Event Driven Architecture and Microservices

Event Driven Architecture (EDA):
EDA is a modern architecture pattern built from small, decoupled services that publish, consume or route events.

Events are sequential and real time flow of data objects. Objects can be anything event based. Event is a change has been occurred and is recorded. An object consists of unique id, value, timestamp of when the event occurred, metadata is optional.

Message queues are in FIFO format for the consumers to consume them. Message queue Kafka is used generally. Kafka filters message queues using topics. Each topic can be consumed by different consumers. Consumer will know that there are events in the topic by subscribing to that topic so that whenever a new event is pushed, consumer will know of it. It will get the event and check what type of event is it and react based on that event. This process is scalable as events are increasing.

Current challenges faced in existing integrations:

- Availability challenges
- How we are handling is an issue too since we are publishing entire hierarchy of data to Kafka
- Fat payloads are sent to Kafka
- Due to large payload times, it takes longer time to process like 12 hours for 2400 records
- Stress on infra and integration systems
- Data is delivered but consumer may not need it
- Maintaining local copy of data is an issue too since there are too many copies
- MDM account data is copied at least 7 times which is not currently scalable
- Data is not real-time since there is a minimum of 15-minute delay with scheduled pollers

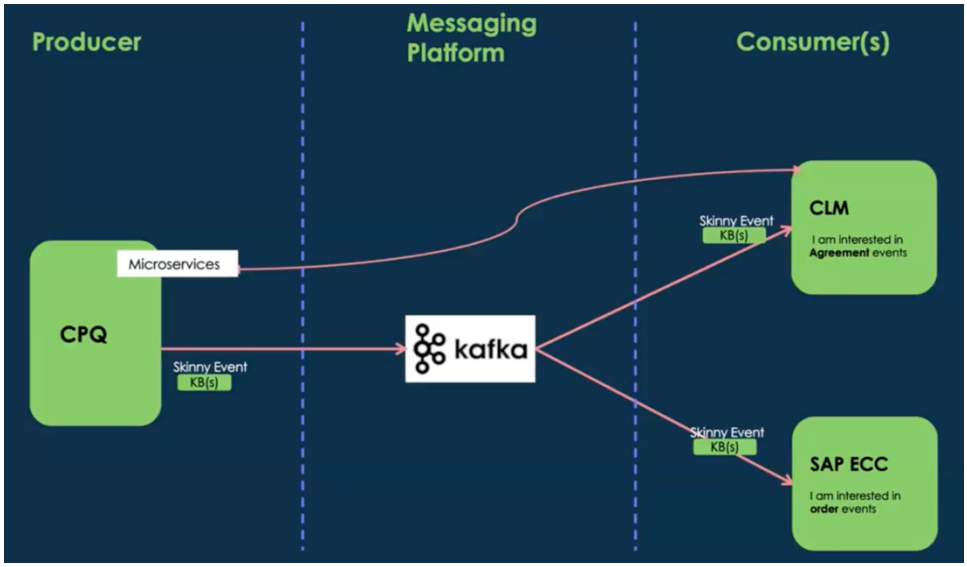
Solution:
Event Driven Architecture (EDA)-

- Decouple Scalable distributed systems- Consumer can consume event on its own timeline since EDA will publish the event before
- Reduce Data duplication- It has skinny messages which are stripped down to perform what is necessary
- Security- OAuth and Okta layer of protection is used for topics
- Fine-grained scalability
- Zero Data loss- Even if there is data loss, consumer can replay and get data again, so potentially there is no data loss
- Increased flexibility and agility- We can add remove and modify services on the go so that event can plug-out and then plug-in

Principles of EDA:

- Real-time events as they get at the producer
- Push notifications
- One-way "fire and forget"
- Immediate action at the consumer
- Informational
- Decouple (Scalable distributed systems)
- Security
- Increased flexibility
- Fine-grained scalability
- Reduce Data Duplication
- Zero data loss

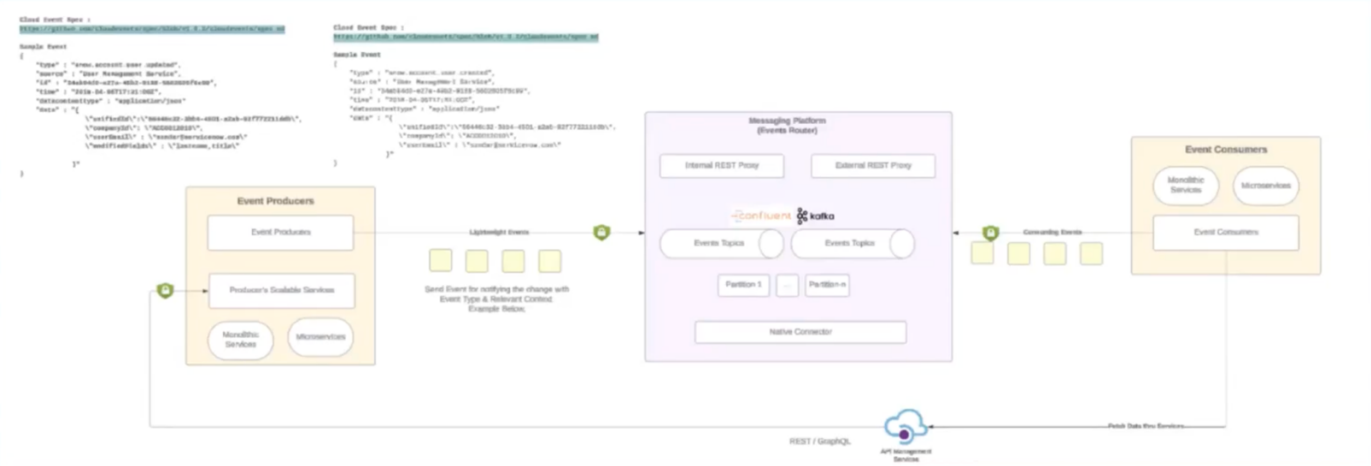
EDA in Action:



Standards followed/ Good practices:

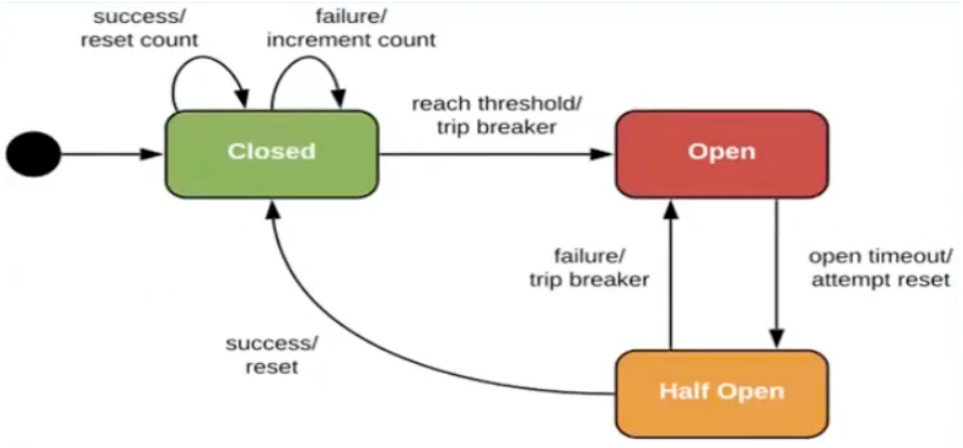
- Light event and skinny event: Event should have minimal data
- Standardized consistent event format: Helps us to align events in analytics.
  Spec- https://github.com/cloudevents/spec/blob/v1.0.2/cloudevents/spec.md
- Events are categorized and separated so that consumers pick what they are interested
- Event Modeling: Important when setting up service/ producer. Consumers which are later developed are developed according to event models.
- Event Versioning: In case in future, we require the models to be changed, then it should be backward compatible which is done through event versioning.
- Error Handling and Dead letter queues: Even if records have failure, consumer can replay the messages, if consumer is not able to do it, then we can pull them from dead letter queues.

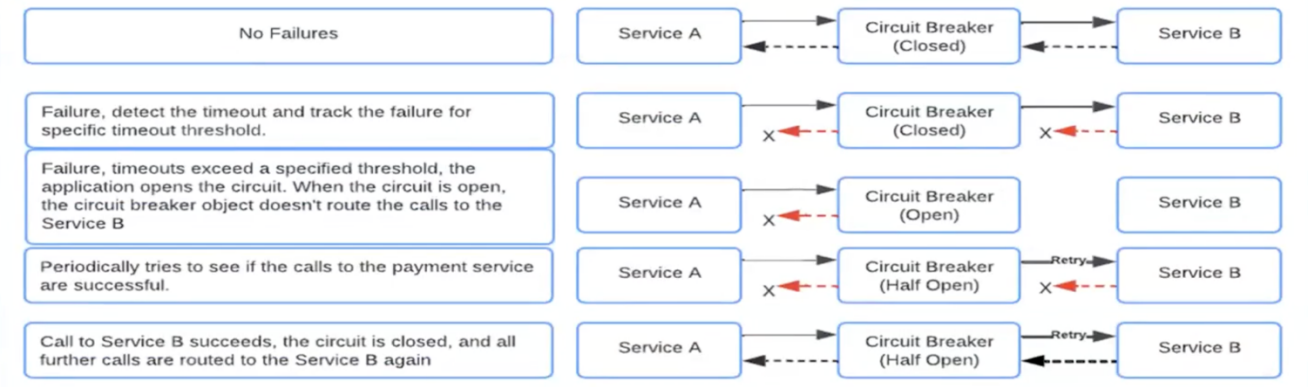
EDA Reference Architecture:



Circuit Breaker:

Prevent a caller service from retrying a call to another service (callee) when the call has previously caused repeated timeouts of failures.





Evolutions from Monoliths to Micro Services:

1. **Monolithic Architecture:**
   Traditionally we used to have monolithic architecture. It is a type of building software where you have all type of code base and components in a single service structure which are very tightly coupled. This meant that for any change or update, the entire application needed to be modified. This led to challenges in scalability, deployment and maintenance.

2. **Service Oriented Architecture:**
   SOA is an architectural style which divides entire application into multiple services where it has a single layer communicating to all services. It focuses on modularizing an application into individual services, each responsible for specific business functionalities. These services communicate with each other through standardized interfaces, typically using web services or message queues.

3. **Micro Services Architecture:**
   Microservice is a type of arch pattern where you build an entire app which is divide into granular services which can run independently. All the services are independent to each other in Microservices. This architecture takes the concept of modularization further by breaking down an application into even smaller, autonomous services. Each microservice focuses on a specific business capability and communicates visa lightweight protocols, such as HTTP or message brokers.

Advantages of Micro Services:

- Smaller and Faster deployments: Every code takes care of 1 functionality so it can be deployed independently
- Ease of understanding: Easier to understand since it is organized in multiple services and each is organized in a different way
- Eliminate vendor or technology lock-in: Every single services can be reused in different technology stack. During trouble shooting, it makes easier to identify what is going wrong.
- Business centricity: Everything can be serviced in micro service

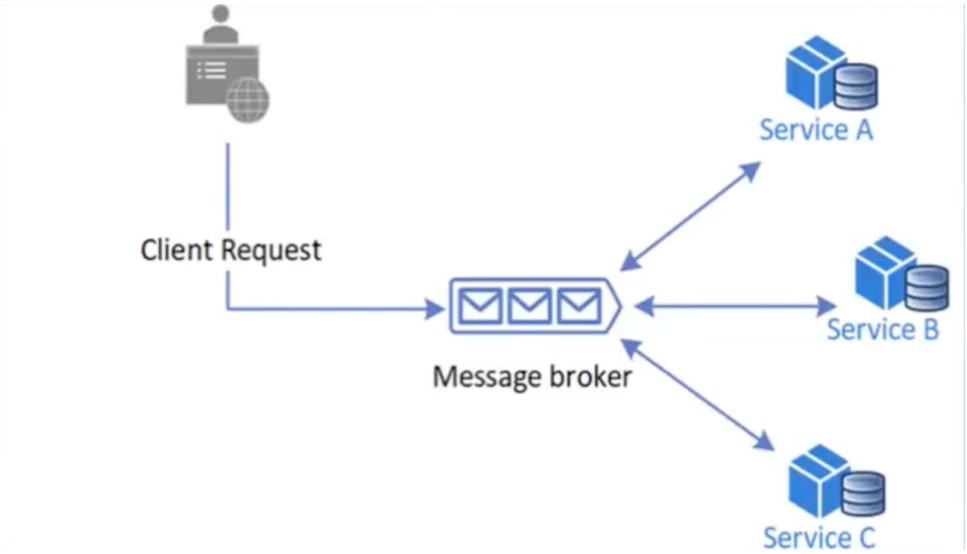
Micro Services Design Patterns:

1. Aggregator Pattern
2. Asynchronous Messaging
3. API Gateway
4. Event Sourcing
5. Command Query Responsibility Segregator
6. Database or Shared Data
7. Chain of Responsibility
8. Decomposition
9. Branch
10. Circuit Breaker

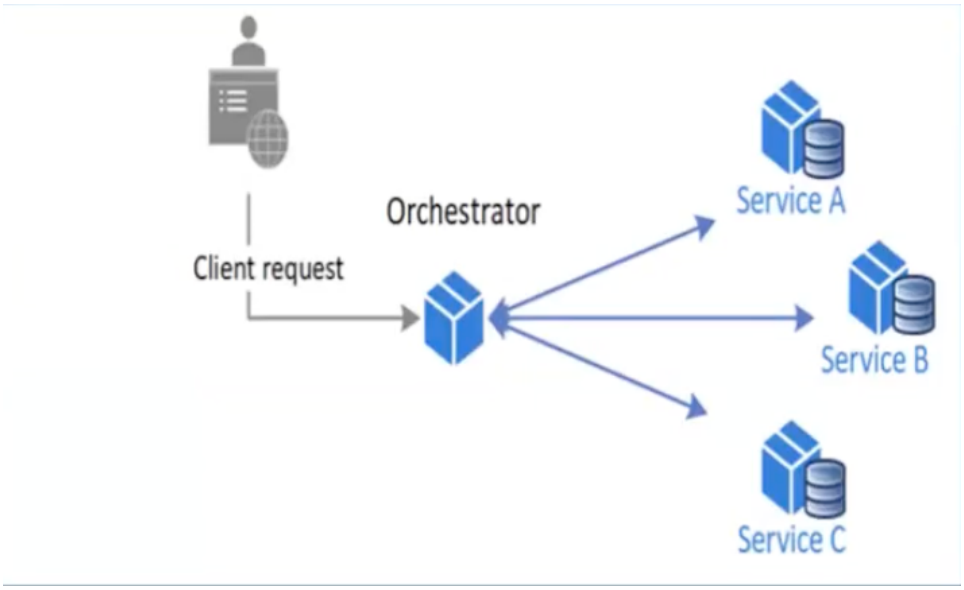
Command Query Responsibility Segregation (CQRS):
Separating the responsibilities of reading and writing data makes more flexible and efficient designs, particularly in domains with high-performance and scalability requirements.

SAGA:
Saga pattern provides state management using a sequence of states.

1. Choreography: Choreography is a way to coordinate sagas where participants exchange events without a centralized point of control.

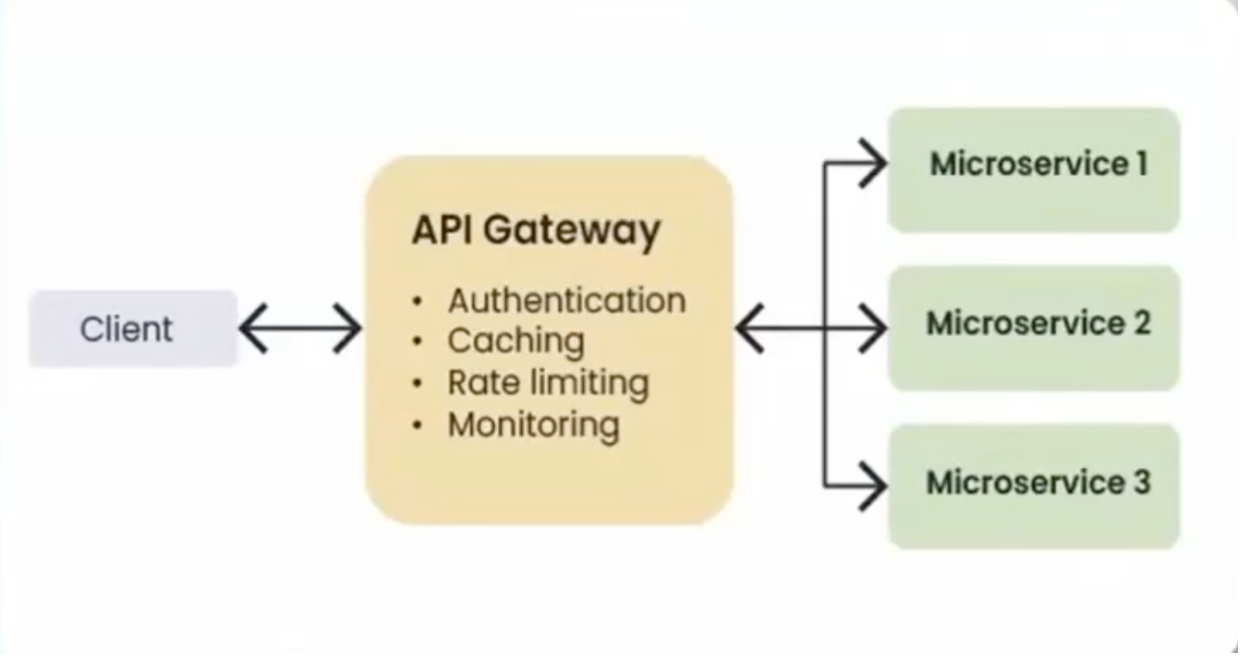


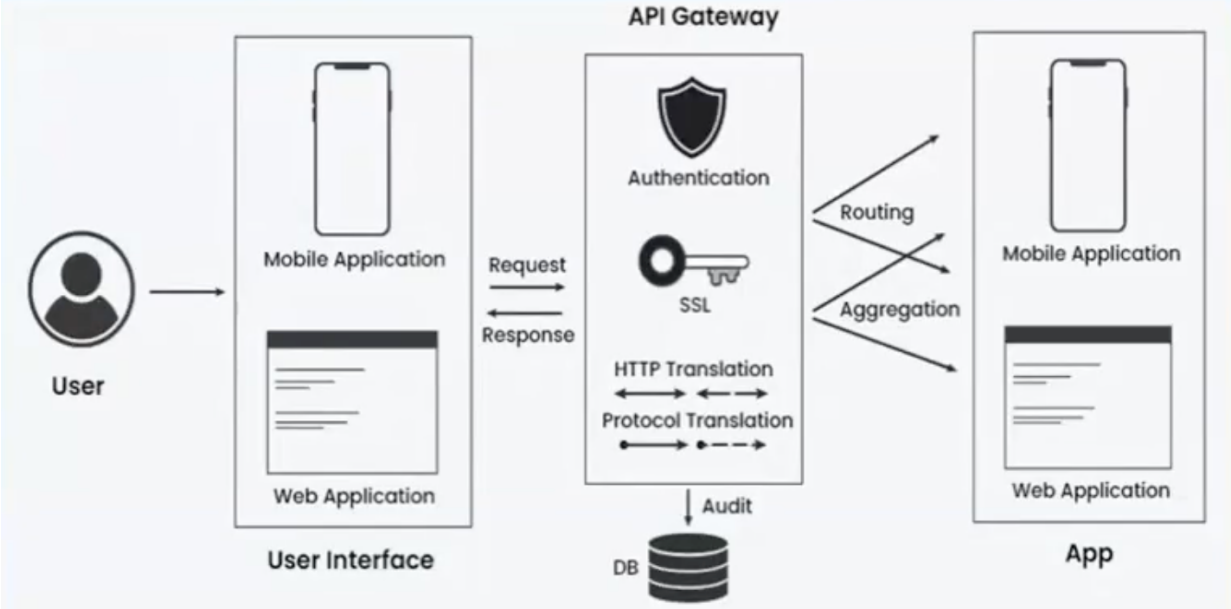
2. Orchestration: The orchestrator handles all the requests and tells the participants which operation to perform based on events.



API Gateway:

Entry points for clients. Instead of calling services directly, clients call the API gateway which forwards the call to the appropriate services on the back end. It serves as a single-entry point for clients to access multiple services, providing a unified interface and abstracting the complexities of the underlying architecture.





Use Case:

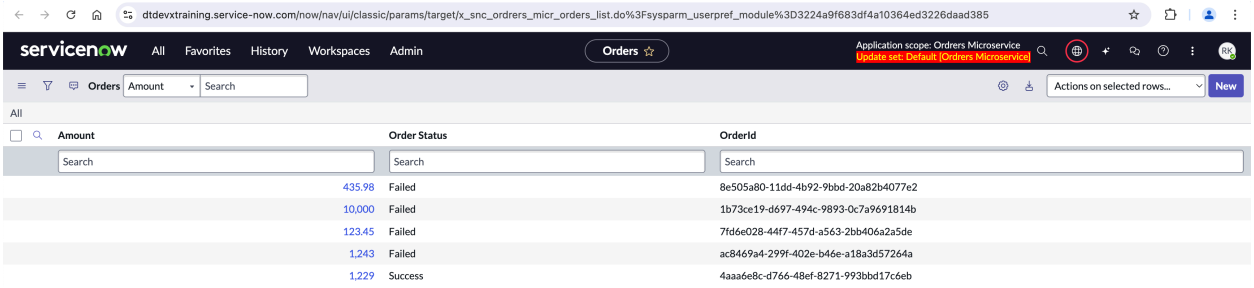
Create services in the now platform using scoped applications and how we can manage their interactions via Rest API and events. We will build the services in a way that they can be easily swapped out and replaced by cloud native services if we want to scale them up.

We will also see how to achieve data consistency across multiple services, and we will explore the SAGA pattern for this. We are going to explore both orchestration and choreography-based SAGA.

We are going to create an orders service and a payments service. Let's assume this is the backend of an e-commerce app. Now the orders service manages the orders creation and then a request is sent to the payment service. The payment service manages the payment flow and once the payment is successful, we also must mark the order as successful. If the payment fails, we also must mark the order as failed. Since these are different services and do not share data, we cannot do this in a single transaction.

Creating the Orders Service:

1. Create a new scoped app Orders Service.
2. Create an orders table from App engine studio or from tables application. It should contain 3 fields- Amount, Order ID and Order status.



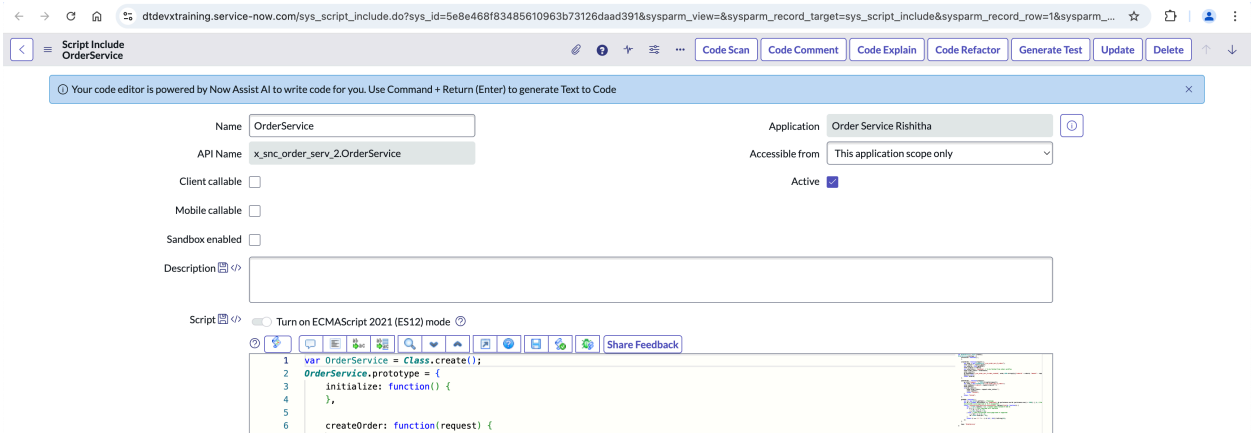
3. Now we need to create the service layer for this. So, we create a Script Include with create order and update order functions in the Orders Service application scope.

Script Include:
Name: OrderService
API Name: x_snc_ordrers_micr.OrderService

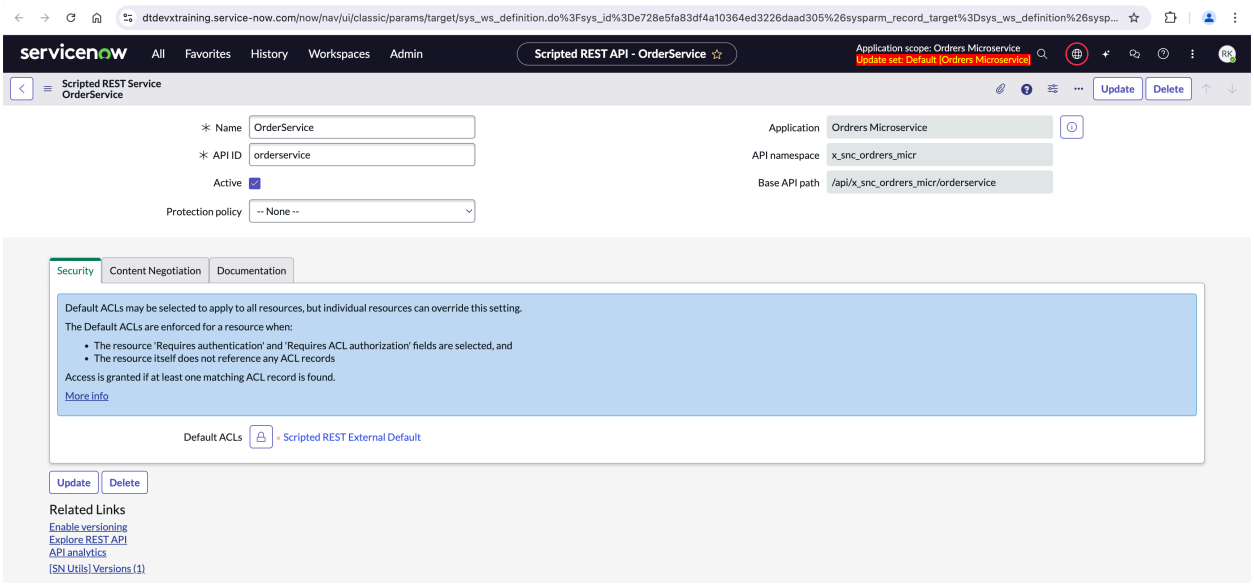
```
var OrderService = Class.create();
OrderService.prototype = {
    initialize: function() {},

    createOrder: function(request) {
        var order = new GlideRecord("x_snc_ordrers_micr_orders");
        order.amount = request.amount;
```

```
        var orderid = this.getUUID();
        order.orderid = orderid;
        var orderStatus = "pending"; // to be fetched from orders workflow
        order.order_status = orderStatus;
        order.insert();
        gs.eventQueue("x_snc_ordrers_micr.order_created",
order,JSON.stringify({"orderid" : orderid, "amount" : request.amount}),null);
        return orderid;
    },

    updateOrder : function(request){
        gs.info("request: " + JSON.stringify(request));
        var order = new GlideRecord("x_snc_ordrers_micr_orders");
        order.addQuery("orderid", request.orderid+"");
        order.query();
        if(order.next()){
            order.order_status = request.order_status+"";
            order.update();
            return "success";
        }
        return "failed";
    },

    getUUID: function() {
        var d = new Date().getTime(); //Timestamp
        var d2 = ((typeof performance !== 'undefined') && performance.now &&
(performance.now() * 1000)) || 0; //Time in microseconds since page-load or 0 if
unsupported
        return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(/[xy]/g, function(c) {
            var r = Math.random() * 16; //random number between 0 and 16
            if (d > 0) { //Use timestamp until depleted
                r = (d + r) % 16 | 0;
                d = Math.floor(d / 16);
            } else { //Use microseconds since page-load if supported
                r = (d2 + r) % 16 | 0;
                d2 = Math.floor(d2 / 16);
            }
            return (c === 'x' ? r : (r & 0x3 | 0x8)).toString(16);
        });
    },

    type: 'OrderService'
};
```

Ignore the part about event queueing. We will explore that when we do the part about choreography saga.

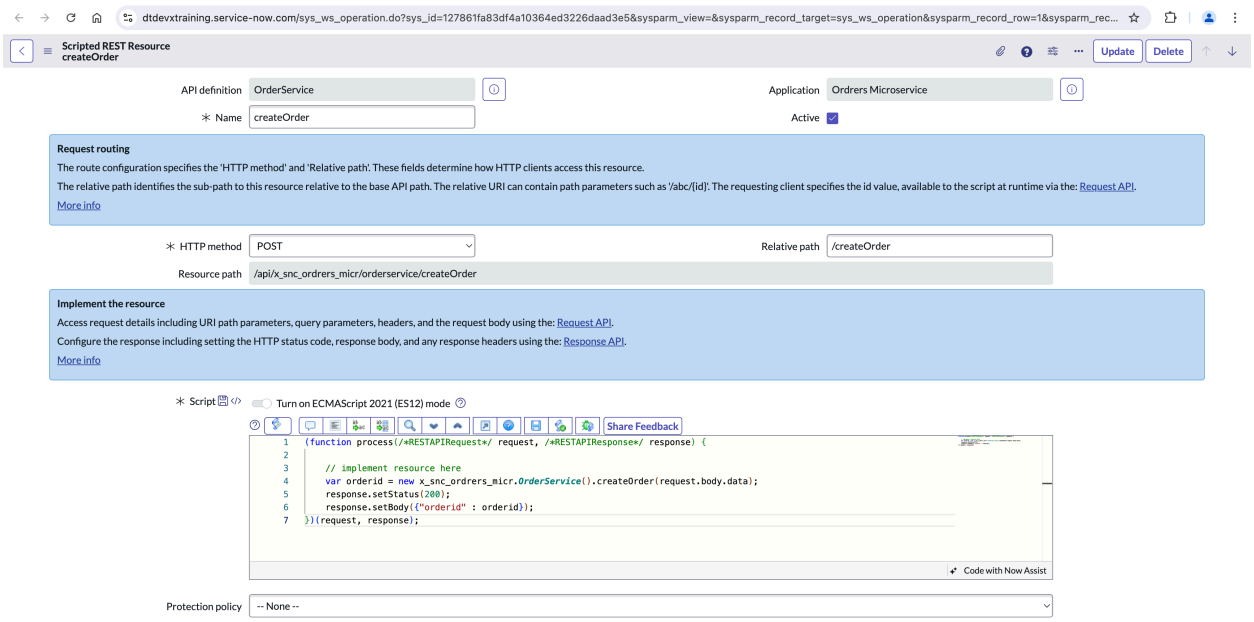


4. Create a Rest API for this service. Other Services will interact with our service only via Rest APIs or events. Create a Scripted REST Service named OrderService. In this Scripted REST Service, we need 2 APIs for create and update orders.



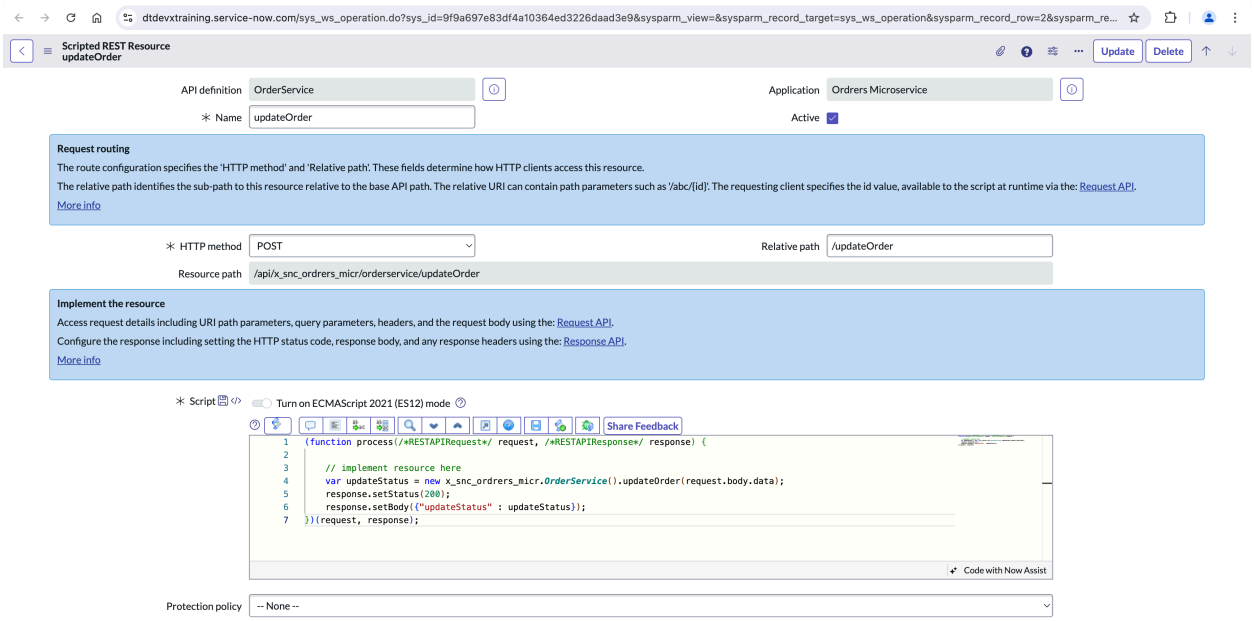
a. Create a new Scripted REST Resource named createOrder with HTTP POST method, relative path as '/createOrder' and the below script:

```
(function process(/*RESTAPIRequest*/ request, /*RESTAPIResponse*/ response) {

    // implement resource here
    var orderid = new
x_snc_ordrers_micr.OrderService().createOrder(request.body.data);
    response.setStatus(200);
    response.setBody({"orderid" : orderid});
})(request, response);
```

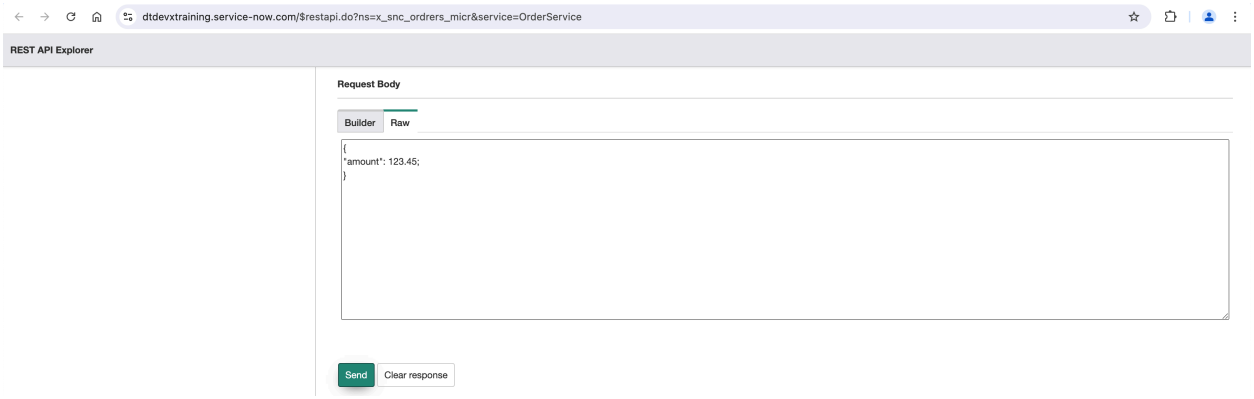


b. Create a new Scripted REST Resource named updateOrder with HTTP POST method, relative path as '/updateOrder' and the below script:

```
(function process(/*RESTAPIRequest*/ request, /*RESTAPIResponse*/ response) {

    // implement resource here
    var updateStatus = new
x_snc_ordrers_micr.OrderService().updateOrder(request.body.data);
    response.setStatus(200);
    response.setBody({"updateStatus" : updateStatus});
})(request, response);
```

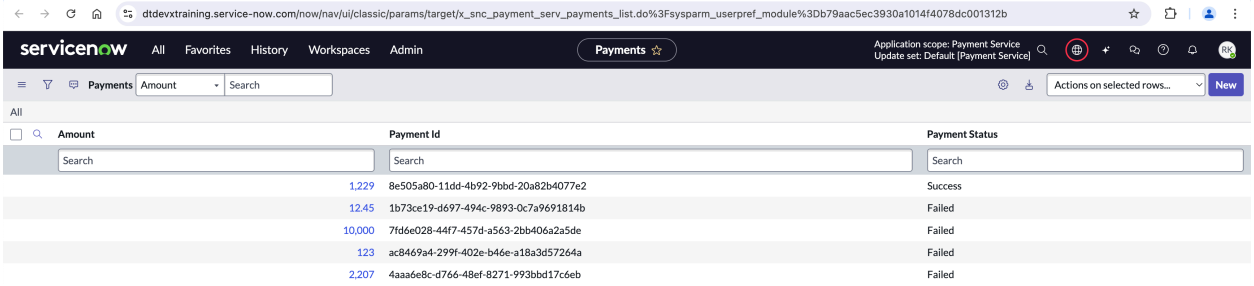


We can test the APIs using the REST API explorer from the related links section in Scripted REST Service or postman and they should be able to create and update the orders in orders table.



Creating the Payments Service:

1. Create a new scoped app Payment Service.
2. Create a payments table from App engine studio or from tables application. It should contain 3 fields- Amount, Payment ID and Payment status.



3. Now we need to create the service layer for this. So, we create a Script Include with create payment function in the Payment Service application scope.

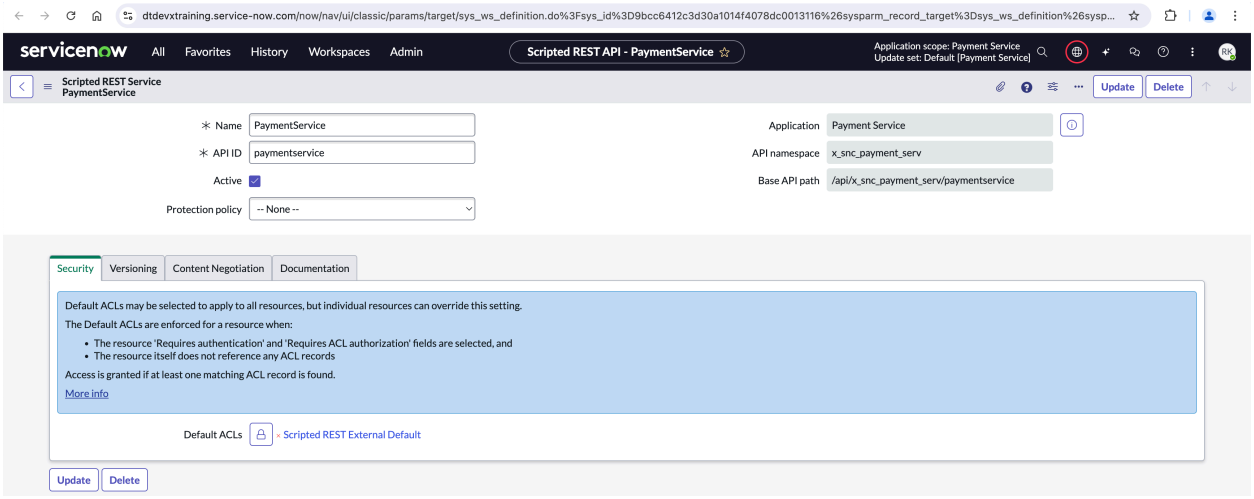
Script Include:
Name: PaymentService
API Name: x_snc_payment_serv.PaymentService

```
var PaymentService = Class.create();
PaymentService.prototype = {
    initialize: function() {
    },

    createPayment : function(request){
        var amount = request.amount;
        var orderid = request.orderid;
        var payment = new GlideRecord("x_snc_payment_serv_payments");
```

```
        payment.amount = amount;
        var paymentStatus = "failed"; // ideally call to payment workflow to determine
payment success or failure
        payment.payment_status = paymentStatus;
        payment.orderid = orderid;
        payment.insert();
        gs.eventQueue("x_snc_payment_serv.payment_status", payment,
JSON.stringify({"orderid" : orderid, "payment_status" : paymentStatus}), null);
        return paymentStatus;
    },

    type: 'PaymentService'
};
```

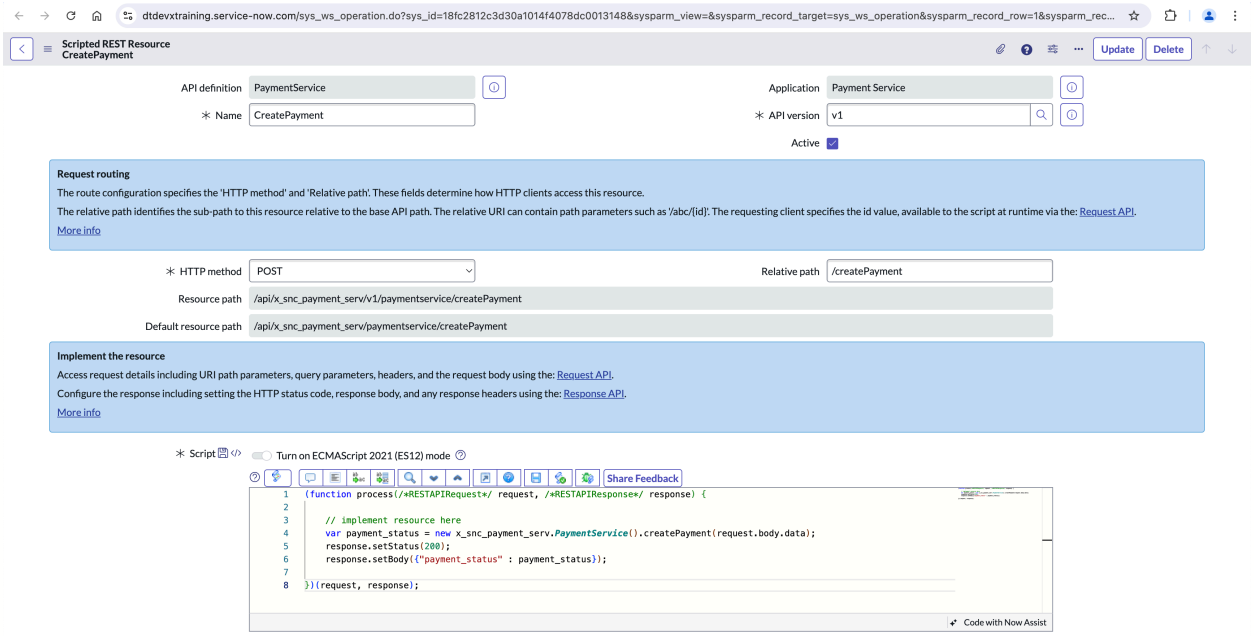
Ignore the part about event queueing. We will explore that when we do the part about choreography saga.

4. Create a Rest API for this service. Other Services will interact with our service only via Rest APIs or events. Create a Scripted REST Service named PaymentService. In this Scripted REST Service, we need 1 API for create payment.



a. Create a new Scripted REST Resource named CreatePayment with HTTP POST method, relative path as '/createPayment' and the below script:

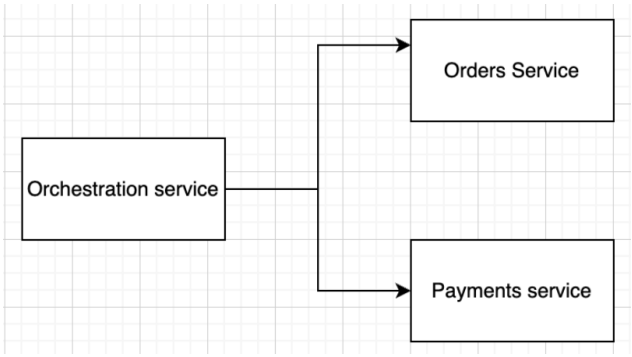
```
(function process(/*RESTAPIRequest*/ request, /*RESTAPIResponse*/ response) {
    // implement resource here
    var payment_status = new
x_snc_payment_serv.PaymentService().createPayment(request.body.data);
    response.setStatus(200);
    response.setBody({"payment_status" : payment_status});
})(request, response);
```



Now if we want to create an order, we will also have to create a payment. One way is that the orders service will call the payments service api but we do not want to do that as the payments service is complex and might take time (we have abstracted it for the lab but ideally the payments service should interact with 3rd party payment portals via a payments workflow). Another issue is that ideally we will have multiple services (shipping, notification, etc) and we don't want our orders service to manage all these interactions.

Orchestration SAGA:

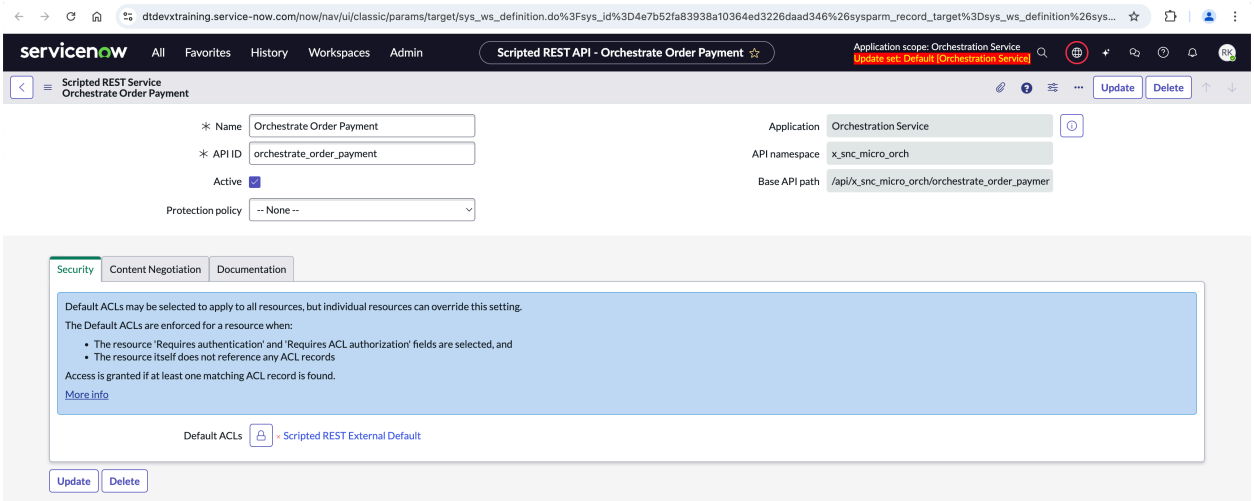
This is an use case for Orchestration based SAGA. We can have an orchestration service and that can manage the interactions between the services.



The orchestration service first creates the order in orders service via the create order api. The order is initially in pending state. Then it creates a payment in payment service via the create payments api. Then once the payment is successful, it updates the order as successful by calling the update order API. If the payment fails, the orchestration service can mark the order as failed. As we can see this design is scalable as the responsibility to manage interactions among all the services (assuming there are more) lies with the orchestration service.

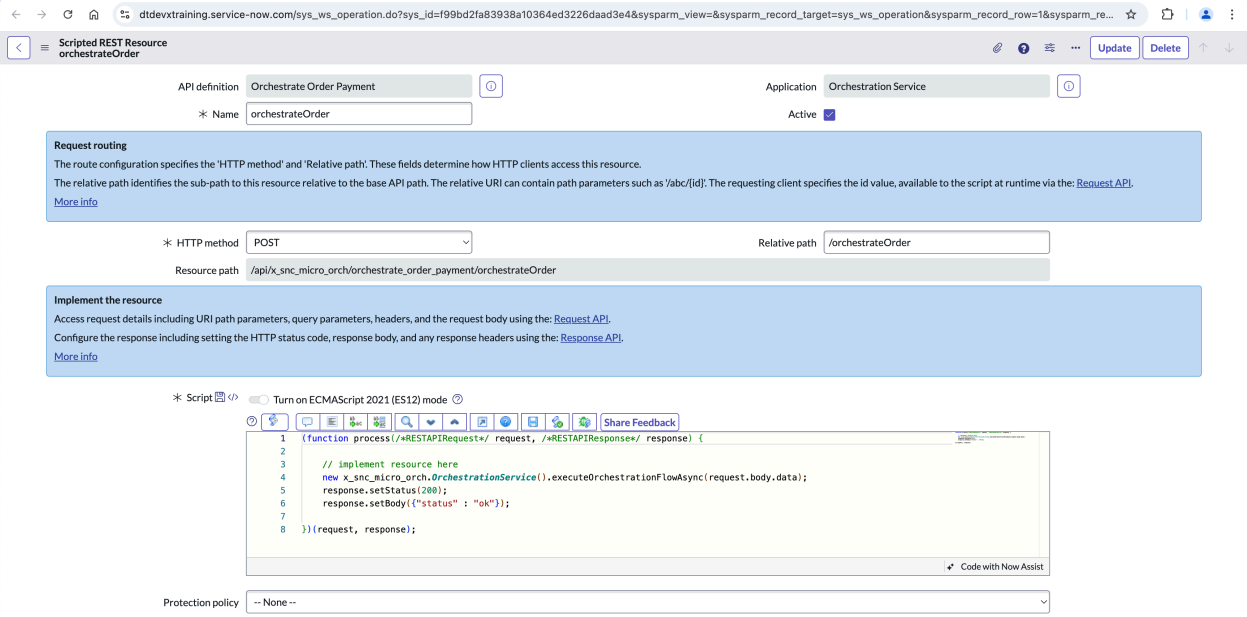
Now let's create an orchestration service. Here we will need a rest API to call the orchestration service and an orchestration subflow and multiple actions to call the payments and orders services.

1. Create a Rest API for orchestration. Create a Scripted REST Service named Orchestrate Order Payment. In this Scripted REST Service, we need 1 API for orchestrate order.



2. Create a new Scripted REST Resource named orchestrateOrder with HTTP POST method, relative path as '/orchestrateOrder and the below script:

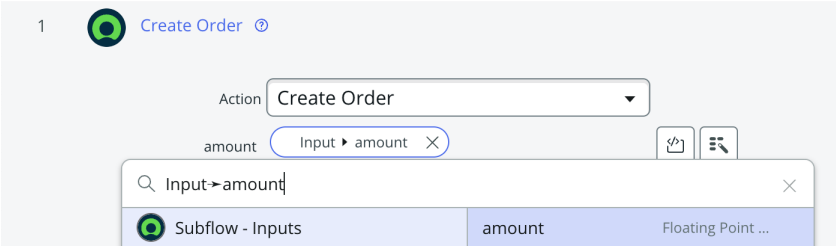
```
(function process(/*RESTAPIRequest*/ request, /*RESTAPIResponse*/ response) {
    new
x_snc_micro_orch.OrchestrationService().executeOrchestrationFlowAsync(request.body.dat
a);
    response.setStatus(200);
    response.setBody({"status" : "ok"});
})(request, response);
```



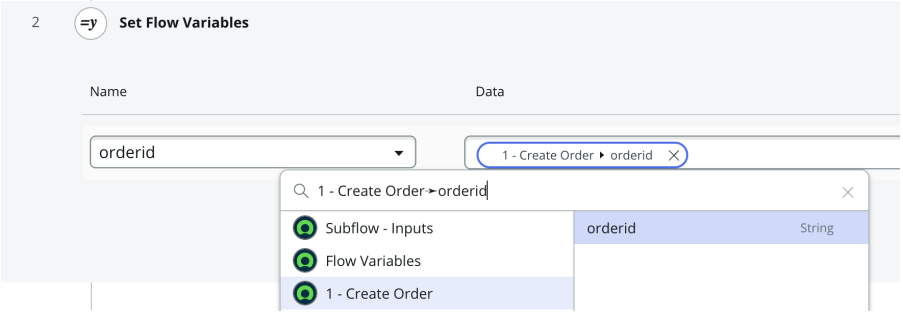
3. Create an Orchestration subflow with flow inputs- amount (Floating point number) and orderDetails (JSON)

Sub-Flow Actions:

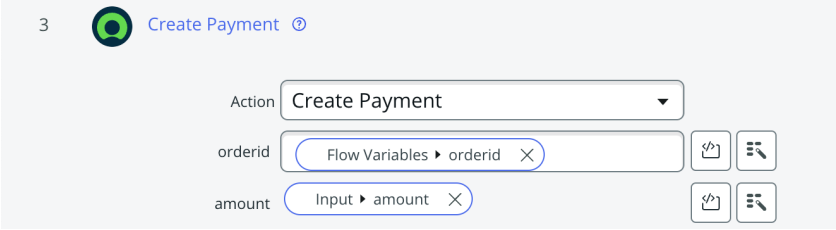
a) Create Order:



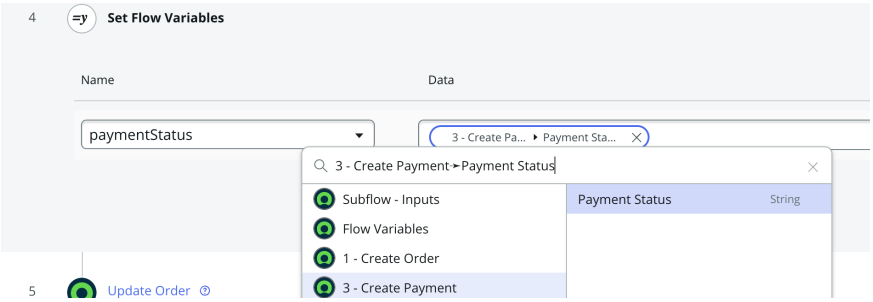
b) Set Flow variables:



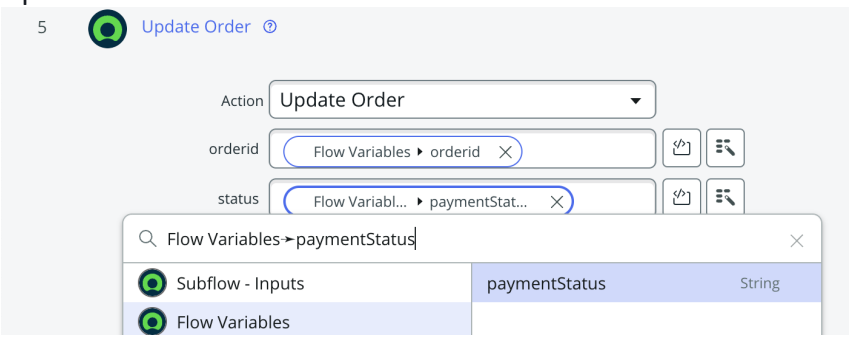
c) Create Payment:



d) Set Flow variables:

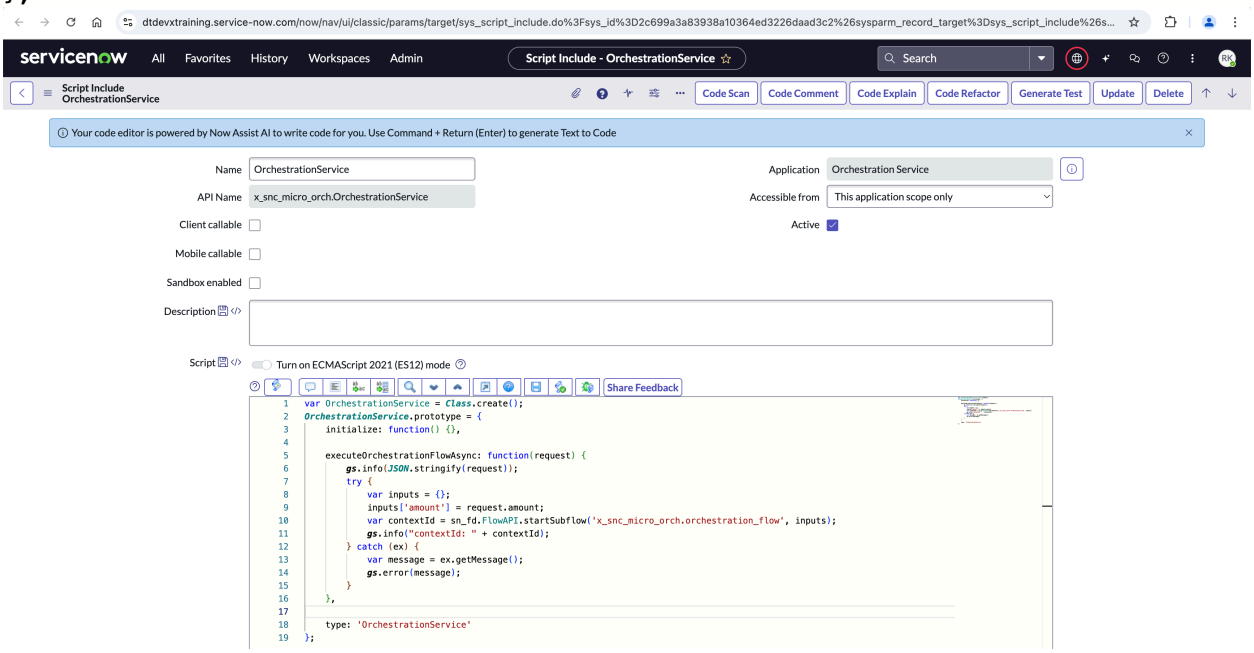


e) Update Order:



4. Create a Script Include named Orchestration service to start the subflow asynchronously.

```
var OrchestrationService = Class.create();
OrchestrationService.prototype = {
    initialize: function() {},

    executeOrchestrationFlowAsync: function(request) {
        gs.info(JSON.stringify(request));
        try {
            var inputs = {};
            inputs['amount'] = request.amount;
            var contextId =
sn_fd.FlowAPI.startSubflow('x_snc_micro_orch.orchestration_flow', inputs);
            gs.info("contextId: " + contextId);
        } catch (ex) {
            var message = ex.getMessage();
            gs.error(message);
        }
    },

    type: 'OrchestrationService'
};
```

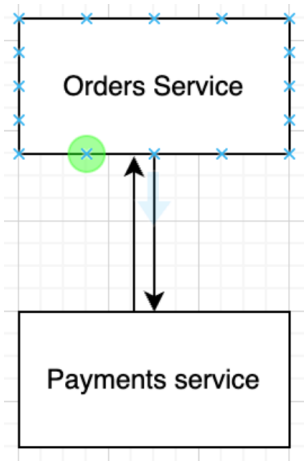


Note that we have an orders detail json in the flow. That is just to emulate an ideal world scenario. We will not be using it in this lab as the purpose is to show the interaction between services.

Now let's call our orchestration api and test the flow and data. If everything is successful, we should have an order record, a payment record and the state of the order record should be changed from pending to success/failed. We can see our orchestration subflow execution to see how the flow orchestrated interaction between these services and held some data in context.

Choreography SAGA:

What if we want to do away with the orchestration service and decide that our services should interact with each other asynchronously. We can do this via event driven architecture. Here we can call the create order api of the orders service and it will create an order. It will then create an event notifying the payment service to create the payment. The payment service after creating the payment can notify the order service that the order is successful via another event and it can also notify downstream services of the same. If any of the services fail, they can generate notifying events to revert the state in previous services. Here we have only two services and if the payment fails it can generate a payment failed event so that the order is also marked as failed.



Now let's create events for these. Please note that we are using ServiceNow events here but we can very well make these services interact via kafka messages and then we can swap out any of these services for cloud native ones and the other services will continue to work with it.

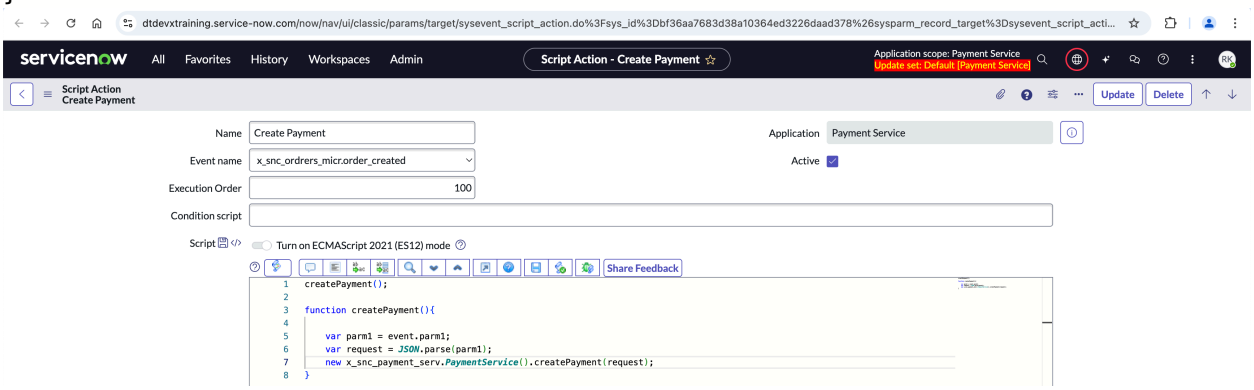
1. Create the event registrations for order created and payment status events like:

Order created:

Payment status:

2. Now we need to queue these events. We can go back to the orders service and can queue the order created event while creating the order in the create order method. We then need a script action named Create Payment with the following script to get triggered when that event is queued. This will then start a payment process.

```
createPayment();
function createPayment(){
    var parm1 = event.parm1;
    var request = JSON.parse(parm1);
    new x_snc_payment_serv.PaymentService().createPayment(request);
}
```



Note that this script action is part of the payment service app. The two services only create events notifying each other. They do not invoke each other's methods or code.

3. Then when the payment is successful/failed, we can create another event regarding the payment status in the payment order service. Then another script action will be triggered based on that and depending on payment status, order status will be updated.

```
updateOrder();
function updateOrder(){
    var parm1 = event.parm1;
    var paymentStatus = JSON.parse(parm1);
    var orderUpdateReq = {"orderid" : paymentStatus.orderid+"", "order_status" :
paymentStatus.payment_status+""};
    new x_snc_ordrers_micr.OrderService().updateOrder(orderUpdateReq);
}
```