



Analysis & Design Guidelines

Technical Design for Service Domain Specifications

Table of Contents

[1 Overview 5](#_Toc26787222)

[2 Domain Driven Design 6](#_Toc26787223)

[3 Technology 15](#_Toc26787224)

[4 References 78](#_Toc26787225)

**Table of Figures**

[Figure 1 - Loans Management Problem Space 6](#_Toc26787226)

[Figure 2 – Subdomains withing Loans Management 7](#_Toc26787227)

[Figure 3 – Loans Management Solutioned as separate Bounded Conexts 8](#_Toc26787228)

[Figure 4 – Metamodel for NDL Architecture 9](#_Toc26787229)

[Figure 5 – Domain Model of a Bounded Context business language to technical language 10](#_Toc26787230)

[Figure 6 – Aggregates / Entities / Value Objects within the Credit Eligibility Assessment Record Bounded Context 11](#_Toc26787231)

[Figure 7 – Domain Rules within Credit Eligibility Policy Administration Bounded Context 12](#_Toc26787232)

[Figure 8 – Commands / Queries within the Eligibility Assessment Bounded Context 13](#_Toc26787233)

[Figure 9 – Events within the Eligibility Assessment Bounded Context 14](#_Toc26787234)

[Figure 10 – Spring Platform Projects 16](#_Toc26787235)

[Figure 11 - Microservices platform requirements 16](#_Toc26787236)

[Figure 12 - Microservices platform components provided by Spring Boot 17](#_Toc26787237)

[Figure 13 - Microservices platform components provided by Spring Cloud 18](#_Toc26787238)

[Figure 14 - Anatomy of a Spring Boot application 19](#_Toc26787239)

[Figure 15 - Deployment Architecture for our Spring Boot-based microservices 20](#_Toc26787240)

[Figure 16 - Package structure for the Bounded Contexts 22](#_Toc26787241)

[ֿFigure 17 - Bounded Context Spring Boot Application’s package structure 23](#_Toc26787242)

[Figure 18 - Package structure for interfaces 24](#_Toc26787243)

[Figure 19 - Package structure for Application services 25](#_Toc26787244)

[Figure 20 - Package structure for domain model 26](#_Toc26787245)

[Figure 21 - Package structure for the infrastructure components 27](#_Toc26787246)

[Figure 22 - Package structure for any of our Bounded Context 28](#_Toc26787247)

[Figure 23 - Logical grouping of the DDD artifacts 29](#_Toc26787248)

[Figure 24 - Loan Microservices solution 31](#_Toc26787249)

[Figure 25 - pom.xml dependency maintainance 32](#_Toc26787250)

[Figure 26 - MySQL connection configuration 32](#_Toc26787251)

[Figure 27 - EligibilityAssessment root aggregate as a JPA Entity 33](#_Toc26787252)

[Figure 28 - Identifier for the EligibilityAssessment root aggregate 33](#_Toc26787253)

[Figure 30 - pom.xml mongoDB dependency maintainance 34](#_Toc26787254)

[Figure 31 - Eligibility Assessment Request Aggregate and its dependent associations 36](#_Toc26787255)

[Figure 32 - EligibilityAssessment root aggregate dependencies as business objects 37](#_Toc26787256)

[Figure 33 - ProductGroup entity Object 38](#_Toc26787257)

[Figure 34 – Product value object implementation 39](#_Toc26787258)

[Figure 35 - Eligibility Assessment Command class implementation 41](#_Toc26787259)

[Figure 36 - Command handler for the EligibilityAssessment command 41](#_Toc26787260)

[Figure 37 - Class diagram for the Command Handler implementation 42](#_Toc26787261)

[Figure 38 - Class diagram for the Error Handling Implementation 46](#_Toc26787262)

[Figure 39 - Named queries within the Eligibility Assessment root aggregate 46](#_Toc26787263)

[Figure 40 - Flow of Events in a Microservices architectureThere are four stages to the implementation of a robust event-driven choreography architecture : 48](#_Toc26787264)

[Figure 41 - AbstracAggregateRoot template class 49](#_Toc26787265)

[Figure 42 - Event registration within the Eligibility Assessment root aggregate 50](#_Toc26787266)

[Figure 43 - Eligibility Assessment Created Event implementation class 51](#_Toc26787267)

[Figure 44 - EligibilityAssessmentCreatedEventData implementation class 51](#_Toc26787268)

[Figure 45 - Class diagram for the Aggregate Event Registration implementation 52](#_Toc26787269)

[Figure 46 - Domain Model Services implementation summary 53](#_Toc26787270)

[Figure 47 - EligibilityAssessmentController implementation class 55](#_Toc26787271)

[Figure 48 - EligibilityAssessmentResource implementation class 56](#_Toc26787272)

[Figure 49 – EligibilityAssessmentCommandDTOAssembler implementation class 57](#_Toc26787273)

[Figure 50 – EligibilityAssessmentCommand implementation class 59](#_Toc26787274)

[Figure 51 - Class diagram for the REST API implementation 59](#_Toc26787275)

[Figure 52 - Inbound Services implementation process summary 60](#_Toc26787276)

[Figure 53 - Implementation process summary for Event Handler implementations 61](#_Toc26787277)

[Figure 54 - Responsibilities of the Application Services 62](#_Toc26787278)

[Figure 55 - EligibilityAssessmentCommand Application services class implementation 63](#_Toc26787279)

[Figure 56 – EligibilityAssessmentQuery Application services implementation 64](#_Toc26787280)

[Figure 57 - Class diagram for our Application Services Command / Query delegation 65](#_Toc26787281)

[Figure 58 - Application Services implementation process summary 66](#_Toc26787282)

[Figure 59 - Outbound Services 67](#_Toc26787283)

[Figure 60 - EligibilityRepository JPA interface 68](#_Toc26787284)

[Figure 61 - Outbound Services - Repository implementation 69](#_Toc26787285)

[Figure 62 - HTTP invocation between two Bounded Contexts 70](#_Toc26787286)

[Figure 63 - Anti-corruption Layer between two Bounded Contexts 71](#_Toc26787287)

[Figure 64 - Outbound Services (HTTP) implemenation process 71](#_Toc26787288)

[Figure 65 - Class diagram for the Event Publisher implementation 73](#_Toc26787289)

[Figure 66 - EligibilityAssessmentEventPublisherService class implementation 74](#_Toc26787290)

[Figure 67 - EligibilityAssessmentEventSource implementation class 74](#_Toc26787291)

[Figure 68 - EligibilityAssessmentCreatedEvent implementation class 75](#_Toc26787292)

[Figure 69 - DDD artifact implementation summary using Spring Boot 77](#_Toc26787293)

Document Information

|  |  |  |  |
| --- | --- | --- | --- |
| **Project Name:** | Loans in Digital Channels | | |
| **Prepared By:** |  | **Document Version No:** | 0.1 |
| **Title:** |  | **Document Version Date:** | 13-Nov-19 |
| **Reviewed By:** |  | **Review Date:** |  |

Distribution List

| **From** | **Date** | **Phone/Fax/Email** |
| --- | --- | --- |
| IBM Architecture Team | 13-Nov-19 |  |
|  |  |  |

| **To** | **Action\*** | **Due Date** | **Phone/Fax/Email** |
| --- | --- | --- | --- |
| BankXXXX Architecture Team | Review |  |  |
|  |  |  |  |
|  |  |  |  |
|  |  |  |  |

\* Action Types: Approve, Review, Inform, File, Action Required, Attend Meeting, Other (please specify)

Document Version History

| **Version Number** | **Version Date** | **Revised By** | **Description** | **Filename** |
| --- | --- | --- | --- | --- |
| 0.1 | 13-Nov-19 |  | Initial version |  |
|  |  |  |  |  |
|  |  |  |  |  |

# Overview

This Document will detail the third DDD implementation of our Loans application using the Spring platform. The Loans application will be designed using a microservices-based architecture, and we will map the DDD artifacts to the corresponding implementations available within the Spring Platform. We are taking as starting point the following inputs:

[[1]](#_References) Loans Design v0.1

[[2]](#_References) Documents from business analisis&design 🡪 To Be add

With the previous inputs, we will proceed to define all the components and structures which are needed to implement the definition from business analysis.

The main technique we are going to use to implement the technical solution is **Domain Driven Design** (DDD), so in the first part of this document we will describe briefly the main concepts of DDD and then we will move on to the implementation using **Spring Boot Framework** as technical platform [**[3]**](#_References).

Because this document is an application document, it is not in the scope of this document describe in detail the technical layer of architecture, which should be built upon the Spring Boot Framework. However, we will raise the gaps and needs from the application point of view (logging, monitoring, tracing,…) that are needed to provide.

# Domain Driven Design

DDD, as described in the book [[4]](#_References) *Domain-driven design* by Eric Evans (Addison-Wesley Professional, 2003), is an approach for building complex software applications that is centered on the development of an object-oriented domain model.

This design technique can be used to solve problems within the domain. DDD also defines the vocabulary used by the team, what DDD calls the **Ubiquitous Language**. The domain model is closely mirrored in the design and implementation of the application.

DDD has two concepts that are very useful when applying the microservice architecture: **subdomains** and **bounded contexts**. We will try to describe these two important concepts in further sections.

## DDD Concept

### Problem space / Business Domain

The first main concept of DDD that we would need to familiarize ourselves with is the identification of the “**Problem Space**” or the “Business Domain”. The Problem Space/Business Domain is the starting point of the DDD journey, and it identifies the main business problem that you intend to solve using DDD.

Let’s try to clarify this concept using a practical sample. The problem space in this case can be classified as **Loans Management** which can also be termed as your **core business domain** and a **business problem** that you would like to solve using Domain Driven Design.

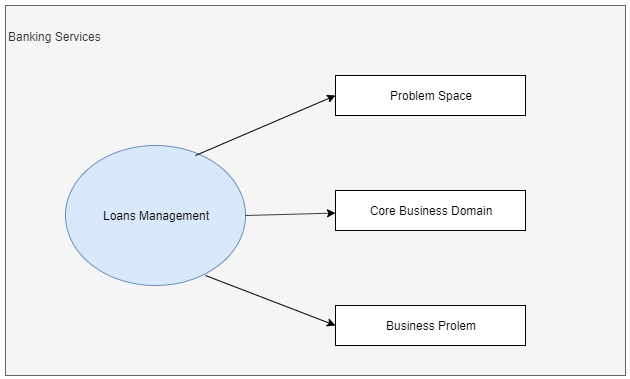


Figure 1 - Loans Management Problem Space

### Subdomains / Bounded Contexts

Once we have identified the main Business Domain, the next step is to break the domain into its sub-domains. The identification of the sub-domains essentially involves the breaking down of the various business capabilities of your main business domain into cohesive units of business functionalities.

Again, we are going to continue with the sample of Loans Management. We are going to show three subdomains, but you have to keep in mind that is not the final decomposition of the system. We are using just to illustrate what we want to mean. If you want further details, please refer to functional document (i.e. **BankXXXX Bank - Component and Business Object Model v3.3.docx** [[5]](#_References)). As is evident, the sub-domains are determined in terms of business capabilities of business, the sub-domains are not defined according technical specifications, business comes first. It is very important remember this point, emphasizing on the actual business capabilities helps in cleanly identifying the sub-domains.

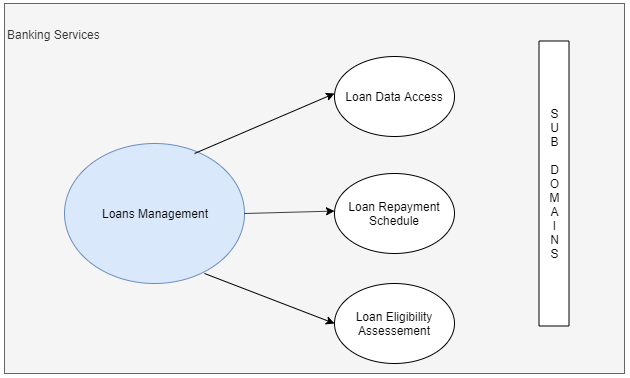


Figure 2 – Subdomains withing Loans Management

* **Loan Data Access – Subdomain**

The "Loan Data Access" handle the access to loans information in the ODS, or in the loan Application in Mainframe.

* **Loan Repayment Schedule – Subdomain**

The "Loan Repayment Schedule Administration" subdomain provide the interfaces with loan processor to calculate repayment schedule and keep track of them for auditing purposes.

* **Loan Eligibility Assessment – Subdomain**

The "Eligibility Assessment" subdomain manages and records the conditions and policies in order to make an eligibility assessment for a loan to a final customer.

Now it is time to define what is our understanding about **Bounded Context**. We define a bounded context as design solutions to implement our identified Business Domains / Subdomains.

Following DDD principles and definitions, Bounded Contexts will help us to move from the Problem Space area to Solution Space area. Let’s try to explain this using a picture with the samples used before[[1]](#footnote-1):

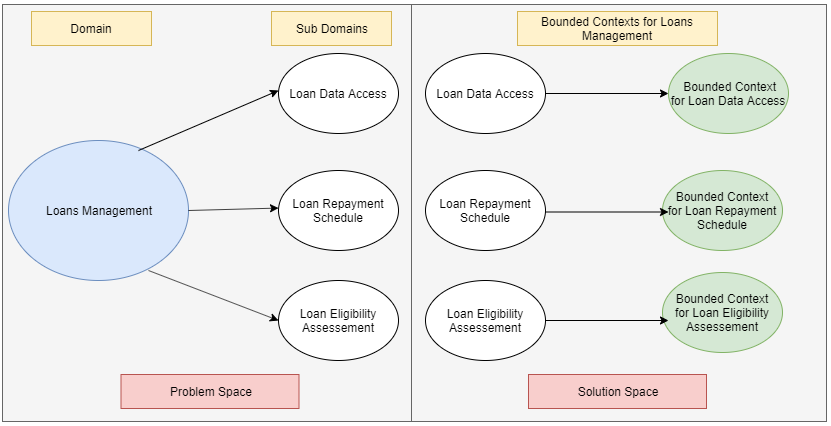


Figure 3 – Loans Management Solutioned as separate Bounded Conexts

## Domain Model

Now, we have arrive to the most important part of our modelling process. Now, we have to define how are we going to implement our **core business logic** within a Bounded Context, to this is what we call Bounded Context Domain Model or in short: **Domain Model**.

According to the deliverable **BankXXXX Bank - Component and Business Object Model v3.3.docx** [[5]](#_References)

The following view describes the elements used to describe the Target Architecture for NDL.

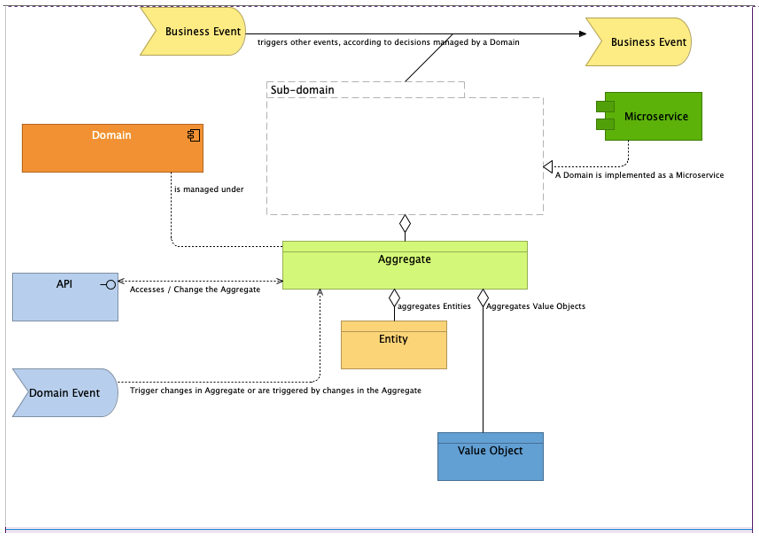


Figure 4 – Metamodel for NDL Architecture

This picture will provide a wider context to help readers to understand the relationships of components (domains and subdomains) and business objects (aggregates, entities and value objects) with other elements in the Architecture from a functional point of view. Our goal in this document is transalate this functional component to technical components. In the following picture we are going to show how the business language maps against the technical language in DDD inside a Bounded Context:

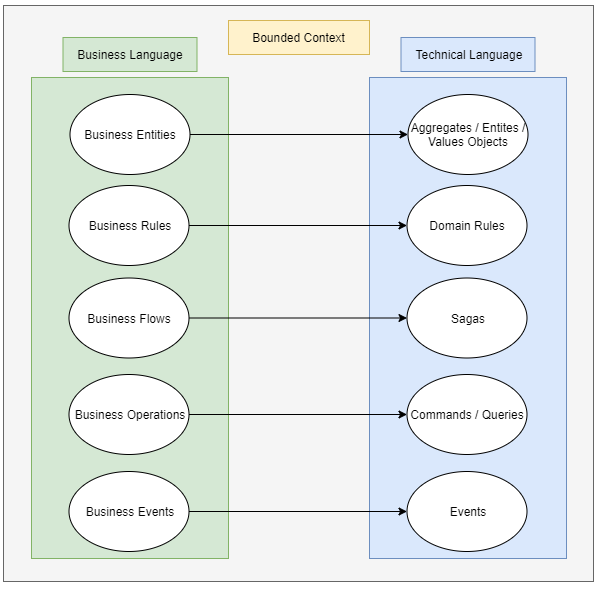


Figure 5 – Domain Model of a Bounded Context business language to technical language

## Domain Model

To explain the main concepts in the domain model, we will use as a sample a simplified diagram for the Eligibility Assessement Subdomain[[2]](#footnote-2) in order to get more clarity in what we want to expose.

### Aggregates / Entity Objects / Value Objects

**Aggregate** (also known as the **root aggregate**) is the key business object within your Bounded Context and defines the scope of consistency within that Bounded context. We have to keep in mind that our Bounded Context begins and ends within our root aggregate.

**Entity Objects** have an identity and structure of their own, however entity objects cannot exist without the root aggregate. What does it means? It means that entity objects are created when the root aggregate is created and are destroyed when the root aggregate is destroyed. The entity objects cicle of life is managed by the root aggregate.

**Value Objects** have no identity and are easily replaceable within an instance of a root aggregate or an entity.

As an example, we are going to take a part of the Eligibility Assessement subdomain to explain the previous concepts:

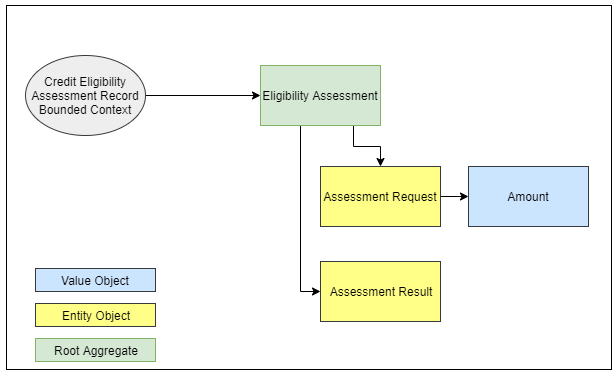


Figure 6 – Aggregates / Entities / Value Objects within the Credit Eligibility Assessment Record Bounded Context

The **Eligibility Assessment Aggregate** is the root aggregate within the Credit Eligibility Assessment Record Bounded Context. Without a Eligibility Assessment, nothing exists within this bounded context, hence no principal identifier within this Bounded Context or the root aggregate.

The **Assessment Request** Entity Object captures the applicant details for the assessment request (account info, customer info, Product info, etc.). It has an identifier of its own but cannot exist without the Eligibility Assessment, that is, when the eligibility assessment is created, the details are created; likewise, when eligibility assessment is cancelled, the details are removed.

The **Assessment Result** Entity Object captures the result for the assessment request (result conditions). It has an identifier of its own but cannot exist without the Eligibility Assessment, that is, when the eligibility assessment is created, the details are created; likewise, when eligibility assessment is cancelled, the details are removed.

The **Amount** Value Object denotes the amount for the assessment. It has no identity of its own and can be replaced in a Eligibility Assessment Aggregate instance.

### Domain Rules

Domain Rules are pure business rule definitions. Modeled as Objects too, they assist the Aggregate for any kind of business logic execution within the scope of a Bounded Context.

To try to explain this, we are going to use the Sub-domain Credit Eligibility Policy Administration. In this sample we will be focused just in the Policy Criteria aggregate.

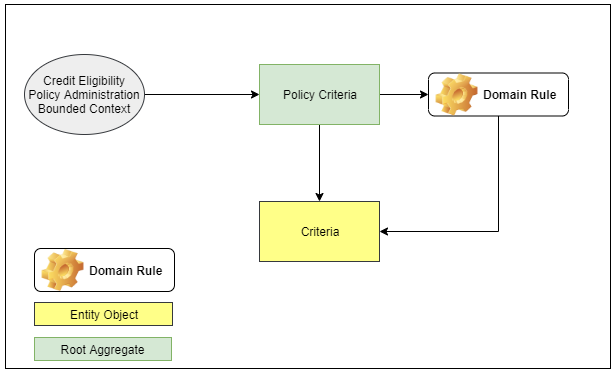


Figure 7 – Domain Rules within Credit Eligibility Policy Administration Bounded Context

Within our Credit Eligibility Policy Bounded Context, a good example of a Domain Rule is a “Validation for State of Eligibility Assessment” Business Rule. The rule basically states that depending upon the “state” of the Credit Eligibility Policy Adminsitration (e.g., active, rejected, pending approval…), additional validation checks could be applicable to the credit eligibility policy applicant.

### Sagas

So far, we have not identified the need to implement this pattern in the design and building of our solutions. So, it is no in the scope of this document provide a detail description of this pattern, we are going just to provided a quick reference to this pattern, and we will be redirecting the readers to several interesting links where this pattern is widely described.

Sagas are the only artifact that is not restricted to a single Bounded Context and may span across multiple Bounded Contexts. Sagas react to multiple business events across Bounded Contexts and “orchestrate the business process” by coordinating interactions among these Bounded Contexts.

For further information about this pattern, please refer to the following links [[6]](#_References).

### Commands / Queries

Commands and Queries represent any kind of operations within the Bounded Context which either affect the state of the aggregate/entity or query the state of the aggregate/entity.

We are going to use the following figure to show some examples of Commands within the Eligibility Assessement Bounded Context include “Create a Eligibility Assessment” and “Delete a Eligibility Assessment,” while examples of queries include “Get Eligibility details for an Assessment” and “Get All Eligibility Assessments”.

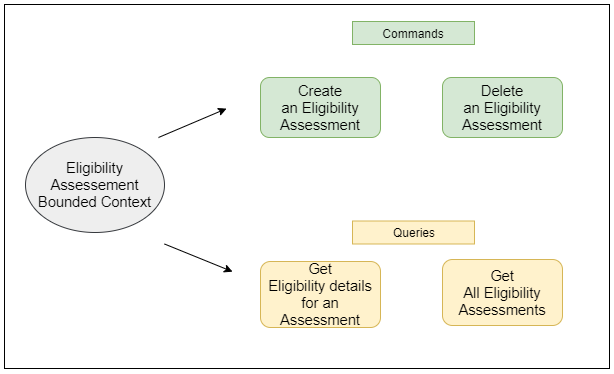


Figure 8 – Commands / Queries within the Eligibility Assessment Bounded Context

### Events

Events capture any kind of state change either with an aggregate or an entity within the Bounded Context.

To illustrate this, we are going to represent a few events within the Eligibility Assessment Bounded Context.

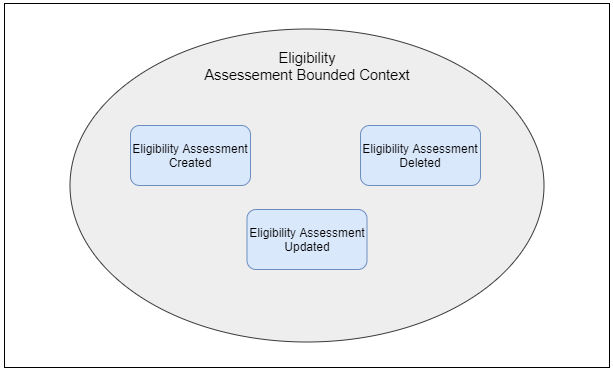


Figure 9 – Events within the Eligibility Assessment Bounded Context

# Technology

## The Spring Platform

Originally released as an alternative to Java EE, the Spring Platform (<https://spring.io/>) has become the leading Java framework to build enterprise applications. The breadth of functionality offered via its project portfolio is extensive and covers almost every aspect required to build enterprise applications.

The project portfolio covers the following main areas:

* ***Core Infrastructure Projects*** which provide a foundational set of projects to build Spring-based applications
* ***Cloud-Native Projects*** which provide capabilities to build Spring applications with cloud-native capabilities
* ***Data Management Projects*** which provide capabilities to manage any kind of data within Spring-based applications

The individual projects within the platform are listed in the following Figure:

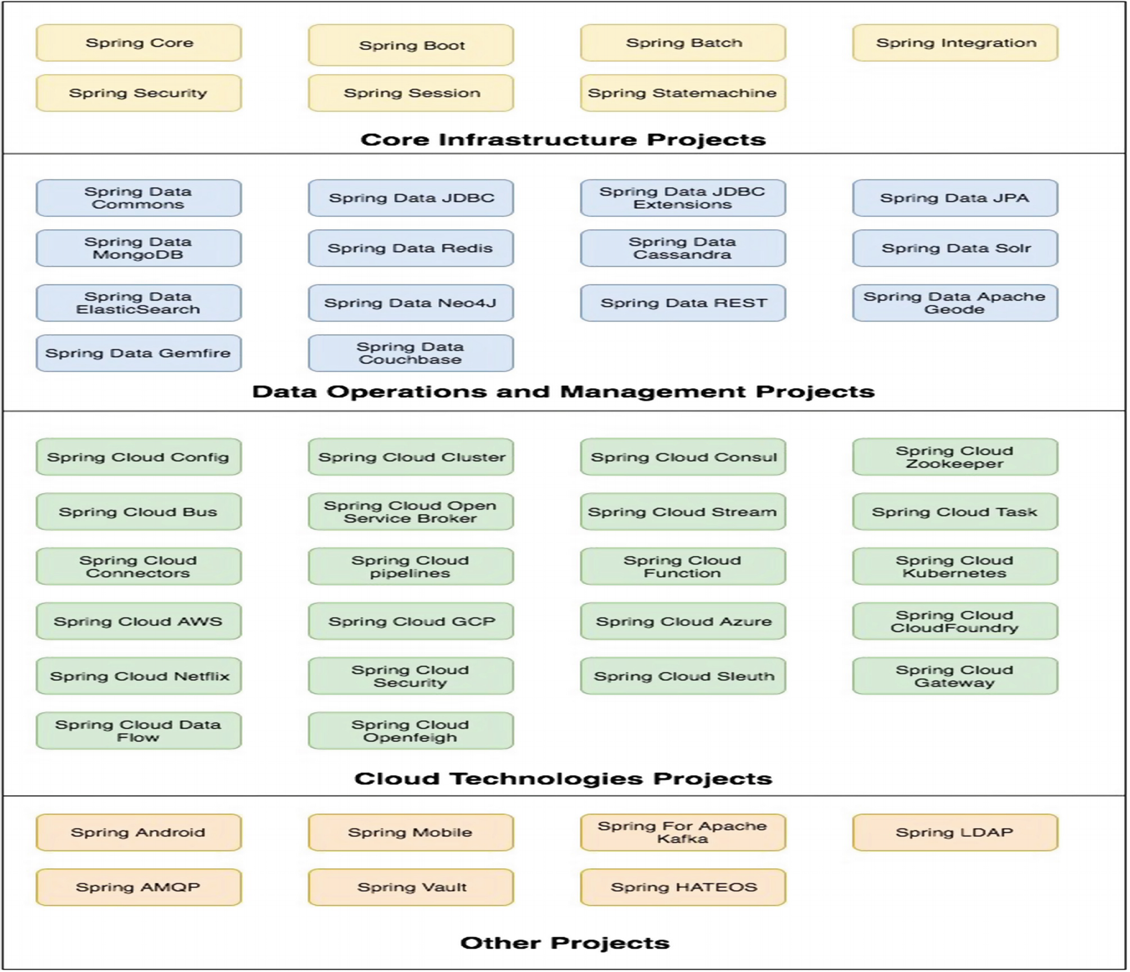


Figure 10 – Spring Platform Projects

As seen, the breadth of projects is large and provides a vast range of capabilities. To reiterate, the stated goal, is to implement our projects utilizing DDD principles based on a microservices architecture. To that extent, we will just use a subset of the available projects ***(Spring Boot, Spring Data, and Spring Cloud Stream)*** to help us achieve our goal.

The requirements of a Microservices platform are illustrated in the following Figure:

A screenshot of a cell phone

Description automatically generated

Figure 11 - Microservices platform requirements

### Capabilities

Spring Boot acts as the foundational piece for any Spring-based microservices application. A highly opinionated platform, Spring Boot helps build microservices with REST, Data, and messaging capabilities using a uniform development experience. This is done by an abstraction/dependency management layer that Spring Boot implements on top of the actual projects that provide REST, Data, and Messaging capabilities(\*for Events and Messaging\*). As a developer, you want to avoid the hassles of managing the dependencies as well as the configuration required when you build your microservices application. Spring Boot abstracts all of these for the developer by providing starter kits. The starter kits provide the required scaffolding to enable the developers to quickly start developing microservices which need to expose API(s), process data, and participate in event-driven architectures. In our implementation, we are going to be relying on three starter projects provided by Spring Boot (***spring-boot-starter-data-jpa…..***).

From a microservices requirements mapping perspective, the boxes in green as illustrated in the following Figure are implemented with Spring Boot:

A screenshot of a cell phone

Description automatically generated

Figure 12 - Microservices platform components provided by Spring Boot

### Poalim Spring Cloud Infrastructure

While Boot provides the foundational technologies for building microservices applications, Spring Cloud helps implement the distributed systems patterns that Spring Boot–based microservices applications require. These include externalized configuration, service registration and discovery, messaging, distributed tracing, and API gateways. In addition, this project also provides projects to natively integrate with a third-party cloud providers like AWS/GCP/Azure.

From a microservices requirements mapping perspective, the boxes in orange as illustrated in the following Figure are implemented with (**Should be completed by Sergy** )

A screenshot of a cell phone

Description automatically generated

Figure 13 - Microservices platform components provided by Spring Cloud

### Bounded Context(s) with Spring Boot

The Bounded Context is the starting point of our solution phase for our DDD implementation of the Loans microservices application based on Spring. In the microservices architectural style, each Bounded Context has to be a ***self-contained independent deployable unit*** with no direct dependency on any other Bounded Context within our problem space.

The pattern for splitting the Loan application into multiple microservices will be as before, that is, we split the core domain into a set of ***Business Capabilities/Sub-Domains*** and ***solution each of them as a separate Bounded Context.***

Implementing the Bounded Contexts involves a logical grouping of our DDD artifacts into a single deployable artifact. Each of our Bounded Contexts within the Loans application is going to be built out as a Spring Boot Application. The resultant artifact of a Spring Boot Application is a ***self-contained fat JAR file*** which contains all the required dependencies (e.g., data access libraries, REST libraries) and configuration. The fat JAR file also contains an embedded web container (in our case Tomcat) as the runtime. This ensures that we do not need any external application server to run our fat *JAR*.

The anatomy of a Spring Boot application is illustrated in the following Figure:

A screenshot of a cell phone

Description automatically generated

Figure 14 - Anatomy of a Spring Boot application

Microservices will need a ***DataStore*** to store their state. We choose to adopt the ***Database per service pattern,*** that is, each of our microservices will have its own separate DataStore. Just like we have a polyglot choice of technology for our application tier, we have a polyglot choice for the DataStore too. We could choose to have a plain Relational Database (e.g., Oracle, MySQL, PostgreSQL), a NoSQL Database (e.g., MongoDB, Cassandra), or even an in-memory datastore (e.g., Redis). The choice depends primarily on the scalability requirements and the type of use case the microservices intends to cater to. For our implementation, we decide to go with mongoDB as the choice of DataStore.

The deployment architecture is illustrated in the following Figure:

A screenshot of a map

Description automatically generated

Figure 15 - Deployment Architecture for our Spring Boot-based microservices

### Bounded Contexts: Packaging

To get started with our packaging, the first step is to create a regular Spring Boot application. We will use the the poalim-starter web UI tool which helps create boilerplate template for the microservice. The output of the tool is a zip file that contains the all boilerplate needed for a microservice.

* *Needs to add the link for the starter .*
* *Needs to add a printscreen of the starter.*

### Bounded Contexts: Package Structure

With the packaging aspect decided, the next step is to decide the package structure of each of our Bounded Contexts, that is, to arrive at a logical grouping of the various DDD artifacts into a single deployable artifact. The logical grouping involves identifying a package structure where we place the various DDD artifacts to achieve our overall solution for the Bounded Context.

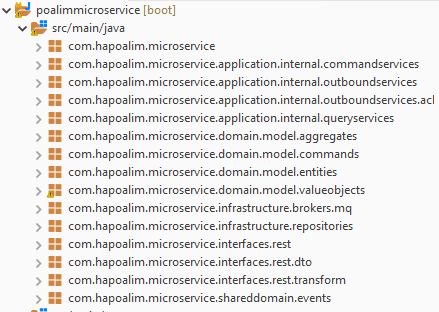
The high-level package structure for any of our Bounded Context is illustrated in the following Figure:

A screenshot of a cell phone

Description automatically generated

Figure 16 - Package structure for the Bounded Contexts

An example of a microservice Bounded Context Spring Boot Application’s package structure is shown in the following Figure:



ֿFigure 17 - Bounded Context Spring Boot Application’s package structure

### Packages Structure

#### Interfaces

This package encloses all the inbound interfaces to our Bounded Context classified by the communication protocol. The main purpose of ***interfaces*** is to negotiate the protocol on behalf of the Domain Model (e.g., REST API(s), WebSocket(s), FTP(s), Custom Protocol).

As an example, the Credit Offer Bounded Context provides REST APIs for ***State Change Request, that is, Command***,to it (Create New Offer) Similarly, the Credit Offer Bounded Context provides REST APIs for sending ***State Retrieval Requests, that is, Queries,*** to it (Retrive an Offer). This is grouped into the “***rest***” package.

It also has Event Handlers which subscribe to the various Events that are generated by other Bounded Contexts. All Event Handlers are grouped into the “***eventhandlers”*** package. In addition to these two packages, the interface package also contains the “***transform***” package. This is used to translate the incoming API Resource/Event data to the corresponding Command/Query model required by the Domain Model.

Since we need to support REST, Events, and data transformation, the package structure is as illustrated in the following Figure:

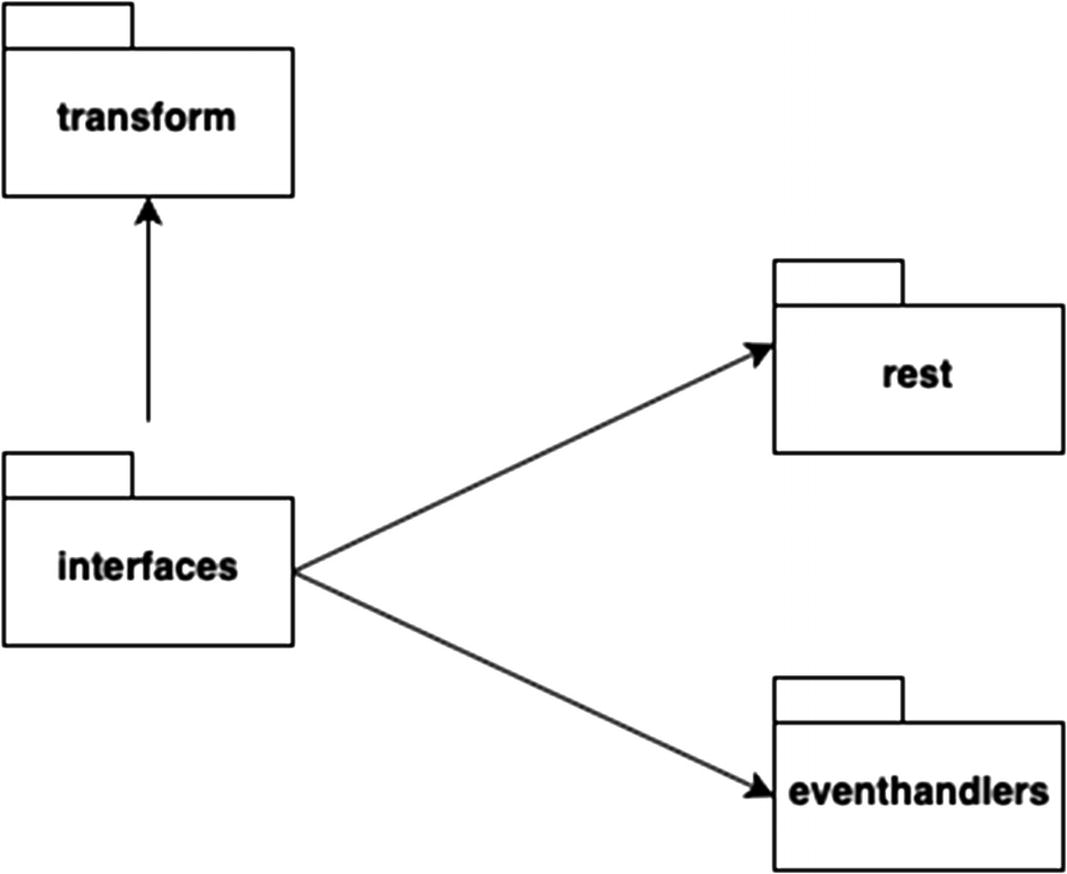


Figure 18 - Package structure for interfaces

#### Application

Application services act as the façade for the Bounded Context’s Domain Model. They provide façade services to dispatch Commands/Queries to the underlying Domain Model. They are also the place where we place outbound calls to other Bounded Contexts as part of the processing of a Command/Query.

To summarize, Application Services

* Participate in Command and Query Dispatching
* Invoke infrastructural components where necessary as part of the Command/Query processing
* Provide Centralized concerns (e.g., Logging, Security, Metrics) for the underlying Domain Model
* Make callouts to other Bounded Contexts

The package structure is illustrated in the following Figure:

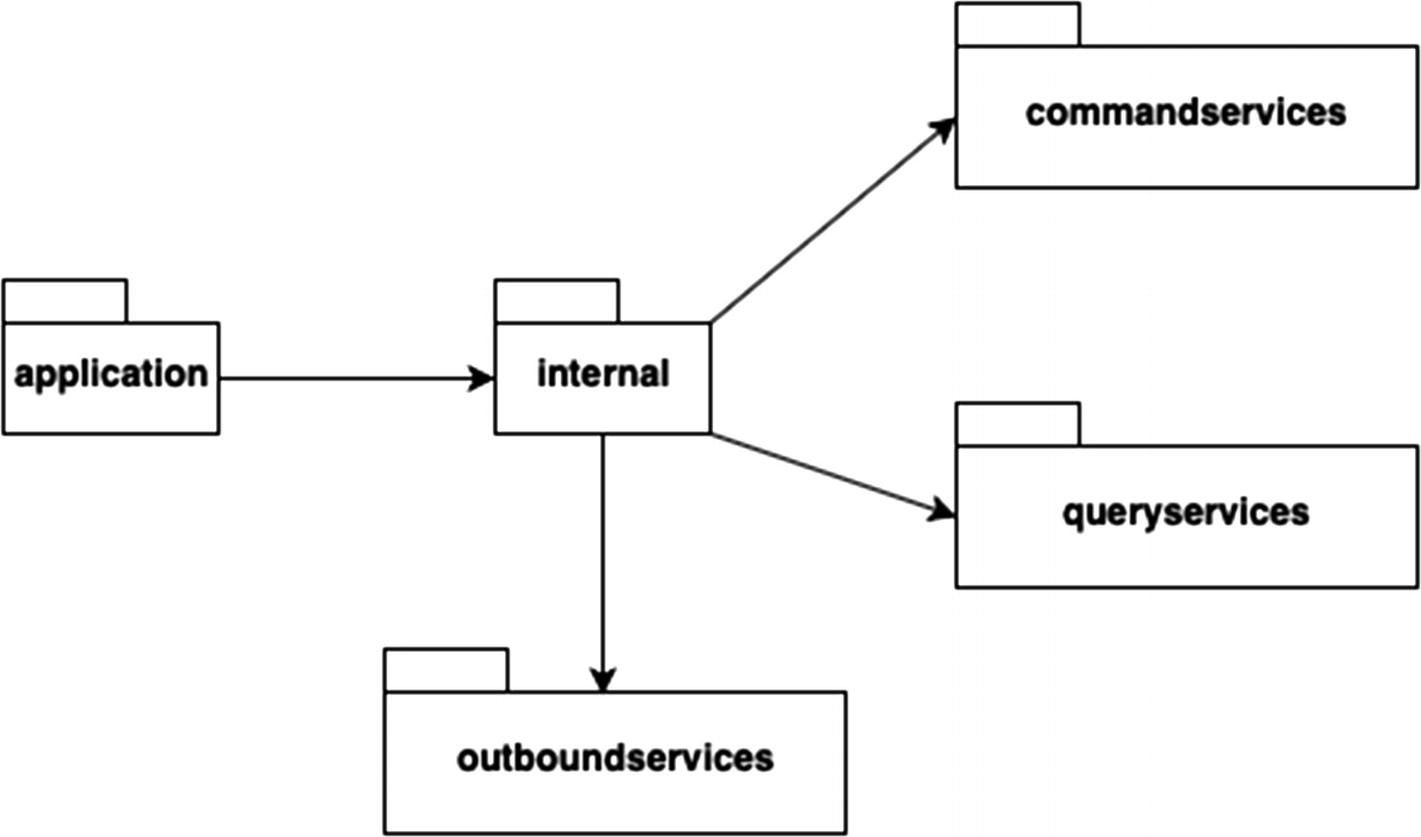


Figure 19 - Package structure for Application services

#### Domain

This package contains the Bounded Context’s Domain Model. This is the heart of the Bounded Context’s Domain Model which contains the implementation of the core Business Logic.

The core classes of our Bounded Contexts are as follows:

* Aggregates
* Entities
* Value Objects
* Commands
* Events

The package structure is illustrated in the following Figure:

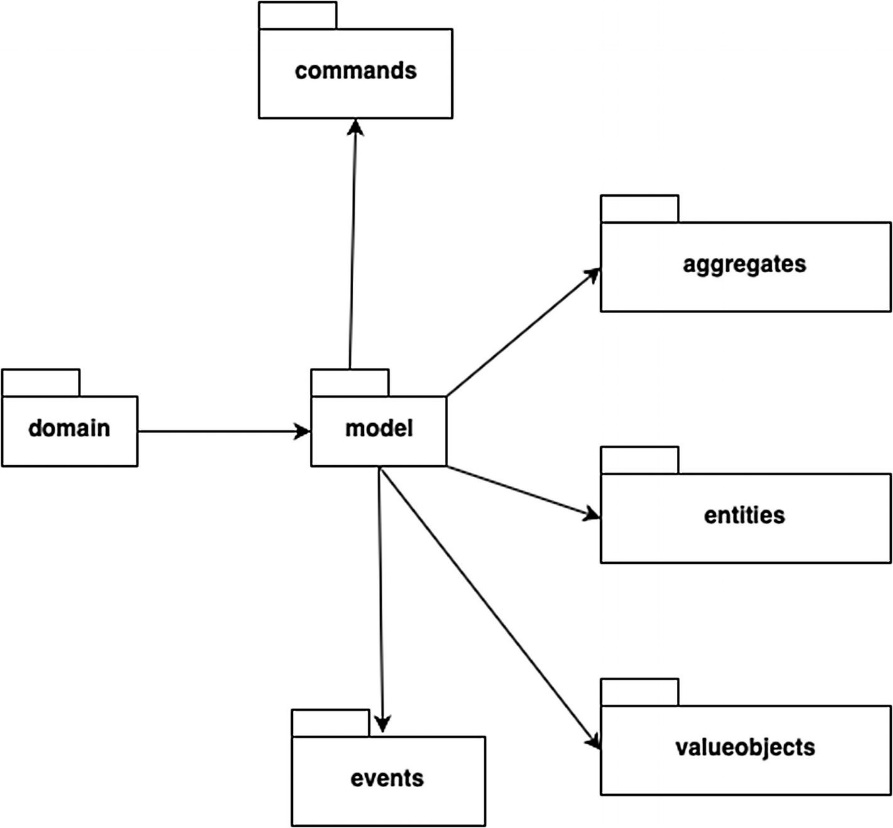


Figure 20 - Package structure for domain model

#### Infrastructure

The infrastructure package serves three main purposes:

* When a Bounded Context receives an operation related to its state (Change of State, Retrieval of State), it ***needs an underlying repository to process the operation***;. The infrastructure package contains all the necessary components required by the Bounded Context to communicate to the underlying repository. As part of our implementation, we intend to use Spring Data to implement these components.
* When a Bounded Context needs to communicate a state change event, it needs an underlying Event Infrastructure to publish the state change event. The infrastructure package contains all the necessary components required by the Bounded Context to communicate to the underlying message broker.
* The final aspect that we include in the infrastructural layer is any kind of Spring Boot-specific configuration.

The package structure is illustrated in the following Figure:

A close up of a logo

Description automatically generated

Figure 21 - Package structure for the infrastructure components

A complete summary of the entire package structure for any of our Bounded Context is illustrated in the following Figure

A close up of a map

Description automatically generated

Figure 22 - Package structure for any of our Bounded Context

This completes the implementation of the Bounded Contexts of our microservices application. Each of our Bounded Contexts is implemented as a Spring Boot application with a fat JAR as an artifact. The Bounded Contexts are neatly grouped by modules in a package structure with clearly separated concerns.

## Implementation

In This section we are going to detail the implementation of the Loans application as a microservices application utilizing DDD and Spring Boot/Spring Cloud.

A high-level overview of the logical grouping of our various DDD artifacts is illustrated in Figure As seen, we need to implement two groups of artifacts:

* The ***Domain Model*** which will contain our ***Core Domain/Business Logic***
* The ***Domain Model Services*** which contain ***supporting services for our Core Domain Model***

A screenshot of a cell phone

Description automatically generated

Figure 23 - Logical grouping of the DDD artifacts

In terms of actual implementation of the Domain Model, this translates to the various Commands, Queries, and Value Objects of a specific Bounded Context/Microservices.

In terms of actual implementation of the Domain Model Services, this translates to ***the Interfaces, Application Services, and Infrastructure*** that the Domain Model of the Bounded Context/Microservices requires.

The following figure illustrates our microservices solution in terms of the various Bounded Contexts and the operations it supports. As seen, this contains the various ***Commands that each Bounded Context will process***, the ***Queries that each Bounded Context will serve,*** and the ***Events that each Bounded Context will subscribe/publish***. Each of the microservices is a separate deployable artifact with its own storage.

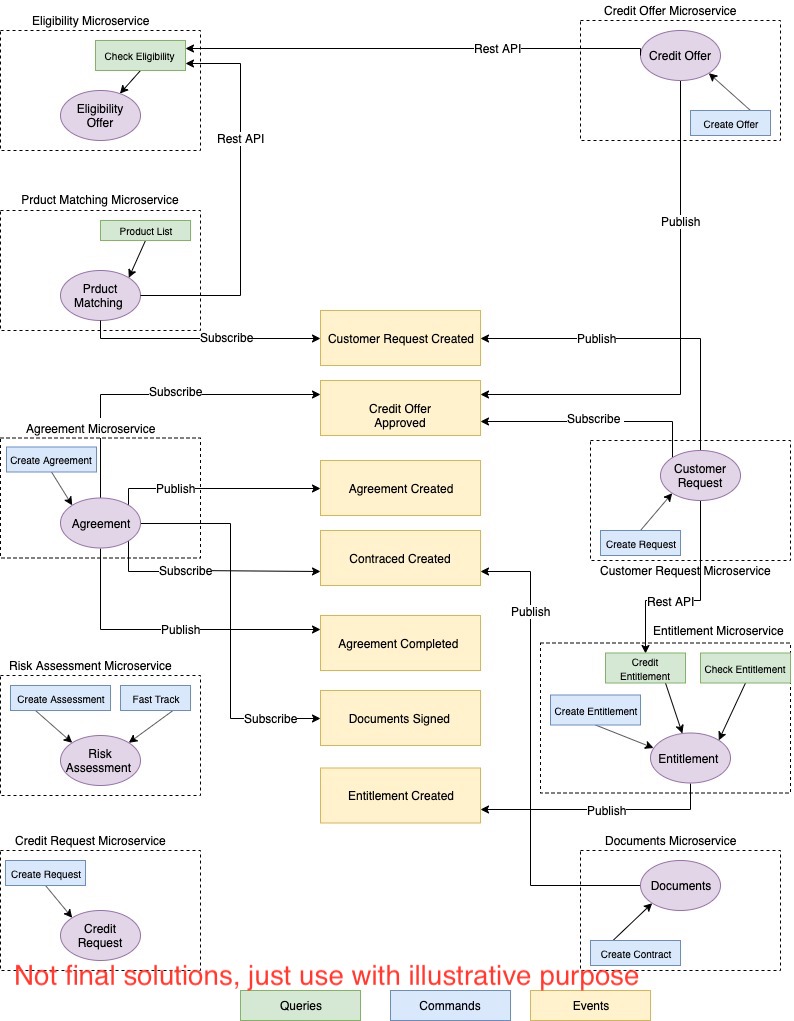


Figure 24 - Loan Microservices solution

### Domain Model: Implementation

Our Domain Model is the central feature of our Bounded Context and as stated earlier has a set of artifacts associated with it. Implementation of these artifacts is done with the help of the tools that Spring Boot provides.

### Core Domain Model: Implementation

The implementation of the Core Domain for any Bounded Context covers the identification of those artifacts that will express the business intent of the Bounded Context clearly. At a high level, this includes the identification and implementation of Aggregates, Entities, and Value Objects.

Implementation of an Aggregate covers the following aspects:

* Aggregate Class Implementation
* Domain Richness via Business Attributes and finally
* Implementing Entities/Value Objects

#### Aggregate Class Implementation

Since we intend to use MySQL[[3]](#footnote-3) as our Datastore for each of our Bounded Contexts, we intend to use JPA (Java Persistence API) from the Java EE specification which provides a standard way of defining and implementing Entities/Services which interact with underlying SQL Datastores.

#### JPA Integration: Spring Data JPA

Spring Boot provides support for JPA by using the Spring Data JPA project ( https://spring.io/projects/spring-data-jpa ) which provides a sophisticated and easy mechanism to implement JPA-based repositories. Spring Boot provides a starter project (spring-boot-starter-data-jpa) which automatically configures a set of sensible defaults (e.g., Hibernate JPA Implementation, Tomcat connection pooling) for Spring Data JPA.

The dependency for the starter data JPA project is automatically added when we configure it as a dependency within the Initializr project. In addition to that, we need to add the MySQL Java driver library to enable connectivity to our MySQL Database instances:



Figure 25 - pom.xml dependency maintainance

In addition to the dependencies, we also need to configure the connection properties for each of our MySQL Instances. This is done in the application.properties file provided by our Spring Boot application. The following picture demonstrates the configuration properties that need to be added. You would need to replace the values with your MySQL Instance(s) Details as necessary:



Figure 26 - MySQL connection configuration

These settings are enough to set up and implement JPA within our Spring Boot application. As stated before, the Spring Data JPA project configures a set of sensible defaults which enable us to get started with minimal effort. Unless stated otherwise, all our Aggregates within all our Bounded Contexts implement the same mechanism.

Each of our root aggregate classes is implemented as a JPA entity. There are no specific annotations that JPA provides to annotate a specific class as a root aggregate, so we take a regular POJO and use the JPA-provided standard annotation **@Entity**. Taking the Eligibility Assessment Bounded Context as an example which has Eligibility Assessment as the root Aggregate, the following figure shows the minimalistic code required for a JPA Entity:

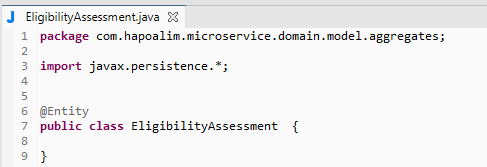


Figure 27 - EligibilityAssessment root aggregate as a JPA Entity

Every JPA Entity requires an identifier. For our Aggregate Identifier implementation, we choose to have a **technical/surrogate identifier (Primary Key)** for our Eligibility Assessment Aggregate derived from a MySQL sequence.

Continuing with our example of the Eligibility Assessment (request) within the Credit Eligibility Bounded Context, we add the Technical Key to the Class implementation until now.

The following picture demonstrates this. The **“@Id”** annotation identifies the primary key on our Eligibility Assessment Aggregate:

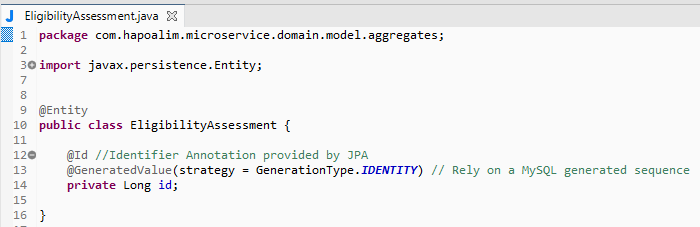


Figure 28 - Identifier for the EligibilityAssessment root aggregate

#### MongoDB Integration : Spring Data mongoDB

Spring Boot provides support for mongoDB by using the Spring Data mongoDB project ( https://spring.io/projects/spring-data-mongodb ) which provides a sophisticated and easy mechanism to implement mongoDB document database. Spring Boot provides a starter project (spring-boot-starter-data-mongodb) which automatically configures a set of sensible defaults.

The dependency for the starter data mongoDB project is automatically added when we configure it as a dependency within the Initializr project.

A screenshot of a social media post

Description automatically generated

Figure 30 - pom.xml mongoDB dependency maintainance

Each of our root aggregate classes is implemented as a document. Taking the Eligibility Assessment Bounded Context as an example which has Eligibility Assessment as the root Aggregate, the following figure shows the minimalistic code required for a document:

A screenshot of a cell phone

Description automatically generated

Figure 31- EligibilityAssessment root aggregate as a mongoDB Document

Every document requires an identifier. For our Aggregate Identifier implementation, we choose to have a **technical/surrogate identifier** for our Eligibility Assessment Aggregate.

The Business key conveys the business intent of the aggregate identifier clear, that is, Eligibility Assessment Identifier of a newly eligibility assessment, and is the key that is exposed to external consumers of the Domain Model. The technical key on the other hand is a pure internal representation of the aggregate identifier and is useful to maintain relationships **within a Bounded Context** between the Aggregates and its Dependent Objects.

The following picture demonstrates this. The **“@Id”** annotation identifies the Idetification on our Eligibility Assessment Aggregate:

A screenshot of a social media post

Description automatically generated

Figure 32- Identifier for the EligibilityAssessment root aggregate

### Domain Richness: Business Attributes

With the bare-bones implementation ready, let us move onto the meat of the Aggregate – Domain richness. ***The Aggregate of any Bounded Context should be able to express the Business Language of the Bounded Context clearly***. Essentially, what it means in pure technical terms is that our Aggregate should not be anemic, that is, only containing getter/setter methods.

An ***anemic aggregate*** goes against the fundamental principle of DDD since it essentially would mean ***the Business Language being expressed in multiple layers of an application*** which in turn leads to an unmaintainable piece of software in the long run.

So how do we implement a Domain-Rich Aggregate? The short answer is ***Business Attributes and Business Methods.*** Our focus in this section is going to be on the Business Attributes aspect while we will cover the Business Methods part as part of the Domain Model Operations implementation.

***Business Attributes of an Aggregate capture the state of an Aggregate as attributes depicted using Business Terms rather than Technical Terms.***

Let’s walk through the example of our Eligibility Assessment (request) aggregate.

Translating state to business concepts, the Eligibility Assessment (request) Aggregate has the following attributes:

* **Account** of the customer the Eligibility Assessment Request
* **Customer** who is the principal actor for the Eligibility Assessment Request
* **Product** for the Eligibility Assessment Request is made

The following picture show the Eligibility Assessment (request) and its relationships with its dependent objects[[4]](#footnote-4).

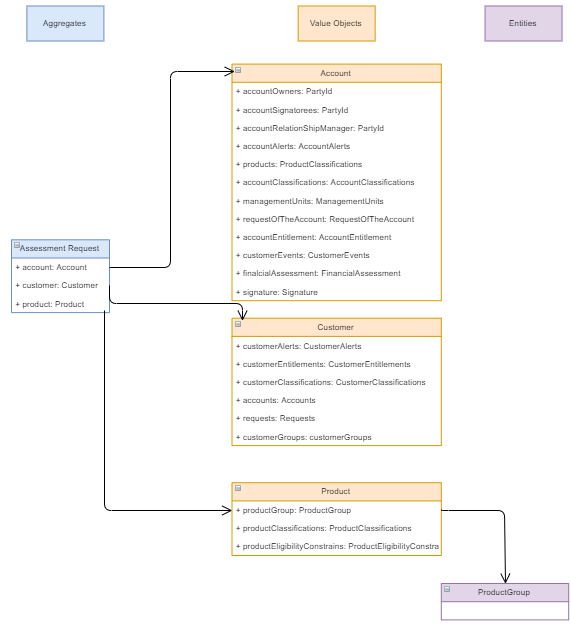


Figure 31 - Eligibility Assessment Request Aggregate and its dependent associations

JPA provides us a set of annotations (**@Embedded, @Embeddable**) to help implement our Aggregate class using Business Objects.

The following figure shows the example of our Eligibility Assessment (Request) Aggregate with all the Dependencies modeled as Business Objects:



Figure 32 - EligibilityAssessment root aggregate dependencies as business objects

Dependent classes for an Aggregate are ***modeled either as Entity Objects or Value Objects***. To recap, Entity Objects within a Bounded Context have an identity of their own but always exist within a root aggregate, that is, they cannot exist independently, and they never change during the complete lifecycle of the aggregate. Value Objects on the other hand have no identity of their own and are easily replaceable in any instance of an aggregate.

They are ***all replaceable and hence modeled*** ***as Value Objects.*** That is the ***thumb rule for modeling Entities and Value Objects within an Aggregate.***

### Implementing Entity Objects / Value Objects

Entity Objects/Value Objects are implemented as JPA Embeddable objects using the ***“@Embeddable”*** annotation provided by JPA. They are then embedded into the Aggregate using the “***@Embedded***” annotation.

The next figure shows the mechanism of embedding into the Aggregate class.

Let’s look at the implementation of the Eligibility Assessment (Request)’s Entity Objects/Value Objects.

The figure demonstrates the ***Product Group Entity Object*** . Notice the package name (***grouped under model.entities***):

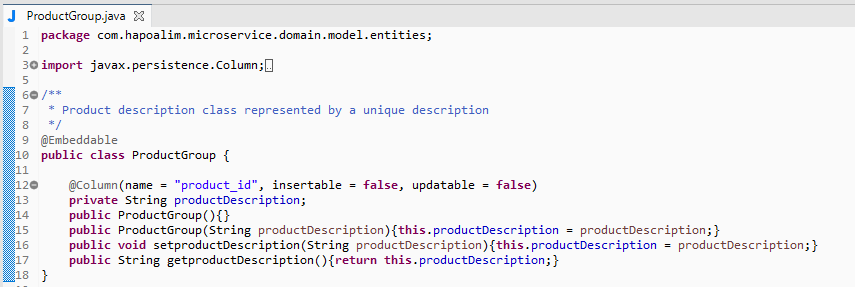


Figure 33 - ProductGroup entity Object

In the following figure examples of the Eligibility Assessment / Account value object(s). Notice the package name (***groupmed under model valueobjects***) and also notice that it is a sample, it is not the final implementation. We are only showing the way to model and build the class strucutre:

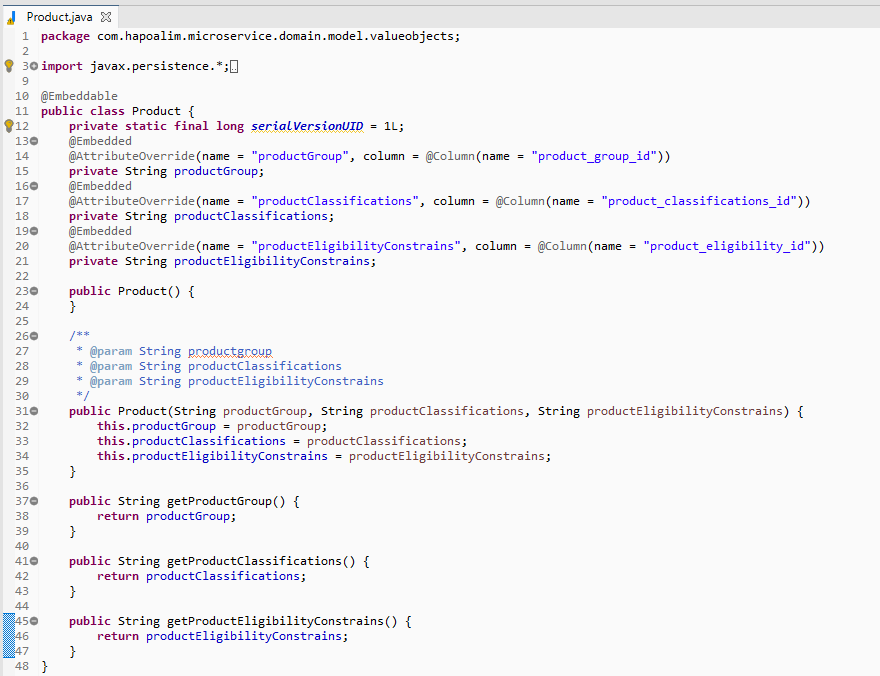


Figure 34 – Product value object implementation

### Domain Model Operations

Domain Model operations within a Bounded Context deal with any kind of operations associated with the state of the Aggregate of the Bounded Context. These include ***inbound operations (Commands/Queries) and outbound operations (Events).***

### Commands

Commands are responsible for changing the state of the Aggregate within a Bounded Context.

Implementation of Commands within a Bounded Context involves the following steps:

* Identification/implementation of Commands
* Identification/implementation of Command Handlers to process Commands

*Identification of Commands*

Identification of Commands revolves around identifying any operation that affects the state of the Aggregate. For example, the Eligibility Assessment Command Bounded Context has the following operations or commands:

* Create a new assessment
* Delete an assessment

Both these operations result in a change of state of the Eligibility Assessment Aggregate within the Bounded Context and are hence identified as Commands.

*Implementation of Commands*

Once identified, implementing the identified Commands within the Spring Boot implementation is done using regular POJOs. In the next figure we are showing the implementation of the EligibilityAssessmentCommand class for the Eligibility AssessmentBook Command:



Figure 35 - Eligibility Assessment Command class implementation

*Identification of Command Handlers*

Every Command will have a corresponding Command Handler. The purpose of the Command Handler is ***to process the input command and set the state of the Aggregate***. Command Handlers are ***the only place within the Domain Model where Aggregate state is set***. This is a strict rule that needs to be followed to help implement a rich Domain Model.

Implementation of Command Handlers

Since the Spring Framework does not provide any out-of-the box capabilities to implement Command Handlers, our methodology of implementation will be to just ***identify the routines on the Aggregates which can be denoted as Command Handlers***. For our first command ***Create New Assessement,*** we identify the constructor of the Aggregate as our Command Handler:

The following figure shows the snippet of code of the constructor of the Eligibility Assessement (Request) Aggregate. The constructor accepts the ***EligibilityAssessmentCommand*** as an input parameter and sets the corresponding state of the Aggregate:

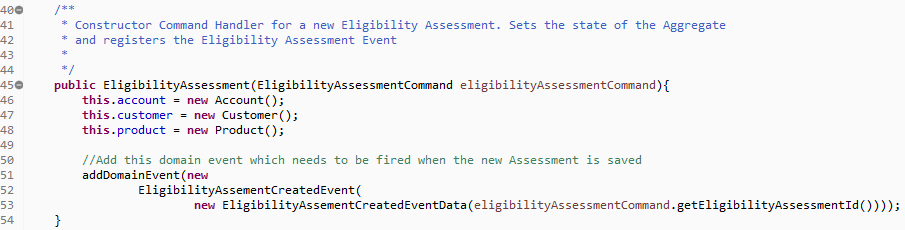


Figure 36 - Command handler for the EligibilityAssessment command

In summary, **Command Handlers play a very important role of managing the Aggregate state within a Bounded Context**. The actual invocation of Command Handlers happens via Application Services which we shall see in the sections that follow.

Next picture will illustrate the class diagram for our Command Handler implementation.

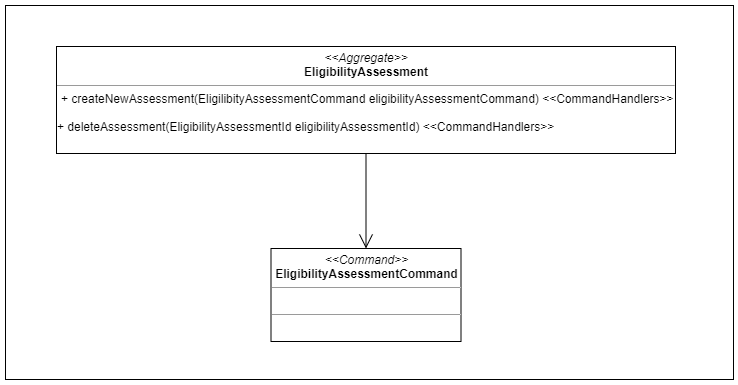


Figure 37 - Class diagram for the Command Handler implementation

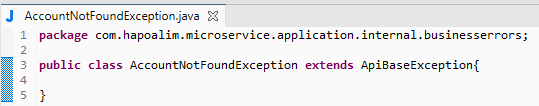
Implementation of Business Errors

All the business errors should be encapsulated at domain level. There is a strick rule to not allow the generation business errors outside the command scope (i.e. create business errors in API Layer). To implement this task we need to be attached to the following rules:

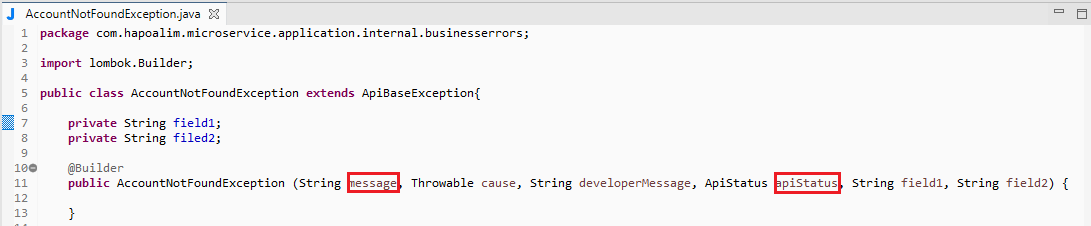
* 1. Our Spring Boot Application must add the dependencies of BHP Starter for error handling. TODO include a screen shot BHP Starter
  2. The rules for error handling in BHP must to be followed. For further information, please go to the following reference: [[7]](#_References_1) **Error Handling Library – Key components**
  3. For every business error we have, we need to create an Exception object and place it in the package: com.BankXXXX.microservice.application.internal.businesserrors



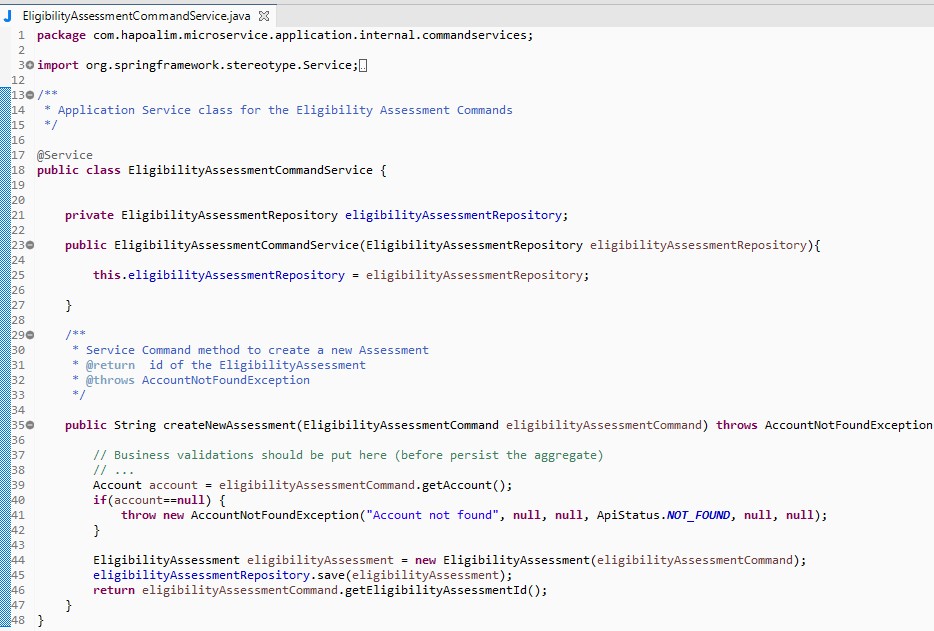
* 1. This Exception should be extends from ApiBaseException.



* 1. If we want to provide more details, Lombok’s builder can be used. The only required fields to full fill are:
     + message: description of the business error message.
     + apiStatus: code (error code, http status code) to define and classify the problem happened for further processing.



* 1. Once the exeception (business error) has been defined, this exception must be handled in the command scope and propagated to the upper lever:



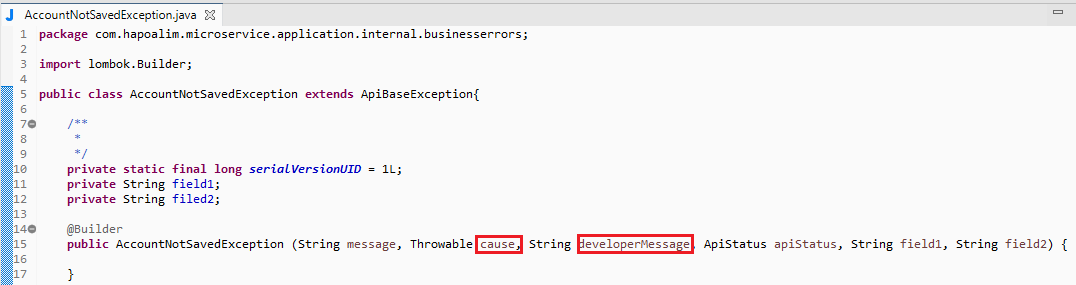
* 1. Finally, the exception must the be handled at API layer (RestController level):



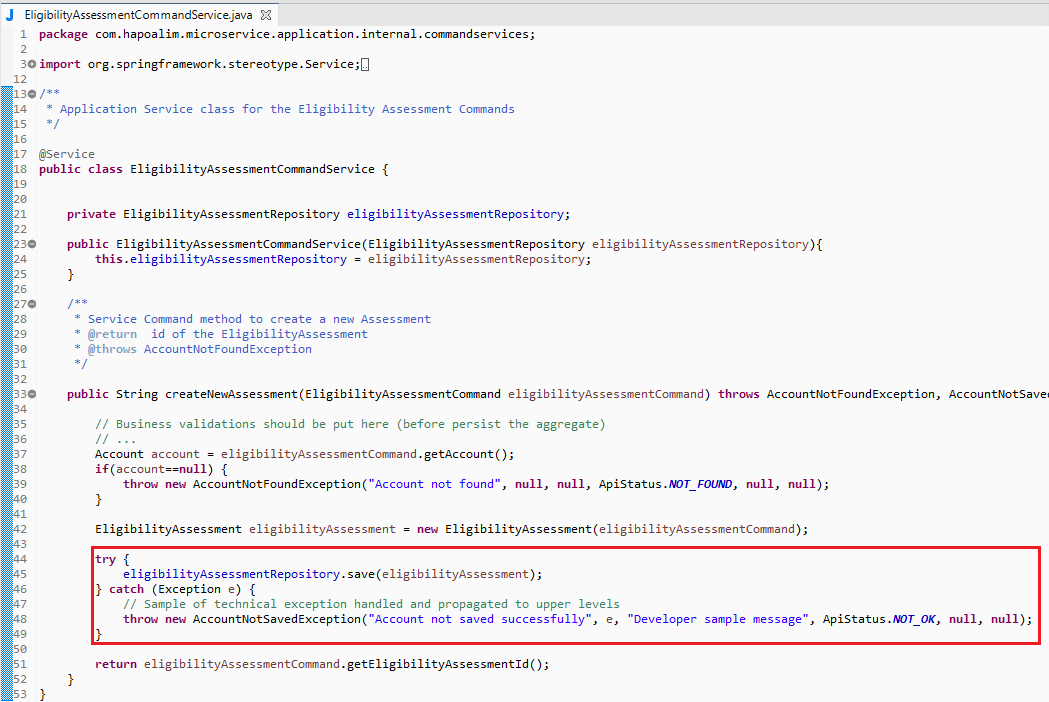
Implementation of Technical Errors

All the guidelines provided above, regarding the business errors, are applicable to handle technical errors. But we there are some changes that we would like to highlight:

1. We will be relying on the exception hierarchy of Java. If there is the need of creating our own technical error, that error should be encapsulated creating a custom exception for this. That technical error will be treated as an business error, so all the recommendations and policies describe above are the same for this scenario.
2. The custom technical errors (if any) must be placed in the following package: com.BankXXXX.microservice.application.internal.businesserrors
3. In this case should be interesting provide more information to help a deep analisis in further actions.
4. So, when we create the custom technical exepction it is mandatory put information in the parameters: “cause” and “developerMessage”.



1. In the same way than before for business errors, the technical exception must be handled in right scope and then propagated to the upper lever:



1. And finally, the exception must the be handled at API layer (RestController level):



In the next figure, we are showing the class diagram for Error Handling implementation inside commands:

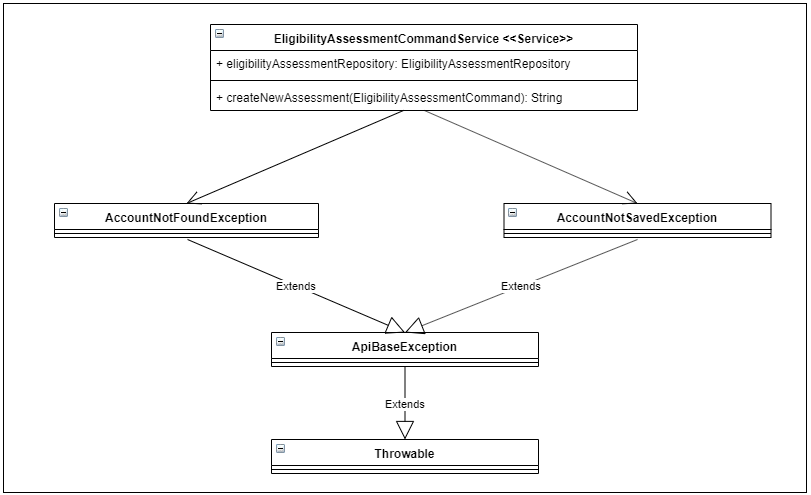


Figure 38 - Class diagram for the Error Handling Implementation

### Queries

Queries within the Bounded Context are responsible for ***providing the state of the Bounded Context’s Aggregate*** to external consumers.

To implement Queries, we utilize ***JPA Named Queries,*** that is, queries that can be defined on an Aggregate to retrieve state in various forms.

The next figure demonstrates the snippet of code from the Eligibility Assessment Aggregate that defines the queries that need to be made available. In this case, we have two queries – ***Find All Eligibility Assessment, Find a Eligibility Assessment by its Eligibility Assessment Identifier:***

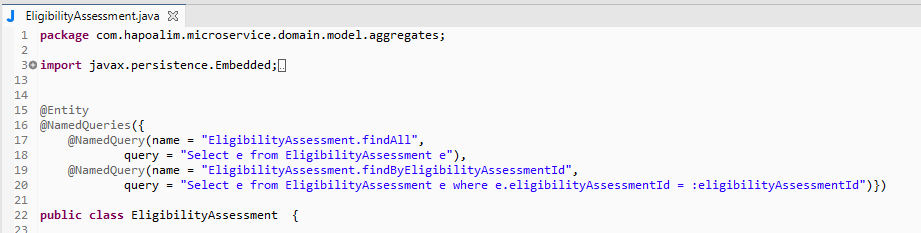
******

Figure 39 - Named queries within the Eligibility Assessment root aggregate

In summary, Query Handlers play the role of presenting the Aggregate state within a Bounded Context. The actual invocation and execution of these queries happens via Application Services and Repository classes.

### Events

An event within a Bounded Context is any operation that ***publishes the Bounded Context’s Aggregate State Changes as Events***. Since Commands change the state of an Aggregate, it is safe to assume that any Command operation within a Bounded Context will result in a corresponding Event.

Domain Events play a central role within a microservices architecture, and it is critical to implement them in a robust manner. The distributed nature of a microservices architecture mandates the usage of Events via a ***choreography mechanism*** to ***maintain state and transactional consistency*** between the various Bounded Contexts of a microservices-based application.

The following figure examples of the events that flow between the various Bounded Contexts of the Loan Application. Let’s keep in mind that this picture is not the final picture, it is just the photo that we have at the moment of writing this document.

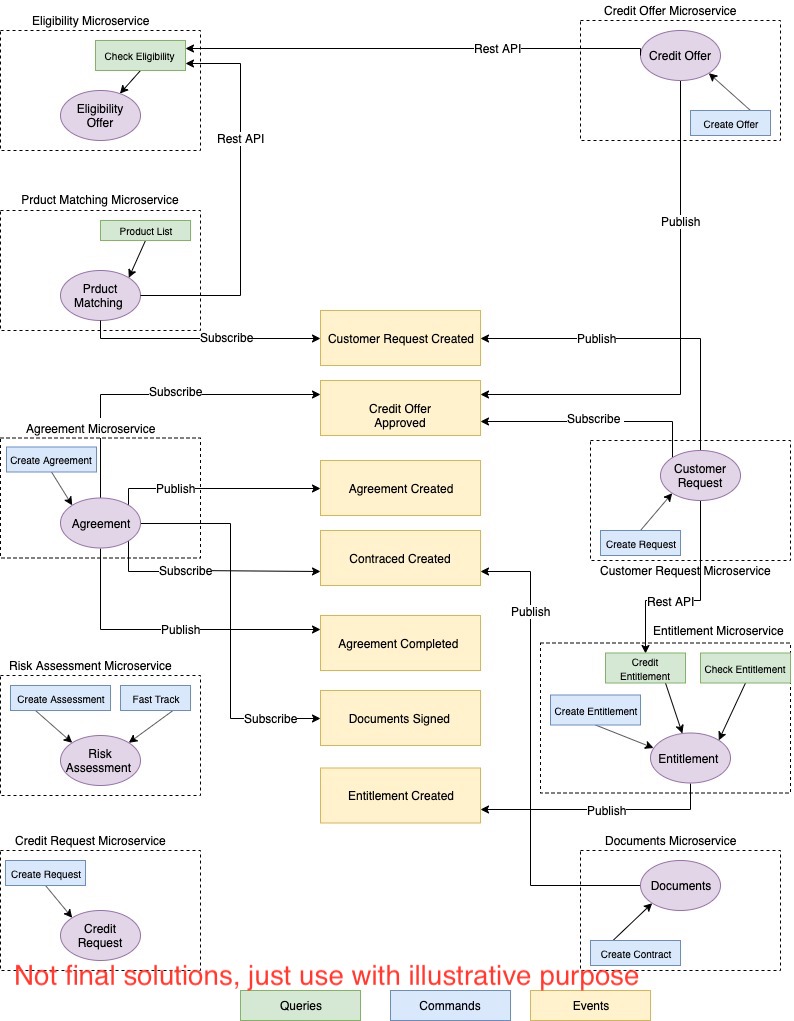


Figure 40 - Flow of Events in a Microservices architectureThere are four stages to the implementation of a robust event-driven choreography architecture :

* ***Register***  the Domain Events that need to be raised from a Bounded Context.
* ***Raise*** the Domain Events that need to be published from a Bounded Context.
* ***Publish*** the Events that are raised from a Bounded Context.
* ***Subscribe*** to the Events that have been published from other Bounded Contexts.

Considering the complexity of this architecture, the implementation is split across multiple areas:

* Registration of Domain Events is implemented ***by the Aggregate.***
* Raising/publishing of Events is implemented ***by the Outbound Services.***
* Subscribing to Events is handled ***by the Interface/Inbound services.***

The only area that we will cover in this section, since we are in the phase of implementing the Domain Model, is the registration of events by the Aggregate. The subsequent sections of the chapter will deal with each of the other aspects (Outbound services will cover the implementation of the raising/publishing of Events, and Inbound Services will cover the implementation of subscribing to the Events).

### Registration of Events

To help implement this, we will utilize the template class “***AbstractAggregateRoot***” provided by Spring Data. This template class provides the capability to register events that occur.

Let us take an example to walk us through the implementation. Listing [5-15](https://learning.oreilly.com/library/view/practical-domain-driven-design/9781484245439/html/473795_1_En_5_Chapter.xhtml#PC16) shows the Eligibility Assessment Aggregate class which extends the AbstractAggregateRoot Template class:



Figure 41 - AbstracAggregateRoot template class

The next step is to ***implement the registered Aggregate events*** whenever the ***state of the Aggregate changes***. As we have stated and seen earlier***, Command Operations on Aggregates change state*** and are ***the most likely place where we would like to register Aggregate Events***. Within the Eligibility Assessment Aggregate, we have one Command Operation: when a new eligibility assessment is created. We will implement the registration and raising of the Aggregate Events within these method using the ***registerEvent()*** method provided by the AbstractAggregateRoot template class.

In the next figure we are showing the sample implementation of the Registration of Aggregate Events within the ***Command Handler methods*** of the Eligibility Assessment Aggregate. We add a new method within the Aggregate “***addDomainEvent()***” which is an encapsulation of the “***registerEvent()”***. It takes as an input parameter a ***Generic Event Object which is the event that needs to be registered:***

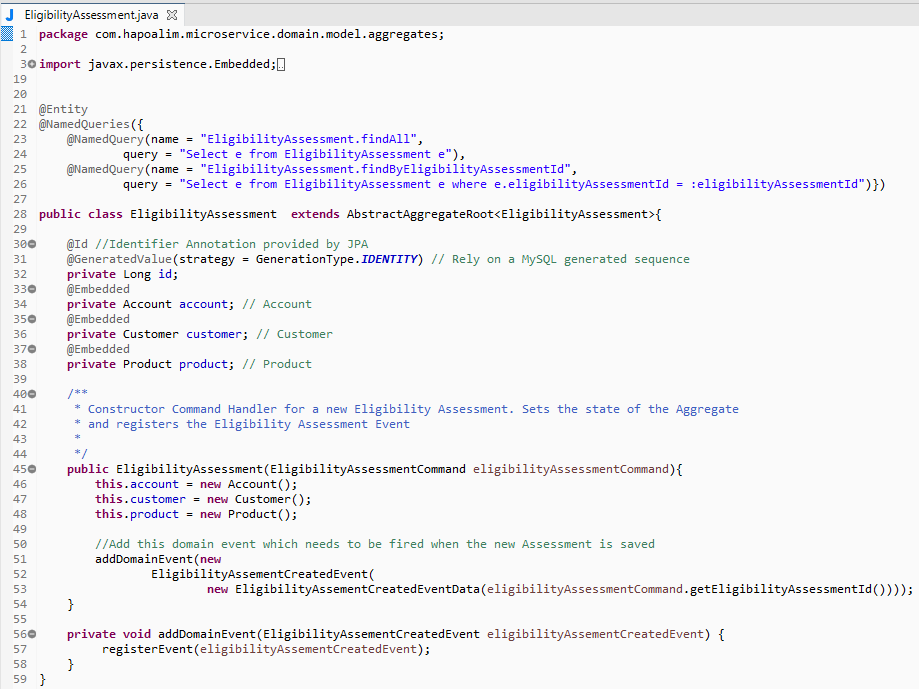


Figure 42 - Event registration within the Eligibility Assessment root aggregate

Next figure shows the implementation of the ***EligibilityAssessmentCreatedEvent*** class. It is a regular POJO which encapsulates the Event Data, that is, ***EligibilityAssesmentCreatedEventData:***

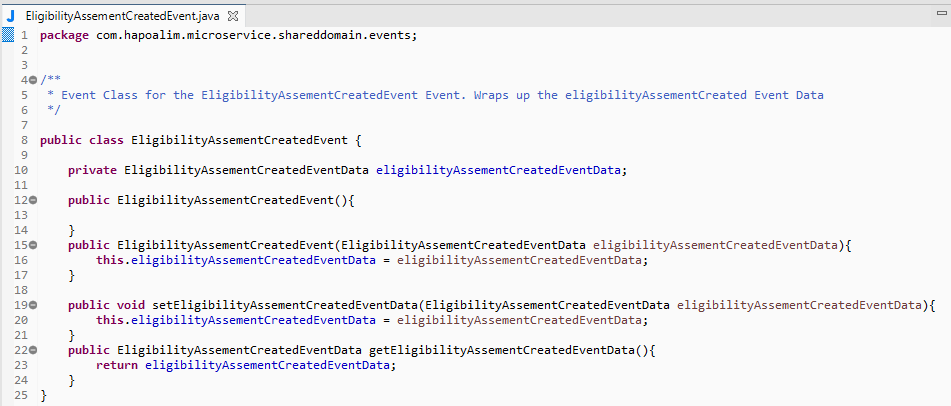


Figure 43 - Eligibility Assessment Created Event implementation class

Next figure shows the implementation of the EligibilityAssessment class. This is again a regular POJO and contains the Event Data, in this case just the Eligibility Assessment Id:

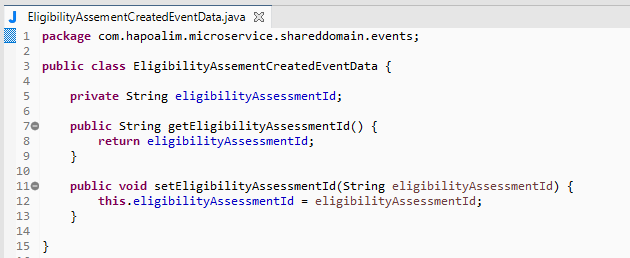


Figure 44 - EligibilityAssessmentCreatedEventData implementation class

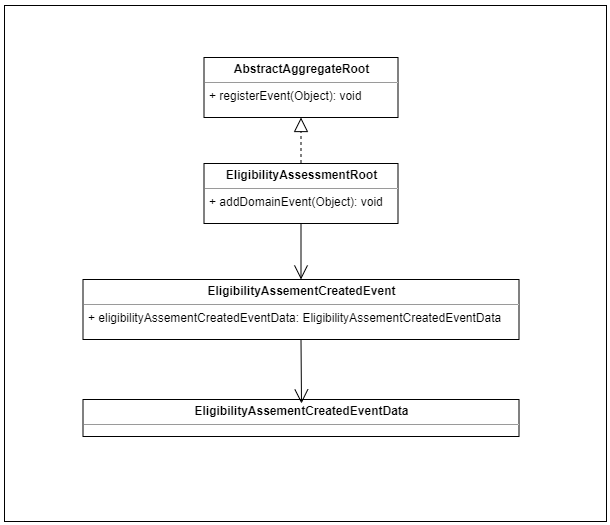


Figure 45 - Class diagram for the Aggregate Event Registration implementation

***In summary, Aggregates register Domain Events after the processing of a Command. The registration of these events is always implemented within the Command Handler methods of the Aggregates.***

## Domain Model Services

Domain Model Services are used for two primary reasons. The first is to enable the Bounded Context’s state to be ***made available to external parties*** through ***well-defined Interfaces.*** The second is ***interacting with external parties*** be it to persist the Bounded Context’s state to ***Datastores*** (Databases), publish the Bounded Context’s state change events to external ***Message Brokers,*** or to ***communicate with other Bounded Contexts.***

There are three types of Domain Model Services for any Bounded Context:

* ***Inbound Services*** where we implement well-defined interfaces which enable external parties to interact with the Domain Model
* ***Outbound Services*** where we implement all interactions with External Repositories/other Bounded Contexts
* ***Application Services*** which act as the façade layer between the Domain Model and both Inbound and Outbound services

The next figure illustrates the Domain Model Services implementation:

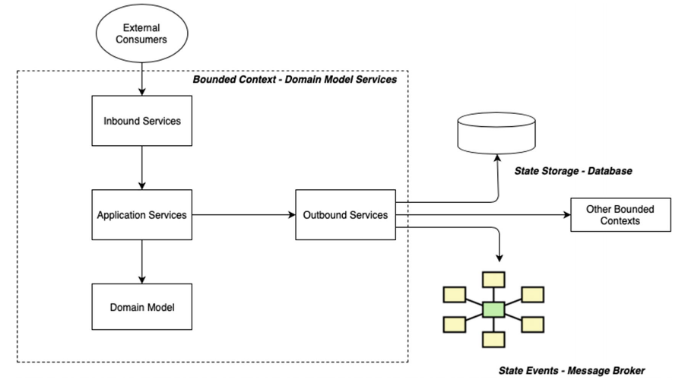


Figure 46 - Domain Model Services implementation summary

### Inbound Services

Inbound services (or Inbound Adaptors as denoted in the Hexagonal Architectural Pattern) act as the outermost gateway for our Core Domain Model. As stated, it involves the implementation of well-defined interfaces which enable external consumers to interact with the core domain model.

The type of inbound services ***depends upon the types of operations*** we need to expose to ***enable the external consumers of the Domain Model.***

Considering that we are implementing the microservices architectural pattern for our Loan Application, we provide two types of Inbound Services:

* ***An API Layer based on REST*** which is used by external consumers to invoke operations on the Bounded Context (***Commands/Queries***)
* ***An Event Handling Layer based on Spring Cloud Stream*** which consumes Events from the Message Broker and processes them.

#### REST API

The responsibility of the REST API is to receive HTTP requests on behalf of the Bounded Context from external consumers. This request could be for Commands or Queries. The responsibility of the REST API layer is to translate it into the Command/Query Model recognized by the Bounded Context’s Domain Model and delegate it to the Application Services Layer to further process it.

Implementation of the REST API in Spring Boot is by utilizing the REST capabilities provided by the Spring Web MVC Project. The ***spring-boot-starter-web*** dependency that we added to our project provides the required capabilities to build the API(s).

Let us walk through an example of a REST API built using Spring Web. The next figure depicts the ***EligibilityAssessmentController class*** which provides a REST API for our ***Eligibility Assesment Command***:

* The REST API is available at the URL “/***eligibilityassessment***”.
* It has a single POST method that accepts a EligibilityAssessmentResource which is the input payload to the API. This is marked with the annotation “***@RequestBody***”.
* It has a dependency on the EligibilityAssessmentCommandService which is an Application services which acts as a façade (see implementation in the following). This dependency is injected into the API class utilizing a Constructor-based Dependency Injection.
* It transforms the Resource Data (***EligibilityAssessmentResource***) to the Command Model (***EligibilityAssessmentCommand***) using an Assembler utility class (***EligibilityAssessmentCommandDTOAssembler***).
* After transforming, it delegates the process to the EligibilityAssessmentCommandService for further processing.
* It returns back a Response to the external consumer with the EligibilityAssessment Identifier of the newly created eligibility assessment.



Figure 47 - EligibilityAssessmentController implementation class



Figure 48 - EligibilityAssessmentResource implementation class



Figure 49 – EligibilityAssessmentCommandDTOAssembler implementation class



Figure 50 – EligibilityAssessmentCommand implementation class



Figure 51 - Class diagram for the REST API implementation

All our inbound REST API implementations follow the same approach which is illustrated in the next figure:

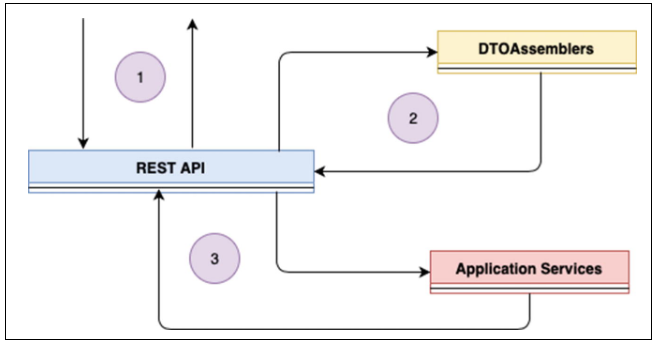


Figure 52 - Inbound Services implementation process summary

|  |  |
| --- | --- |
|  | The inbound request for a Command/Query comes to the REST API. API classes are implemented using the Spring Web MVC project which gets configured when we add the ***spring-boot-starter-web*** dependency to the project. |
|  | The REST API class uses a utility Assembler component to convert the Resource Data format to the Command/Query Data format required by the Domain Model. |
|  | The Command/Query Data is sent to the Application Services for further processing. |

#### Event Handlers

The other type of interfaces that exist within our Bounded Contexts are the Event Handlers. Within a Bounded Context, Event Handlers are responsible for processing Events that the Bounded Context is interested in. These Events are raised by other Bounded Contexts within the application. These “***EventHandlers*** ” are created within the subscribing Bounded Context which resides within the ***inbound/interface*** layer. The Event Handlers receive the Event along with the Event payload data and process them as a regular Command operation.

Implementation of the Event Handlers will be done utilizing the capabilities provided by Spring Cloud Stream. In order to implement some real implementation, a message broker will be selected (i.e. RabbitMQ, ActiveMQ, ZeroMQ,…) **🡪 TODO: include Sergey technical framework design**

No matter the message broker selected, but all our Event Handler implementations follow the same approach as illustrated in the next figure:

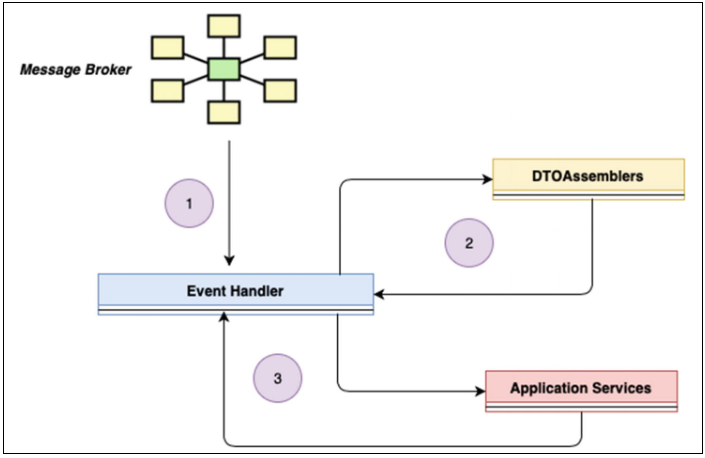


Figure 53 - Implementation process summary for Event Handler implementations

|  |  |
| --- | --- |
|  | Event Handlers receive inbound events from a Message Broker. |
|  | Event Handlers use a utility Assembler component to convert the Resource Data format to the Command Data format required by the Domain Model. |
|  | The Command Data is sent to the Application Services for further processing. |

### Application Services

Application Services act as a façade or a port between the Inbound/Outbound Services and the Core Domain Model within a Bounded Context.

Within a Bounded Context, Application services are responsible for ***receiving requests from the Inbound Services*** and ***delegating them to the corresponding services,*** that is, Commands are delegated to ***Command Services*** and Queries are delegated to ***Query Services.*** As part of a ***command delegation process***, Application services are responsible for persisting the Aggregate state in the underlying datastore. As part of a query delegation process, Application services are responsible for retrieving the Aggregate state from the underlying datastore.

As part of these responsibilities, Application services rely on ***outbound services*** to complete these tasks. Outbound services provide the necessary infrastructural components required to connect to the physical datastores. We will deep dive into the outbound services implementation separately (***see section on Outbound Services***).

The next figure illustrates the responsibilities of the Application Services:

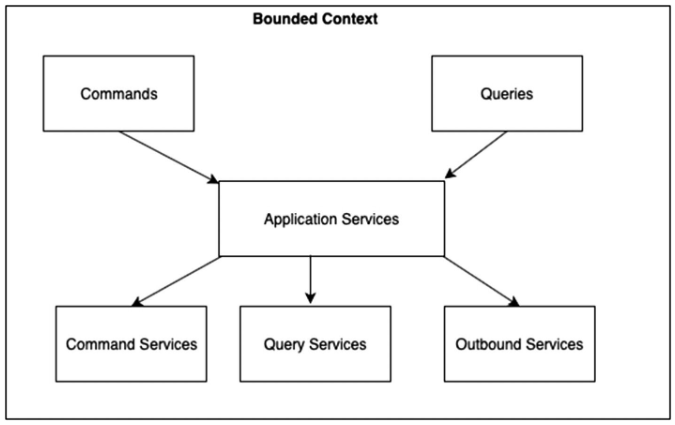


Figure 54 - Responsibilities of the Application Services

#### *Application Services: Command/Query Delegation*

As part of this responsibility, Application services within a Bounded Context receive requests for processing Commands/Queries. These requests come in typically from the Inbound Services (API Layer). As part of the processing, Application services first utilize the ***CommandHandlers/QueryHandlers*** (see section on Domain Model) of the Domain model to set state or query state. They then utilize the Outbound Services to persist state or execute queries on the state of the Aggregate.

Let us first walk through an example of a Command Delegator Application Services Class, the ***Eligibility Assessment Command Application Services Class***. This class has one routine - “***createNewAssessment()***” which handle the ***Eligibility Assessment Command:***

* The Application services class is implemented as a regular Spring Managed Bean with an “***@Service”*** marker annotation attached to it which indicates that it is a Service class.
* The Application services class is provided with the necessary dependencies via the Constructor Dependency Injection capabilities of Spring. In this case, the EligibilityBookingCommandApplicationService class has dependencies on an ***outbound repository class*** (***EligibilityAssessmentRepository***).
* The Application services relies on the Command Handler defined on the Eligibility Aggregate to set its state.
* The Application services utilizes the EligibilityRepository outbound service to store the state of the Eligibility Assessment in the operation.

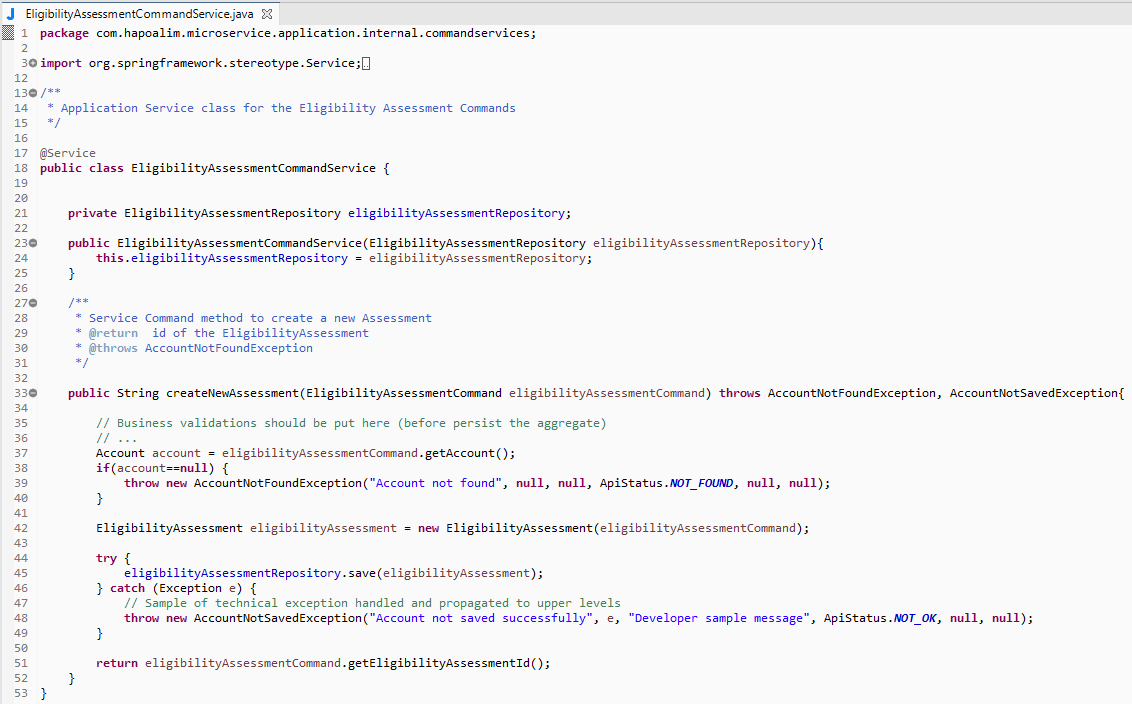


Figure 55 - EligibilityAssessmentCommand Application services class implementation



Figure 56 – EligibilityAssessmentQuery Application services implementation

Now, let’s illustrate the class diagram for our implementation in the next figure:

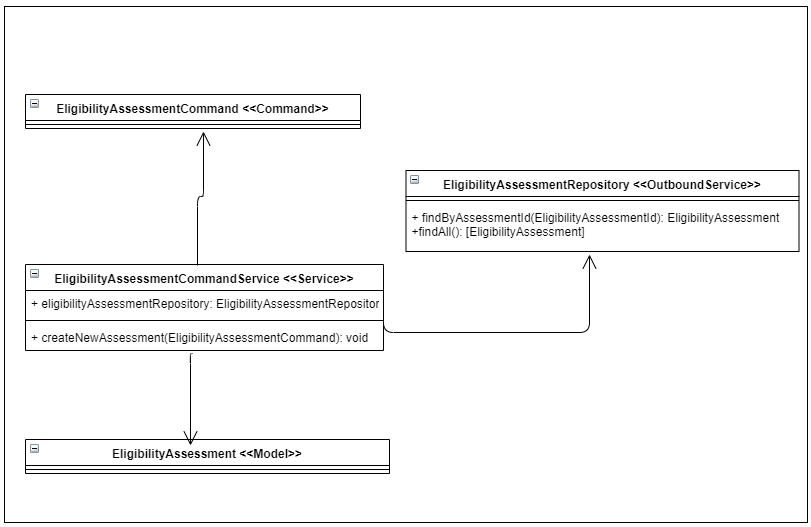


Figure 57 - Class diagram for our Application Services Command / Query delegation

All our Application Services implementations which are responsible for Command/Query delegations follow the same approach which is illustrated in the next figure:

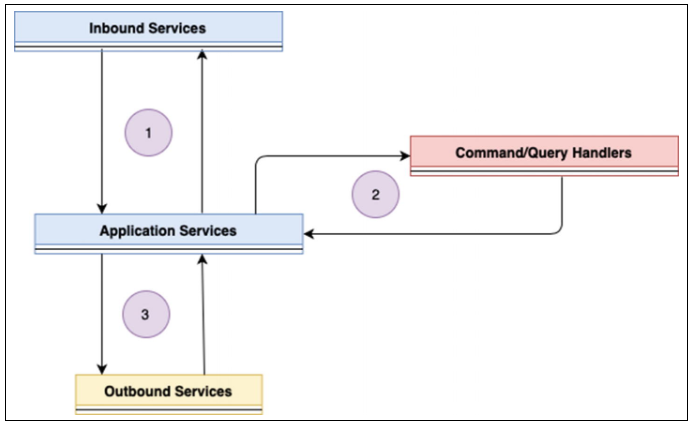


Figure 58 - Application Services implementation process summary

|  |  |
| --- | --- |
|  | The request for a Command/Query operation comes to the Application Services of a Bounded Context typically from the Inbound Services layer. Application Services Classes are implemented as Spring Managed Beans with the @Service marker annotation, and they have all their dependencies injected via the Constructor. |
|  | Application Services rely on CommandHandlers/QueryHandlers defined within the Domain Model to set/query Aggregate state. |
|  | Application Services utilize Outbound Services (e.g., Repositories) to persist the state of the Aggregate or execute the query on the Aggregate. |

#### *Outbound Services*

As we have seen in the Application Services implementation earlier, during the processing of a Command/Query, Application services might be required to communicate with ***external services*** such as the following:

* ***Repositories*** to store/retrieve state of the Bounded Context
* ***Message Brokers*** to communicate state change of the Bounded Context
* ***Other Bounded Contexts***

Application Services rely on ***Outbound Services*** to help in this communication.

Outbound Services provide capabilities to interact with ***these external services***. The ***external service could be the Datastore*** where we store the Bounded Context’s Aggregate State, it could be the ***message broker where we publish the Aggregate state,*** or it could be an ***interaction with another Bounded Context.***

The following figure illustrates the responsibilities of the Outbound Services. They receive requests to communicate with the external services as part of an operation (Commands, Queries, Events). They use APIs (Persistence APIs, REST APIs, Broker APIs) based on the external service type to interact with them.

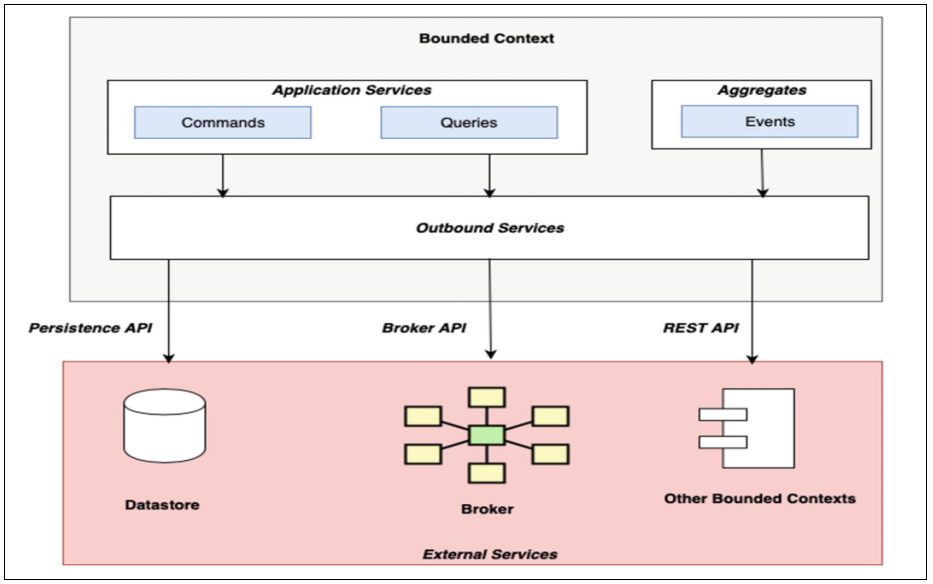


Figure 59 - Outbound Services

##### Outbound Services: Repository Classes

The outbound services for Database access are implemented as ***“*** ***Repository*** ***” classes.*** A repository class is built around a specific aggregate and deals with all database operations for that aggregate including the following:

* Persistence of a new aggregate and its associations
* Update of an aggregate and its associations
* Querying the aggregate and its associations

Spring Data JPA helps us implement JPA repository classes with ease.

Let us walk through an example of a Repository class, the ***Eligibility Assessment Repository Class,*** which handles all Database operations related to the ***Eligibility Assessment Aggregate:***

* The Eligibility Assessment Repository is implemented as an interface extending the JpaRepository<T,ID> interface.
* Spring Data JPA automatically implements the default CRUD operations required for the Eligibility Assessment Aggregate.
* We just add the methods required for any kind of custom queries which are mapped to the corresponding named queries defined within the Eligibility Assessment Aggregate.

The next picture demonstrates the implementation of the Eligibility Assessment Repository class:

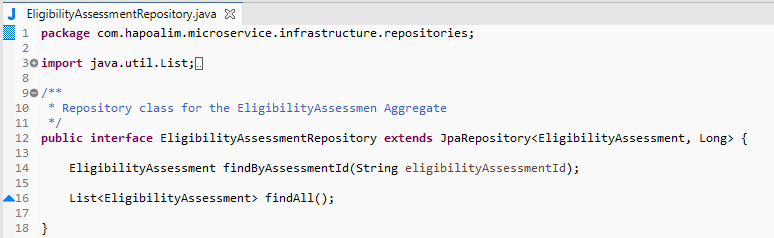


Figure 60 - EligibilityRepository JPA interface

Next figure illustrates the class diagram for our implementation:

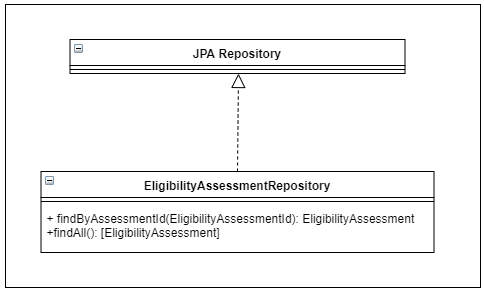


Figure 61 - Outbound Services - Repository implementation

All our Repository implementations follow the same approach.

##### Outbound Services: Rest API(s)

Usage of REST API(s) as a mode of communication between microservices is quite a common requirement. While we have seen event choreography as one mechanism to do it, sometimes a direct call between Bounded Contexts might be a requirement too.

Let us explain this through an example. As part of the Eligibility Assessment process, we need to get some information about the Customer. The data required for the customer is maintained as part of the Customer Bounded Context which maintains the data of the customer. This requires the Eligibility Assessment Bounded Context’s Service to make an outbound call to the Customer Bounded Context’s Service which provides a REST API to retrieve all possible data of the customer.

This is illustrated in the next figure:

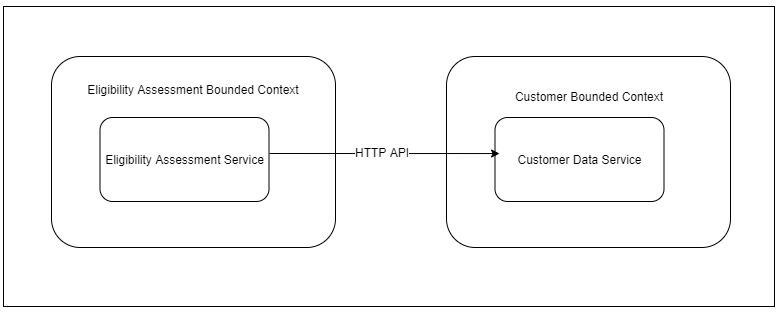


Figure 62 - HTTP invocation between two Bounded Contexts

This however does pose a challenge in terms of the Domain Model. The Eligibility Assessment Bounded Context’s aggregate has a representation of the Customer Data as a “***Customer***” object, while the Customer Bounded Context has a representation of the Customer as a “***CustomerData***” object. Thus, the invocation between the two Bounded Contexts will require a translation of sorts between their domain models.

This translation is typically done in the Anti-corruption Layer which acts as a bridge to communicate between two Bounded Contexts.

This is illustrated in the following figure:

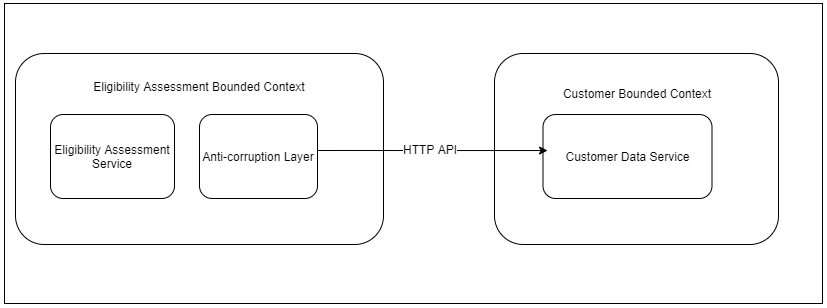


Figure 63 - Anti-corruption Layer between two Bounded Contexts

The Eligibility Assessment Bounded Context relies on the Rest Template capabilities provided by Spring Web to invoke the Customer Data Service’s REST API.

All our Outbound Service implementations which require to communicate to other Bounded Contexts follow the same approach which is illustrated in the following figure:

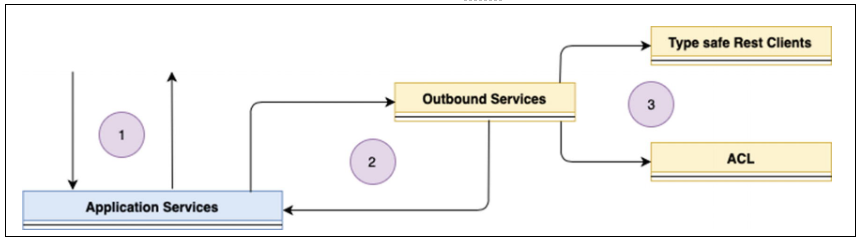


Figure 64 - Outbound Services (HTTP) implemenation process

##### Outbound Services: Message Broker

**TODO: architectural decision (AD-002) 🡪 How the events are going to be implemented. A) Using directly the interfaces of the different brokers (Kafka, RabbitMQ,…), B) using a custom interface which abstract this behaviour to the final developer.**

The final responsibility of outbound services is to raise and publish the Domain Events registered by the Aggregate during the processing of a Command.

In the next figure we illustrate the entire mechanism of the event flow within a Bounded Context.



Let us explain the steps with the following table:

|  |  |
| --- | --- |
|  | Application Services receive requests to process a particular Command (e.g., Eligibility Assessment (Request)). |
|  | The Application services delegates the processing to the Aggregate Command Handlers. |
|  | Command Handlers register the event (e.g., Eligibility Assessment Created, Eligibility Assessment Deleted) that needs to be published. |
|  | The Application services persists the aggregate state utilizing the Repositories of the Outbound Services. |
|  | The repository operation triggers Event Listeners within Outbound services. The Event Listeners collect all pending registered Domain Events that need to be published. |
|  | The Event Listeners publish the Domain Events to the external Message Broker (i.e., RabbitMQ) within the same transaction. |

The following figure illustrates the class diagram for our implementation and the implementation classes too:

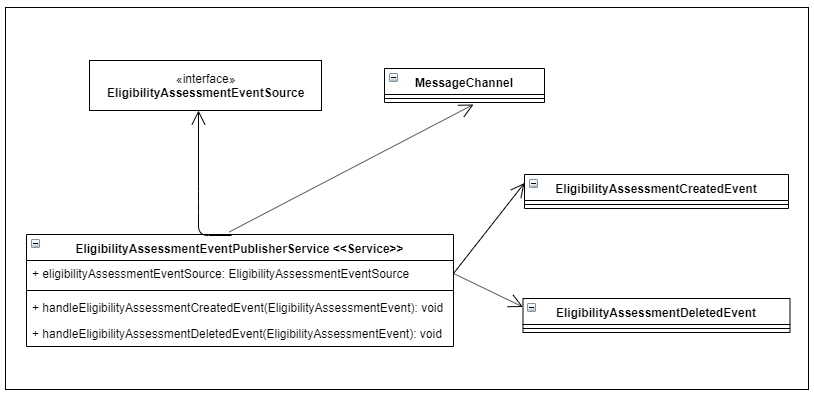


Figure 65 - Class diagram for the Event Publisher implementation



Figure 66 - EligibilityAssessmentEventPublisherService class implementation

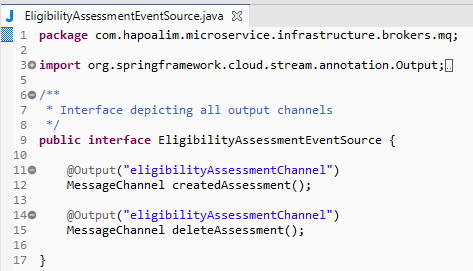


Figure 67 - EligibilityAssessmentEventSource implementation class

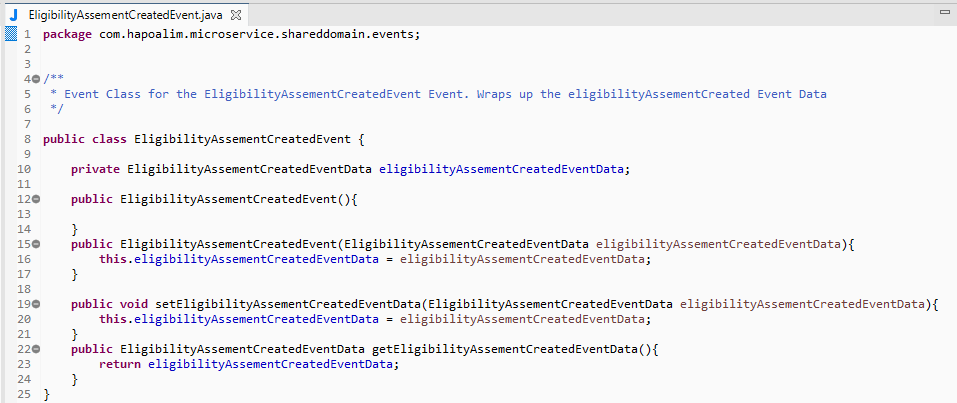


Figure 68 - EligibilityAssessmentCreatedEvent implementation class

## Implementation Summary

* We started by establishing the details about the Spring platform and the various capabilities it provides.
* We decided to use a subset of the projects (Spring Boot, Spring Web, Spring Cloud Stream, and Spring Data) from the Spring Platform’s complete portfolio to help build our Eligibility Assessment (Request) as a microservices application.
* We rounded off by deep diving into the development of the various DDD artifacts – first the Domain Model and then the Domain Model Services using the technologies chosen.

In the following picture, we can summarize the DDD artifacts implementation using Spring Boot as development framework:

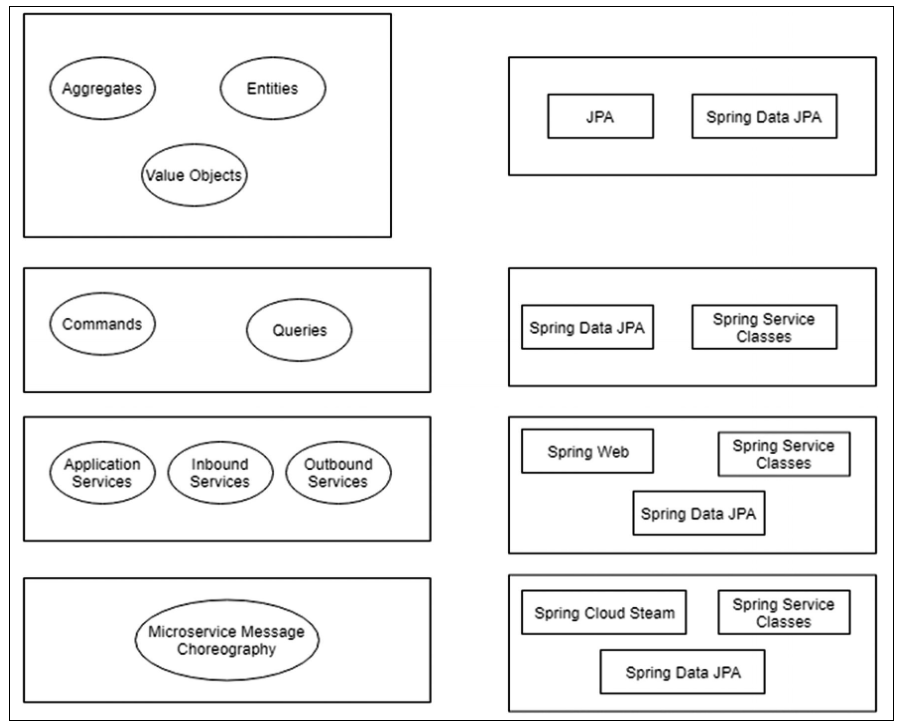


Figure 69 - DDD artifact implementation summary using Spring Boot

# References

**[1]** BankXXXX Bank - Loans Design v0.1: add link

**[2]** Functionals documents: add link

**[3]** Spring Boot: <https://spring.io/projects/spring-boot>

**[4]** *Domain-driven design:* <https://www.amazon.com/Domain-Driven-Design-Tackling-Complexity-Software/dp/0321125215>

**[5]** BankXXXX Bank - Component and Business Object Model: add link

**[6]** Sagas links:

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

https://developers.redhat.com/blog/2018/10/01/patterns-for-distributed-transactions-within-a-microservices-architecture/

https://dzone.com/articles/microservices-using-saga-pattern

**[7]** BHP Error Handling: https://devops.resource.bank/bitbucket/projects/M38200MDS/repos/sample-error-handling/browse

**[8]** XXXX: XXXX

|  |
| --- |
| **END OF DOCUMENT** |

1. Remember that we are not showing all the subdomains for Loans. Just three for educational purposes and simplicity. [↑](#footnote-ref-1)
2. Further details in this model can be seen in the repository for Archi tool (Eligibility Domain) [↑](#footnote-ref-2)
3. **Important note:** Our solution doesn’t depend on the database type selected, so non relational databases (as MongoDB) are also supported in our solution. Now, for educational purposes we will use a relational database (MySQL) to illustrate the sample. [↑](#footnote-ref-3)
4. For a better clarity of the diagram we are not showing all the Entities and the attributes inside the Entities [↑](#footnote-ref-4)