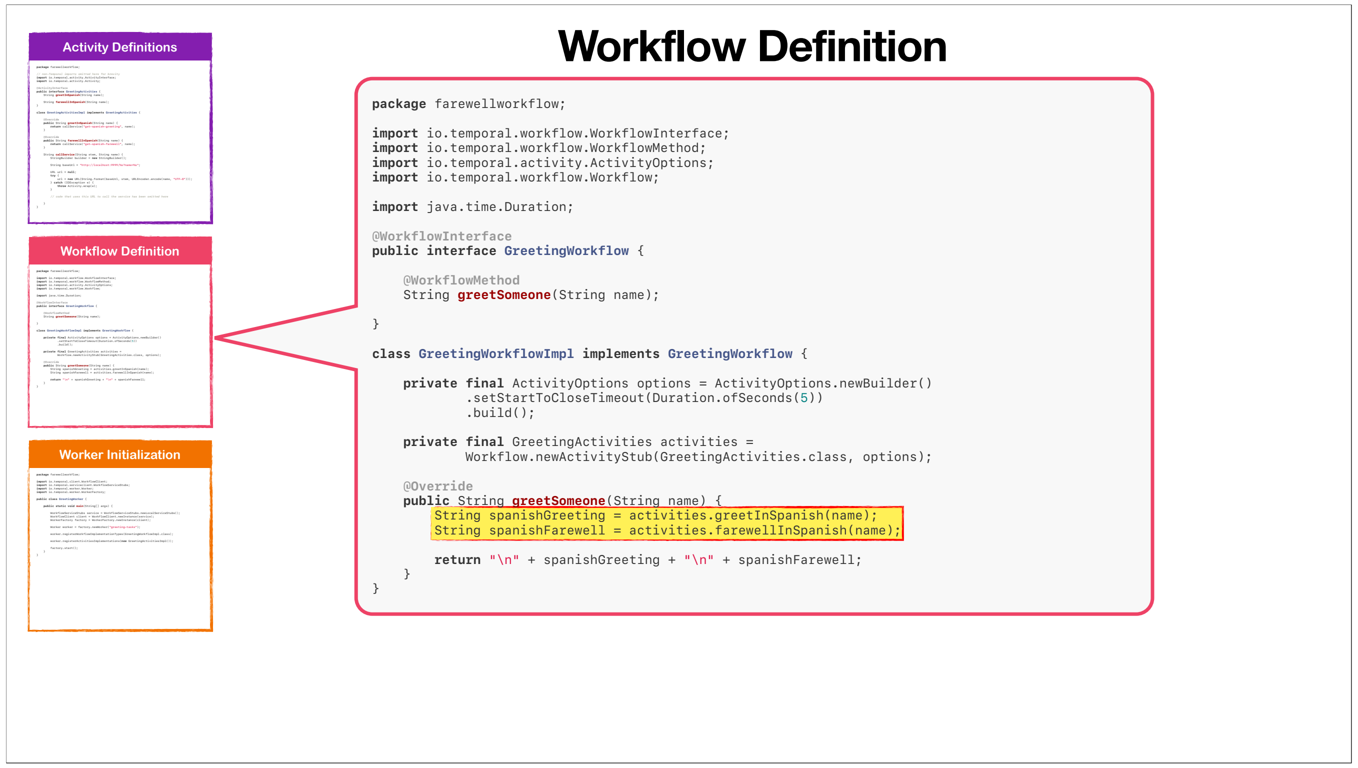
```java
package farewellworkflow;

// non-Temporal imports omitted here for brevity
import io.temporal.activity.ActivityInterface;
import io.temporal.activity.Activity;

@ActivityInterface
public interface GreetingActivities {
    String greetInSpanish(String name);

    String farewellInSpanish(String name);
}

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        try {
            url = new URL(String.format(baseUrl, stem, URLEncoder.encode(name, "UTF-8")));
        } catch (IOException e) {
            throw Activity.wrap(e);
        }

        // code that uses this URL to call the service has been omitted here

    }
}
```

This is just a utility function for calling the microservice and is not specific to Temporal

The application defines two Activities: GreetInSpanish and FarewellInSpanish, plus a utility function that both Activities use to call the translation service.

--

NOTE: For the sake of clarity, the code in this walkthrough omits the logging statements that are included in the version from the exercise.

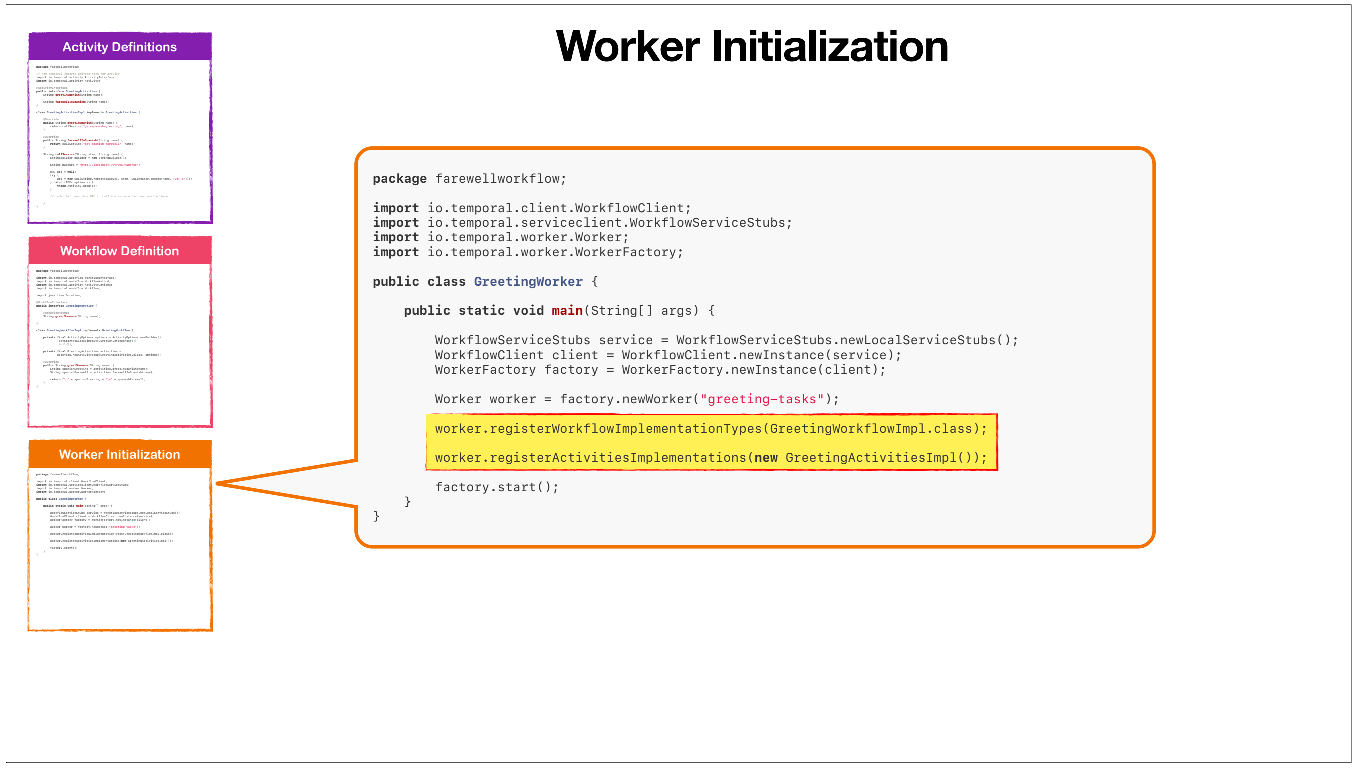


Workflow Definition

```java
package farewellworkflow;

import io.temporal.workflow.WorkflowInterface;
import io.temporal.workflow.WorkflowMethod;
import io.temporal.activity.ActivityOptions;
import io.temporal.workflow.Workflow;

import java.time.Duration;

@WorkflowInterface
public interface GreetingWorkflow {

    @WorkflowMethod
    String greetSomeone(String name);

}

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

The Workflow Definition executes those two Activities and returns a string created from their output.

--



Worker Initialization

Activity Definitions
Workflow Definition
Worker Initialization

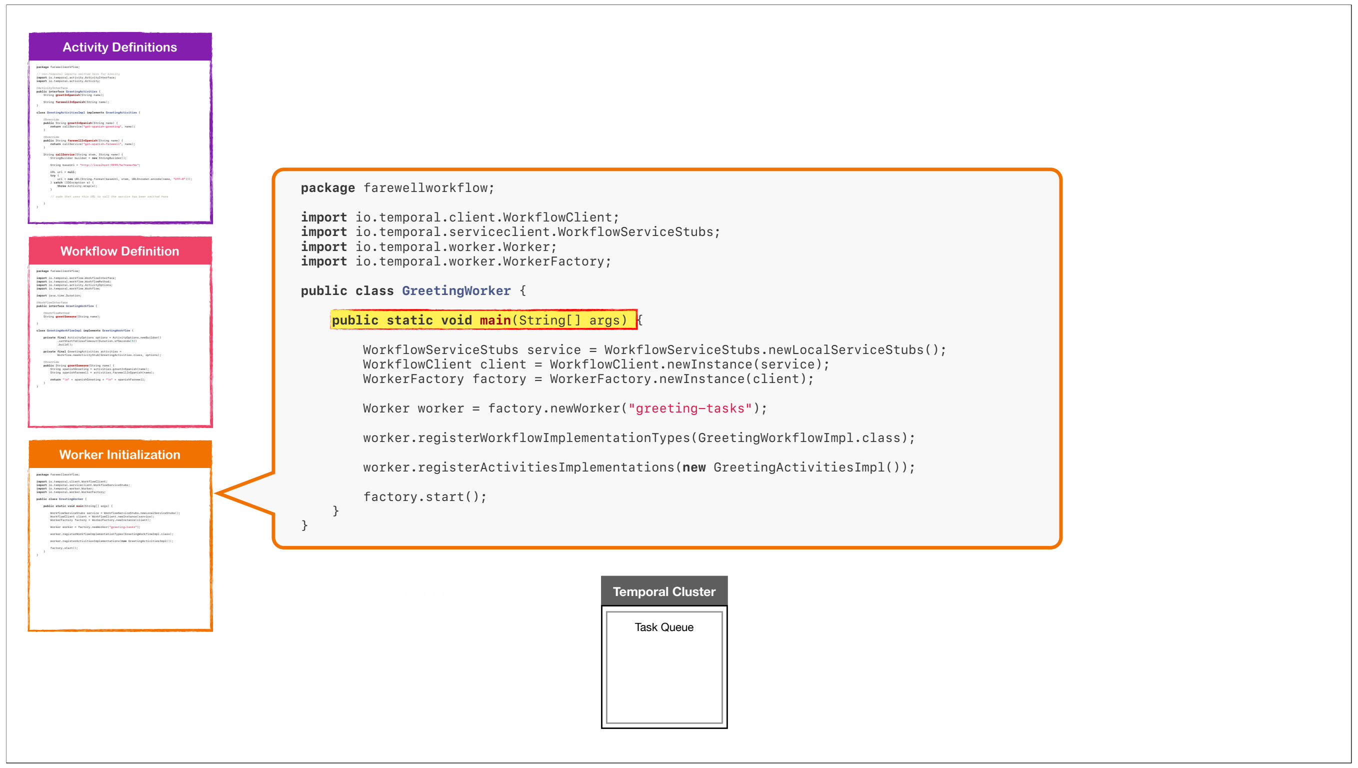
```java
package farewellworkflow;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);

        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

And here's the worker initialization code, which registers the Workflow and Activity Definitions.

I will walk through all of this code in a moment, but first, I want to point out that while I'm showing these as three distinct source files for the sake of clarity, you could organize them however you like, even putting all of the code into a single file if you prefer.

Regardless of how the files are organized, for a production deployment you'd typically compile all of these source files into a single executable that you'll deploy and run on your application servers.

--

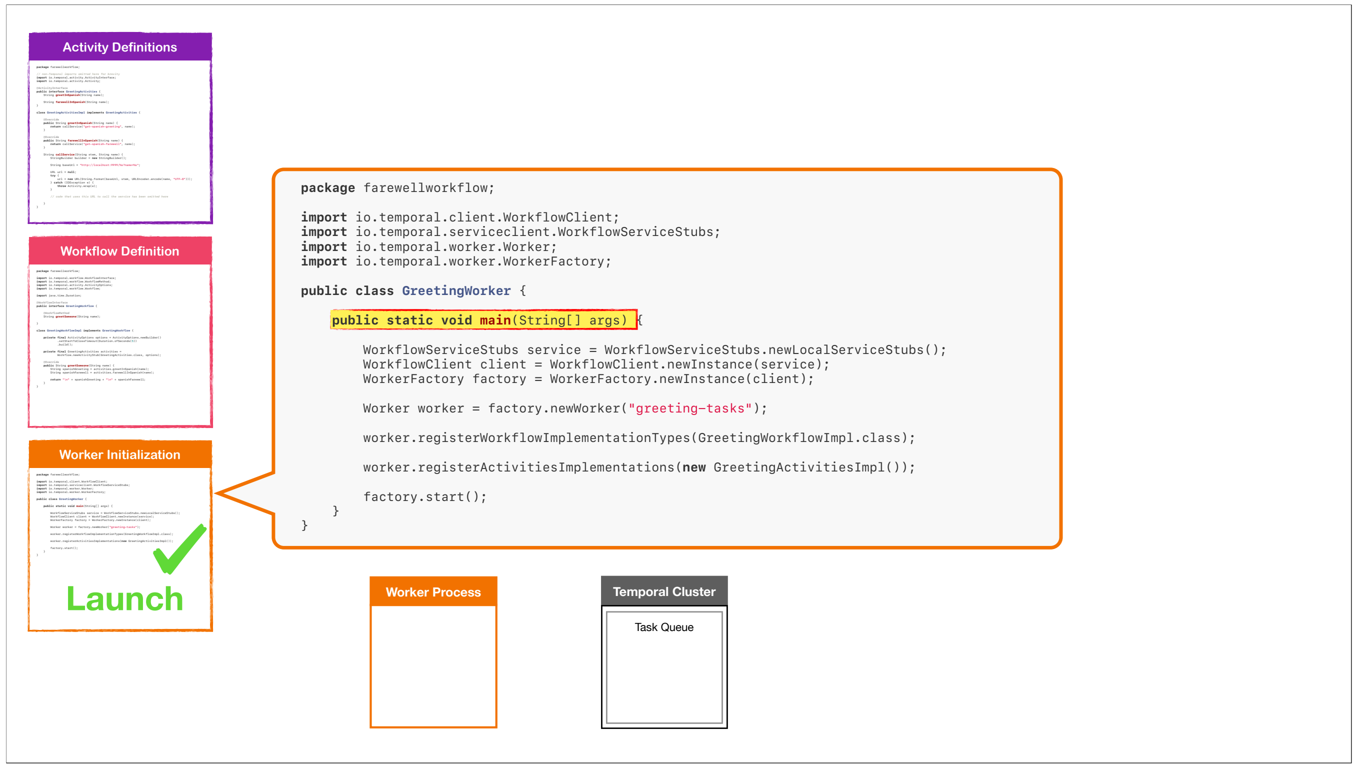


```
package farewellworkflow;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);

        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

As you learned, the Worker executes your Workflow and Activity code, so a Workflow Execution cannot progress unless at least one Worker is running.

Launching the Worker creates a new process. Since this example is written in Go, a program begins by locating the main package and running the main function.

--

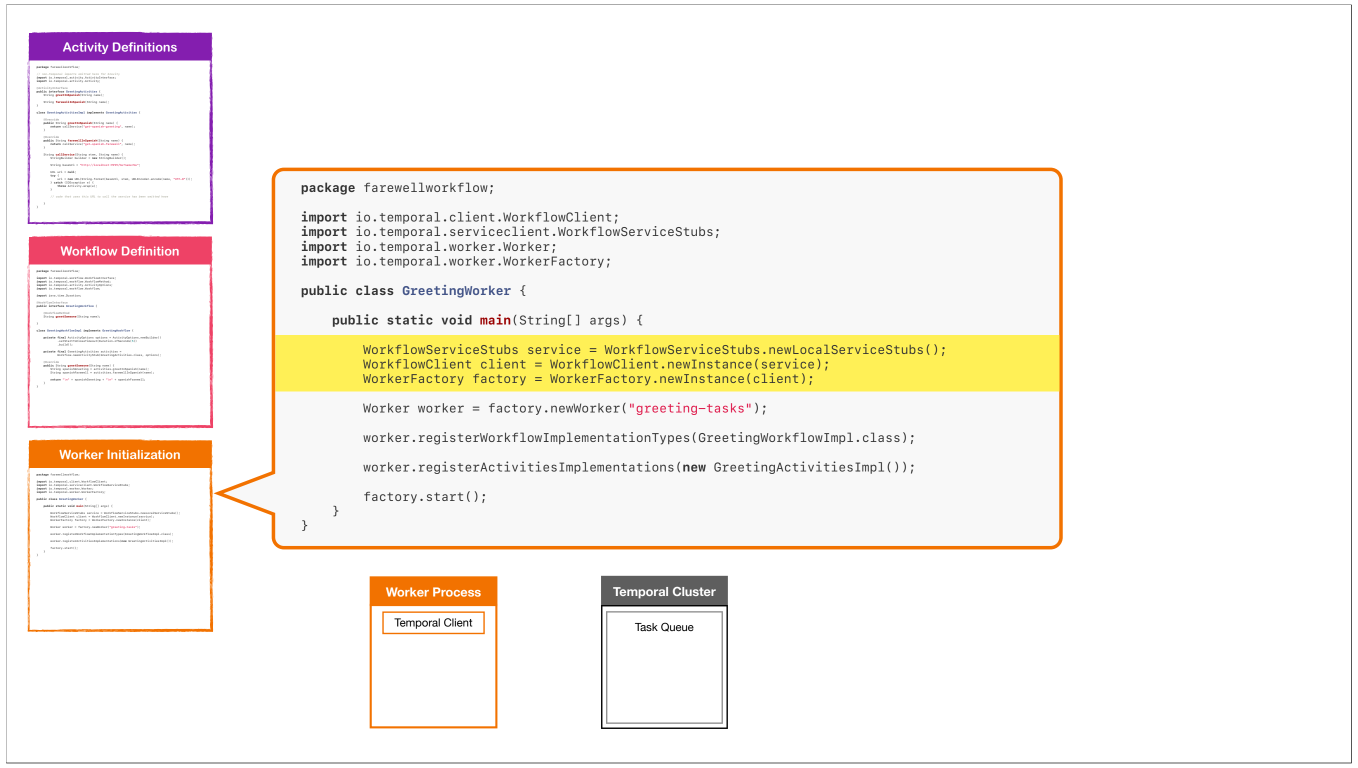


```
package farewellworkflow;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);

        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

As you learned, the Worker executes your Workflow and Activity code, so a Workflow Execution cannot progress unless at least one Worker is running.

Launching the Worker creates a new process. Since this example is written in Go, a program begins by locating the main package and running the main function.

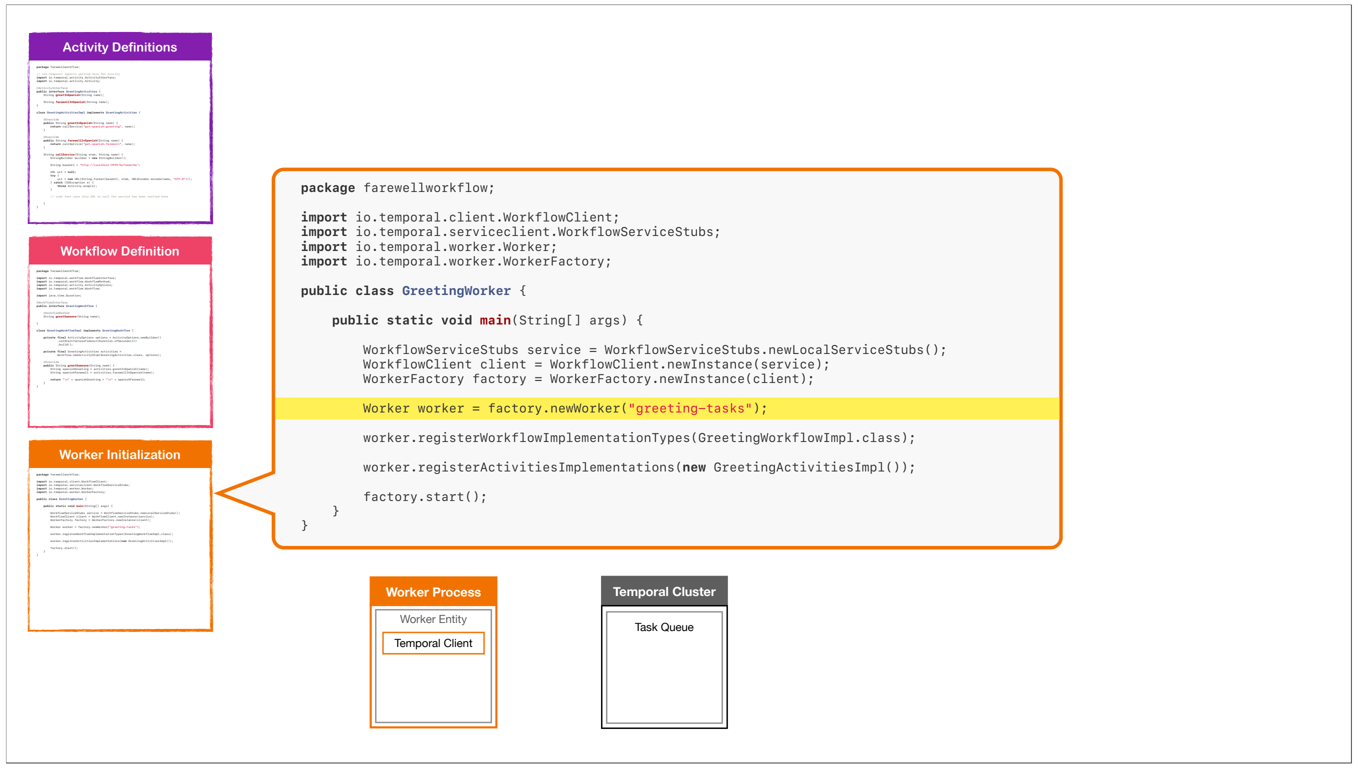
--



```java
package farewellworkflow;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);

        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

This function first creates a Temporal client.

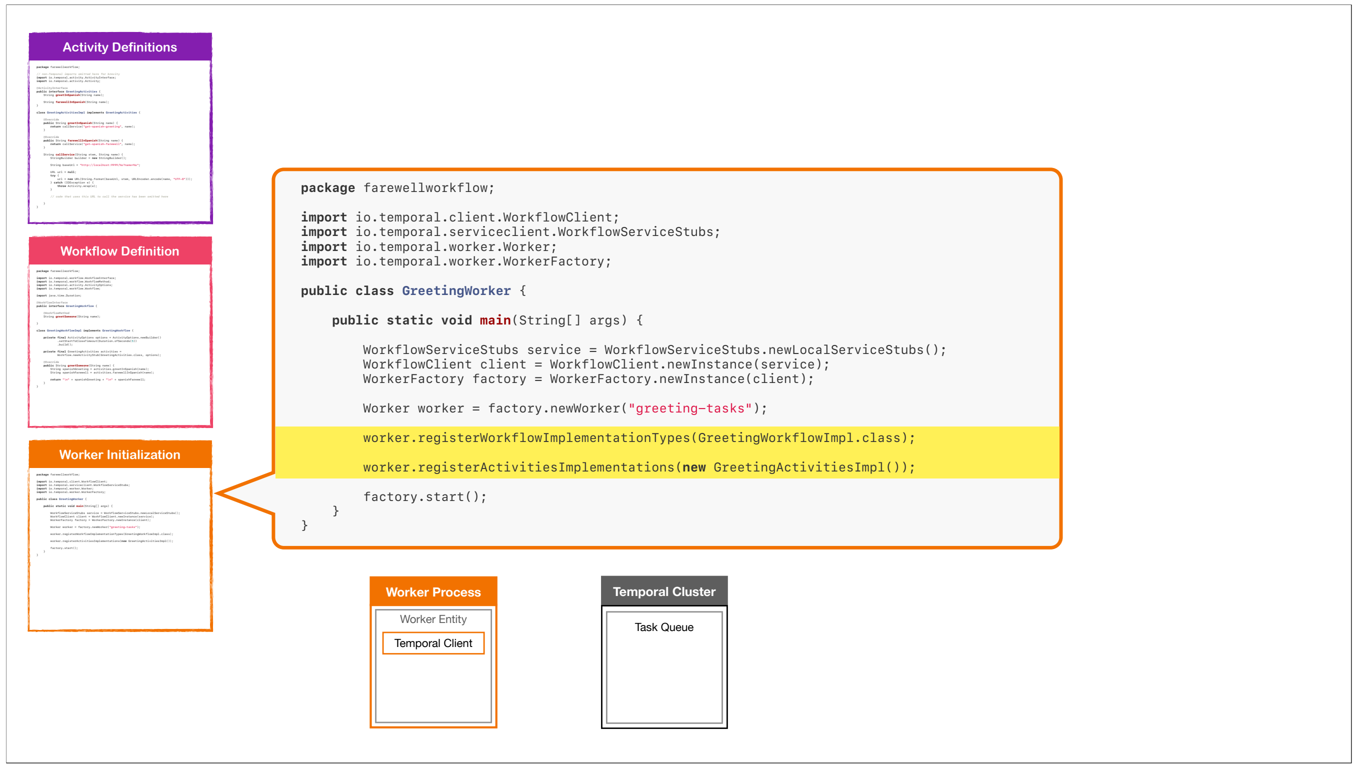
--



```
package farewellworkflow;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);

        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

Next, it creates a new Worker Entity with that client, the name of the Task Queue, and options for configuring its behavior. This example uses the default options.

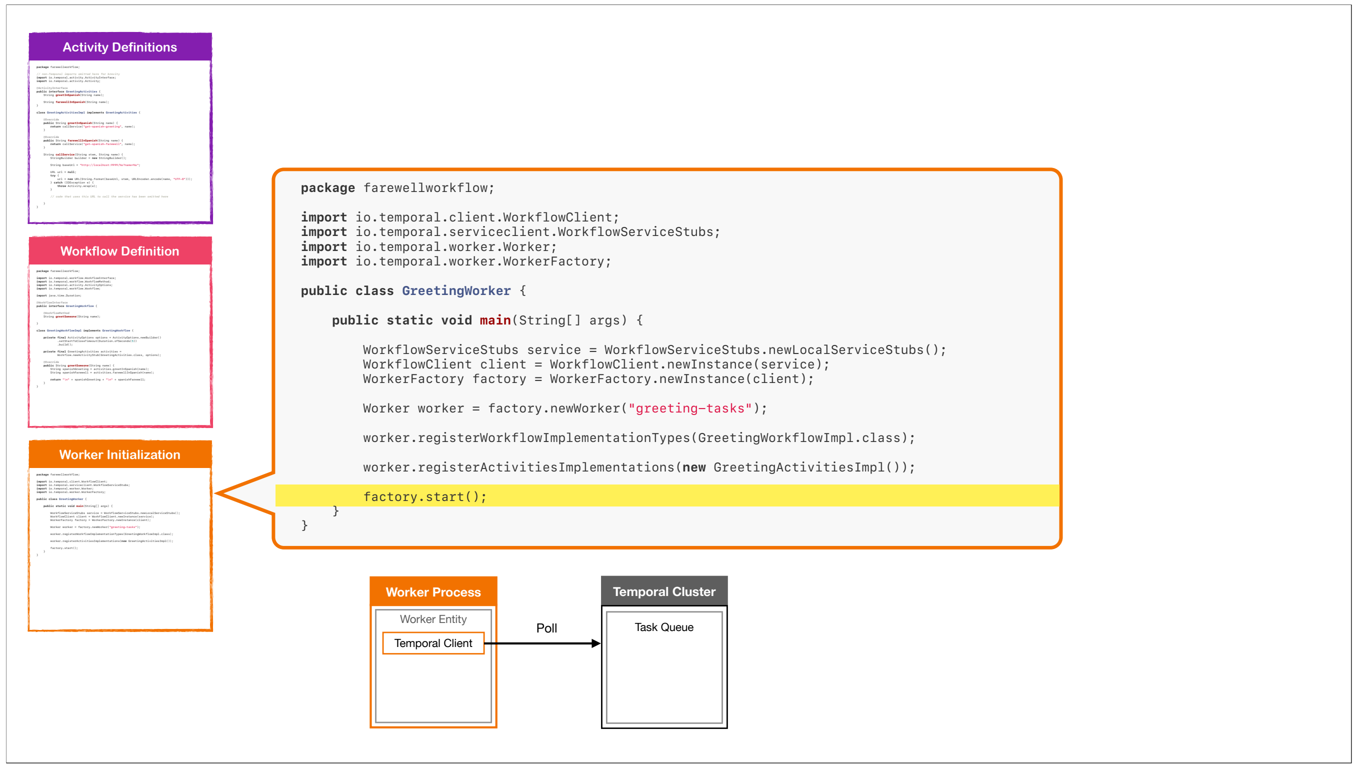
--



```java
package farewellworkflow;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);

        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

A Worker can execute Workflow and Activity Tasks for types that are registered with it. The highlighted lines include references to function names in the Workflow and Activity Definitions.

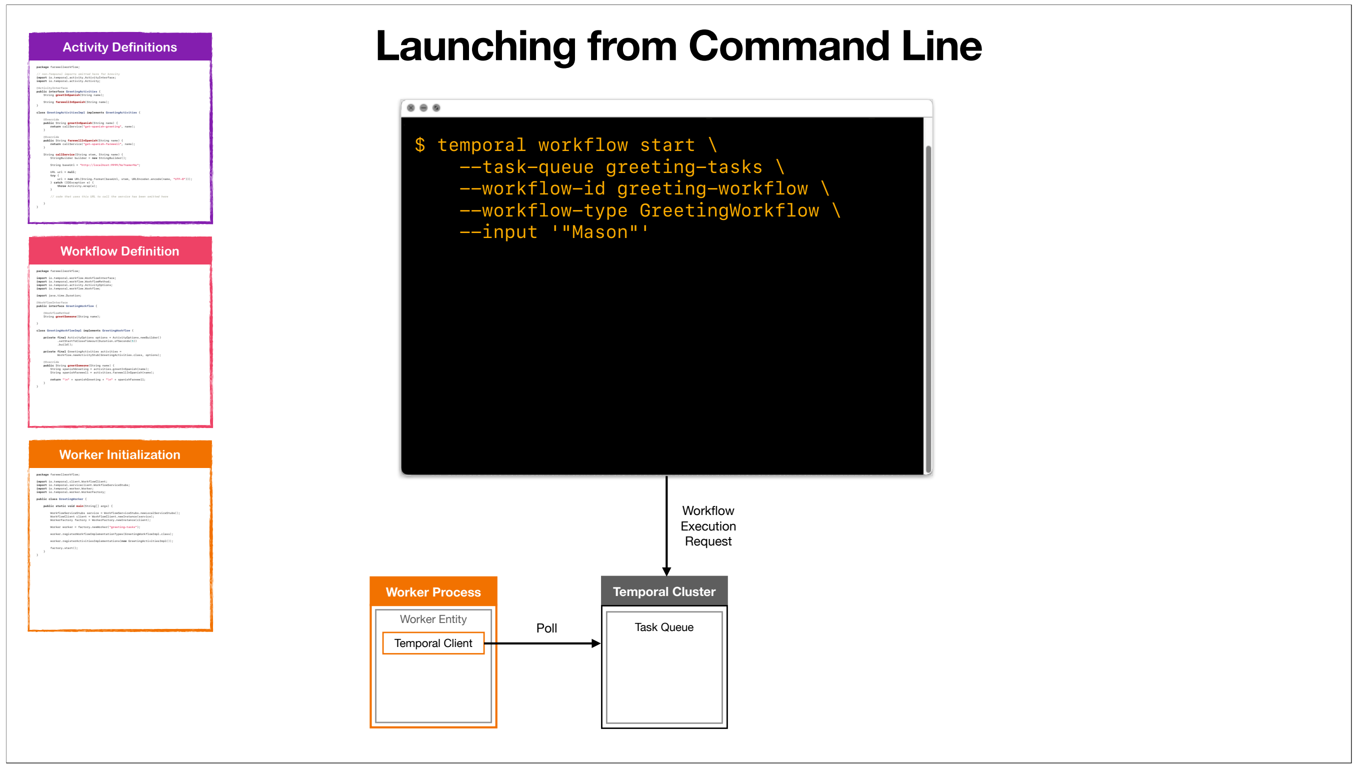
--



```java
package farewellworkflow;

import io.temporal.client.WorkflowClient;
import io.temporal.serviceclient.WorkflowServiceStubs;
import io.temporal.worker.Worker;
import io.temporal.worker.WorkerFactory;

public class GreetingWorker {

    public static void main(String[] args) {

        WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
        WorkflowClient client = WorkflowClient.newInstance(service);
        WorkerFactory factory = WorkerFactory.newInstance(client);

        Worker worker = factory.newWorker("greeting-tasks");

        worker.registerWorkflowImplementationTypes(GreetingWorkflowImpl.class);

        worker.registerActivitiesImplementations(new GreetingActivitiesImpl());

        factory.start();
    }
}
```

Running the Worker Entity opens a long-lasting connection to the Temporal Cluster, which it uses to continuously poll for new tasks. Although the Worker is running, the Workflow is not, so the task queue is empty and the Worker has nothing to do.

--

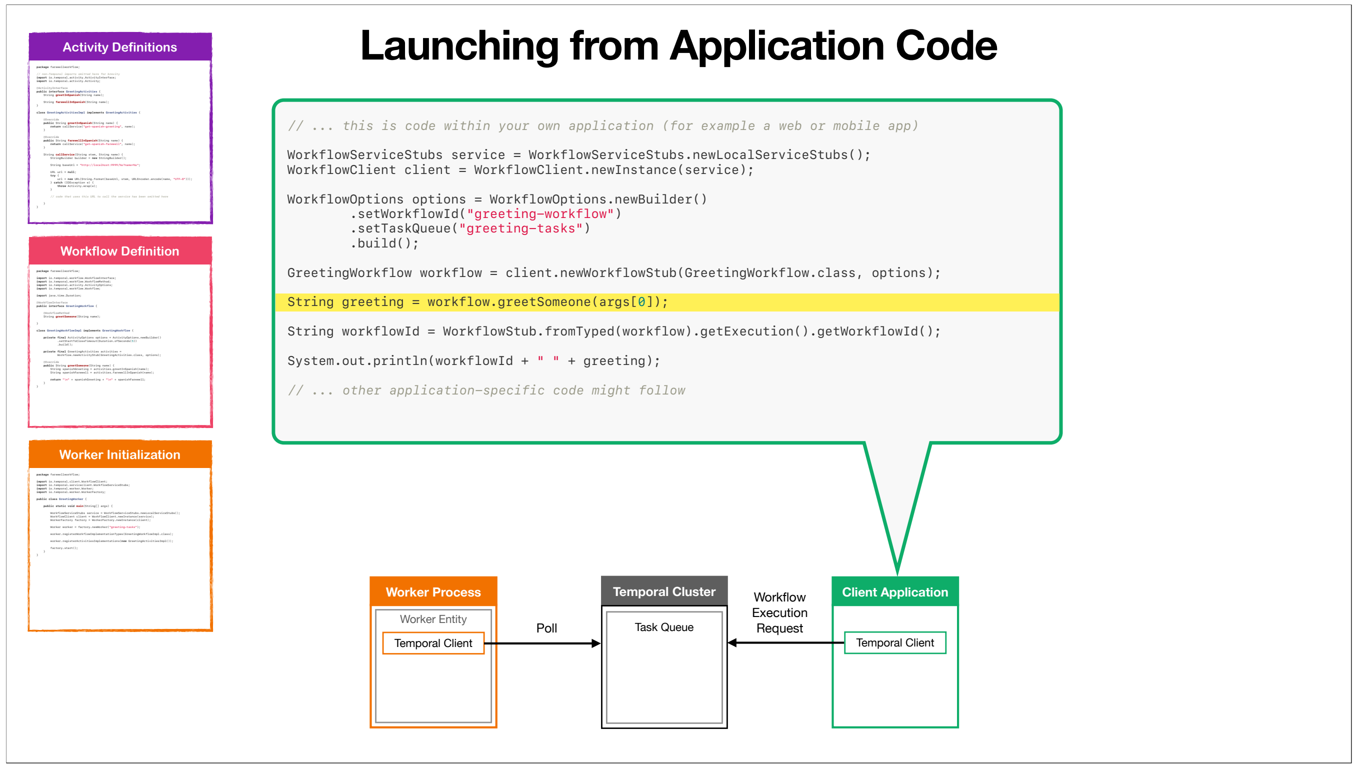


Launching from Command Line

```
$ temporal workflow start \
    --task-queue greeting-tasks \
    --workflow-id greeting-workflow \
    --workflow-type GreetingWorkflow \
    --input '"Mason"'
```

One way to start the Workflow is with the tctl command-line tool. This example specifies the name of the Worker's task queue, a user-defined Workflow ID, the Workflow Type, and the input data.

--

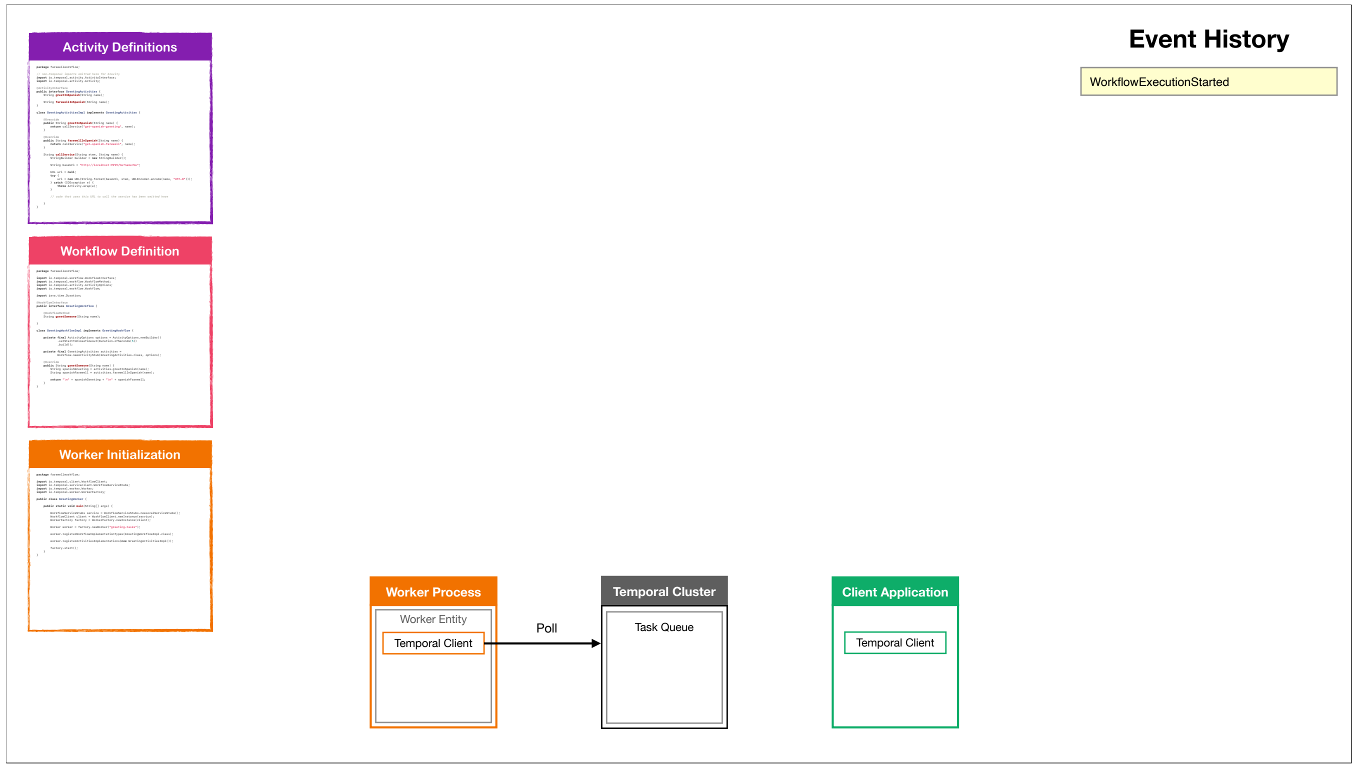


Launching from Application Code

```
// ... this is code within your own application (for example a web or mobile app)

WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
WorkflowClient client = WorkflowClient.newInstance(service);

WorkflowOptions options = WorkflowOptions.newBuilder()
        .setWorkflowId("greeting-workflow")
        .setTaskQueue("greeting-tasks")
        .build();

GreetingWorkflow workflow = client.newWorkflowStub(GreetingWorkflow.class, options);

String greeting = workflow.greetSomeone(args[0]);

String workflowId = WorkflowStub.fromTyped(workflow).getExecution().getWorkflowId();

System.out.println(workflowId + " " + greeting);

// ... other application-specific code might follow
```

An alternative is to start it from code within your own application by using a Temporal client to call the ExecuteWorkflow function with your input.

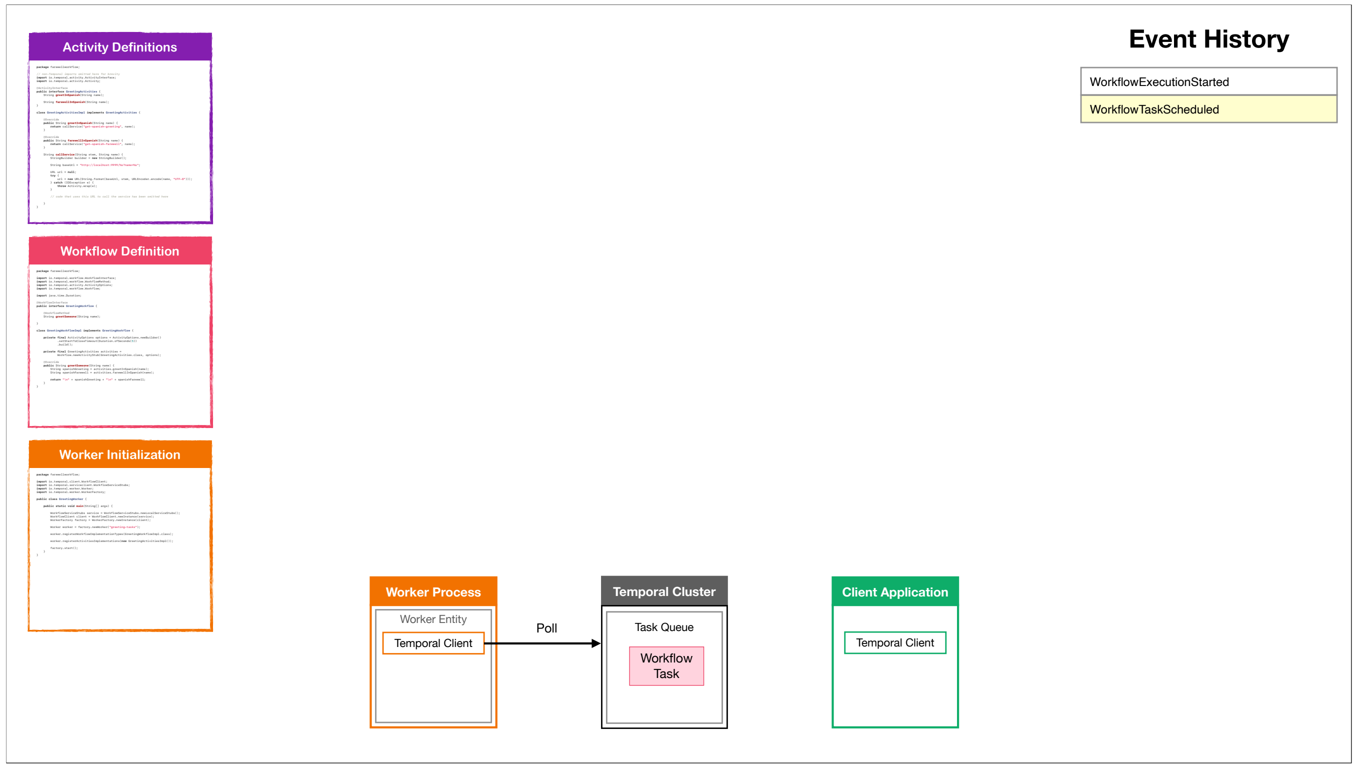
--



Regardless of how you start the Workflow, the behavior will be the same: the Temporal Cluster records a new Event into the Event History of this Workflow Execution. WorkflowExecutionStarted is always the first Event.

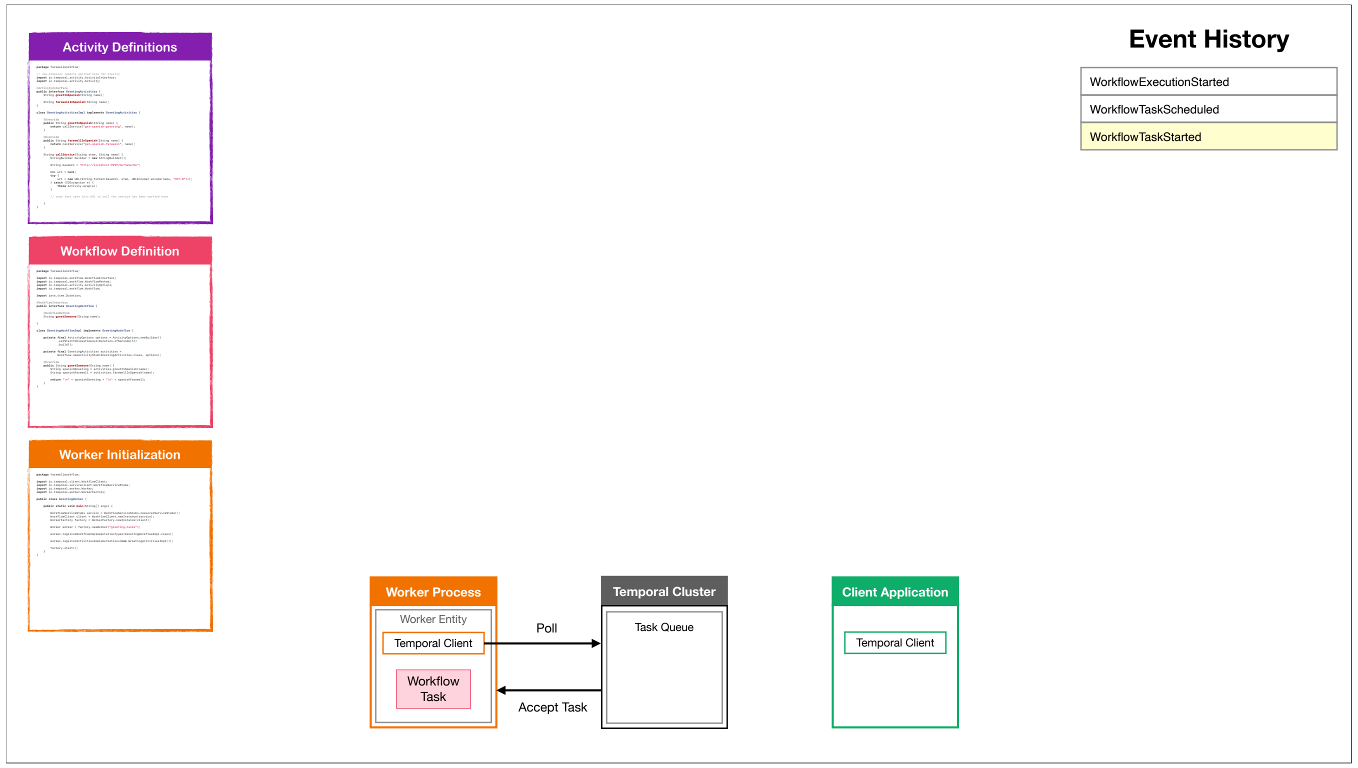
As I continue with my explanation, pay attention to the Event History shown on the right. Additional events will begin appearing below this one as Workflow Execution progresses. I won't mention all of them, but I highlight them in yellow when they first appear so they're easier to spot.

--

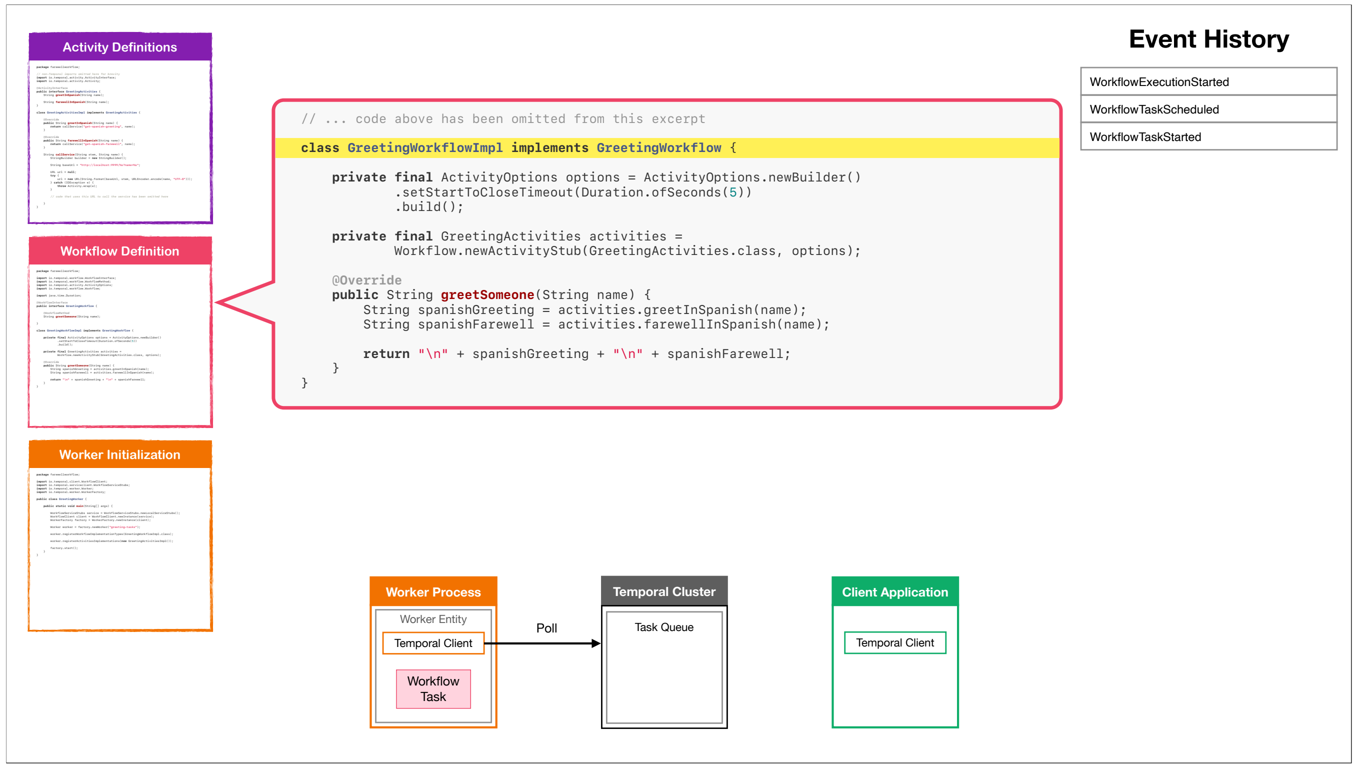


The Temporal Cluster adds a Workflow Task to the Task Queue and records another event, WorkflowTaskScheduled, into the Event History. Its name follows a pattern: when a new Task is added to the queue, the name ends with "Scheduled."

--



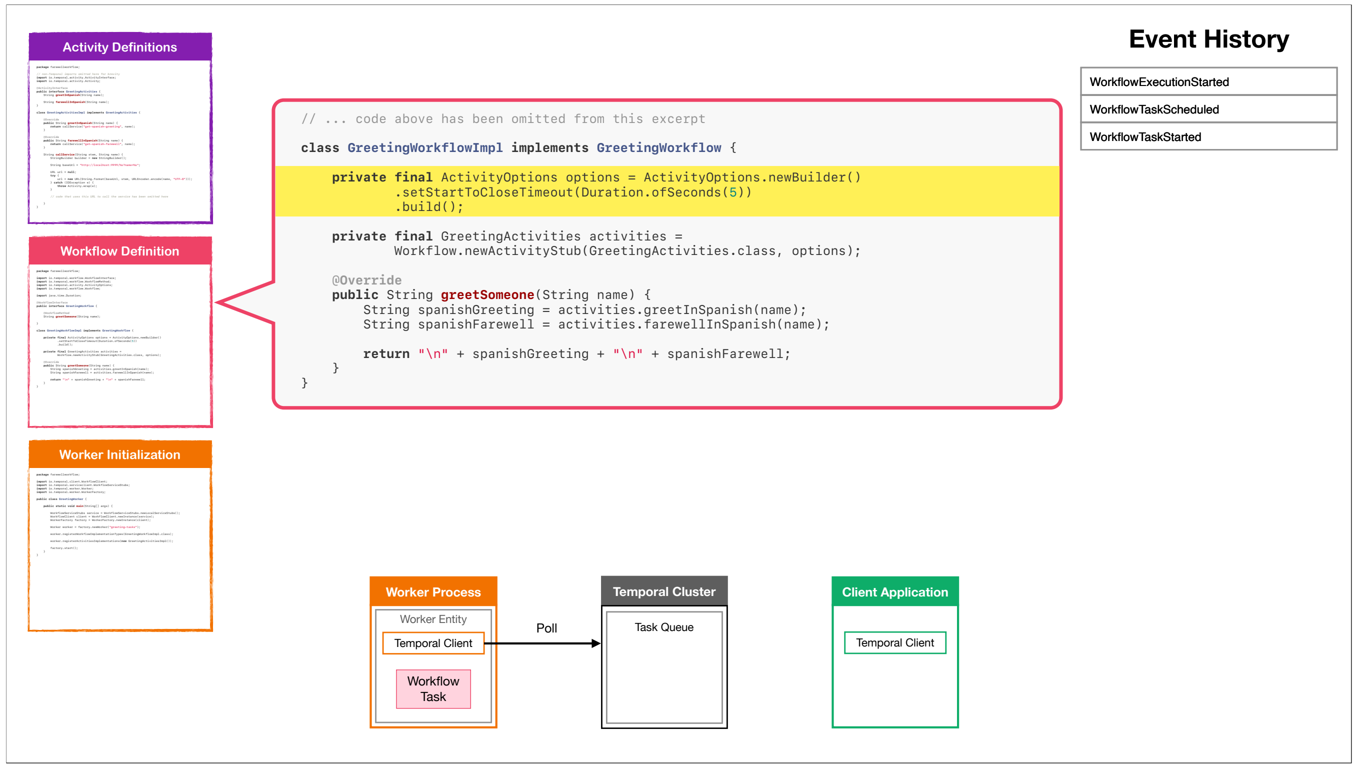
Since the Worker Process has capacity to do some processing work, it accepts this new Task. This results in a new Event, one whose name also follows a pattern. When a Worker dequeues a Task, the Cluster generates an event whose name ends with "Started."
--



```
// ... code above has been omitted from this excerpt
class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

The Worker Process begins the Workflow Task by invoking the function from the Workflow Definition.

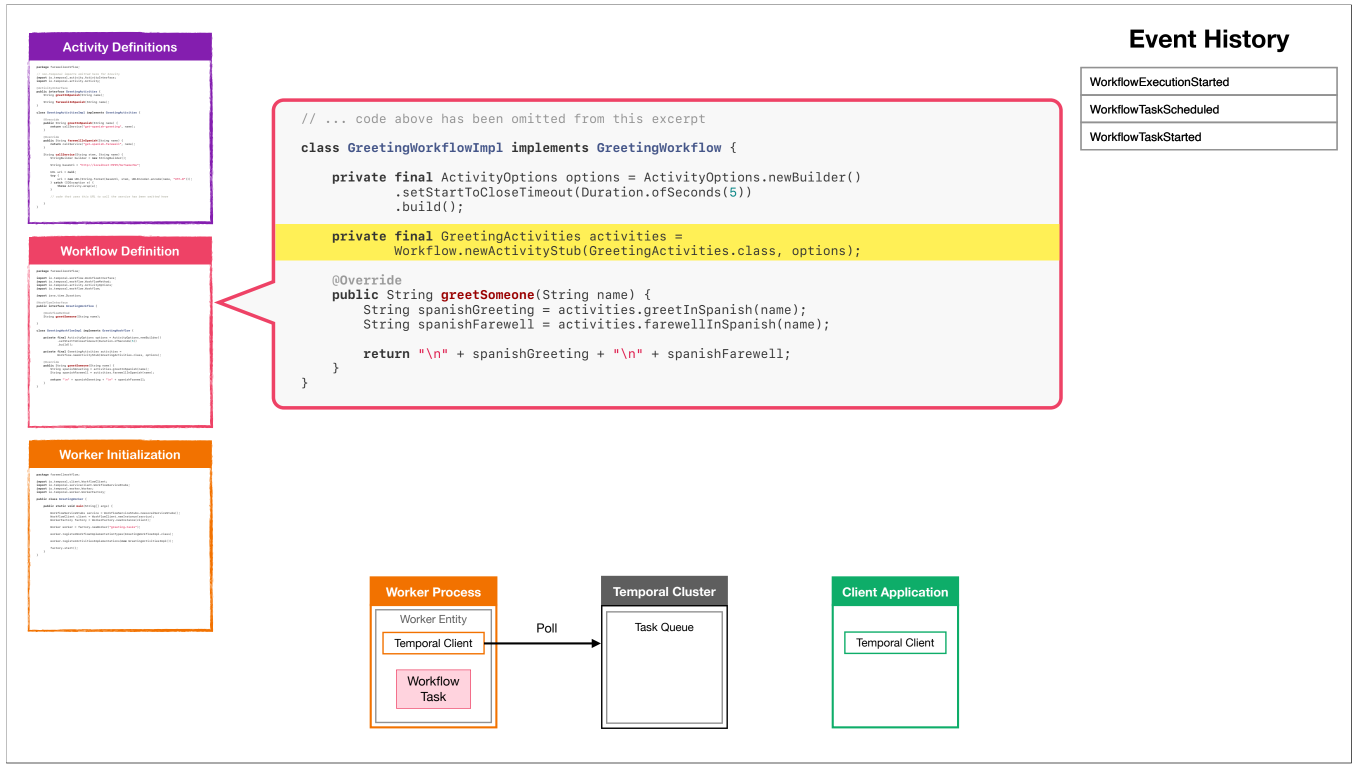
--



```
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

It continues by running code within this function. In this example, the first few statements configure timeout options for the Activities.

--



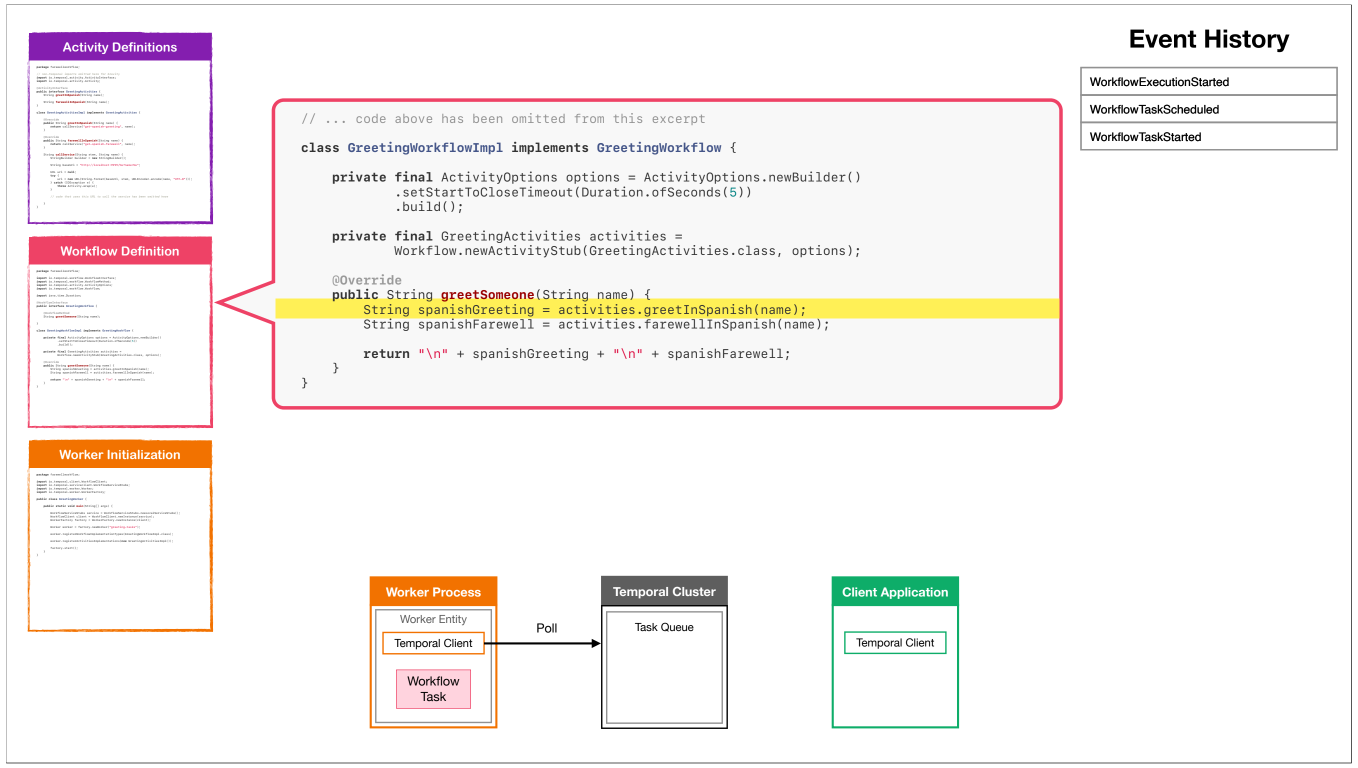
```
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

Event History

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted

It then creates the activities object taking in the ActivityOptions and the Activity Interface

--



```
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

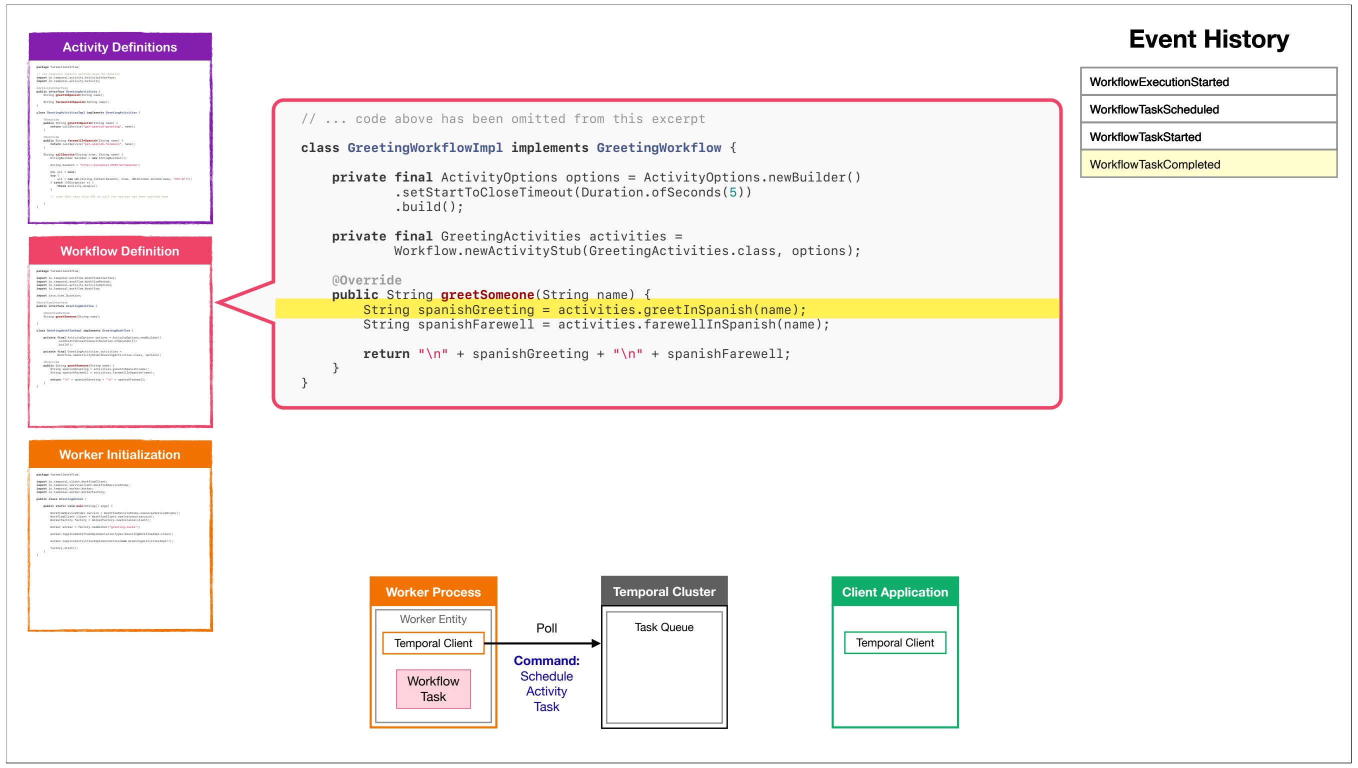
Event History

WorkflowExecutionStarted
WorkflowTaskScheduled
WorkflowTaskStarted

The Workflow code highlighted here declares a variable that will receive the output of our first Activity and then requests execution of that Activity: GreetInSpanish. Since ExecuteActivity returns a Future, and this example invokes a Get function on that, it will block until the Activity Execution completes, at which point we can access the output if the execution was successful or the error if it was not.

A few important things happen as a result of the ExecuteActivity call. The Worker can't make further progress on the Workflow until the Activity Execution concludes, so it notifies the Cluster that *the* current Workflow Task is complete. In response, the Cluster adds a new Event to history. The Worker also sends a command to the cluster requesting it to schedule an Activity Task.

--

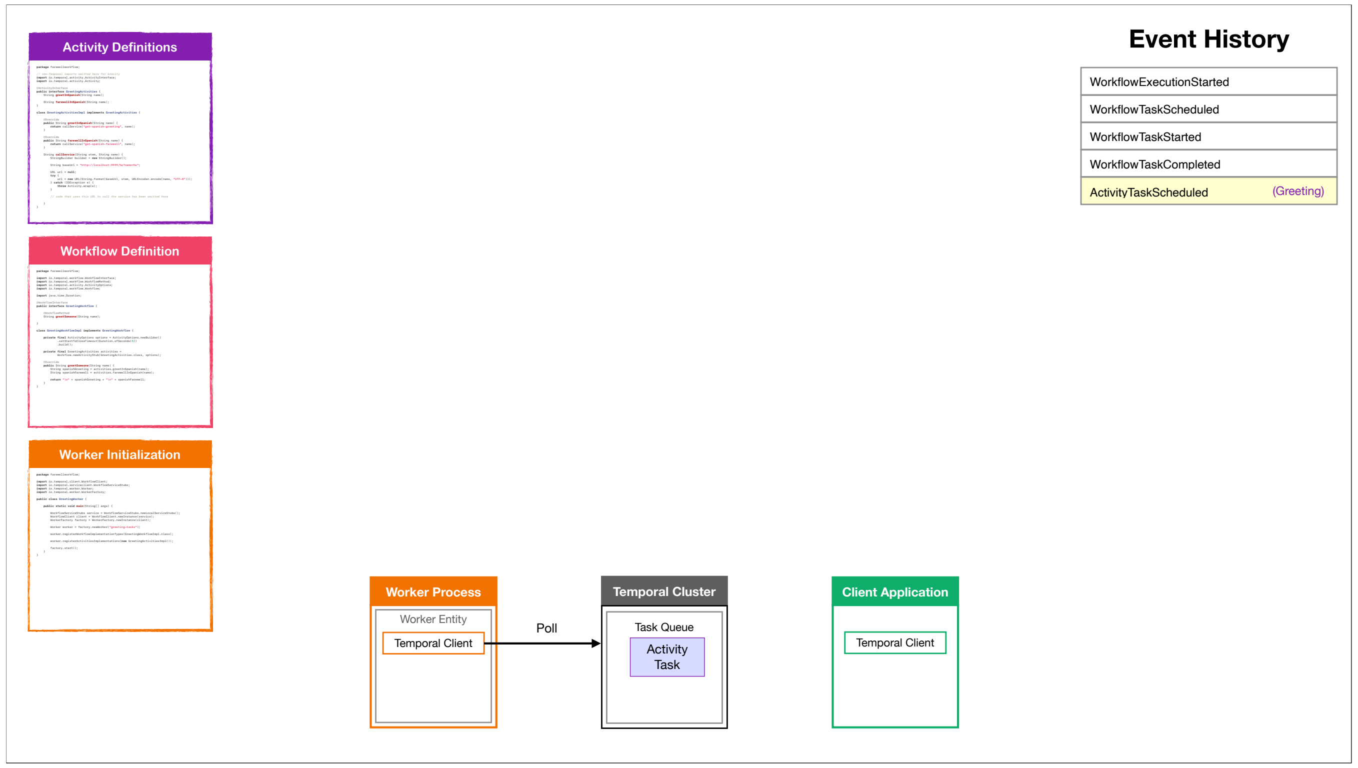


```java
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

The Workflow code highlighted here declares a variable that will receive the output of our first Activity and then requests execution of that Activity: GreetInSpanish. Since ExecuteActivity returns a Future, and this example invokes a Get function on that, it will block until the Activity Execution completes, at which point we can access the output if the execution was successful or the error if it was not.

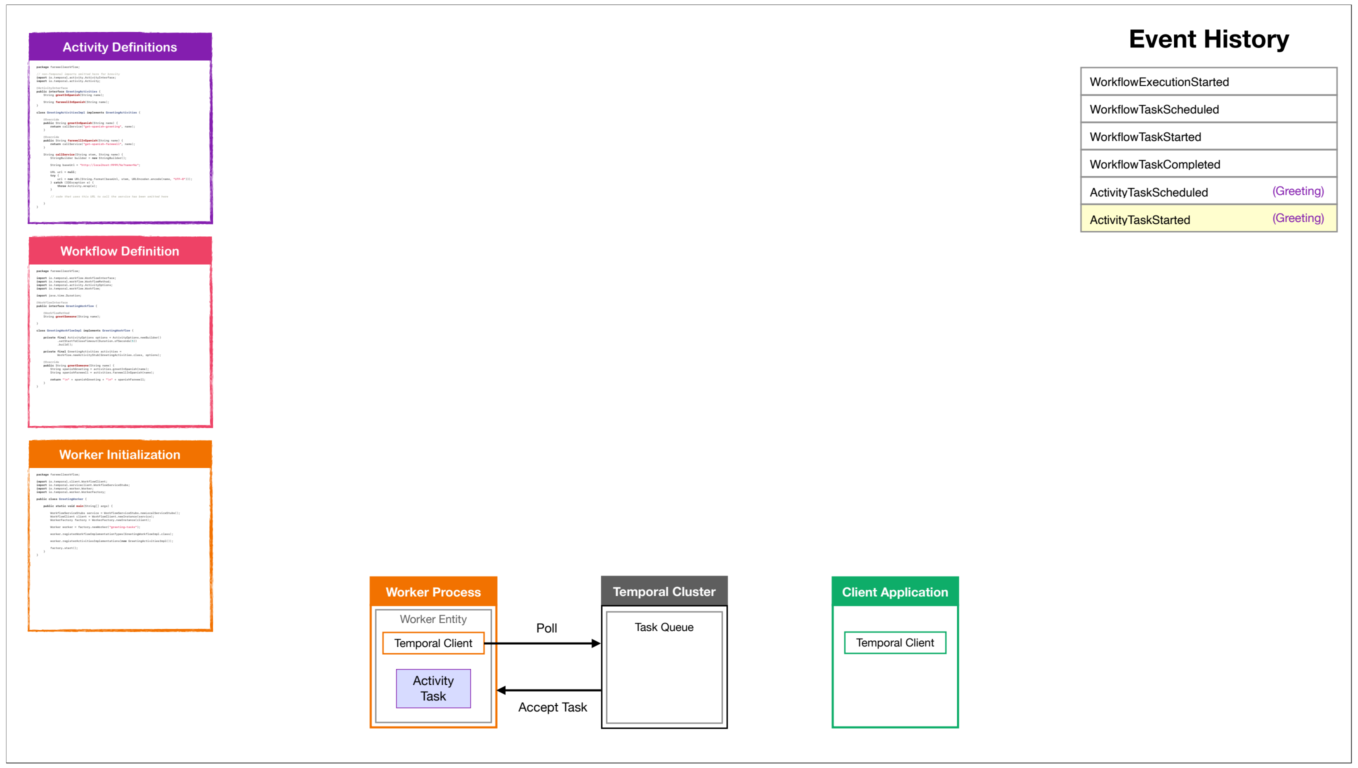
A few important things happen as a result of the ExecuteActivity call. The Worker can't make further progress on the Workflow until the Activity Execution concludes, so it notifies the Cluster that *the* current Workflow Task is complete. In response, the Cluster adds a new Event to history. The Worker also sends a command to the cluster requesting it to schedule an Activity Task.

--



The Temporal Cluster creates an Activity Task and adds it to the Task Queue, resulting in a new Event.

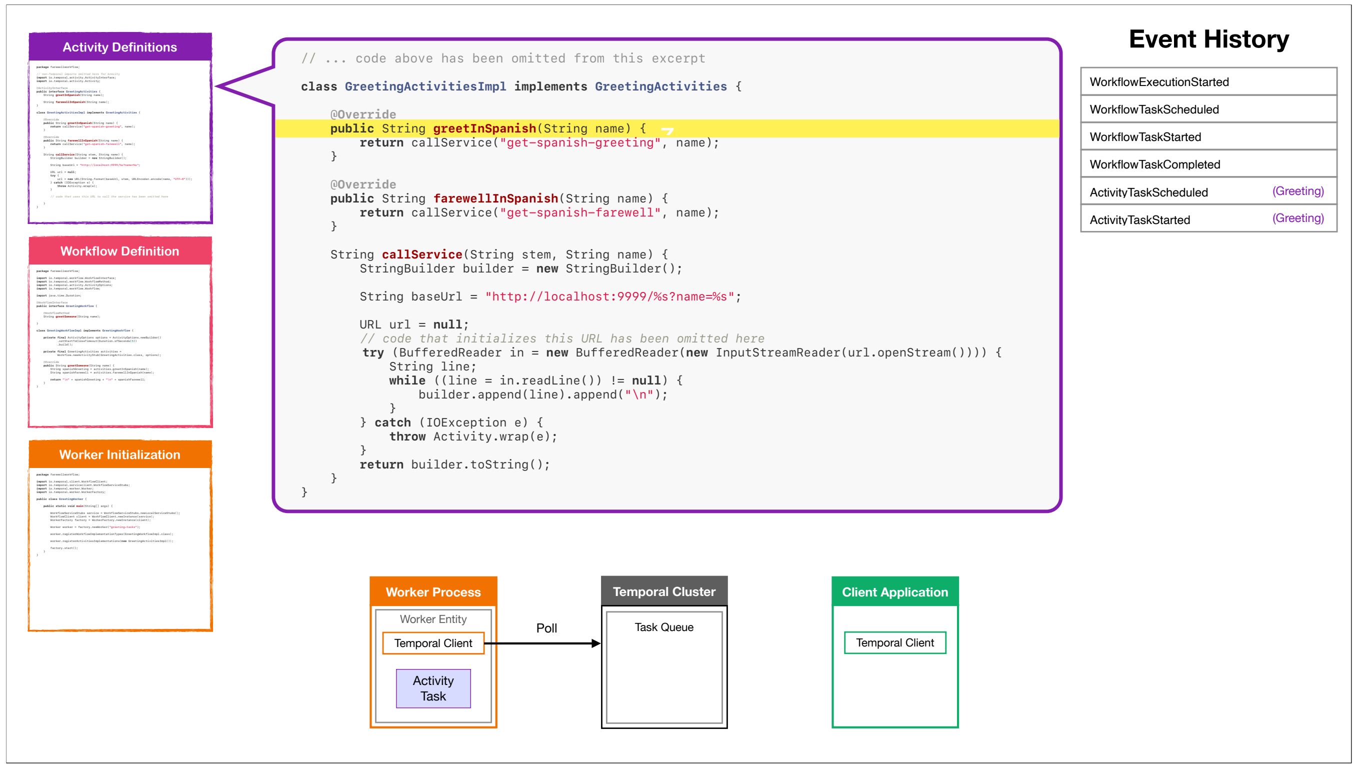
--



WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)

Since the Worker Process has capacity to perform additional work, it accepts the Activity Task.

--

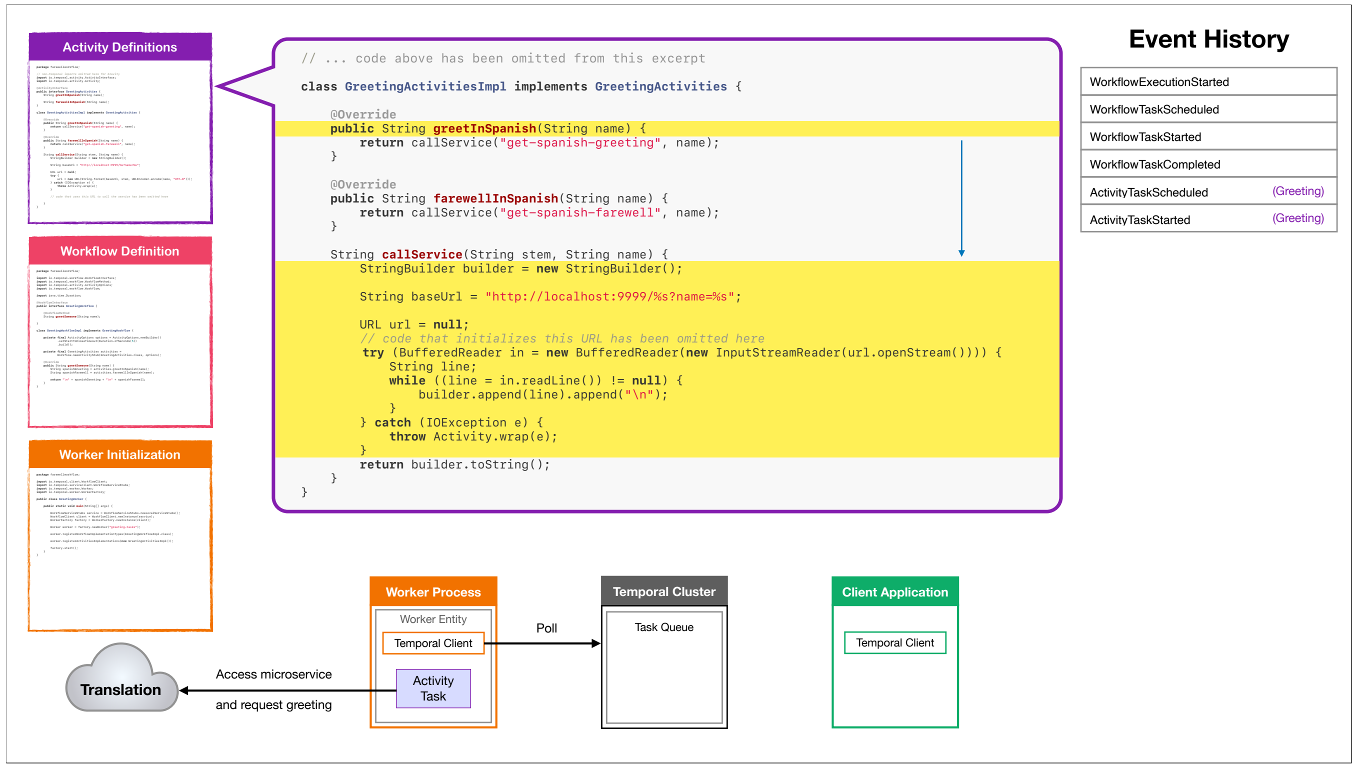


```
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
        try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)

The Worker Entity now invokes the function corresponding to the Activity Definition for the GreetInSpanish Activity.

--



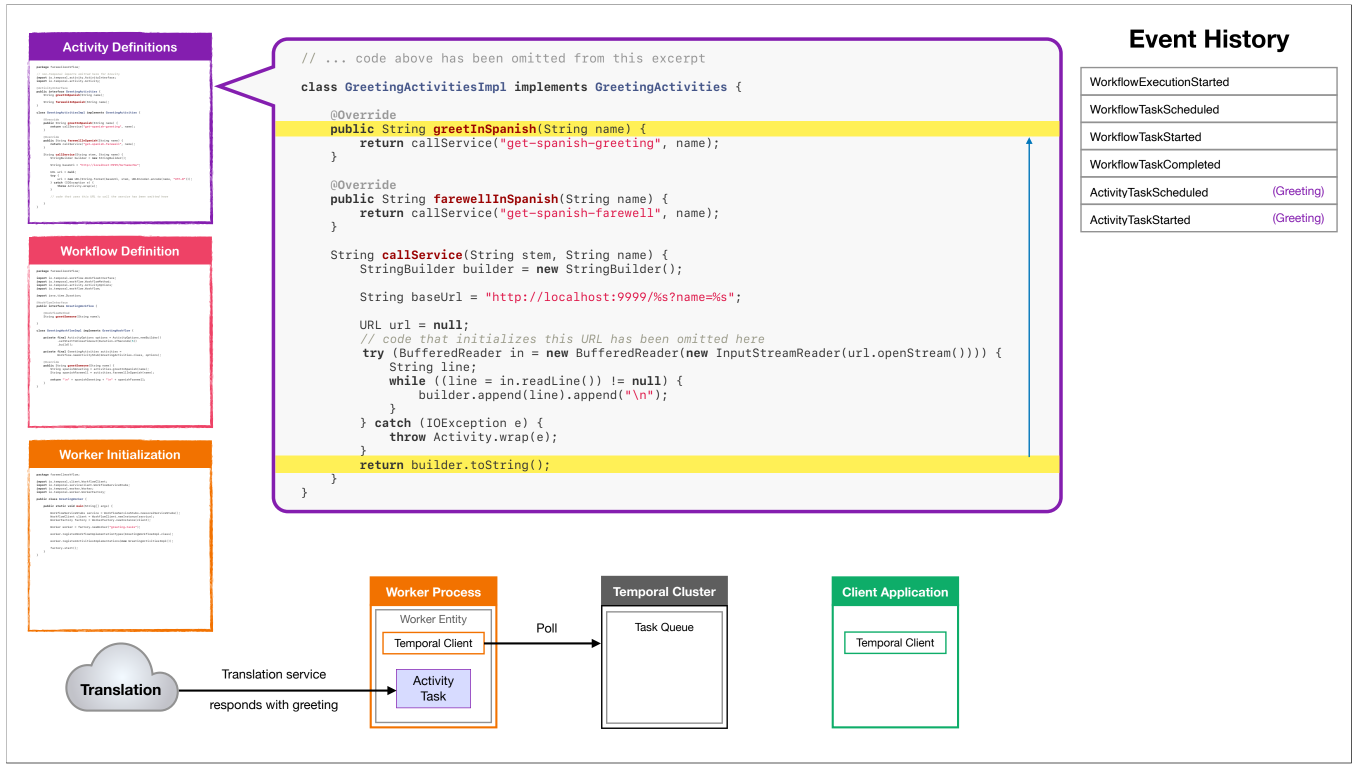
```
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
        try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)

The Worker then runs the code within the function. In this case, the Activity calls the utility function, which in turn issues a request to the microservice.

--



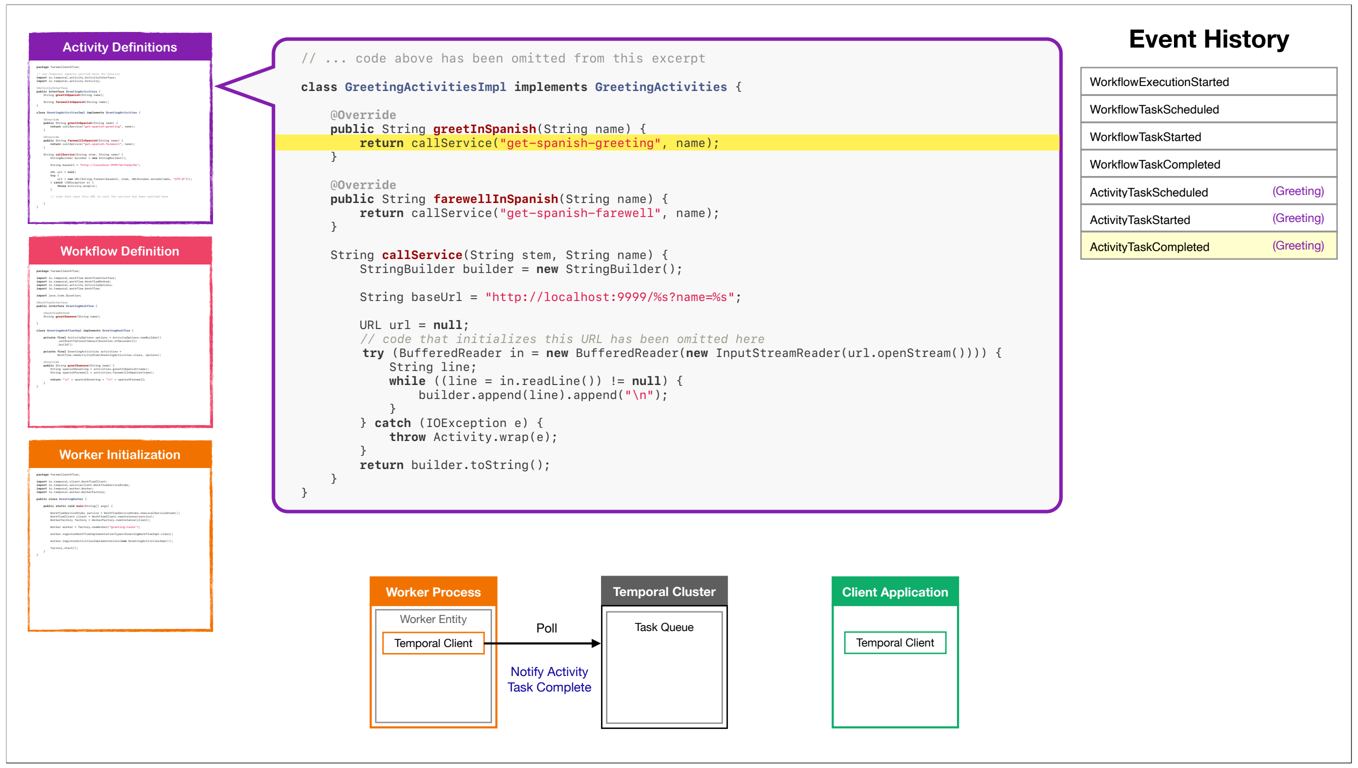
```java
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
         try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)

This request was successful and the service responds by providing a customized greeting in Spanish.

--



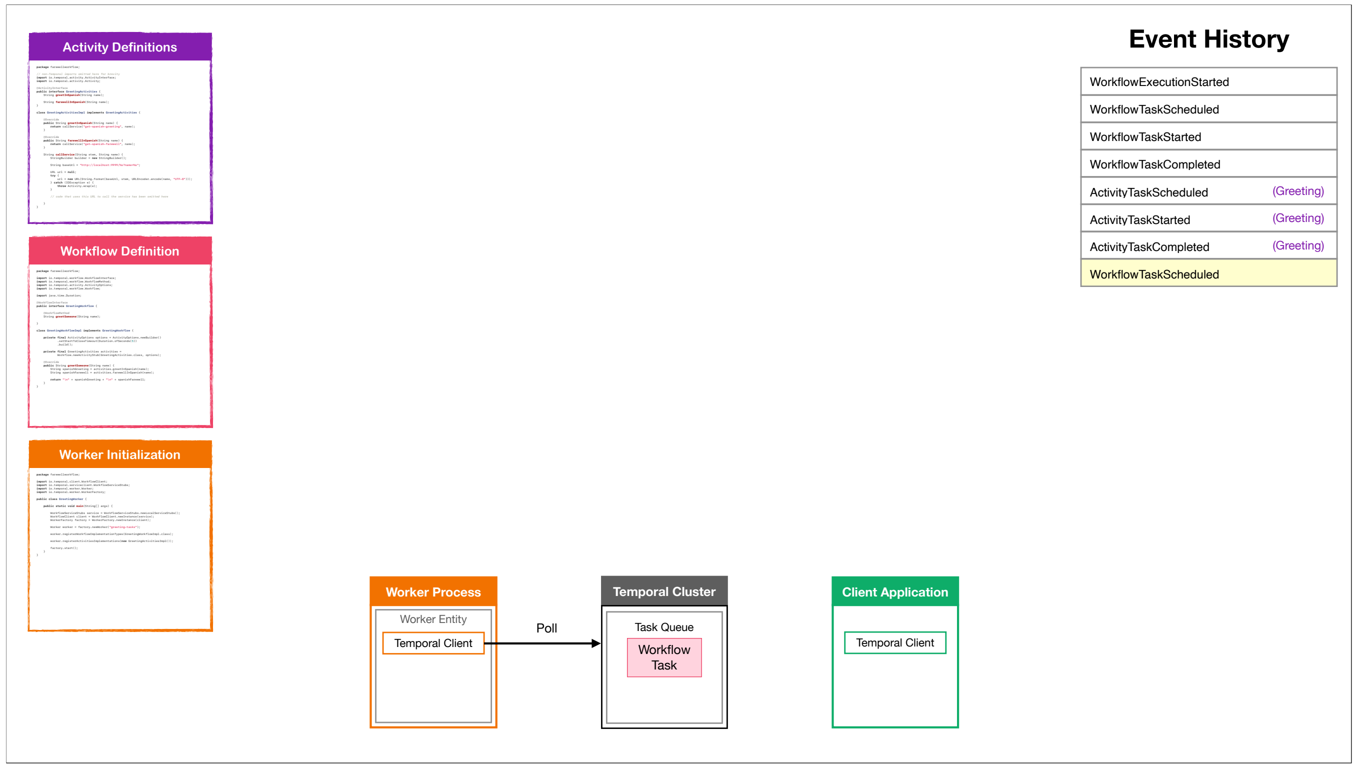
```
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
        try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)

When the Activity function returns, Worker notifies the cluster that the Activity Task is complete, resulting in a new Event.

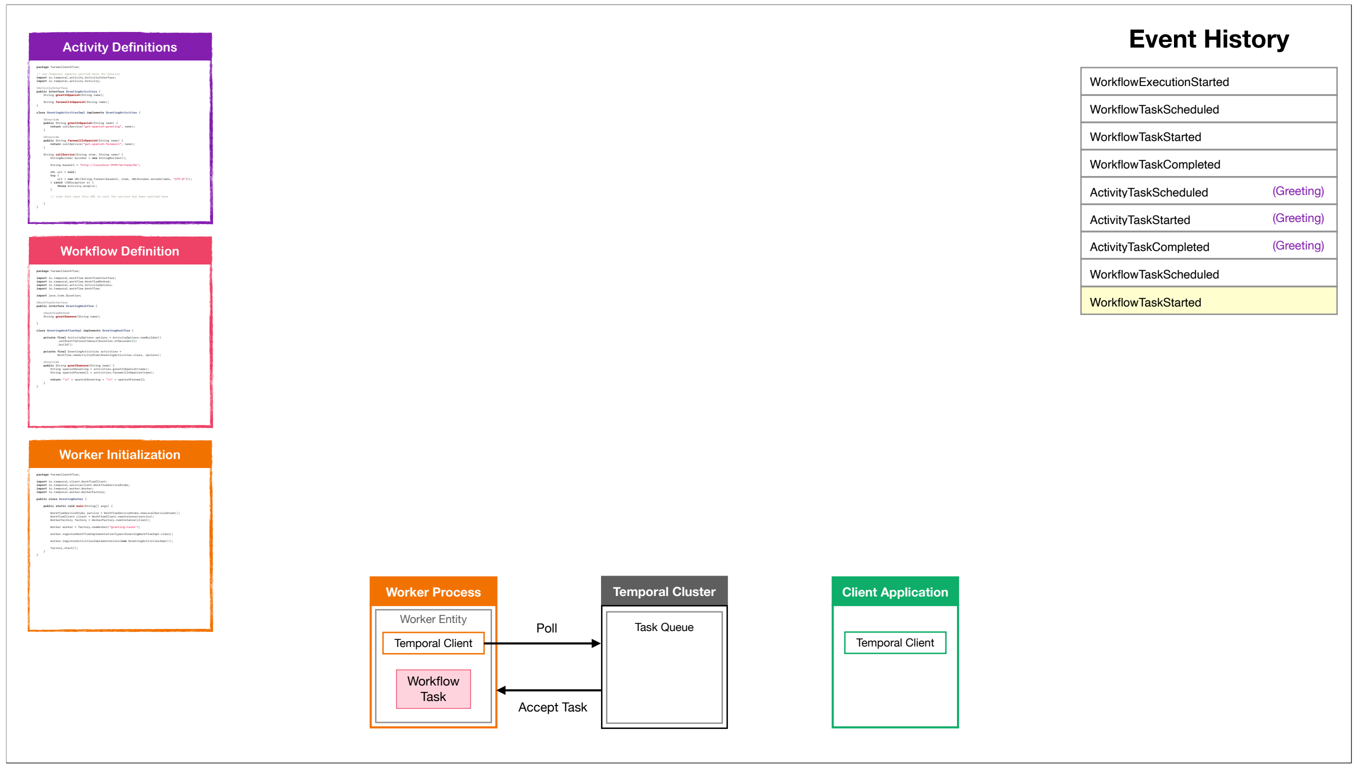
--



Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	

In response, the Temporal Cluster queues a new Workflow Task and logs another Event.
--

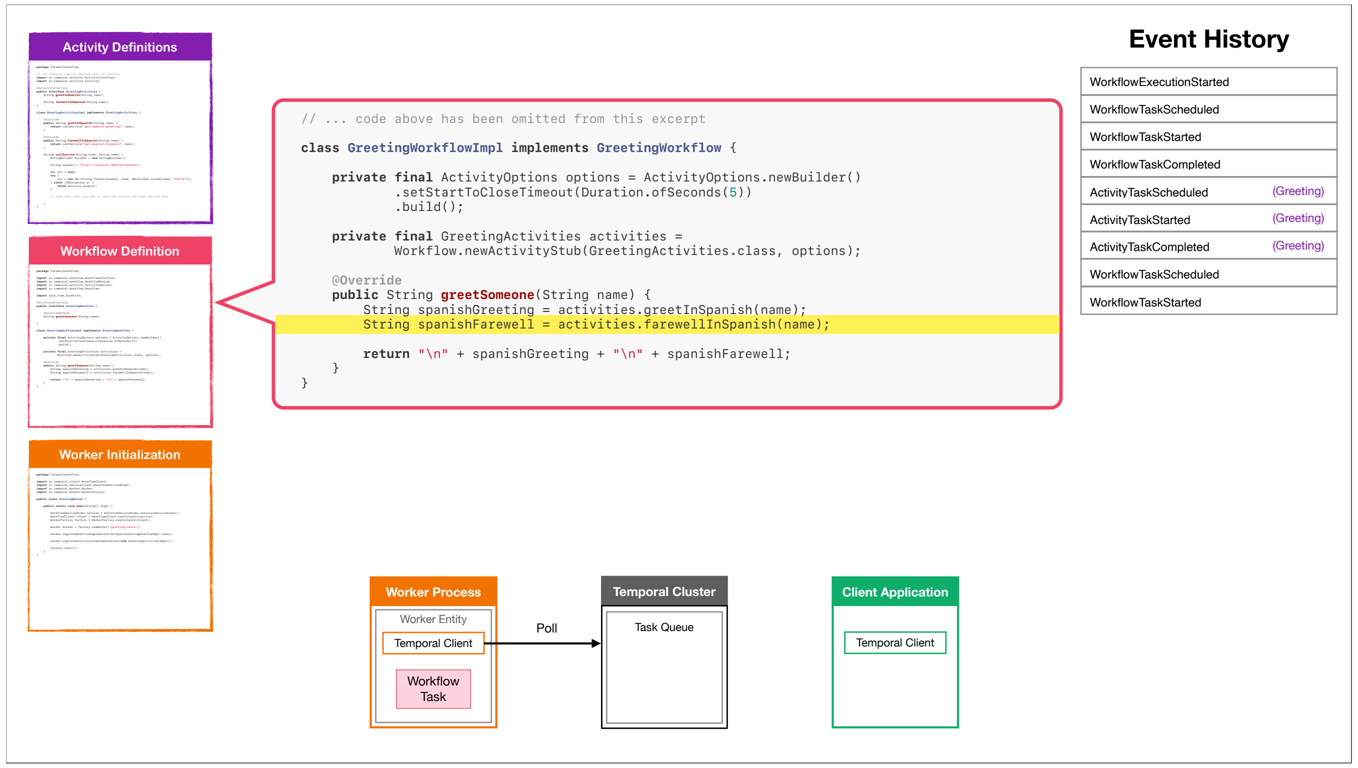


Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	

When the Worker accepts this new Task, the Temporal Cluster adds a WorkflowTaskStarted Event to the History.

--



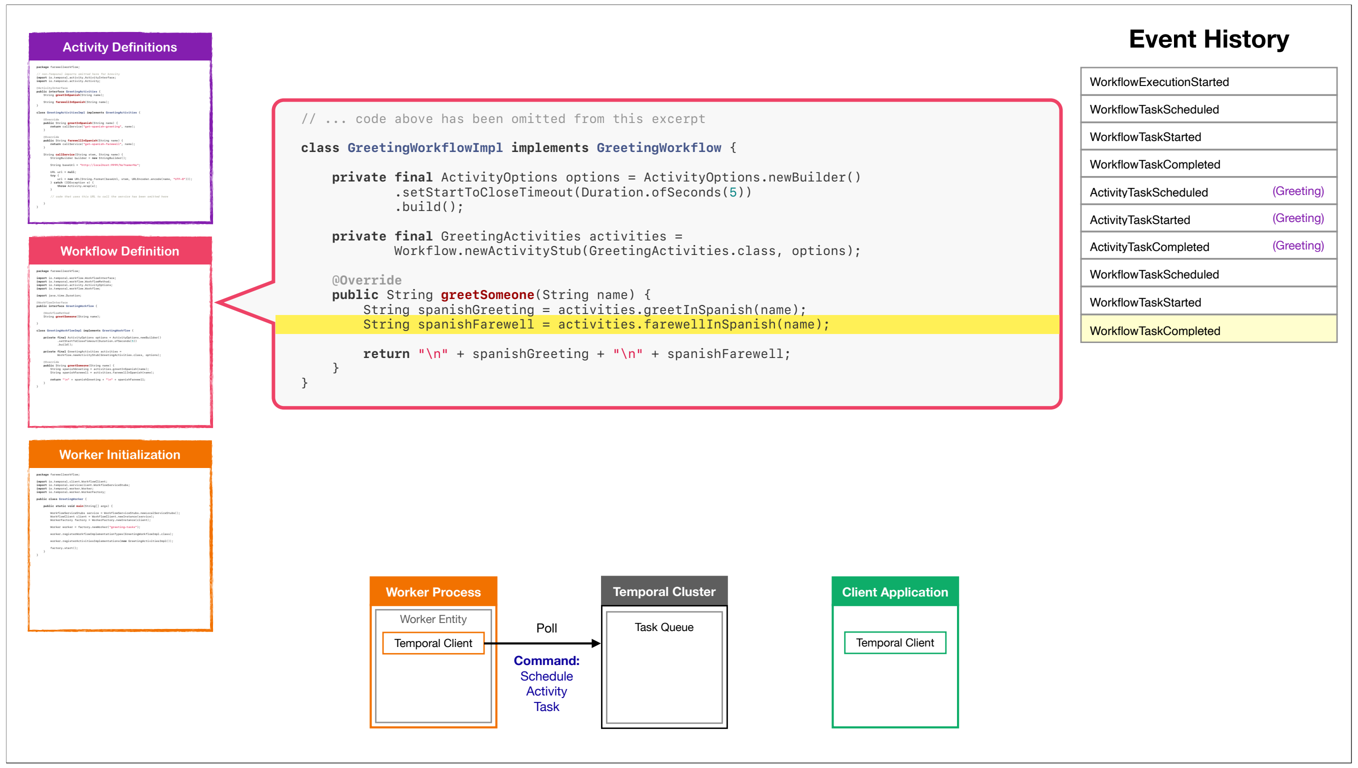
```java
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	

The Worker continues where it left off by executing the next statement in the Workflow Definition.

--



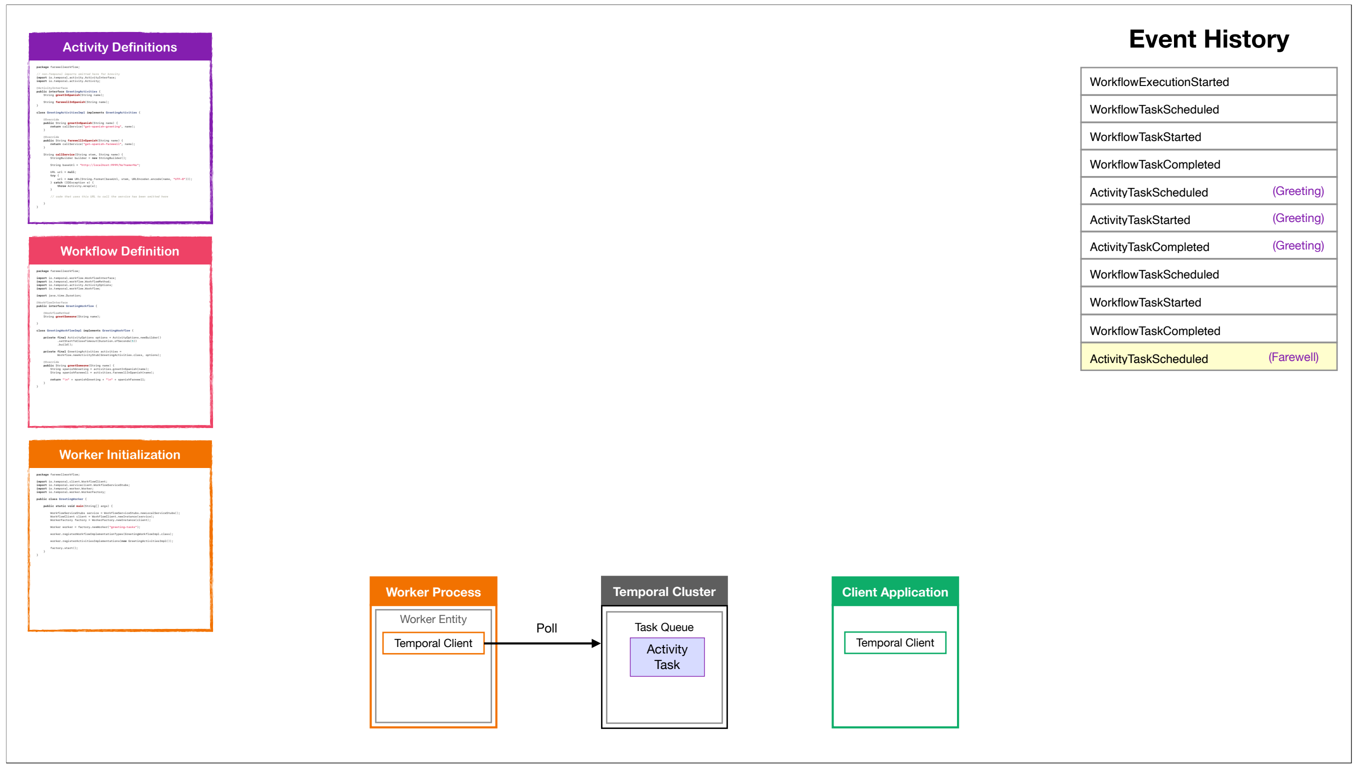
```java
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

Event History

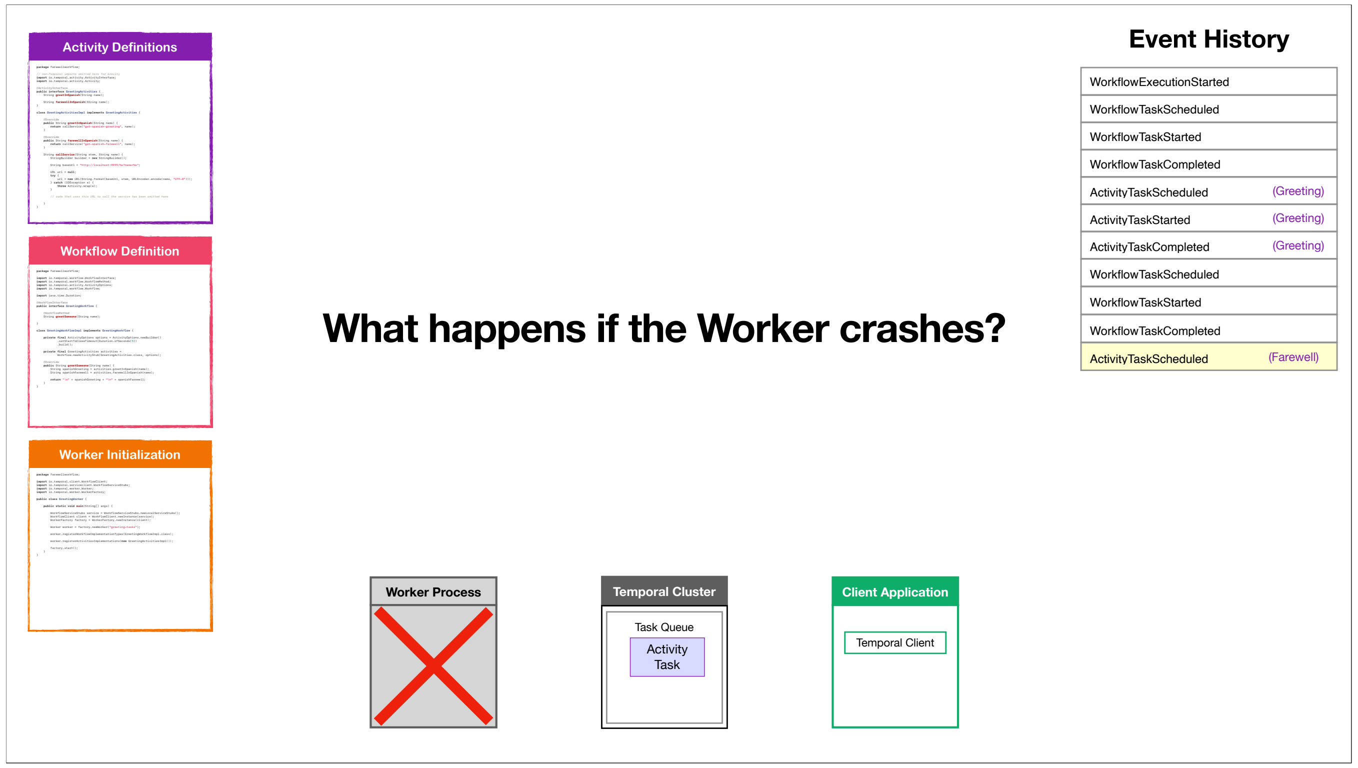
WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	

It is now time to execute the second Activity, so the Worker notifies the Temporal Cluster that the current Workflow Task is complete and sends a Command to schedule an Activity Task.

--



The Temporal Cluster queues an Activity Task for the second Activity and logs an ActivityTaskScheduled Event to the history.
--



What happens if the Worker crashes?

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)

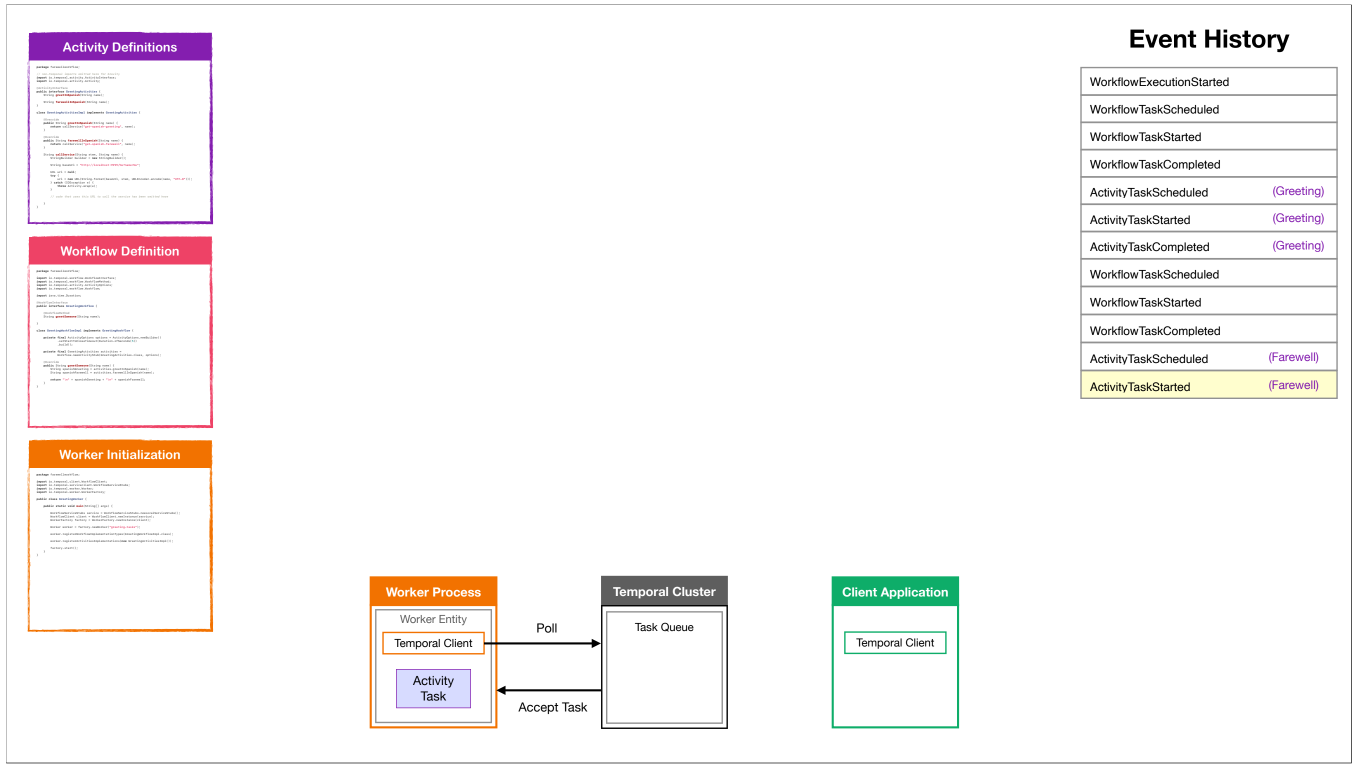
Let's take a moment to look at a failure scenario. What happens if the Worker crashes; for example, because it ran out of memory?

You can recover from this by restarting the Worker or launching a new Worker on a different machine. In either case, Temporal will automatically recreate the state of the Workflow up to the point of failure, so progress will continue on from there, as if the Worker never crashed at all.

Activities that completed successfully before the crash won't be executed again; instead, Temporal reuses the values returned by their previous executions.

This is something we'll cover in detail in Temporal 102

--

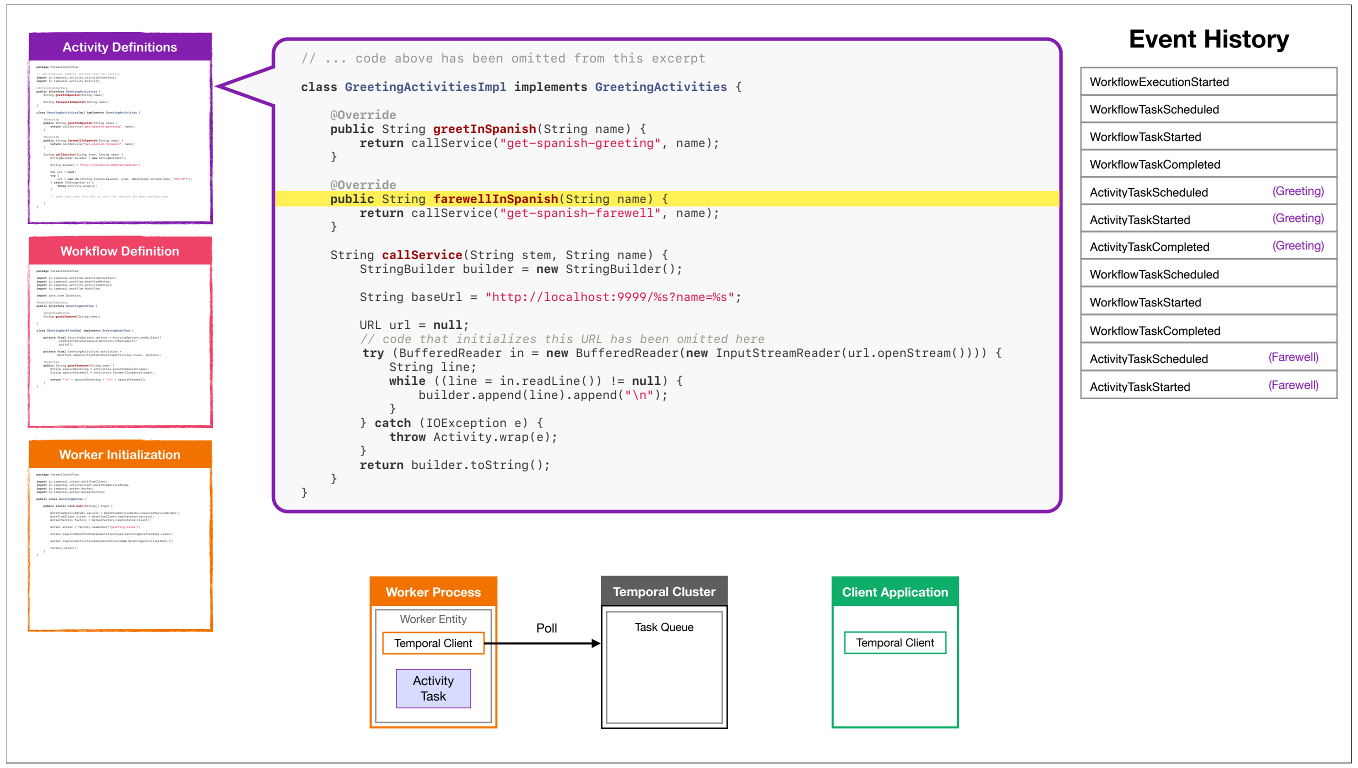


Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)

When the Worker accepts the Activity Task. The Temporal Cluster adds ActivityTaskStarted to the Event History.

--



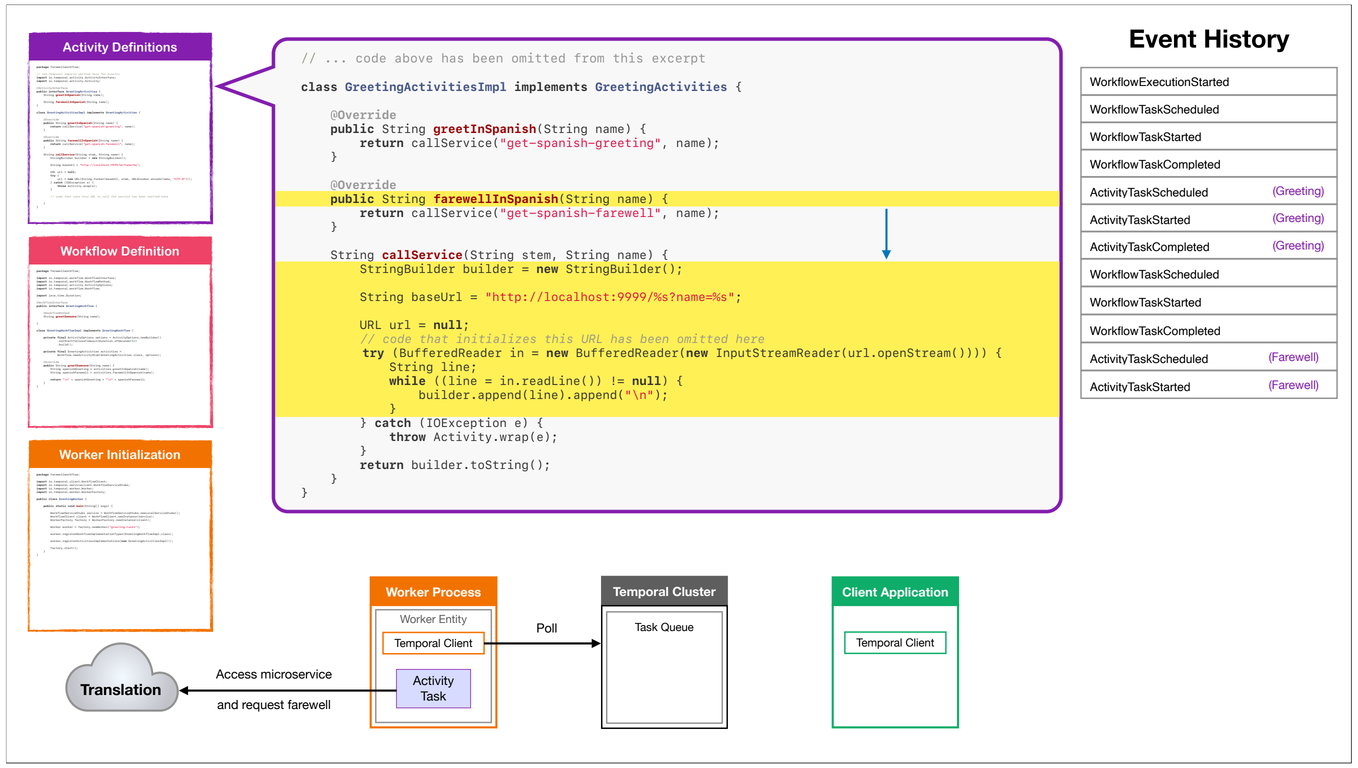
```
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
         try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)

The Worker now invokes the function for the second Activity.

--



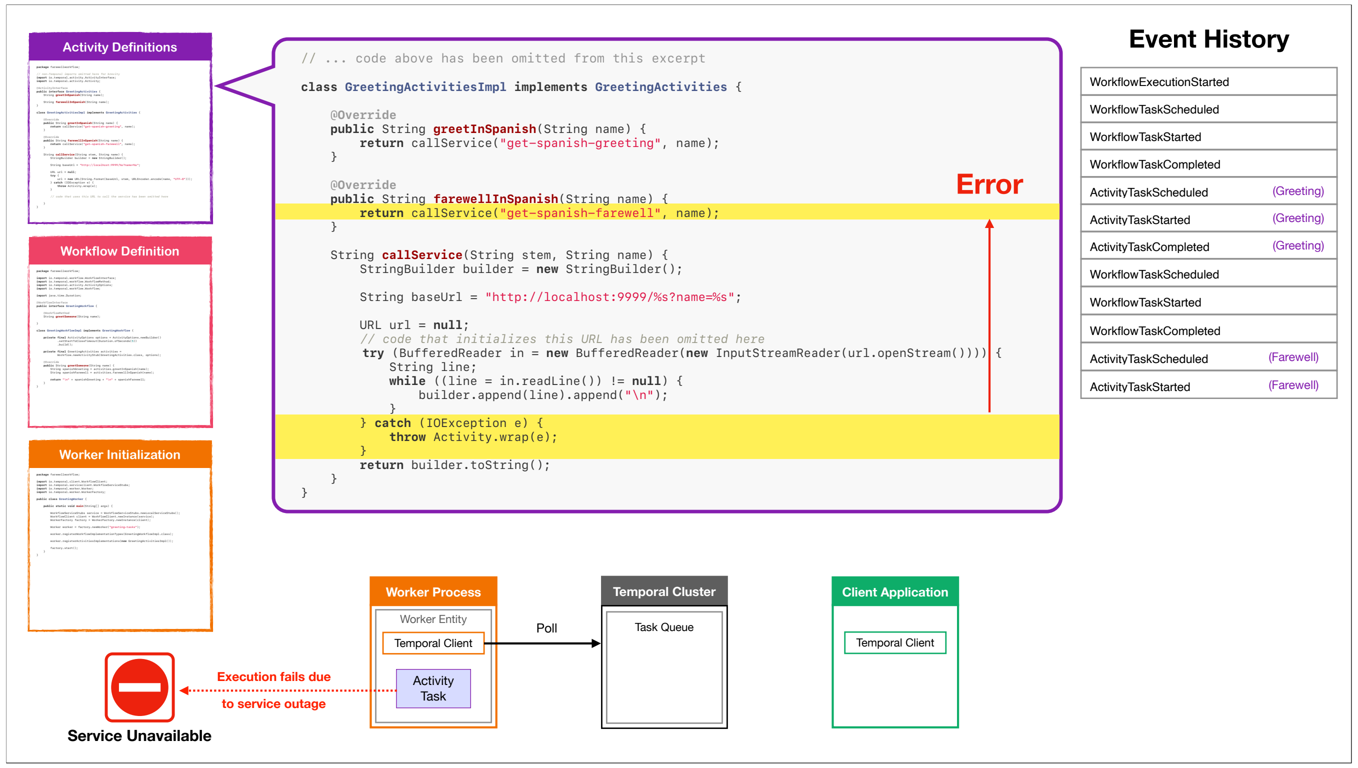
```
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
         try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)

As before, it then runs the code within the function, which uses the utility method to call a microservice.

--



```java
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
        try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

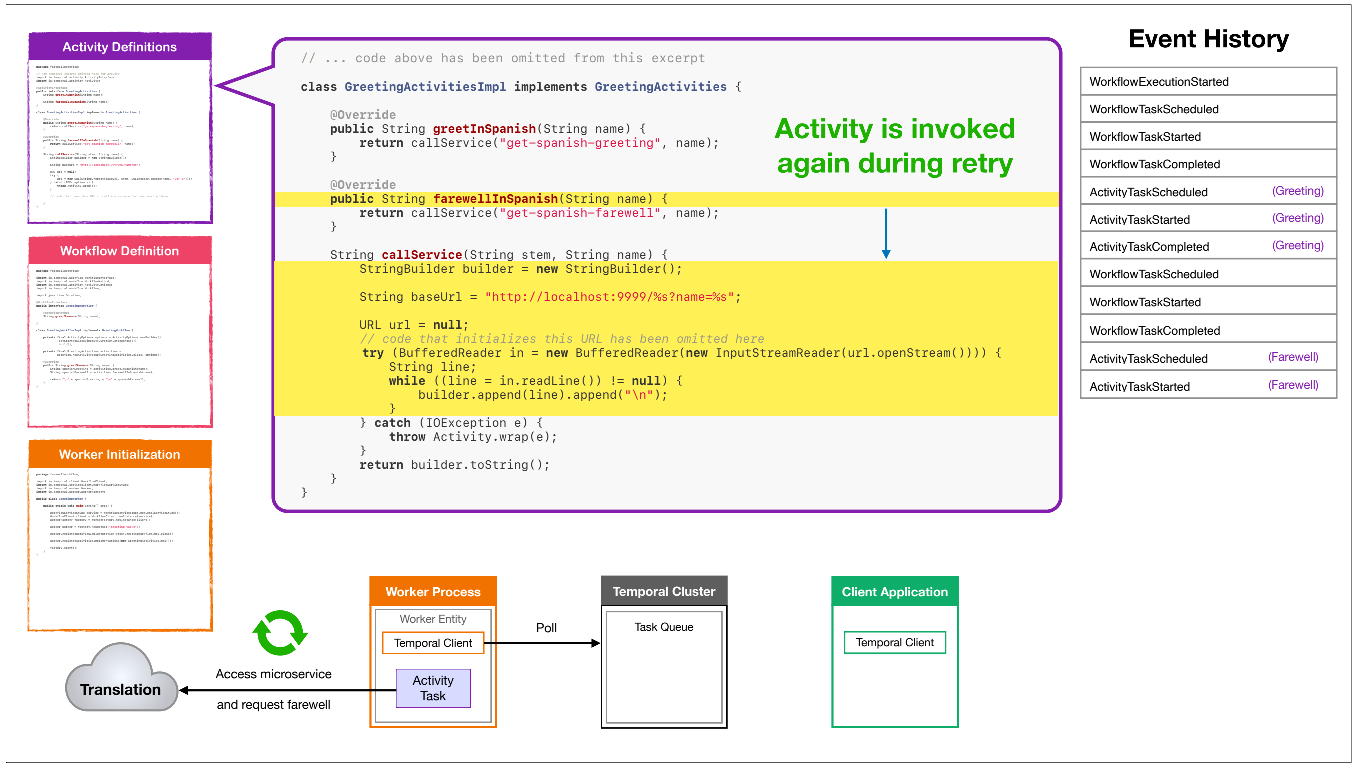
Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)

But what if that microservice went offline just before the request? In this case, the request would fail, ultimately causing the Activity function to return an error.

The default behavior in Temporal is for a failed Activity to be automatically retried, with a short delay, until it succeeds or is canceled. You can customize this behavior with a Retry Policy.

--



```java
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
        try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

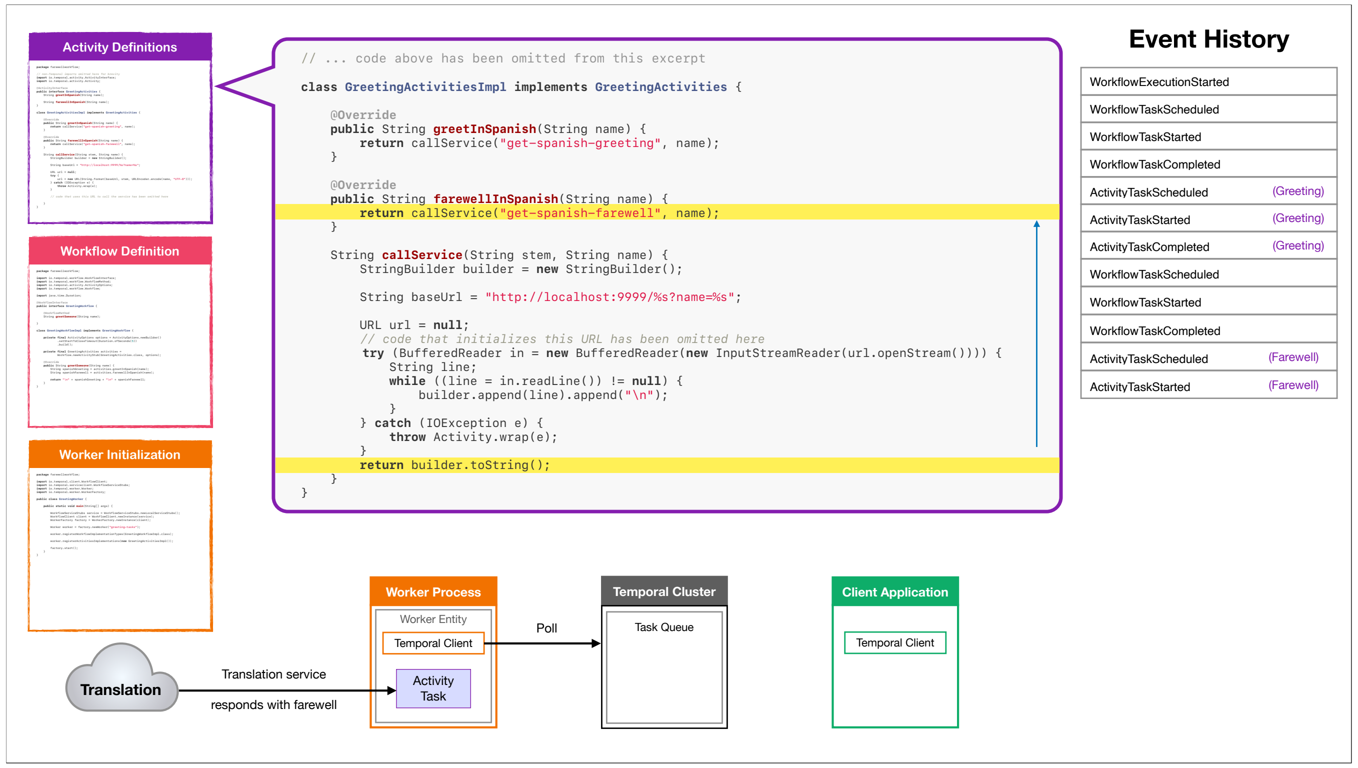
Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)

Through a retry, the Worker invokes the Activity function again, which in turn invokes the utility function and calls out to the microservice.

For this example, let's assume that the service outage was an intermittent failure, so the request made during the retry is successful.

--



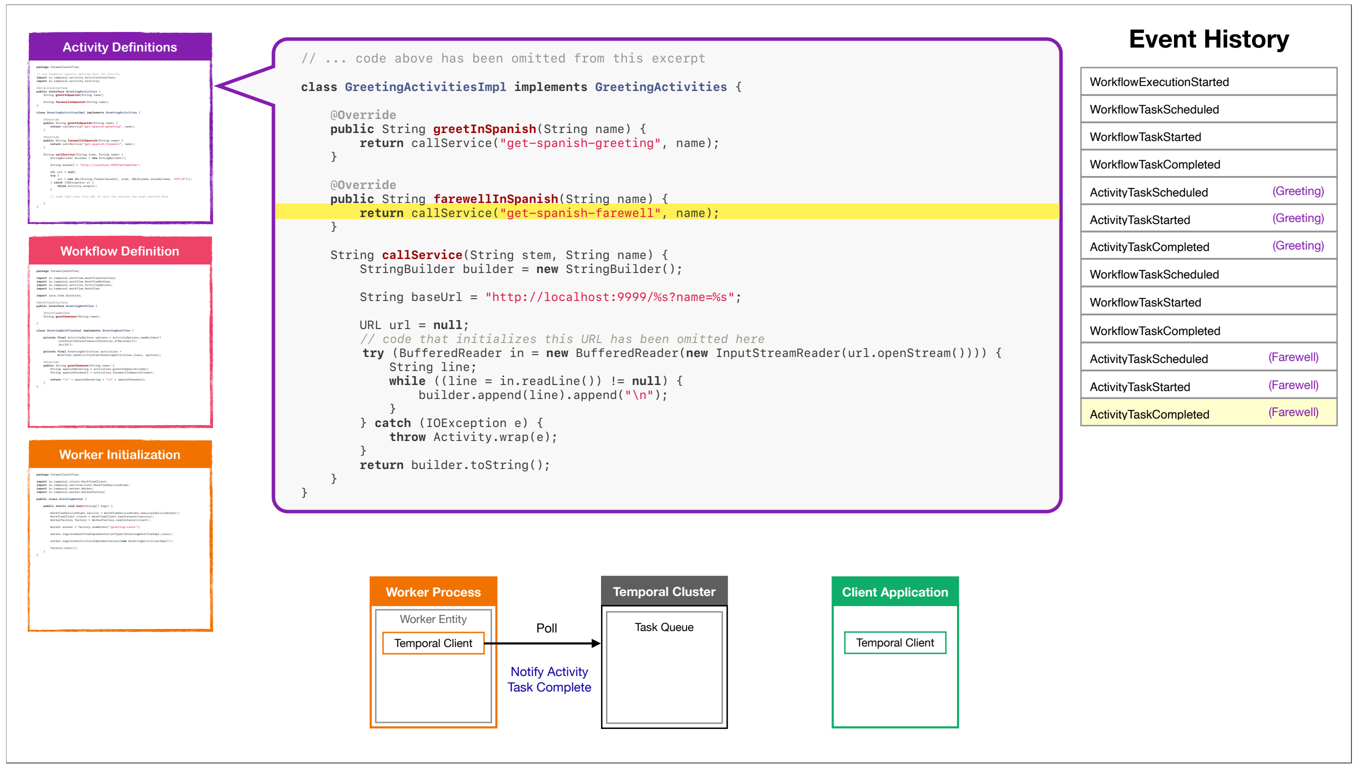
```
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
         try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)

Since the service is now back online, it responds to our latest request and provides the requested farewell message.

--



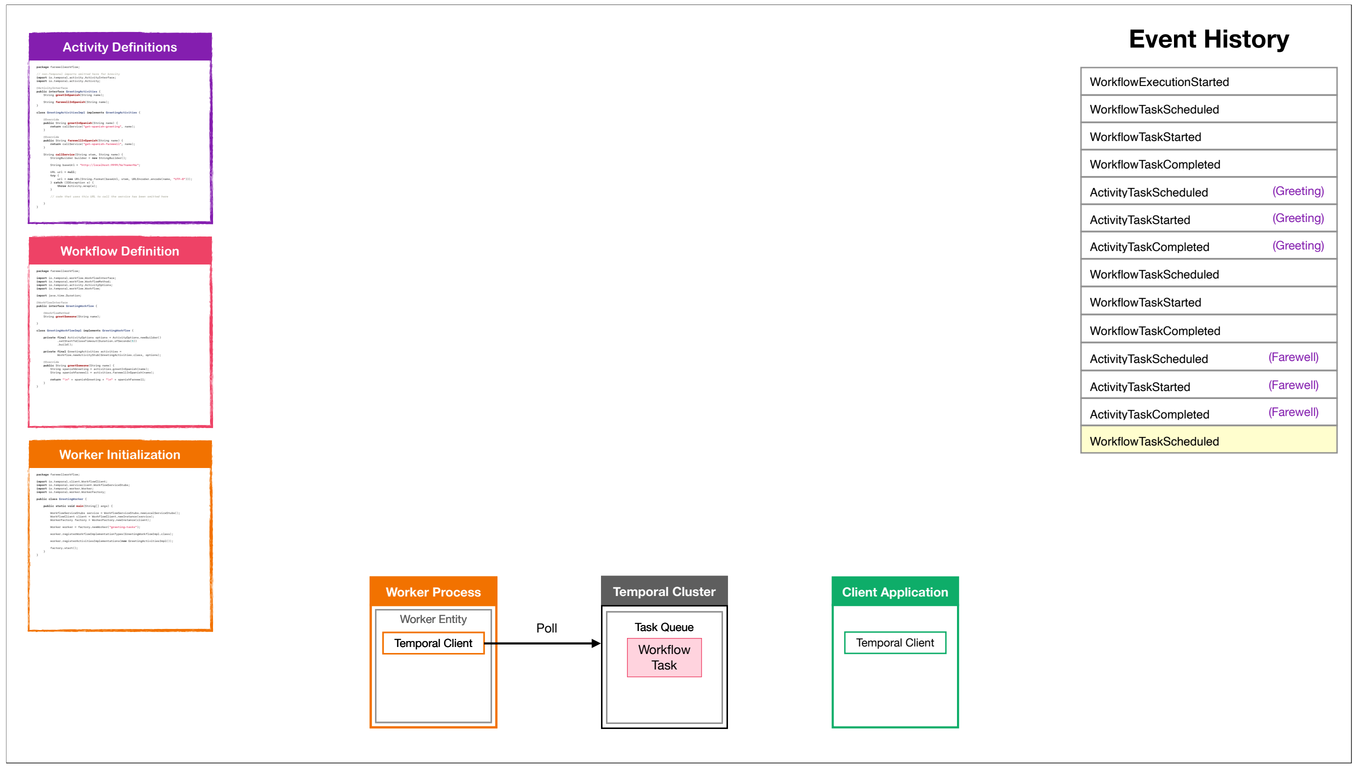
```
// ... code above has been omitted from this excerpt

class GreetingActivitiesImpl implements GreetingActivities {

    @Override
    public String greetInSpanish(String name) {
        return callService("get-spanish-greeting", name);
    }

    @Override
    public String farewellInSpanish(String name) {
        return callService("get-spanish-farewell", name);
    }

    String callService(String stem, String name) {
        StringBuilder builder = new StringBuilder();

        String baseUrl = "http://localhost:9999/%s?name=%s";

        URL url = null;
        // code that initializes this URL has been omitted here
        try (BufferedReader in = new BufferedReader(new InputStreamReader(url.openStream()))) {
            String line;
            while ((line = in.readLine()) != null) {
                builder.append(line).append("\n");
            }
        } catch (IOException e) {
            throw Activity.wrap(e);
        }
        return builder.toString();
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)
ActivityTaskCompleted	(Farewell)

When the function returns, the Worker notifies the Temporal Cluster that the Activity Task is complete.

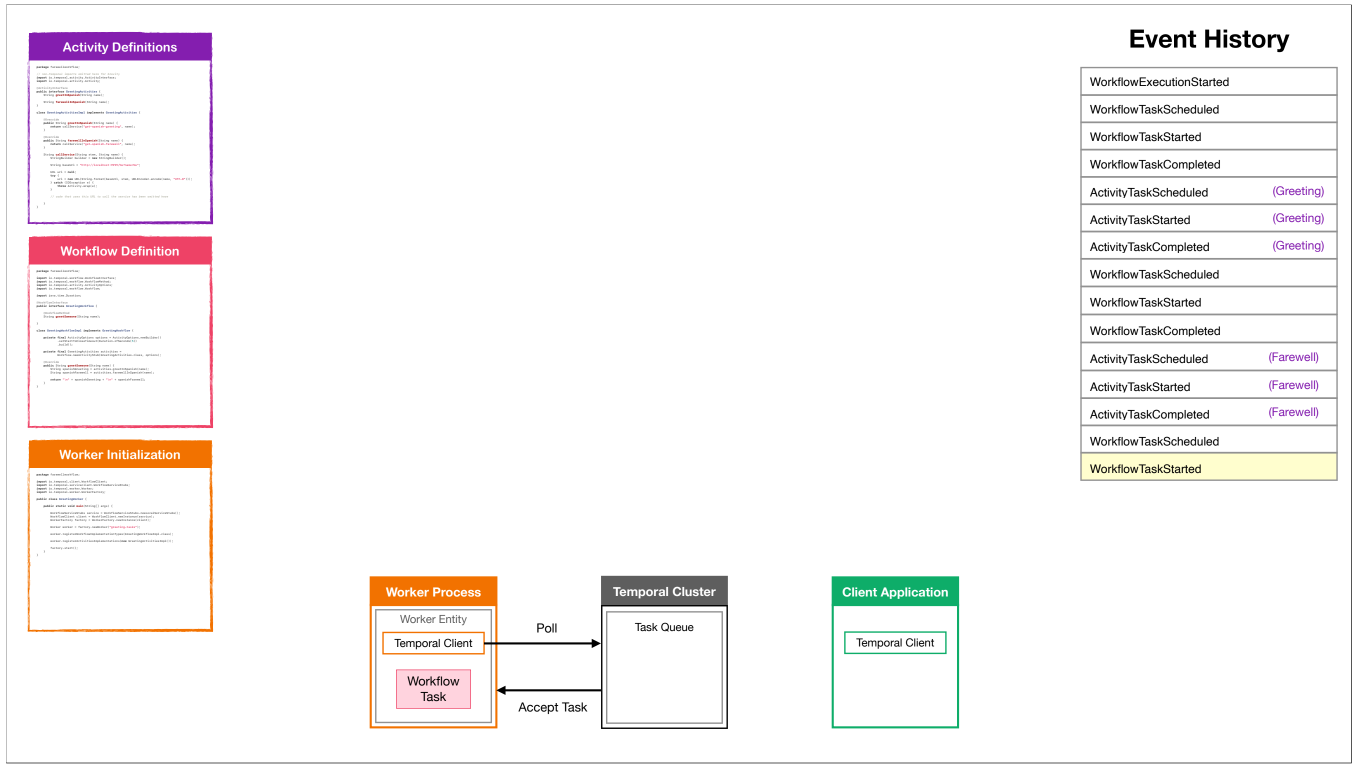
--



Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)
ActivityTaskCompleted	(Farewell)
WorkflowTaskScheduled	

There are still a few lines of the Workflow code that haven't been run yet, so the Temporal Cluster adds a new Workflow Task to the queue.
--

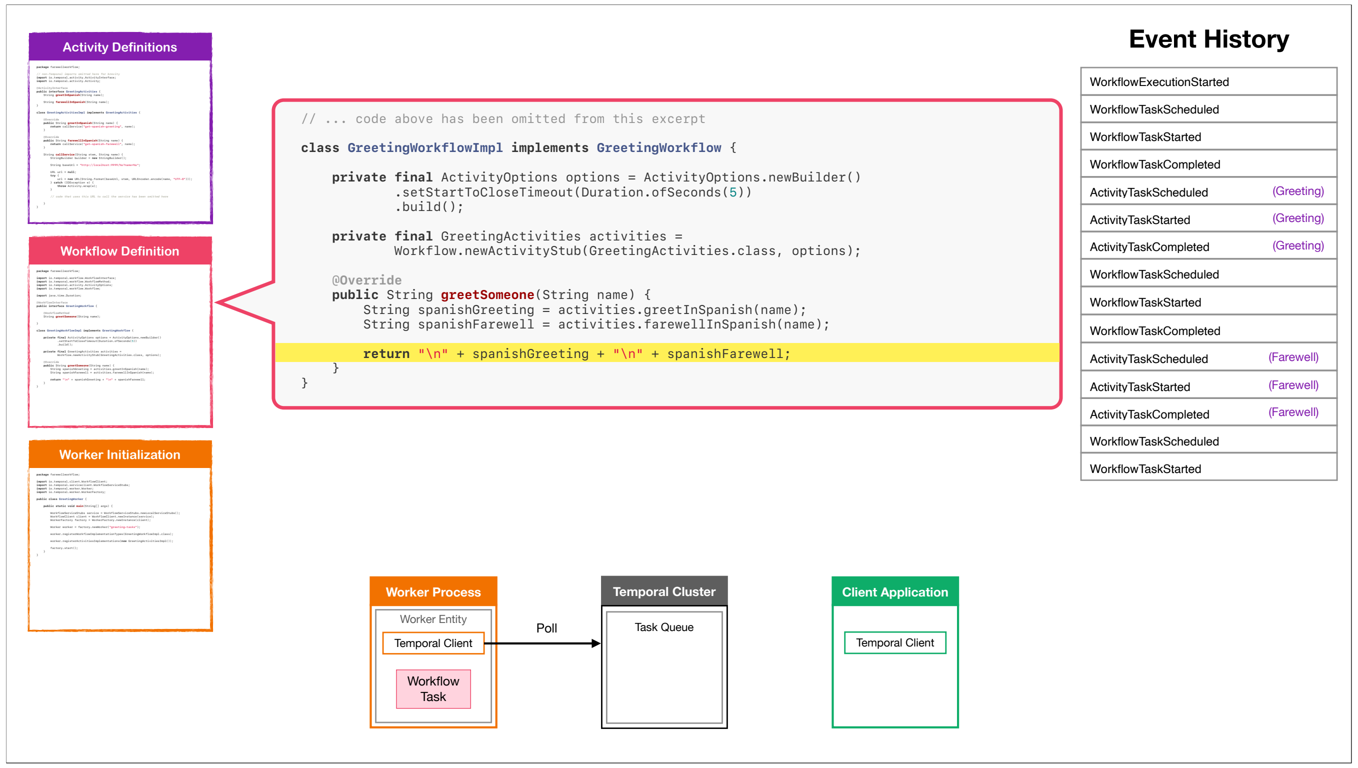


Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)
ActivityTaskCompleted	(Farewell)
WorkflowTaskScheduled	
WorkflowTaskStarted	

When the Worker accepts this new Task, the Temporal Cluster adds a WorkflowTaskStarted Event to the history.

--



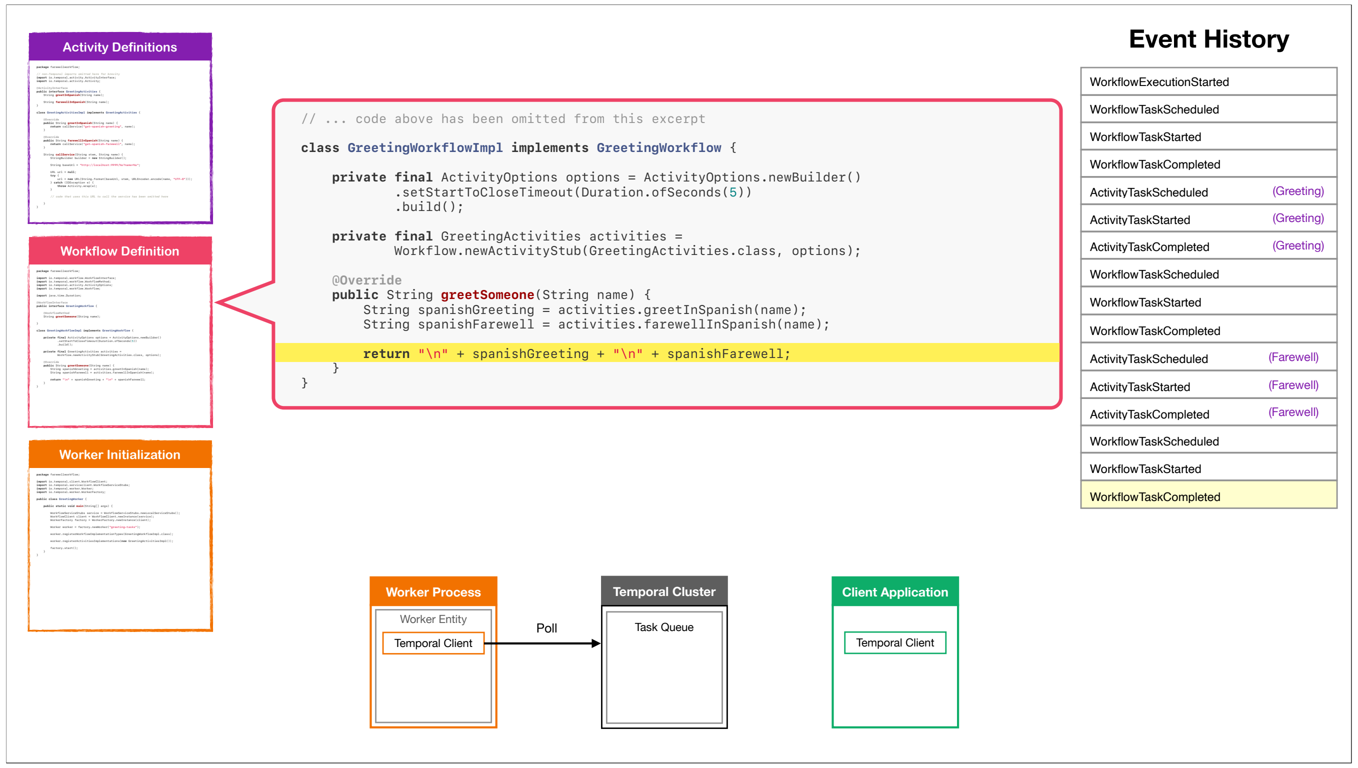
```
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)
ActivityTaskCompleted	(Farewell)
WorkflowTaskScheduled	
WorkflowTaskStarted	

The Worker continues where it left off, executing the remaining statements in the Workflow Definition.

--



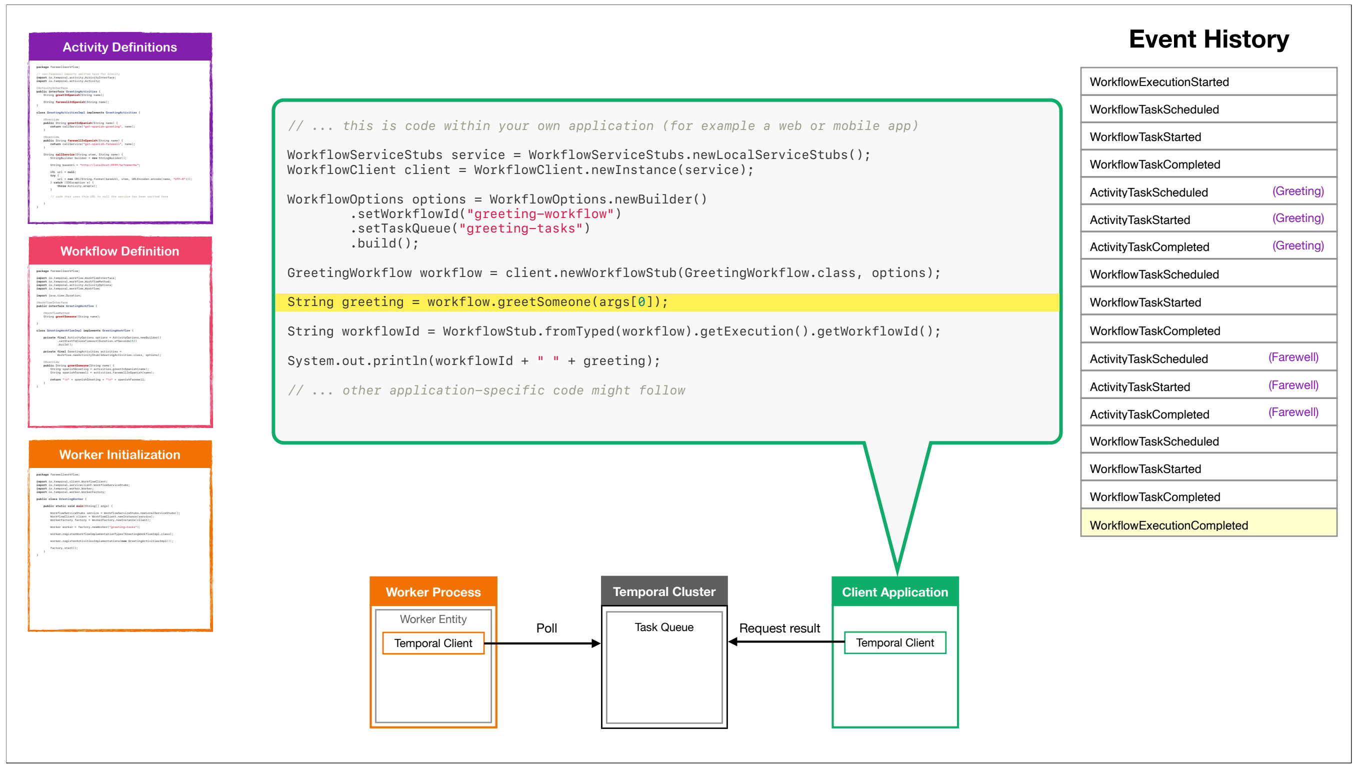
```
// ... code above has been omitted from this excerpt

class GreetingWorkflowImpl implements GreetingWorkflow {

    private final ActivityOptions options = ActivityOptions.newBuilder()
            .setStartToCloseTimeout(Duration.ofSeconds(5))
            .build();

    private final GreetingActivities activities =
            Workflow.newActivityStub(GreetingActivities.class, options);

    @Override
    public String greetSomeone(String name) {
        String spanishGreeting = activities.greetInSpanish(name);
        String spanishFarewell = activities.farewellInSpanish(name);

        return "\n" + spanishGreeting + "\n" + spanishFarewell;
    }
}
```

Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)
ActivityTaskCompleted	(Farewell)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	

Once this function returns, the Workflow Task is complete.

--



```
// ... this is code within your own application (for example a web or mobile app)

WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
WorkflowClient client = WorkflowClient.newInstance(service);

WorkflowOptions options = WorkflowOptions.newBuilder()
        .setWorkflowId("greeting-workflow")
        .setTaskQueue("greeting-tasks")
        .build();

GreetingWorkflow workflow = client.newWorkflowStub(GreetingWorkflow.class, options);

String greeting = workflow.greetSomeone(args[0]);

String workflowId = WorkflowStub.fromTyped(workflow).getExecution().getWorkflowId();

System.out.println(workflowId + " " + greeting);

// ... other application-specific code might follow
```

Event History

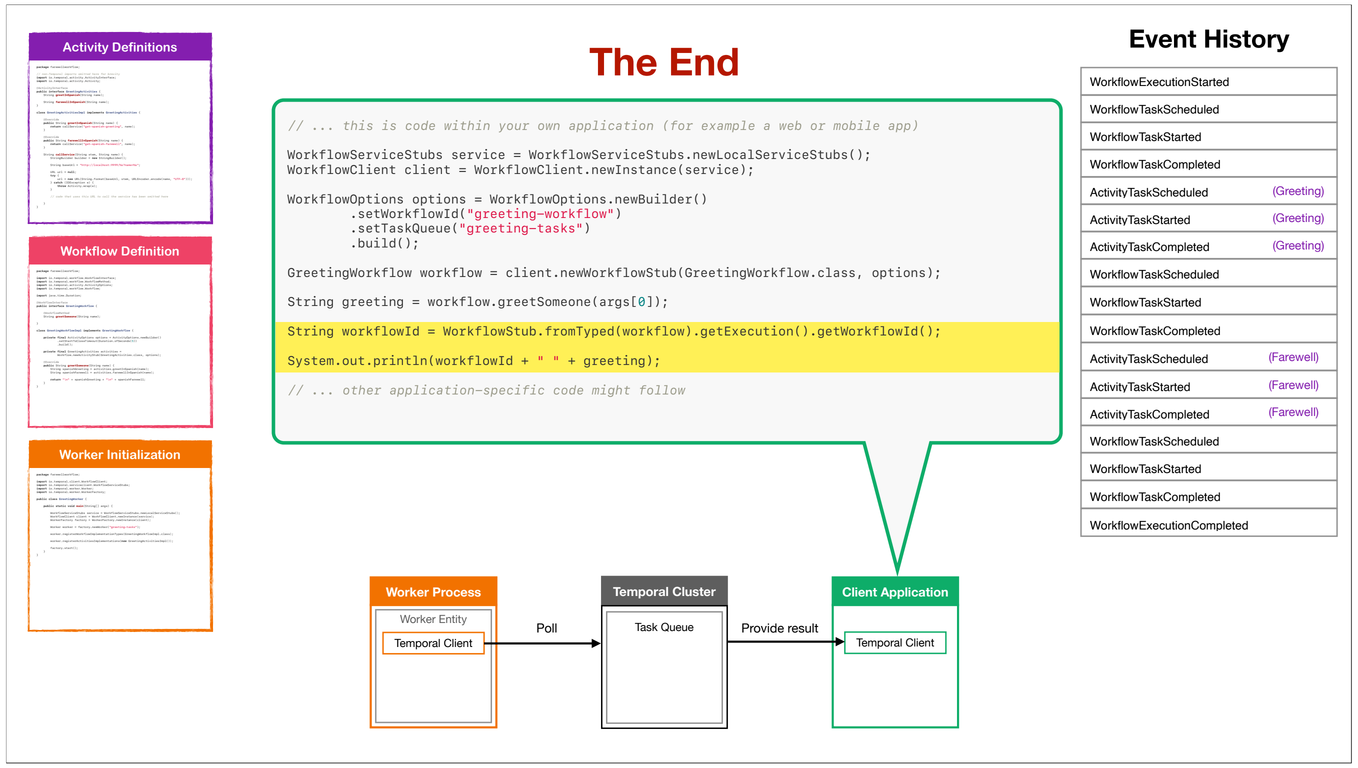
WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)
ActivityTaskCompleted	(Farewell)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
WorkflowExecutionCompleted	

Since the Workflow function returned, Workflow Execution is now complete, and the Cluster adds the final event to its history.

The Worker continues polling for new Tasks, but there is no more work related to *this* Workflow Execution.

The client application, which has been awaiting the result of the Workflow Execution because it's blocked on the Get call, will now receive that value.

--



The End

```
// ... this is code within your own application (for example a web or mobile app)

WorkflowServiceStubs service = WorkflowServiceStubs.newLocalServiceStubs();
WorkflowClient client = WorkflowClient.newInstance(service);

WorkflowOptions options = WorkflowOptions.newBuilder()
        .setWorkflowId("greeting-workflow")
        .setTaskQueue("greeting-tasks")
        .build();

GreetingWorkflow workflow = client.newWorkflowStub(GreetingWorkflow.class, options);

String greeting = workflow.greetSomeone(args[0]);

String workflowId = WorkflowStub.fromTyped(workflow).getExecution().getWorkflowId();

System.out.println(workflowId + " " + greeting);

// ... other application-specific code might follow
```

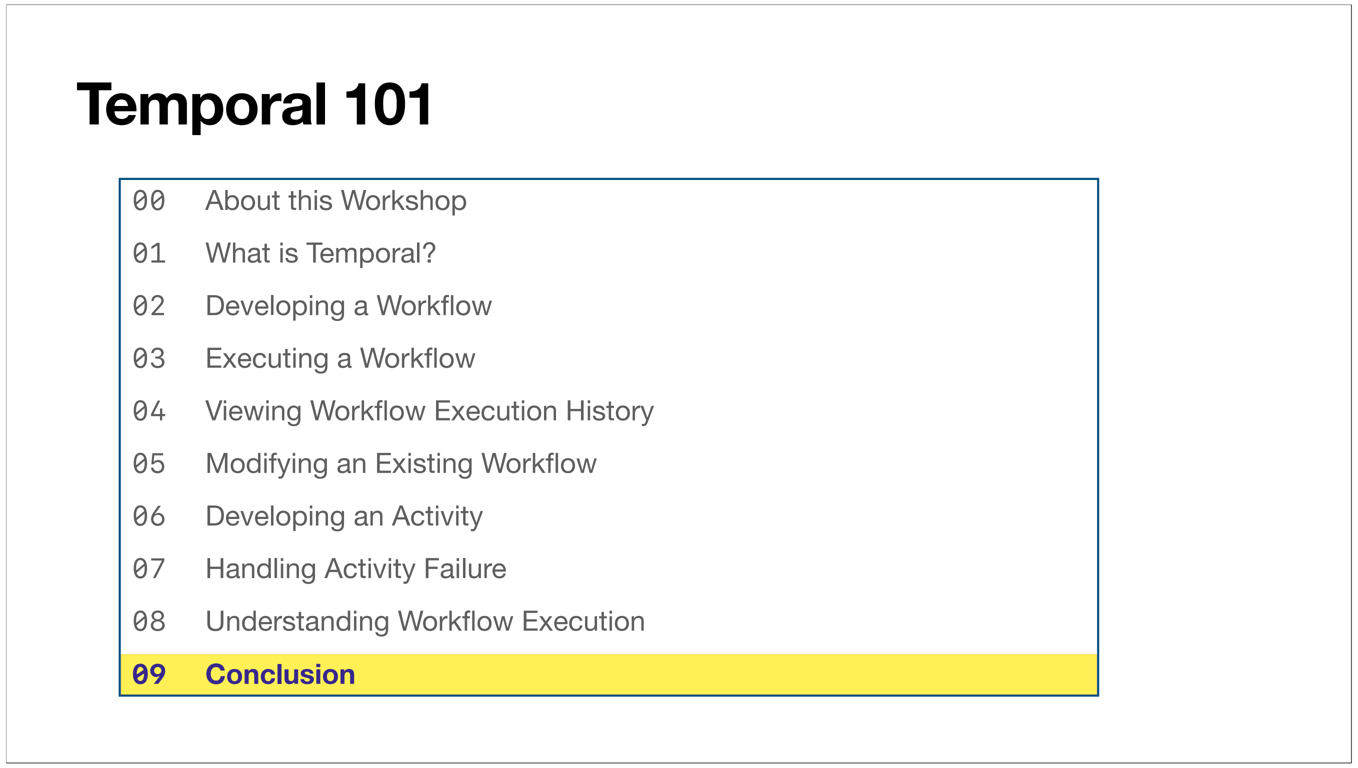
Event History

WorkflowExecutionStarted	
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Greeting)
ActivityTaskStarted	(Greeting)
ActivityTaskCompleted	(Greeting)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
ActivityTaskScheduled	(Farewell)
ActivityTaskStarted	(Farewell)
ActivityTaskCompleted	(Farewell)
WorkflowTaskScheduled	
WorkflowTaskStarted	
WorkflowTaskCompleted	
WorkflowExecutionCompleted	

The cluster provides the result to the application, which can process it however it wishes.

And now you've seen what happens during a Workflow Execution.

--



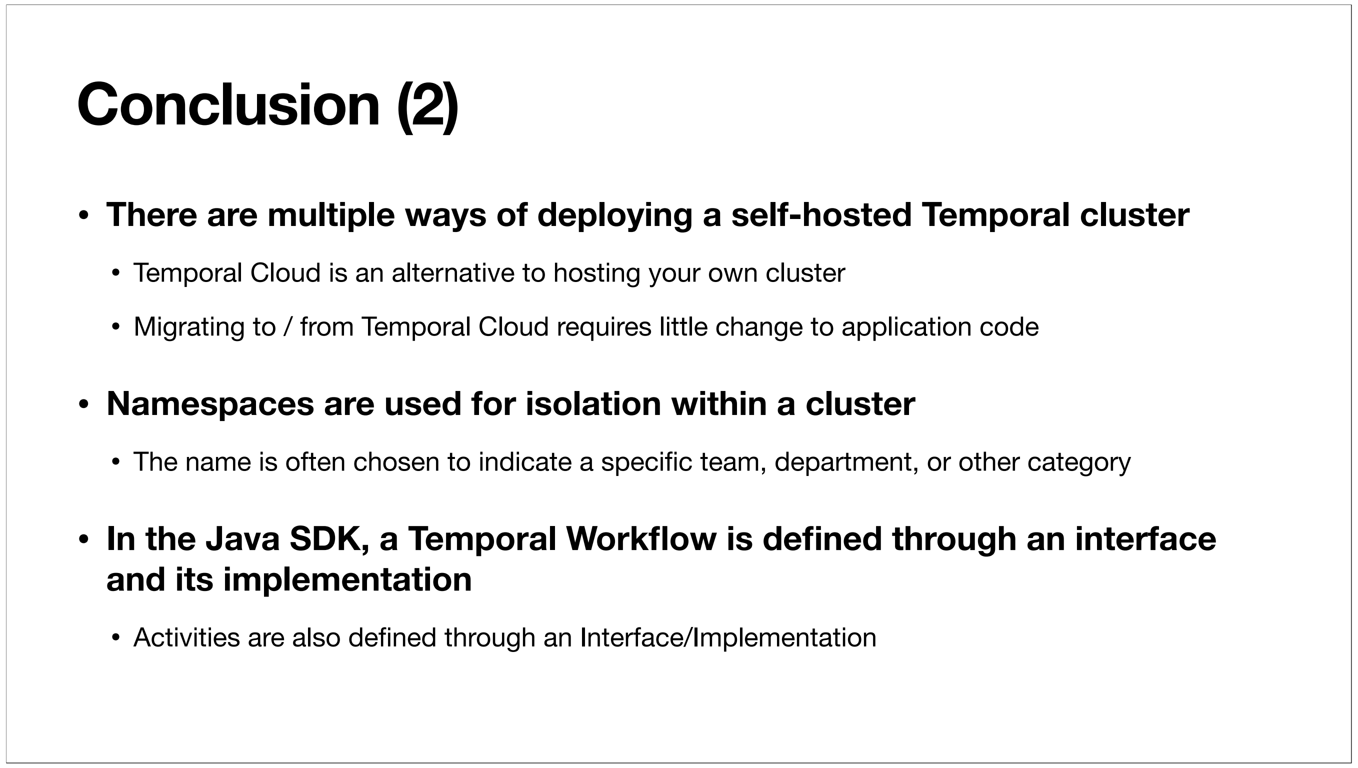
Temporal 101

--

Conclusion (1)

- **Temporal guarantees the durable execution of your applications**
  - In Temporal, Workflows are defined through code (using a Temporal SDK)
- **Temporal Clusters orchestrate code execution**
  - Workers are responsible for actually executing the code
- **The Temporal Cluster maintains dynamically-created task queues**
  - Workers continuously poll a task queue and accept tasks if they have spare capacity
  - You can increase application scalability by adding more Workers
  - You must restart Workers after deploying a code change

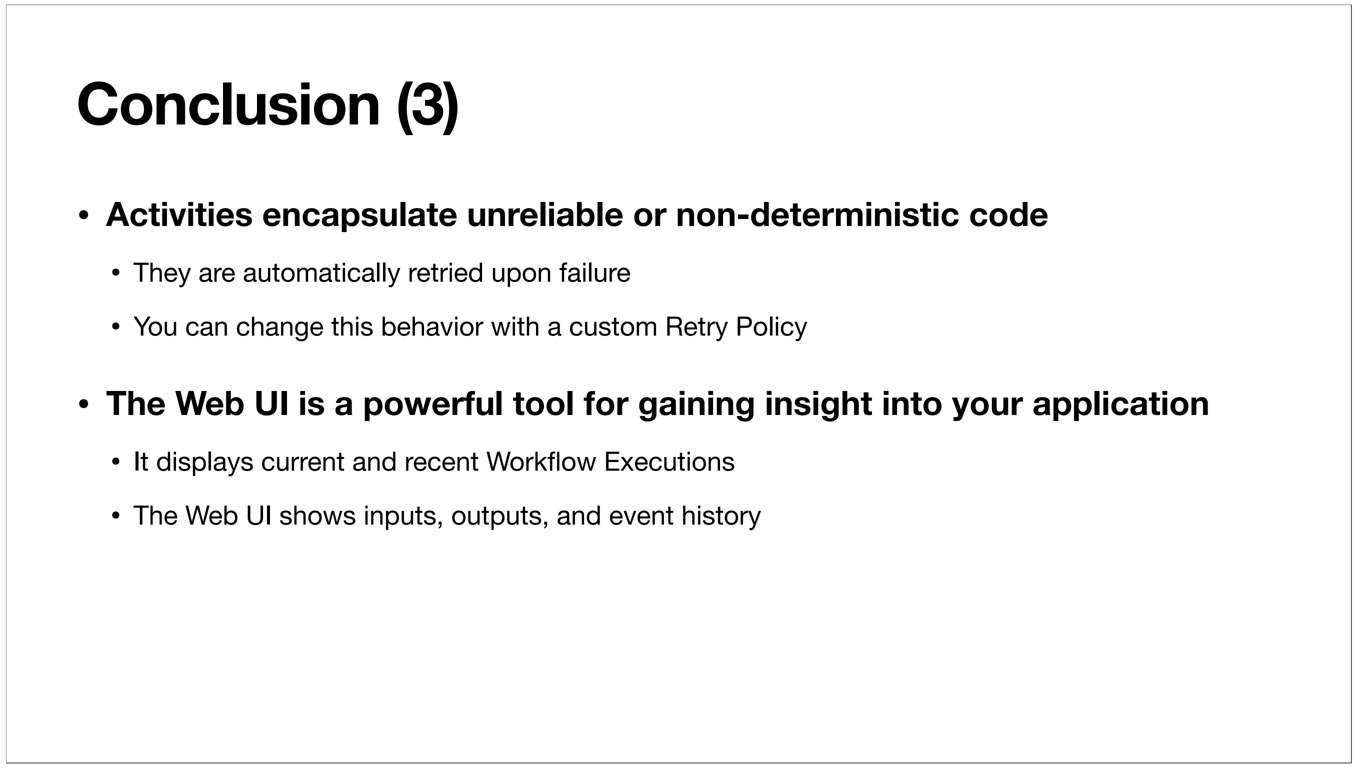
--



Conclusion (2)

- **There are multiple ways of deploying a self-hosted Temporal cluster**
  - Temporal Cloud is an alternative to hosting your own cluster
  - Migrating to / from Temporal Cloud requires little change to application code
- **Namespaces are used for isolation within a cluster**
  - The name is often chosen to indicate a specific team, department, or other category
- **In the Java SDK, a Temporal Workflow is defined through an interface and its implementation**
  - Activities are also defined through an Interface/Implementation

--



Conclusion (3)

- **Activities encapsulate unreliable or non-deterministic code**
  - They are automatically retried upon failure
  - You can change this behavior with a custom Retry Policy
- **The Web UI is a powerful tool for gaining insight into your application**
  - It displays current and recent Workflow Executions
  - The Web UI shows inputs, outputs, and event history

--

Exercise #4: Finale Workflow

- **During this exercise, you will**
  - Observe that a Workflow and its Activities can be implemented in different languages
    - This example provides a Java Activity and a Go Workflow for you to run
- **Refer to the README.md file in the exercise environment for details**
  - The code is below the **`exercises/finale-workflow`** directory

NOTE: Allow 5 minutes for this exercise. Also mention that they won't modify any of the code for this exercise (thus, there is no "practice" or solution directory). They just run the specified commands and observe the output from a Workflow.

If time is tight, attendees can do this after the session as homework.

Thank You

--
NOTE: This is the end of the presentation. The remaining slides are here for the benefit of the presenter and/or curriculum developer.