Introduction

Evento Framework is an Ecosystem of software built to implement JavaEE Reactive Systems applying RECQ Patterns, allowing you to build Enterprise applications clearly and efficiently following the state-of-the-art for distributed systems.

With this framework, you can implement Responsive, Resilient, Elastic, Maintainable, Extensible and Observable Microservices Architectures by applying concepts like CQRS (Command Query Responsibility Separation), Evento Sourcing and DDD (Domain Driven Design)

Please visit the Evento Framework website to learn more about the project, leave a start on GitHub and follow the author on LinkedIn to support us!

Architecture Overview

WIP

Distributed Systems & Microservices

Distributed systems are networks of interconnected computers that work together to achieve a common goal. They distribute tasks and data across multiple machines for improved performance, fault tolerance, and scalability. Distributed systems require careful coordination and communication to ensure consistency and synchronization of data. They provide a robust framework for efficient resource utilization and support the demands of modern computing environments.

Microservicesarchitecture is an approach to building applications as a collection of **small**, **loosely coupled**, and **independently deployable services**. Each service in a microservices architecture focuses on a **specific business capability**, and they **communicate with each other** through well-defined APIs.

https://microservices.io/patterns/microservices.html

Recative Manifesto & Reactive Principles

The Reactive Manifestooutlines principles for building responsive, resilient, elastic, and message-driven systems. It emphasizes timely feedback, fault tolerance, dynamic scalability, and efficient communication. By following these principles, developers can create applications capable of **handling** high concurrency, adapting to changing conditions, and delivering reliable user experiences.

The Reactive Principlesis a document that provides guidance and techniques established among experienced Reactive practitioners for building individual ser-

vices, applications, and whole systems. As a companion to the Reactive Manifesto it incorporates the ideas, paradigms, methods, and patterns from both Reactive Programming and Reactive Systems into a set of practical principles that software architects and developers can apply in their transformative work.

https://www.reactivemanifesto.org/

State-of-the-art Patterns

DDD (Domain Driven Design) is a design approach based on the domain model to analyze, design and implement systems for scalability and maintainability.

CQRS (Command Query Responsibility Separation) is a design pattern that separates the read and write operations of an application, optimizing performance, scalability, and flexibility.

Event Sourcingis a pattern where the state of an application is derived by replaying events. Events are stored as a log, enabling auditability, temporal queries, and reconstructing state at any point in time.

Messagingpattern involves communication between microservices through asynchronous message passing using message queues or event-driven systems, promoting loose coupling, scalability, and fault tolerance in distributed systems.

Sagais a design pattern for handling long-lived transactions in distributed systems. It ensures eventual consistency by coordinating a sequence of local transactional steps across multiple services.

Quick Start

Evento Server

To start building a RECQ based Architecture you need a Message Gateway to handle and manage message communication between components (microservices). To do this we use Broken link.

To start using Evento Server you need a Postgres Database and an instance of Evento Server that you can find on Docker Hub: https://hub.docker.com/r/eventoframework/evento-server

We have also prepared a simple docker-compose.yml to set up your development environment:

```
version: '3.3'
services:
  evento-db:
    image: 'postgres:latest'
    restart: always
```

```
    environment:
      - POSTGRES_PASSWORD=secret
      - POSTGRES_DB=evento
    volumes:
      - ./data/postgres:/var/lib/postgresql/data/
  evento-server:
    image: 'eventoframework/evento-server:latest'
    privileged: true
    restart: on-failure
    depends_on:
      - evento-db
    environment:
      # Cluster name visualized on the GUI
      - evento_cluster_name=evento-server
      # Capture rate for internal telemetry
      - evento_performance_capture_rate=1
      # Telemetry data TTL
      - evento_telemetry_ttl=365
      # Upload directory for Bundle Registration
      - evento_file_upload-dir=/server_upload
      # Secret key used to generate JWT access tokens
      - evento_security_signing_key=MY_JWT_SECRET_TOKEN_SEED
      # Evento Deploy Spawn Script Path
      - evento_deploy_spawn_script=/script/spawn.py
      # Postgres Database Connection Parameters
      - spring_datasource_url=jdbc:postgresql://evento-db:5432/evento
      - spring_datasource_username=postgres
      - spring_datasource_password=secret
    ports:
      - '3000:3000'
      - '3030:3030'
    volumes:
      - ./data/evento/files:/server_upload
      - ./docker-spawn.py:/script/spawn.py
```

You need to specify a Script for the automatic bundle deployment, add an empty Python script and bind it, it will be fine at the start.

Evento Framework

To develop RECQ components you need the Broken link Bundle Library.

{% hint style=“danger” %} Evento Framework is compatible withJava 21 or more. {% endhint %}

You can find the library on Maven Central: https://central.sonatype.com/artif

act/com.eventoframework/evento-bundle

Gradle

```
implementation group: 'com.eventoframework', name: 'evento-bundle', version: 'ev1.6.0'
```

Maven

```
<dependency>
    <groupId>com.eventoframework</groupId>
    <artifactId>evento-bundle</artifactId>
    <version>ev1.6.0</version>
</dependency>
```

{% hint style="info" %} Evento framework is independent of any other structured known framework like Spring, Micronautor Quarkus, so you can implement a RECQ application using your preferred technology even plain JavaEE. {% endhint %}

Tu understands how to use properly Evento Server and Evento Framework we suggest you follow our Tutorial in the next chapter.

TodoList - RECQ Tutorial

In this tutorial, you will explore how to design and implement a RECQ Application starting from the Requirement Gathering straight to the REST API implementation in Java using Evento Framework and Spring Boot.

Structure

- Problem Description and Requirement Gathering
- RECQ Payload Design
- RECQ Components Design
- Set up your Development Environment
- RECQ Payload Evento Implementation
- RECQ Components Evento Implementation with Spring Data
- Expose the RECQ architecture with Spring Web
- Test Your App
- Functional Analysis - Event Storming Approach (Bonus Chapter)

Problem Description and Requirement Gathering

We need to implement a Todo list application, only the backend in terms of REST API.

Problem Description

We need to implement a Todo List app where a user can create Todo Lists. Each Todo list has a name. Each Todo list contains a collection of a maximum of five todos with a proper description and a checked flag. Once a todo is checked we cannot delete or edit it and also we cannot delete the Todo list containing it. We want to keep auditing each action knowing who has created or edited a to-do list and each contained todo.

Todo list concept

Requirement Gathering

To analyse the problem we need to decompose the description in a list of requirements. Let's split the analysis into two sides: Domain and Constrains, in the next chapter we will see another technique.

Domain Starting from the prompt we need to individuate all the entities involving the domain:

> We need to implement a Todo List app where a user can create Todo Lists. Each Todo list has a name. Each Todo list contains a collection of a maximum of five todos with a proper description and a checked flag. Once a todo is checked we cannot delete or edit it and also we cannot delete the Todo list containing it. We want to keep auditing each action knowing who has created or edited a to-do list and each contained todo.

By a term inspection, we can individuate three entities: TodoList, Todo and User.

With a second inspection, we can define properties and relations between entities:

- TodoList
  - name - the list name
  - todos - the collection of Todo
- Todo
  - description -the todo description
  - checked - the flag indicating the completion
- User
  - identifier - we do not have enough information about it we can use the username

---

In the end, we need to identify non-functional fields and technical ones.

- TodoList

  - identifier - the list UUID
  - name - the list name
  - todos - the collection of Todo
  - createdAt - creation audit
  - createdBy - creation audit
  - updatedAt - update audit
  - updatedBy - update audit
- Todo
  - identifier - the list UUID
  - description -the todo description
  - checked - the flag indicating the completion
  - createdAt - creation audit
  - createdBy - creation audit
  - checkedAt - check audit
  - checkedBy - check audit
- User
  - identifier - we do not have enough information about it we can use the username

---

Functional Requirements Once we've analysed our domain and all the information let's express and formalize any requirement in the form of user stories:

- As a user, I want to create a to-do list in order to fill it with todos.
- As a user, I want to delete a to-do list that does not contain checked todos because now is useless.
- As a user, I want to add a to-do inside a to-do list to check it later.
- As a user, I want to remove a todo from a todo list because it will never be checked.
- As a user, I want to check a to-do inside a to-do list to mark it as done.
- As a user, I want to get a list of all the Todo Lists in the systems to explore them.
- As a user, I want to get the details of a to-do list in order to know every todo status.

RECQ Payload Design

To design a RECQ architecture we need to define the four main payloads that describe actions and data.

To do this we need to follow the RECQ Communication Pattern that forces us to describe our system's interactions in terms of distinguished message classes: Commands, Events, Queries and Views.

By analysing each requirement and given the domain we can define these payloads:

- As a user, I want to create a to-do list to fill it with todos.
  - TodoListCreateCommand
  - TodoListCreatedEvent
- As a user, I want to delete a to-do list that does not contain checked todos because now is useless.
  - TodoListDeleteCommand
  - TodoListDeleteEvent
- As a user, I want to add a to-do inside a to-do list to check it later.
  - TodoListAddTodoCommand
  - TodoListTodoAddedEvent
- As a user, I want to remove a todo from a todo list because it will never be checked.
  - TodoListRemoveTodoCommand
  - TodoListTosoRemovedEvent
- As a user, I want to check a to-do inside a to-do list to mark it as done.
  - TodoListCheckTodoCommand
  - TodoListTodoCheckedEvent
- As a user, I want to get a list of all the Todo Lists in the systems to explore them.
  - TodoListListItemViewFindAllQuery
  - TodoListListItemView
- As a user, I want to get the details of a to-do list to know every to-do status.
  - TodoListViewFindByIdentifierQuery
  - TodoListView

RECQ Components Design

Now that you have defined all the Messages you need to identify the components handling those messages.

We need to choose between:

- Aggregate
- Projector
- Projection
- Invoker
- Service
- Saga
- Observer

Aggregate

Let's start from the Domain logic handling aspects of the application, we have previously individuated the TodoList domain composed of the TodoList, Todo and User entities.

The component and also the pattern used to manage Domain Change Request is Aggregate which collect a group of entities in a Tree Relational Structure with a root representing the Consistency Constraint Boundaries and branches or leaves representing functional depending entities. A **TodoListAggregate** is needed to handle TodoListCreateCommands, TodoListDeleteCommand, TodoListAddTodoCommand, TodoListRemoveTodoCommand, TodoListCheck-TodoCommand and produce the related events to communicate to the entire system that the state is changed.

TodoList Aggregate (Pattern View)

TodoList Aggregate Handlers

Projector

Data changes must be materialized in some way, to create a database representing a practical and queryable system state we need to use a projector that handles Domain Events and writes changes inside a Repository. A **TodoListProjector** is needed to handle TodoListCreatedEvents, TodoListDeleteEvents, TodoListTodoAddedEvent, TodoListTosoRemovedEvent, TodoListTodoCheckedEvent and materialize the changes.

TodoList Projector Handlers (igonre the ErpUserActivityRegisteredEvent will be added in the next tutorial)

Projection

Then we need to access the system state and make queries to receive views, so we require a **TodoListProjection** to handle TodoListListItemViewFindAllQueries returning a collection of TodoListListItemView and TodoListViewFindByIdentifierQueries returning a TodoListView.

TodoList Projection Handlers

Invoker

In the end, we need a way to forge Commands and Queries accessible from the outside of a RECQ Architecture. To do this we need an invoker that exposes all the functions and implements logic (the Service Layer of the Layered Architecture). So we are gonna to define a **TodoListInvoker** generating payloads for every single TodoList-related command and query.

TodoList Invoker handlers

Service, Sagaand Observer

In this tutorial we do not need to handle cross-domain logic or do particular behaviour extensions, we will dedicate a specific tutorial to handle complex scenarios in RECQ Architectures. extend-todolist-handle-complexity-tutorial.md

Final Architecture

TodoList RECQ Architecture

Set up your Development Environment

Create a Spring Boot Application using Spring Initializr and assign Spring Web, Lombok, Spring data JPA, H2 Database.

```
implementation 'org.springframework.boot:spring-boot-starter-data-jpa'
implementation 'org.springframework.boot:spring-boot-starter-web'
compileOnly 'org.projectlombok:lombok'
runtimeOnly 'com.h2database:h2'
annotationProcessor 'org.projectlombok:lombok'
testImplementation 'org.springframework.boot:spring-boot-starter-test'
annotationProcessor 'org.springframework.boot:spring-boot-configuration-processor'
```

Then add your Evento Framework Bundle Dependency and follow the Quick Start section to set up an Evento Server Instance.

```
implementation group: 'com.eventoframework', name: 'evento-bundle', version: 'ev1.8.0'
```

Evento Config

Instantiate the Evento Bundle Object as a Bean

```
import com.eventoframework.demo.todo.TodoApplication;
import com.evento.application.EventoBundle;
import com.evento.application.bus.ClusterNodeAddress;
import com.evento.application.bus.EventoServerMessageBusConfiguration;
import com.evento.application.performance.TracingAgent;
import com.evento.common.modeling.messaging.message.application.Message;
import com.evento.common.modeling.messaging.message.application.Metadata;
import org.springframework.beans.factory.BeanFactory;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration
public class EventoConfig {

    @Bean
    public EventoBundle eventoBundle(BeanFactory factory) throws Exception {
        String bundleId = "ToDoList-Bundle";
        int bundleVersion = 1;
        var evento =  EventoBundle.Builder.builder()
                // Starting Package to detect RECQ components
                .setBasePackage(TodoApplication.class.getPackage())
                // Name of the bundle
```

```
                .setBundleId(bundleId)
                // Bundle's version
                .setBundleVersion(bundleVersion)
                // Set up the Evento message bus
                .setEventoServerMessageBusConfiguration(new EventoServerMessageBusConfigurat
                        // Evento Server Addresses
                        new ClusterNodeAddress("localhost",3030)
                ))
                .setTracingAgent(new TracingAgent(bundleId, bundleVersion){
                    @Override
                    public Metadata correlate(Metadata metadata, Message<?> handledMessage)
                        if(handledMessage!=null && handledMessage.getMetadata() != null && h
                            if(metadata == null) return handledMessage.getMetadata();
                            metadata.put("user", handledMessage.getMetadata().get("user"));
                            return metadata;
                        }
                        return super.correlate(metadata, handledMessage);
                    }})
                .setInjector(factory::getBean)
                .start();
        evento.getPerformanceService().setPerformanceRate(1);
        return evento;
    }
}
```

RECQ Payload Evento Implementation

Once we've defined all our payloads or "message types" we need to implement them using Evento Framework creating a Class for each Payload and adding required fields to handle those messages properly.

- Domain Command
- Domain Events
- Queries
- Views

{% hint style="info" %} As a best practice, we suggest storing all Payloads inside a common library inside a package called "API" like: `com.eventoframework.demo.todo.api`

Then device payloads by domain and then by type following this sample scaffolding:

```
user
    command
    event
    query
```

```
      view
      enum
  todo
      command
      event
      query
      view
      enum
```

{% endhint %}

Domain Commands

First, let's implement all the Domain Commands: domain commands are all commands related to an Aggregate and generating events after the approval.

In Evento Framework every Domain Command implement the class `com.evento.common.modeling.messaging.payload.DomainCommand` and needs the `getAggregateId()` method implementation. Thath method returns the **Unique Aggregate Identifier** used to compute the Aggregate State of the Event Sourcing Pattern. Then you have to specify every single required information as a field.

{% hint style="danger" %} Every single AggregateId in the System must be different you cannot use the same ID in different aggregate Types. {% endhint %}

{% hint style="info" %} Usually, the Resource Identifier (in this case the TodoList Id) is used as an aggregate identifier, and, during the generation, a prefix to identify the aggregate type. {% endhint %}

```
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainCommand;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListCreateCommand implements DomainCommand {

    // The TodoList identifier
    private String identifier;
```

```
    // The TodoList Name
    private String name;

    @Override
    public String getAggregateId() {
        return identifier;
    }
}
```

```
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainCommand;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListDeleteCommand  implements DomainCommand {

    // Identifier of the TodoList to delete
    private String identifier;

    @Override
    public String getAggregateId() {
        return identifier;
    }
}
```

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainCommand;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListAddTodoCommand implements DomainCommand {

    // Identifier of the TodoList to update
```

```
    private String identifier;
    // Identifier of the To-do to delete
    private String todoIdentifier;
    // The To-do content
    private String content;
    @Override
    public String getAggregateId() {
        return identifier;
    }
}
```

```
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainCommand;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListRemoveTodoCommand implements DomainCommand {

    // Identifier of the TodoList to update
    private String identifier;
    // Identifier of the To-do to remove
    private String todoIdentifier;
    @Override
    public String getAggregateId() {
        return identifier;
    }
}
```

```
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainCommand;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
```

```java
public class TodoListCheckTodoCommand implements DomainCommand {

    // Identifier of the TodoList to update
    private String identifier;
    // Identifier of the To-do to check
    private String todoIdentifier;
    @Override
    public String getAggregateId() {
        return identifier;
    }
}
```

Domain Events

For each Domain Command, we need to create a Domain Event representing the System Change State.

Domain Events in Evetno Framework are implemented by extending the abstract class `com.evento.common.modeling.messaging.payload.DomainEvent`. This class has no required method to implement but extends the generic Event class that includes a property called *Context* which we will discuss later.

{% hint style="info" %} Use Past verbs to indicate Events and Present for Commands.

Events can contain more information than the relative command for optimization purposes. (See TodoListTodoCheckedEvent) {% endhint %}

Usually, each vent has a very similar name to the relative command but with an ending Event and a Part verbal time.

```java
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainEvent;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListCreatedEvent extends DomainEvent {

    private String identifier;
    private String content;
```

```
}
```

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainEvent;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListDeletedEvent extends DomainEvent {

    private String identifier;
}
```

```
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainCommand;
import com.evento.common.modeling.messaging.payload.DomainEvent;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListTodoAddedEvent extends DomainEvent {

    private String identifier;
    private String todoIdentifier;
    private String content;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainEvent;

@Domain(name = "TodoList")
```

```java
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListTodoCheckedEvent extends DomainEvent {

    private String identifier;
    private String todoIdentifier;
    // Communicate if all Todos inside this TodoList are checked with this check
    private boolean allChecked;
}
```

```java
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.DomainCommand;
import com.evento.common.modeling.messaging.payload.DomainEvent;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListTodoRemovedEvent extends DomainEvent {

    private String identifier;
    private String todoIdentifier;
}
```

Views

Before asking data from the System we need to know the resulting structure, that's why we need to implement Objects representing data in a formal way.

It constantly happens that the same Domain or Aggregate could be represented in multiple ways based on the request purpose, such as in our case, we have two requirements:

- The list of all TodoList
- The single TodoList

In the first case, we only need generic information to explore the situation and maybe peek at a particular list, in the second case, probably we need a more specific representation.

This separation helps us to optimize requests, traffic and workloads.

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.View;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListListItemView implements View {
    private String identifier;
    private String name;
}
```

```
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.View;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.ZonedDateTime;
import java.util.ArrayList;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListView implements View {
    private String identifier;
    private String name;
    private ArrayList<TodoView> todos;
    private String createdBy;
    private String updatedBy;
    private ZonedDateTime createdAt;
    private ZonedDateTime updatedAt;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
```

```
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.View;

import java.time.ZonedDateTime;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoView implements View {
    private String identifier;
    private String content;
    private boolean completed;
    private String createdBy;
    private String completedBy;
    private ZonedDateTime createdAt;
    private ZonedDateTime completedAt;
}
```

Queries

All actions that are a Data Request by the System are mapped as Query Messages.

A Query in Evento Framework implements the `com.evento.common.modeling.messaging.payload.Query` class, that requires a Parameter indicating the return type: a Single or a Multiple of View extending classes.

{% hint style="info" %} I suggest implementing Queries and Views at the same time in order to properly map data requests and structure. {% endhint %}

In our requirements, we got two specifications: the list all and the find one.

```
import com.eventoframework.demo.todo.api.todo.view.TodoListListItemView;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.Query;
import com.evento.common.modeling.messaging.query.Multiple;

@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
```

```
@Getter
@Setter
public class TodoListListItemViewSearchQuery
        implements Query<Multiple<TodoListListItemView>> {
    // A like filter for the TodoList name
    private String nameLike;
    // Pagination infos
    private int page;
    private int size;
}
```

In the above case, we have used the `com.evento.common.modeling.messaging.query.Multiple` type for return because we are going to return a Collection of `TodoListListItemView.`

```
import com.eventoframework.demo.todo.api.todo.view.TodoListView;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.documentation.Domain;
import com.evento.common.modeling.messaging.payload.Query;
import com.evento.common.modeling.messaging.query.Single;
@Domain(name = "TodoList")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListViewFindByIdentifierQuery
        implements Query<Single<TodoListView>> {
    private String identifier;
}
```

For the find by Id case, we are gonna return only one TodoListView object so we need to use the `com.evento.common.modeling.messaging.query.Single` class as Query return type.

RECQ Components Evento Implementation with Spring Data

Now that we have implemented our payloads we can proceed with implementing components (Aggregates, Projector, Projections, Invokers) handling messages carrying those payloads.

{% hint style="info" %} We suggest developing Aggregates and Services first, then Projector and Projections at the end Sagas or Observers. Invokers are the last. {% endhint %}

We will proceed with this order:

- TodoListAggrgeate
- TodoList Domain in Spring Data
- TodoListProjector
- TodoListProjection
- TodoListInvoker
- Todo List COntroller in Spring Web

TosoListAggregate

Aggregate State

To implement properly an Aggregate with Evento Framework we need first to define the Aggregate State. To do this we need to create a specific class for each Aggregate extending the `com.evento.common.modeling.state.AggregateState` class.

```
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;
import com.evento.common.modeling.state.AggregateState;

import java.util.HashMap;

@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoListAggregateState extends AggregateState {
    private HashMap<String, Boolean> todos;
}
```

As state, we are gonna to use a Map Structure representing the todos contained in the TodoList with the boolean indication for checked/unlocked.

Aggregate

In evento Framework, an Aggregate is implemented with a simple class decorated with the annotation @Aggregate (com.evento.common.modeling.annotations.component.Aggregate).

The Aggregate class will contain two classes of methods:

- `handle(c: Command): Event` - The AggregateCommandHandlers handle Domain Commands and return a System State Changed Event (a Domain Event) after a validation of the command given the aggregate state.

- `on(e: Event): AggregateState` - The EventSourcingHandlers are used to compute the aggregate state given an event stream.

In order to properly implement these handlers you need to use the proper annotation:

- `com.evento.common.modeling.annotations.handler.AggregateCommandHandler`
- `com.evento.common.modeling.annotations.handler.EventSourcingHandler`

Each aggregate instance represents a unique object with its lifecycle (born, live, dead).

To create a new Aggregate for a Domain we need to define a command that gives birth to that specific instance, to indicate with the command handler is used to generate a new instance there is the `init` parameter that must be sat as `true`;

{% hint style="info" %} We suggest implementing the Command Handler and the Event Handler related to a particular action together and one after the other as in the example below to improve readability. {% endhint %}

To indicate the Aggregate dead you need to use the EventSourcingHandler and mark the state as deleted with the `setDeleted(b: Boolean)` method.

```
import com.evento.common.modeling.annotations.component.Aggregate;
import com.evento.common.modeling.annotations.handler.AggregateCommandHandler;
import com.evento.common.modeling.annotations.handler.EventSourcingHandler;
import com.eventoframework.demo.todo.api.todo.command.*;
import com.eventoframework.demo.todo.api.todo.event.*;
import org.springframework.util.Assert;

import java.util.HashMap;

@Aggregate
public class TodoListAggregate {

    @AggregateCommandHandler(init = true)
    public TodoListCreatedEvent handle(TodoListCreateCommand command){
        // Validation
        Assert.isTrue(command.getAggregateId() != null,
                "Error: Todo Id is null");
        Assert.isTrue(command.getName() != null && !command.getName().isBlank(),
                "Error: Content is empty");
        // Command is valid
        return new TodoListCreatedEvent(command.getIdentifier(), command.getName());
    }

    @EventSourcingHandler
    public TodoListAggregateState on(TodoListCreatedEvent event){
```

```
    var state = new TodoListAggregateState();
    state.setTodos(new HashMap<>());
    return state;
}

@AggregateCommandHandler
public TodoListDeletedEvent handle(TodoListDeleteCommand command, TodoListAggregateState
    // Validation
    Assert.isTrue(state.getTodos().values().stream().noneMatch(a -> a),
            "Error: List contains a checked todo");

    // Command is valid
    return new TodoListDeletedEvent(command.getIdentifier());
}

@EventSourcingHandler
public void on(TodoListDeletedEvent event, TodoListAggregateState state){
    state.setDeleted(true);
}

@AggregateCommandHandler
public TodoListTodoAddedEvent handle(TodoListAddTodoCommand command, TodoListAggregateSt
    // Command Validation
    Assert.isTrue(command.getTodoIdentifier() != null && !command.getTodoIdentifier().is
            "Error: Invalid todo identifier");
    Assert.isTrue(command.getContent() != null && !command.getContent().isBlank(),
            "Error: Invalid todo content");
    // State Validation
    Assert.isTrue(!state.getTodos().containsKey(command.getTodoIdentifier()),
            "Error: Todo already present");
    Assert.isTrue(state.getTodos().size() < 5,
            "Error: Todo list is full");
    // Command is valid
    return new TodoListTodoAddedEvent(
            command.getIdentifier(),
            command.getTodoIdentifier(),
            command.getContent());
}

@EventSourcingHandler
public void on(TodoListTodoAddedEvent event, TodoListAggregateState state){
    state.getTodos().put(event.getTodoIdentifier(), false);
}

@AggregateCommandHandler
public TodoListTodoRemovedEvent handle(TodoListRemoveTodoCommand command, TodoListAggreg
```

```
        // Validation
        Assert.isTrue(state.getTodos().containsKey(command.getTodoIdentifier()),
                "Error: Todo not present");
        Assert.isTrue(!state.getTodos().get(command.getTodoIdentifier()),
                "Error: Todo already checked");
        // Command is valid
        return new TodoListTodoRemovedEvent(
                command.getIdentifier(),
                command.getTodoIdentifier());
    }

    @EventSourcingHandler
    public void on(TodoListTodoRemovedEvent event, TodoListAggregateState state){
        state.getTodos().remove(event.getTodoIdentifier());
    }

    @AggregateCommandHandler
    public TodoListTodoCheckedEvent handle(TodoListCheckTodoCommand command, TodoListAggrega
        // Validation
        Assert.isTrue(state.getTodos().containsKey(command.getTodoIdentifier()),
                "Error: Todo not present");
        Assert.isTrue(!state.getTodos().get(command.getTodoIdentifier()),
                "Error: Todo already checked");
        // Command is valid
        return new TodoListTodoCheckedEvent(
                command.getIdentifier(),
                command.getTodoIdentifier(),
                state.getTodos().values().stream().allMatch(b -> b));
    }

    @EventSourcingHandler
    public void on(TodoListTodoCheckedEvent event, TodoListAggregateState state){
        state.getTodos().put(event.getTodoIdentifier(), true);
    }
}
```

TodoList Model with Spring Data

Usually, we start building the Domain Fisical Representation from The Query Point of View. One particular characteristic of RECQ Architecture is the application of the Database-per-query pattern where you design entire databases only to fulfil a Query most efficiently. In order to do that we need to explore all the Queries to define indexes and database paradigms that handle better the request and analyse Views to define the required fields.

Model

In this tutorial, we have chosen to use Spring Data JPA and a relational database to store objects, but we are gonna to implement the Document Paradigm to represent OneToMany Relations.

So, we can start defining the Todo object and inside of it the mapper method toView() that returns the Entity as RECQ View Payload:

```
import com.eventoframework.demo.todo.api.todo.view.TodoView;
import jakarta.persistence.Embeddable;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.ZonedDateTime;

@Embeddable
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class Todo {
    private String identifier;
    private String content;
    private String createdBy;
    private String completedBy;
    private ZonedDateTime createdAt;
    private ZonedDateTime completedAt;

    public TodoView toView() {
        return new TodoView(
                identifier,
                content,
                completedAt != null,
                createdBy,
                completedBy,
                createdAt,
                completedAt
        );
    }
}
```

Then the parent object, the TodoList with two different mappers, one for the List Representation and one for the detailed.

```
import com.eventoframework.demo.todo.api.todo.view.TodoListListItemView;
```

```java
import com.eventoframework.demo.todo.api.todo.view.TodoListView;
import jakarta.persistence.ElementCollection;
import jakarta.persistence.Entity;
import jakarta.persistence.FetchType;
import jakarta.persistence.Id;
import lombok.AllArgsConstructor;
import lombok.Getter;
import lombok.NoArgsConstructor;
import lombok.Setter;

import java.time.ZonedDateTime;
import java.util.ArrayList;
import java.util.List;

@Entity
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
public class TodoList {
    @Id
    private String identifier;
    private String name;
    private String createdBy;
    private String updatedBy;
    private ZonedDateTime createdAt;
    private ZonedDateTime updatedAt;
    @ElementCollection(fetch=FetchType.EAGER)
    private List<Todo> todos;

    public TodoListView toView() {
        return new TodoListView(getIdentifier(),
                getName(),
                new ArrayList<>(getTodos().stream().map(Todo::toView).toList()),
                getCreatedBy(),
                getUpdatedBy(),
                getCreatedAt(),
                getUpdatedAt());
    }

    public TodoListListItemView toListItemView() {
        return new TodoListListItemView(getIdentifier(), getName());
    }

}
```

Repository

Once the model is properly implemented, we can define the Repository to access database Data and also adding specific methods to answer the proper Queries.

```
import org.springframework.data.domain.Page;
import org.springframework.data.domain.PageRequest;
import org.springframework.data.domain.Pageable;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;

public interface TodoListRepository extends JpaRepository<TodoList, String> {
    @Query("select t from TodoList t where t.name like ?1")
    Page<TodoList> search(String query,
                          Pageable pageable);
}
```

TodoListProjector

After the proper definition of the Event Generation in our System and the Model/Repository Layers, we can start to build our Projector, also known as, the domain materializer. The main goal of this component is to store in a database a particular View of the System State.

In Evento Framework a Projector is a simple class annotated with `@Projector (com.evento.common.modeling.annotations.component.Projector)`. This annotation requires a parameter called version that indicate the materialization version in order to recreate the database or change the structure. A complete description of a projector can be found in the Projector Chapter.

```
import com.evento.common.modeling.messaging.message.application.Metadata;
import com.eventoframework.demo.todo.api.todo.event.*;
import com.eventoframework.demo.todo.query.model.Todo;
import com.eventoframework.demo.todo.query.model.TodoList;
import com.eventoframework.demo.todo.query.model.TodoListRepository;
import com.evento.common.modeling.annotations.component.Projector;
import com.evento.common.modeling.annotations.handler.EventHandler;
import com.evento.common.modeling.messaging.message.application.EventMessage;

import java.time.Instant;
import java.time.ZoneId;
import java.util.ArrayList;

@Projector(version = 1)
public class TodoListProjector{

    private final TodoListRepository repository;
```

```
public TodoListProjector(TodoListRepository repository) {
    this.repository = repository;
}

@EventHandler
public void on(TodoListCreatedEvent event, Metadata metadata, Instant timestamp) {
    repository.save(new TodoList(
            event.getIdentifier(),
            event.getContent(),
            metadata.get("user"),
            null,
            timestamp.atZone(ZoneId.systemDefault()),
            null,
            new ArrayList<>(),
            null
    ));
}

@EventHandler
public void on(TodoListDeletedEvent event, EventMessage<TodoListCreatedEvent> message) {
    repository.delete(repository.findById(event.getIdentifier()).orElseThrow());
}

@EventHandler
public void on(TodoListTodoAddedEvent event, Metadata metadata, Instant timestamp) {
    var list = repository.findById(event.getIdentifier()).orElseThrow();
    var td = new Todo(
            event.getTodoIdentifier(),
            event.getContent(),
            metadata.get("user"),
            null,
            timestamp.atZone(ZoneId.systemDefault()),
            null
    );
    list.getTodos().add(td);
    list.setUpdatedAt(td.getCreatedAt());
    list.setUpdatedBy(td.getCreatedBy());
    repository.save(list);
}

@EventHandler
public void on(TodoListTodoRemovedEvent event, Metadata metadata, Instant timestamp) {
    var list = repository.findById(event.getIdentifier()).orElseThrow();
    list.getTodos().removeIf(t -> event.getTodoIdentifier().equals(t.getIdentifier()));
```

```
        list.setUpdatedAt(timestamp.atZone(ZoneId.systemDefault()));
        list.setUpdatedBy(metadata.get("user"));
        repository.save(list);
    }

    @EventHandler
    public void on(TodoListTodoCheckedEvent event, Metadata metadata, Instant timestamp) {
        var list = repository.findById(event.getIdentifier()).orElseThrow();
        var td = list.getTodos().stream().filter(t -> event.getTodoIdentifier().equals(t.get
        td.setCompletedAt(timestamp.atZone(ZoneId.systemDefault()));
        td.setCompletedBy(metadata.get("user"));
        list.setUpdatedAt(td.getCompletedAt());
        list.setUpdatedBy(td.getCompletedBy());
        repository.save(list);
    }
}
```

TodoListProjection

To handle Query Messages we need to implement a Projection.

In EVento Framework a projection is a standard class annotated with @Projection (com.evento.common.modeling.annotations.component.Projection). It contains Query Handlers, methods annotated with `@QueryHandler (com.evento.common.modeling.annotations.handler.QueryHandler)` with the Query as a parameter and returning a Single or a Multiple of a particular View.

```
import com.eventoframework.demo.todo.api.todo.query.TodoListListItemViewSearchQuery;
import com.eventoframework.demo.todo.api.todo.query.TodoListViewFindByIdentifierQuery;
import com.eventoframework.demo.todo.api.todo.view.TodoListListItemView;
import com.eventoframework.demo.todo.api.todo.view.TodoListView;
import com.eventoframework.demo.todo.query.model.TodoList;
import com.eventoframework.demo.todo.query.model.TodoListRepository;
import com.evento.common.modeling.annotations.component.Projection;
import com.evento.common.modeling.annotations.handler.QueryHandler;
import com.evento.common.modeling.messaging.query.Multiple;
import com.evento.common.modeling.messaging.query.Single;
import org.springframework.data.domain.PageRequest;

@Projection()
public class TodoListProjection {

    private final TodoListRepository repository;

    public TodoListProjection(TodoListRepository repository) {
```

```
        this.repository = repository;
    }

    @QueryHandler
    public Single<TodoListView> handle(TodoListViewFindByIdentifierQuery query) {
        return Single.of(repository.findById(query.getIdentifier()).map(TodoList::toView).or
    }

    @QueryHandler
    public Multiple<TodoListListItemView> handle(TodoListListItemViewSearchQuery query) {
        return Multiple.of(repository.search(
                        "%" + query.getNameLike() + "%",
                        PageRequest.of(query.getPage(),
                                query.getSize()))
                .map(TodoList::toListItemView).toList());
    }
}
```

TodoList Invoker

In the end, we need to implement the Invoker, the bridge component between the standard application and the RECQ architecture.

An Invoker is a class annotated with `@Invoker` (`com.evento.common.modeling.annotations.component.Invo` and extending the `com.evento.application.proxy.InvokerWrapper` class. An invoker contains methods annotated with `@InvocationHandler` (`com.evento.common.modeling.annotations.handler.InvocationHandler`) used to implement business logic using the Command Gateway and the Query Gateway that you can access with `getCommandGateway()` and `getQueryGateway()` methods.

```
import com.eventoframework.demo.todo.api.todo.command.*;
import com.eventoframework.demo.todo.api.todo.query.TodoListListItemViewSearchQuery;
import com.eventoframework.demo.todo.api.todo.query.TodoListViewFindByIdentifierQuery;
import com.eventoframework.demo.todo.api.todo.view.TodoListListItemView;
import com.eventoframework.demo.todo.api.todo.view.TodoListView;
import com.evento.application.proxy.InvokerWrapper;
import com.evento.common.modeling.annotations.component.Invoker;
import com.evento.common.modeling.annotations.handler.InvocationHandler;
import com.evento.common.modeling.messaging.message.application.Metadata;
import com.evento.common.modeling.messaging.query.Multiple;
import com.evento.common.modeling.messaging.query.Single;

import java.util.Collection;
import java.util.UUID;
import java.util.concurrent.CompletableFuture;
```

```java
@Invoker
public class TodoListInvoker extends InvokerWrapper {

    @InvocationHandler
    public String createTodoList(String name, String user){
        var identifier = "TDLS_" + UUID.randomUUID();
        getCommandGateway().sendAndWait(new TodoListCreateCommand(identifier, name), toUserM
        return identifier;
    }

    @InvocationHandler
    public void deleteTodoList(String identifier, String user){
        getCommandGateway().sendAndWait(new TodoListDeleteCommand(identifier), toUserMetadat
    }

    @InvocationHandler
    public String addTodo(String identifier, String content, String user){
        var todoIdentifier = "TODO_" + UUID.randomUUID();
        getCommandGateway().sendAndWait(new TodoListAddTodoCommand(identifier, todoIdentifie
        return todoIdentifier;
    }

    @InvocationHandler
    public void checkTodo(String identifier, String todoIdentifier, String user){
        getCommandGateway().sendAndWait(new TodoListCheckTodoCommand(identifier, todoIdentif
    }

    @InvocationHandler
    public void removeTodo(String identifier, String todoIdentifier, String user){
        getCommandGateway().sendAndWait(new TodoListRemoveTodoCommand(identifier, todoIdenti
    }

    @InvocationHandler
    public CompletableFuture<TodoListView> findTodoListByIdentifier(String identifier){
        return getQueryGateway().query(new TodoListViewFindByIdentifierQuery(identifier)).th
    }

    @InvocationHandler
    public CompletableFuture<Collection<TodoListListItemView>> searchTodoList(String nameLik
        return getQueryGateway().query(new TodoListListItemViewSearchQuery(nameLike, page, 1
                .thenApply(Multiple::getData);
    }

    private Metadata toUserMetadata(String user) {
        var m = new Metadata();
```

```
        m.put("user", user);
        return m;
    }
}
```

{% hint style="info" %} An invoker implements the Service Layer of the Layered Architecture, also, this separation gives the correct level of abstraction when you need to interact with other java frameworks or libraries. {% endhint %}

Expose the RECQ architecture with Spring Web

The last level of the layered architecture after the service, implemented by a RECQ Invoker, is the Controller.

The main goal of a controller is let user interact with the software core logic using a particular protocol or interface.

In this tutorial, we are going to expose our System with a REST Interface implemented by a Spring Controller.

```
import com.eventoframework.demo.todo.api.todo.view.TodoListListItemView;
import com.eventoframework.demo.todo.api.todo.view.TodoListView;
import com.eventoframework.demo.todo.service.invoker.TodoListInvoker;
import com.evento.application.EventoBundle;
import com.eventoframework.demo.todo.web.dto.CreatedResponse;
import com.eventoframework.demo.todo.web.dto.TodoCreateRequest;
import com.eventoframework.demo.todo.web.dto.TodoListCreateRequest;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import java.util.Collection;
import java.util.concurrent.CompletableFuture;

@RestController
@RequestMapping("/todo-list")
public class TodoListController {

    private final TodoListInvoker todoListInvoker;

    public TodoListController(EventoBundle eventoBundle) {
        // Instantiate the invoker
        todoListInvoker = eventoBundle.getInvoker(TodoListInvoker.class);
    }

    @GetMapping("/")
    public CompletableFuture<ResponseEntity<Collection<TodoListListItemView>>> searchTodoLis
```

```
        @RequestParam(defaultValue = "") String nameLike, @RequestParam(defaultValue = "0
) {
    return todoListInvoker
            .searchTodoList(nameLike, page)
            .thenApply(ResponseEntity::ok);
}

@GetMapping("/{identifier}")
public CompletableFuture<ResponseEntity<TodoListView>> findTodoListByIdentifier(
        @PathVariable String identifier) {
    return todoListInvoker
            .findTodoListByIdentifier(identifier)
            .thenApply(ResponseEntity::ok);
}

@PostMapping("/")
public ResponseEntity<CreatedResponse> createTodoList(
        @RequestBody TodoListCreateRequest request, @RequestHeader(name = "Authorizatio
    return ResponseEntity
            .status(HttpStatus.CREATED)
            .body(new CreatedResponse(
                    todoListInvoker.createTodoList(request.getName(), user)
            ));
}


@DeleteMapping("/{identifier}")
public ResponseEntity<Void> deleteTodoList(
        @PathVariable String identifier,
        @RequestHeader(name = "Authorization") String user) {
    todoListInvoker.deleteTodoList(identifier, user);
    return ResponseEntity.noContent().build();
}

@PostMapping("/{identifier}/todo/")
public ResponseEntity<CreatedResponse> addTodo(
        @PathVariable String identifier,
        @RequestBody TodoCreateRequest request,
        @RequestHeader(name = "Authorization") String user) {
    return ResponseEntity.status(HttpStatus.CREATED)
            .body(new CreatedResponse(
                    todoListInvoker.addTodo(identifier, request.getContent(), user)
            ));
}

@DeleteMapping("/{identifier}/todo/{todoIdentifier}")
```

```
    public ResponseEntity<Void> removeTodo(
            @PathVariable String identifier,
            @PathVariable String todoIdentifier,
            @RequestHeader(name = "Authorization") String user) {
        todoListInvoker.removeTodo(identifier, todoIdentifier, user);
        return ResponseEntity.noContent().build();
    }

    @PutMapping("/{identifier}/todo/{todoIdentifier}")
    public ResponseEntity<Void> checkTodo(
            @PathVariable String identifier,
            @PathVariable String todoIdentifier,
            @RequestHeader(name = "Authorization") String user) {
        todoListInvoker.checkTodo(identifier, todoIdentifier, user);
        return ResponseEntity.accepted().build();
    }
}
```

DTOs

```
import lombok.Data;

@Data
public class TodoListCreateRequest {
    private String name;
}
```

```
import lombok.Data;

@Data
public class TodoCreateRequest {
    private String content;
}
```

```
import lombok.AllArgsConstructor;
import lombok.Data;

@Data
@AllArgsConstructor
public class CreatedResponse {
    private String identifier;
}
```

Test Your App

```
POST http://localhost:8080/todo-list/
```

```
Authorization: user1
Content-Type: application/json

{
  "name": "Sample Todo List"
}

> {%
    client.test("Request executed successfully (Created - 201)", function() {
        client.assert(response.status === 201, "Response status is not 201");
    });
 client.global.set("lastId", response.body.identifier)
 %}

###

POST http://localhost:8080/todo-list/{{lastId}}/todo/
Authorization: user1
Content-Type: application/json

{
  "content": "Simple Todo"
}

> {%
    client.test("Request executed successfully (No Content - 201)", function() {
        client.assert(response.status === 201, "Response status is not 201");
    });
    client.global.set("lastTodoId", response.body.identifier)
%}

###

GET http://localhost:8080/todo-list/{{lastId}}
Authorization: user1


> {%
    client.test("Request executed successfully (Ok - 200)", function() {
        client.assert(response.status === 200, "Response status is not 200");
    });
    client.test("Response correct", function() {

        client.assert(response.body.todos[0].identifier === client.global.get("lastTodoId"),
    });
    console.log(response)
```

```
%}


###

DELETE http://localhost:8080/todo-list/{{lastId}}/todo/{{lastTodoId}}
Authorization: user1


> {%
    client.test("Request executed successfully (Ok - 204)", function() {
        client.assert(response.status === 204, "Response status is not 204");
    });
%}

###

GET http://localhost:8080/todo-list/{{lastId}}
Authorization: user1


> {%
    client.test("Request executed successfully (Ok - 200)", function() {
        client.assert(response.status === 200, "Response status is not 200");
    });
    client.test("Response correct", function() {

        client.assert(response.body.todos.length === 0, "TodoList not updates");
    });
    console.log(response)
%}

###

POST http://localhost:8080/todo-list/{{lastId}}/todo/
Authorization: user1
Content-Type: application/json

{
  "content": "Simple Todo"
}

> {%
    client.test("Request executed successfully (No Content - 201)", function() {
        client.assert(response.status === 201, "Response status is not 201");
    });
```

```
    client.global.set("lastTodoId", response.body.identifier)
%}

###

PUT http://localhost:8080/todo-list/{{lastId}}/todo/{{lastTodoId}}
Authorization: user1

> {%
    client.test("Request executed successfully (No Content - 202)", function() {
        client.assert(response.status === 202, "Response status is not 202");
    });
%}


###

DELETE http://localhost:8080/todo-list/{{lastId}}/todo/{{lastTodoId}}
Authorization: user1

> {%
    client.test("Request executed successfully (No Content - 500)", function() {
        client.assert(response.status === 500, "Response status is not 500");
    });
%}

###

DELETE http://localhost:8080/todo-list/{{lastId}}
Authorization: user1

> {%
    client.test("Request executed successfully (No Content - 500)", function() {
        client.assert(response.status === 500, "Response status is not 500");
    });
%}
```

Functional Analysis - Event Storming Approach (Bonus Chapter)

Extend TodoList - Handle Complexity Tutorial

Introduction

RECQ (Reactive, Event-Driven Commands and Queries) is a set of principles and patterns architectural, methodological, behavioural, and structural aimed at the creation of systems software with event-oriented microservices architectures compliant with the Reactive Manifesto and the more recent Reactive Principles.

Another objective of the proposed methodology is to comply with the following rules, guidelines guides and patterns:

- Avoid Big Ball of Mud(Foote & Yoder, 1997)
- Command-Query Separation (Fowler, CommandQuerySeparation, 2005)
- Separation of Concerns (Dijkstra, 1982)
- Single Source of Truth (Pang & Szafron, 2014)
- SOLID Principles (Martin, PrinciplesOfOod, 2005)
- Uniform Access Principle (Meyer, 1997)
- Event Sourcing
- Messaging
- Domain-Driven Design

There are three Patterns:

- RECQ System Pattern– an architectural pattern that defines high-level modules that make up a RECQ system.
- RECQ Communication Pattern – a behavioural pattern that defines how the elements communicate with each other.
- RECQ Component Patten – rigorously defining methodological aspects and individual responsibilities of each component of a RECQ system.

RECQ System Pattern

A RECQ System consists of the following modules:

- **Components** – Self-contained portions of software that implement application logic.
- **Message Gateway** – Component that manages communication between components in terms of requests and responses.
- **System State Store** – Component responsible for persisting system state in the form of event logs.

So a RECQ system is a set of computational units called components that communicate with each other by exchanging messages; if a component causes

the system state to change, this information is published as an event in the System State Store, furthermore, the components can listen for system state changes to change their internal state without an explicit message from another component.

RECQ System Big Picture

So a RECQ system is a set of computational units called components which they communicate with each other by exchanging messages; if a component has a consequence the system state changes, this information is published as an event in the System State Store, and components can listen for changes of the system state to change their internal state without an explicit message from of another component.

Example interactin in a RECQ System

Component

Properties

A component must submit to the following principles:

- Isolation– i.e. it is decoupled from each other, both from a temporal point of view and a spatial one. Therefore it does not have to make assumptions about the presence of other components e not even on their position in the system (Location Transparency).
- Containment – in case a component needs to implement cross-resource logic these logics must be contained in the component same. Often this concept is also described as aggregation above all regarding Domain Driven Design: if a particular concept is strongly 33 coupled with another one creates an aggregate or a more complex domain than them contains both.
- Delegation– unlike the containment property, the concept of delegation forces us to limit the component to a minimal set of responsibilities. All that not is the direct responsibility of the component and must be delegated externally.

Capabilities

A component can implement application logic only by exploiting the capabilities listed below:

- It can receive messages dedicated to him and reply with a message to the sender – this capability allows the component to implement the concept of action since they depend on an external request. receiving a message, therefore, it corresponds to application logic.
- Can have a persistent internal state, also called a local state.
- Can publish System State Change Events.
- Can consume System State Change Events.

- Can send messages to other components.
- t must be replicable – therefore even in the presence of infinite instances of this component active in the system at the same time, no problems should arise.

Message Gateway

Components cannot directly interact with each other with direct invocations. A component only knows possible messages and responses from the entire system. It then sends messages through the Message Gateway and waits for replies.

The message gateway implements the Publish/Async Response pattern via "correlation id". This module has two sub-modules, the Command Gateway and the Query Gateway.

Command Gateway

This module is used to be able to make command-type requests. It exposes a single type of asynchronous method called "send" which takes a message of type Command as input and, in the event of an action that has taken place, returns the event generated by the component that handled the request, otherwise a failure error.

Query Gateway

This module is used to make query-type requests. It exposes only one type of asynchronous method called "query" which takes as input a message of type Query and returns an object of type `QueryResponse` containing the requested data.

Message Gateway Structure

System State Store

The state of the system is represented by the ordered sequence of change-of-state events; this module is to all intents and purposes an **Event Store**.

This module is our system's **Single Source of Truth (SSOT)**. It is the only system component that actually knows the state throughout its history, any other local representation of the system state is a reference.
Components can be notified of state change events from this module.

The SSS must exhibit the following actions:

- **Publishing a domain event** – then adding a new state change event to the event store, the event can be associated with an aggregate.
- **Get an event Stream** – get all ordered events from a starting point, which can also be filtered by aggregate and type of event.

System State Store Structure

RECQ Communication Pattern

We have defined that in a RECQ System all the components cooperates by sending messages, with this pattern we are gonna to define the communication structure based on the expected behavior that each message want to produce inside a RECQ System.

Messages

In a RECQ System we have messages travelling from a component to an other passing by a Message Gateway. In this section we are gonna to diefine the five type of messages that a RECQ System can handle:

- Commands: Messages in the system that want to change the state of the system
- Events: A consequence of an accepted command that is telling us that the System State has changed
- Queries: Messages to extract data from the system
- View: Messages representing data extracted by a Query (this message is a response)
- Void: A symple response message telling that a Command request has been performed withouth errors (this message is a response)

RECQ Payloads

So, the message bus can only handle this kind of messages and each of it with a particular behavior.

Message Handling

In the next chapters we will explain how those messages are handled and wich communication protocol is applied for every of them:

- Component to Component: the request/response communication solution to obtain data and performa changes
- Component to System State Store: How the system state store is updated
- System State Store to Component: the pub/sub communication solution to distribute system changes across componente

Component to Component

Communication from one component to another component is implemented with a protocol of asynchronous requests and responses from the Message Gateway.

However, each component can only send two types of messages:

- Commands – a request to a system change action which, in response, will only have confirmation of whether or not the action has taken place.
- Query – a request that does not change the system, but returns the data (the views or View)

This is because the components must obey the CQRS pattern.

Component to Component Communication

Component to System State Store

Communication from a componentto the System State Store (SSS)can only be a request for posting a state change event, in which case the request will leverage a synchronous protocol managed by the SSS itself.

System State Store to Component

The communication takes place with a pub/sub protocol managed not by theMessage Gateway but from the combination of the System State Store and the Consumer State Stores (for le Sagasand Projections).

System STate Store to Component Communication

Consumer State Stores are modules that aim to make the state persistent progress of event consumption from the SSS, so as to implement retry logics e ensure orderly progress and consistency.

RECQ Component Pattern

This is a set of definitions with distinct and minimal semantics to implement any application, where minimality is not rigorous, but empirical deriving from the fact that most use cases can be designed with only the components described below.

RECQ Components Big Picture

Capability Table

In the following sections, the types will be defined, for each of which will be summarized capacity and scalability properties according to the CAP theoremusing a defined table such as the “Capability Table”, of which there is a generic version in Table 1. The first five entries refer to message handlers they can react to and the invocations they can make. By type of state we mean if a handler to be executed requires, in addition to the input message, also a state of some sort kept consistent by component;

There are two types of status:

- **Instance**: A type of state that is based on a single instance of a domain object or a particular resource. So there can be multiple Handlers at the same time running in this component as long as they are handling resource requests or different objects.
- **Component**: There can be no more than one handler for this particular component active that handles a request.

The last entry refers to the properties of the CAP theorem respected by the component:

- C: component implements strong consistency, it happens in components that have a state that must be kept consistent regardless of the type of message handled.
- c: the component implements weak consistency, so the consistency is only in managed message function.
- A: The component implements strong availability, and the response time is known a priori.
- a: the component implements weak availability, the response time is known a priori provided that the requests do not refer to the same resource.
- P: partitioning tolerance, being a distributed system this constraint must always be present.

Capability	
Can handle Command Messages	Yes/No
Can handle Query Messages	Yes/No
Can handle Events	Yes/No
Can send Command Messages	Yes/No
Can Send Query Messages	Yes/No
State type	Instance/Component/No
CAP Properties	C/c/A/a/P

Aggregate

RECQ Aggregate Big Picture

This component represents a single instance that is part of our domain and has its own personal status. The state of this instance is reconstructed by doing a replay of events contained in the SSS for this particular aggregate starting from a identifier contained within the received message.

An aggregate, therefore, has only one "handle" method (the command handler), which takes in input a particular message of type Command and a State of the aggregate (represented as a sequence of events) and only as a function of these two returns the consequent event of a change of state or an error. The state change event, if returned, comes next published in the SSS.

So the aggregates implement the actual domain logic.

A command handler cannot send Query-type requests, as all information required by him to define whether the command is acceptable or not must be present in its state, moreover, as defined previously, the queries are possibly consistent and therefore not usable to define whether the request is consistent with the rest of the system.

However, it can carry out commands, since the latter is by definition always consistent and some commands may depend on others to work: for example, the generation of a unique identifier throughout the system, can not by definition be generated knowing only a portion of the system and therefore, this action depends on a command to generate a unique id towards another responsible aggregate.

As for scalability, being there is a state that needs to be maintained consistent (C) there is a lock to avoid concurrent access to the same aggregate, therefore not being able to guarantee availability (a) but only partitioning tolerance (P).

However, it must be said that the consistency is not at the level of the entire system but is local (Partitioned State) and consequently also the lack of availability is local; therefore, as a whole the system can be both Consistent and partially available (Basic Availability).

Capability	
Can handle Command Messages	Yes
Can handle Query Messages	No
Can handle Events	No
Can send Command Messages	Yes
Can Send Query Messages	No
State type	Instance (Bu Aggregate Key)
CAP Properties	CaP

Aggregate Structure

Projector

RECQ Projector Big Picture

It presents an “on” method (called event Handler) which takes an Event as input, ie one of the state changes from the SSS, and builds the local domain model of the system without returning anything.

Each projector is consistent and processes only one event at a time in sequence, so it cannot scale. In particular, a projector implements the singleton pattern (Gamma, Helm, Johnson, & Vlissides, 1994) at the whole system level.

Projectors have an internal state which consists of knowing how far they have consumed from the SSS (to ensure they are consistent with the SSOT up to the time stored) in a Shared Consumer State Store using the Shared Database technique (Richardson, Microservices Patterns: With examples in Java, 2018) by type of projector.

The Consumer State Store is a submodule that implements the methods to be able to save the identifier of the last consumed event, return the last consumed event and manage concurrent access to this data.

Being within the Query Model, each Event Handler cannot perform actions aimed at changing the state of the system, but can possibly access the data of other components making requests of the Query type even if the answers may not be consistent with the local model.

Capability	
Can handle Command Messages	No
Can handle Query Messages	No
Can handle Events	Yes
Can send Command Messages	No
Can Send Query Messages	Yes
State type	Component
CAP Properties	CP

Projector Structure

Projection

RECQ Projection Big Picture

It only presents a "handle" method (in this case defined as Query Handler), then it receives Query type messages to which it replies with the data stored in the structures local data held by the Projector and can scale as you like since it has no state and the state of the local model is asserted to be consistent.

Again, the Query Handler, due to the same constraint mentioned above, cannot send requests of type Command, but only Queries to implement patterns such as Federated Query, but even then there is a possibility that data from other projections are not consistent with the local model.

Capability	
Can handle Command Messages	No
Can handle Query Message	Yes
Can handle Events	No
Can send Command Messages	No

Capability	
Can Send Query Messages	Yes
State type	No
CAP Properties	AP

Projection Structure

Service

RECQ Service Big Picture

This component also has a command handler, but unlike the aggregate, the latter takes only the command as input and may not return events, other than the fact that cannot make query-type requests.

An example of a service would be the component responsible for sending the emails mentioned above. Sending an email is something external to the application system you are on developing, moreover, this action must be available and consistent but the fact of being able whether or not to send the mail does not depend on the information contained in the system. Like the invokers it was the components that bridged the gap between the outside and the inside, the services do the opposite. A further example would be the implementation of a payment service delegated to an external provider, requests to the provider must also be consistent but this consistency is not our responsibility.

For scalability aspects, consistency is not taken into account because it is not dependent on the internal system (c) therefore as a whole it is available, except for one of its own ACID implementation, (a) and partitioning tolerant (P).

Capability	
Can handle Command Messages	Yes
Can handle Query Messages	No
Can handle Events	No
Can send Command Messages	Yes
Can Send Query Messages	No
State type	No
CAP Properties	caP

Service Structure

Invoker

RECQ Invoker Big Picture

The invoker is a bridge component between the outside of the system and the inside.

An example of an Invoker is a REST controller that exposes system actions for being able to allow access via the network. It can also be a GUI controller, a CLI application or a trigger from CRON.

Having no internal state, this component can scale at will, therefore, it can turn out always available (A) and is partitioning tolerant (P)

Capability	
Can handle Command Messages	No
Can handle Query Messages	No
Can handle Events	No
Can send Command Messages	Yes
Can Send Query Messages	Yes
State type	No
CAP Properties	AP

Invoker Structure

Saga

RECQ Big Picture

There are several cases in which two aggregates or services must share information in order to be consistent at the application level. This component is like a projector only that the changes are made on the system itself.

It has an "on" method (in this case called Saga Event Handler) which, taking the current state of the saga and an Event as input, returns the new state of the saga. This method internally can send both command-type and query-type messages

The state of a saga is similar to that of an aggregate, the difference is that the persistence is local and shared among all active instances of a certain saga. While for the aggregate the state is held in the SSS, shared among all, sagas of a particular type using a Saga Shared Consumer State Store following the Shared Database pattern: this sub-module is an extension of the Consumer State Store that implements additional methods to save and retrieve the particular instance of the saga managed by the component (the transaction state).

It also has the same scalability constraints as a projector.

Capability	
Can handle Command Messages	No
Can handle Query Messages	No
Can handle Events	Yes
Can send Command Messages	No
Can Send Query Messages	Yes
State type	Component
CAP Properties	CP

Saga Structure

Observer

RECQ Observer Big Picture

While a saga must have an internal state in order to understand how to move a transaction forward, an Observer (Observer) has no historical memory and reacts to a single event.

It has the usual "on" method (Event Handler) which, like the Projector, depends on a single event and returns nothing, however, an Observer can make Command and Query type requests like the sagas.

In reality, an Observer is a special case of a saga that begins and ends with a single event.

Its behaviour is also comparable to a service that is invoked from within via an event.

Not having an internal state it can scale easily, furthermore, unlike the implementation of sagas or projectors, for an observer there is no orderly and consistent consumption constraint of the events.

Capability	
Can handle Command Messages	No
Can handle Query Messages	No
Can handle Events	Yes
Can send Command Messages	No
Can Send Query Messages	Yes
State type	No
CAP Properties	AP

Observer Structure

Introduction

Payload

WIP

Command

WIP

```
package org.evento.common.modeling.messaging.payload;

public abstract class Command extends Payload {
}
```

Domain Command

```
package org.evento.common.modeling.messaging.payload;

public abstract class DomainCommand extends Command {
    public abstract String getAggregateId();
}
```

```
package org.evento.demo.api.command;
import org.evento.common.modeling.messaging.payload.DomainCommand;

public class DemoCreateCommand extends DomainCommand {

    private String demoId;
    private String name;
    private Long value;

    /** Getter, Setter, Constructors **/

    @Override
    public String getAggregateId() {
        return demoId;
    }
}
```

Service Command

```
package org.evento.common.modeling.messaging.payload;

public abstract class ServiceCommand extends Command {
```

```
    public String getLockId(){
        return null;
    }
}
```

```
package org.evento.demo.api.command;

import org.evento.common.modeling.messaging.payload.ServiceCommand;

public class NotificationSendCommand extends ServiceCommand {
    private String body;

    /** Getter, Setter, Constructors **/

}
```

Event

WIP

```
public abstract class Event extends Payload {
}
```

Domain Event

WIP

```
package org.evento.common.modeling.messaging.payload;

public abstract class DomainEvent extends Event {
}
```

```
package org.evento.demo.api.event;
import org.evento.common.modeling.messaging.payload.DomainEvent;

public class DemoCreatedEvent extends DomainEvent {

    private String demoId;
    private String name;
    private Long value;

    /** Getter, Setter, Constructors **/
}
```

ServiceEvent

WIP

```
package org.evento.common.modeling.messaging.payload;

public abstract class ServiceEvent extends Event {
}
```

```
package org.evento.demo.api.event;
import org.evento.common.modeling.messaging.payload.ServiceEvent;

public class NotificationSentEvent extends ServiceEvent {
    private String notificationId;
    private String body;

    /** Getter, Setter, Constructors **/
}
```

View

WIP

```
package org.evento.common.modeling.messaging.payload;

public abstract class View extends Payload {
}
```

```
package org.evento.demo.api.view;
import org.evento.common.modeling.messaging.payload.View;

public class DemoView extends View {
    private String demoId;
    private String name;
    private Long value;

    /** Getter, Setter, Constructors **/
}
```

```
package org.evento.demo.api.view;

import org.evento.common.modeling.messaging.payload.View;

public class DemoRichView extends View {

    private String demoId;
    private String name;
    private Long value;
```

```
    private long createdAt;
    private long updatedAt;
    private Long deletedAt;

    /** Getter, Setter, Constructors **/
}
```

Query

WIP

```
package org.evento.demo.api.query;

import org.evento.common.modeling.messaging.payload.Query;
import org.evento.common.modeling.messaging.query.Multiple;
import org.evento.demo.api.view.DemoRichView;

public class DemoRichViewFindAllQuery extends Query<Multiple<DemoRichView>> {

    private String filter;
    private String sort;
    private Integer limit;
    private Integer offset;

    /** Getter, Setter, Constructors **/
}
```

Query Response

WIP

```
package org.evento.common.modeling.messaging.query;

import org.evento.common.modeling.messaging.payload.View;
import java.io.Serializable;

public abstract class QueryResponse<T extends View> implements Serializable {
}
```

Multiple

WIP

```
package org.evento.common.modeling.messaging.query;
```

```java
import org.evento.common.modeling.messaging.payload.View;

import java.util.Collection;
import java.util.List;

public class Multiple<T extends View> extends QueryResponse<T> {

    private Collection<T> data;

    public static <R extends View> Multiple<R> of(Collection<R> data) {
        var r = new Multiple<R>();
        r.setData(data);
        return r;
    }

    public static <R extends View> Multiple<R> of(R... items) {
        var r = new Multiple<R>();
        r.setData(List.of(items));
        return r;
    }

    /** Getter, Setter, Constructors **/

}
```

Single

WIP

```java
package org.evento.common.modeling.messaging.query;

import org.evento.common.modeling.messaging.payload.View;

public class Single<T extends View> extends QueryResponse<T> {

    private T data;

    public static <R extends View> Single<R> of(R data) {
        var r = new Single<R>();
        r.setData(data);
        return r;
    }

    /** Getter, Setter, Constructors **/
}
```

@Component

WIP

@Aggregate

WIP

```
package org.evento.common.modeling.annotations.component;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Component
public @interface Aggregate {
    int snapshotFrequency() default -1;
}
```

```
import org.evento.common.modeling.annotations.component.Aggregate;

@Aggregate(snapshotFrequency = 10)
public class DemoAggregate {
    /** Command Handlers, Event Sourcing Handlers **/
}
```

Aggregate State

```
package org.evento.common.modeling.state;

import java.io.Serializable;

public abstract class AggregateState implements Serializable {
    private boolean deleted = false;

    /** Getter, Setter, Constructors **/
}
```

```
package org.evento.demo.command.aggregate;

import org.evento.common.modeling.state.AggregateState;

public class DemoAggregateState extends AggregateState {
```

```
    private long value;

    /** Getter, Setter, Constructors **/
}
```

@AggregateCommandHandler

```
package org.evento.common.modeling.annotations.handler;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Handler
public @interface AggregateCommandHandler {
    boolean init() default false;
}
```

@EventSourcingHandler

```
package org.evento.common.modeling.annotations.handler;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
@Handler
public @interface EventSourcingHandler {
}
```

@Projector

WIP

```
package org.evento.common.modeling.annotations.component;

import java.lang.annotation.ElementType;
import java.lang.annotation.Retention;
```

```java
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Target;

@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
@Component
public @interface Projector {
    int version();
}
```

```java
package org.evento.demo.query;

import org.evento.common.modeling.annotations.component.Projector;

@Projector(version = 2)
public class DemoMongoProjector {

    private final DemoMongoRepository demoMongoRepository;

    public DemoMongoProjector(DemoMongoRepository demoMongoRepository) {
        this.demoMongoRepository = demoMongoRepository;
    }

    /** Event Handlers **/
}
```

Projector @EventHandler

```java
package org.evento.demo.query;
import org.evento.common.modeling.annotations.handler.EventHandler;
//...

@EventHandler
void on(DemoCreatedEvent event,
        QueryGateway queryGateway,
        EventMessage eventMessage) {
    var now = Instant.now();
    demoMongoRepository.save(new DemoMongo(event.getDemoId(),
            event.getName(),
            event.getValue(), now, now, null));
}

@EventHandler
void on(DemoUpdatedEvent event) {
    demoMongoRepository.findById(event.getDemoId()).ifPresent(d -> {
```

```
        d.setName(event.getName());
        d.setValue(event.getValue());
        d.setUpdatedAt(Instant.now());
        demoMongoRepository.save(d);
    });
}
```

@Projection

WIP

```
@Projection
public class DemoProjection {

    private final DemoMongoRepository demoMongoRepository;

    public DemoProjection(DemoMongoRepository demoMongoRepository) {
        this.demoMongoRepository = demoMongoRepository;
    }

    /** Query Handlers **/
}
```

@QueryHandler

```
@QueryHandler
Single<DemoView> query(DemoViewFindByIdQuery query, QueryMessage<DemoViewFindByIdQuery> quer
    Utils.logMethodFlow(this, "query", query, "BEGIN");
    var result = demoMongoRepository.findById(query.getDemoId())
            .filter(d -> d.getDeletedAt() != null)
            .map(DemoMongo::toDemoView).orElseThrow();
    Utils.logMethodFlow(this, "query", query, "END");
    return Single.of(result);
}

@QueryHandler
Multiple<DemoView> query(DemoViewFindAllQuery query) {
    Utils.logMethodFlow(this, "query", query, "BEGIN");
    var result = demoMongoRepository.findAll().stream()
            .filter(d -> d.getDeletedAt() != null)
            .map(DemoMongo::toDemoView).toList();
    Utils.logMethodFlow(this, "query", query, "END");
    return Multiple.of(result);
}
```

```
@QueryHandler
Single<DemoRichView> queryRich(DemoRichViewFindByIdQuery query) {
    Utils.logMethodFlow(this, "query", query, "BEGIN");
    var result = demoMongoRepository.findById(query.getDemoId())
            .map(DemoMongo::toDemoRichView).orElseThrow();
    Utils.logMethodFlow(this, "query", query, "END");
    return Single.of(result);
}
```

@Service

WIP

@Invoker

WIP

@Saga

WIP

@Observer

WIP

Bundle

WIP

Consumer State Store

WIP

Autoscaling Protocol

WIP

Introduction

Bundle Deploy Plugin

WIP

Message BUS

WIP

Payload Catalog

WIP

Component Catalog

WIP

Bundle Catalog

WIP

Cluster Status

WIP

Flows

WIP

Performance Evaluation

WIP

Application Graph

WIP

Update Version

WIP

Publish

WIP