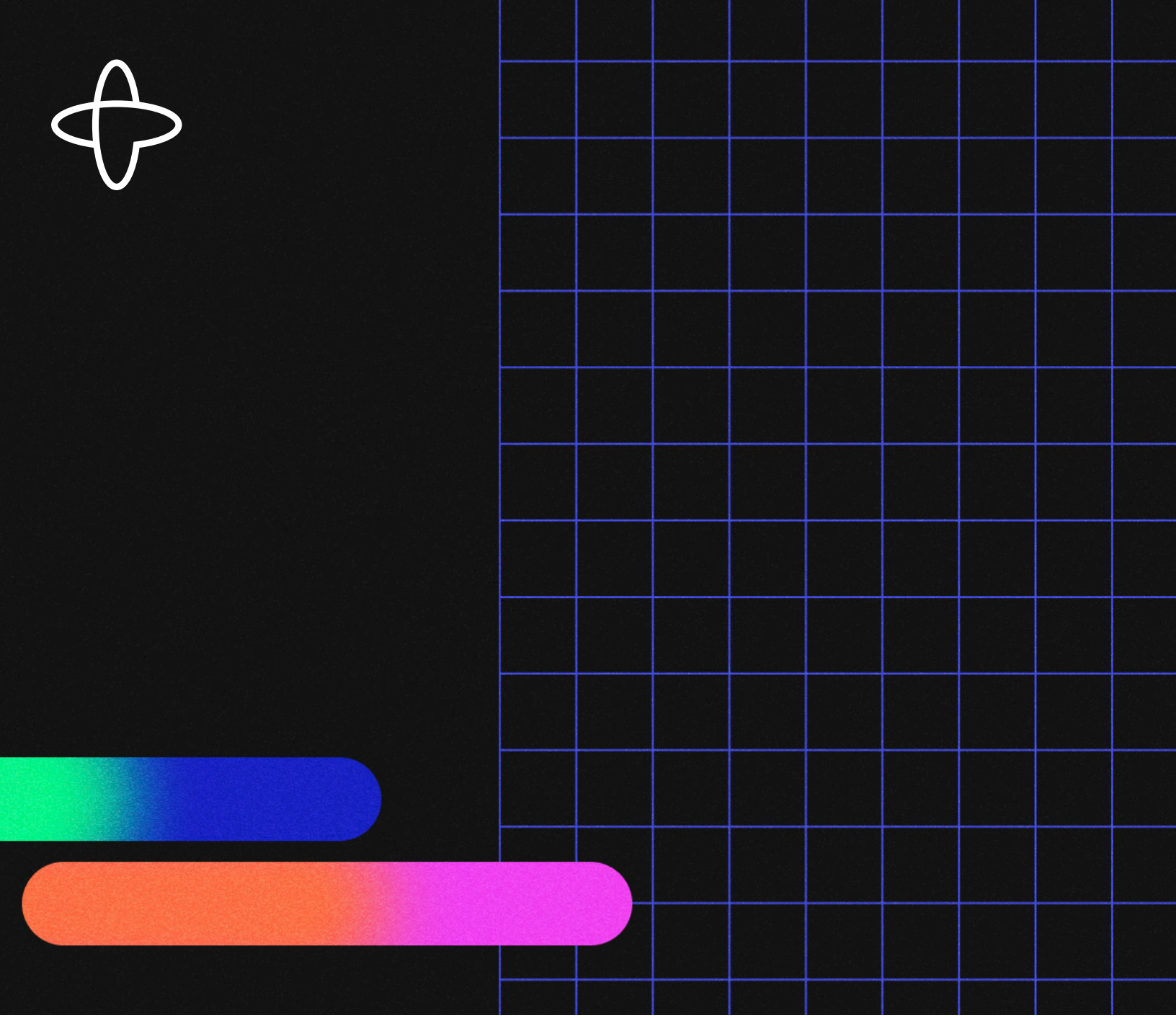


THE SAGA PATTERN MADE EASY

Automating the Saga Pattern with Temporal

Multi-step business transactions are hard to handle. They're much more complex than accommodating issues in a database transaction, where rollback is a matter of resetting tables defined within the database server. If any step in the transaction fails to complete, the entire state of the transaction, both in terms of data and execution logic, becomes invalid. Its consistency is compromised, and the system under which the business transaction is executed needs to be reverted to its last known good state. Resetting state can be a laborious, complex undertaking full of risk.

Fortunately, a number of architectural design patterns have emerged that make programming multi-step transactions easier. One solution that continues to grow in popularity is the Saga pattern, which is the focus of this article. The Saga pattern provides a blueprint for ensuring state consistency in complex business transactions. Also, the pattern is easy to apply when creating durable Workflows using Temporal. In fact, Temporal reduces a lot of the work required to execute a comprehensive implementation of the Saga pattern.

In this article, we're going to describe what the Saga pattern is, how it works, and how it can be easily applied to Workflows running under the Temporal framework. In order to get full benefit from reading this article, we recommend that you have a working understanding of the primitives of Temporal: Workflows, and the Temporal Server, Worker and Client. Having some previous knowledge about Saga Patterns may also be helpful.

Let's start by explaining what the Saga pattern is.

What is the Saga Pattern?

The Saga pattern is an architectural design pattern that's intended to ensure state consistency in multi-step and distributed business transactions. In order to understand the importance of ensuring state consistency, imagine the four step business transaction shown below in Figure 1:

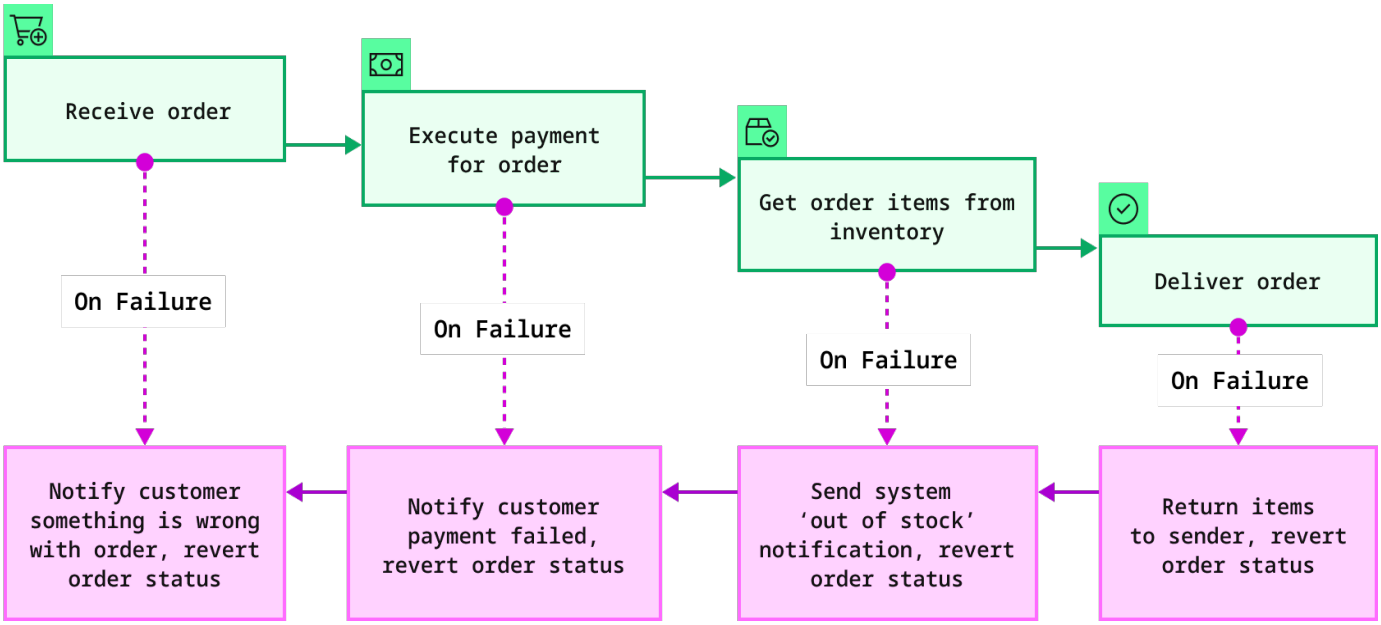


Figure 1: Reverting to last known good state is a core principle of the Saga pattern (adapted from a graphic in Baeldung's Saga Pattern in Microservice)

If any step in the transaction fails, the state of the system becomes inconsistent. For example, should step 3 in the business transaction fail because the requested order item is not available in inventory, a consistency problem occurs because the customer will have paid for an item they won't receive. Furthermore, the customer will have no idea that something is wrong with the order. The entire transaction is in a bad state.

In order for the system to regain consistency in terms of failure at Step 3, it must be reverted to the last known good state by refunding the customer's payment and notifying the customer that something is wrong with the order as indicated by the red rectangles shown in Figure 1.

Under the Saga pattern, reversion behavior for a particular step in a business transaction is called a ***compensation***.

Compensations are an essential, required component of the Saga pattern. They need to be idempotent, which means they can be executed multiple times without causing side effects to the system or to the steps in the transaction. No matter how many times you execute a compensation, it should achieve the same results. Idempotency is important because it's possible that a compensation may need to be retried many times, should it fail.

To sum it all up at the conceptual level, you can think of the Saga pattern as a design pattern for complex, multi-step business transactions that ensures application consistency by using compensations to revert the system to its last known good state when faced with failure at any point in the transaction. It's a simple concept, but as you'll see, implementing the Saga pattern requires some work.

Saga Pattern Implementation

Implementing the Saga pattern requires a mechanism to control the flow between the various steps in a business transaction. This mechanism is formally called the Saga Execution Coordinator (SEC). The SEC "runs" the Saga. It does three things.

- The SEC tracks the transient progress state from the results of the transaction steps.
- It determines the next valid step in the business transaction based on the progress in the state of the business transactions.
- And, optionally, it executes compensation steps in response to failure conditions within the business transaction.

Failures in an implementation of the Saga pattern could be exceptions, invalid values, or other results which should cause the transaction to be aborted, hence necessitating a return to the prior application state. Key to the SEC implementation is that it has knowledge of failure, and is programmed with the ability to address the given failure with a compensation.

In terms of the diagram shown above in Figure 1, it's the SEC that directs movement between the various steps in the business transactions. And, should the business transaction need to terminate due to failure, it's the SEC that will coordinate executing the compensations that will revert that application back to its last known good state.

The thing to keep in mind is that compensation logic needs to be programmed into the SEC. There is no magic. Creating a reliable compensation can be complex, but is just as important as programming the steps within a Saga, particularly when those steps are weakly consistent. Not only can Sagas be hard to build and get right, they can also be hard to test and maintain. The scope and complexity that goes with programming the Saga depends on the framework being used. Fortunately, Temporal alleviates a good deal of the labor involved.

Automating the Saga Pattern with Temporal

Temporal is an open-source durable execution platform that abstracts away the complexity of building scalable distributed systems. It autosaves application state for long-running and distributed processes, eliminating the need for complex failure and retry logic.

The essential benefit that Temporal provides for the Saga design pattern is that it automates most, if not all of the "heavy lifting" required for implementation. With Temporal, the Workflow is the Temporal primitive that actually implements the Saga. A Temporal Workflow is the equivalent of a SEC. All the overhead the Saga requires is easily minimized by the Temporal primitives.

The Temporal Server does the work of scheduling the steps in the Saga for execution. It manages retry behavior for each step. It also keeps track of the history and progress of Workflows, which are displayed in a UI.

The Temporal Worker communicates with Temporal Server to provide durable execution. Thus, all a developer needs to do is to focus on two things. The first is programming the Saga business rules as a Workflow along with its associated Temporal activities. The second is writing the compensation that goes with each step in the Workflow. Temporal does everything else.

Listing 1 below shows an excerpt of code from programming of a four step Saga implemented using a Temporal Workflow programmed in TypeScript. The code is extracted from a file in Temporal's example library.

Granted, the code is a bit lengthy but still well worth examining because it demonstrates the power and elegance of using Temporal to implement the Saga design pattern. The important point to notice about the code is that all the steps and compensations of the Saga are programmed between lines 8 and 60.

```
// View the entire workflow code here…
// … particularly the Retry configuration which can be viewed here.

// workflow implementations as sequential executions
export async function openAccount(params: OpenAccount): Promise<void> {
  const compensations: Compensation[] = [];

  try { //Step 1
    await createAccount({ accountId: params.accountId });
  } catch (err) {
    log.error('creating account failed. stopping.');
    // this is fatal so fails fast. no compensations are needed
    throw err;
  }

  try {
    // addAddress, Step 2
    await addAddress({
      accountId: params.accountId,
      address: params.address,
    });
    // successfully called, so clear if a failure occurs later
    compensations.unshift({
      message: prettyErrorMessage('reversing add address'),
      fn: () => clearPostalAddresses({ accountId: params.accountId }),
    });

    // addClient, Step 3
    await addClient({
      accountId: params.accountId,
      clientEmail: params.clientEmail,
    });
    // successfully called, so clear if a failure occurs later
    compensations.unshift({
      message: prettyErrorMessage('reversing add client'),
      fn: () => removeClient({ accountId: params.accountId }),
    });

```

Listing 1: An example of core code required to implement the Saga design pattern as a Temporal Workflow

```
39      // addBankAccount, Step 4
40      await addBankAccount({
41        accountId: params.accountId,
42        details: params.bankDetails,
43        shouldThrow: prettyErrorMessage('add bank account failed'),
44      });
45      // successfully called, so clear if a failure occurs later
46      compensations.unshift({
47        message: prettyErrorMessage('reversing add bank account'),
48        fn: () => disconnectBankAccounts({ accountId: params.accountId }),
49      });
50    } catch (err) {
51      if (err instanceof ActivityFailure && err.cause instanceof ApplicationFailure) {
52        log.error(err.cause.message);
53      } else {
54        log.error(`error while opening account: ${err}`);
55      }
56      // an error occurred so call compensations
57      await compensate(compensations);
58      throw err;
59    }
60  }
61
62  //Some simple compensation behavior
63  async function compensate(compensations: Compensation[] = []) {
64    if (compensations.length > 0) {
65      // view the actual code here
66    }
67  }
68
69  function prettyErrorMessage(message: string, err?: any) {
70    // view the actual code here
71  }
```

Notice that invoking a compensation requires nothing more than running a step's code within a try...catch statement as shown at Line 16. The elegance and labor savings are apparent.

And just as importantly, all the overhead that goes with running the Saga is handled by the other primitives – Server, Worker and Client – in the Temporal framework. All the work a developer needs to do to create even the most complex of Sagas is centralized in the Temporal Workflow.

Confining Saga programming to the Workflow is a significant benefit that Temporal provides, but there are other benefits as well.

Why choose Temporal

In addition to providing an elegant, efficient programming model, Temporal provides additional benefits as follows:

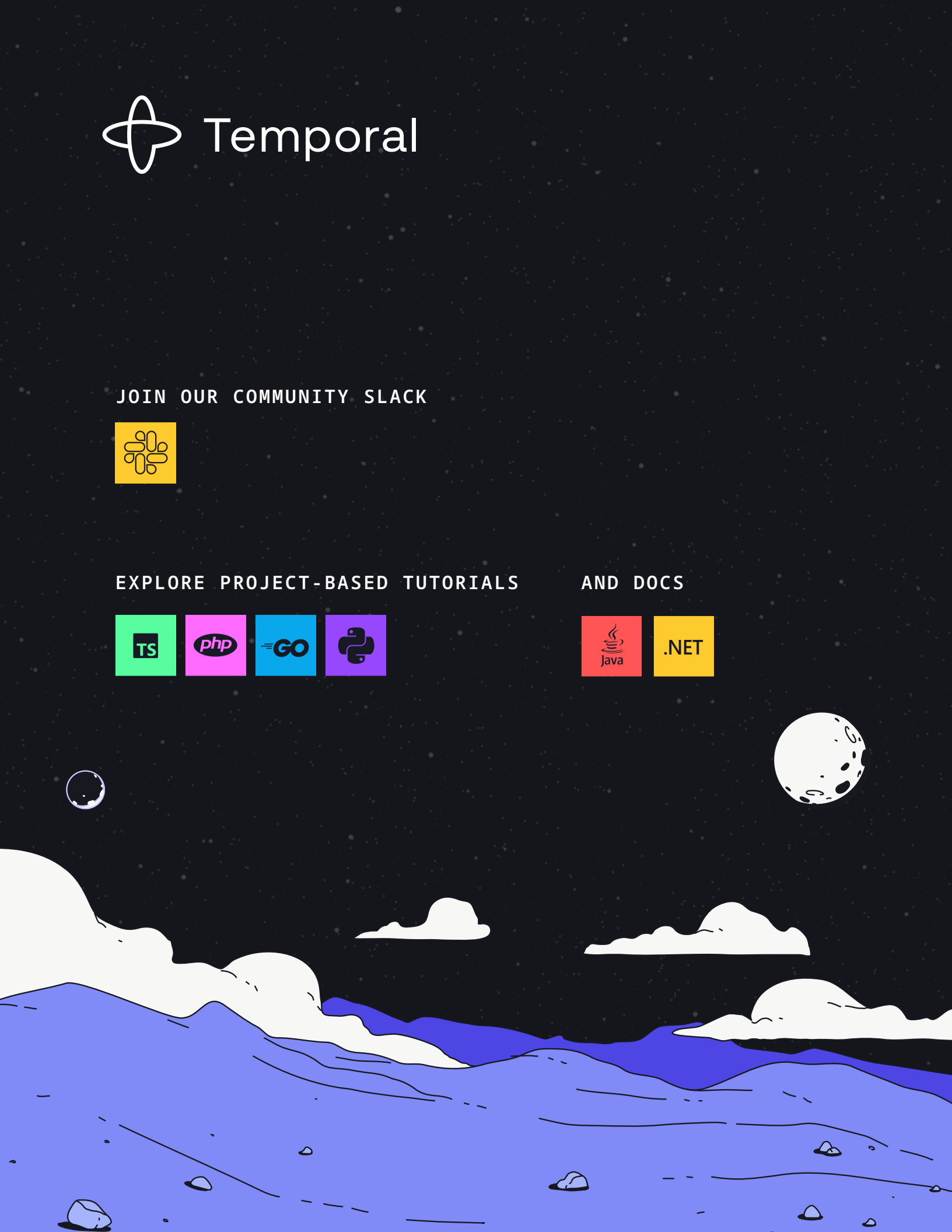
- With Temporal Saga behavior is centralized in a Workflow, thus requiring less cognition to understand the overall logic of the Saga and in turn, making it easier to extend and maintain a given Saga.
- Compensations on nested Sagas are painstaking to handle without an orchestrator; Temporal makes these compensations much easier to reason about and implement.
- Temporal provides the capability to query a Workflow to determine the current state of the transaction process, and these insights let you debug your application more easily.

In conclusion

The Saga design pattern provides a time tested way for creating consistent, multi-step business transactions even in the event of failure. However, the design pattern comes with tradeoffs. Sagas can be hard to program, hard to execute, and hard to maintain, even more so when implementing a Saga from scratch without the help of a reliable framework.

Fortunately the Temporal framework alleviates a good deal of the labor and complexity that goes into creating comprehensive implementations of the Saga design pattern. Temporal is designed in such a way that most of the work that goes into executing a Saga at runtime is automated by the framework even under concurrent conditions. As a result, developers are free to spend their time focused on programming the logic and compensation behavior of the business transaction.

In short, Temporal makes implementing the Saga design pattern easy, safe and reliable.


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