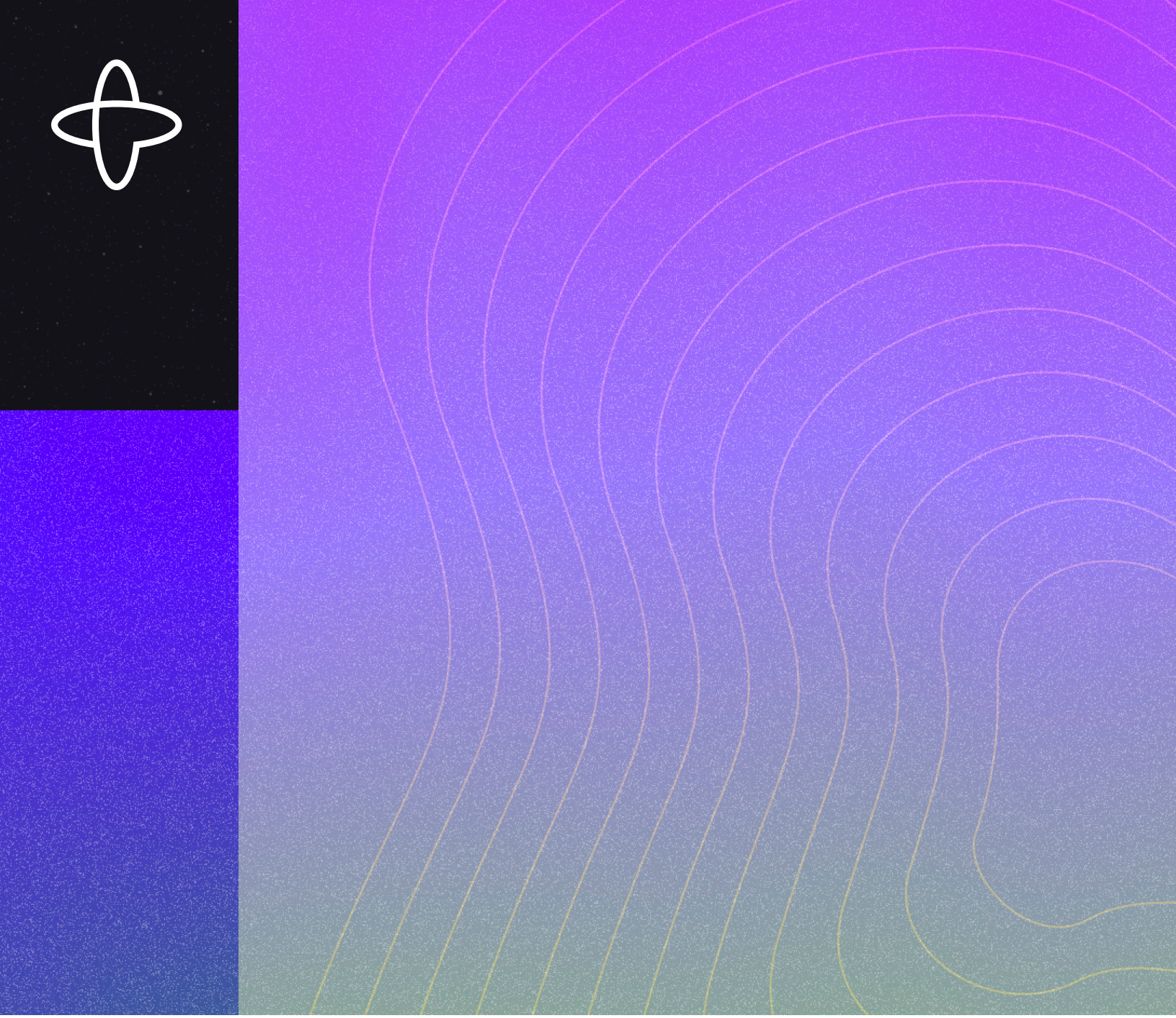


STATE MACHINES SIMPLIFIED

Reducing the complexity of state machines with Temporal

A state machine is a software design pattern used to modify a system's behavior in response to changes in its state. While state machines are widely used in software development, applying them to complex business processes can be a difficult undertaking.

The alternative to a state machine is to use an orchestration tool specifically intended for business workflows. One such tool is Temporal, which is the focus of this article.

In this article, we'll present a brief introduction to state machines and show how the design pattern can be used to implement a business process. Then, we'll compare a process implemented as a state machine to a process implemented using Temporal.

We provide Java code that demonstrates both implementation approaches described in this article. You'll find the code in this GitHub repository, which provides a Maven project for the State Machine for the demonstration process and another Maven project that implements the same process with Temporal Java SDK

In order to get full benefit from reading this article, The more you know about Temporal – from Workflows, and Activities, to the Workers and Clients – the more you'll get from this article.

State machines: an overview and an example

A state machine is a software design pattern used to build applications that move between many possible states. The role of a state machine is to manage state changes and make the current state of the system explicit.

Figure 1 below illustrates a state machine for a system that supports a document publishing workflow. This use case is implemented in the demonstration code that accompanies this article. At the conceptual level, the workflow accepts a document and then copy edits and graphic edits the document simultaneously. Once copy editing and graphic editing completes, the document is published.

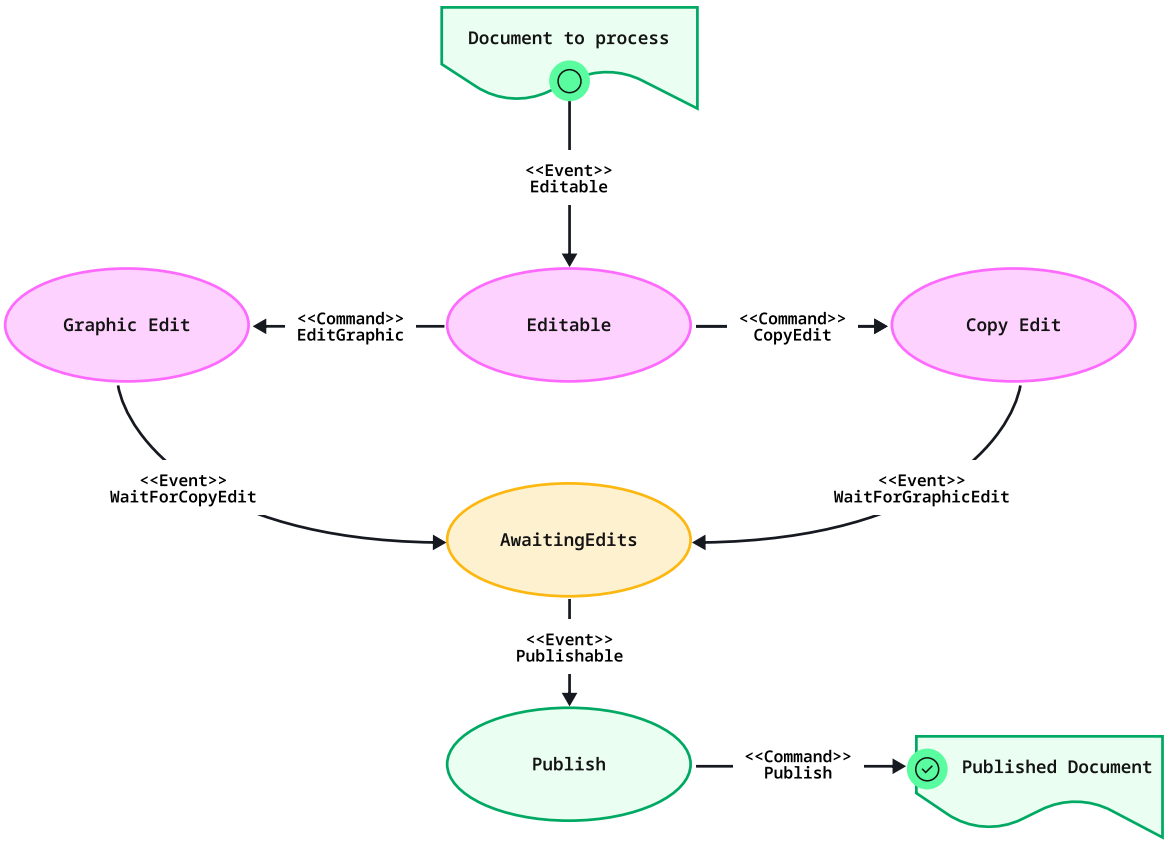


Figure 1: A state machine for a document publishing workflow.

When a document is submitted to the workflow, a controller component, which is responsible for changing the state of the system, triggers an `Editable` event. How the event is triggered depends on the nature of the controller. The event could be triggered manually by a user by clicking a “Submit” button on a web page. Or the event could be triggered automatically via a message queue working in conjunction with an application’s controller software. (Using a message queue is the approach taken in the implementation of the state machine in the demonstration code).

The controller receives the event and changes the current state of the system. For example in this article’s demonstration code, when the controller gets an `EVENT_EDITABLE` event, it will set the current state to `Editable` like so...

```
AbstractState.current = AbstractState.editable;
```

...then, the code in the class instance assigned to `AbstractState.editable` executes the transition rules for that state.

The thing to understand about the state reassignment statement shown above is that `AbstractState` is an abstract class that defines a set of public variables that describe each state possible in the demonstration use case. Each state variable is of type `AbstractState` too as shown in Listing 1 below.

```
package pubstatemachine.state;

import pubstatemachine.message.AbstractMessage;
import pubstatemachine.queue.SimpleMessageQueue;

public abstract class AbstractState {
  // add a constructor
  public AbstractState(SimpleMessageQueue queue) {
    AbstractState.queue = queue;
  }

  static SimpleMessageQueue queue;
  public static AbstractState inactive;
  public static AbstractState editable;
  public static AbstractState graphicEdit;
  public static AbstractState copyEdit;
  public static AbstractState awaitingEdits;
  public static AbstractState awaitingPublish;
  public static AbstractState publish;
  public static AbstractState current;

  public void enter() {}

  public void update(AbstractMessage message) throws InterruptedException {}
}
```

Listing 1: The abstract class named AbstractState defines the various states of the demonstration use case

Later on logic in the `Controller` class assigns each of the public variables an actual implementation of a class that inherits from `AbstractState`. The example below shows an excerpt from the `Controller` code that assigns an instance of the `Editable` class to the member variable `AbstractState.editable`.

```
AbstractState.editable = new Editable(queue);
```

As mentioned above, the class named `Editable` inherits from `AbstractState` and provides the required behavior for the `enter()` and `update(AbstractMessage message)` methods as defined in `AbstractState`. Thus, the statement, `AbstractState.current = AbstractState.editable` changes the current state of the application to the behavior defined by the class named `Editable`. Logic in the `Controller` then calls the `enter()` and `update()` methods of that current state.

In the case shown in Figure 1, at the start of the workflow the current state is transitioned to `Editable`. Part of the transition behavior in the `Editable` state is to issue two commands, `EditGraphic` and `CopyEdit`. Eventually, this puts the system in a current state of `AwaitingEdits`. As the name implies, the state of `AwaitingEdits`, waits to receive two events, `WaitforCopyEdit` and `WaitForGraphicEdit`. These two events indicate that both editing tasks have been completed. `WaitforCopyEdit` and `WaitForGraphicEdit` can occur in any order and thus represents a rule composed of multiple conditions that must be satisfied in order for the workflow to progress. (It's worth noting that the work required to support this composed rule in this demonstration use case is trivial. However, in a real-world production scenario, supporting a composed rule can be a daunting undertaking).

Once the `AwaitingEdits` state receives `WaitforCopyEdit` and `WaitForGraphicEdit` events, it **emits a** `Publishable` command that transitions the system into a current state of `Publish`. From there, the command `Publish` is executed and the process is complete.

Listing 1 below shows the code excerpt from the `Controller` class of the demonstration project. Event emission and command execution are facilitated by messages received from a message queue. Those messages are processed by a **switch statement in the state machines Controller** class as shown in Listing 2:

```
while (true) {
    try {
        AbstractMessage msg = queue.getMessage();
        System.out.println("Controller received message: " + msg.getMessageType());
        switch (msg.getMessageType()) {
          case EVENT_EDITABLE:
            processEventEditable(msg);
            break;
          case EVENT_AWAIT_GRAPHIC_EDIT:
          case EVENT_AWAIT_COPY_EDIT:
            processEventAwaitEdits(msg);
            break;
          case EVENT_PUBLISHABLE:
            processEventPublishable(msg);
            break;
          case EVENT_PUBLISHED:
            System.out.println("Document published");
            break;
          case COMMAND_GRAPHIC_EDIT:
            processCommandGraphicEdit(msg);
            break;
          case COMMAND_COPY_EDIT:
            processCommandCopyEdit(msg);
            break;
          case COMMAND_PUBLISH:
            processCommandPublish(msg);
            break;
          default:
            throw new IllegalStateException("Unexpected value: " +
                   msg.getMessageType());
        }
    } catch (InterruptedException e) {
        System.err.println("Polling interrupted: " + e.getMessage());
        break;
    }
  }
});
```

Listing 2: The logic in the controller class of the state machine demonstration code that processes messages from a message queue.

When a message is received, the `switch` statement invokes a processing method within the code according to the message. For example, when the `switch` statement receives an `EVENT_EDITABLE` messageType, it invokes the method `processEventEditable(msg)` in which the `msg` parameter has two properties, `msg.messageType` and `msg.document`. The `processEventEditable(msg)` code will extract the document from the msg parameter and execute its processing logic on that document. When a message `COMMAND_GRAPHIC_EDIT` is received, the switch statement calls the processCommandGraphicEdit(msg) method.

An important point to keep in mind is that the switch statement is only routing messages to processing methods. It is not managing the sequence in which state changes occur. State change order is governed by logic distributed among the various transitions within the state machine. In other words, there is no central place in the code that says, “go from the Editable state to the CopyEdit and GraphicEdit states and then onto the Publish state.” Sequence management is conducted implicitly via message emission dictated by a given transition rule. For simple event-driven state machines this is manageable, but as you’ll see, when state machines manage many sequential state changes, their complexity increases.

The challenges of state machines

A significant concern with state machines is that they are often complex to program and difficult to maintain, particularly when it comes implementing a business workflow that has many composable rules. For instance, the `AwaitEdits` state in the document publishing use case waits for both `WaitforCopyEdit` and `WaitForGraphicEdit` events in order for the workflow to move forward. This is an example of a composable rule. Supporting a limited number of composable rules within a state machine is manageable. However, as more composable rules are added to a state machine, the complexity of the machine increases, as does the effort required to maintain and upgrade it. State machines that support consensus require additional complexity.

Another complex task when building a state machine is maintaining the order of actions in a sequential business process. In other words, making sure that Step 1 is followed by Step 2 and then by Step 3, etc. The way the demonstration application ensures that the order of steps in the document publication process is to introduce a class named **StateMonitor**. The `StateMonitor` class keeps track of all the state transformations that have occurred in the application, and makes it so a particular command associated with a particular state executes in the expected order. Listing 3 below shows an excerpt of the **update()** method for the `AwaitingEdits` class. As the name implies, the `AwaitingEdit` class represents the state in which the document publication process is waiting for edits to complete. The `update()` method is the associated transition.

```
StateMonitor sm = StateMonitor.getStateMonitor(message.getDocument());
.
.
.

if (sm.isGraphicEdited() && sm.isCopyEdited() && !sm.isPublishable()) {
      queue.putMessage(new MessageImpl(MessageType.COMMAND_PUBLISH, document));
      sm.setPublishable(true);
}
```

Listing 3: The state machine demonstration used a class named StateMonitor to keep track of the various states that the document publication process has passed through.

The transition rule emits a `COMMAND_PUBLISH` event, which tells the system to publish the document, when the document has moved through `GraphicEdit` and `CopyEdit` states.

An important thing to remember about a state machine in general, and a message-driven state machine in particular, is that unless a sequence management mechanism like a `StateMonitor` is put in place, state transformations might occur in an arbitrary manner. The effort required to manage the sequence of events in a business process that has one, two or event three steps is minimal. However, ensuring order in a business process with dozens, if not hundreds, of steps can require a herculean programming effort—not to mention code maintenance.

Another challenge with state machines is maintaining system state in the event of a failure. Failures are commonplace, even with the simplest of business processes. Some failures, such as timeout, can be remedied by a retry. Others require the system to be reverted to its last known good state. Addressing failure in a State Machine is difficult; in fact, it can be just as much, if not more work as creating the State Machine's happy path.

When all is said and done, using a state machine for an application that has a limited number of states and has infrequent changes makes sense (like computer games or applications that run a streaming service on your television). But, for applications that can have a large number of states and are subject to rapidly changing requirements, there are better alternatives to state machines. Temporal is one such alternative.

Temporal: a simpler way to manage state and transitions

Temporal is an open-source durable execution platform that abstracts away the complexity of building scalable distributed systems. Temporal is specifically intended for programming workflows, particularly complicated workflows. As such, it has many of the characteristics of a state machine, yet avoids a good deal of the work that goes with programming a state machine. Additionally, because it provides durability, Temporal ensures state is always saved and all processes complete, even in the event of a failure.

At the conceptual level, Temporal includes some of the parts typically found in a state machine. These parts are built into the framework. A Temporal Workflow is an essential primitive of the framework that can be thought of as a controller. A Temporal Workflow manages the sequence and execution of activities. Temporal Activities can be thought of as command handlers.

Figure 2 below illustrates the document publication process previously implemented as a state machine, this time implemented as a Temporal Workflow.

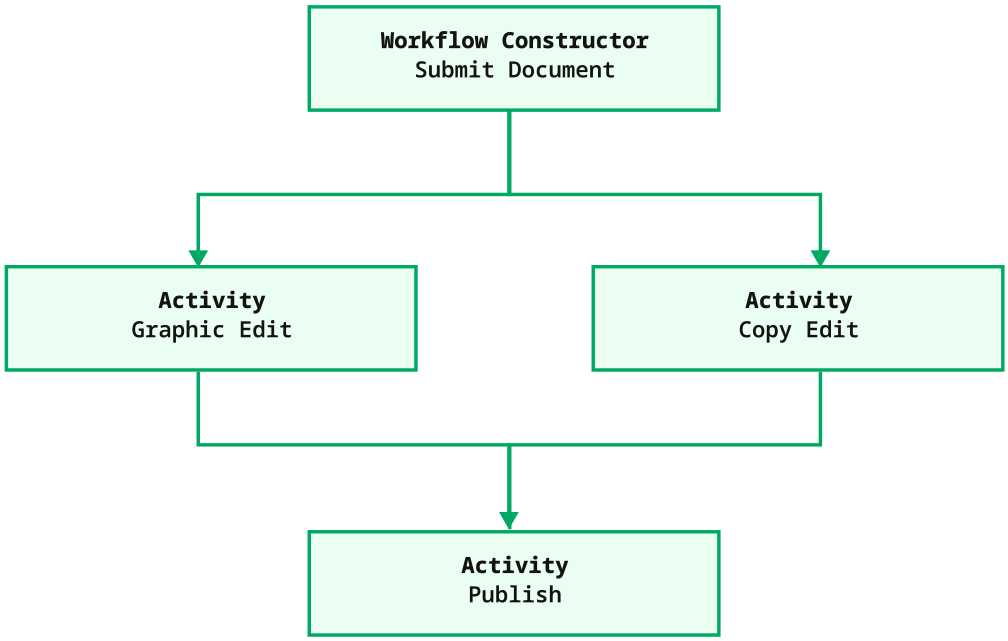


Figure 2: Implementing the documentation publication process as a Temporal Workflow.

As with the state machine example mentioned previously, a document is submitted to a workflow for processing. Then, the document is graphic edited and copy edited simultaneously. The document is published after editing completes. The high level behavior of both the state machine and Temporal Workflow is similar. However, the effort that goes into implementing the Temporal Workflow is apparent and straightforward, and requires significantly less time. As shown below in Listing 4, all the code that manages the Workflow is in one place.

```
public void startWorkflow(Document document) {
  logger.info("Starting workflow for publishing document: " + document.getUrl());
  try {
// Make it so that the copyEdit() and graphicEdit() activities are executed in parallel ...
    List<Promise<Void>> promisesList = new ArrayList<>();
    promisesList.add(Async.procedure(activities::copyEdit, document));
    promisesList.add(Async.procedure(activities::graphicEdit, document));

    Promise.allOf(promisesList).get();

    // ...then execute the publish() activity
    Promise<Void> publishPromise = Async.procedure(activities::publish, document);
    publishPromise.get();
    logger.info("Publishing complete for document: " + document.getUrl());

    } catch (ActivityFailure e) {
      throw e;
    }
  }
}
```

Listing 4: A Temporal workflow for the document publishing use case

The interesting thing to notice about the Workflow code in Listing 4 above is that the editing and publishing work is encapsulated within Temporal Activities as shown at Lines 7, 8 and 13. Temporal Activities are the fundamental, atomic 'steps' that act as the 'doers' in a Temporal Workflow. This implementation demonstrates that they may be executed concurrently using Promises while the Workflow blocks to await the results of both steps.

Separating Activity implementation from the Workflow makes the application easier to understand and easier to manage. Should a developer need to add a new Workflow Activity such as `LegalReview`, they can just add a method for the Activity to the file in which the Temporal Activities are defined and then add that Activity method to the Temporal Workflow file according to its position in the Workflow Execution.

By contrast, implementing a new Activity within the document publication state machine would require a lot more work.

Why choose Temporal

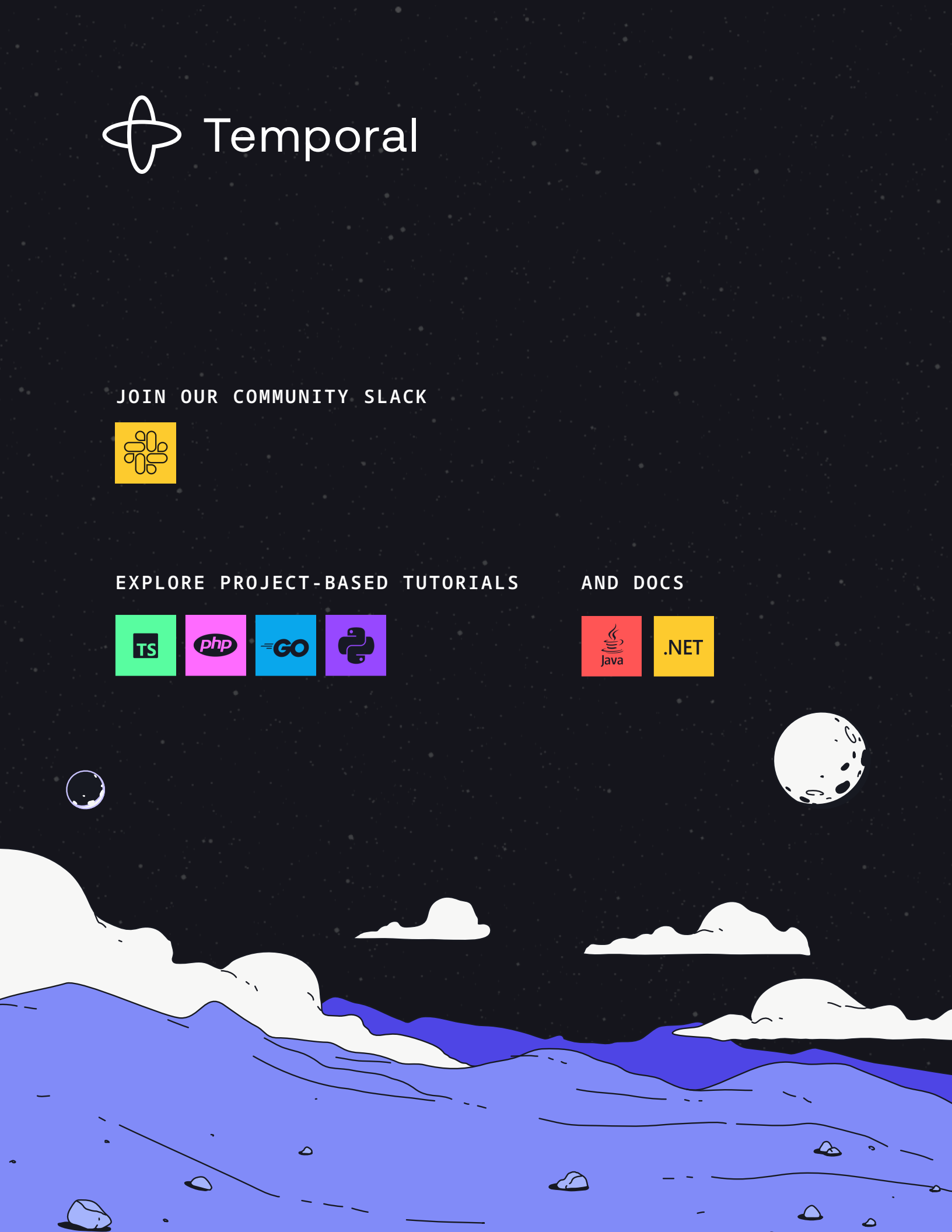
With Temporal, you don't need to develop the controller, events, transitions, or commands that make up the state machine. That functionality is built into Temporal. Developers program the outcome of Temporal Workflows: describe the sequence of Activities in the Workflow, and program the behavior for each Activity executed in the Workflow. These actions are straightforward to accomplish because Temporal's programming model is well defined with its Workflow and Activity primitives.

Additional benefits of Temporal compared to a state machine are:

- **Durability.** Temporal's durability is one of its strongest values. Temporal durably stores any progress that has occurred in a Workflow. Regardless of faults in a system, progress in Temporal will always be recovered, and therefore the business state is always consistent.
- **Configurable retries.** Retry logic takes tremendous time to implement in state machines. With Temporal, implementing retries is a matter of a configuration setting that Temporal will execute automatically.
- **Easy compensations.** In a state machine, even under the best case when a state machine is well structured, compensation behavior would need to be programmed into each state. Implementing compensation behavior in a loosely structured state machine can result in spaghetti code that's hard to program and test. In Temporal, implementing system reversions is a straightforward undertaking. Compensation code is programmed in one place: within the Workflow file.
- **Message queues.** If a state machine is message-driven, then the developer needs to implement the message queue(s) and the messages that the queue will support too. This can be significant work, particularly for distributed applications that operate at web scale. In Temporal, there's no need to create a message queue from scratch because it provides a user-accessible equivalent called a Task Queue that automatically queues up tasks.
- **Built-in messaging primitives that interact with applications.** Temporal's built-in messaging primitives for interacting with applications remove yet another infrastructure task developers must undertake to apply transitions to their application state.

In conclusion

Temporal's essential value proposition is that the framework allows developers to focus on what they do best: creating effective, efficient Workflows that meet the needs of the business. When it comes to implementing complex business processes, Temporal provides a structured and straightforward approach to Workflow development that is hard for a state machine to emulate.


Temporal
JOIN OUR COMMUNITY SLACK
EXPLORE PROJECT-BASED TUTORIALS
AND DOCS
TS
php
GO
Java
.NET