instaclustr
Now part of Spot by NetApp

Cadence® Child Workflows Cookbook

Copyright © 2022 Instaclustr, All rights reserved.

Introduction

Who Is This Cookbook For?

This cookbook is for developers and engineers of all levels looking to understand how child workflows work in Cadence®. The recipe in this book provides "*Hello, World!*" type examples based on simple scenarios and use cases.

What You Will Learn

How to setup a simple Cadence application which implements child workflows on Instaclustr's Managed Service Platform.

What You Will Need

- An account on Instaclustr's managed service platform (sign up for a free trial using the following signup link)
- Basic Java 11 and Gradle installation
- IntelliJ Community Edition Visual Studio Code, or any other IDE with Gradle support
- Docker (optional: only needed to run Cadence command line client)

What Is Cadence?

A large number of use cases span beyond a single request-reply, require tracking of a complex state, respond to asynchronous events, and communicate to external unreliable dependencies. The usual approach to building such applications is a hodgepodge of stateless services, databases, cron jobs, and queuing systems. This negatively impacts developer productivity as most of the code is dedicated to plumbing, obscuring the actual business logic behind a myriad of low-level details.

Cadence is an orchestration framework that helps developers write fault-tolerant, long-running applications, also known as workflows. In essence, it provides a durable virtual memory that is not linked to a specific process, and preserves the full application state, including function stacks, with local variables across all sorts of host and software failures. This allows you to write code using the full power of a programming language while Cadence takes care of durability, availability, and scalability of the application.

What Are Child Workflows?

Cadence's core abstraction is a fault-oblivious stateful **workflow**. A workflow may start other workflows and wait for them to complete—either synchronously or asynchronously. These are known as **child workflows**, and are a powerful tool which enable developers to coordinate large and complex tasks within the Cadence framework.

Workflows evolve over time and child workflows may develop as a result of that evolutuion, such as when 2 existing workflow processes are joined, or designed from the start to take advantage the features available in Cadence.

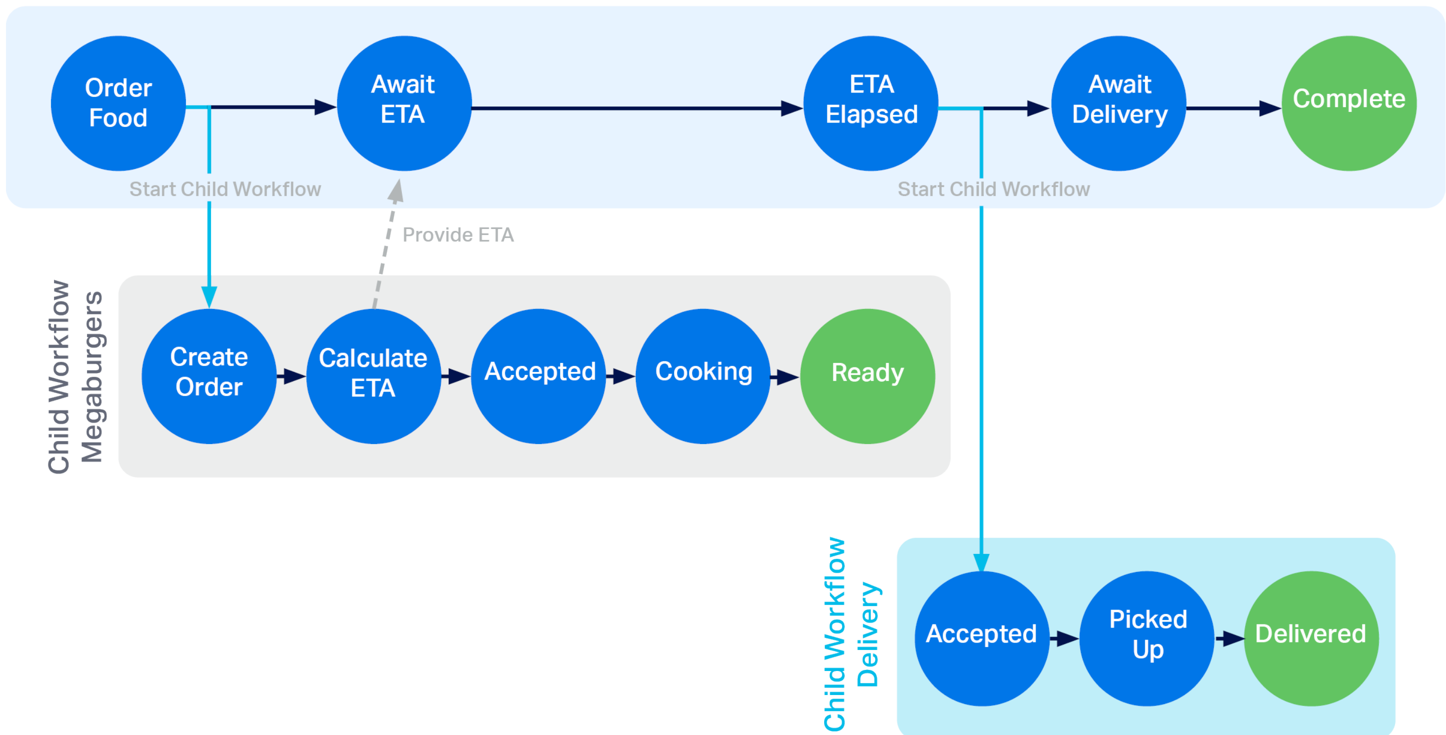


Fig 1. Flow diagram for a child workflow process

What Are the Advantages of Child Workflows?

Child workflows have similar functionality to **activities**. In order to fulfill deterministic execution requirements, workflows are not allowed to call any external API directly. Instead they orchestrate execution of activities. An activity is business-level function that implements application logic such as calling a service or transcoding a media file.

Like activities, child workflows are failure agnostic from the perspective of the parent workflow.

When invoking child workflows, parent workflows are able to monitor the child workflow and track its progress, and signals can be passed back and forth between the 2 with client SDK methods.

Child workflows can be implemented by **separate worker processes**. A worker process is the process which is invoked to execute a particular workflow. Separating out a set of activities into

a child workflow allows you to ensure the code that gets executed the most often has the most resources by scaling those workers appropriately.

Another advantage of using child workflows is to work around the built in Cadence limits. Cadence limits the number of activities that a single workflow can execute. The limits were developed to ensure the most efficient operation of a Cadence cluster. For workflows running a large number of activities, we can use child workflows to distribute the work, thereby removing the limit and also gaining the other advantages mentioned.

When Should I Use a Child Workflow?

In many ways, the answer to this question is similar to "when should I extract this code into a function?". If you find your workflows calling the same set of activities, in the same order, it is a prime candidate to be invoked via child workflow.

If your workflow is approaching the limit of activity invocations, using child workflows is required to avoid running into the limits previously mentioned.

Finally, child workflows are a great way to set boundaries between areas of functionality. Keeping all the code related to a specific functional area or responsibility makes them more reusable. We will see an example of this in the example below, where we use child workflows to encapsulate the workflows of different restaurants and delivery processes.

Invoking a Child Workflow

The code to invoke a child workflow is almost identical to invoking a regular workflow:

```
// We first create a stub for the child workflow and include a set of options
MyCustomChildWorkflow child = Workflow.newChildWorkflowStub
(MyCustomChildWorkflow.class, options);

// Asynchronous
// Then we call the function to start it
// This is a non blocking call that returns immediately.
Promise<String> result = Async.function(child::doWork);

// Or synchronous
// Block until the child workflow completes
child.doWork();
```

We start by creating a **child workflow stub**. When we create this stub we can optionally pass it some **child workflow options**, which allow you to change the rules of how this child workflow executes compared to its parent. Some of the options we can change are the domain, tasklist, timeout settings, retry settings, and more. If we choose to omit this value, the child workflow will inherit all of these settings from its parent.

The workflow stub has the methods to start the workflow, and they can be invoked in 2 ways:

1. **Asynchronously**: This returns a promise object, a *Future* which will be used to store the result of the child workflow after it has completed. After making this call, the workflow code will continue and the child workflow will execute in parallel.
2. **Synchronously**: This will block like a regular function call and is invoked as such. The parent workflow won't continue until the child workflow has completed.

Child Workflow Limitations

Since workflows are the main building blocks of Cadence, there are very few limitations to child workflows.

However one to be aware of is that there is no shared state between the parent and child workflows. Asynchronous communication can be built between the workflow instances, but if there is constant communication of state going back and forth, its probably a better candidate for a single workflow.

Use Case Example: Instafood Meets MegaBurgers

In order to see this pattern in action, we'll create some child workflows in our sample project.

Instafood Brief

Instafood is an online app-based meal delivery service. Customers can place an order for food from their favorite local restaurants via Instafood's mobile app. Orders can be for pickup or delivery. If delivery is chosen, Instafood will organize to have one of their many delivery drivers pickup the order from the restaurant and deliver it to the customer.

Instafood provides each restaurant a kiosk/tablet which is used for communication between Instafood and the restaurant. Instafood notifies the restaurant when an order is placed, and then the restaurant can accept the order, provide an ETA, mark it as ready, etc. For delivery orders, Instafood will coordinate to have a delivery driver pick up based on the ETA.

Ordering from "MegaBurgers"

MegaBurgers is a large multinational fast food hamburger chain. They have an existing mobile app and website that uses a back-end REST API for customers to place orders. Instafood's backend will directly integrate with MegaBurgers's existing REST-based ordering system to place orders and receive updates.

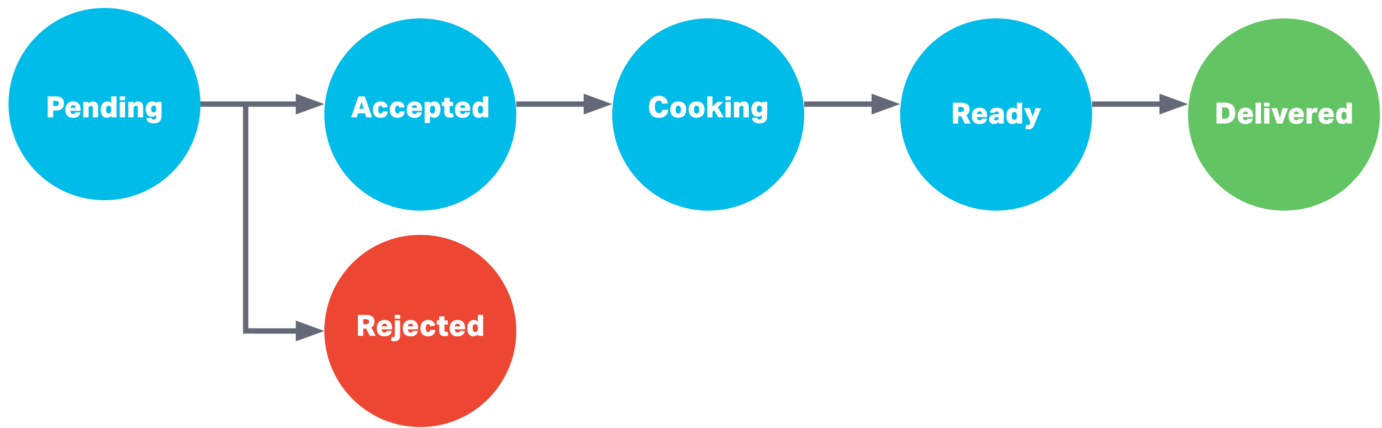


Fig 2. MegaBurger's order state machine

We are designing Instafood to offer meals from any restaurant that signs up to our service. Each company will have its own method to take orders and monitor progress, which makes it a perfect candidate to implement via a child workflow.

In the case of MegaBurger this is done via the following if statment, which is made at the time of ordering.

```
if (Restaurant.MEGABURGER.equals(order.getRestaurant())) {
   MegaBurgerOrderWorkflow megaBurgerOrderWorkflow = Workflow.newChildWorkflowStub(MegaBurgerOrder
   Workflow.class); Async.procedure(megaBurgerOrderWorkflow::orderFood, order);
}
```

The above example shows how simple this makes the parent workflow, it doesn't have to know about the various peculiarities of each company's order process. That's the job of the child workflow to implement.

Setting up Instafood Project

In order to run the sample project yourself you'll need to set up a Cadence cluster. We'll be using Instaclustr's Managed Service platform to do so.

Step 1: Creating Instaclustr Managed Clusters

A Cadence cluster requires an Apache Cassandra® cluster to connect to for its persistence layer. In order to set up both Cadence and Cassandra clusters we'll follow "Creating a Cadence Cluster" documentation.

By using Instaclustr platform, the following operations are handled automatically for you:

- Firewall rules will automatically get configured on the Cassandra cluster for Cadence nodes.
- Authentication between Cadence and Cassandra will get configured, including client encryption settings.
- The Cadence default and visibility keyspaces will be created automatically in Cassandra.
- A link will be created between the 2 clusters, ensuring you don't accidentally delete the Cassandra cluster before Cadence.
- A load balancer will be created. It is recommended to use the load balancer address to connect to your cluster.

Step 2: Setting up Cadence Domain

Cadence is backed by a multi-tenant service where the unit of isolation is called a domain. In order to get our Instafood application running we first need to register a domain for it.

1. In order to interact with our Cadence cluster, we need to install its command line interface client.

macOS

If using a macOS client the Cadence CLI can be installed with Homebrew as follows:

```
brew install cadence-workflow
# run command line client
cadence <command> <arguments>
```

Other Systems

If not, the CLI can be used via Docker Hub image `ubercadence/cli`:

```
# run command line client
docker run --network=host --rm ubercadence/cli:master <command> <arguments>
```

For the rest of the steps we'll use `cadence` to refer to the client.

2. In order to connect, it is recommended to use the load balancer address to connect to your cluster. This can be found at the top of the *Connection Info* tab, and will look like this: "ab-cd12ef23-45gh-4baf-ad99-df4xy-azba45bc0c8da111.elb.us-east-1.amazonaws.com". We'll call this the <cadence_host>.

3. We can now test our connection by listing current domains:

```
cadence --ad <cadence_host>:7933 admin domain list
```

4. Add `instafood` domain:

```
cadence --ad <cadence_host>:7933 --do instafood domain register
```

5. Check it was registered accordingly:

```
cadence --ad <cadence_host>:7933 --do instafood domain describe
```

Step 3: Run Instafood Sample Project

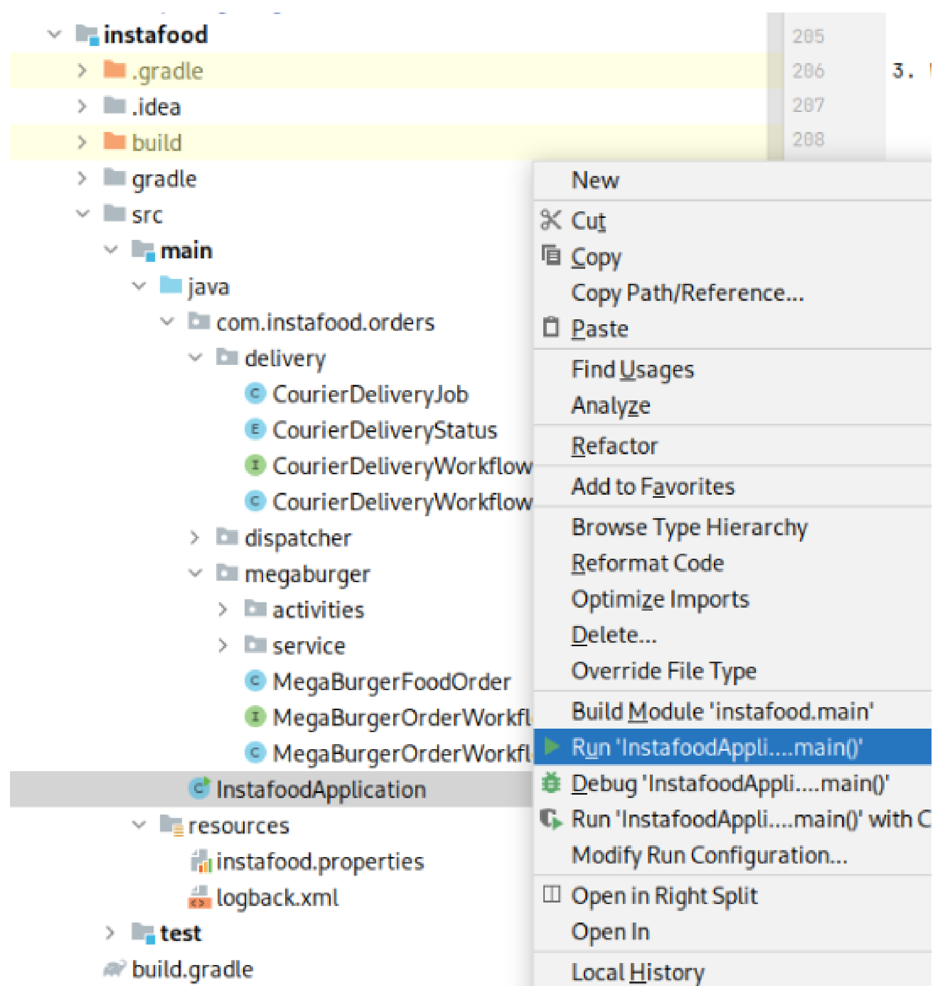
1. Clone Gradle project from Instafood project git repository.
2. Open property file at `instafood/src/main/resources/instafood.properties` and replace `cadenceHost` value with your load balancer address:

```
cadenceHost=<cadence_host>
```

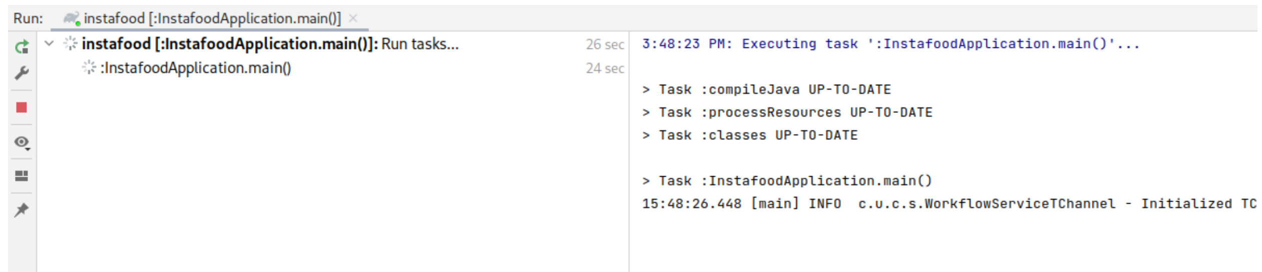
3. You can now run the app by

```
cadence-cookbooks-instafood/instafood$ ./gradlew run
```

or executing *InstafoodApplication* main class from your IDE:



4. Check it is running by looking into its terminal output:



Instafood Order Workflow Review

Now that we have everything set up, lets look at the actual integration between Instafood and Megaburger, and how child workflows are used.

First, lets look at the Instafood workflow. The main function is `orderFood`, which gets started when an order is placed:

Instafood Workflow

```
public void orderFood(FoodOrder order) {
        if (Restaurant.MEGABURGER.equals(order.getRestaurant())) {
             MegaBurgerOrderWorkflow mega BurgerOrderWorkflow = Workflow.newChildWorkflowStub(Mega
             BurgerOrderWorkflow.class);
              Async.procedure(megaBurgerOrderWorkflow::orderFood, order);
        } else {
            throw new RuntimeException("Restaurant option not available");
        }
        // Wait for an ETA or abort if restaurant rejected order
        Workflow.await(() -> etaInMinutes != -1 || OrderStatus.REJECTED.equals(currentStatus));
        if (OrderStatus.REJECTED.equals(currentStatus)) {
            throw new RuntimeException("Order was rejected by restaurant");
        }

        if (!order.isPickup()) {
            // Wait for predicted ETA or until order marks as ready
            Workflow.await(Duration.ofMinutes(getTimeToSendCourier()), () -> OrderStatus.READY.
           equals(currentStatus));

            CourierDeliveryWorkflow courierDeliveryWorkflow = Workflow.newChildWorkflowStub(CourierDe
           liveryWorkflow.class);
            Async.procedure(courierDeliveryWorkflow::deliverOrder, new CourierDeliveryJob(order.
           getRestaurant(), order.getAddress(), order.getTelephone()));

            Workflow.await(() -> OrderStatus.COURIER_DELIVERED.equals(currentStatus));
        } else {
            Workflow.await(() -> OrderStatus.RESTAURANT_DELIVERED.equals(currentStatus));
        }
    }
```

We can see here, we currently only support the Megaburger restaurant, but there is scope to add more later!

As we mentioned earlier, we invoke the child workflow by creating the **child workflow stub** and starting the `orderFood` workflow on it.

This particular workflow continues executing while the child workflow is progressing, and then it will block while it waits for a signal from the child workflow. Once this message is received, it can progress to the next stage.

Later in the workflow definition, we can see that we call another child workflow. This one is responsible for dispatching a courier to pickup the order from the restaurant and then deliver the order to the customer.

As we have explained, communication between the parent and child workflow is possible via asynchronous messages.

Let's look at how that is implemented in this workflow.

```
// ...

// Wait for an ETA or abort if restaurant rejected order
Workflow.await(() -> etaInMinutes != -1 || OrderStatus.REJECTED.equals(currentStatus));

// ...
```

Here our workflow is **polling** until the order ETA is updated, so how is that happening?

(For more details on polling in cadence workflows, have a read of our polling cookbook)

Let's look at our parent workflow and its **interface definition**, where it has defined the following method:

Instafood Workflow—Interface

```
@SignalMethod
void updateEta(int estimationInMinutes);
```

The `@SignalMethod` annotation decorates the method and informs Cadence that this method is used to receive signals from an external process, in this case it's coming from the child workflow, which we can see here:

```
private OrderWorkflow getParentOrderWorkflow() {
   String parentOrderWorkflowId = Workflow.getWorkflowInfo().getParentWorkflowId();
   return Workflow.newExternalWorkflowStub(OrderWorkflow.class, parentOrderWorkflowId);
}

OrderWorkflow parentOrderWorkflow = getParentOrderWorkflow();

// Send ETA to parent workflow
parentOrderWorkflow.updateEta(getOrderEta(orderId));

// ...
```

Let's break this down a bit:

1. First, we create a stub to our parent workflow. We do this by calling the Cadence SDK to get the ID of the parent workflow, the create the stub for it.
2. Now that we have the stub, we gain access to the methods on it. We call **updateEta** and provide the ETA, which gets sent asynchronously.
3. This child workflow continues as the order is being prepared.
4. Our parent workflow receives the signal, and then it can also progress.

Reflection—Instafood and Child Workflows

Above is a good example of how to use child workflows, and it's also a great example of **why** you want to use them.

Here we are calling a child workflow asynchronously and letting it signal to the parent workflow when the important information is ready. The parent workflow can keep working until this information is ready. This is cruicial for a workflow such as this, where we want to inform the customer of an ETA or dispatch a courier, but it doesn't make sense for the restaurant preparation workflow to have this responsibility.

Similarly, the primary workflow doesn't concern itself with how each restaurant implements the various requirements for updating ETA and status. This makes the workflow simple to implement and easy to understand.

Running a Happy-Path Scenario

To wrap-up, let's run a whole order scenario. This scenario is part of the test suite included with our sample project. The only requirement is running both Instafood and MegaBurger server as described in the previous steps. This test case describes a client ordering through Instafood MegaBurger's new *Vegan Burger* for pick-up:

Let's start by running the server. This can be accomplished by running

```
cadence-cookbooks-instafood/instafood$ ./gradlew test
```

or *InstafoodApplicationTest* from your IDE

```
class InstafoodApplicationTest {

    // ...

    @Test
    public void givenAnOrderItShouldBeSentToMegaBurgerAndBeDeliveredAccordingly() {
        FoodOrder order = new FoodOrder(Restaurant.MEGABURGER, "Vegan Burger", 2, "+54 11 2343-
        2324", "Díaz velez 433, La lucila", true);

        // Client orders food
        WorkflowExecution workflowExecution = WorkflowClient.start(orderWorkflow::orderFood, order);

        // Wait until order is pending Megaburger's acceptance
        await().until(() -> OrderStatus.PENDING.equals(orderWorkflow.getStatus()));

        // Megaburger accepts order and sends ETA
        megaBurgerOrdersApiClient.updateStatusAndEta(getLastOrderId(), "ACCEPTED", 15);
        await().until(() -> OrderStatus.ACCEPTED.equals(orderWorkflow.getStatus()));

        // Megaburger starts cooking order
        megaBurgerOrdersApiClient.updateStatus(getLastOrderId(), "COOKING");
        await().until(() -> OrderStatus.COOKING.equals(orderWorkflow.getStatus()));

        // Megaburger signals order is ready
        megaBurgerOrdersApiClient.updateStatus(getLastOrderId(), "READY");
        await().until(() -> OrderStatus.READY.equals(orderWorkflow.getStatus()));

        // Megaburger signals order has been picked-up
        megaBurgerOrdersApiClient.updateStatus(getLastOrderId(), "RESTAURANT_DELIVERED");
        await().until(() -> OrderStatus.RESTAURANT_DELIVERED.equals(orderWorkflow.getStatus()));

        await().until(() -> workflowHistoryHasEvent(workflowClient, workflowExecution, EventType.
        WorkflowExecutionCompleted));
    }
}
```

We have 3 actors in this scenario: Instafood, MegaBurger, and the Client.

1. The Client sends order to Instafood.
2. Once the order reaches MegaBurger (order status is `PENDING`), MegaBurgers marks it as `ACCEPTED` and sends an ETA.
3. We then have the whole sequence of status updates:
    i. MegaBurger marks order as `COOKING`.

    ii. MegaBurger marks order as `READY` (this means it's ready for delivery/pickup).

    iii. MegaBurger marks order as `RESTAURANT_DELIVERED`.

4. Since this was an order created as pickup, once the Client has done so the workflow is complete.

Wrapping Up

In this article we got first-hand experience with Cadence and how to use child workflows. We also showed you how to get a Cadence cluster running with our Instaclustr platform and how easy it is to get an application connect to it.

If you're interested in Cadence and want to learn more about it, you may read about other use cases and documentation at Cadence workflow—Use cases.

© 2022 Instaclustr Copyright | Apache®, Apache Cassandra®, Apache Kafka®, Apache Spark™, and Apache ZooKeeper™ are trademarks of The Apache Software Foundation. Elasticsearch™ and Kibana™ are trademarks for Elasticsearch BV. Kubernetes® is a registered trademark of the Linux Foundation. OpenSearch is a registered trademark of Amazon Web Services. Postgres®, PostgreSQL® and the Slonik Logo are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission. Redis™ is a trademark of Redis Labs Ltd. *Any rights therein are reserved to Redis Labs Ltd. Cadence is a trademark of Uber Technologies, Inc. Any use by Instaclustr Pty Limited is for referential purposes only and does not indicate any sponsorship, endorsement or affiliation between Redis and Instaclustr Pty Limited. All product and service names used in this website are for identification purposes only and do not imply endorsement.