# Introduction

The Evento Framework empowers you to design and develop state-of-the-art, distributed JavaEE applications using [Reactive Systems](https://www.reactivemanifesto.org/en) principles and the [RECQ architectural patterns](broken-reference). This framework streamlines the creation of enterprise-grade microservices that are:

* **Responsive:** Deliver a seamless user experience with fast and efficient communication patterns.
* **Resilient:** Maintain high availability and recover from failures gracefully.
* **Elastic:** Scale effortlessly to accommodate changing workloads.
* **Maintainable:** Foster clean, organized codebases for long-term usability.
* **Extensible:** Easily adapt and evolve your system to meet future requirements.
* **Observable:** Gain deep insights into system behaviour through comprehensive monitoring capabilities.

**Unlocking the Power of RECQ, CQRS, Event Sourcing, and DDD**

Evento leverages these industry-proven design patterns to empower your development process:

* **RECQ Patterns:** A robust architecture for building scalable and maintainable event-driven microservices.
* **CQRS (Command Query Responsibility Separation):** Simplifies data access by segregating commands that modify system state from queries that retrieve data.
* **Event Sourcing:** Provides a reliable and auditable record of all system state changes.
* **DDD (Domain-Driven Design):** Focuses on building software that reflects the core domain concepts of your application.

**Embrace a Modern Development Experience**

With Evento, you can enjoy a streamlined development workflow. The framework provides the building blocks and tools you need to focus on your application’s unique business logic.

**Join the Evento Community**

* [**Website**](https://www.eventoframework.com/)**:** Visit the Evento Framework website to learn more about the project, explore its features, and get started with development.
* [**GitHub**](https://github.com/EventoFramework/evento-framework)**:** Star the project on GitHub to show your support and stay updated on the latest developments.
* [**LinkedIn**](https://www.linkedin.com/in/gabor-galazzo/)**:** Follow the author on LinkedIn to connect with the community and contribute to the project’s growth.

**Build the future of distributed systems with Evento. Start creating robust and scalable reactive microservices today!**

# Architecture Overview

RECQ stands for **Reactive, Event-driven, CQRS (Command Query Responsibility Segregation), and Microservices**. It’s an architectural style designed to build scalable, responsive, and maintainable applications. Here’s a breakdown of its key principles:

**1. Reactive:**

* Systems built on RECQ principles are responsive to user interactions and data changes.
* They utilize asynchronous programming techniques to handle events and updates efficiently.
* This leads to a smoother user experience and better handling of high workloads.

**2. Event-driven:**

* Communication between different parts of the application happens through events.
* Events represent something that happened in the system, carrying relevant data about the change.
* Components subscribe to specific events, allowing them to react and update their functionalities accordingly.

**3. CQRS (Command Query Responsibility Segregation):**

* This principle promotes the separation of concerns by having separate components handle data reads (queries) and writes (commands).
* Command Service: This service handles user commands for adding, editing, and deleting data. It publishes events representing these actions.
* Query Service: This service handles user queries for retrieving data. It subscribes to relevant events published by the Command Service to keep its data up-to-date and respond to user requests efficiently.

**4. Microservices:**

* The application is broken down into smaller, independent services that communicate with each other.
* Each microservice has a well-defined purpose and can be developed, deployed, and scaled independently.
* This modularity improves flexibility, maintainability, and fault tolerance.

**Benefits of RECQ Architecture:**

* **Scalability:** The microservices architecture allows for horizontal scaling of individual services to handle increased load.
* **Resilience:** Failure in one microservice is less likely to cascade to others due to independent deployments and event-driven communication.
* **Maintainability:** Modular design with clear separation of concerns makes the application easier to understand and manage.
* **Responsiveness:** Reactive principles enable the application to react quickly to user interactions and data changes.

**Who should use RECQ Architecture?**

This architecture is well-suited for building:

* Modern, large-scale web applications
* Real-time and data-intensive applications
* Applications requiring high availability and scalability

**Tools for Implementing RECQ:**

* There are several frameworks available that can help implement RECQ principles.
* Examples include:
  + Akka (Scala)
  + Spring Cloud Stream (Java)
  + Lagom (Java)
  + Axon Framework (Java)
  + Evento Framework (Java) (specifically designed for RECQ with Java)

**Conclusion:**

RECQ Architecture provides a valuable set of principles for building robust and scalable applications. By leveraging its core principles of reactive programming, event-driven communication, CQRS, and microservices, developers can create applications that can handle high traffic, respond quickly to changes, and are easier to maintain.

# Distributed Systems & Microservices

The world of software development is constantly evolving, and the need for powerful, scalable applications is ever-growing. Two key concepts that facilitate this growth are **distributed systems** and **microservices architecture**. Let’s delve into each of these to lay the foundation for understanding RECQ architectures and the Evento Framework.

**Distributed Systems: Power in Numbers**

Imagine a team working on a complex project. By dividing tasks and collaborating, they can achieve more than any individual could alone. Distributed systems operate on a similar principle. They are networks of interconnected computers that **collaborate** to achieve a common goal. These computers, spread across various locations, work together to:

* **Distribute tasks and data:** By distributing workloads across multiple machines, distributed systems can significantly improve **performance**.
* **Enhance fault tolerance:** If one computer fails, others can pick up the slack, ensuring the system remains operational (increased **resilience**).
* **Scale effortlessly:** As demands grow, additional machines can be added to the network, enabling the system to handle increased loads (improved **scalability**).

However, managing a distributed system requires careful coordination. Communication between computers needs to be **consistent** (data reflects the same state across all machines) and **synchronized** (tasks are completed in the intended order).

[**Microservices Architecture**](https://microservices.io/)**: Breaking Down the Monolith**

Traditional monolithic applications are self-contained entities where all functionalities are tightly coupled within a single codebase. This approach can become cumbersome and difficult to maintain as the application grows. Microservices architecture offers a solution by breaking down a large application into **smaller, independent services**. Each service has a well-defined **business capability** and communicates with others through well-defined **APIs** (Application Programming Interfaces).

Here’s what makes microservices architecture so appealing:

* **Loose coupling:** Services are independent, allowing them to be developed, deployed, and scaled independently. This fosters agility and simplifies maintenance.
* **Focus on business capabilities:** Each service owns a specific functionality, making the codebase easier to understand and manage.
* **Improved fault isolation:** If one service fails, it only impacts its specific functionality, minimizing the overall system disruption.
* **Technology flexibility:** Different services can be built using different technologies based on their specific needs.

**The Road Ahead**

Understanding distributed systems and microservices architecture provides a solid foundation for exploring RECQ architectures and the Evento Framework. RECQ leverages these concepts to create a specific type of microservices architecture with a focus on event-driven communication. The Evento Framework, in turn, offers tools and patterns to simplify the implementation of RECQ architectures in JavaEE environments.

By combining these concepts, you can build robust, scalable, and maintainable applications well-suited for the demands of modern distributed computing.

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

# Recative Manifesto & Reactive Principles

Modern applications face unprecedented demands: delivering seamless user experiences, handling high volumes of data, and adapting to ever-changing environments. Enter the world of **Reactive Systems**, a powerful approach championed by the **Reactive Manifesto** and the **Reactive Principles**.

[**The Reactive Manifesto**](https://www.reactivemanifesto.org/)**: Building Resilient Systems for the Modern Age**

The Reactive Manifesto outlines four key principles for designing responsive, resilient, and elastic systems:

* **Responsive:** Systems should deliver timely feedback to users, ensuring a smooth and interactive experience.
* **Resilient:** They must be able to withstand failures and recover gracefully, minimizing downtime and data loss.
* **Elastic:** The ability to scale resources effortlessly, adapt to fluctuations in workloads is crucial.
* **Message-Driven:** Asynchronous message exchange promotes loose coupling and simplifies communication between components.

By adhering to these principles, developers can create applications that excel in:

* **High Concurrency:** Handling multiple requests simultaneously without performance degradation.
* **Dynamic Scalability:** Adapting to changing demands by adding or removing resources as needed.
* **Reliable User Experiences:** Delivering consistent and responsive experiences even under heavy load.

https://www.reactivemanifesto.org/

[**The Reactive Principles**](https://www.reactiveprinciples.org/)**: Putting Theory into Practice**

The Reactive Principles serve as a companion document to the Manifesto. It delves deeper, providing practical guidance and established techniques from experienced practitioners. These principles guide the development of individual services, applications, and entire systems aligned with the Reactive approach. It translates the high-level ideas of the Manifesto into actionable steps for software architects and developers, incorporating concepts from:

* **Reactive Programming:** A programming paradigm that embraces asynchronous data streams and event-driven models.
* **Reactive Systems:** The architectural style is built upon these programming principles for robust distributed applications.

**Evento Framework: Simplifying RECQ Architecture for Your Next Project**

The Evento Framework enters the scene by specifically targeting the implementation of **RECQ architectures**. RECQ stands for **Reactive Event-Driven CQRS** and is a specific type of Reactive architecture focused on building event-driven microservices. By leveraging the principles outlined in the Reactive Manifesto and the practical guidance of the Reactive Principles, Evento empowers developers to construct highly scalable and maintainable microservices within JavaEE environments.

In the following sections, we’ll delve deeper into RECQ architectures, explore the specific features of the Evento Framework, and discover how it streamlines the development of robust and responsive applications for the modern world.

# State-of-the-art Patterns

The ever-growing complexity of modern applications demands innovative approaches to ensure scalability, resilience, and maintainability. Distributed systems, where tasks and data are spread across multiple interconnected computers, have become the de facto standard for tackling these challenges. However, effectively managing distributed systems requires careful consideration of design patterns and architectural best practices.

This guide explores a collection of state-of-the-art patterns that empower developers to construct robust and performant distributed systems. We’ll delve into each pattern, highlighting its core concepts, benefits, and how it specifically contributes to the design and implementation of **RECQ architectures**. RECQ stands for **Reactive Event-Driven CQRS**, a specific type of architecture that leverages these patterns to create efficient and scalable microservices.

By understanding these core design patterns, developers can leverage the strengths of distributed systems while mitigating their inherent complexities. Let’s dive into the essential tools for building the distributed systems of tomorrow:

[**Domain-Driven Design (DDD)**](https://en.wikipedia.org/wiki/Domain-driven_design)**:**

* **Concept:** DDD is a strategic design approach that focuses on modeling an application around the core concepts of its business domain. It promotes the creation of a ubiquitous language shared by both technical and domain experts.
* **Benefits:** Improved maintainability, better communication between stakeholders, and a system that reflects real-world business processes.
* **RECQ Relevance:** DDD plays a crucial role in defining the domain model for RECQ architectures, ensuring services operate on well-defined business concepts.

[**Command Query Responsibility Separation (CQRS)**](https://cqrs.wordpress.com/wp-content/uploads/2010/11/cqrs_documents.pdf)**:**

* **Concept:** CQRS separates read (queries) and write (commands) operations into distinct models. This allows for optimization of each model for its specific purpose.
* **Benefits:** Improved performance, scalability, and flexibility. Read models can be optimized for retrieval speed, while write models can focus on data consistency.
* **RECQ Relevance:** CQRS is a fundamental principle in RECQ architectures. Commands trigger state changes in the system, while queries retrieve data from specialized read models.

[**Event Sourcing**](https://martinfowler.com/eaaDev/EventSourcing.html)**:**

* **Concept:** Event sourcing stores the history of all state changes in an application as a sequence of events. The current state is derived by replaying these events.
* **Benefits:** Improved auditability, easier implementation of temporal queries (e.g., historical data analysis), and the ability to reconstruct the application state at any point in time.
* **RECQ Relevance:** Event sourcing is a cornerstone of RECQ architectures. Events published by components represent state changes and are persisted in the System State Store (SSS).

[**Messaging Pattern (Asynchronous Message Passing)**](https://microservices.io/patterns/communication-style/messaging.html)**:**

* **Concept:** Microservices communicate with each other by asynchronously sending and receiving messages through message queues or event-driven systems.
* **Benefits:** Loose coupling between services, improved scalability (services can process messages at their own pace), and fault tolerance (failures in one service don’t necessarily block others).
* **RECQ Relevance:** RECQ architectures heavily rely on asynchronous messaging for communication between components. This promotes loose coupling and facilitates the flow of events throughout the system.

[**Saga Pattern**](https://microservices.io/patterns/data/saga.html)**:**

* **Concept:** The saga pattern coordinates a sequence of local transactions across multiple services to handle long-lived business processes. It ensures eventual consistency, meaning the system might have temporary inconsistencies, but they will eventually be resolved.
* **Benefits:** Manages complex transactional workflows that involve multiple services, ensuring data consistency across the system.
* **RECQ Relevance:** Sagas are often used within RECQ architectures for complex business processes that require coordination across multiple services. The Evento Framework might provide specific features to simplify saga implementation.

**Additional Notes:**

* These patterns are often used in conjunction with each other to build robust and scalable distributed systems.
* The RECQ architecture specifically leverages these patterns to create a specific type of event-driven microservices architecture.

# 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](broken-reference).

To start using Evento Server you need a [Postgres Database](https://www.postgresql.org/) and an instance of Evento Server that you can find on [Docker Hub](https://hub.docker.com/): <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](broken-reference) Bundle Library.

{% hint style=“danger” %} Evento Framework is compatible with[Java 21](https://openjdk.org/projects/jdk/21/) or more. {% endhint %}

You can find the library on [Maven Central](https://central.sonatype.com/): <https://central.sonatype.com/artifact/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](https://spring.io/), [Micronaut](https://micronaut.io/)or [Quarkus](https://quarkus.io/), so you can implement a RECQ application using your preferred technology even plain [JavaEE](https://it.wikipedia.org/wiki/Jakarta_EE). {% 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](broken-reference) and [Spring Boot](https://spring.io/projects/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

# 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](../../recq-patterns/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](../../recq-patterns/recq-component-pattern/aggregate.md)

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, TodoListCheckTodoCommand and produce the related events to communicate to the entire system that the state is changed.

TodoList Aggregate (Pattern View)

TodoList Aggregate Handlers

## [Projector](../../recq-patterns/recq-component-pattern/projector.md)

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](../../recq-patterns/recq-component-pattern/projection.md)

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](../../recq-patterns/recq-component-pattern/invoker.md)

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](../../recq-patterns/recq-component-pattern/service.md), [Saga](../../recq-patterns/recq-component-pattern/saga.md)and [Observer](../../recq-patterns/recq-component-pattern/observer.md)

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](../extend-todolist-handle-complexity-tutorial/)

## 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](../quick-start.md) 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 EventoServerMessageBusConfiguration(  
 // 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 && handledMessage.getMetadata().get("user") != null){  
 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](../../../recq-patterns/recq-component-pattern/aggregate.md) 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](../../../evento-framework/component/aggregate/aggregate-state.md) 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  
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.

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")  
@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;  
}

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 state){  
 // 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, TodoListAggregateState state){  
 // Command Validation  
 Assert.isTrue(command.getTodoIdentifier() != null && !command.getTodoIdentifier().isBlank(),  
 "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, TodoListAggregateState state){  
 // 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, TodoListAggregateState state){  
 // 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;  
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](../../../evento-framework/component/projector/).

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.getIdentifier())).findFirst().orElseThrow();  
 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).orElseThrow());  
 }  
  
 @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.Invoker) 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;  
  
@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), toUserMetadata(user));  
 return identifier;  
 }  
  
 @InvocationHandler  
 public void deleteTodoList(String identifier, String user){  
 getCommandGateway().sendAndWait(new TodoListDeleteCommand(identifier), toUserMetadata(user));  
 }  
   
 @InvocationHandler  
 public String addTodo(String identifier, String content, String user){  
 var todoIdentifier = "TODO\_" + UUID.randomUUID();  
 getCommandGateway().sendAndWait(new TodoListAddTodoCommand(identifier, todoIdentifier, content), toUserMetadata(user));  
 return todoIdentifier;  
 }  
   
 @InvocationHandler  
 public void checkTodo(String identifier, String todoIdentifier, String user){  
 getCommandGateway().sendAndWait(new TodoListCheckTodoCommand(identifier, todoIdentifier), toUserMetadata(user));  
 }  
   
 @InvocationHandler  
 public void removeTodo(String identifier, String todoIdentifier, String user){  
 getCommandGateway().sendAndWait(new TodoListRemoveTodoCommand(identifier, todoIdentifier), toUserMetadata(user));  
 }  
  
 @InvocationHandler  
 public CompletableFuture<TodoListView> findTodoListByIdentifier(String identifier){  
 return getQueryGateway().query(new TodoListViewFindByIdentifierQuery(identifier)).thenApply(Single::getData);  
 }  
  
 @InvocationHandler  
 public CompletableFuture<Collection<TodoListListItemView>> searchTodoList(String nameLike, int page){  
 return getQueryGateway().query(new TodoListListItemViewSearchQuery(nameLike, page, 15))  
 .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>>> searchTodoList(  
 @RequestParam(defaultValue = "") String nameLike, @RequestParam(defaultValue = "0") int page  
 ) {  
 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 = "Authorization") String user) {  
 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"), "Wrong todo identifier");  
 });  
 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");  
 });  
%}

# Extend TodoList - Handle Complexity Tutorial

# Unique identifier generation

# Extends behaviors with Observer and Services

# Cross Domain Consistency with Sagas

# Handle Real time data updates with MQTT and Save-Notify Pattern

# Introduction

RECQ stands for **Reactive, Event-driven, CQRS (Command Query Responsibility Segregation)**. It’s a comprehensive architectural approach that guides the development of modern, scalable software systems. RECQ emphasizes event-oriented microservices architectures and aims to create applications that adhere to the principles of reactive programming as outlined in the [Reactive Manifesto](https://www.reactivemanifesto.org/) and the more recent [Reactive Principles](https://www.reactiveprinciples.org/).

**Beyond the Manifesto: Adherence to Best Practices**

RECQ goes beyond just subscribing to the Reactive Manifesto. It advocates for the adoption of established design principles and patterns for building robust and maintainable systems. Here’s what RECQ promotes:

* **Modular Design:** Avoids the creation of a “[Big Ball of Mud](http://www.laputan.org/mud/)” by promoting modularity and clear separation of concerns (Dijkstra, 1982).
* [**CQRS (Command Query Separation)**](https://martinfowler.com/bliki/CommandQuerySeparation.html)**:** Enhances performance and scalability by segregating components that handle data reads (queries) from those that handle data writes (commands) (Fowler, CommandQuerySeparation, 2005).
* [**Single Source of Truth**](https://en.wikipedia.org/wiki/Single_source_of_truth)**:** Ensures data consistency by maintaining a single definitive source for all data within the system (Pang & Szafron, 2014).
* [**SOLID Principles**](https://en.wikipedia.org/wiki/SOLID)**:** Encourages the use of SOLID principles (Single Responsibility, Open/Closed, Liskov Substitution, Interface Segregation, Dependency Inversion) for well-designed and maintainable object-oriented code (Martin, PrinciplesOfOod, 2005).
* [**Uniform Access Principle**](https://en.wikipedia.org/wiki/Uniform_access_principle)**:** Promotes consistent data access patterns across the application (Meyer, 1997).

**Building Blocks of RECQ: A Focus on Patterns**

RECQ leverages specific patterns to define the structure and communication within the system:

* [**RECQ System Pattern**](recq-system-pattern/)**:** Defines the high-level modules that make up a RECQ system, such as Command Service, Query Service, Event Store, and more.
* [**RECQ Communication Pattern**](recq-communication-pattern/)**:** Establishes how these modules interact with each other using event-driven messaging.
* [**RECQ Component Pattern**](recq-component-pattern/)**:** Rigorously defines the functionalities and responsibilities of each individual component within the RECQ ecosystem.

**Additional Techniques for Enhanced Systems:**

RECQ also encourages the use of other established techniques for building robust applications:

* [**Event Sourcing**](https://microservices.io/patterns/data/event-sourcing.html)**:** Manages application state through a sequence of immutable events, providing a complete audit trail of changes.
* [**Messaging**](https://microservices.io/patterns/communication-style/messaging.html)**:** Enables asynchronous communication between microservices, improving responsiveness and scalability.
* [**Domain-Driven Design**](https://it.wikipedia.org/wiki/Domain-driven_design)**:** Guides the development process by focusing on modeling the core domain concepts of the application.

**Conclusion: A Holistic Approach to System Development**

By following the principles and patterns outlined in the RECQ architecture, developers can create event-driven, scalable, and maintainable software systems. RECQ fosters clean code practices, promotes modularity, and encourages the use of well-established design techniques for building robust and responsive applications.

**Further Exploration:**

* The Reactive Manifesto: <https://www.reactivemanifesto.org/>
* SOLID Principles: <http://principles-wiki.net/collections:robert_c._martin_s_principle_collection>

This enhanced version provides a clearer explanation of the RECQ architecture, emphasizing its focus on established design principles and patterns. It also highlights the role of additional techniques like event sourcing and messaging. Additionally, including references to the Reactive Manifesto and SOLID principles allows for further exploration.

# RECQ System Pattern

The RECQ System Pattern defines the high-level architectural components that make up a RECQ-based application. These components work together to create a modular, scalable, and event-driven system. Here’s a breakdown of the key components:

* [**Components**](component.md)**:** These are self-contained units of software that encapsulate specific functionalities of the application. Each component implements a well-defined business logic and interacts with other components through messages. Examples of RECQ components include:
  + **Command Service (Aggregate and Services):** Handles user commands for adding, editing, and deleting data. It publishes events representing these actions.
  + **Query Service (Projector and Projections):** Respond to user queries for retrieving data. It subscribes to relevant events published by the Command Service to keep its data up-to-date.
  + **Domain Logic Components (Invoker, Sagas and Observers):** Implement core business logic of the application and may interact with both Command and Query Services.
* [**Message Gateway:**](message-gateway.md) This component acts as an intermediary for communication between different components within the system. It can handle functionalities like:
  + Routing messages to the appropriate recipient component based on defined rules.
  + Implementing message transformation or validation if needed.
  + Providing a single point of entry for external systems to interact with the RECQ application.
* [**System State Store**](system-state-store.md)**:** This persistent storage mechanism is responsible for storing all the events that occur within the system. It serves as the central repository for the complete history of state changes. Both Command and Query Services can access the Event Store to retrieve historical data or for rebuilding the current state of the application if needed.

RECQ System Big Picture

**Communication and Event-Driven Interactions (**[**RECQ Communication Patten**](../recq-communication-pattern/)**):**

* Components in a RECQ system communicate with each other primarily through asynchronous messaging. They publish events representing state changes and subscribe to relevant events published by other components.
* When a component performs an action that modifies the system state, it publishes an event to the Event Store. This event contains details about the change that occurred.
* Other components that are interested in these state changes can subscribe to relevant events. When a new event is published, these subscribing components receive a notification and can update their internal state accordingly. This event-driven approach promotes loose coupling between components and enables them to react asynchronously to changes in the system.

**Benefits of the RECQ System Pattern:**

* **Modularity and Scalability:** By breaking down the application into independent components, the system becomes more modular and easier to scale. Individual components can be scaled independently based on their specific load requirements.
* **Maintainability:** Clear separation of concerns makes the application easier to understand, maintain, and modify.
* **Resilience:** Asynchronous communication and event-driven architecture make the system more resilient to failures. If one component fails, it doesn’t necessarily bring down the entire system. Other components can continue to operate based on the events they have already received.

**Beyond the Basics:**

While the core RECQ System Pattern defines these essential components, additional considerations can be factored into the design:

* **API Gateway:** An API Gateway can be introduced as a single entry point for external clients to interact with the microservices within the RECQ system.
* **Circuit Breaker Pattern:** Implementing circuit breaker patterns can improve fault tolerance by handling failing services gracefully.
* **Security Mechanisms:** Security considerations like authentication and authorization need to be addressed when designing the communication between components.

**Conclusion:**

The RECQ System Pattern provides a solid foundation for building event-driven, scalable, and maintainable microservices architectures. By understanding the core components and their interactions, developers can design robust applications that can handle the demands of modern software systems.

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.

Sample interaction of modules in a RECQ System

# Component

A RECQ component is a self-contained unit of software that implements a well-defined set of functionalities within a RECQ-based application. It plays a crucial role in promoting modularity, loose coupling, and asynchronous communication within the system.

**Properties of a RECQ Component:**

* [**Isolation**](https://www.reactivemanifesto.org/glossary#Isolation)**:**
  + **Temporal Isolation:** Components operate independently and asynchronously without relying on other components to be available at any specific time.
  + **Spatial Isolation:** Components don’t need knowledge about the location or existence of other components in the system. This concept is also known as Location Transparency.
* **Containment:**
  + Components encapsulate the logic and resources they need to fulfill their responsibilities.
  + This aligns with the concept of aggregation in Domain-Driven Design, where tightly coupled entities are grouped together within a component.
* [**Delegation**](https://www.reactivemanifesto.org/glossary#Delegation)**:**
  + Components focus on a well-defined set of tasks and delegate any unrelated functionalities to other components or external services.

**Capabilities of a RECQ Component:**

* **Message Handling:**
  + Components can receive messages addressed specifically to them.
  + They can process the message and optionally respond with a reply message to the sender.
  + This capability allows components to implement reactive behaviour and respond to external actions.
* **Internal State Management:**
  + Components can maintain a persistent internal state that reflects their specific data and information relevant to their functionalities.
* **Event-Driven Communication:**
  + Components can publish System State Change Events whenever their internal state changes due to processing a message.
  + Publishing events allows other components to be notified about these changes and potentially update their own state accordingly.
  + Components can also subscribe to specific System State Change Events published by other components.
  + By consuming events, components can react asynchronously to changes in the system without needing direct communication with the originating component.
* **Messaging Other Components:**
  + Components can send messages to other components within the system.
  + This capability allows for inter-component communication to coordinate activities and share information.
* **Replicability:**
  + A RECQ component can be replicated across multiple instances to achieve horizontal scaling.
  + The design of the component should ensure consistency even with multiple instances operating concurrently.

Strict definition:

* Can **receive messages** and may **respond** to them
* Can manage a persistent **internal state**
* Can **publish** events
* Can **subscribe** to events
* Can **send messages**
* Must be **replicable**

**Benefits of RECQ Components:**

* **Modularity:** Breaking down the application into independent components improves code organization and maintainability.
* **Loose Coupling:** Asynchronous messaging and event-driven communication reduce dependencies between components, making the system more flexible and adaptable.
* **Scalability:** Components can be scaled independently based on their specific load requirements.
* **Resilience:** Event-driven communication and independent components enhance system resilience. If one component fails, others can continue operating based on the events they’ve received.

**Conclusion:**

RECQ components are the fundamental building blocks of a RECQ architecture. By adhering to the principles of isolation, containment, and delegation, and leveraging their capabilities effectively, developers can create robust, scalable, and maintainable event-driven microservices applications.

# Message Gateway

The Message Gateway is the component in a RECQ architecture that acts as a central hub for message routing and communication between components. It promotes loose coupling and asynchronous communication by mediating interactions between components.

**Why Use a Message Gateway?**

* **Reduced Coupling:** By routing messages through the gateway, components don’t need to be aware of the specific location or implementation details of other components they need to interact with. This reduces coupling and simplifies development and maintenance.
* **Centralized Message Routing:** The gateway can implement routing logic to determine the appropriate recipient component for a message based on its content or message type.
* **Message Transformation and Validation (Optional):** The gateway can potentially transform messages to a common format or validate messages before sending them to the recipient component.

**Message Gateway Functionality:**

* [**Publish/Async Response Pattern with Correlation IDs**](https://microservices.io/patterns/communication-style/messaging.html)**:**
  + The gateway implements the Publish/Async Response pattern for command-like messages.
  + It uses correlation IDs to associate responses with the original requests, enabling the sender to identify the corresponding result.

**Sub-Modules of the Message Gateway:**

* **Command Gateway:**
  + This sub-module facilitates sending commands to the system.
  + It typically exposes a single asynchronous method called “send” which accepts a message of type “Command” as input.
  + Upon sending the command, the gateway may:
    - Route the command to the appropriate Command Service based on the command type.
    - Return the event generated by the component that handled the command (if successful).
    - Return a failure message if the command execution fails.
* **Query Gateway:**
  + This sub-module facilitates sending queries to the system.
  + It typically exposes a single asynchronous method called “query” which takes a message of type “Query” as input.
  + The gateway routes the query to the appropriate Query Service.
  + It then receives and returns an object of type “QueryResponse” containing the data retrieved by the Query Service.

Message Gateway Structure

**Benefits of Using a Message Gateway:**

* **Improved Maintainability:** Centralized message routing simplifies changes to message formats or routing logic.
* **Enhanced Monitoring and Tracing:** The gateway can act as a central point for monitoring message flows and tracing requests/responses within the system.
* **Potential Security Advantages:** The gateway can be used to implement security measures like message authorization or encryption.

**Considerations:**

* Introducing a message gateway adds another layer of complexity to the system.
* It may introduce a single point of failure if not designed and implemented with proper redundancy and fault tolerance.

**Conclusion:**

The Message Gateway can be a valuable tool in a RECQ architecture by promoting loose coupling and simplifying communication between components.

# System State Store

The System State Store (SSS) is a crucial component in a RECQ architecture. It serves as the central repository for all events that occur within the system, acting as the Single Source of Truth (SSOT) for the system state. Unlike traditional data stores that hold the current state directly, the SSS maintains an ordered sequence of change-of-state events, providing a complete audit trail of the system’s history.

**Key Responsibilities of the SSS:**

* **Publishing Domain Events:**
  + The SSS is responsible for persisting domain events published by components within the system. These events represent atomic changes in the system’s state.
  + When a component performs an action that modifies the system state, it publishes an event to the SSS.
  + The event can be associated with an aggregate, which is a group of related domain objects treated as a single unit.
* **Event Stream Retrieval:**
  + Components can retrieve event streams from the SSS.
  + An event stream is an ordered sequence of events starting from a specific point in time.
  + The retrieval can be filtered by:
    - Starting point: Specify the point in time or event sequence number from which to start retrieving events.
    - Aggregate: Retrieve events associated with a specific aggregate or group of related domain objects.
    - Event Type: Filter events based on their specific type (e.g., UserCreatedEvent, OrderPlacedEvent).

System State Store Structure

**Benefits of the SSS:**

* **Single Source of Truth:** The SSS ensures data consistency by maintaining a single definitive source for all events that have occurred within the system.
* **Audit Trail:** The complete sequence of events provides a comprehensive history of the system’s state changes, allowing for easier debugging, tracing, and compliance purposes.
* **Replayability:** The system state can be reconstructed at any point in time by replaying the sequence of events stored in the SSS. This enables functionalities like disaster recovery or rolling back to a previous state.

**Additional Considerations:**

* **Event Schema:** The SSS needs a well-defined schema for storing and retrieving events. This schema should capture relevant information about the event, such as timestamp, type, associated aggregate, and event payload.
* **Event Persistence:** The choice of a persistence mechanism for the SSS is crucial. Common options include databases (relational or NoSQL) or dedicated event streaming platforms. The chosen technology should offer scalability, durability, and efficient retrieval capabilities.
* **Event Sourcing vs. Snapshotting:**
  + While the core concept is event storage, some systems utilize snapshots in conjunction with event streams. Snapshots are periodic summaries of the system state at specific points in time. This can improve performance for retrieving the current state compared to replaying the entire event stream.

**Conclusion:**

The SSS plays a vital role in RECQ systems by providing a reliable and consistent mechanism for managing the system state. By understanding its functionalities and considerations, developers can design robust event-driven applications that leverage a complete history of state changes for various purposes.

# RECQ Communication Pattern

The RECQ Communication Pattern defines the structure and expected behaviour of messages exchanged between components within a RECQ system. This pattern promotes asynchronous communication and loosely coupled interactions, facilitating the development of scalable and resilient applications.

**Message Types in RECQ:**

* **Commands:** Messages that represent requests to modify the system state. They trigger actions within the system and potentially result in state changes. Commands are typically sent from components (e.g., user interface) to Command Services.
* **Events:** Messages published as a consequence of a successful command execution. They represent a change that has occurred in the system state. Events are published by Command Services and can be consumed by other components interested in those changes.
* **Queries:** Messages used to retrieve data from the system without modifying the state. They are typically sent from components (e.g., user interface) to Query Services.
* **View (Query Responses):** Messages containing the data retrieved by a Query Service in response to a query message. They are sent from Query Services back to the requesting component.
* **Void Responses (Optional):** Simple messages sent back by a component in response to a command, indicating successful processing without additional data. (Consideration: Some systems might prefer specific success/failure messages with details instead of a simple void response.)

RECQ Payloads

**Message Handling Protocols:**

* [**Component-to-Component**](component-to-component.md) **Communication:**
  + This pattern utilizes asynchronous message exchange (e.g., using message queues or brokers) for communication between components. This promotes loose coupling as components don’t need to wait for a response synchronously.
  + The specific protocol (e.g., request/reply, publish/subscribe) might vary depending on the message type and the desired level of interaction.
* [**Component-to-System State Store**](component-to-system-state-store.md)**:**
  + Upon successful command execution, the Command Service publishes an event to the System State Store.
  + The event contains details about the state change caused by the command.
* [**System State Store-to-Component**](system-state-store-to-component.md)**:**
  + The System State Store utilizes a publish/subscribe pattern for event distribution.
  + Components can subscribe to specific events or event types that they are interested in.
  + When a new event is published to the store, subscribed components receive a notification and can react asynchronously to the state change.

**Benefits of the RECQ Communication Pattern:**

* **Asynchronous Communication:** Enables loosely coupled components to interact without waiting for synchronous responses.
* **Event-Driven Architecture:** Promotes reactivity and allows components to react to state changes asynchronously.
* **Scalability:** Asynchronous messaging facilitates the horizontal scaling of components based on their load.
* **Resilience:** Event-driven communication makes the system more resilient to failures. If a component fails, it won’t necessarily block other components from functioning.

**Conclusion:**

The RECQ Communication Pattern provides a clear structure for message exchange within a RECQ architecture. By understanding the different message types and their corresponding handling protocols, developers can design event-driven microservices that communicate efficiently and asynchronously, leading to scalable and maintainable applications.

# Component to Component

In a RECQ architecture, components communicate with each other indirectly through asynchronous message exchange facilitated by the Message Gateway. This communication adheres to the Command Query Responsibility Segregation (CQRS) pattern, enforcing a clear separation between commands that modify the system state and queries that retrieve data without causing state changes.

**Message Types and Protocols:**

* **Commands:**
  + Commands typically follow a request/reply pattern:
    - The component sends a command message to the Message Gateway, specifying the desired action.
    - The Message Gateway routes the command to the appropriate Command Service.
    - The Command Service processes the command and performs the requested action.
    - The Command Service sends a response message back through the Message Gateway, indicating the success or failure of the operation.
    - The Message Gateway delivers the response message to the originating component.
  + Since the CQRS pattern is enforced, the response message for a command typically only includes confirmation of success or failure (e.g., boolean flag or error code) and not the updated data itself.
* **Queries:**
  + Queries typically follow a request/reply pattern:
    - The component sends a query message to the Message Gateway, specifying the requested data.
    - The Message Gateway routes the query to the appropriate Query Service.
    - The Query Service retrieves the data from the system state or relevant data store.
    - The Query Service sends a response message back through the Message Gateway, containing the requested data (the Views).
    - The Message Gateway delivers the response message containing the data (Views) to the originating component.

Component to Component Communication

**Benefits of Asynchronous Messaging and CQRS:**

* **Improved Scalability and Performance:** Asynchronous message exchange allows components to send requests without waiting for immediate responses. This decouples components and enables independent scaling based on their workload.
* **Enhanced Maintainability:** Separating commands and queries simplifies the logic within each component and promotes code clarity.
* **Optimized Event Sourcing:** Separating commands from queries, updating the system state with commands and retrieving data through queries promotes a cleaner separation of concerns for event sourcing.

**Considerations:**

* **Increased Complexity:** Asynchronous messaging and CQRS introduce additional complexity compared to simpler synchronous communication patterns.
* **Potential for Data Inconsistency:** In rare cases, if events haven’t been processed by the System State Store when a query is sent, the retrieved data might not reflect the latest state. Techniques like eventual consistency or materialized views can be used to mitigate this.

**Conclusion:**

Leveraging asynchronous messaging with the mandatory Message Gateway and adhering to the CQRS pattern provides a robust and scalable approach for component-to-component communication in a RECQ architecture. This approach promotes loose coupling, improves maintainability, and optimizes event sourcing for event-driven microservices.

# Component to System State Store

In a RECQ architecture, components interact with the System State Store (SSS) in a specific manner for managing state changes. This interaction follows a request/response pattern with a synchronous protocol managed by the SSS itself.

**Communication Protocol:**

* **Synchronous Event Posting:** Unlike the asynchronous communication used for component-to-component interaction, posting a state change event to the SSS is a synchronous operation.
  + Components send a request message directly to the SSS containing the event data representing the desired state change.
  + The SSS validates the event and attempts to persist it in its storage mechanism.
  + Upon successful persistence, the SSS sends a response message back to the component, confirming the success or failure of the operation.
  + The synchronous nature ensures the component receives confirmation before proceeding with further actions that might depend on the state change.

**Benefits of Synchronous Event Posting:**

* **Strong Consistency:** Synchronous event posting guarantees that the event is successfully persisted in the SSS before the component receives a confirmation. This ensures strong consistency between the component’s view of the state and the actual state stored in the SSS.
* **Simplified Error Handling:** The component receives immediate feedback on the success or failure of the event posting, allowing for easier error handling and potential retries if necessary.

**Considerations:**

* **Reduced Scalability:** Synchronous communication can introduce bottlenecks if a large number of components frequently update the state. This might require careful configuration and optimization of the SSS to handle high volumes of event posting requests.
* **Potential Latency:** If components rely heavily on immediate confirmation, synchronous event posting can introduce slight latency compared to asynchronous approaches.

**Conclusion:**

The RECQ architecture utilizes a synchronous request/response pattern for component-to-SSS communication when posting state change events. This approach guarantees strong consistency and simplifies error handling, but it’s important to consider potential scalability and latency impacts for high-volume systems. The specific communication strategy might be adjusted based on the application’s specific needs and performance requirements.

# System State Store to Component

In a RECQ architecture, the System State Store (SSS) utilizes a pub/sub (publish/subscribe) protocol for distributing information about state changes to interested components. This approach decouples the SSS from individual components and enables efficient event delivery.

**The Role of** [**Consumer State Stores**](../../evento-framework/bundle/consumer-state-store.md) **(CSS):**

* **Persistent Event Consumption Progress:** CSS modules are introduced to maintain the state of event consumption by individual components. This persistence allows for:
  + **Retry Logic:** If a component fails to process an event, the CSS can track the consumed events and enable retries upon recovery.
  + **Orderly Progress:** The CSS ensures that events are processed in the correct order, even in the face of failures or restarts.
  + **Consistency:** By tracking consumption progress, the CSS helps maintain consistency between the state reflected in the component and the actual state stored in the SSS.

**Communication Protocol:**

* **Pub/Sub with the SSS:**
  + The SSS acts as the publisher in the pub/sub model.
  + Whenever a new event is persisted in the SSS (e.g., after a state change), the SSS publishes the event message to a topic.
  + Topics are named channels that categorize events based on their type or purpose.
* **Subscription by Components:**
  + Components interested in receiving specific events subscribe to relevant topics managed by the SSS.
  + Subscription allows components to filter the events they receive, ensuring they only process the data they need.
* **Consumer State Stores:**
  + Components retrieve their specific consumption progress information from their associated CSS.
  + This information allows components to identify the last successfully processed event and avoid duplicate processing.
  + The CSS can also be used to store additional context related to event consumption by the component.

System STate Store to Component Communication

**Benefits of Pub/Sub and Consumer State Stores:**

* **Scalability:** The pub/sub pattern decouples the SSS from individual components, allowing for independent scaling of both components and the SSS.
* **Flexibility:** Components can subscribe to specific topics, receiving only the events relevant to their functionality.
* **Resilience:** Consumer State Stores enable retry logic and ensure orderly event processing even in the face of failures.

# RECQ Component Pattern

The RECQ architecture establishes a set of fundamental building blocks, known as RECQ components, designed to implement various functionalities within event-driven microservices. These components are defined with clear yet minimal semantics, enabling the construction of diverse applications without excessive complexity. This minimality is based on empirical observations, as most use cases can be effectively addressed with the component types described below.

RECQ Components Big Picture

**Understanding RECQ Component Capabilities**

A crucial aspect of RECQ components is their capability table. This table summarizes the specific actions each component type can perform, along with their scalability and consistency properties in accordance with the CAP theorem. Table 1 provides a generic template for this capability table.

**Table 1: Generic Capability Table for RECQ Components**

* Message Handlers - List of message handlers the component can react to.
* Invocations - List of invocations the component can make to other components.
* State Type - Indicates if the handler requires a component or instance state.
* Consistency (CAP) - Properties of the CAP theorem respected by the component:
  + C: Strong consistency
  + c: Weak consistency within managed message functions
  + A: Strong availability
  + a: Weak availability (except for same resource requests)
  + P: Partitioning tolerance (always 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 |

**State Types in RECQ Components**

There are two primary state types associated with RECQ components:

* **Instance State:** This state represents a single instance of a domain object or a specific resource. Multiple handlers within the same component can operate concurrently, as long as they handle requests for different objects or resources.
* **Component State:** This state is unique to the entire component. Only one handler can be active within a component at any given time when processing a request related to this state.

**Understanding CAP Theorem Properties**

The capability table utilizes specific terms derived from the CAP theorem to define the consistency and availability guarantees of each component type. Here’s a breakdown of these terms:

* **C (Strong Consistency):** Components with this property ensure consistency across all replicas, regardless of the message handled. This typically applies to components managing critical data that requires strict consistency.
* **c (Weak Consistency):** These components guarantee consistency only within the context of a managed message function. Data consistency across replicas might have a slight delay, but will eventually converge.
* **A (Strong Availability):** Components with strong availability offer a predictable response time regardless of the request.
* **a (Weak Availability):** These components provide predictable response times except for situations where requests target the same resource concurrently.
* **P (Partitioning Tolerance):** Being a fundamental characteristic of distributed systems, partitioning tolerance is always present in RECQ components, ensuring functionality even if network partitions occur.

In the following sections, we’ll delve deeper into each specific RECQ component type, exploring their unique capabilities and limitations within the capability table framework. This analysis will provide a comprehensive understanding of how these components enable the development of scalable and robust event-driven microservices in a RECQ architecture.

# Aggregate

RECQ Aggregate Big Picture

An Aggregate represents a single, well-defined entity within your domain model. It encapsulates the state and logic associated with that entity, acting as the central hub for managing its lifecycle.

**State Reconstruction and Replay:**

The state of an Aggregate is not directly stored within the component itself. Instead, it’s reconstructed by replaying a sequence of events pertaining to that specific Aggregate stored in the System State Store (SSS). This event stream provides a complete and auditable history of all state changes the Aggregate has undergone.

**The Command Handler: Orchestrator of Change**

Aggregates boast a single “handle” method – the command handler. This handler plays a pivotal role in processing commands that modify the state of the Aggregate. It takes two key inputs:

1. **Command Message:** This message encapsulates the specific action to be performed on the Aggregate.
2. **Aggregate State:** Represented as a sequence of events retrieved from the SSS, this provides the current state of the Aggregate necessary for the command handler to make informed decisions.

**Transformation Through Commands:**

The command handler acts as the brains of the Aggregate. It analyzes the incoming command message in the context of the current state (event history). Based on this analysis, it performs two possible actions:

* **Generate a State Change Event:** If the command is valid, the handler generates a new event that reflects the state change resulting from the command execution.
* **Return an Error:** If the command violates any business rules or constraints, the handler returns an error message indicating the failure.

**Publishing the Change Story:**

If a state change event is generated, it’s published to the SSS. This event serves as a record of the change and allows for eventual consistency to be achieved across the system.

**Domain Logic at Its Core:**

Aggregates are the primary architects of the domain logic within a RECQ architecture. They encapsulate the business rules and constraints that govern the behavior of your domain entities.

**Restrictions on Queries and Consistency:**

Command handlers within Aggregates are restricted from sending Query-type messages. This stems from the principle of CQRS and ensures that all information required for command processing is present within the Aggregate’s state (event history). Additionally, since queries offer eventual consistency, relying on them for command validation could lead to inconsistencies.

**Command Dependency and Communication:**

While Aggregates cannot send queries, they can execute other commands. This allows for complex domain logic that might involve interactions with other Aggregates. For instance, generating a unique identifier across the system might involve sending a command to a dedicated “Unique ID Generation” Aggregate.

**Scalability and Consistency Considerations:**

Maintaining strong consistency for Aggregate state requires locking mechanisms to prevent concurrent access from multiple components. This impacts availability (a), as only one operation can be processed at a time. However, it’s important to note that this is a localized consistency guarantee (Partitioned State). The overall system can still achieve eventual consistency across all Aggregates through the asynchronous event processing model. As a result, the system exhibits a trade-off between local consistency (C) and partial availability (a), but maintains overall Basic Availability.

In essence, RECQ Aggregates are powerful building blocks that provide a well-defined and consistent approach to managing domain entities and their associated logic within event-driven microservices architectures.

| 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

Projectors are specialized components dedicated to processing events published to the System State Store (SSS) and transforming them into a format optimized for efficient querying. This processed data forms the foundation of your application’s read models.

**The EventHandler Method: The Heart of Projection Logic**

Projectors expose a single method “on” – the EventHandler. This method acts as the workhorse, taking an individual Event (state change) as input. Here’s what happens within the EventHandler:

* **Event Processing:** The Projector analyzes the received Event and utilizes it to update its internal representation of the domain model. This internal model reflects the current state of the data relevant to the Projector’s purpose.

**No Return, All Transformation:**

Unlike command handlers in Aggregates, Projectors do not return any value from their EventHandler method. Their primary focus lies on transforming the event data and integrating it into the internal read model.

Projector Structure

**Ensuring Consistency, One Event at a Time**

Projectors prioritize consistency over raw processing speed. They process events in a strictly sequential manner, ensuring that the internal read model reflects the state changes in the correct order. This sequential processing limits their scalability, as they cannot efficiently handle high volumes of events concurrently.

**Singleton Pattern for System-Wide Consistency**

To maintain strong consistency across the entire system, Projectors implement the Singleton pattern at the system level. This ensures that only one instance of a specific Projector exists, preventing inconsistencies arising from parallel processing of events by multiple Projector instances.

**Internal State and Shared Consumer State Stores**

Projectors maintain an internal state that tracks their progress in consuming events from the SSS. This internal state, typically stored in a Shared Consumer State Store (CSS) using a Shared Database technique, serves two key purposes:

1. **Consistency with the SSS (SSOT):** By keeping track of the last consumed event ID, Projectors ensure they are always in sync with the SSS (Single Source of Truth) up to that point.
2. **Managing Concurrent Access:** The CSS implements mechanisms to manage concurrent access to the Projector’s internal state, preventing conflicts when multiple components might attempt to update it simultaneously.

**Query-Oriented Nature and External Data Access**

Projectors reside within the Query Model of a RECQ architecture. This means they cannot directly modify the system state. However, they can access data from other components by making Query-type requests. It’s important to remember that these queries might not always reflect the latest state due to the eventual consistency nature of the system.

**Projectors in Action: A Streamlined Approach to Read Models**

By processing events and constructing optimized read models, Projectors empower your application to deliver efficient and consistent query performance. Their focus on sequential processing guarantees strong consistency within the read models, ensuring data integrity for querying purposes. While limited in raw processing power, Projectors play a vital role in building reliable and scalable read models within RECQ architectures.

| 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 |

# Projection

RECQ Projection Big Picture

Projections act as dedicated components responsible for handling queries and delivering relevant data to the application. They leverage the materialized views constructed by Projectors for efficient data retrieval.

**The QueryHandler Method: The Power of Query Processing**

Projections expose a single prominent method – the QueryHandler. This method takes centre stage when a query message arrives:

* **Query Reception:** The QueryHandler receives a query message specifying the data the application needs.
* **Data Retrieval from Materialized View:** The Projection retrieves the requested information from the relevant materialized view, which is typically stored in a database managed by a Projector.
* **Data Transformation and Delivery:** Projections might further process or transform the retrieved data to match the specific format required by the query before sending it back as a response.

Projection Structure

**Scalability Unleashed: Stateless Design for High Performance**

Like Projectors, Projections embrace a stateless design. This means they do not maintain any persistent state information about the queries they have processed. This stateless nature allows for horizontal scaling – you can add more Projection instances to handle increasing query loads without compromising consistency.

**Consistency Considerations: Leveraging Materialized Views**

Projections rely on the eventual consistency provided by Projectors and the materialized views they create. This means the data returned by a Projection might not always reflect the absolute latest state changes, as some events might still be propagating through the system. However, this eventual consistency is often acceptable for many read-heavy scenarios, where near real-time data is sufficient.

**Query-Centric and Federated Query Support:**

Projections strictly operate within the Query Model. They cannot directly modify the system state or send Command-type messages. However, they can leverage patterns like Federated Queries to retrieve data from other Projections, even if the underlying materialized views might not be perfectly consistent due to eventual consistency. This approach can still be valuable for complex queries that require data from multiple sources.

**The Synergy Between Projectors and Projections:**

* Projectors act as the workhorses, continuously processing events from the System State Store (SSS) and materializing them into optimized, query-friendly structures within a database.
* Projections, in turn, act as the gateways to this materialized data. They efficiently retrieve and potentially transform the data stored by Projectors to fulfill incoming query requests.

**Projections in Action: Simplifying Data Retrieval with Materialized Views**

By offering a scalable and efficient way to handle queries using materialized views, Projections empower applications to retrieve data rapidly. Their stateless design allows for horizontal scaling, making them well-suited for high-traffic query workloads. While eventual consistency might introduce slight delays in reflecting the latest state changes, Projections often provide an optimal balance between performance and consistency for read-heavy scenarios.

**In essence, Projectors and Projections work in tandem to deliver a robust and scalable approach to managing read models in a RECQ architecture.**

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

# Service

RECQ Service Big Picture

Services act as specialized components responsible for handling interactions with external systems. They serve as bridges between your application and external resources like email services, payment gateways, or any other third-party functionality.

**Command-Driven Operations: Focus on External Actions**

Services primarily utilize a CommandHandler method. Unlike Aggregate command handlers, a Service’s CommandHandler takes only the command message as input. This command encapsulates the desired action for the external system.

**Event Emission: Optional, Not Mandatory**

While not every Service interaction necessitates event emission, some Services might return events. These events could signal the success or failure of the external operation and potentially be used for further processing within the RECQ system.

Service Structure

**Limited Query Functionality: Maintaining Domain Focus**

Similar to Aggregates, Services refrain from sending Query-type messages. This aligns with the principles of CQRS, ensuring a clear separation of concerns between command handling and data retrieval.

**Examples in Action: External Communication Made Easy**

* **Email Service:** A Service could be responsible for sending email notifications. The CommandHandler within the Service would receive a command containing the email content and recipient details. It would then interact with an external email provider to deliver the message.
* **Payment Processing:** Another example is a Service that delegates payment processing to a third-party provider. The Service’s CommandHandler would receive a payment command, interact with the payment gateway, and potentially emit an event reflecting the processing outcome.

**Scalability Focus: Availability Reigns Supreme**

Services prioritize availability over strong consistency within the internal system (c). Since Service interactions involve external systems, consistency guarantees are primarily the responsibility of those external providers. However, Services themselves strive to be highly available (a) to ensure smooth communication with external resources. This availability might be subject to the specific implementation of ACID properties (Atomicity, Consistency, Isolation, Durability) within the chosen external system.

**Partitioning Tolerance: A Core Strength**

Like other RECQ components, Services exhibit partitioning tolerance (P). This means they can continue functioning even if network partitions occur, preventing complete system outages due to external communication challenges.

**Services in Action: Enabling Seamless External Interactions**

Services bridge the gap between your application and the external world. They allow for scalable and focused interactions with external systems, ensuring availability for core application functionalities. By leveraging Services, your RECQ architecture can seamlessly integrate with various external resources, enhancing its overall capabilities.

| 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 |

# Invoker

RECQ Invoker Big Picture

Invokers serve as the entry points for interacting with your RECQ application. They act as bridges, allowing external entities like user interfaces (UIs), APIs, scheduled tasks, or any other source to trigger actions within the system.

**Manifestations of External Interaction:**

* **REST Controllers:** Invokers can manifest as REST controllers, exposing system functionalities through well-defined RESTful APIs. These controllers receive incoming HTTP requests and translate them into appropriate commands for the internal system.
* **UI Controllers:** Graphical user interfaces (GUIs) can also leverage Invokers. UI controllers capture user interactions and translate them into commands that Invokers can handle.
* **CLI Applications:** Command-line interfaces (CLIs) can be another way to interact with the system. CLI applications send commands directly to Invokers, which then trigger the necessary actions within the RECQ components.
* **Cron Triggers:** Scheduled tasks triggered by cron jobs can also utilize Invokers. These tasks might send commands to initiate specific system processes at predefined intervals.

Invoker Structure

**Stateless Design for Unmatched Scalability**

Invokers embrace a stateless design. This means they do not maintain any persistent state information about the interactions they handle. This stateless nature allows for significant scalability – you can effortlessly add more Invoker instances to handle increasing workloads without compromising consistency.

**Availability Reigns Supreme: Always Ready to Serve**

Due to their stateless design and focus on external communication initiation, Invokers prioritize availability (A) within the system. Their primary goal is to be readily accessible for external actors to trigger system actions.

**Partitioning Tolerance: Ensuring Robustness**

Like other RECQ components, Invokers exhibit partitioning tolerance (P). This means they can continue functioning even if network partitions occur. This ensures that external actors can still interact with the system to some extent, even if communication with other internal components might be temporarily disrupted.

**Relationship with Services: A Shared Focus**

While Invokers and Services share some similarities in facilitating external interactions, there’s a subtle distinction. Services primarily focus on communication with well-defined external systems like payment gateways or email providers. Invokers, on the other hand, act as a more generic entry point, handling a wider range of external triggers and translating them into commands for the internal system.

**Invokers in Action: Simplifying System Interaction**

By providing a centralized point for external interaction, Invokers simplify system access for various actors. Their stateless design and partitioning tolerance ensure scalability and robustness, allowing your RECQ application to effectively handle incoming requests and commands from diverse external sources. You can leverage different Invoker implementations like REST controllers, GUI controllers, or CLI applications based on your specific use case, offering a flexible approach to user interaction.

| 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 |

# Saga

RECQ Big Picture

Sagas are specialized components responsible for coordinating and ensuring consistency across distributed transactions that involve multiple Aggregates or Services. Unlike Projectors that focus on materializing read models, Sagas modify the system state itself.

**Saga Event Handler: The Heart of Transaction Orchestration**

Sagas expose a single key method – the SagaEventHandler. This method takes centre stage when a relevant Event (related to the Saga’s workflow) arrives:

* **Current State and Event as Inputs:** The SagaEventHandler receives both the current state of the Saga and the incoming Event as inputs.
* **State Update and Command/Query Execution:** Based on the current state and the received Event, the Saga can:
  + Update its internal state to reflect the progress of the workflow.
  + Send Command-type messages to relevant Aggregates or Services to trigger state changes within those components.
  + Send Query-type messages to retrieve information from Aggregates or Services to support its decision-making process.

Saga Structure

**Local State with Shared Persistence: Balancing Consistency and Performance**

Similar to Aggregates, Sagas maintain an internal state that tracks the progress of the ongoing workflow. However, unlike Aggregates whose state resides in the globally accessible System State Store (SSS), Saga state is persisted in a **Saga Shared Consumer State Store**, which follows the Shared Database pattern.

* **Shared Consumer State Store with Local Focus:**
  + This dedicated state store manages Saga instances of a specific type.
  + It provides methods to save and retrieve the state of individual Saga instances.
* **Consistency Similar to Projectors:**
  + The implementation of the Saga Shared Consumer State Store shares similarities with the Projector Consumer State Store in terms of consistency.
  + However, the details of state retrieval and synchronization across instances are optimized for Saga-specific needs.

**Scalability Constraints: Sequential Processing for Guaranteed Consistency**

Sagas, like Projectors, exhibit limitations in raw processing power due to their focus on consistent cross-component workflows. Their SagaEventHandler typically processes events in a sequential manner to ensure the correct order of operations within the Saga workflow. This sequential processing can limit scalability for high-volume workloads involving frequent Sagas.

**Relationship with Projectors: Addressing Different Needs**

Both Projectors and Sagas play critical roles in RECQ architectures, but they cater to distinct needs:

* **Projectors:** Focus on materializing optimized read models from event streams for efficient query processing.
* **Sagas:** Orchestrate complex workflows across multiple components while ensuring consistency within those workflows.

**Sagas in Action: Ensuring Consistency in Complex Workflows**

Sagas are invaluable for managing scenarios where a single action might trigger a sequence of interactions across multiple Aggregates or Services. By orchestrating these interactions and ensuring consistent state changes, Sagas guarantee the overall integrity of distributed transactions within your RECQ application. While scalability constraints might exist, Sagas offer a powerful mechanism for handling complex workflows that require strong consistency across multiple components.

| 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 |

# 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

Evento Framework is a Java library designed to simplify the development of reactive systems based on the RECQ architecture. RECQ stands for:

* **Reactive:** Systems are responsive to user interactions and data changes.
* **Event-driven:** Communication between different parts of the application happens through events.
* **CQRS (Command Query Responsibility Segregation):** Separate components handle data reads (queries) and writes (commands) for better performance and scalability.
* **Microservices:** The application is broken down into smaller, independent services that communicate with each other.

Evento Framework provides functionalities to implement these RECQ principles effectively. Here are some key features of Evento:

* **Event Handling:** Evento offers functionalities for publishing and subscribing to events, enabling communication between different parts of the application.
* **Reactive Components:** The framework helps build reactive components that respond asynchronously to events and data changes.
* **CQRS Support:** Evento streamlines the implementation of CQRS architecture by providing separate mechanisms for handling commands and queries.
* **Simplified Communication:** It facilitates communication between microservices using asynchronous messaging patterns.

**Benefits of using Evento Framework:**

* **Improved Maintainability:** By promoting modularity and clear separation of concerns, Evento makes applications easier to understand and maintain.
* **Scalability:** The microservices architecture supported by Evento allows for easier scaling of the application as needed.
* **Resilience:** Event-driven communication helps build more resilient applications as failures in one component are less likely to cascade to others.
* **Faster Development:** The framework provides reusable components and patterns, speeding up development compared to building everything from scratch.

**Who should use Evento Framework?**

Evento Framework is well-suited for developers who want to build:

* Reactive and scalable Java applications
* Applications leveraging the RECQ architecture
* Microservices-based systems with asynchronous communication

# 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;  
  
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

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;  
import java.lang.annotation.RetentionPolicy;  
import java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.RUNTIME)  
@Target({ElementType.TYPE})  
@Component  
public @interface Projector {  
 int version();  
}

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

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> queryMessage) {  
 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