Camunda 8 - Error Handling

What Will I Learn?

This course will provide you with an introduction to handling errors in Camunda 8.

You will make a series of changes to a *Payment Application* so that it can handle errors encountered during process execution more effectively.

What Will I Learn?

At the end of this course you will be able to:

- Access a Camunda 8 Environment
    - Access Console
    - Access Modeler
    - Access Tasklist
    - Access Operate
- Create a Camunda 8 Application
    - Create a Project using the Modeler
    - Import a Process Definition using the Modeler
    - Modify a Process Definition using the Modeler
    - Import a Form Definition using the Modeler
    - Add BPMN Error Handling to a Process Definition
    - Add BPMN Error Handling to a Job Worker
- Deploy a Camunda 8 Application
    - Deploy a Process Definition using the Modeler
    - Deploy an updated Process Definition using the Modeler
- Test a Process Definition
    - Start a Process Instance using the Modeler
    - Claim a Task using Tasklist
    - Populate a Form using Tasklist
    - Complete a Task using Tasklist

- Monitor a Process Definition
    - View a Process Instance using Operate
    - Modify a Process Instance using Operate
    - Cancel a Process Instance using Operate
    - Delete a Process Instance using Operate
- Monitor an Incident
    - Differentiate between a BPMN Error and an Incident
    - Retry a Failed Job

Business Requirement

We will be using a fictitious organization; Camundanzia, as the scenario for this course.

Camundanzia

Scenario

The Camundanzia insurance company has started to introduce process automation throughout their organization. A process that has already been automated is their **Payment** process.

Camundanzia has developed an eCommerce website (Webshop) which allows insurance partners to order Camundanzia branded products such as pens, notepads and water bottles for marketing events. Camundanzia partners must purchase these items using a credit card.

Partner credit card purchases are processed automatically without the need for approval by an **Accountant**.

The **Payment** process has been running smoothly in production for most of the time since deployment. However, there have been some issues when the **Credit Card Service** has been unavailable or orders have been processed with the wrong credit card information.

Camundanzia would like to improve the **Payment** process so that these issues are handled correctly.

User Story

Three simple User Stories to capture Camundanzia's requirements can be found below:

User Story

- As an **Accountant**
- I want a **Process Operator** to manage the incidents caused by the **Credit Card Service** being unavailable
- So that the orders are processed efficiently

User Story

- As an **Accountant**
- I want a **Process Operator** to manage errors caused by incorrect credit card details
- So that all payments are correctly received by Camundanzia

User Story

- As a **Process Operator**
- I want the **Payment** process to automatically retry processing (with an appropriate back off strategy) in the event that there are problems with the **Credit Card Service**
- So that short outages do not require manual intervention

Acceptance Criteria

The initial acceptance criteria that we will be working to are:

- When an order fails because the **Credit Card Service** is unavailable
    - A **Process Operator** is able to see the details of the failed order in Camunda
    - A **Process Operator** is able to manually re-trigger the invocation of the **Credit Card Service** for an order
    - The **System** automatically retries the invocation of the **Credit Card Service** twice, with a 20-second delay between requests
- When an order fails because there is an issue with the credit card details which have been provided
    - A **Process Operator** is able to review and modify the incorrect information
    - A **Process Operator** is able to re-submit the order

Iterative Development

Camunda 8 is well suited to an iterative development approach; applications can be easily revised and upgraded without affecting in-flight processes.

Outline Solution

We now have a clear set of requirements and acceptance criteria for updating the Camundanzia webshop *Payment Process*.

The sections below will provide you with an overview of how to modify the existing application in order to meet the new requirements.

Camundanzia

Scenario

Imagine that you have recently joined Camundanzia as a new Developer and have been assigned to the Camunda 8 project team. Your first assignment is to update the *Payment Process* using the requirements from the previous lesson.

Fortunately for you, a more experienced Camunda Developer has drafted an outline solution for you. Your task is to implement a solution based upon the outline that has been provided.

Outline Solution

The more experienced Developer has provided an overview of the work that must be completed for each of the stated acceptance criteria. Both the configuration that you, the Developer, must carry out and the Camunda components that provide the necessary capabilities are covered by this.

Review Payment Application

The *Payment Application* has already been deployed into Production and has processed many orders. We first need to deploy this into a Test Environment and start a process instance so that we can become familiar with its behavior.

Deploy the Payment Process

- Deploy the existing *Payment Process* into your Test Environment

Configure the Payment Worker

- Configure the *Payment Worker* to connect to your Test Environment

Start a Process Instance

- Start a new instance of the *Payment Process* in your Test Environment

Handle Incidents

- The CreditCardService will throw an exception in the event of an error. We next need to understand how Operate can be used to handle and resolve Incidents.

Handle Incident in Operate

- Identify and resolve an Incident in the *Payment Process* from Operate

Handle Incident in Payment Worker

- Retry a Failed Job in the *Payment Process*

Handle BPMN Errors

- The CreditCardService will throw an exception in the event of an error. This default behavior can be improved with the addition of BPMN Errors in the *Payment Process* and *Payment Application*.

Update the Job Worker

- Update the existing *Payment Application* to throw a BPMN Error when invalid credit card details have been provided.

Update the Process Definition

- Update the *Payment Process* to catch the BPMN Error thrown by the *Payment Application* and create a User Task so that the error can be resolved.

Development Plan

Now that we have identified the steps to implement our solution, we can create a plan of the work that needs to be performed and the tools that will be required.

- **Create a Project**
    - This will be performed from the Web Modeler
- **Import the Payment Process**
    - This will be performed from the Web Modeler
- **Deploy the Payment Process**
    - This will be performed from the Web Modeler
- **Configure the Payment Worker**
    - This will be performed from a Java IDE
- **Start a Process Instance**
    - This will be performed from the Web Modeler
- **Handle an Incident in Operate**
    - This will be performed from Operate

- **Handle an Incident in the Job Worker**
    - This will be performed from a Java IDE
- **Throw a BPMN Error from a Job Worker**
    - This will be performed from a Java IDE
- **Catch a BPMN Error inside a Process Definition**
    - This will be performed from the Web Modeler
- **Test the updated Payment Application**
    - This will be performed from a Java IDE, Modeler and Operate

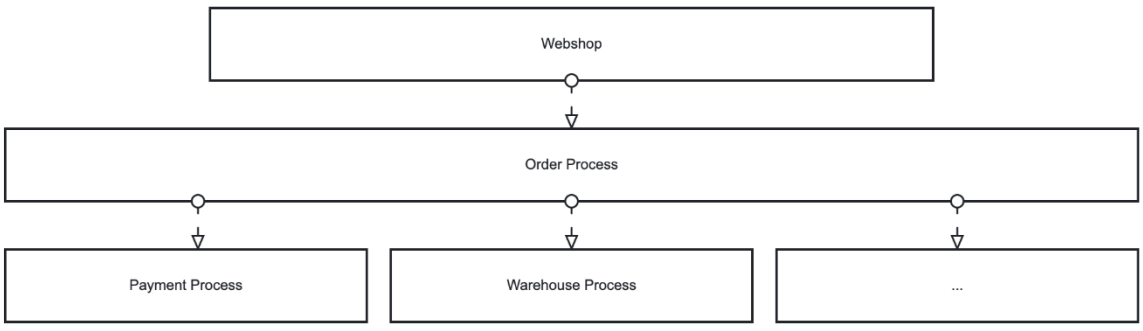
Overview

The Camundanzia *Payment Process* has been running smoothly in production for most of the time since deployment. However, there have been some issues when the **Credit Card Service** has been unavailable or orders have been processed with invalid credit card information.

Payment Process

The *Payment Process* is one part of the *Order Process* which will be started whenever a **Partner** places an order from the webshop. The *Order Process* orchestrates all of the tasks which need to be completed to fulfil the order.

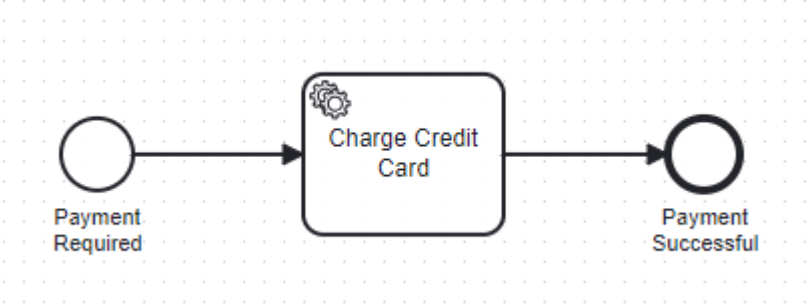
The *Order Process* contains an activity to invoke the *Payment Process* which is the focus of this course.



Review the Payment Process

Let's take a look at the current version of the *Payment Process*. It is quite simple and consists of the following BPMN elements:

Name	Value
Start Event	Payment Required
Service Task	Charge Credit Card
End Event	Payment Successful



The business logic of the process is encapsulated in the *Charge Credit Card* Service Task.

How are Errors Handled?

This is the first iteration of the *Payment Process* and it works well for majority of scenarios encountered so far. However, what happens to the process instance when something unexpected occurs?

- The **Credit Card Service** cannot be reached
- The **Credit Card Service** provides an unexpected response
- The **Partner** has provided incorrect credit card information

The initial implementation of the *Payment Process* does not explicitly handle any of these scenarios. Therefore, we need to provide a more robust implementation by modifying the *Payment Process* and the *Payment Worker*.

Create Project

Overview

To start implementing our business requirements, we will need to create a **Project**, all of the application configuration that we will be working with will be contained in this **Project**.

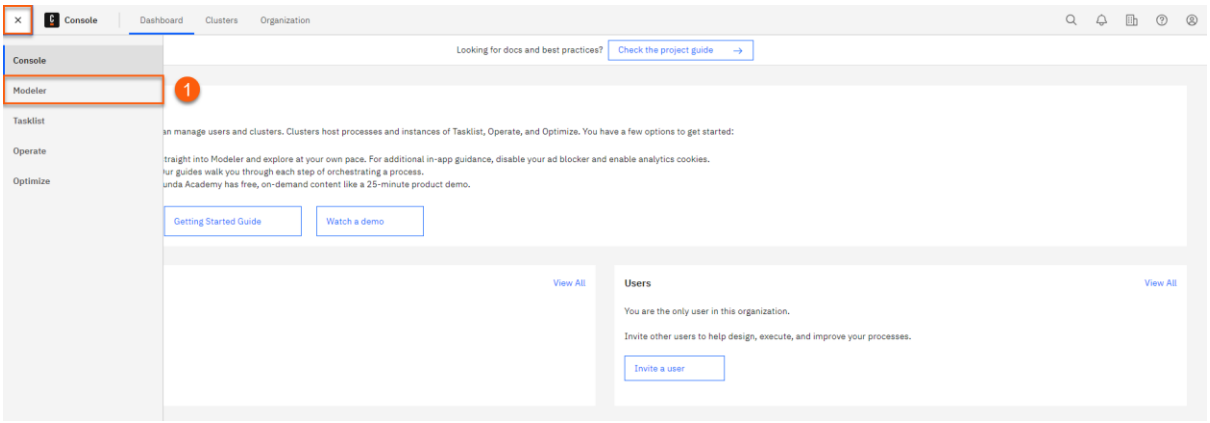
What is a Project?

A **Project** is a top-level organizational component for objects in Camunda 8. **Projects** group multiple process definitions, forms and folders into a logical project that represents a business process.

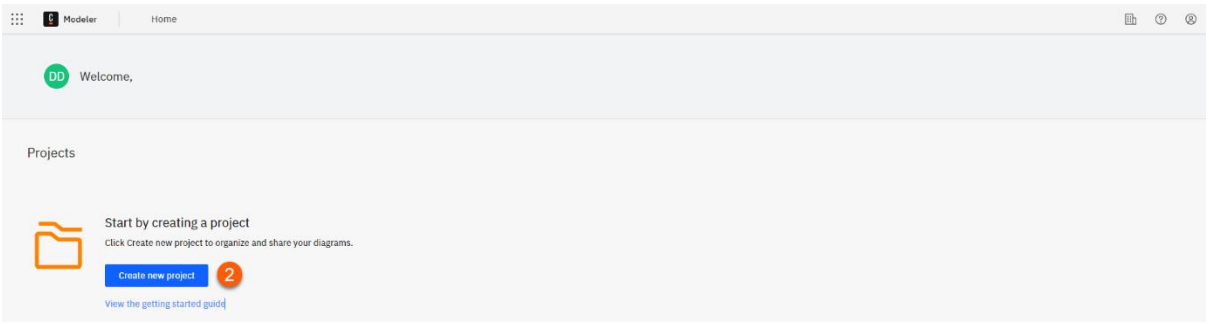
Create a Project

Create a new **Project** by following the steps below:

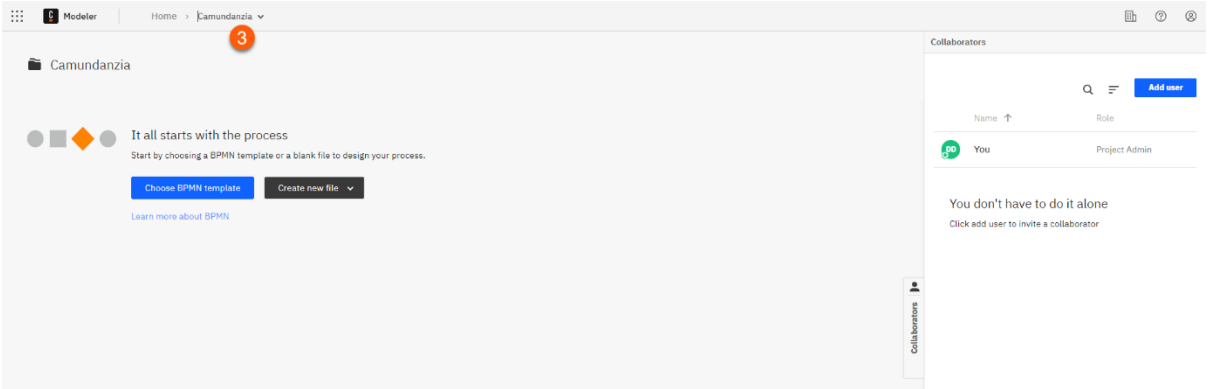
1. Click on the **Modeler** link from the top left panel of the **Console**



2. From the center of the screen, click on the **Create new project** button to create a new project



3. A text box will appear in the menu bar into which you will need to enter a **Project Name**. For this course we will be using *Camundanzia*



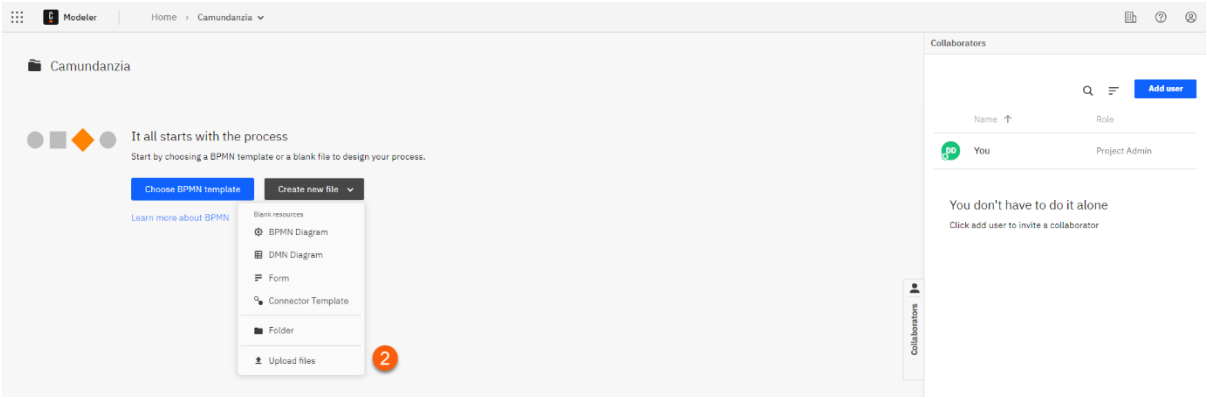
Import Payment Process

We will now import the current version of the *Payment Process* into our **Project**.

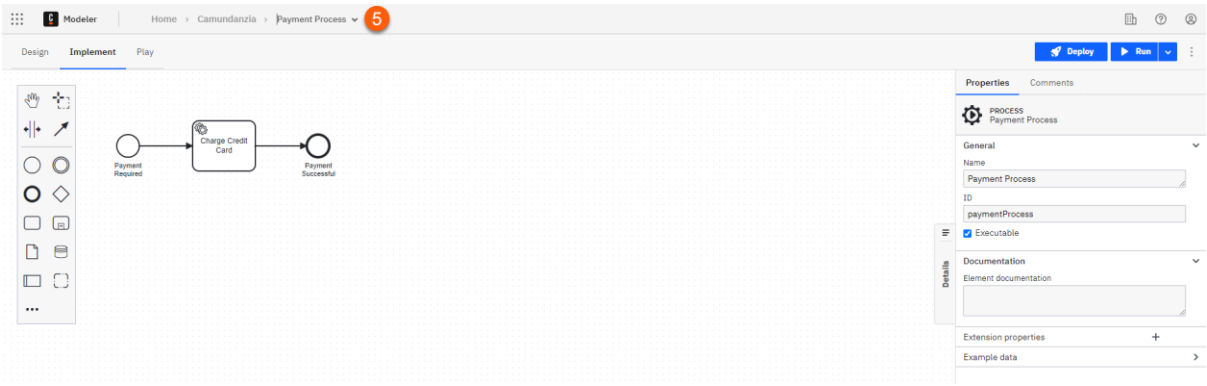
Import Process Definition

Import the existing **Process Definition** by following the steps below:

1. Download the *Payment Process* from this GitHub Repository and save it to your local filesystem
2. From the *Camundanzia* project in the **Modeler**, click the **Create new file** button and then **Upload files** from the menu that appears



3. From the dialog box that appears, select the paymentProcess.bpmn that you just downloaded. This will import the existing **Process Definition** into our project.
4. Double click on the process that you just imported to open it in the Modeler.
5. Click on the **Diagram Name** and then **Edit Name** from the top menu to change the name of the BPMN Diagram. For this course we will use *Payment Process*.



Deploy Payment Process

We will now need to deploy our *Payment Process* so that we can start a **Process Instance**.

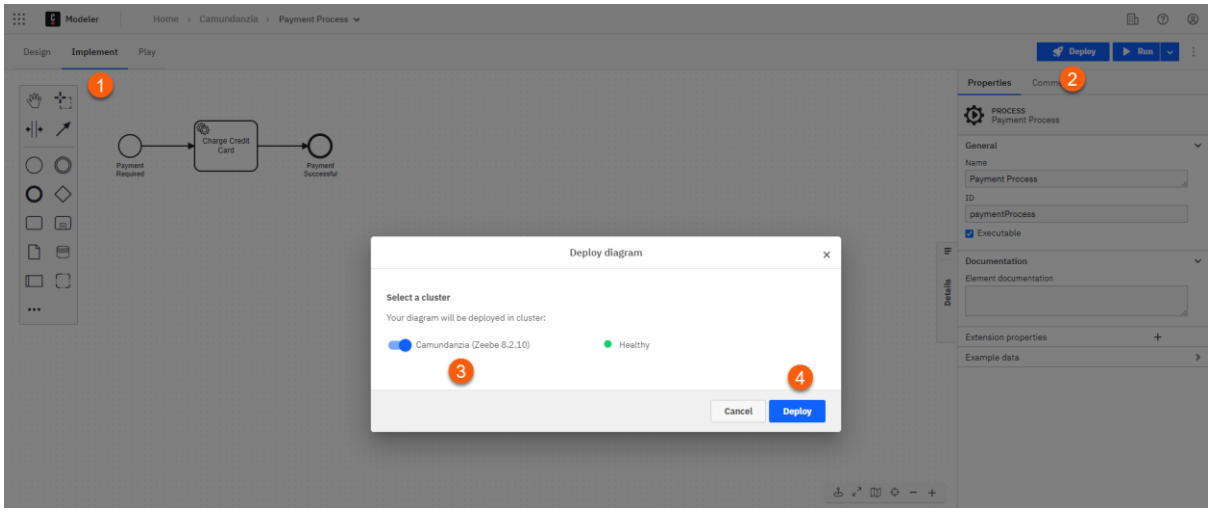
Deploy Process Definition

Deploy the **Process Definition** by following the steps below:

What is a Deployment?

A **Deployment** allows the **Process Definition** to be executed by the **Zeebe Engine**.

1. Select the **Implement** view if it has not already been selected
2. Click on the **Deploy** button in the top right corner of the **Modeler**
3. Ensure that the *Camundanzia* **Cluster** is selected from the dialog box that appears.
4. Click the **Deploy** button to deploy the **Process Definition** to the **Cluster**



5. A **Diagram Deployed** message will be displayed at the bottom of the **Modeler** to show that this was successful.

View the Deployment in Operate

Next, we can take a look into **Operate** to confirm that the deployment has been successful.

What is Operate?

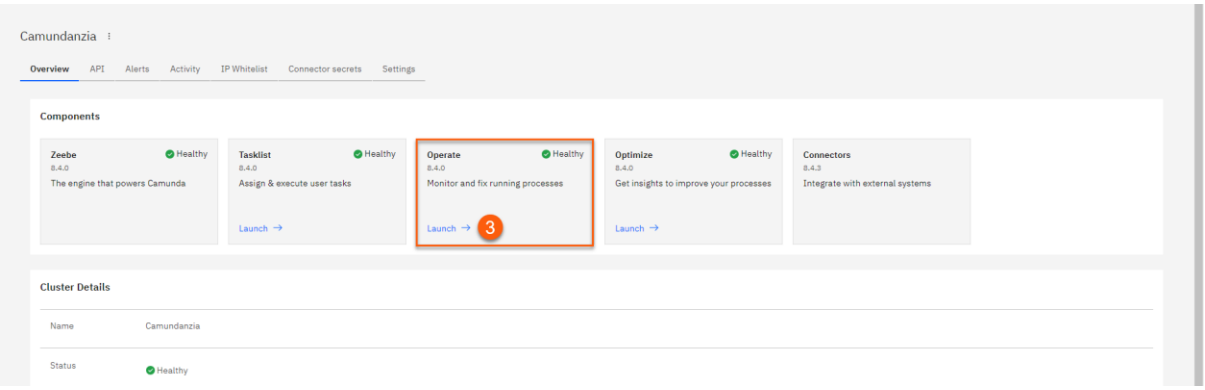
Operate is a tool for monitoring and troubleshooting process instances running in the **Zeebe Engine**.

In addition to providing visibility into active and completed process instances, Operate also makes it possible to carry out key operations such as resolving incidents, and updating process instance variables.

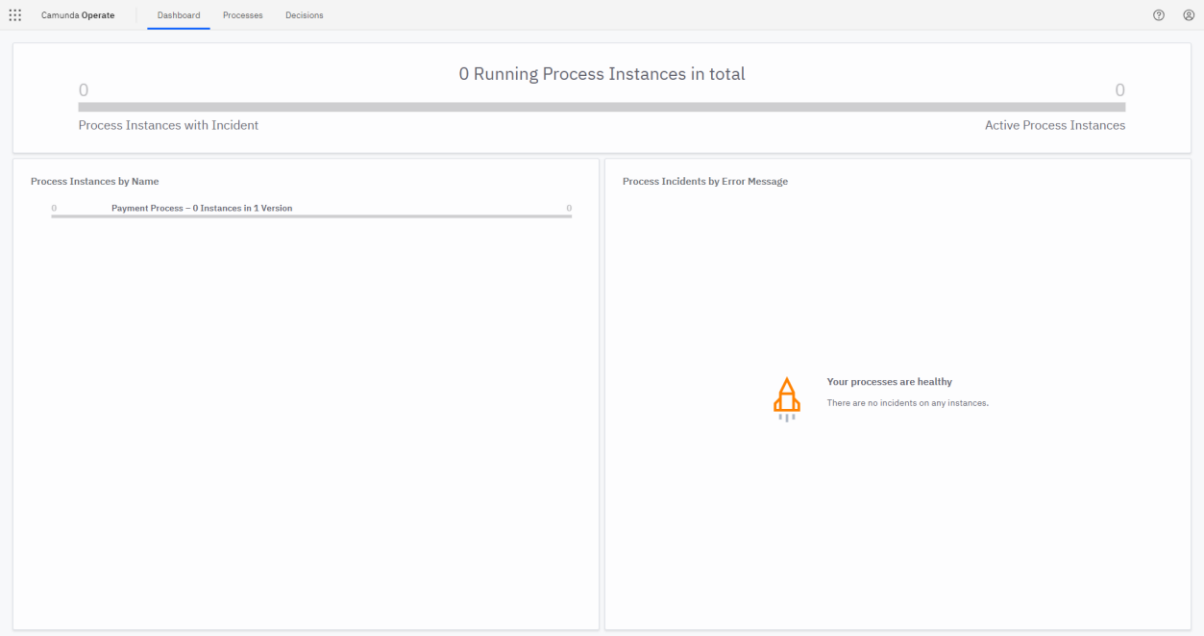
Launch Operate

Launch **Operate** by following the steps below:

1. From the **Console**, select **Clusters**.
2. Select *Camundanzia* from the **Clusters** page.
3. Click the **Launch** link next to **Operate** in the **Cluster Details** page



4. After a few seconds, the **Operate** Dashboard will be displayed.
5. You will also be able to see that the *Payment Process* is visible in the Dashboard.



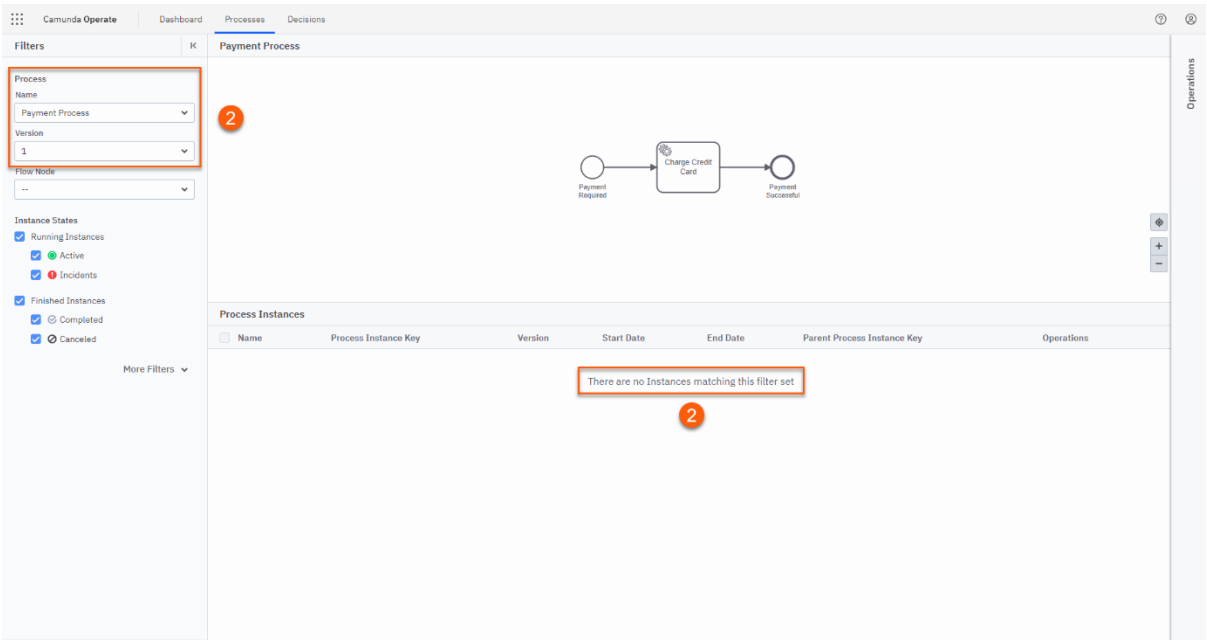
View Process Instances

We can now view the process instances in **Operate** by following the steps below:

What is a Process Instance?

One execution of a **Process Definition** is called a **Process Instance**. Many **Process Instances** can be launched from one **Process Definition**.

1. Click on *Payment Process* from the **Process Instances by Name** panel on the left-hand side of the screen
2. **Operate** will not show any active process instances at this point. However, we can see that version 1 of our *Payment Process* has been successfully deployed.



Configure Payment Worker

We will now configure the Payment Worker to connect to our Test Environment. This will allow us to invoke the *Charge Credit Card* task in the *Payment Process*.

What is a Job Worker?

A **Job Worker** is a service capable of performing a particular task in a process. **Job Workers** are used to manage the interactions between the Zeebe Engine and the outside world.

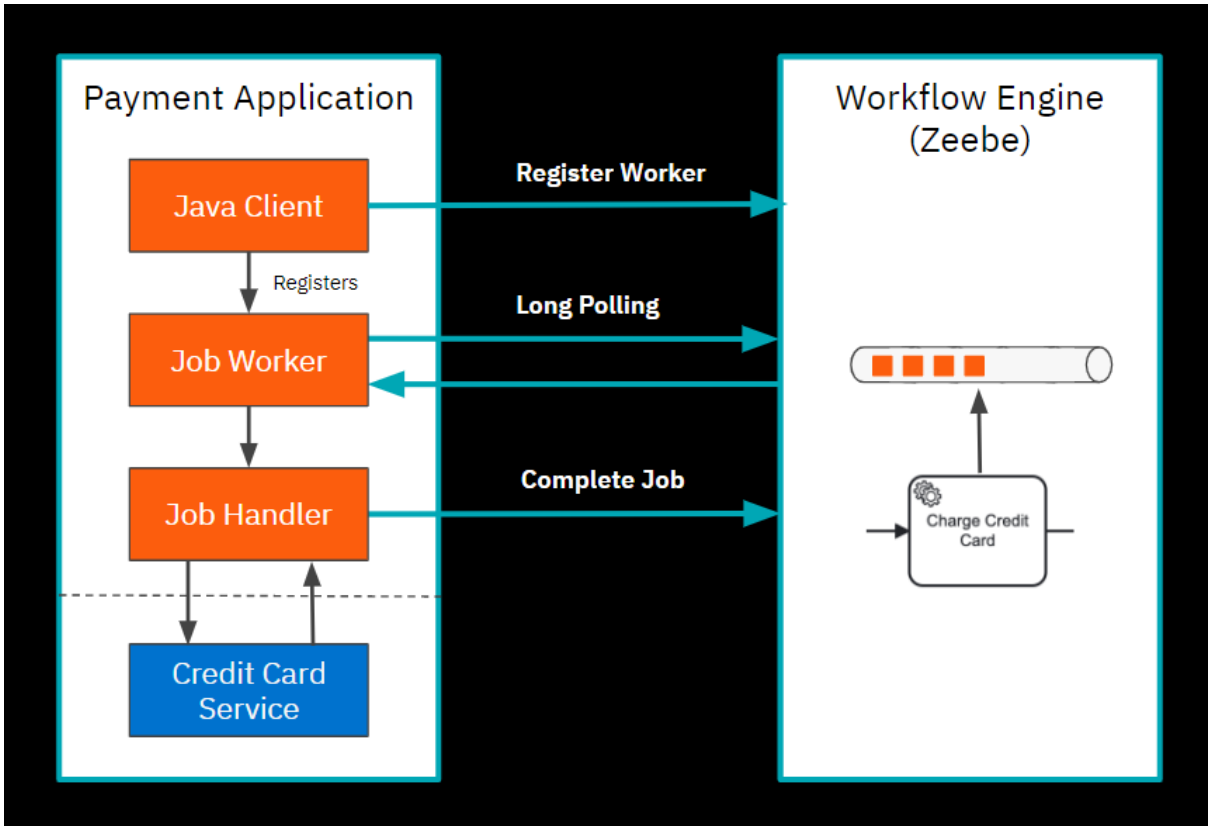
In this course we will be orchestrating a *Credit Card Service* from the *Payment Process*. The *Payment Process* is represented by a **Process Definition** and is executed as a **Process**

Instance by Zeebe. The *Credit Card Service* is external to Zeebe and could be hosted anywhere; it is the **Job Worker** that connects the *Credit Card Service* to the *Payment Process*.

Payment Application

A *Payment Application* containing the following elements has already been developed:

- Zeebe Client (Java) - The Zeebe Client allows you to interact with Zeebe using the API
- Job Worker (PaymentWorker.java) - The Job Worker is responsible for requesting Jobs of a defined type from the Zeebe Engine
- Job Handler (CreditCardServiceHandler.java) - The Job Handler is responsible for performing the actions required to complete a Job when it has been acquired by the Job Worker
- Credit Card Service (CreditCardService.java) - This is a simple, mock representation of a Credit Card Service



Jobs are pushed to a queue in Zeebe whenever a **Process Instance** reaches a **Service Task**. The existing *Payment Application* connects to a Camunda 8 Cluster and polls for jobs of the type *chargeCreditCard*.

Import Payment Application

Import the *Payment Application* into your Java IDE by following the steps below:

1. Download the *Payment Application* from this GitHub Repository and save it to your local filesystem
2. Open the *Payment Application* in your IDE as a Maven Project.
3. The *Payment Application* is a Java Maven project; PaymentWorker.java is the main class of the application.

Add Connection Properties

In order to connect the *Payment Application* to your Test Environment, you will need to provide the relevant Client Credentials

1. Open PaymentApplication.java in your IDE and update the Zeebe Client Credentials to reflect your Test Environment.

Client Credentials

//Zeebe Client Credentials

private static final String ZEEBE_ADDRESS = "[ZEEBE_ADDRESS]";

private static final String ZEEBE_CLIENT_ID = "[ZEEBE_CLIENT_ID]";

private static final String ZEEBE_CLIENT_SECRET = "[ZEEBE_CLIENT_SECRET]";

private static final String ZEEBE_AUTHORIZATION_SERVER_URL = "[ZEEBE_AUTHORIZATION_SERVER_URL]";

private static final String ZEEBE_TOKEN_AUDIENCE = "[ZEEBE_TOKEN_AUDIENCE]";

Start the Payment Application

Finally, we can start the Payment Application using the following:

1. Launch PaymentApplication.java as a Java Application from your IDE
2. If everything has been configured successfully then you should see output similar to the below:

2023-08-01 12:50:14 INFO PaymentApplication:50 - Zeebe Client Connected to: TopologyImpl{...}

2023-08-01 12:50:14 INFO PaymentApplication:58 - Payment Worker: Connected to Cluster

2023-08-01 12:50:14 INFO PaymentApplication:59 - Payment Worker: Processing Jobs for the next 60 seconds

Start Payment Process

We will now need to execute our *Payment Process* for the first time; this is also known as starting a **Process Instance**.

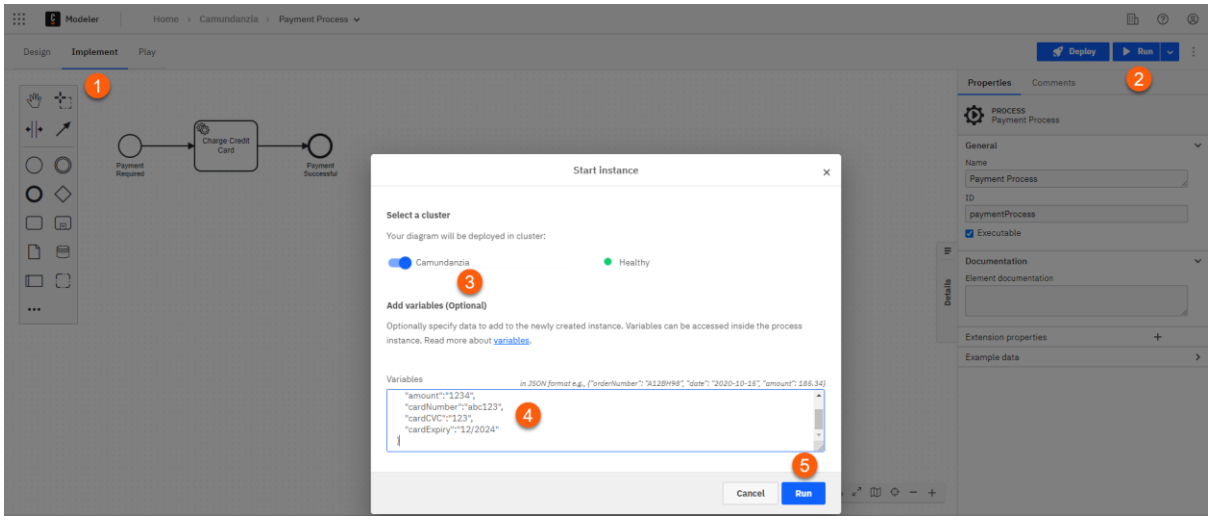
Start a Process Instance

Start an instance of the *Payment Process* by following the steps below:

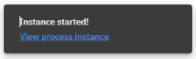
1. Select the **Implement** view if it has not already been selected
2. Click on the **Run** button in the top right corner of the **Modeler**
3. Ensure that the *Camundanzia* **Cluster** is selected from the dialog box that appears.
4. We need to start the process with some Credit Card details; enter the following into the **Add Variables** section

```
{
  "reference":"CMDZ000123",
  "amount":"1234",
  "cardNumber":"abc123",
  "cardCVC":"123",
  "cardExpiry":"12/2024"
}
```

5. Click the **Run** button to start a new instance of *Payment Process* using the *Camundanzia* **Cluster**



6. An **Instance Started!** message will be displayed at the bottom of the **Modeler** to show that this was successful.



Start the Payment Worker

Start an instance of the *Payment Worker* by following the steps below:

1. Launch PaymentApplication.java as a Java Application from your IDE
2. If everything has been configured successfully then you should see output similar to the below:

2023-08-01 12:55:25 INFO PaymentApplication:50 - Zeebe Client Connected to: TopologyImpl{...}

2023-08-01 12:55:25 INFO PaymentApplication:58 - Payment Worker: Connected to Cluster

2023-08-01 12:55:25 INFO PaymentApplication:59 - Payment Worker: Processing Jobs for the next 60 seconds

2023-08-01 12:55:26 INFO PaymentApplication:25 - Starting Transaction: CMDZ000123

2023-08-01 12:55:26 INFO PaymentApplication:26 - Card Number: abc123

2023-08-01 12:55:26 INFO PaymentApplication:27 - Card Expiry Date: 12/2024

2023-08-01 12:55:26 INFO PaymentApplication:28 - Card CVC: 123

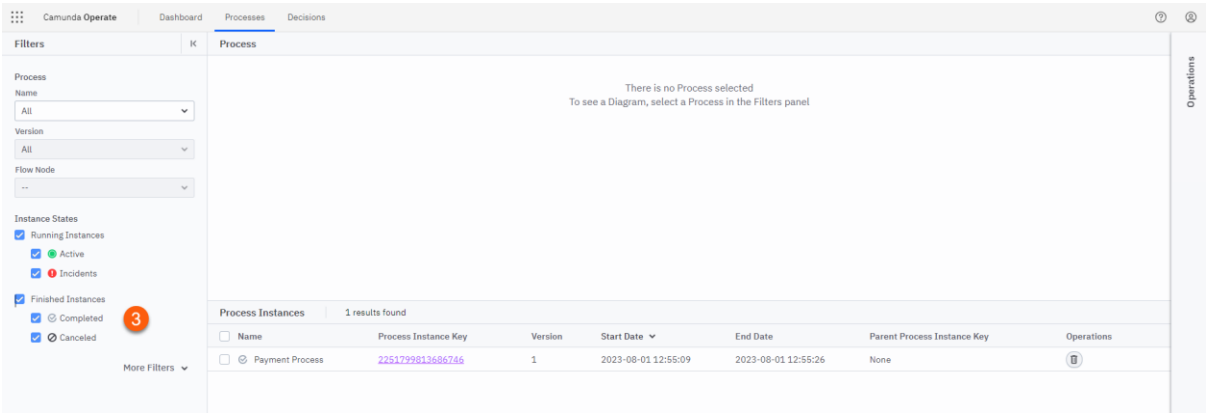
2023-08-01 12:55:26 INFO PaymentApplication:29 - Amount: 1234

2023-08-01 12:55:26 INFO PaymentApplication:43 - Successful Transaction: 1690890926105

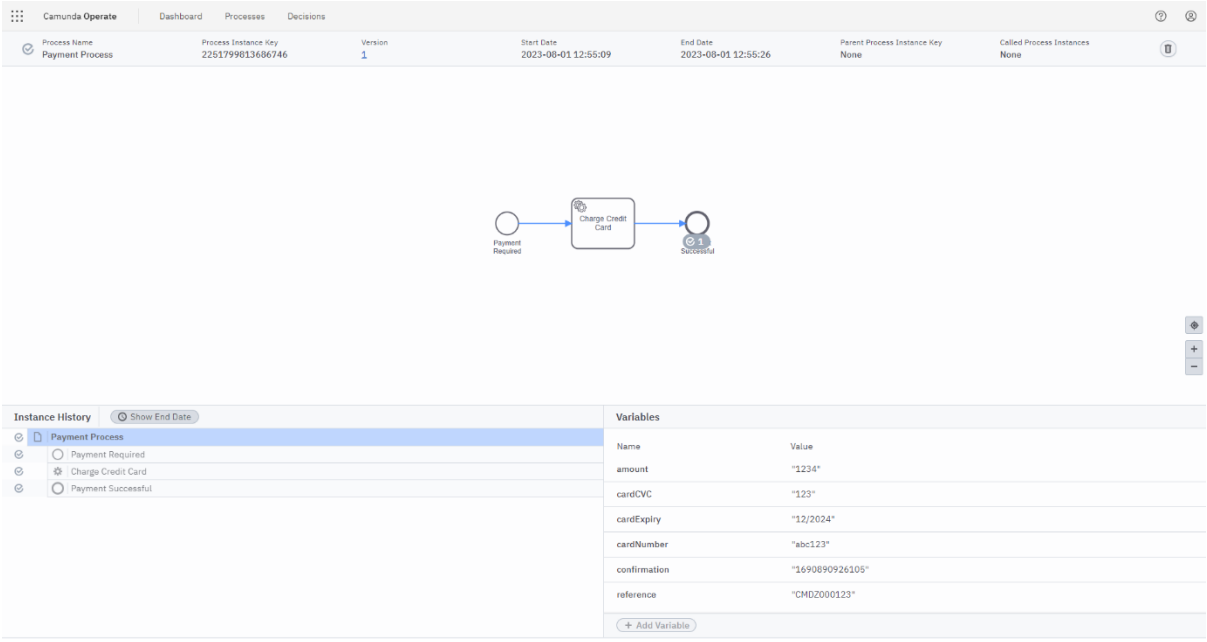
View Process Instances

We can now view the **Process Instances** in **Operate** by following the steps below:

1. Launch **Operate** by following the steps in the previous lesson. Alternatively, refresh the window if you still have **Operate** open.
2. Click on *Payment Process* from the **Process Instances by Name** panel on the left-hand side of the screen
3. Check the **Finished Instances** filter on the left hand side of the screen to show completed process instances



4. Click on the **Process Instance Key** which now appears in the bottom part of the screen; this will open up the **Process Instance View**



5. We can now see that the process completed with the *Payment Successful* event and that the Credit Card details have been populated.

Handle Incidents in Operate

In this lesson, we will see how Camunda 8 provides a generic mechanism for catching and handling exceptions that occur during process execution; this mechanism is known as an Incident.

What is an Incident?

In Camunda 8, an incident represents a problem in process execution. This means a process instance is stuck at a particular point, and requires user interaction to resolve the problem.

Incidents are created in different situations, including the following:

- A job is failed and it has no retries left.
- An input or output variable mapping can't be applied.
- A condition can't be evaluated.
- A decision can't be evaluated.

Handling Incidents in Operate

The CreditCardService in our *Payment Application* will throw a checked exception if error is provided instead of a valid credit card number.

```
private static final String CARD_NUMBER_ERROR = "error";


//Simulate an Exception in the Credit Card Service
 if(cardNumber.equalsIgnoreCase(CARD_NUMBER_ERROR)){
    throw new CreditCardServiceException("Credit Card Service: Internal Error");
 }
```

As this CreditCardServiceException is not being caught in the CreditCardServiceHandler it will be automatically handled by the Job Worker.

```
@Override
public void handle(JobClient client, ActivatedJob job) throws CreditCardServiceException,
InvalidCreditCardException {


 //Obtain the Process Variables
```

```
final Map<String, Object> inputVariables = job.getVariablesAsMap();
final String reference = (String) inputVariables.get(VARIABLE_REFERENCE);
```

Specifically, the Worker will issue a FailJob RPC Command instead of the expected CompleteJob RPC Command.

Start a Process Instance with an Incident

We can see this in action by following the steps below:

1. Start an instance of *Payment Process* from the Web Modeler; enter the following into the **Variables** section of the dialog box when prompted.

```
{
  "reference":"CMDZ000123",
  "amount":"1234",
  "cardNumber":"error",
  "cardCVC":"123",
  "cardExpiry":"12/2024"
}
```

2. Launch PaymentApplication.java as a Java Application from your IDE
3. If everything has been configured successfully then you should see output similar to the below:

```
2023-08-01 13:02:05 INFO  PaymentApplication:50 - Zeebe Client Connected to: TopologyImpl{...}
2023-08-01 13:02:05 INFO  PaymentApplication:58 - Payment Worker: Connected to Cluster
2023-08-01 13:02:05 INFO  PaymentApplication:59 - Payment Worker: Processing Jobs for the next 60 seconds
2023-08-01 13:02:06 INFO  PaymentApplication:25 - Starting Transaction: CMDZ000123
2023-08-01 13:02:06 INFO  PaymentApplication:26 - Card Number: error
2023-08-01 13:02:06 INFO  PaymentApplication:27 - Card Expiry Date: 12/2024
2023-08-01 13:02:06 INFO  PaymentApplication:28 - Card CVC: 123
2023-08-01 13:02:06 INFO  PaymentApplication:29 - Amount: 1234
2023-08-01 13:02:06 WARN  worker:46 - Worker default failed to handle job with key 2251799813687097 of type chargeCreditCard, sending fail command to broker
```

com.camunda.academy.service.CreditCardServiceException: Credit Card Service: Internal Error

at com.camunda.academy.service.CreditCardService.chargeCreditCard(CreditCardService.java:33) ~[classes/:?]

at com.camunda.academy.handler.CreditCardServiceHandler.handle(CreditCardServiceHandler.java:40) ~[classes/:?]

at io.camunda.zeebe.client.impl.worker.JobRunnableFactory.executeJob(JobRunnableFactory.java:44) ~[zeebe-client-java-8.2.10.jar:8.2.10]

at io.camunda.zeebe.client.impl.worker.JobRunnableFactory.lambda$create$0(JobRunnableFactory.java:39) ~[zeebe-client-java-8.2.10.jar:8.2.10]

at java.util.concurrent.Executors$RunnableAdapter.call(Executors.java:515) [?:?]

at java.util.concurrent.FutureTask.run(FutureTask.java:264) [?:?]

at java.util.concurrent.ScheduledThreadPoolExecutor$ScheduledFutureTask.run(ScheduledThreadPoolExecutor.java:304) [?:?]

at java.util.concurrent.ThreadPoolExecutor.runWorker(ThreadPoolExecutor.java:1128) [?:?]

at java.util.concurrent.ThreadPoolExecutor$Worker.run(ThreadPoolExecutor.java:628) [?:?]

at java.lang.Thread.run(Thread.java:834) [?:?]

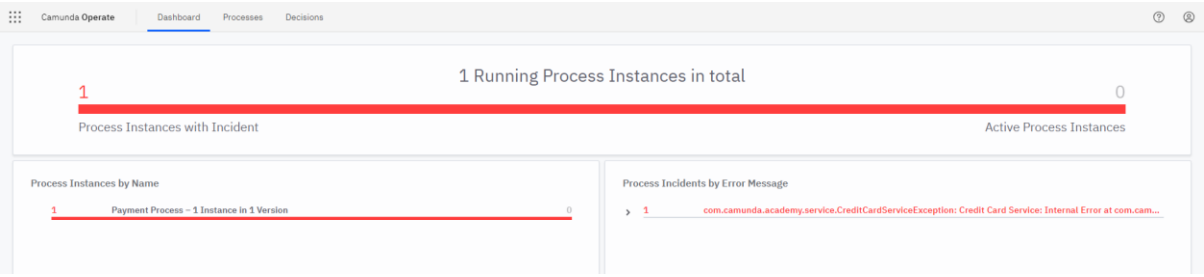
Inspect the Incident in Operate

We can view the Incident that has been created in Operate by following the steps below:

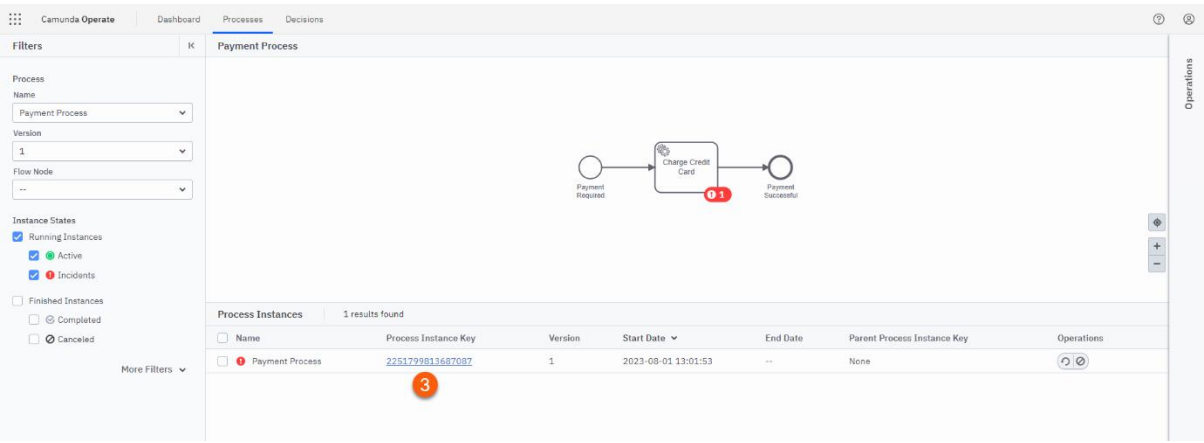
Handling Incidents in Operate

Operate is a great tool which can be used by a **Process Operator** to identify and resolve Incidents.

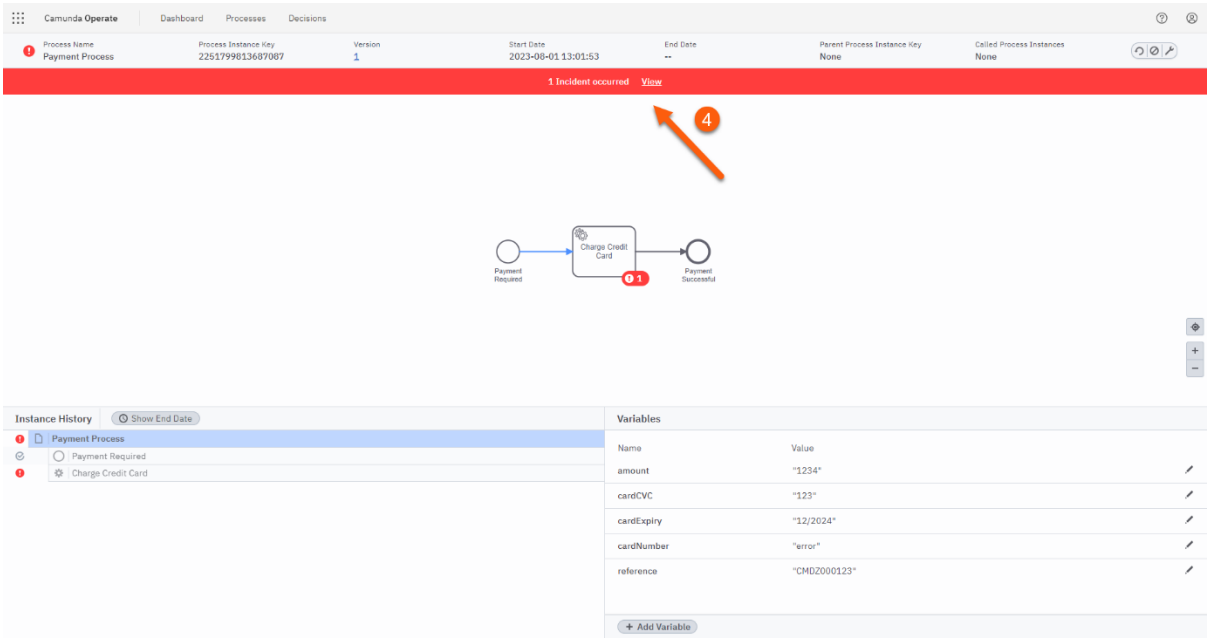
1. Launch **Operate** by following the steps in the previous lesson. Alternatively, refresh the window if you still have **Operate** open.
2. Click on *Payment Process* from the **Process Instances by Name** panel on the left-hand side of the screen



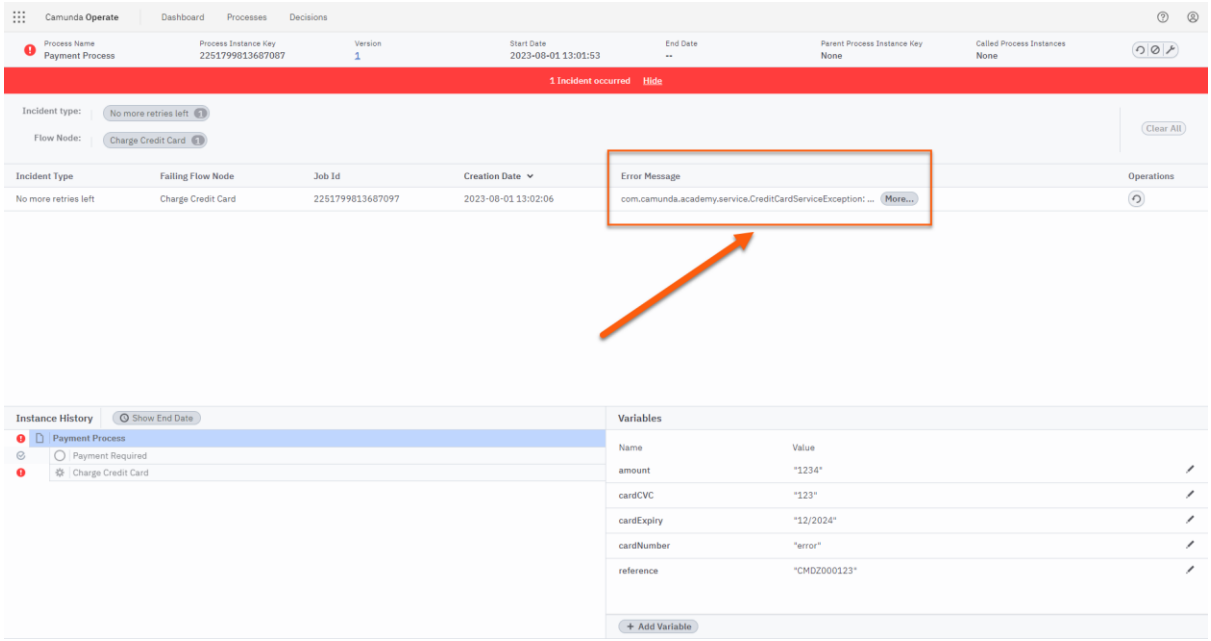
3. Click the **Process Instance Key** in the bottom panel to be taken to the process instance details



4. Operate provides several options for viewing details of the Incident; in the example we will simply click on the **View** link in the **Incident Banner**.



5. Operate will now provide further details of the Incident as well as several actions which can be used to resolve the Incident.



Resolving the Incident in Operate

We can now resolve the Incident by following the steps below:

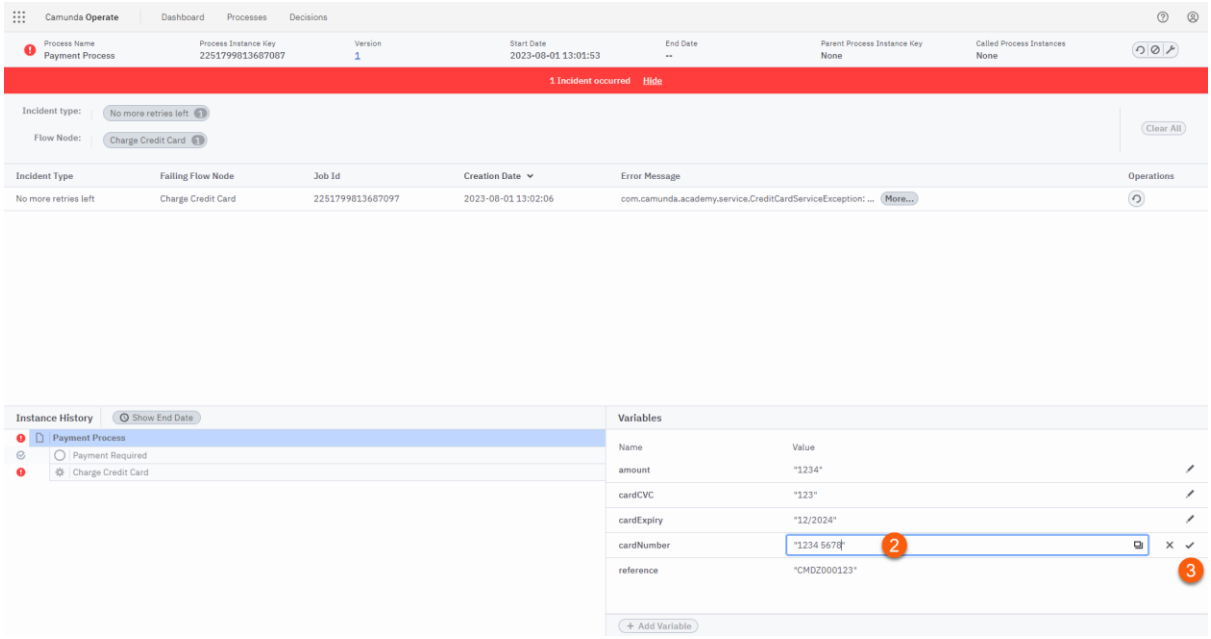
1. Identify the problem
2. Resolve the problem
3. Retry the process execution

If the problem still exists after following these steps, a new Incident will be created in Operate.

Modifying the Process Instance in Operate

We know that this Incident was caused by providing an invalid credit card number to the CreditCardService, we can resolve this problem by following the steps below:

1. From the Process Instance View in Operate, select the **Edit** icon next to the cardNumber variable in the bottom right hand corner of the screen.
2. Edit the existing value of error and replace this with a valid credit card number e.g. 1234 5678
3. Click the **Save Variable** icon next to the cardNumber variable in the bottom right hand corner of the screen.



Why is the Incident not resolved?

You may be wondering why the Incident is still being shown against the Process Instance now that the error has been resolved. This is because the number of retries associated with the **Job** have been decremented to zero.

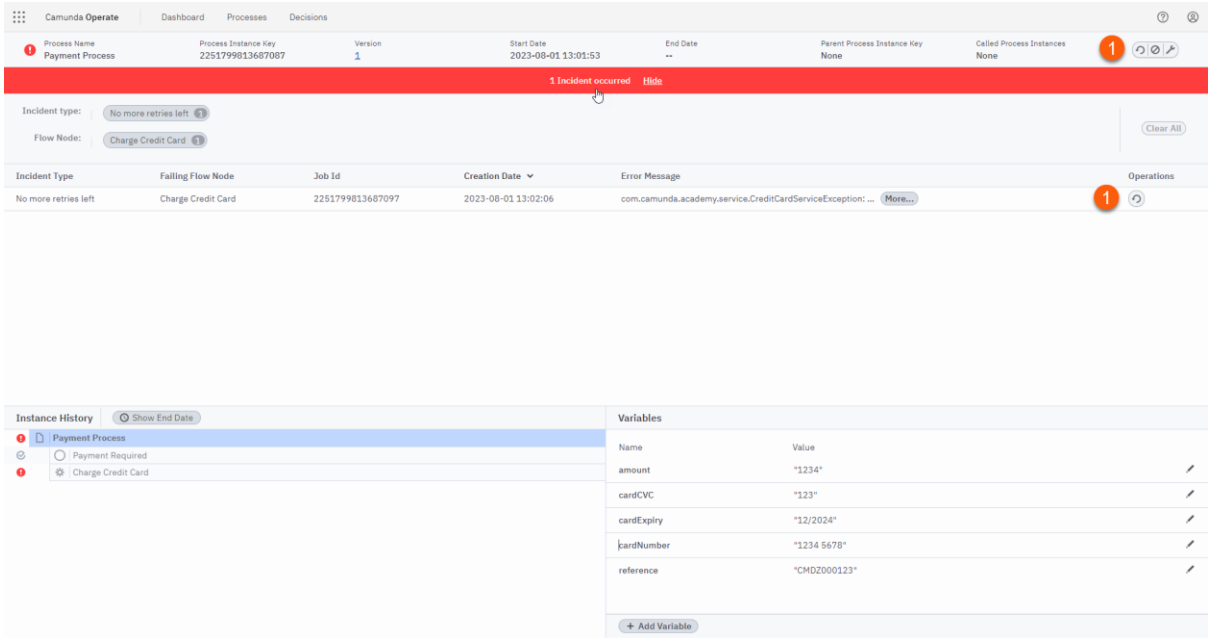
A **Job** always has a defined number of retries, by default, each **Job** is configured to retry 3 times in the event of an error. After this, an Incident is created and no further retries will be attempted. The next lessons will demonstrate how this can be configured.

The Incident will only be resolved once you have attempted to execute the Job again.

Retrying the Job in Operate

Now that you have resolved the original issue, you can retry the Job in Operate to determine whether our modifications have been successful.

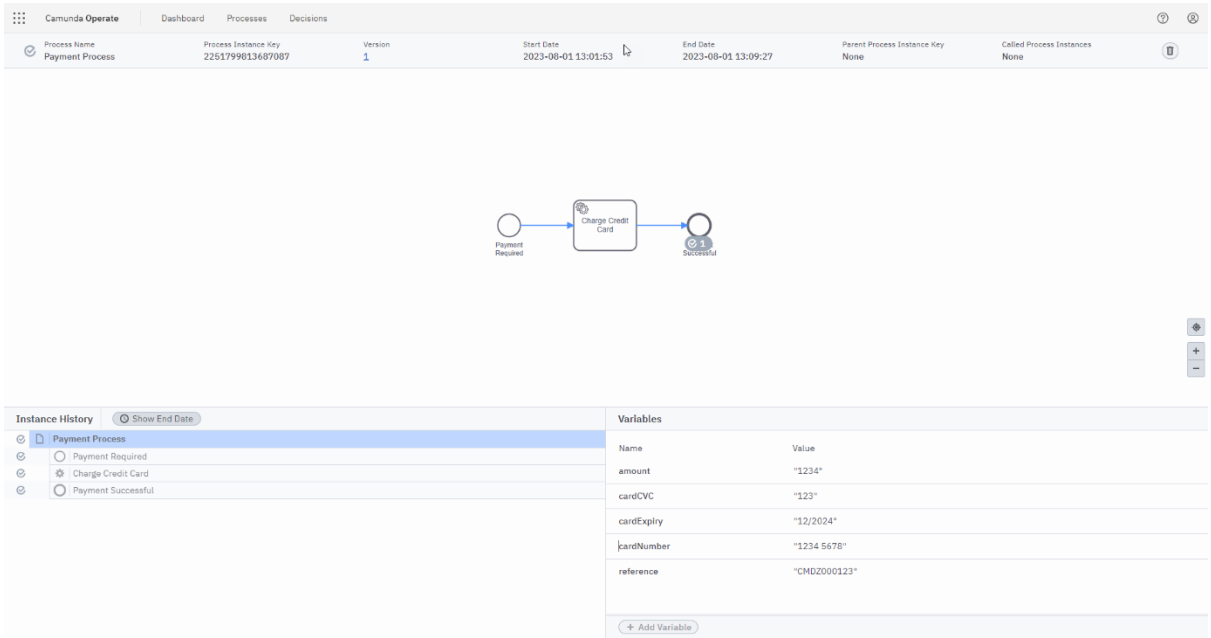
1. From the **View** link in the **Incident Banner** click on either the **Retry Incident** or **Retry Instance** icon to retry the specific Job that failed or all failed Jobs in the Process Instance.



2. By clicking **Retry Job** you are actually incrementing the number of retries available on the **Job**; as the number of retries is now 1, the **Job** will become eligible for execution again.

3. You should now see that the Process Instance has completed successfully.

4. You may need to launch PaymentApplication.java as a Java Application from your IDE if it has already timed out.



Handle Incidents in the Job Worker

In the last lesson, we saw how a **Process Operator** is able to identify and resolve **Incidents** using Operate. This provides a generic mechanism for catching and handling all exceptions that occur during process execution.

However, it would quickly become tedious for a **Process Operator** to restart hundreds or thousands of process instances every time a temporary error occurred e.g. loss of network connectivity. Fortunately, it is possible to configure a **Job Worker** to re-execute the task automatically.

Retrying a Service Task Automatically

As you saw in the previous lesson, by default, a Service Task will be retried three times in the event of an error. It is also possible to configure the number of retries directly in the Service Task or to enhance a Job Worker with additional retry logic.

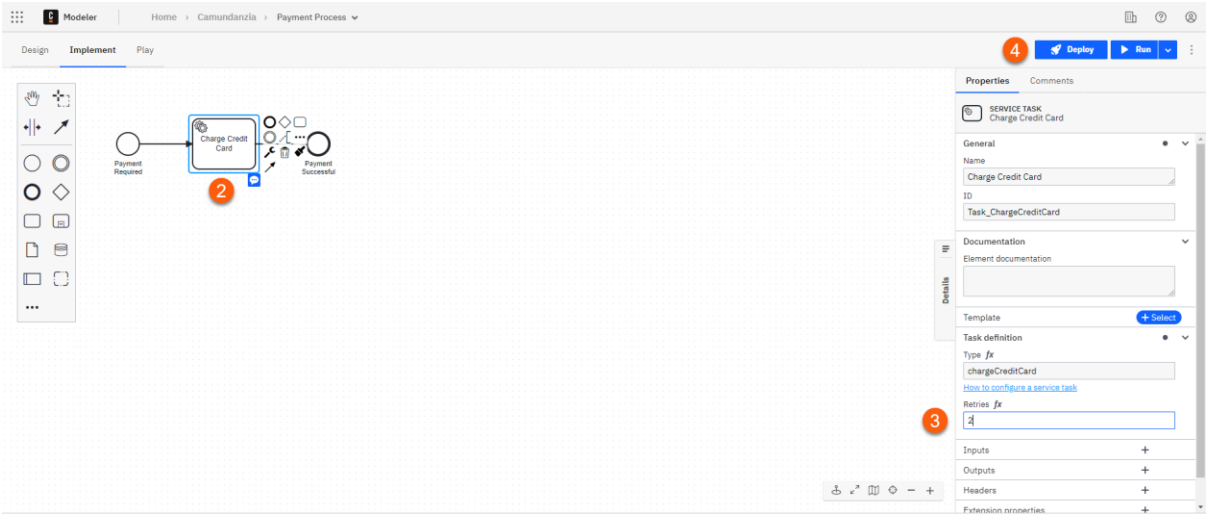
However, a more common approach is to obtain the number of retries for the Service Task and to update this number through the FailJob RPC Command.

We will now update our *Payment Process* and *Payment Application* so that failed jobs are automatically retried.

Update Payment Process

Let's start by updating the *Payment Process* so that *Charge Credit Card* is retried twice in the event of an error.

1. From the *Camundanzia* project in the **Modeler**, open the *Payment Process*
2. Click on the *Charge Credit Card* Service Task in the process model to display the **Properties Panel**
3. Enter 2 into the **Retries** field
4. Deploy the updated process into your cluster



Update Payment Application

We can now update the **Job Worker** so that it obtains the specified number of retries should an error occur during processing.

1. Open the CreditCardServiceHandler.java in your Java IDE.
2. Surround the call to the CreditCardService in the handle(JobClient client, ActivatedJob job) method with a try / catch block.

```
try {


    //Charge the Credit Card

    final String confirmation = creditCardService.chargeCreditCard(reference, amount, cardNumber, cardExpiry, cardCVC);


    //Build the Output Process Variables

    final Map<String, Object> outputVariables = new HashMap<String, Object>();

    outputVariables.put(VARIABLE_CONFIRMATION, confirmation);


    //Complete the Job

    client.newCompleteCommand(job.getKey()).variables(outputVariables).send().join();


} catch (CreditCardServiceException e) {

    // TODO Auto-generated catch block
```

```
    e.printStackTrace();
}
```

3. In the catch block, retrieve the remaining number of retries from the Job using the job.getRetries() method
4. In the catch block, use the client.newFailCommand() method to explicitly indicate failure of the Job to Zeebe. This command should include the updated number of retries as well as a retry back-off of 20 seconds.

Completing and Failing Jobs

When a Job Worker is able to successfully complete its work, it will send a CompleteJobCommand. Equally, when a Job Worker cannot successfully complete its work, it will send a FailJobCommand. The FailJobCommand includes the number of remaining retries which is set by the Job Worker.

- If the remaining retries is greater than zero, the **Job** is retried and reassigned.
- If the remaining retries is zero or negative, an **Incident** is raised and the **Job** is not retried until the **Incident** is resolved.

Job Retry Back Off

When failing a Job, it is also possible to specify a retry back-off. This back-off allows the Job to wait for a specified amount of time before it is picked up for processing again.

Configuring a retry back-off can be useful for when a Job Worker is communicating with an external system. If the external system is unreliable, the retry back-off can be used to wait for an appropriate period of time before retrying. This may allow the external system to recover before all Job retries have been exhausted.

5. The final version of the handle(JobClient client, ActivatedJob job) method should look similar to the below once you have made the relevant changes.

```
@Override
public void handle(JobClient client, ActivatedJob job) throws CreditCardServiceException,
InvalidCreditCardException {

    //Obtain the Process Variables
    final Map<String, Object> inputVariables = job.getVariablesAsMap();
    final String reference = (String) inputVariables.get(VARIABLE_REFERENCE);
    final String amount = (String) inputVariables.get(VARIABLE_AMOUNT);
```

```
    final String cardNumber = (String) inputVariables.get(VARIABLE_CARD_NUMBER);
    final String cardExpiry = (String) inputVariables.get(VARIABLE_CARD_EXPIRY);
    final String cardCVC =  (String) inputVariables.get(VARIABLE_CARD_CVC);

    try {

      //Charge the Credit Card
      final String confirmation = creditCardService.chargeCreditCard(reference, amount,
cardNumber, cardExpiry, cardCVC);

      //Build the Output Process Variables
      final Map<String, Object> outputVariables = new HashMap<String, Object>();
      outputVariables.put(VARIABLE_CONFIRMATION, confirmation);

      //Complete the Job
      client.newCompleteCommand(job.getKey()).variables(outputVariables).send().join();

    } catch (CreditCardServiceException e) {
      int retries = job.getRetries();
      client.newFailCommand(job.getKey())
            .retries(retries-1)
            .retryBackoff(Duration.ofSeconds(20))
            .errorMessage(e.getMessage())
            .send()
            .join();
    }
}
```

Start the Payment Process and Payment Worker

Start a new instance of the *Payment Process* from the **Modeler** by following the steps below:

1. Select the **Implement** view if it has not already been selected
2. Click on the **Run** button in the top right corner of the **Modeler**
3. Ensure that the *Camundanzia* **Cluster** is selected from the dialog box that appears.
4. We need to start the process with some Credit Card details; enter the following into the **Add Variables** section

```
{
  "reference":"CMDZ000123",
  "amount":"1234",
  "cardNumber":"error",
  "cardCVC":"123",
  "cardExpiry":"12/2024"
}
```

5. Launch PaymentApplication.java as a Java Application from your IDE
6. If everything has been configured successfully then you should see output similar to the below:

```
2023-08-01 01:48:45 INFO  PaymentApplication:51 - Connected to: TopologyImpl
2023-08-01 13:18:26 INFO  PaymentApplication:58 - Payment Worker: Connected to Cluster
2023-08-01 13:18:26 INFO  PaymentApplication:59 - Payment Worker: Processing Jobs for the next 60 seconds
2023-08-01 13:18:26 INFO  PaymentApplication:25 - Starting Transaction: CMDZ000123
2023-08-01 13:18:26 INFO  PaymentApplication:26 - Card Number: error
2023-08-01 13:18:26 INFO  PaymentApplication:27 - Card Expiry Date: 12/2024
2023-08-01 13:18:26 INFO  PaymentApplication:28 - Card CVC: 123
2023-08-01 13:18:26 INFO  PaymentApplication:29 - Amount: 1234
2023-08-01 13:18:46 INFO  PaymentApplication:25 - Starting Transaction: CMDZ000123
2023-08-01 13:18:46 INFO  PaymentApplication:26 - Card Number: error
2023-08-01 13:18:46 INFO  PaymentApplication:27 - Card Expiry Date: 12/2024
```

2023-08-01 13:18:46 INFO PaymentApplication:28 - Card CVC: 123

2023-08-01 13:18:46 INFO PaymentApplication:29 - Amount: 1234

7. This time, you will see that the Job is processed twice (as per the configuration in the Process Definition) and will fail for the same reasons as before.
8. You can follow the steps from the previous lesson to resolve the incident and allow the process instance to complete successfully.

Handling BPMN Errors

Overview

In the last two lessons, you saw how Camunda 8 provides a generic mechanism for catching and handling unexpected exceptions that occur during process execution; this mechanism is known as an Incident.

In this lesson, you will see how the BPMN Error Event can be used to catch and gracefully handle expected errors that occur during process execution.

What is a BPMN Error Event?

The BPMN 2.0 Error Event allows you to explicitly model errors and how they are handled in your process

Handle BPMN Errors

When modeling business processes, you will often find that you need to handle expected deviations from the Happy Path in your process models. In these situations, rather than creating an Incident, it is better to handle and resolve the error within the process.

In our *Payment Process* example, we can assume that the **Partner** may enter invalid credit card details at any time during process execution. In this situation, rather than creating an Incident which will need to be resolved by a **Process Operator**, we can handle the invalid credit card details using a **BPMN Error Event**.

Why not use client-side validation?

In a real-life scenario, it is much more likely that client-side validation would be applied to any forms that allow the user to enter their credit card details. We are simply using this as an example to demonstrate the use of the BPMN Error Event.

In the next lessons, we will update the *Payment Application* to detect and recover from expected errors without the need to create an Incident. Specifically, we will:

1. Modify the *Payment Worker* to **throw** a BPMN Error when the **Partner** has entered invalid card details
2. Modify the *Payment Process* to **catch** a BPMN Error and route it to a resolution task
3. Test the *Payment Application* to ensure that the error is now being handled as expected

Throw the Error (Job Worker)

Let's start by updating the CreditCardServiceHandler so that it throws a BPMN Error when an InvalidCreditCardException is encountered.

Update Payment Application

Update the *Payment Application* to throw a BPMN Error by following the steps below:

1. Open the CreditCardService.java in your Java IDE and review the following snippet of code

   ```
   private static final int CARD_EXPIRY_DATE_LENGTH = 7;


   //Simulate an Exception in the Credit Card Expiry Date
   if(cardExpiryDate.length() != CARD_EXPIRY_DATE_LENGTH){
      throw new InvalidCreditCardException("Credit Card Service: Invalid Expiry Date");
   }
   ```

   We can see that an InvalidCreditCardException will be thrown if the length of the Credit Card Expiry Date is incorrect.

   Open the CreditCardServiceHandler.java in your Java IDE.

   Add an additional catch to the existing try / catch block

   ```
   try {


      //Charge the Credit Card
      final String confirmation = creditCardService.chargeCreditCard(reference, amount, cardNumber, cardExpiry, cardCVC);


      //Build the Output Process Variables
   ```

```
    final Map<String, Object> outputVariables = new HashMap<String, Object>();
    outputVariables.put(VARIABLE_CONFIRMATION, confirmation);

    //Complete the Job

  client.newCompleteCommand(job.getKey()).variables(outputVariables).send().join();

  } catch (CreditCardServiceException e) {
    int retries = job.getRetries();
    client.newFailCommand(job.getKey())
        .retries(retries-1)
        .retryBackoff(Duration.ofSeconds(20))
        .errorMessage(e.getMessage())
        .send()
        .join();
  } catch (InvalidCreditCardException e) {
    //TODO
    e.printStackTrace();
  }
```

2. In the catch block, notify the Zeebe Engine that a BPMN Error needs to be thrown by issuing the ThrowError RPC Command

```
client.newThrowErrorCommand(job.getKey())
    .errorCode("cardExpiryDateError")
    .send()
    .join();
```

ThrowError Command

ThrowError reports a business error (i.e. non-technical) that occurs while processing a job.

The error is handled in the process by a **BPMN Error Catch Event**. If there is no **BPMN Error Catch Event** with the specified *errorCode*, an Incident is raised instead.

3. The final version of the handle(JobClient client, ActivatedJob job) method should look similar to the below once you have made the required changes.

```
//BPMN Errors

private static final String BPMN_ERROR_INVALID_CARD_EXPIRY_DATE =
"cardExpiryDateError";


@Override

public void handle(JobClient client, ActivatedJob job) throws
CreditCardServiceException, InvalidCreditCardException {


        //Obtain the Process Variables

        final Map<String, Object> inputVariables = job.getVariablesAsMap();

        final String reference = (String) inputVariables.get(VARIABLE_REFERENCE);

        final String amount = (String) inputVariables.get(VARIABLE_AMOUNT);

        final String cardNumber = (String)
inputVariables.get(VARIABLE_CARD_NUMBER);

        final String cardExpiry = (String) inputVariables.get(VARIABLE_CARD_EXPIRY);

        final String cardCVC =  (String) inputVariables.get(VARIABLE_CARD_CVC);


        try {


                        //Charge the Credit Card

                        final String confirmation =
creditCardService.chargeCreditCard(reference, amount, cardNumber, cardExpiry,
cardCVC);


                        //Build the Output Process Variables

                        final Map<String, Object> outputVariables = new
HashMap<String, Object>();

                        outputVariables.put(VARIABLE_CONFIRMATION, confirmation);
```

```
                    //Complete the Job

            client.newCompleteCommand(job.getKey()).variables(outputVariables).send()
.join();


                } catch (CreditCardServiceException e) {
            int retries = job.getRetries();
            client.newFailCommand(job.getKey())
                .retries(retries-1)
                .retryBackoff(Duration.ofSeconds(20))
                .errorMessage(e.getMessage())
                .send()
                .join();
            } catch (InvalidCreditCardException e) {
                client.newThrowErrorCommand(job.getKey())
                        .errorCode(BPMN_ERROR_INVALID_CARD_EXPIRY_DATE)
                        .send()
                        .join();
            }
}
```

Catch the Error (Process Model)

We now need to update the *Payment Process* so that it catches the BPMN Error that is thrown by the *Payment Worker*.

User Task

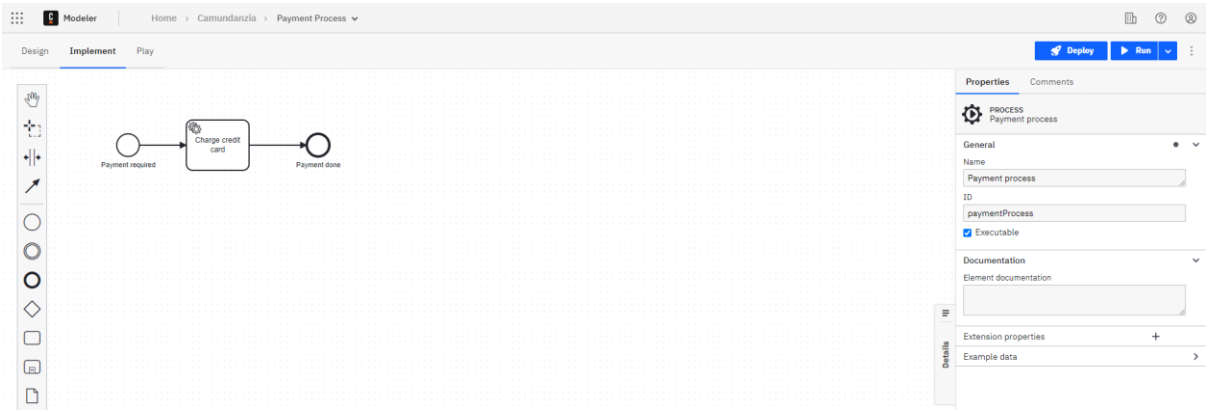
The **Process Operator** will be able to review and update the credit card details that have been provided. If the error has been resolved then the credit card will be charged. Otherwise, the process will end.

Update Payment Process

We need to update the *Payment Process* so that a BPMN Error Boundary Event is attached to the *Charge Credit Card* task.

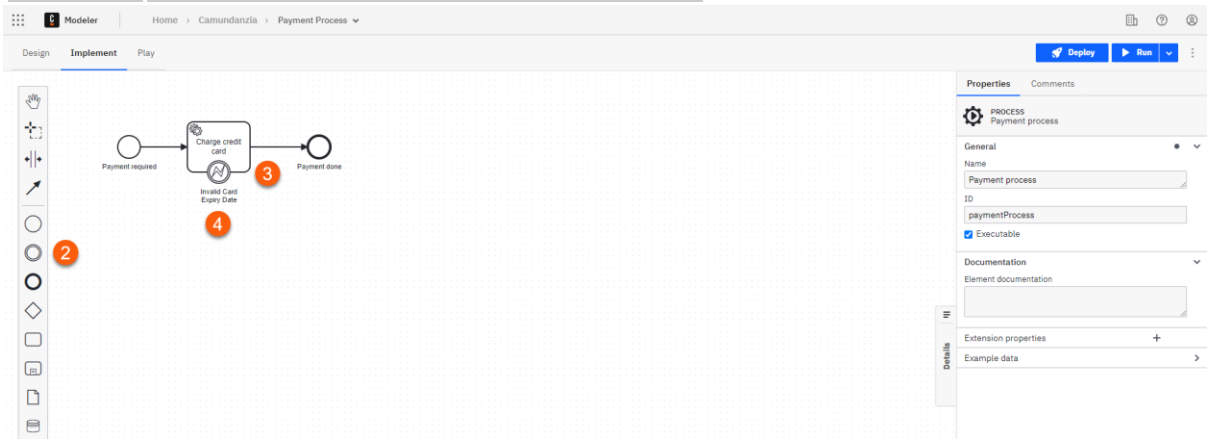
Model the Payment Process

1. From the *Camundanzia* project in the **Modeler**, open the *Payment Process*



2. Drag an **Intermediate / Boundary Event** from the **Palette** and add it to the bottom edge of the *Charge Credit Card* task.
3. Select the Error Event and then click on the **Wrench Icon** in the context menu to choose an **Error Boundary Event**.
4. The Name and ID of the Error Event should be configured as follows:

Name	Value
ID	BoundaryEvent_InvalidCardExpiryDate
Name	Invalid Card Expiry Date



5. Add a new **User Task** beneath the *Charge Credit Card* task with the following configuration:

Name	Value
ID	Task_ReviewCreditCard
Name	Review Credit Card Details

6. Add a new **Exclusive Gateway** to the right of the *Review Credit Card Details* task with the following configuration:

Name	Value
ID	Gateway_Resolved
Name	Resolved?

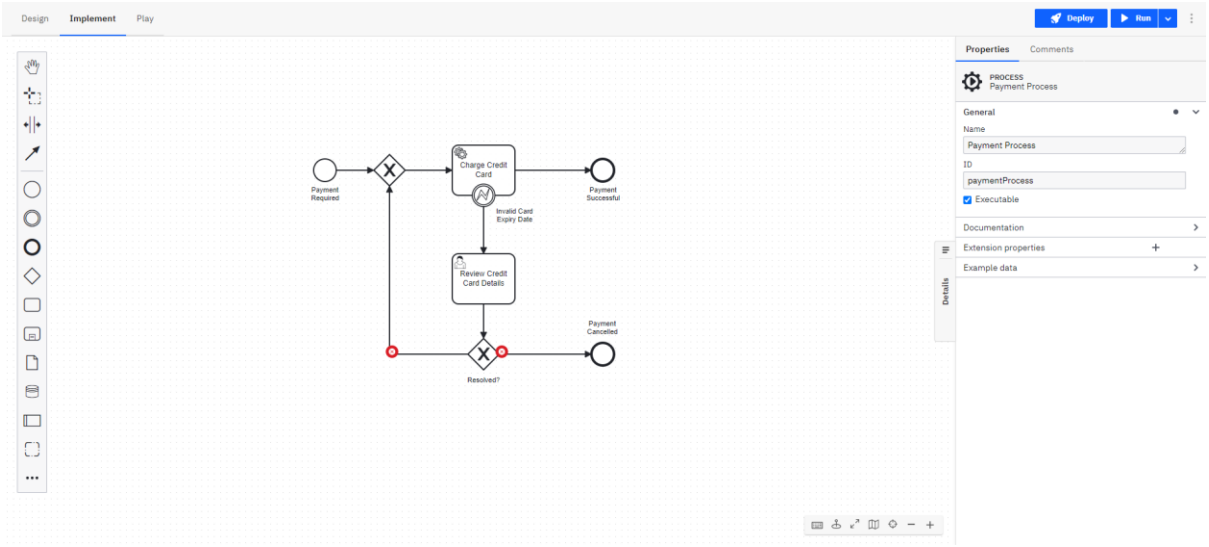
7. Add a new **End Event** to the right of the *Resolved?* gateway with the following configuration:

Name	Value
ID	EndEvent_PaymentCancelled
Name	Payment Cancelled

8. Add a new **Exclusive Gateway** between the *None Start Event* and the *Charge Credit Card* service task. This will merge the *start flow* and the *resolved flow*. Configure it like this:

Name	Value
ID	Gateway_Merge

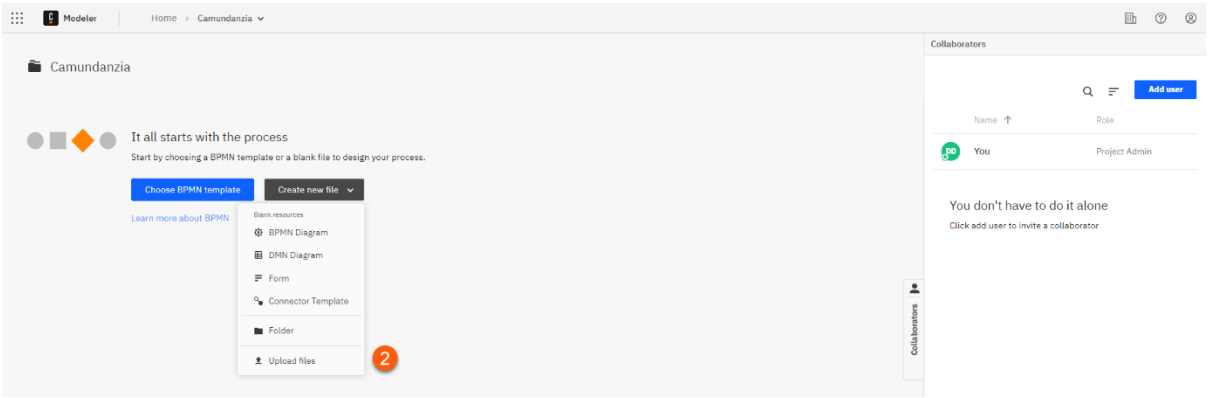
9. Add **Sequence Flows** to the newly added gateway so that your process model resembles the below:



Import the Check Credit Card Details Form

As we have added a **User Task** to the *Payment Process*, we will need a **Form** for a **Process Operator** to interact with the process.

1. Download the *Check Error* form from this GitHub Repository and save it to your local filesystem
2. From the *Camundanzia* project in the **Modeler**, click the **Create new file** button and then **Upload files** from the menu that appears



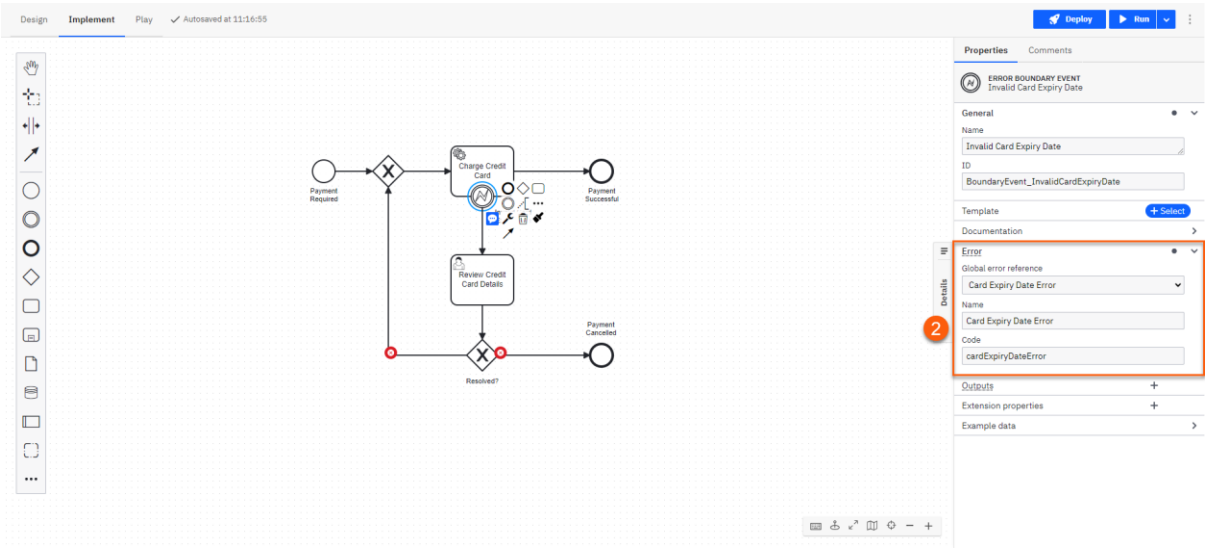
3. From the dialog box that appears, select the checkError.form that you just downloaded. This will import the existing **Form** into our project.
4. Finally, open the **Form** from the **Modeler** and use the **Copy JSON** button to copy the form definition to your clipboard.

Configure the Payment Process

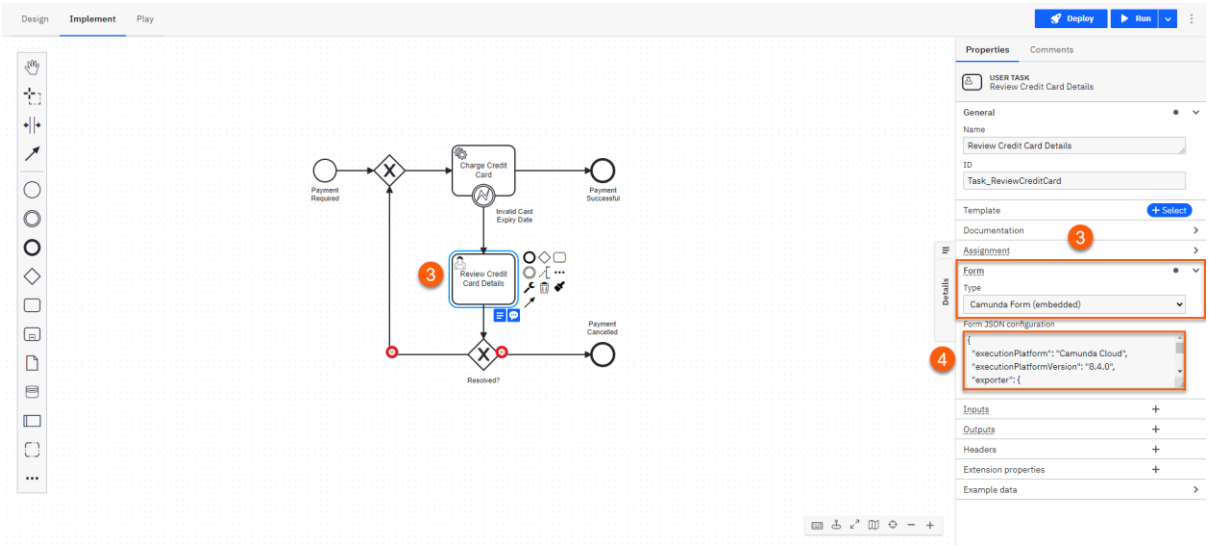
We can now add additional process logic to our *Payment Process* so that it correctly handles the BPMN Error.

1. Click on the **Boundary Error Event** and select **Create new ...** from the **Error** panel on the right hand side of the screen.
2. Configure the **Name** and **Code** as follows:

Label	Value
Name	Card Expiry Date Error
Code	cardExpiryDateError



3. Click on the **User Task** and select **Form** -> **Camunda Form (embedded)**
4. Paste the JSON form definition into the **Form JSON Configuration** field



Form Definition

The Form that has been defined for this User Task will allow a **Process Operator** to review the process variables that have been passed in and will also allow them to indicate whether the credit card details are valid through the *Valid Credit Card?* checkbox.

The *Valid Credit Card?* checkbox will be used by the **Exclusive Gateway** to determine whether the *Charge Credit Card* task should be executed again.

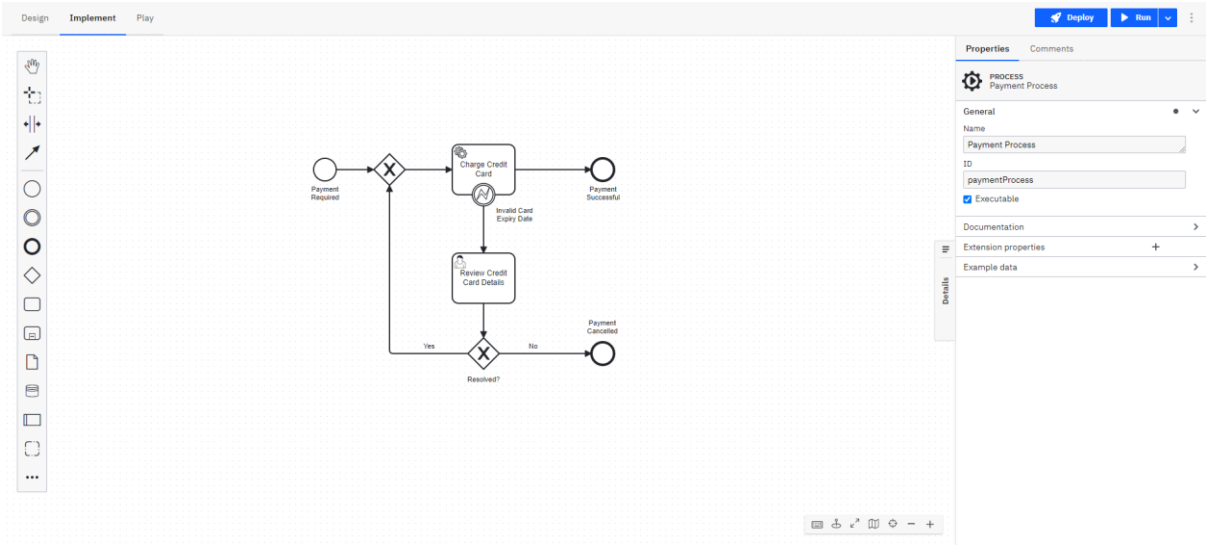
5. Click on the **Sequence Flow** between *Resolved?* and *Payment Cancelled*
6. Configure the **ID**, **Name** and **Condition Expression** as follows:

Name	**Value**
ID	Flow_GatewayResolved-EndEvent_PaymentCancelled
Name	No
Condition Expression	isValidCreditCard = false

7. Click on the **Sequence Flow** between *Resolved?* and the *merge gateway*
8. Configure the **ID**, **Name** and **Condition Expression** as follows:

Name	**Value**
ID	Flow_Gateway_Resolved-Task_ChargeCreditCard
Name	Yes
Condition Expression	isValidCreditCard = true

9. Your *Payment Process* should now look similar to the diagram below:



10. Finally, redeploy your process using the **Modeler**.

Validation

Cleanup Environment

We have now met all of the business requirements that have been defined in our **Scenario**. We are now in a position where we can perform an end-to-end test of our process to ensure that everything is working as expected.

However, as we have been testing our process definition as we have been developing it, it is likely that we have several incomplete process instances still running. Having incomplete (and possibly incorrect) process instances running in our environment may make it more difficult to confirm that everything is working as expected in an end-to-end test.

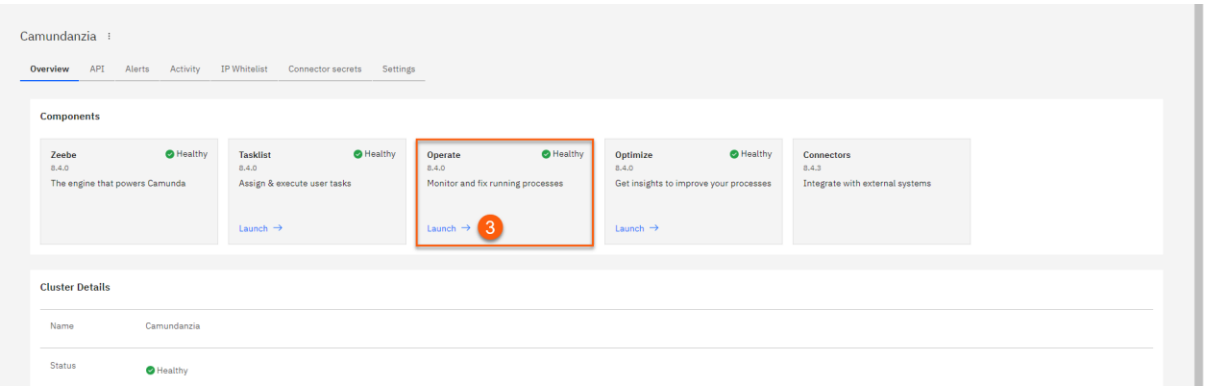
We will now **Cancel** and **Delete** any remaining instances so that we have a completely clean environment in which we can conduct our acceptance tests.

Cancel & Delete Process Instances

Launch Operate

Launch **Operate** by following the steps below:

1. From the **Console**, select **Clusters**
2. Select *Camundanzia* from the **Clusters** page
3. Click the **Launch** link next to **Operate** in the **Cluster Details** page



4. After a few seconds, the **Operate** Dashboard will be displayed

Cancel Process Instances

We can now **Cancel** the process instances in **Operate** by following the steps below:

1. Click on *Payment Process* from the **Instances by Process** panel on the left-hand side of the screen
2. On the bottom left of the screen is a series of filters that will allow you to select *Running* or *Finished* process instances
3. Ensure that the *Running* filters are selected and the *Finished* filters are not selected
4. Then, for each **Process Instance** that is shown in the **Instances** panel, click the **Cancel Instance** button to the right-hand side of the process

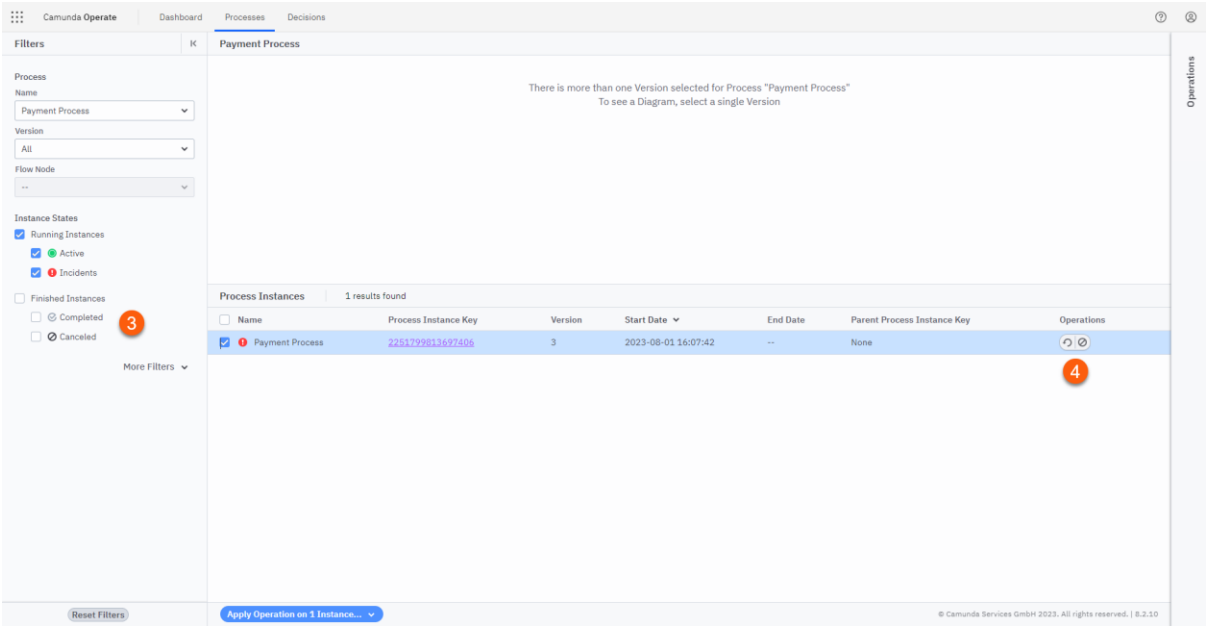
No Running Processes

This section can be skipped if there are no running processes shown in the **Instances** panel.

5. The **Process Instance** will then be cancelled and will disappear from the **Instances** panel

What happened to the Process Instance?

When you **Cancel** a process in Operate it changes to a stopped state. As the process is no longer running it is not picked up by *Running* filter.



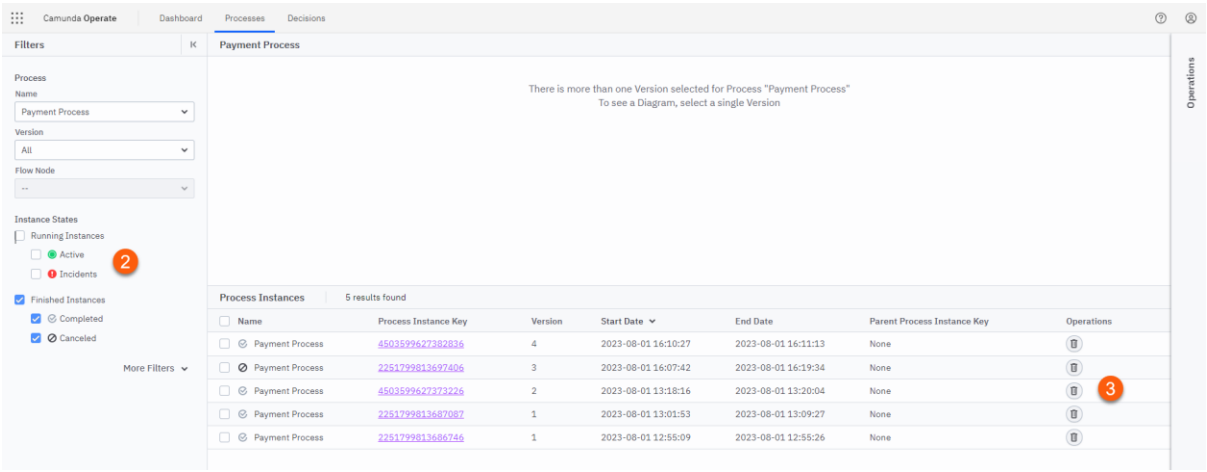
Delete Process Instances

We can now **Delete** the finished process instances in **Operate** by following the steps below:

1. On the bottom left of the screen is a series of filters that will allow you to select *Running* or *Finished* process instances
2. Ensure that the *Finished* filters are selected and the *Running* filters are not selected
3. Then, for each **Process Instance** that is shown in the **Instances** panel, click the **Delete Instance** button to the right-hand side of the process
4. The **Process Instance** will then be deleted and will disappear from the **Instances** panel

What happened to the Process Instance?

When you **Delete** a process in Operate it is completely removed from the **Zeebe Engine**.



Test Process

We are now ready to test the updated *Payment Application*.

Review Acceptance Criteria

Let's start by reviewing the acceptance criteria that was defined earlier; we need to ensure that all of the requirements below can be met by our modifications and understanding.

- When an order fails because the **Credit Card Service** is unavailable
    - A **Process Operator** is able to see the details of the failed order in Camunda
    - A **Process Operator** is able to manually re-trigger the invocation of the **Credit Card Service** for an order
    - The **System** automatically retries the invocation of the **Credit Card Service** twice, with a 20-second delay between requests
- When an order fails because there is an issue with the credit card details which have been provided
    - A **Process Operator** is able to review and modify the incorrect information
    - A **Process Operator** is able to re-submit the order

Test Scenarios

The following scenarios will be required to fully confirm that the acceptance criteria above has been met.

Test Application

Test Scenarios

Four scenarios will be required to confirm that the acceptance criteria above has been met.

Test Scenario	Reference	Amount	Card Number	CVC	Expiry Date	Expected Result
1 - Payment Processed Successfully	CMD000001	1234	12345678	123	12/2024	Process Instance Completes

Test Scenario	Reference	Amount	Card Number	CVC	Expiry Date	Expected Result
2 - Credit Card Service Exception	CMD000002	1234	error	123	12/2024	Incident Created
3 - Credit Card Exception - Cancel	CMD000003	1234	12345678	123	error	Check Credit Card Details
4 - Credit Card Exception - Success	CMD000004	1234	12345678	123	error	Check Credit Card Details

Launch Process Instances

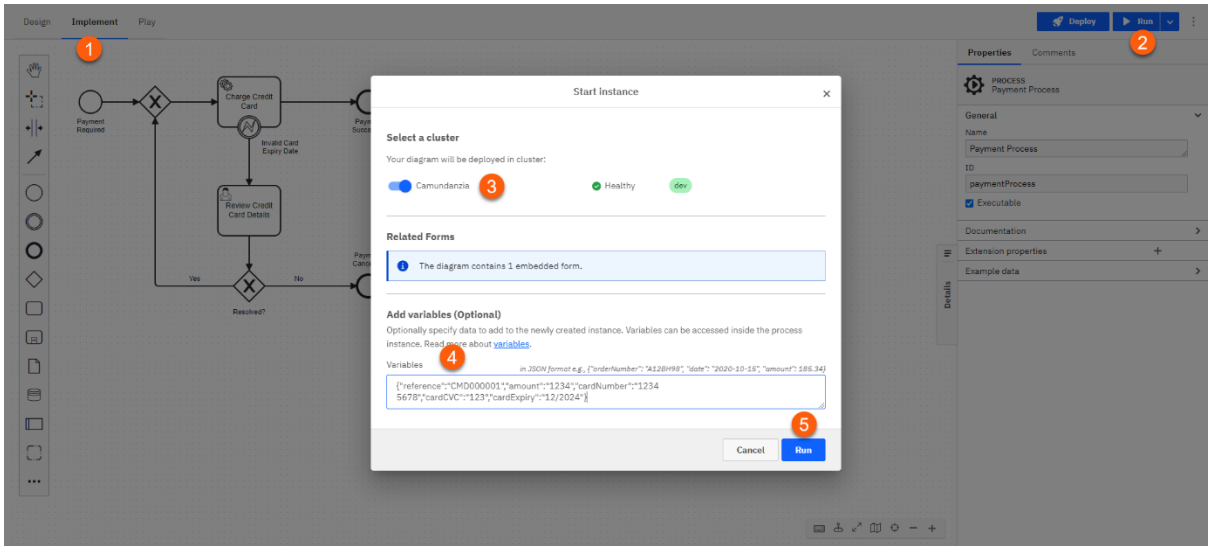
Start four instances of the *Payment Process* by following the steps below:

1. Select the **Implement** view if it has not already been selected
2. Click on the **Run** button in the top right corner of the **Modeler**
3. Ensure that the *Camundanzia* **Cluster** is selected from the dialog box that appears.
4. We need to start each process with the correct credit card details; enter the following JSON string for each instance into the **Add Variables** section

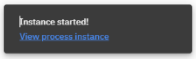
Test Scenario	JSON
1 - Payment Processed	{"reference":"CMD000001","amount":"1234","cardNumber":"12345678","cardCVC":"123","cardExpiry":"12/2024"}

Test Scenario	**JSON**
Successfully	
2 - Credit Card Service Exception	{"reference":"CMD000002","amount":"1234","cardNumber":"error","cardCVC":"123","cardExpiry":"12/2024"}
3 - Credit Card Exception - Cancel	{"reference":"CMD000003","amount":"1234","cardNumber":"12345678","cardCVC":"123","cardExpiry":"error"}
4 - Credit Card Exception - Success	{"reference":"CMD000004","amount":"1234","cardNumber":"12345678","cardCVC":"123","cardExpiry":"error"}

5. Click the **Run** button to start a new instance of *Payment Process* using the *Camundanzia* **Cluster**



6. An **Instance Started!** message will be displayed at the bottom of the **Modeler** to show that this was successful.



Start the Payment Worker

Start an instance of the *Payment Worker* by following the steps below:

1. Launch PaymentApplication.java as a Java Application from your IDE
2. If everything has been configured successfully then you should see output similar to the below:

2023-08-02 10:20:21 INFO PaymentApplication:50 - Zeebe Client Connected to: TopologyImpl{...}

2023-08-02 10:20:21 INFO PaymentApplication:58 - Payment Worker: Connected to Cluster

2023-08-02 10:20:21 INFO PaymentApplication:59 - Payment Worker: Processing Jobs for the next 60 seconds

2023-08-02 10:20:21 INFO PaymentApplication:25 - Starting Transaction: CMD000001

2023-08-02 10:20:21 INFO PaymentApplication:26 - Card Number: 1234 5678

2023-08-02 10:20:21 INFO PaymentApplication:27 - Card Expiry Date: 12/2024

2023-08-02 10:20:21 INFO PaymentApplication:28 - Card CVC: 123

2023-08-02 10:20:21 INFO PaymentApplication:29 - Amount: 1234

2023-08-02 10:20:21 INFO PaymentApplication:43 - Successful Transaction: 1690968021919

2023-08-02 10:20:21 INFO PaymentApplication:25 - Starting Transaction: CMD000002

2023-08-02 10:20:21 INFO PaymentApplication:26 - Card Number: error

2023-08-02 10:20:21 INFO PaymentApplication:27 - Card Expiry Date: 12/2024

2023-08-02 10:20:21 INFO PaymentApplication:28 - Card CVC: 123

2023-08-02 10:20:21 INFO PaymentApplication:29 - Amount: 1234

2023-08-02 10:20:22 INFO PaymentApplication:25 - Starting Transaction: CMD000003

2023-08-02 10:20:22 INFO PaymentApplication:26 - Card Number: 1234 5678

2023-08-02 10:20:22 INFO PaymentApplication:27 - Card Expiry Date: error

2023-08-02 10:20:22 INFO PaymentApplication:28 - Card CVC: 123

2023-08-02 10:20:22 INFO PaymentApplication:29 - Amount: 1234

2023-08-02 10:20:22 INFO PaymentApplication:25 - Starting Transaction: CMD000004

2023-08-02 10:20:22 INFO PaymentApplication:26 - Card Number: 1234 5678

2023-08-02 10:20:22 INFO PaymentApplication:27 - Card Expiry Date: error

2023-08-02 10:20:22 INFO PaymentApplication:28 - Card CVC: 123

2023-08-02 10:20:22 INFO PaymentApplication:29 - Amount: 1234

Restart Payment Worker

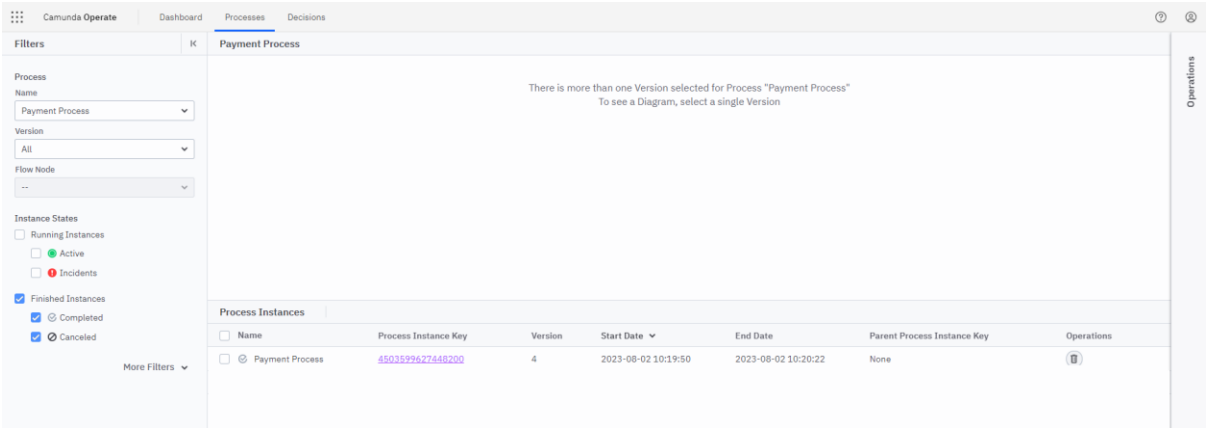
The Payment Worker has been configured to shutdown after processing requests for 60 seconds. You can either restart the Payment Worker as and when required, alternatively you can extend the default 60 seconds by
modifying WORKER_TIME_TO_LIVE in PaymentApplication.

Review Test Scenarios

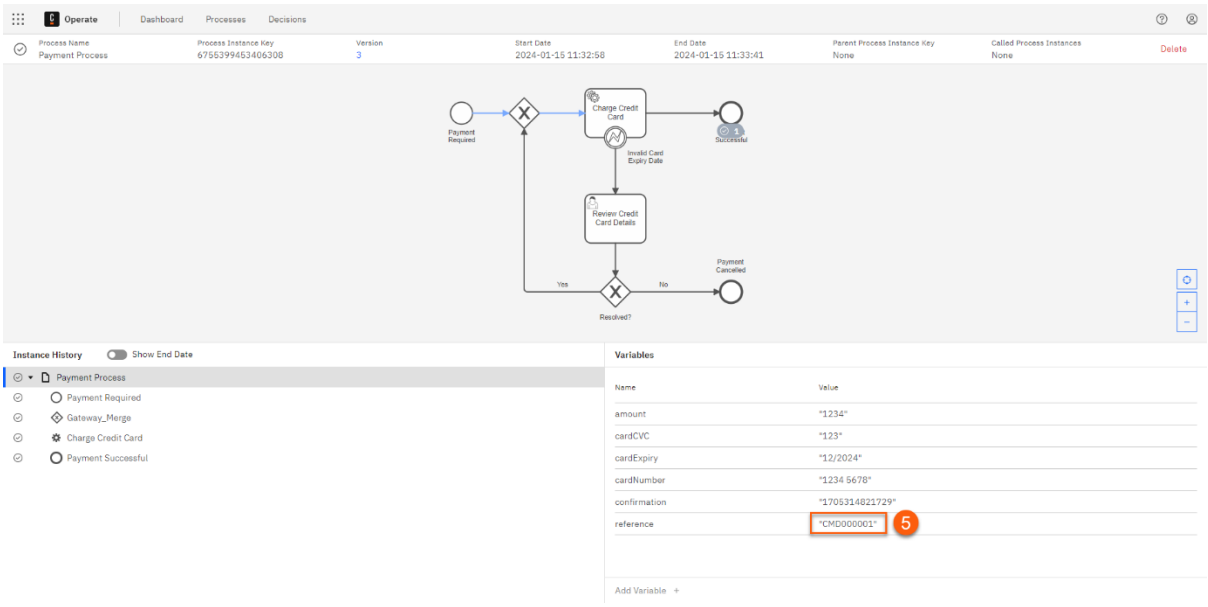
We can now review the results from the test scenarios to confirm that everything is working as expected.

Test Scenario 1

1. Launch **Operate** by following the steps in the previous lesson. Alternatively, refresh the window if you still have **Operate** open.
2. Click on *Payment Process* from the **Process Instances by Name** panel on the left-hand side of the screen
3. Check the **Finished Instances** filter on the left hand side of the screen to show completed process instances



4. Click on the **Process Instance Key** which now appears in the bottom part of the screen; this will open up the **Process Instance View**



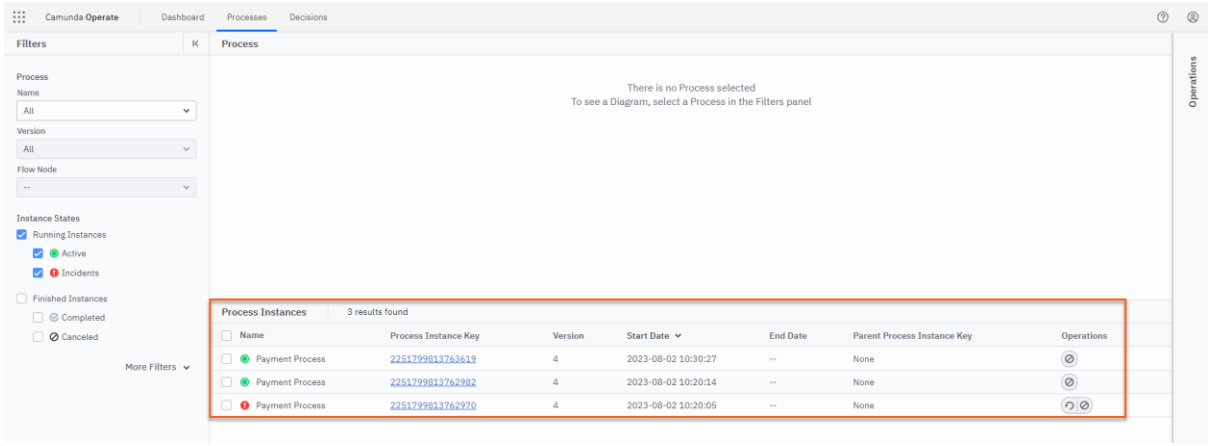
5. We can now see that the process completed with the *Payment Successful* event and that the Credit Card details have been populated.

Test Scenario 2

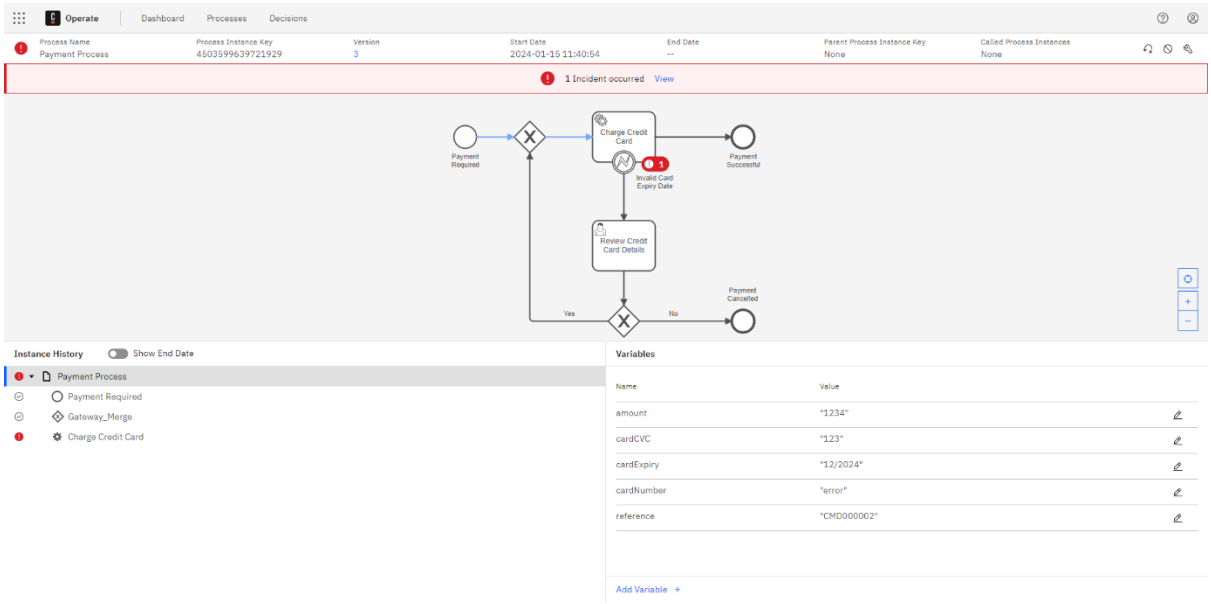
1. Launch **Operate** by following the steps in the previous lesson. Alternatively, refresh the window if you still have **Operate** open.

2. Click on *Payment Process* from the **Process Instances by Name** panel on the left-hand side of the screen

3. Check the **Running Instances** filter on the left hand side of the screen to show running process instances

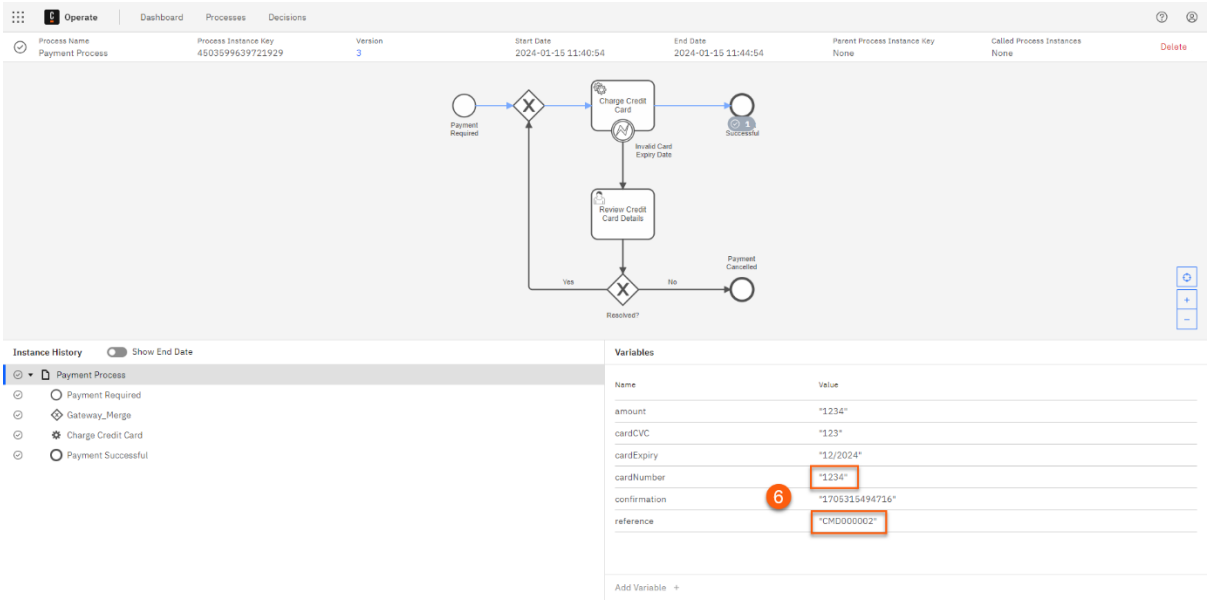


4. Click on the **Process Instance Key** for the process which contains an Incident.



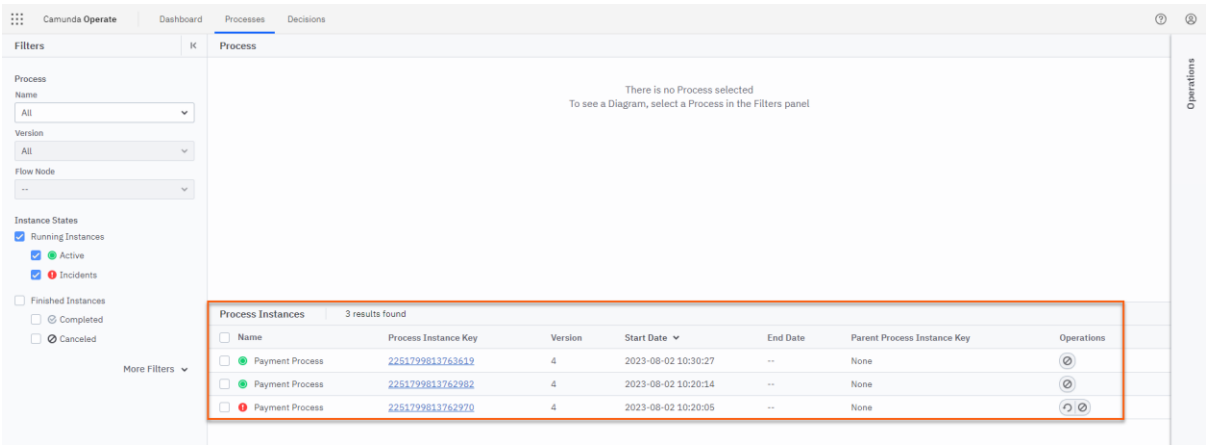
5. Using the same steps as documented in the **Handle Incidents in Operate** lesson; modify the cardNumber to a valid number and retry the Incident.

6. We can now see that the process completed with the *Payment Successful* event and that the Credit Card details have been updated.

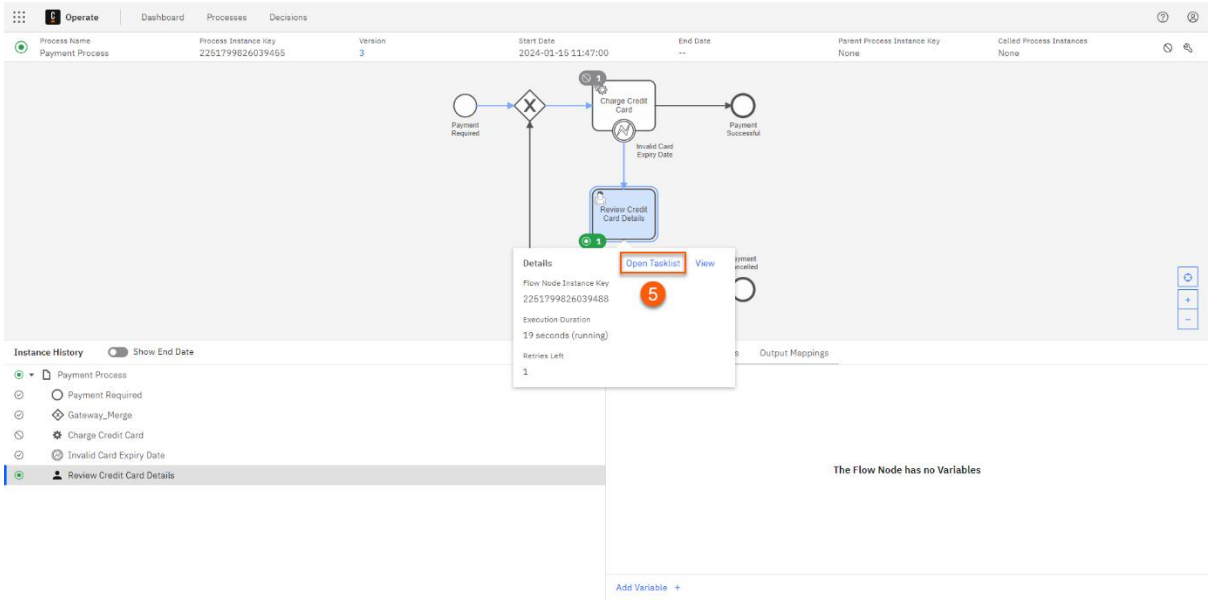


Test Scenario 3

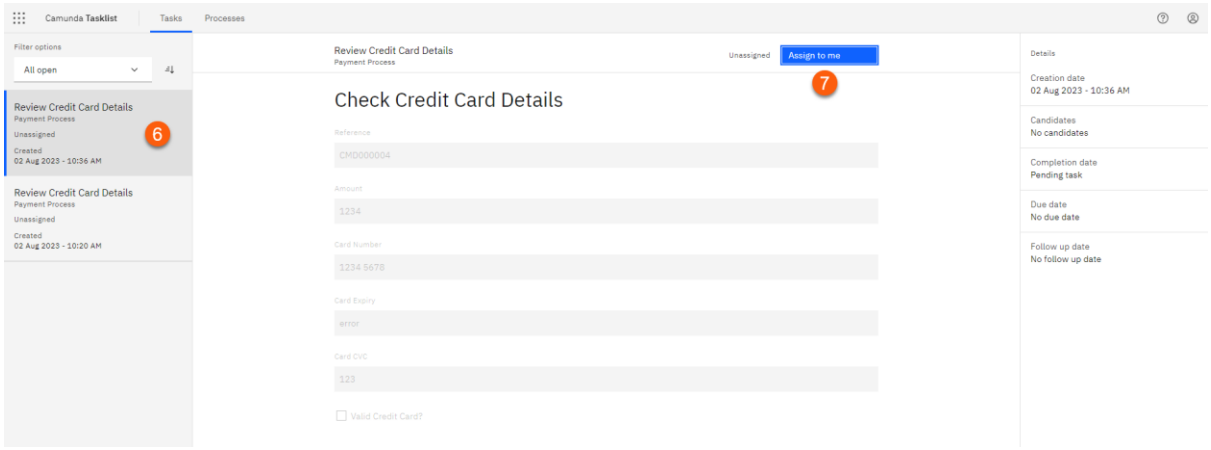
1. Launch **Operate** by following the steps in the previous lesson. Alternatively, refresh the window if you still have **Operate** open.
2. Click on *Payment Process* from the **Process Instances by Name** panel on the left-hand side of the screen
3. Check the **Running Instances** filter on the left hand side of the screen to show running process instances



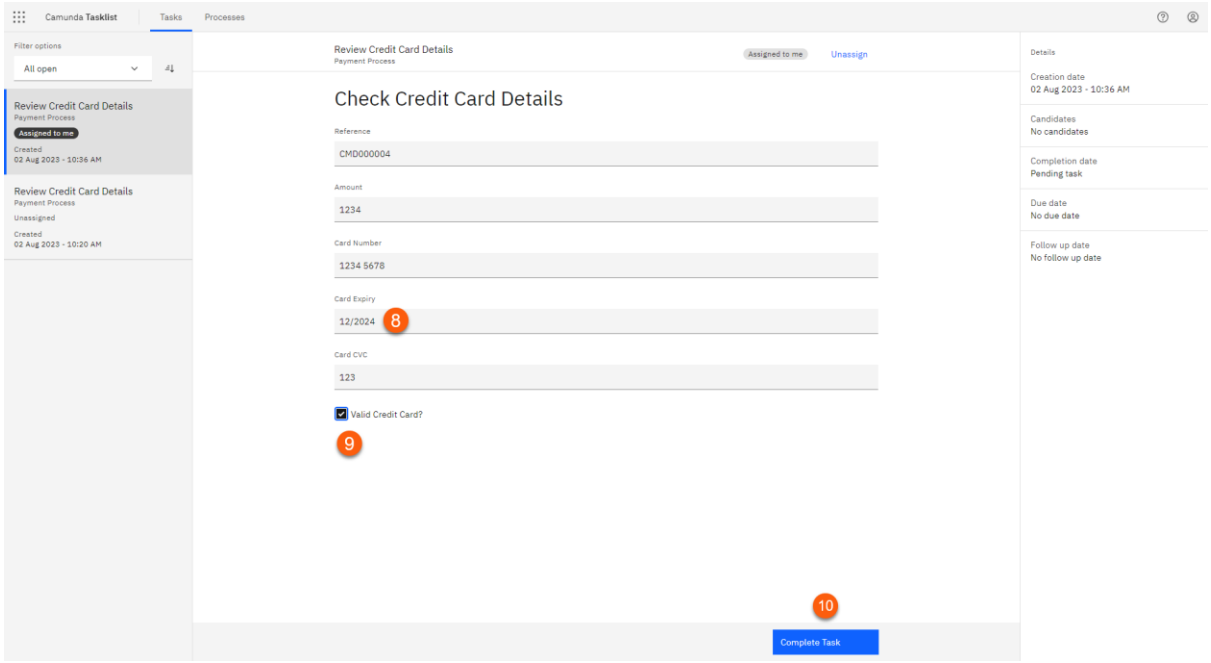
4. Click on the **Process Instance Key** for the first process in the list which is active
5. Click on the **Open Tasklist** link to launch Tasklist



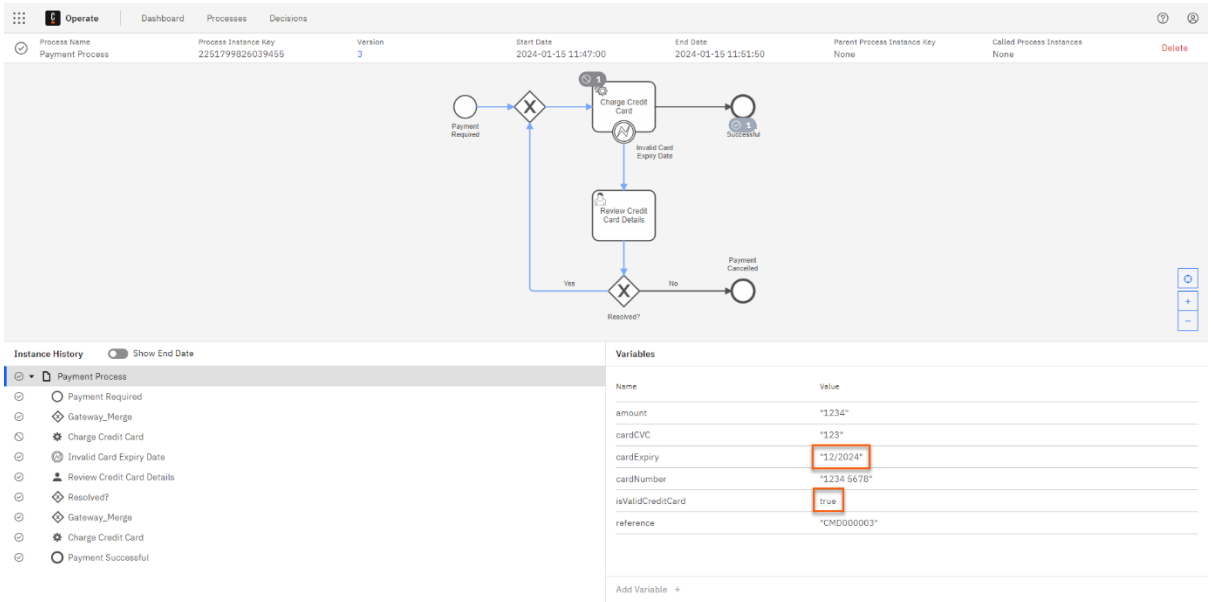
6. Click on the first task that appears in the Tasks panel on the left hand side of the screen
7. Click on **Assign to me** to claim the Task



8. Update the Card Expiry date to be 12/2024
9. Check the **Valid Credit Card?** checkbox
10. Click the **Complete** button to complete the task

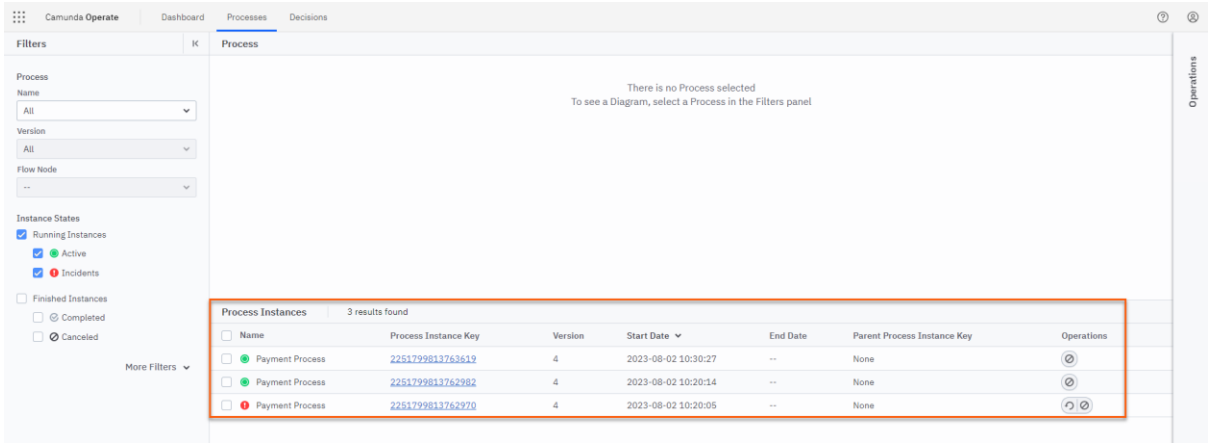


11. Finally, return to the previous screen in Operate to confirm that the process instance has been successfully completed.



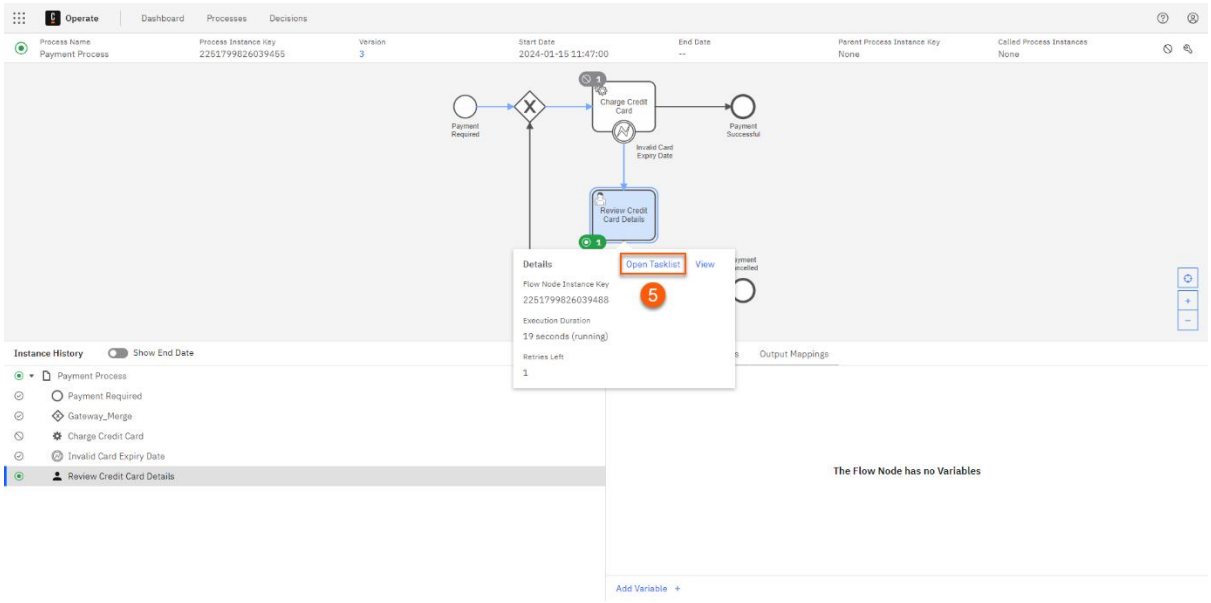
Test Scenario 4

1. Launch **Operate** by following the steps in the previous lesson. Alternatively, refresh the window if you still have **Operate** open.
2. Click on *Payment Process* from the **Process Instances by Name** panel on the left-hand side of the screen
3. Check the **Running Instances** filter on the left hand side of the screen to show running process instances



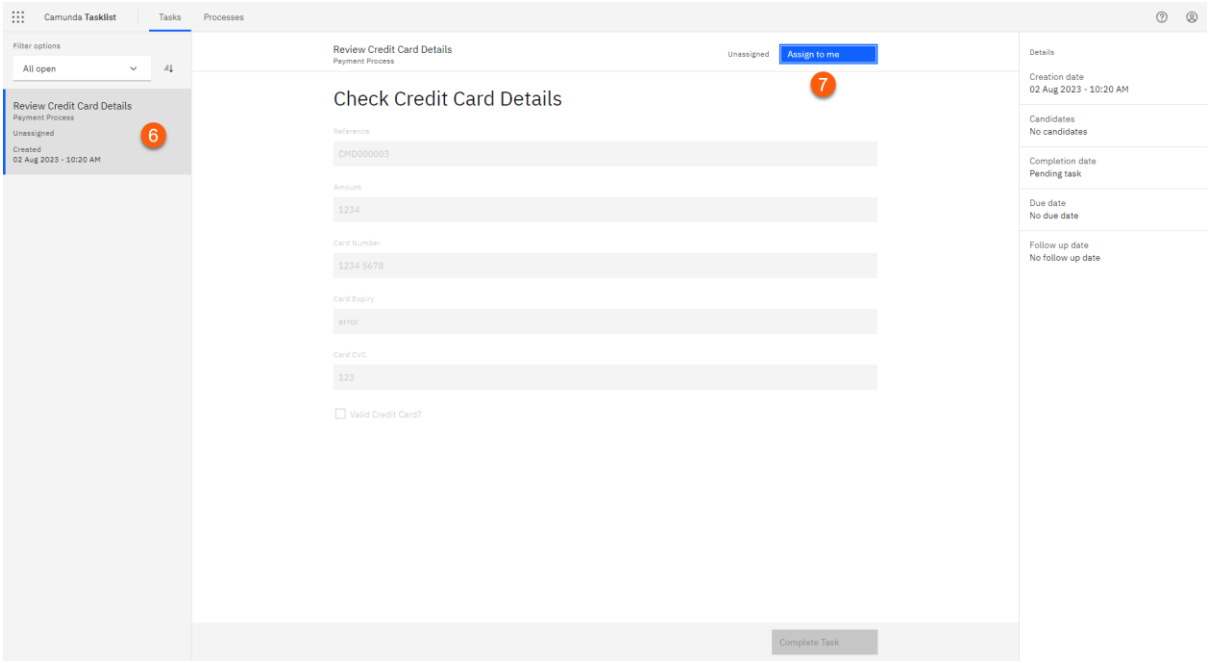
4. Click on the **Process Instance Key** for the first process in the list which is active

5. Click on the **Open Tasklist** link to launch Tasklist

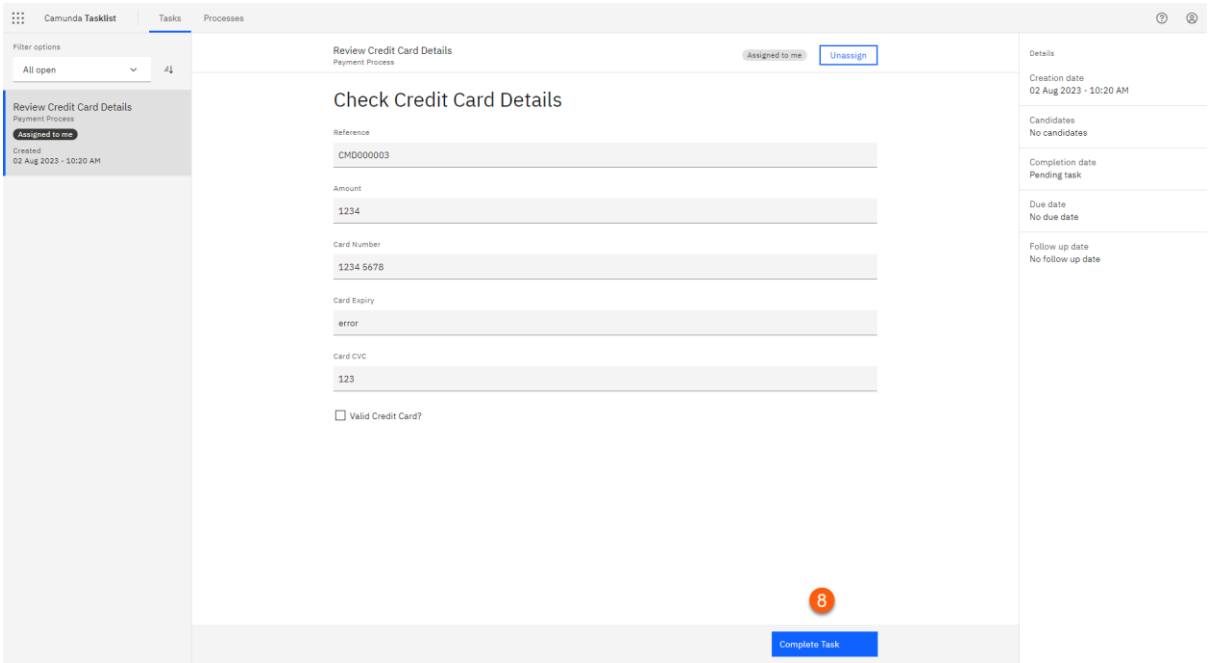


6. Click on the remaining task that appears in the Tasks panel on the left hand side of the screen

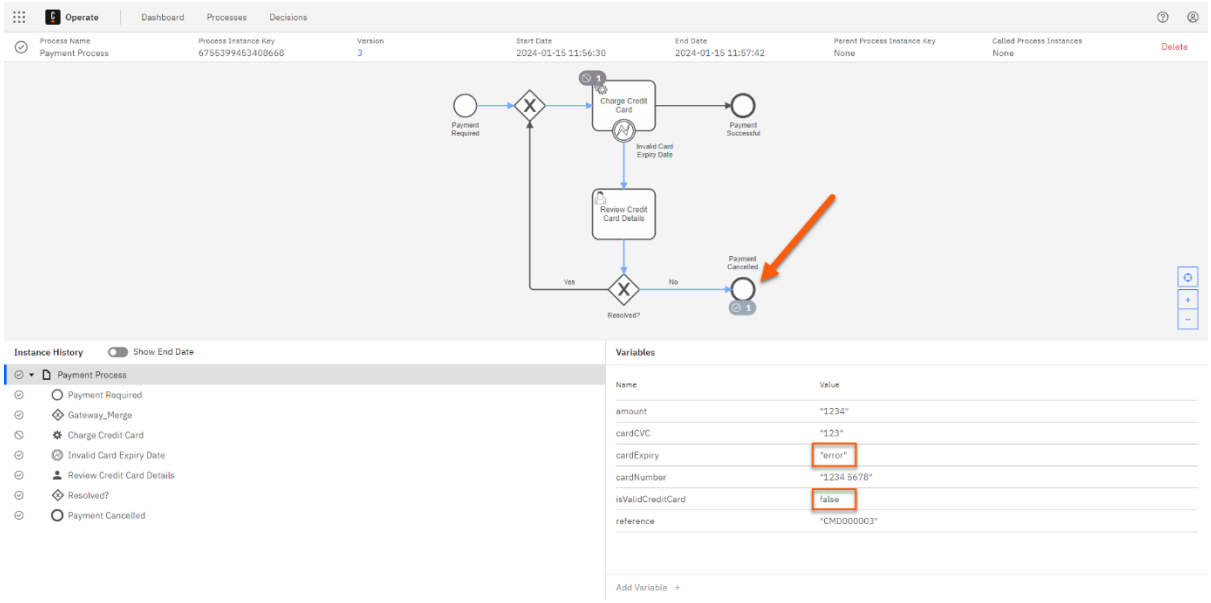
7. Click on **Assign to me** to claim the Task



8. Click the **Complete** button to complete the task



9. Finally, return to the previous screen in Operate to confirm that the process instance has been successfully completed and has ended at the **Payment Cancelled** event.



Example Solution

Solution

If you encountered any problems during this course or were unable to complete the process configuration, you can compare your solution with our example solution below.

GitHub Repository

Our version of the Process Definition including the Form definitions as well as the *Payment Application* can be found in the GitHub Repository below:

- Solution - Payment Application

Process Diagram

Our version of the Process Diagram can be found below:

Review

This course has provided you with an introduction to handling errors in Camunda 8.

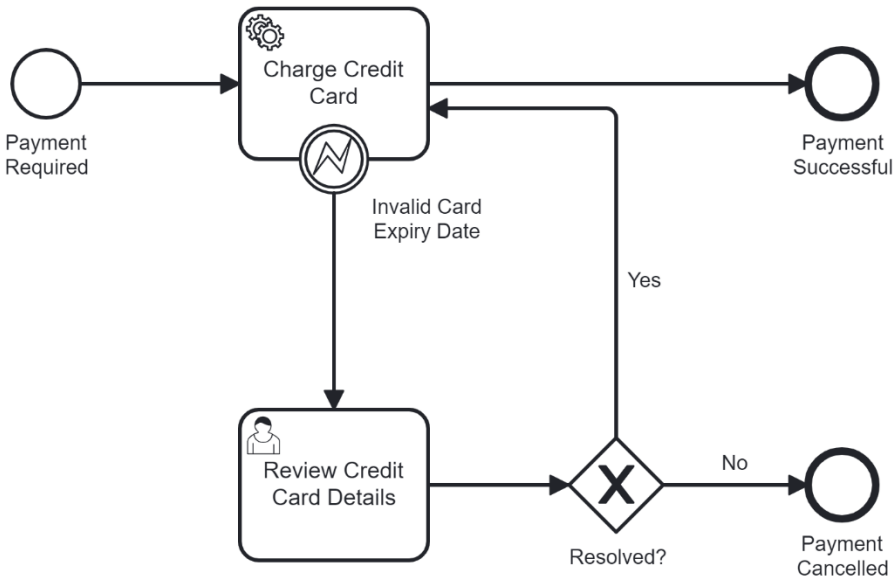
You have made a series of changes to a *Payment Application* so that it can handle errors encountered during process execution more effectively.

What Did I Learn?

You should now be able to:

- Access a Camunda 8 Environment
    - Access Console

    - Access Modeler
    - Access Tasklist
    - Access Operate
- Create a Camunda 8 Application
    - Create a Project using the Modeler
    - Import a Process Definition using the Modeler
    - Modify a Process Definition using the Modeler
    - Import a Form Definition using the Modeler
    - Add BPMN Error Handling to a Process Definition
    - Add BPMN Error Handling to a Job Worker
- Deploy a Camunda 8 Application
    - Deploy a Process Definition using the Modeler
    - Deploy an updated Process Definition using the Modeler
- Test a Process Definition
    - Start a Process Instance using the Modeler
    - Claim a Task using Tasklist
    - Populate a Form using Tasklist
    - Complete a Task using Tasklist
- Monitor a Process Definition
    - View a Process Instance using Operate
    - Modify a Process Instance using Operate
    - Cancel a Process Instance using Operate
    - Delete a Process Instance using Operate
- Monitor an Incident
    - Differentiate between a BPMN Error and an Incident
    - Retry a Failed Job


Payment
Required
Charge Credit
Card
Payment
Successful
Invalid Card
Expiry Date
Yes
Review Credit
Card Details
Resolved?
No
Payment
Cancelled