Table of Contents

1. Introduction To Microservices

1.1. Get Started

In this section, you will get the architecture of this framework. Other than that, you will get what are the limitations you have to face when you are going to use microservice architecture and how overcome those difficulties by using StackSAGA framework.

1.2. Microservice Architecture

Before dive in to the framework, let’s have the basic idea of microservice architecture. A microservice' architecture consists of a collection of small, autonomous services. Each service is self-contained and should implement a single business capability within a bounded context. A bounded context is a natural division within a business and provides an explicit boundary within which a domain model exists.

1.3. What are microservices?

Microservices are a modern approach to software whereby application code is delivered in small, manageable pieces, independent of others. Read more through spring microservices.

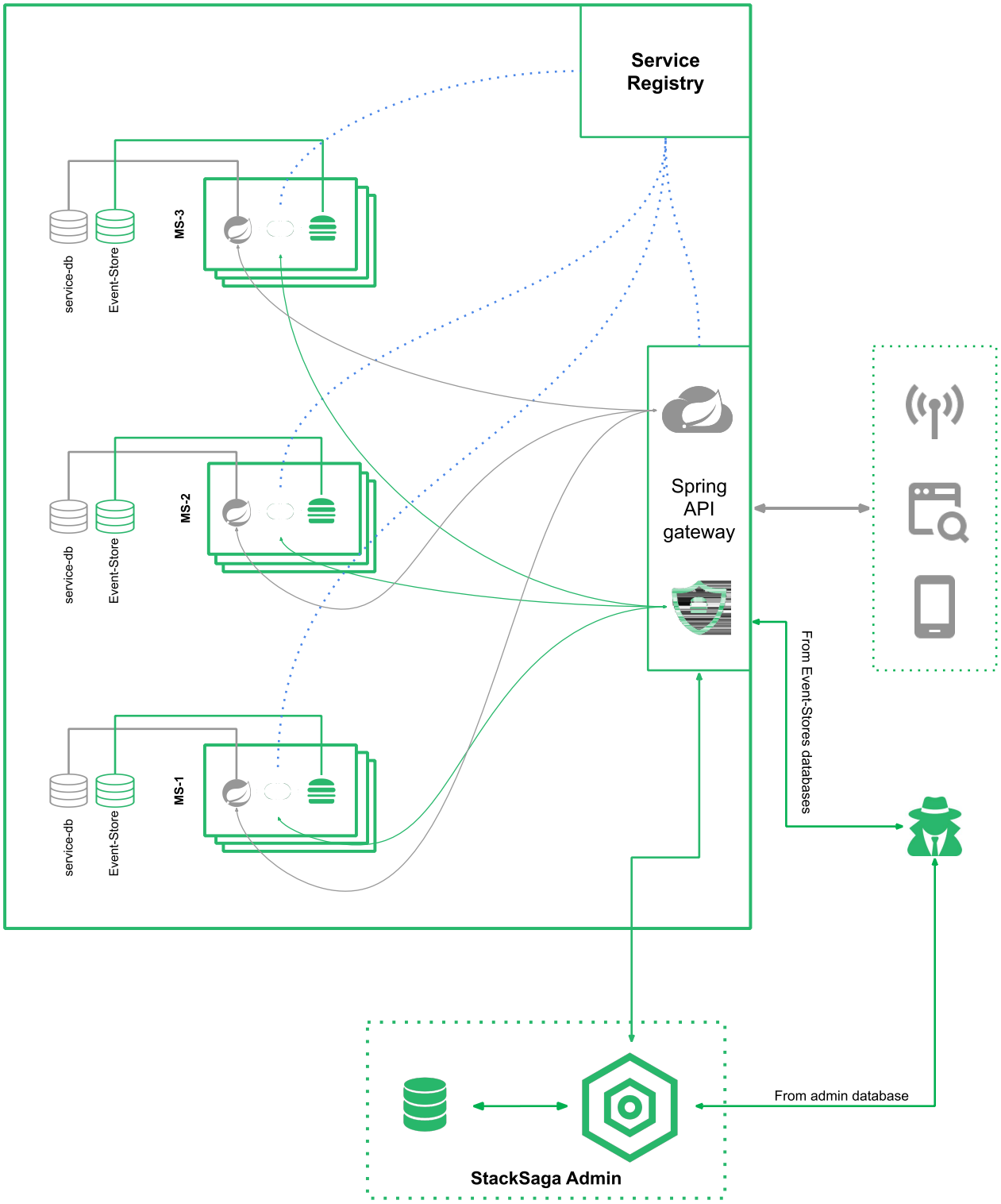
1.4. Database per Service Pattern

One of the benefits of microservice architecture is that it lets us choose the technology stack per service. For instance, we can decide to use a relational database for service A and opt for a NoSQL database for service B. This model lets the services manage domain data independently on a data store that best suites its data types and schema. Further, it also lets the service scale its datastore on-demand and insulates it from the failures of other services. However, at times a transaction can span across multiple services, and ensuring data consistency across the service database is a challenge. In the next section, let us examine the challenge of distributed transaction management with an example.

2. Architecture

2.1. StackSaga in High-level

in the introduction, we got a clear idea of how microservice architecture works, and what are the challenges that we have to face when implementing the microservice. here, we are going to explain how the stack saga works together with typical microservice architecture. here we have demonstrated a spring boot microservice architecture and how the StackSaga framework provides the Saga execution coordinator (SEC) capabilities without bothering to the default architecture.

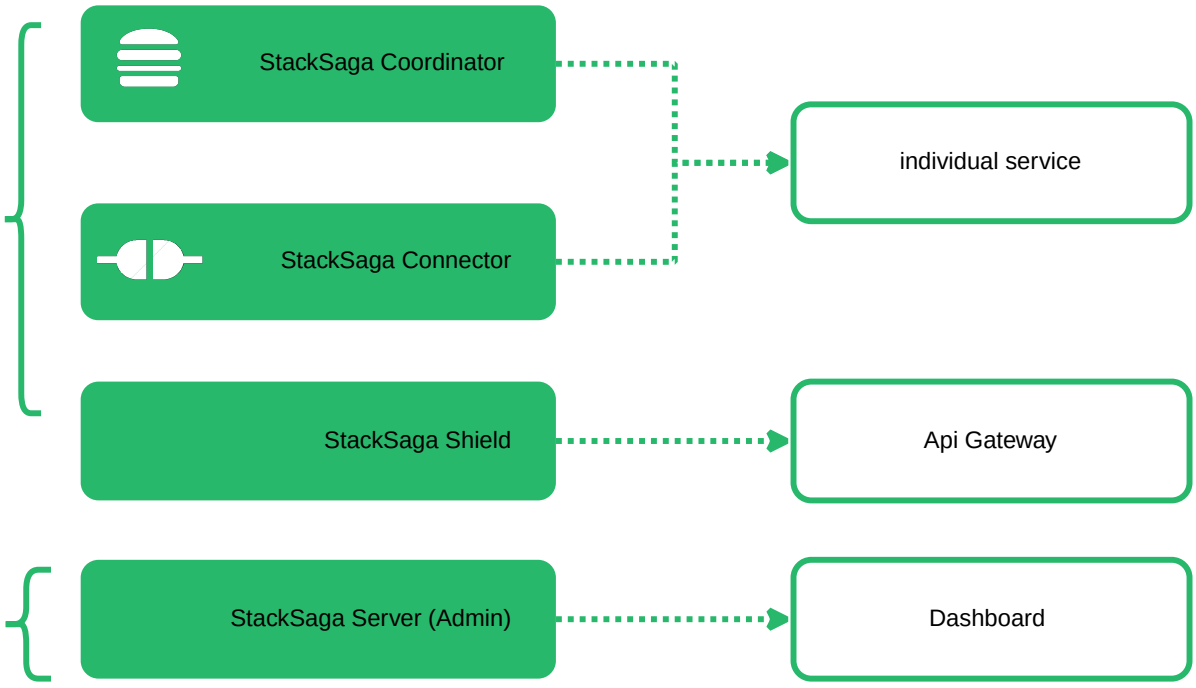


After adding stack saga in the microservice architecture, you can see there are some additional components in the high-level architecture diagram. In the diagram, the gray color components and

lines are related to the general microservice architecture and additional stacksaga related components and lines have been colored by green color. Let's discuss all the components and how they are interacted with general architecture. When you use StackSaga, you have to provide the facilities to create a database in addition to the database that you use for each microservice. This is also the same as the database per service design pattern. A service group (multiple instances of one service) uses one database to store the events of the executions. The only thing is newly added an extra schema. If you like to use one database for all microservices for StackSaga, it doesn't matter you can use one database as the event-store. But as the best practice, one database per service is recommended.

As a summary of the image above, you can identify mainly 4 components as new.in the next image below you can see those new components how to interact with the general components.

1. StackSaga Coordinator (A Library) and StackSaga Connector (A Library) is located in the individual services.
2. StackSaga Shield (A Library) is located in the API gateway.
3. StackSaga Server (A Standalone Application) is one separate unit that provides a dashboard.



Text is not SVG - cannot display

2.2. Key Processes of ecosystem

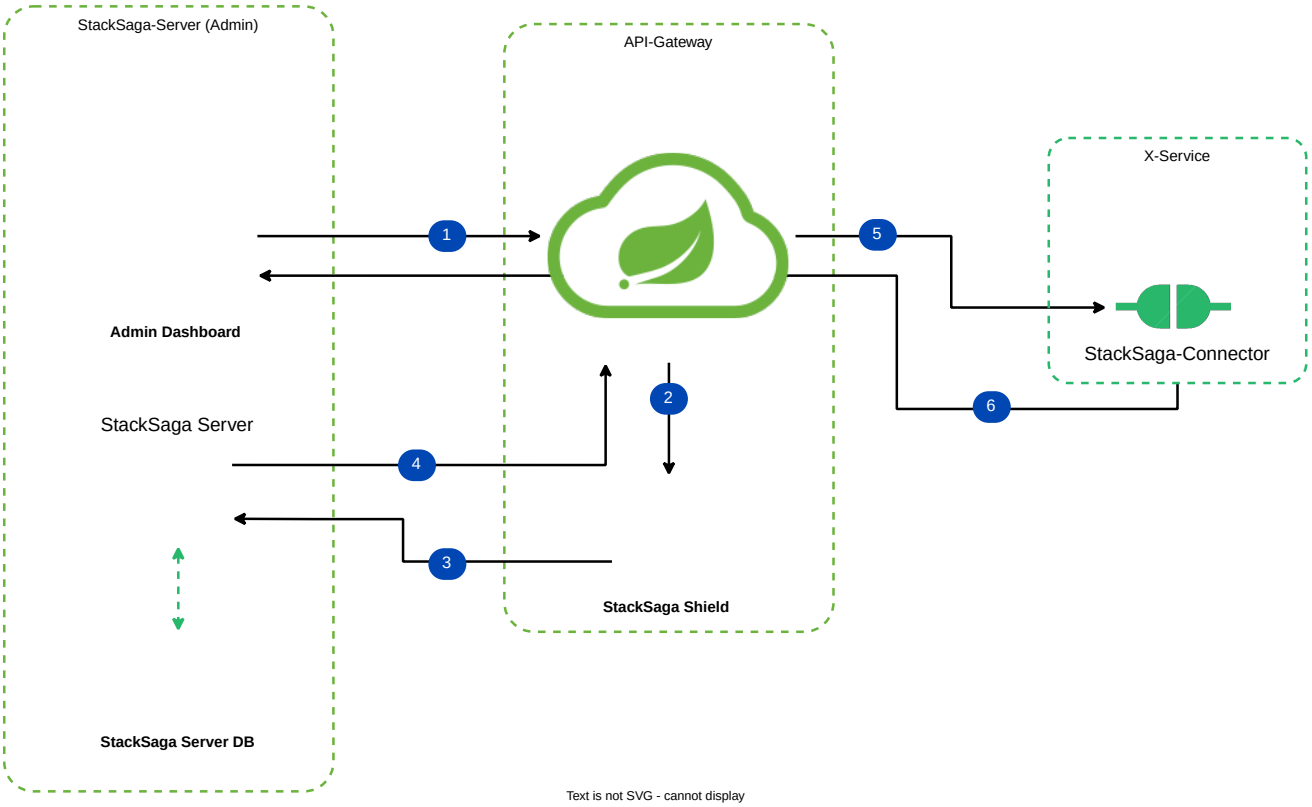
To understand the relation of those components whole process can be summarised as follows.

1. Initialize the admin server with super admin.
    - As the first step, you have to create initialize the admin server. See the initialization steps.
2. Register API gateways with admin server
    - To connect the StackSaga, your API gateway should provide an API-gateway-user credentials. Therefore, super Admin should create a user with API gateway authorization

(role). After creating the API gateway user, the api gateway can be run because StackSaga Shield will check your credentials connecting with the admin server. For this process, StackSaga server and StackSaga Shield were involved. See the implementation to see more in detail.

3. Register each microservice with admin server.
    - To register the services in the admin server, the super admin or admin has to create a user for service with service privileges. As a best-practice, the StackSaga team recommends you to create one user for one service group. (A service group can have multiple instances, but the service names are the same.) For instance, order-service should have a service user called order-service-general-user. And payment service should have a service user called payment-service-general-user. At the start-up, the connector will verify your service credentials and let you the access to register and run the service. For this process, StackSaga Server and StackSaga Connector were involved. See the implementation to see more in detail.
4. Execute a transaction.
    - This is the main part that you are willing to see. The request comes through the API gateway as usual, and the StackSaga coordinator will obtain the request data that you pass, and the coordinator will handle all the process and executions as you have guided the framework. For this process only StackSaga coordinator was involved. Read the implementation Or read how stack saga execution types work to see more, in detail.
5. Monitor transactions execution data.
    - To see the execution data, you should have to have a user account on the admin server. If you are the super admin, the super admin can see all the details. But as a best-practice, make sure to create a separate individual account for each user that wat to access the admin server to see the transaction data. After login to the dashboard, you have to give the access path of the API gateway that you want to access the services. You will be able to see the transaction data.

For this process, StackSaga server, StackSaga Shield and StackSaga Connector were involved.



2.3. Aggregator

An **aggregator** is the main location that all the data should be kept for the entire transaction. According to the StackSaga example, the entire process has set pf sub processes. Create order, check user activeness, make payment, increase points and dispatch the order. While those processes, you have to keep some data regarding the transaction. For instance, the process is started from creating order. After creating the order, you have a data to be kept. As a response, you will get the order id. You have to keep it, and you have to reuse it in your whole process. After creating the order, you check the user activeness. Then the user service is called and get the response of user's activeness and keep that data as user is active at the time of the process started. Likewise, each and every process will return you to data. Those data you have to keep in one location. That location called as the aggregator. At the start, you have to declare the variables, and after doing each process, you can update the values in the aggregator. On the other hand, we can say it is the data bucket that you bring the data to each executor.

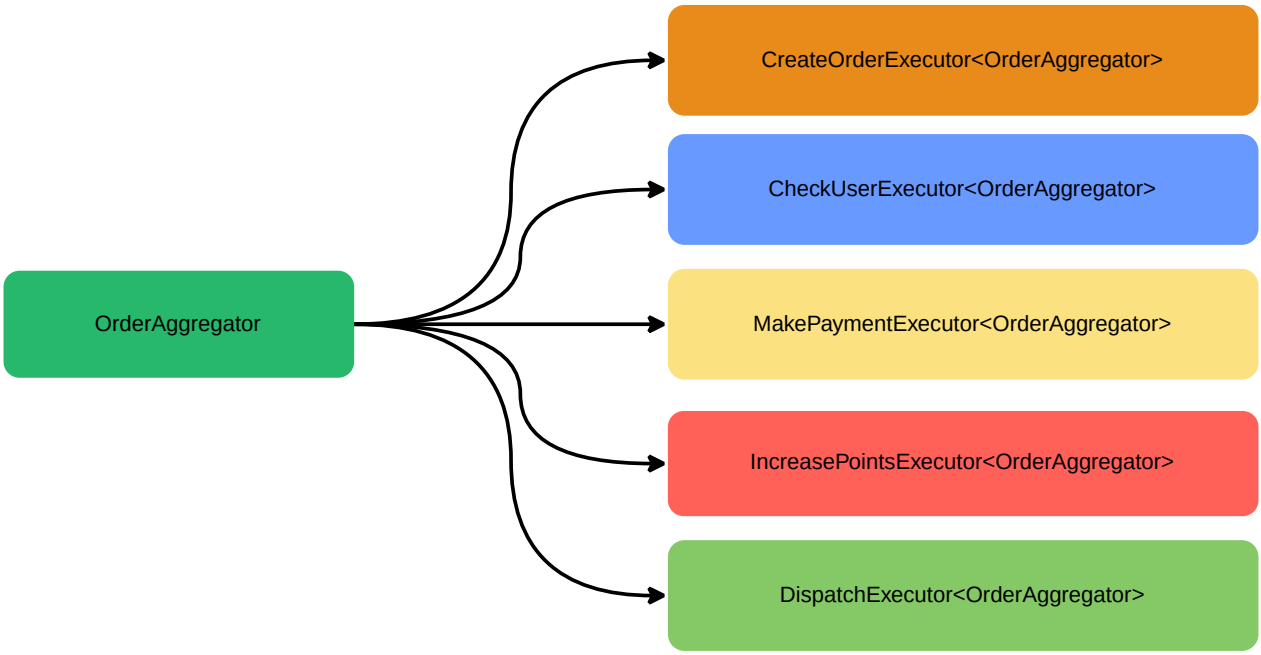
Another special thing is that the aggregator is used as the key of your entire transaction. That means the executors are identified by the aggregator class that you use. Because your entire transaction can have only one aggregator. Therefore, the aggregator class is used in each and every time by representing the transaction. For instance, according to the StackSaga example, the place order is the entire process. It contains many sub processes like create order, check user ect. Each process is introduced to the framework by using executors. So, for each process have separate executors. When you create each executor, you should mention what ks the aggregator that you want to use in that executor.

You can store/update the data in the aggregator by using the process execution only (doProcess()). If you want to keep some metadata while your revert/compensation processes, you can use the hint-store that the framework

> provides.

2.3.1. Aggregator and Executors

Here you can see all the sub sets of processes are connected with the aggregator. Each sub process wants the aggregator to get run. That means the data that executor wants to execute the sub process, is located at the aggregator. If some executor is not linked with the aggregator, it is not configured in the framework. You can see the executor-introduction or the real implementation of the example to learn more about the executors.



2.3.2. How the Aggregator is used through the Executors


initialize aggregator
OrderAggregator
String userUid = 001;
Double amount = 400.00;
String orderUid;
boolean userIsActive;
LocalDateTime userActiveAt;
String transactionId;
String trackingNumber;
INIT
CreateOrderExecutor<OrderAggregator>...
doProcess
doRevert
OrderAggregator
String userUid = 001;
Double amount =400.00;
String orderUid= 5566;
boolean userIsActive;
LocalDateTime userActiveAt;
String transactionId;
String trackingNumber;
01
CheckUserExecutor<OrderAggregator>...
doProcess
OrderAggregator
String userUid= 001;
Double amount = 400.00;
String orderUid = 5566;
boolean userIsActive = true;
LocalDateTime userActiveAt 13434314333
String transactionId;
String trackingNumber;
02
MakePaymentExecutor<OrderAggregator>...
doProcess
doRevert
OrderAggregator
String userUid= 001;
Double amount = 400.00;
String orderUid = 5566;
boolean userIsActive = true;
LocalDateTime userActiveAt 13434314333
String transactionId = 5577889;
String trackingNumber;
03
IncreasePointsExecutor<OrderAggregator>
doProcess
doRevert
OrderAggregator
String userUid= 001;
Double amount = 400.00;
String orderUid = 5566;
boolean userIsActive = true;
LocalDateTime userActiveAt 13434314333
String transactionId = 5577889;
String trackingNumber;
04
DispatchExecutor<OrderAggregator>...
doProcess
doRevert
OrderAggregator
String userUid= 001;
Double amount = 400.00;
String orderUid = 5566;
boolean userIsActive = true;
LocalDateTime userActiveAt 13434314333
String transactionId = 5577889;
String trackingNumber = 009989;
05
Text is not SVG - cannot display

3. Developer Guide

3.1. Aggregator Introduction

Here we are going to discuss how we can create an Aggregator and what are the configurations of the Aggregator.

3.2. Creating Aggregator Class

The aggregator[ref] class does represent your business domain, and it is the container for carrying out the data for the executors. In side of the aggregator class, you should provide all the data regarding your entire transaction.

example: Just think you are will be to make a transaction for place and order by using StackSaga. then you have to create an Aggregator class to put the data that you want to while the whole process. you can create an Aggregator class called `PlaceOrderAggregator` and provide all the data that you want like *username, orderId, transactionId* etc... and also you might want to add complex data structure inside that aggregate class like *list of items* regarding the items that the customer's order. then you have to create a java pojo as well. as a best practice, the framework suggests you to create all related pojo classes in the same package of the Aggregator class or as the inner classes of the Aggregator class.

To be an aggregator in StackSaga, Mainly, The aggregator class should be extended from `org.stacksaga.SagaAggregate` class and also the aggregator class should be annotated with `org.stacksaga.core.annotation.Aggregator`.

All the classes (main aggregator class and other classes that are used inside it) that are used for the SagaAggregator should be implemented by the `java.io.Serializable` class. By default, The main aggregator class is Serializable due to the fact that it is implemented from the `SagaAggregate` class. See complex object implementation

Let's see how it is created in StackSaga project.

```
@Aggregator(①
        version = @AggregatorVersion(major = 1, minor = 0, patch = 1), ②
        idPrefix = "po", ③
        name = "PlaceOrderAggregator", ④
        sagaSerializable = PlaceOrderAggregatorSample.class ⑤
)
@Getter ⑥
@Setter ⑦
public class PlaceOrderAggregator extends SagaAggregate  { ⑧

⑨
    public PlaceOrderAggregator() {
        super(PlaceOrderAggregator.class);
```

```
    }

    private String orderId;
    private String username;
    private int isActive;
}
```

① The aggregator class should be annotated with `org.stacksaga.core.annotation.Aggregator`.

② **version** will be used for the identification of the aggregator versioning. it is helpful for the **event-upper-casting** and **event-down-casting**. `org.stacksaga.core.annotation.AggregatorVersion` annotation help you to provide the version of the aggregator.

③ **idPrefix** will be used as the prefix of the transaction id. you can give a prefix according to the aggregator name. The maximum length of the prefix is 4 characters.

④ **name** The name of the aggregator. this is used for identification of the aggregator by the name.

⑤ **sagaSerializable** is used for pre-serialization of the samples of the aggregator. see the implementation

⑥ **Getter** you can create getters as usual for the aggregator.

⑦ **Setter** you can create setters as usual for the aggregator.

⑧ The aggregator class should be implemented from the `org.stacksaga.SagaAggregate` class. It provides the shep of aggregator in the framework.

⑨ The aggregator class should have the default constructor and also the **super()** method should be called by passing the same aggregator class.

The name of the aggregator should be the same forever. Because, all the event-store data is mapped with the aggregator name.

Due to the fact that the aggregator name is configured by an attribute, you can change the package of the aggregator class anytime.

In StackSaga, The aggregator is not a spring bean at all. Therefore, it is not necessary to have inside the **component scan** area.

3.3. Creating SagaSerializable Class for Aggregator

The `SagaSerializable` class provides you a method called `<T> getSamples()` to provide the sample array of objects of your real Aggregator. for each aggregator class that you made should have a separate implementation of `SagaSerializable` for validating the sample object before the application is started. that process ensures that your aggregator object will not face a serialization error while the transactions are being processed in the production.

you can see a sample SagaSerializable `(PlaceOrderAggregatorSample.class)` implementation of the PlaceOrderAggregator.

```
public class PlaceOrderAggregatorSample implements SagaSerializable
<PlaceOrderAggregator> { ①
    @Override
    public PlaceOrderAggregator[] getSamples() {
        //samples objects for the PlaceOrderAggregator

        //sample-1
        PlaceOrderAggregator aggregator1 = new PlaceOrderAggregator();
        aggregator1.setOrderId(UUID.randomUUID().toString());

        //sample-1
        PlaceOrderAggregator aggregator2 = new PlaceOrderAggregator();
        aggregator2.setOrderId(UUID.randomUUID().toString());
        aggregator2.setUsername("mafei");

②
        return new PlaceOrderAggregator[]{
                aggregator1,
                aggregator2,
        };
    }
}
```

① Implement your custom `PlaceOrderAggregatorSample` from the `SagaSerializable<T>`. The generic type should be the aggregator that you wish to check the sample object. According to the example, the Type is `PlaceOrderAggregator`. Then you will have a method for providing the sample objects for that given type.

② Rerun your initialized sample objects by creating a new array for the validation process.

Make sure to keep the default constructor for each SagaSerializable implementation.

3.4. Custom Aggregator Mapper

You can decide your custom configurations of the objectMapper for your target Aggregator. By default, the system will use an `ObjectMapper` that the framework provides for the aggregator serialization and deserialization. But if you want to customize the objectMapper for your target aggregator, you can create and provide a custom objectMapper object for the target aggregator by using `SagaAggregatorMapperProvider` implementation. Then The framework will use the custom Aggregator Mapper that you provided to the target aggregator.

The class should be a Spring bean (Annotate with `@Component`).

Here you can see a custom implementation of the `AggregatorMapper`.

```
@Component ①
public class PlaceOrderAggregatorJsonMapper implements SagaAggregatorMapperProvider
```

```
{②

    private final ObjectMapper objectMapper;

    public PlaceOrderAggregatorJsonMapper() {
        this.objectMapper = new ObjectMapper(); ③
        //your custom object mapper configurations
        ...
    }

④
    @Override
    public SagaAggregatorMapper getSagaAggregatorMapper() {
        return SagaAggregatorMapper.Builder.build(
                this.objectMapper
        );⑤
    }
}
```

① **@Component**: Mark your custom object mapper implementation as a Spring bean.

② **SagaAggregatorMapperProvider**: Implement by the `SagaAggregatorMapperProvider` interface.

③ Initialize and set your custom configurations for the `objectMapper` object that you provide for the target aggregator.

④ Override the method for providing the `SagaAggregatorMapper` object.

⑤ **SagaAggregatorMapper.Builder**: The builder class provides the methods for building `SagaAggregatorMapper` object. you can build `SagaAggregatorMapper` object by adding your custom configured objectMapper object and return it back.

The `SagaAggregatorMapper.Builder` class can have several build methods with several mapper types other than the `ObjectMapper`. Currently, StackSaga framework only supports `ObjectMapper`.

You can provide your custom object `SagaAggregatorMapperProvider` class to the target Aggregator like below.

```
@Aggregator(
        version = @AggregatorVersion(major = 1, minor = 0, patch = 1),
        idPrefix = "po",
        name = "PlaceOrderAggregator",
        sagaSerializable = PlaceOrderAggregatorSample.class,
        mapper = PlaceOrderAggregatorJsonMapper.class ①
)
@Getter
@Setter
public class PlaceOrderAggregator extends SagaAggregate {

    public PlaceOrderAggregator() {
        super(PlaceOrderAggregator.class);
```

```
    }
    ...
}
```

① **mapper**: provide your custom aggregator mapper provider class.

3.5. Full Aggregate implementation.

```
//Aggregtor Classs
@Aggregator(
        version = @AggregatorVersion(major = 1, minor = 0, patch = 1),
        idPrefix = "po",
        name = "PlaceOrderAggregator",
        sagaSerializable = PlaceOrderAggregatorSample.class
)
@Getter
@Setter
public class PlaceOrderAggregator extends SagaAggregate {

    public PlaceOrderAggregator() {
        super(PlaceOrderAggregator.class);
    }

    private String orderId;
    private String username;
    private int isActive;
    private List<ItemDetail> itemDetails = new ArrayList<>();
}

//Pojo Class for Item Details
@Getter
@Setter
@Builder
class ItemDetail implements Serializable { ①
    private String itemName;
    private int qty;
    private double price;
}

//Saga Serializable Class
class PlaceOrderAggregatorSample implements SagaSerializable<PlaceOrderAggregator> {
    @Override
    public PlaceOrderAggregator[] getSamples() {
        //sample-1
        PlaceOrderAggregator aggregator1 = new PlaceOrderAggregator();
        aggregator1.setOrderId(UUID.randomUUID().toString());
        //sample-1
        PlaceOrderAggregator aggregator2 = new PlaceOrderAggregator();
```

```
        aggregator2.setOrderId(UUID.randomUUID().toString());
        aggregator2.setUsername("mafei");
        //sample-3
        PlaceOrderAggregator aggregator3 = new PlaceOrderAggregator();
        aggregator3.getItemDetails().add(
                ItemDetail
                        .builder()
                        .itemName("Item-01")
                        .price(10.50)
                        .qty(2)
                        .build()
        );

        return new PlaceOrderAggregator[]{
                aggregator1,
                aggregator2,
                aggregator3,
        };
    }
}
```

① It is necessary to be implemented by the `Serializable`.

3.6. Saga Executors.

According to the saga design pattern, sub processes (atomic executions) can have two executions called process execution and compensation execution. And those executions you van create as the methods in your application anywhere.

1. **Query executors**: If some atomic process has no compensation (query execution), that kind of process is used in a query executor.
2. **Command executors**: If some atomic process has a compensation (command execution), that kind of process is used in a command executor.
    a. **Revert executors**: For the compensation process of the command executor it can be used other sub processes if you need.
        i. **Revert Before Executors**: If you want to execute an atomic process before executing the main revert process, you can create a Revert-Before-Executor.
        ii. **Revert After Executors**: If you want to execute an atomic process after executing the main revert process, you can create a Revert-Before-Executor.

3.6.1. Query Executor.

You can create any number of your custom Query-Executors regarding the Aggregator. Here you can see how you can create a Query-Executor for your `PlaceOrderAggregator`.

```
①
@SagaExecutor(
```

```
        executeFor = "user-service", ②
        liveCheck = true, ③
        value = "CheckUserExecutor" ④
)
public class CheckUserExecutor implements QueryExecutor<PlaceOrderAggregator> { ⑤

⑥
    @Override
    public ProcessStepManager<PlaceOrderAggregator> doProcess(
            ProcessStack processStack,
            PlaceOrderAggregator aggregator,
            ProcessStepManagerUtil<PlaceOrderAggregator> stepManagerUtil
    ) throws RetryableExecutorException, NonRetryableExecutorException {
        //execute the service and check the conndtion.

        //return or throw ⑦
    }
}
```

① **@SagaExecutor**: annotate your query executor with `@org.stacksaga.annotation.SagaExecutor` annotation. The annotation provides the spring bean capabilities and also another metadata for the StackSaga framework.

② **executeFor**: The name of the service that particular executor is going to be connected. it can be a service name of another service. if the service is spring boot service, make sure to keep the application name as its. because it will be helpful for the retry process.

③ **liveCheck**: When the retry process is executed, liveCheck is considered by the engine. if the target service is a spring boot service or registered service with the service registry, the availability is checked before executing the retry by the StackSaga engine.

④ **value**: The bean name of the executor. The executor is identified with this value by StackSaga engine. after configuring the bean name, it cannot be changed at all. if you want to the class package, it does not matter without changing the configured name.

⑤ **QueryExecutor<T>** IF the executor is Query one, The executor should be implemented by `org.stacksaga.executor.QueryExecutor<T>`. `T` should be the target Aggregator class for the domain.

⑥ Override the `doProcess()` method and put your execution code block that should be executed when the executor is executed by the StackSaga engine.

⑦ Finally, you can either return the next executor with the method that provides by the `ProcessStepManager`.

3.6.2. Command Executor.

```
①
@SagaExecutor(
        executeFor = "delivery-service", ②
        liveCheck = true, ③
        value = "DispatchOrderExecutor" ④
)
```

```
public class DispatchOrderExecutor implements CommandExecutor<PlaceOrderAggregator> {
⑤

⑥
    @Override
    public ProcessStepManager<PlaceOrderAggregator> doProcess(
            ProcessStack processStack, ⑦
            PlaceOrderAggregator aggregator, ⑧
            ProcessStepManagerUtil<PlaceOrderAggregator> stepManager ⑨
    ) throws RetryableExecutorException, NonRetryableExecutorException {
        //execute the service and check the conndtion.
        ...
        //return or throw ⑩
    }

    @RevertBefore(startFrom = DispatchRevertNotifierExecutor.class)  ⑪
    @RevertAfter(startFrom = DispatchRevertCompleteLogExecutor.class) ⑫
    @Override
    public void doRevert(
            ProcessStack processStack,⑬
            NonRetryableExecutorException nonRetryableExecutorException,⑭
            PlaceOrderAggregator aggregator, ⑮
            RevertHintStore revertHintStore ⑯
    ) throws RetryableExecutorException {
        //call the atomic process that you want to as the compensation,
        ...
    }
}
```

① **@SagaExecutor**: annotate your query executor with `@org.stacksaga.annotation.SagaExecutor` annotation. The annotation provides the spring bean capabilities and also another metadata for the StackSaga framework.

② **executeFor**: The name of the service that particular executor is going to be connected. it can be a service name of another service. if the service is spring boot service, make sure to keep the application name as its. because it will be helpful for the retry process.

③ **liveCheck**: When the retry process is executed, liveCheck is considered by the engine. if the target service is a spring boot service or registered service with the service registry, the availability is checked before executing the retry by the StackSaga engine.

④ **value**: The bean name of the executor. The executor is identified with this value by StackSaga engine. after configuring the bean name, it cannot be changed at all. if you want to the class package, it does not matter without changing the configured name.

⑤ **CommandExecutor<T>** IF the executor is command one, The executor should be implemented by `org.stacksaga.executor.CommandExecutor<T>`. `T` should be the target Aggregator class for the domain.

⑥ Override the `doProcess()` method and put your execution code block that should be executed when the executor is executed by the StackSaga engine.

⑦ **ProcessStack** Object provides all the executed execution until the process so far. It will help for

gen an idea about the execution history. And also you can get the decision of the execution base on the execution history.

⑧ **PlaceOrderAggregator**: The current aggregator state.

⑨ **ProcessStepManagerUtil<T>**: This object provides the methods for giving the navigation to the engine regarding the next step. it can be another executor (command or query) or a process completion.

⑩ Inside the method scope, you can provide the execution that should be executed when the StackSaga engine executes. And finally you can navigate to the next step by using the `ProcessStepManagerUtil` object.

⑪ **@RevertBefore**: If the executor has any revert-before-executors, you can mention that executor class with `@RevertBefore` annotation. According to the example, the revert before execution will be started from DispatchRevertNotifierExecutor.

⑫ **@RevertAfter**: If the executor has any revert-after-executors, you can mention that executor class with `@RevertAfter` annotation. According to the example, the revert after execution will be started from DispatchRevertCompleteLogExecutor.

⑬ **ProcessStack** The final process stack that was when the final execution was executed.

⑭ **NonRetryableExecutorException** The exception that caused the transition process stopping to forward.

⑮ **PlaceOrderAggregator** The final aggregator state when the NonRetryableExecutorException is thrown.

⑯ **RevertHintStore** If you want to add some data to the backward process, you can use the `RevertHintStore` to put the data.

3.6.3. Revert Before Executor.

```
①
@SagaExecutor(
        executeFor = "delivery-service",②
        liveCheck = true,③
        value = "DispatchRevertNotifierExecutor" ④
)
public class DispatchRevertNotifierExecutor implements RevertBeforeExecutor
<PlaceOrderAggregator, DispatchOrderExecutor> {⑤

⑥
    @Override
    public RevertBeforeStepManager<PlaceOrderAggregator, DispatchOrderExecutor>
doProcess(
            PlaceOrderAggregator aggregator,⑦
            ProcessStack previousProcessStack,⑧
            NonRetryableExecutorException processException,⑨
            RevertHintStore revertHintStore,⑩
            RevertBeforeStepManagerUtil<PlaceOrderAggregator, DispatchOrderExecutor>
```

```
revertStepManagerUtil ⑪
    ) throws RetryableExecutorException { ⑫
        return revertStepManagerUtil.next(NextBeforExecutor.class); ⑬
        //or
        return revertStepManagerUtil.complete(); ⑭
    }
}
```

① **@SagaExecutor**: annotate your query executor with `@org.stacksaga.annotation.SagaExecutor` annotation. The annotation provides the spring bean capabilities and also another metadata for the StackSaga framework.

② **executeFor**: The name of the service that particular executor is going to be connected. it can be a service name of another service. if the service is spring boot service, make sure to keep the application name as its. because it will be helpful for the retry process.

③ **liveCheck**: When the retry process is executed, liveCheck is considered by the engine. if the target service is a spring boot service or registered service with the service registry, the availability is checked before executing the retry by the StackSaga engine.

④ **value**: The bean name of the executor. The executor is identified with this value by StackSaga engine. after configuring the bean name, it cannot be changed at all. if you want to the class package, it does not matter without changing the configured name.

⑤ **RevertBeforeExecutor<A, E>**: The Revert Before Executor should be implemented from the `RevertBeforeExecutor<A,E>`. Generic **A** is the aggregator. Generic **E** is the command-executor of the given aggregator as the Generic **A**.

⑥ Override the `doProcess()` method that `RevertBeforeExecutor` provides. That is the method is executed by the StackSaga engine.

⑦ **PlaceOrderAggregator**: The final aggregator state when the process exception is thrown.

⑧ **ProcessStack**: The final process stack when the process exception is thrown.

⑨ **NonRetryableExecutorException**:The process-exception that was thrown by the last executor.

⑩ **RevertHintStore** If you want to add some data to the backward process, you can use the `RevertHintStore` to put the data as container.

⑪ **RevertBeforeStepManagerUtil<A, E>**: This object provides the methods for giving the navigation to the engine regarding the next step. it can be another revert-before-executor or a revert-before completion.

⑫ **RetryableExecutorException**: If the revert before execution is failed due to a retryable exception, you can provide that by warping with `RetryableExecutorException`.

⑬ **next()**: You can provide (only if you have) the next revert-before-executor with `revertStepManagerUtil.next()`.

⑭ **complete()**: If you don't have any revert-before-executor to be executed as next regarding the command-executor, you can return `revertStepManagerUtil.complete()`.

3.6.4. Revert After Executor.

```
①
```

```
@SagaExecutor(
        executeFor = "delivery-service",②
        liveCheck = true,③
        value = "DispatchRevertNotifierExecutor" ④
)
public class DispatchRevertCompleteLogExecutor implements RevertAfterExecutor
<PlaceOrderAggregator, DispatchOrderExecutor> {⑤

⑥
    @Override
    public RevertAfterStepManager<PlaceOrderAggregator, DispatchOrderExecutor>
doProcess(
            PlaceOrderAggregator aggregator,⑦
            ProcessStack previousProcessStack,⑧
            NonRetryableExecutorException processException,⑨
            RevertHintStore revertHintStore,⑩
            RevertAfterStepManagerUtil<PlaceOrderAggregator, DispatchOrderExecutor>
revertStepManagerUtil ⑪
    ) throws RetryableExecutorException { ⑫
        return revertStepManagerUtil.next(NextAfterExecutor.class); ⑬
        //or
        return revertStepManagerUtil.complete(); ⑭
    }
}
```

① **@SagaExecutor**: annotate your query executor with `@org.stacksaga.annotation.SagaExecutor` annotation. The annotation provides the spring bean capabilities and also another metadata for the StackSaga framework.

② **executeFor**: The name of the service that particular executor is going to be connected. it can be a service name of another service. if the service is spring boot service, make sure to keep the application name as its. because it will be helpful for the retry process.

③ **liveCheck**: When the retry process is executed, liveCheck is considered by the engine. if the target service is a spring boot service or registered service with the service registry, the availability is checked before executing the retry by the StackSaga engine.

④ **value**: The bean name of the executor. The executor is identified with this value by StackSaga engine. after configuring the bean name, it cannot be changed at all. if you want to the class package, it does not matter without changing the configured name.

⑤ **RevertAfterExecutor<A, E>**: The Revert After Executor should be implemented from the `RevertAfterExecutor<A,E>`. Generic **A** is the aggregator. Generic **E** is the command-executor of the given aggregator as the Generic **A**.

⑥ Override the `doProcess()` method that `RevertAfterExecutor` provides. That is the method is executed by the StackSaga engine.

⑦ **PlaceOrderAggregator**: The final aggregator state when the process exception is thrown.

⑧ **ProcessStack**: The final process stack when the process exception is thrown.

⑨ **NonRetryableExecutorException**:The process-exception that was thrown by the last executor.

⑩ **RevertHintStore** If you want to add some data to the backward process, you can use the `RevertHintStore` to put the data as container.

⑪ **RevertAfterStepManagerUtil<A, E>**: This object provides the methods for giving the navigation to the engine regarding the next step. it can be another revert-after-executor or a revert-after completion.

⑫ **RetryableExecutorException**: If the revert after execution is failed due to a retryable exception, you can provide that by warping with `RetryableExecutorException`.

⑬ **next()**: You can provide (only if you have) the next revert-after-executor with `revertStepManagerUtil.next()`.

⑭ **complete()**: If you don't have any revert-after-executor to be executed as next regarding the command-executor, you can return `revertStepManagerUtil.complete()`.

3.7. ProcessStack

3.8. RetryableExecutorException

If you are having an exception while processing the executor due to a connection issue or that kind of re-invokable error (The exceptions that can be occurred temporarily), you can throw a `RetryableExecutorException`. Then the StackSaga engine will temporarily pause the execution, and the transaction does expose to the next scheduler. Because, the engine knows that even though the executor has an exception at this moment, the execution can be retried and run again successfully due to the retractable error. IF you were unable to catch the exception properly, the execution might throw `RuntimeException` s. That kind of unhandled exception will be caught as non-retryable exception by the StackSaga engine. IF the exception is a **non-retryable** error, you can throw it by warping with `NonRetryableExecutorException`.

Retryable Executor Exceptions are allowed for the following executors.

Executor	DoProcess() Method	doRevert() Method
Query Executor	✓	✓
Command Executor	✓	✓
Revert Before Executor	✓	
Revert After Executor	✓	

It is very important to identify whether the exception is a retryable exception or not. If you don't identify the exceptions that are thrown when the execution is executed every time that an exception is thrown, the transaction will be stopped to forward. Therefore, it is necessary to capture and identify all the exceptions that are thrown inside the `doProcess()` method.

```
@SagaExecutor(
        executeFor = "user-service",
        liveCheck = true,
```

```
        value = "CheckUserExecutor"
)
@AllArgsConstructor
public class CheckUserExecutor implements QueryExecutor<PlaceOrderAggregator> {

    private final UserService userService;
    @Override
    public ProcessStepManager<PlaceOrderAggregator> doProcess(
            ProcessStack processStack,
            PlaceOrderAggregator aggregator,
            ProcessStepManagerUtil<PlaceOrderAggregator> stepManagerUtil
    ) throws RetryableExecutorException, NonRetryableExecutorException {

        try {

            ResponseEntity<UserDetailDto> userDetail = this.userService.getUserDetail
(aggregator.getUsername());
            ...
        } catch (FeignException.ServiceUnavailable unavailableException) {
            throw RetryableExecutorException.buildWith(unavailableException).build();
        }
    }

}
```

3.9. NonRetryableExecutorException

If you are having an exception while processing the executor that is not temporally occurred (The exceptions that have not any point of executing again), you can throw a `NonRetryableExecutorException`. Then the StackSaga engine will stop the execution forward, and start reverting back, according to the configuration.

IF you were unable to catch the exception properly, the execution might throw `RuntimeException` s. That kind of unhandled exception will be caught as **non-retryable** exception by the StackSaga engine as well. But if you want to have proper error handling, you have to catch exceptions properly.

Non-Retryable Executor Exceptions are allowed for the following executors.

Executor	DoProcess() Method	doRevert() Method
Query Executor	✓	✓
Command Executor	✓	✓
Revert Before Executor	✓	
Revert After Executor	✓	

You can see in the summary table, all the revert executions are not allowed `NonRetryableExecutorException` s. The reason for that is according to the StackSaga

> architecture, any **command-execution** can have revert execution (compensation execution). But all the **revert-executions** can have retryable-exception only. Because, even though the process can be failed, the revert execution cannot be failed at all. If the process is failed, there is a revert execution to have a compensation. But if the revert is failed, there are any other executions to have compensation for the reverting fail.
> **The summary is that only forward execution can have non-retryable-exception.**

```
@SagaExecutor(
        executeFor = "user-service",
        liveCheck = true,
        value = "CheckUserExecutor"
)
@AllArgsConstructor
public class CheckUserExecutor implements QueryExecutor<PlaceOrderAggregator> {

    private final UserService userService;

    @Override
    public ProcessStepManager<PlaceOrderAggregator> doProcess(
            ProcessStack processStack,
            PlaceOrderAggregator aggregator,
            ProcessStepManagerUtil<PlaceOrderAggregator> stepManagerUtil
    ) throws RetryableExecutorException, NonRetryableExecutorException {

        try {

            ResponseEntity<UserDetailDto> userDetail = this.userService.getUserDetail
(aggregator.getUsername());
            ...
        } catch (FeignException.ServiceUnavailable unavailableException) {
            throw RetryableExecutorException
                    .buildWith(unavailableException)
                    .build();
        } catch (FeignException.BadRequest badRequestException) {
①
            throw NonRetryableExecutorException
                    .buildWith(badRequestException)
                    .put("time", LocalDateTime.now())  ②
                    .put("reason", "BadRequest") ②
                    ...
                    .build();
        }
    }

}
```

① **NonRetryableExecutorException** is thrown due to the exception is a `BadRequest`. specially

`NonRetryableExecutorException` provides a way to add the metadata regarding the exception.

② Put the metadata regarding exception as a key and value.

In java, we usually pass the data regarding the exception with the exception object. But in the StackSaga it is **not recommended**. Because, any time any object can be serialized (even Exceptions) for saving to the event-store. Serializing an exception object (like Json or XML) is not easy, and it might be a performance issue without any benefit. And also most of the time if you pass a complex exception, it might throw an exception when it is serialized. Therefore, StackSaga engine just saves your exception as a string in the event-store(Only for seeing through the Admin-Dashboard). If you want to put some metadata to the backward regarding the exception, **NonRetryableExecutorException** has a method to put the data as key and value pare called `put(key, value)`. You can put any number of key/value pare into that by using the put method.

There is nothing like you can throw an exception when some exception is occurred. If you don't want to continue the execution of the transaction to forward, and if you decide that the transaction should be stopped at some point, due to a condition is not met, you can throw a **NonRetryableExecutorException** exception as well. See the full implementation

3.10. Usage of Executors[Q&C] With Exceptions

Here you can see how you can handle the **success**, **failure** and **retryable** scenario inside the **query-executor** executor. In the same way, you can handle all things in **command-executor** as well.

Create your own exception annotating with `@SagaException` class to handle to inactive user exception.

The executor gets the userdata from the user-service by calling another service. Based on the return of the `userService.getUserDetail` method, the next step is managed here. If the user is in active mode, The current executor allows the next executor by providing the next executor that should be invoked to the StackSaga engine.

```
@SagaExecutor(
        executeFor = "user-service",
        liveCheck = true,
        value = "CheckUserExecutor"
)
@AllArgsConstructor
public class CheckUserExecutor implements QueryExecutor<PlaceOrderAggregator> {

    private final UserService userService;

    @Override
    public ProcessStepManager<PlaceOrderAggregator> doProcess(
```

```
            ProcessStack processStack,
            PlaceOrderAggregator aggregator,
            ProcessStepManagerUtil<PlaceOrderAggregator> stepManagerUtil
    ) throws RetryableExecutorException, NonRetryableExecutorException {

        try {
            ResponseEntity<UserDetailDto> userDetail = this.userService.getUserDetail
(aggregator.getUsername());
            if (userDetail.getBody().getIsActive() == 1) {
                return stepManagerUtil.next(DispatchOrderExecutor.class); ①
            } else {
                UserInactiveException inactiveException = new UserInactiveException(
"User is not active!"); ②
                throw NonRetryableExecutorException
                        .buildWith(inactiveException)
                        .build();
            }
        } catch (FeignException.ServiceUnavailable unavailableException) {
            throw RetryableExecutorException
                    .buildWith(unavailableException)
                    .build(); ③
        } catch (FeignException.BadRequest badRequestException) {
            throw NonRetryableExecutorException
                    .buildWith(badRequestException)
                    .put("test", LocalDateTime.now())
                    .put("reason", "BadRequest")
                    .build(); ④
        }
    }
}
```

① **Success scenario**: If the user is active, the next executor will be provided to the engine by using `stepManagerUtil.next()` method.

② **Failed scenario**: IF the user is inactive, the process is stopped to forward by throwing `NonRetryableExecutorException` with `UserInactiveException`.

③ **Failed scenario**: IF the `userService.getUserDetail()` service throw a `FeignException.BadRequest` exception, the process is stopped to forward by throwing `NonRetryableExecutorException` with `FeignException.BadRequest` that occurred.

④ **Retryable scenario** IF the `userService.getUserDetail()` service throw a `FeignException.ServiceUnavailable` exception, the process is stopped temporally by throwing `NonRetryableExecutorException` with `FeignException.ServiceUnavailable` that occurred.

3.11. @SagaException Annotation

If you want to throw an exception as a NonRetryableExecutorException, The framework suggests you to create a new exception for that by warping the real exception and specially that can be annotated with `@SagaException` annotation like below.

Make sure to don't add the variables for exception metadata in your new exception class. Keep the exception class clear. Because, the framework does provide another way to provide the metadata in NonRetryableExecutorException

```
@SagaException(name = "UserInactiveException") ①
public class UserInactiveException extends RuntimeException {
    //for just throw an exception.
    public UserInactiveException() {
    }

    //for just throw an exception with message.
    public UserInactiveException(String message) {
        super(message);
    }

    //for wrapping the exception.
    public UserInactiveException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

① According to the usage of exceptions example, you can see this exception is used to throw if the user is inactive. And also if you think the package of the exception class will be changed in some cases in the future, you can set a fixed name for the exception. By annotating `@SagaException`.

At the first glance, you might think that there is no point in annotating by `@SagaException` when it is thrown. However, it is important when it is used in the revert executions (revert execution of the command executors or revert after or revert before). As you know, the revert process goes through revert method of each and every command executors that successfully executed in the past regarding the transaction. Then you have to check what is the real exception that was thrown, and what is the metadata that you want to make the decisions. As an example, just imagine that there is an event in the event-store to be executed. While the event is in the event-store, a new version is released of the particular service. And also the exception class has been moved to another package by mistake, or as a requirement. But while then, the old event is trying to be invoked through the StackSaga engine with old data. However, At this moment the exception class that caused the exception does not exist in the path when the exception was thrown initially.

you can use the following methods that `NonRetryableExecutorException` does provide for checking the exception in the revert process.

1. `getRealExceptionName()`: Get the name with the path of the exception class that was thrown. (This name is provided by the framework when the exception is thrown).
2. `getRealExceptionSimpleName()`: Get just the name of the exception class that was thrown. (This name is provided by the framework when the exception is thrown).
3. **`getRealSagaExceptionName()`**: Get the name that has been given with `@SagaException` annotation.

You can get the `Real Saga Exception Name` only if you have annotated with

> `@SagaException` annotation.

Here you can see an example of how you can use `SagaException` in the revert process.

```
@SagaExecutor(
        executeFor = "stock-management-service",
        liveCheck = true,
        value = "UpdateStockExecutor"
)
@AllArgsConstructor
public class UpdateStockExecutor implements CommandExecutor<PlaceOrderAggregator> {


    @Override
    public ProcessStepManager<PlaceOrderAggregator> doProcess(
            ProcessStack processStack,
            PlaceOrderAggregator aggregator,
            ProcessStepManagerUtil<PlaceOrderAggregator> stepManager
    ) throws RetryableExecutorException, NonRetryableExecutorException {
        ...
    }

    @Override
    public void doRevert(
            ProcessStack previousProcessStack,
            NonRetryableExecutorException realException,
            PlaceOrderAggregator finalAggregateState,
            RevertHintStore revertHintStore
    ) throws RetryableExecutorException {

        realException.getRealSagaExceptionName().ifPresent(realSagaExceptionName ->
{①
            if (realSagaExceptionName.equals("UserInactiveException")) {②
③
                realException.get("reason").ifPresent(reason -> {
                    if (reason.equalsIgnoreCase("BadRequest")) {
                        //do the revert process
                    }
                });
            }
        });
    }
}
```

① You can get the `RealSagaExceptionName` by calling `getRealSagaExceptionName()` method. And if you have annotated the exception class with `@SagaException` annotation when the exception was thrown, it will return a `Optional<String>` object with the name that has been mentioned in the `@SagaException` annotation. Or otherwise, the object will be empty.

② check the equality of Saga exception name with the exception that you want.

③ due to that, the exception is the exact same, you can get and check the metadata of the exception that you have given at the moment the exception was thrown.

3.12. RevertHintStore

`RevertHintStore` does provide you a map to put the data that the revert processes want. After having a NonRetryableExecutorException, the engine will start the revert process by stopping to forward anymore. You know already, when the transaction going forward, you can update or add the data in the aggregator object. But when it is starting to revert, you cannot change the values in the aggregator object anymore. Because the aggregator object is frozen. Therefore, you should have to have a way to carry out the data to the backwards as well. To overcome that problem, the system provides the RevertHintStore object to add your extra hints that are necessary for the revert process through each executor. After adding the data in the `RevertHintStore` object, you can read that data from other next revert-processes.

Here you can see an implementation for revert method of one of command executors. `RevertHintStore` can be used in all revert processes like revert-before, revert-after executors as well.

```
@SagaExecutor(
        executeFor = "delivery-service",
        liveCheck = true,
        value = "DispatchOrderExecutor"
)
public class DispatchOrderExecutor implements CommandExecutor<PlaceOrderAggregator> {


    @Override
    public ProcessStepManager<PlaceOrderAggregator> doProcess(
            ProcessStack processStack,
            PlaceOrderAggregator aggregator,
            ProcessStepManagerUtil<PlaceOrderAggregator> stepManager
    ) throws RetryableExecutorException, NonRetryableExecutorException {
        //
    }

    @Override
    public void doRevert(ProcessStack processStack,
                         NonRetryableExecutorException realException,
                         PlaceOrderAggregator aggregator,
                         RevertHintStore revertHintStore
    ) throws RetryableExecutorException {

        revertHintStore
                .get("MakePaymentExecutor") ①
                .ifPresent(makePaymentExecutor -> {
                    if (makePaymentExecutor.equalsIgnoreCase("success")) {
②
                        revertHintStore.put("last_updated_time", LocalDateTime.now()
.toString());
```

```
②
                    revertHintStore.put("DispatchOrderExecutor", "success");
                }
            });
        ...
    }
}
```

① Obtaining the `RevertHintStore` data (Key: MakePaymentExecutor) that have been added in another revert process before. If the value exists in the `RevertHintStore` you will have an `Optional<String>` object with the data.

② Based on the `RevertHintStore` 's old data that bes been added by another executor, adding another data into the `RevertHintStore`. This value will be helpful for the next revert process.

3.13. Saga Execution Event Listener

`SagaExecutionEventListener` is the place where you're notified about each and every action during the process by the SEC. As an example, you hand over your transaction to the StackSaga engine by mentioning the starting executor. After handing over, the engine will invoke your executors one by one. During the process, the engine does provide some events regarding the execution to notify. To get notified, you can create a separate EventListener classes for each and every Aggregator. For creating the EventListeners, the EventListeners should be implemented from `ExecutionEventListener<A>` interface. `A` is the Aggregator class that you want to get notified. And also the class should be annotated with **`@SagaExecutionEventListener`** or **`@SagaAsyncExecutionEventListener`**. Those annotations provide Spring bean capabilities for your implementation as well.

If you want to call the event methods in **Async-mode** inside the handler, the handler class can be annotated with `@SagaAsyncExecutionEventListener` instead of `@SagaExecutionEventListener`. Then it will run the event methods in separate threads asynchronously.

Make sure to not use spring `@Async` annotation with each method inside the listener or the entire class. Because StackSaga does create a separate thread pool for executing the event methods and also, you can customize it as per the requirements.

`ExecutionEventListener<A>` provides the following events to be notified.

1. **`onEachProcessPerformed`**
    - This method will be invoked after executing each sub process successfully. — Just think about our example. After payment process success, the order status should be updated as payment successful and also after successfully dispatched the order, the order status should be updated as order dispatched. And just think, you want to update your customer by sending an email after the payment process and after dispatching the process. Handler is the place that you can invoke your processes.

2. `onEachRevertPerformed`
    - This is the opposite of the onEachProcess. That works after every successful process. If the whole transaction has to be reverted at some point, the compensation process starts from that point. Then the revert method of each executed executor starts to be executed as the compensations. If you want to get notified after each compensation, onEachRevert method will be invoked by the framework with all the data that you want.
      According to the example, if the payment process is failed, you have to update the order status as payment failed. This kind of execution can be proceeded in this method.
3. `onTransactionCompleted`
    - There are two meanings of transaction-complete. One is that the transaction was processed without any process exception. When considering the example the create-order success, user checking is successful, make payment is success, increase point is success, and finally dispatching the order is also success. The 2nd success is, a process exception was happened and after that all the relevant compensation has been processed successfully. (That means a revert error hasn't been occurred while the compensation process) otherwise we can say the compensation has been processed smoothly without any exceptions.
4. `onProcessException`
    - This is not relevant to the sub process. That means the process exception can be happened only one time in the entire process. When considering the example, you are going to dispatch the order, and it is failed because of non-retryable exception. (Just think the order is not existing in the database) then the compensation process will be started. This is the turning point to compensation start. At that time, this method will be invoked by the framework with the data that you want.
5. `onTransactionTerminated`
    - According to the flow, any revert process can't have non-retryable exception. That is the rule. But sometimes the programmer may haven't been handled that. Then the revert process also can't be processed anymore. Then the framework marks that transaction as a garbage transaction and terminates the process. This is a very rare case. But it can be happened.

In `ExecutionEventListener`, all the abstract methods are default. Therefore, you don't want to override all the methods, and you can only override the methods that you want as per the requirement. And also you can create multiple listeners for the target Aggregator.

Here you can see an example how you can create a listener of `ExecutionEventListener`.

```
@SagaExecutionEventListener
public class PlaceOrderEventListener implements ExecutionEventListener
<PlaceOrderAggregator> {
    @Override
    public void onEachProcessPerformed(Class<? extends SagaExecutor<? extends
SagaAggregate>> processedExecutor, PlaceOrderAggregator currentAggregate) {
        //do whatever you want
    }
```

```
    @Override
    public void onEachRevertPerformed(Class<? extends SagaExecutor<? extends
SagaAggregate>> revertedExecutor, PlaceOrderAggregator finalAggregateState,
NonRetryableExecutorException nonRetryableExecutorException, RevertHintStore
revertHintStore) {
        //do whatever you want
    }

    @Override
    public void onTransactionCompleted(TransactionCompletedDetail<
PlaceOrderAggregator> transactionCompletedDetail, CompleteStatus completeStatus) {
        //do whatever you want
    }

    @Override
    public void onTransactionTerminated(TransactionTerminationDetail
<PlaceOrderAggregator> transactionTerminationDetail) {
        //do whatever you want
    }

    @Override
    public void onProcessException(PlaceOrderAggregator finalAggregateState,
NonRetryableExecutorException exception, Class<? extends SagaExecutor<? extends
SagaAggregate>> executorClass) {
        //do whatever you want
    }
}
```

3.14. SagaTemplate<A>

`SagaTemplate` is the class which is responsible for starting the StackSaga engine to invoke your transaction. By providing the necessary data, you can say to start the process of the transaction for StackSaga engine. `SagaTemplate` is an instance, and it provides one method called `process(…)` for starting the process.

`SagaTemplate` Is a spring bean that you can Autowire (Injectable) in Spring environment. You can autowire it by using spring @Autowire annotation or by creating the constructor.

```
@RestController
@RequestMapping("/order")
@AllArgsConstructor
public class PlaceOrderSagaController {

    private final SagaTemplate<PlaceOrderAggregator> sagaTemplate; ①

    @PostMapping
```

```
    @ResponseStatus(HttpStatus.ACCEPTED)
    public Map<String, String> placeOrder() {
②
        PlaceOrderAggregator aggregator = new PlaceOrderAggregator();
        aggregator.setUsername("mafei");
        aggregator.setTotal(200.00);

③
        this.sagaTemplate.process(
                aggregator,
                CheckUserExecutor.class
        );
④
        return Collections.singletonMap(
                "order_id",
                aggregator.getAggregateTransactionId()
        );
    }
}
```

① Autowire(inject) the `SagaTemplate` by providing the target Aggregator Class.

② Initialize your aggregator by providing the initial data.

③ Use the autowired `SagaTemplate` object and call the `process(..)` method. As the first argument, you have to provide the initialized target aggregator object. As the second argument you have to provide, which is the first executor that should be started the transaction. According to the example, The 1st executor is `CheckUserExecutor.class`. It can be either Command-Executor or Query-Executor.

④ Returns the response by obtaining the aggregatorTransactionId from the initialized object.

Due to the fact that the target aggregator class has been extended from the `SagaAggregate`, you can use inherited methods from `SagaAggregate` class. `getAggregateTransactionId()` method is one of inherited methods from `SagaAggregate` class. It will provide you the **unique ID** for each object (each transaction) that you initialize. It will be the transaction id for the entire transaction.

Even though `SagaTemplate<A>` can be autowired anywhere in the application, As the best practice it is recommended to use `SagaTemplate<A>` inside of an `EventHandler` class according to the StackSaga design pattern called CHES.

3.15. TrashFileListener

`TrashFileListener` is used to be notified about the **Trash-Files**.

3.16. Custom Thread-pool configuration

In StackSaga framework, there are several thread-pools are used behind the scene with default configurations. All the executions are not executed with a single thread-pool to ensure the resiliency of the application.

Table 1. Thread-pools in StackSaga framework

pool-name	Provider Interface	dependency	default-implementation	prefix	core pool size	max pool size	Queue Capacity	WaitOnShutdown
`SagaDiscoveryTransactionTaskExecutor`	`SagaDiscoveryTransactionTaskExecutorProvider`	`stacksaga-spring-boot-starter-core`	`SagaDiscoveryTransactionTaskExecutorProviderDefault`	**saga-tx-**	Available Processors * 1	Available Processors * 3	Default	True
`SagaEventTaskExecutor`	`SagaEventTaskExecutorProvider`	`stacksaga-spring-boot-starter-core`	`SagaEventExecutorProviderDefault`	**saga-event-**	Available Processors * 1	Available Processors * 2	Default	True
`SagaAdminTaskExecutor`	`SagaAdminTaskExecutorProvider`	`stacksaga-spring-boot-starter-discovery`	`SagaAdminTaskExecutorProviderDefault`	**saga-admin-**	2	5	Default	True
`SagaDiscoveryFileTaskExecutor`	`SagaDiscoveryFileTaskExecutorProvider`	`stacksaga-spring-boot-starter-discovery`	`SagaDiscoveryFileTaskExecutorProviderDefault`	**saga-file-**	Available Processors * 1	Available Processors * 2	Default	True
`SagaDiscoveryRetryTransactionTaskExecutor`	`SagaDiscoveryRetryTransactionTaskExecutorProvider`	`stacksaga-spring-boot-starter-discovery`	`SagaDiscoveryRetryTransactionTaskExecutorProviderDefault`	**saga-R-tx-**	Available Processors * 1	Available Processors * 2	Default	True

In case if you want to change the default configuration, you can do it by implementing following

3.16.1. Saga Discovery Transaction TaskExecutor

`SagaDiscoveryTransactionTaskExecutor` is the main pool. because this thread-pool is used for starting your transaction when it is called the `SagaTemplate.process()` method. after starting the transaction, this thread-pool will handle all the executions of the executors.

— In case if you want to customize the default configuration, you can update the default configuration as follows:

```
@Component ①
public class CustomSagaDiscoveryTransactionTaskExecutor
    implements SagaDiscoveryTransactionTaskExecutorProvider { ②

    @Override
    public ThreadPoolTaskExecutor getTaskExecutor() {
③
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(Runtime.getRuntime().availableProcessors() * 5);
        ...
        return executor;
    }
}
```

① Mark the class as a spring `@Component` (bean).

② Implement the custom task-executor provider class from `SagaDiscoveryTransactionTaskExecutorProvider` and override the `getTaskExecutor()` method.

③ Provide your custom task-executor configurations by creating a new instance of `ThreadPoolTaskExecutor`.

Even though you change the prefix of the thread-pool when the task-executor is created, the prefix is restored for maintaining the thread-name uniqueness. See the prefix of each thread-pool.

3.16.2. Saga Event TaskExecutor

— In case if you want to customize the default configuration, you can update the default configuration as follows:

```
@Component ①
public class CustomSagaEventTaskExecutor implements SagaEventTaskExecutorProvider { ②
    @Override
    public ThreadPoolTaskExecutor getTaskExecutor() {
③
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(Runtime.getRuntime().availableProcessors() * 5);
        ...
        return executor;
    }
}
```

① Mark the class as a spring `@Component` (bean).

② Implement the custom task-executor provider class from `SagaEventTaskExecutorProvider` and override the `getTaskExecutor()` method.

③ Provide your custom task-executor configurations by creating a new instance of `ThreadPoolTaskExecutor`.

Even though you change the prefix of the thread-pool when the task-executor is created, the prefix is restored for maintaining the thread-name uniqueness. See the prefix of each thread-pool.

3.16.3. Saga Admin TaskExecutor

— In case if you want to customize the default configuration, you can update the default configuration as follows:

```
@Component ①
public class CustomSagaAdminTaskExecutor
    implements SagaAdminTaskExecutorProvider {②

    @Override
    public ThreadPoolTaskExecutor getTaskExecutor() {
③
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(Runtime.getRuntime().availableProcessors() * 5);
        ...
        return executor;
    }
}
```

① Mark the class as a spring `@Component` (bean).

② Implement the custom task-executor provider class from `SagaAdminTaskExecutorProvider` and override the `getTaskExecutor()` method.

③ Provide your custom task-executor configurations by creating a new instance of `ThreadPoolTaskExecutor`.

Even though you change the prefix of the thread-pool when the task-executor is created, the prefix is restored for maintaining the thread-name uniqueness. See the prefix of each thread-pool.

3.16.4. Saga Discovery File TaskExecutor

— In case if you want to customize the default configuration, you can update the default configuration as follows:

```
@Component ①
public class CustomSagaDiscoveryFileTaskExecutor
    implements SagaDiscoveryFileTaskExecutorProvider { ②
```

```
    @Override
    public ThreadPoolTaskExecutor getTaskExecutor() {
③
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(Runtime.getRuntime().availableProcessors() * 5);
        ...
        return executor;
    }
}
```

① Mark the class as a spring `@Component` (bean).

② Implement the custom task-executor provider class from `SagaDiscoveryFileTaskExecutorProvider` and override the `getTaskExecutor()` method.

③ Provide your custom task-executor configurations by creating a new instance of `ThreadPoolTaskExecutor`.

Even though you change the prefix of the thread-pool when the task-executor is created, the prefix is restored for maintaining the thread-name uniqueness. See the prefix of each thread-pool.

3.16.5. Saga Discovery Retry Transaction TaskExecutor

Even though the transaction is started by the `SagaDiscoveryTransactionTaskExecutor` thread-pool, if the transaction is paused due a `RetryableExecutorException`, the transaction should be retried again after some time (configured schedule). For those executions, the `SagaDiscoveryRetryTransactionTaskExecutor` thread-pool is used.

It ensures the application's resiliency. And also the application can receive the new transaction without interrupting the transaction retry-bulk. The summary is that the retry execution does not disturb to the brand-new transaction.

— In case if you want to customize the default configuration, you can update the default configuration as follows:

```
@Component ①
public class CustomSagaDiscoveryRetryTransactionTaskExecutor
    implements SagaDiscoveryRetryTransactionTaskExecutorProvider {②

    @Override
    public ThreadPoolTaskExecutor getTaskExecutor() {
③
        ThreadPoolTaskExecutor executor = new ThreadPoolTaskExecutor();
        executor.setCorePoolSize(Runtime.getRuntime().availableProcessors() * 5);
        ...
        return executor;
```

```
    }
}
```

① Mark the class as a spring `@Component` (bean).

② Implement the custom task-executor provider class from `SagaDiscoveryRetryTransactionTaskExecutorProvider` and override the `getTaskExecutor()` method.

③ Provide your custom task-executor configurations by creating a new instance of `ThreadPoolTaskExecutor`.

Even though you change the prefix of the thread-pool when the task-executor is created, the prefix is restored for maintaining the thread-name uniqueness. See the prefix of each thread-pool.

3.17. Custom Admin Connector configuration

As per the architecture, all the StackSaga clients connect to the StackSaga admin server initially (**for submitting the instance metadata**) and as well as during the StackSaga-client is being run (**for submitting the terminated transaction data**).

IF you want to connect your service with the StackSaga admin server, you have to have a **service-account** that has been created by the StackSaga admin. IF you have a **service-account**, you can provide your service-account username and password for the basic authentication purpose.

In the microservice architecture, you will have hundreds of instances running on. It is enough to have a one service-account for one service-name. (The `spring.application.name` is considered as the service-name.)

After creating the `service-account`, you will have the basic authentication for the registered service. Those basic authentication can be provided as follows in the configuration property file:

```
stacksaga.connect.admin-url=http://localhost:4444
stacksaga.connect.admin-username=order-service
stacksaga.connect.admin-password=*****************
```

In addition to that, if you want to provide more complex configurations for the http client (RestTemplate) like enable custom SSL configurations, You can provide your own `RestTemplate` for the StackSaga framework to use when the client connects to the admin-server.

```
@Component ①
public class CustomAdminConnectRestTemplate implements
SagaAdminConnectRestTemplateProvider {②
    @Override
    public RestTemplate getRestTemplate() {
③
```

```
        RestTemplate restTemplate = new RestTemplate();
        //provide your custom configurations
        return restTemplate;
    }
}
```

① Annotate

② Implement the custom RestTemplate provider class from `SagaAdminConnectRestTemplateProvider`.

3.18. Custom Service Communication configuration