# Software Design Description

**Abstract:**

This document provides a representation or ARYA Testbench to facilitate analysis, planning and implementation.

**Keywords:**

ARYA, ESB, SOA, Testbench

**Approved:**

| Authors : | Project Manager | Product Owner |
| --- | --- | --- |
| Aurélien Tamas-Leloup | Yes | |
| Yanchao Wang | | |
| Yuanbo Wang | | |
| Alexandre Vey | | |
| Ryan Shipp | | |

# Revision History

| Version | Date | Author | Change Description |
| --- | --- | --- | --- |
| 1.0 | 11/20/2015 | A Tamas-Leloup | Added a few general details |
| 1.1 | 12/14/2015 | A Vey | Added Component descriptions |
| 1.2 | 12/16/2015 | A Vey | Added diagram of design of our application |
| 1.3 | 12/30/2015 | A Vey | Reviewed some parts and added some details in architecture parts. |
| 1.4 | 1/16/2016 | A Tamas-Leloup | Added some data class diagram |
| 1.5 | 1/20/2016 | A Tamas-Leloup | Sequence diagrams |
| 1.6 | 1/25/2016 | A Vey | Sequence diagrams |
| 1.7 | 1/25/2016 | R Shipp | Review |

# TABLE OF CONTENTS

# 1. INTRODUCTION

## 1.1. Purpose

Based on the Software Requirements Specifications (SRS), the purpose of this Software Design Document is to provide a description of the design of ARYA. This document needs to be sufficiently detailed for software development to proceed. The developer team should easily understand what needs to be built and how to build it.

## 1.2. Scope

This document will be used as a base for the developers to work on the project.

This document can and should be reviewed by the client in order to validate the architecture and the design of our system.

## 1.3. Definitions, Acronyms, and Abbreviations

| Term/Acronym | Definition |
|---|---|
| ESB | Enterprise Service Bus |
| SRS | Software Requirements Specification |
| UR | User Requirements |
| SDD | Software Design Document |
| STD | Software Test Document |
| QAP | Quality Assurance Plan |
| UC | Use Case |
| DP | Design Pattern |
| SOA | Software Oriented Architecture |
| SOAP | Software Object Access Protocol |
| KPI | Key Performance Indicator |

**Node** : generic term used to describe either a producer or a consumer

### 1.4. References

- SRS :
- Quality Plan :

# 2. SYSTEM ARCHITECTURAL DESIGN

ARYA is composed of several distributed entities that are detailed in section 2.1. We will go through the design of each entity as well as the global design.

### 2.1. System Description

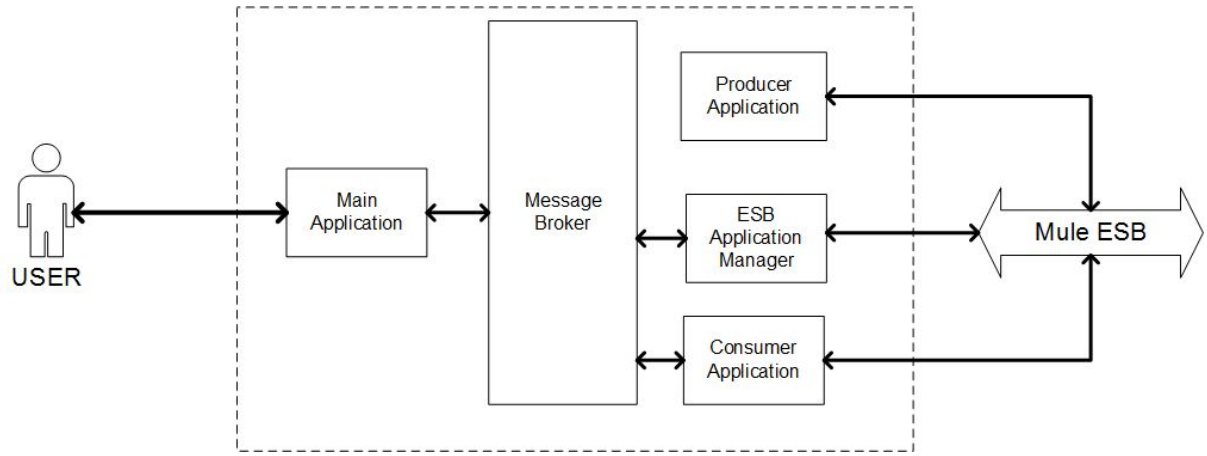


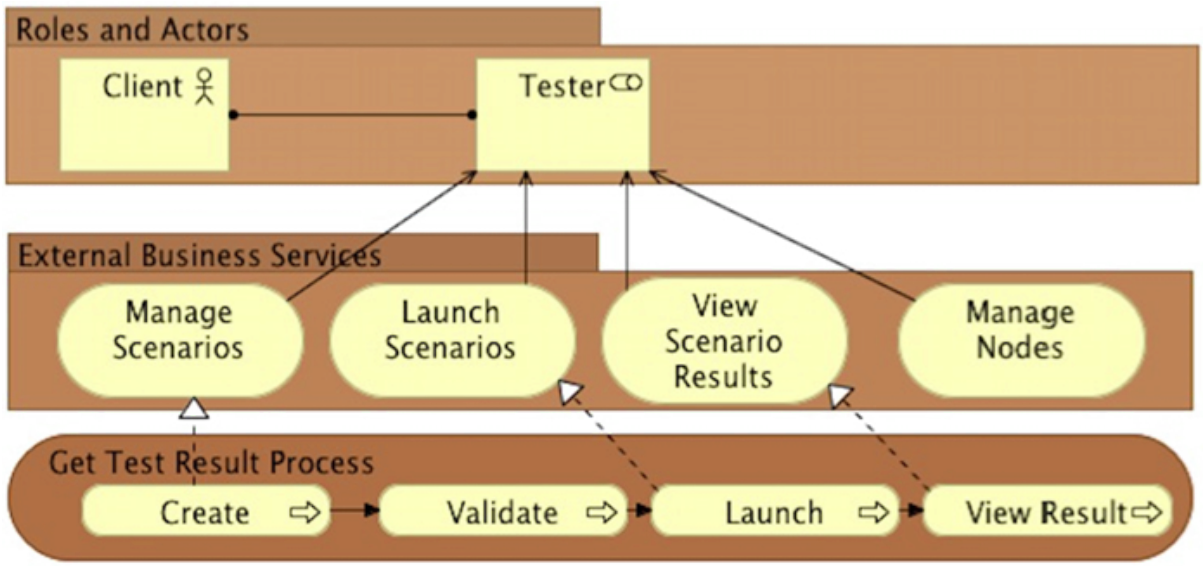

**Figure 1: ARYA WhiteBox Diagram**

The **Main Application** is a web application that interacts with the user. It communicates with the other elements through the **Message Broker.** It also stores the information about the scenarios. Both of **Producer** and **Consumer Application** are distributed. They encapsulate the dummies that will exchange some datas through the ESB. It is possible to have several instances of those applications. Finally, the **ESB Application Manager** deploys new applications on the ESB to make it able to process the different configurations that we can use. This is the only component that is ESB dependant which means the only one that will need to change if we want to test another ESB.

### 2.2. System Architecture

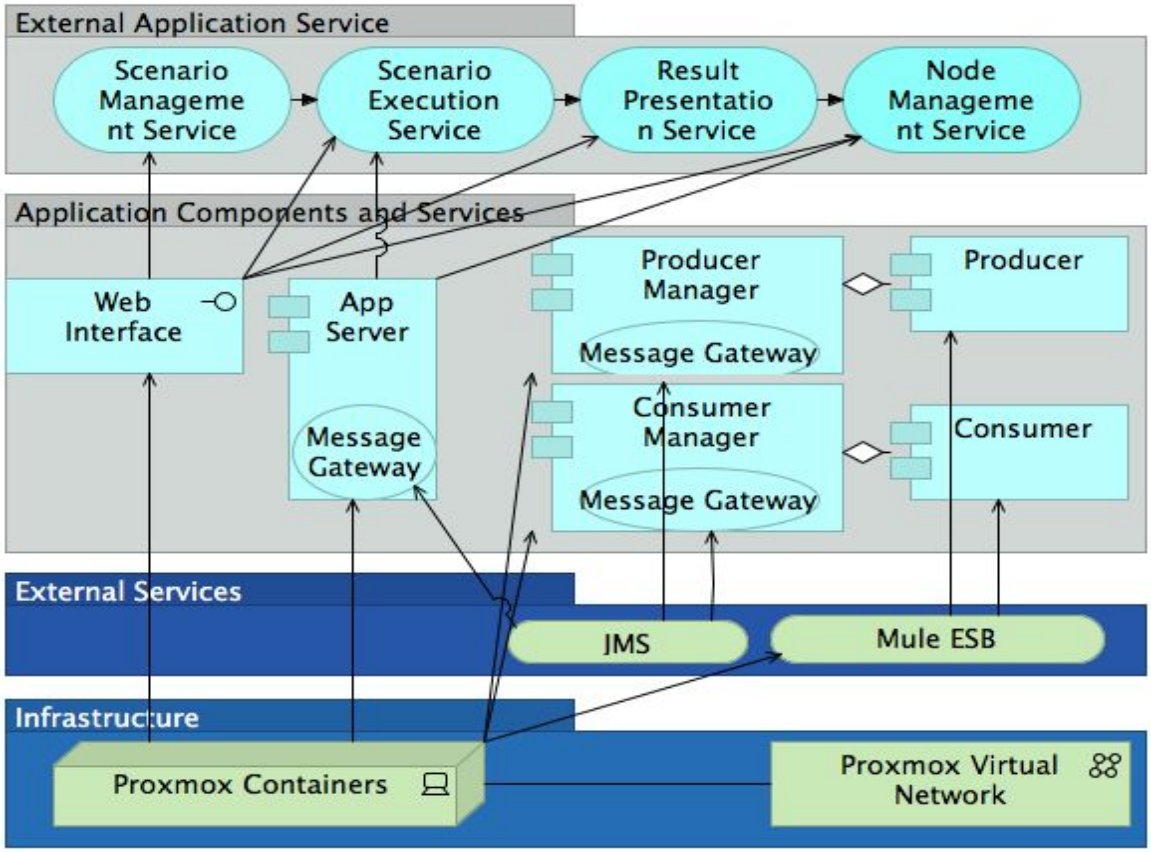
We made the ARYA system architecture with considerations two levels: Business level and system level.

Firstly, we began designing the ARYA system with the business architecture. Here is a diagram for the architecture of this level.




We have the user represented by a Role: Tester, who is trying to get a conclusion with performance obtained with the ESB selected. Our system provides four services listed above to the external world. You can see that a critical user case is satisfied by using these services.

In order to provide these services to the user, we designed our system accordingly. Meanwhile, we consider that the system architecture should be easily modularized. Below is a diagram for the system level architecture.




Proxmox containers connected by a proxmox virtual network worked for us as our infrastructure. We will be leveraging JMS and Mule ESB instances to provide communicating means. The scenario is executed by the app server which orchestre the producer manager and consumer manager. They will be communicating through JMS with a “message gateway” component integrated into them. When a producer and consumer are created by their managers they will communicate through Mule ESB, which is our testing target.

This way, the four services listed above at the top of this diagram can be provided by application components and services.

Alternatively, we had considered other architectural solutions, including putting a consumer/producer in a container and waking it up when needed. We have decided this solution is too heavy and will not be able to satisfy our needs.

## *2.3. Design Constraints*

Below shows some limitations on the conditions under which our system is developed. The constraints include hardware as well as software aspects.

### 2.3.1. General Constraints

We need to allow the user to easily test the performance of different ESB, so our system needs to be like a generic system which can test any ESB.

### 2.3.2. HW Constraints

The main hardware constraints are to have enough resources for the system. If it does not have enough resources the system could run with some delays and distort the results.

Another constraint is the network. In case of a large number of communications between Consumers or Producers and ESB, if the communication are not fast enough, the results can be distorted.

#### 2.3.3. SW Constraints

All the applications will be implemented using JAVA EE. They will communicate through a Message Broker using the JMS API. Applications node will include either a SOAP Web Service Consumer or a SOAP Web Service Producer. The scenarios will be described using XML and validated with XSD.

# 3. COMPONENTS DESCRIPTION

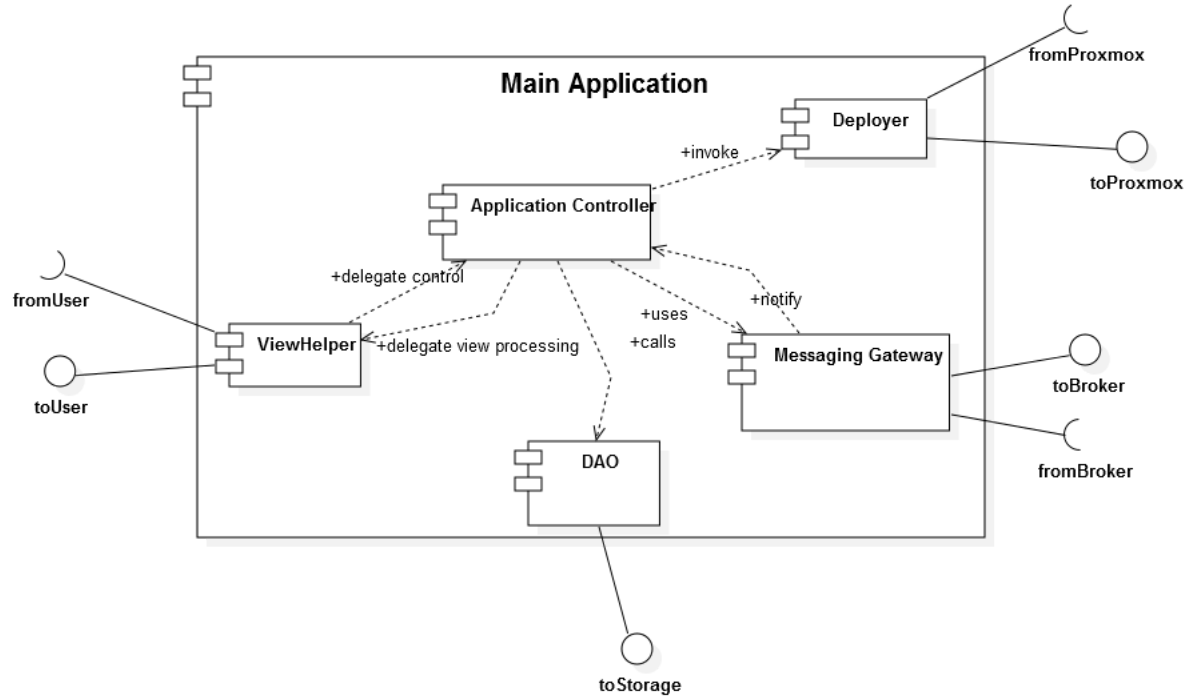
## *3.1. Introduction*

This part will show you the differents elements of our system. It presents our system's deployment and the class diagram. The black-box will be decomposed and explained.

## *3.2. Decomposition description*

### 3.2.1. Main Application

This is the component diagram of our main application :




We will use the Model-View-Controller design pattern.

**View :** Our view is a viewHelper, which acts like an adapter for the view and will help to integrate the new view in the main application.

**Controller :** Our controller is the ApplicationController. It will communicate with the messaging gateway, with the DAO and with the deployer. We will implement a singleton to make sure that there is only one instance.

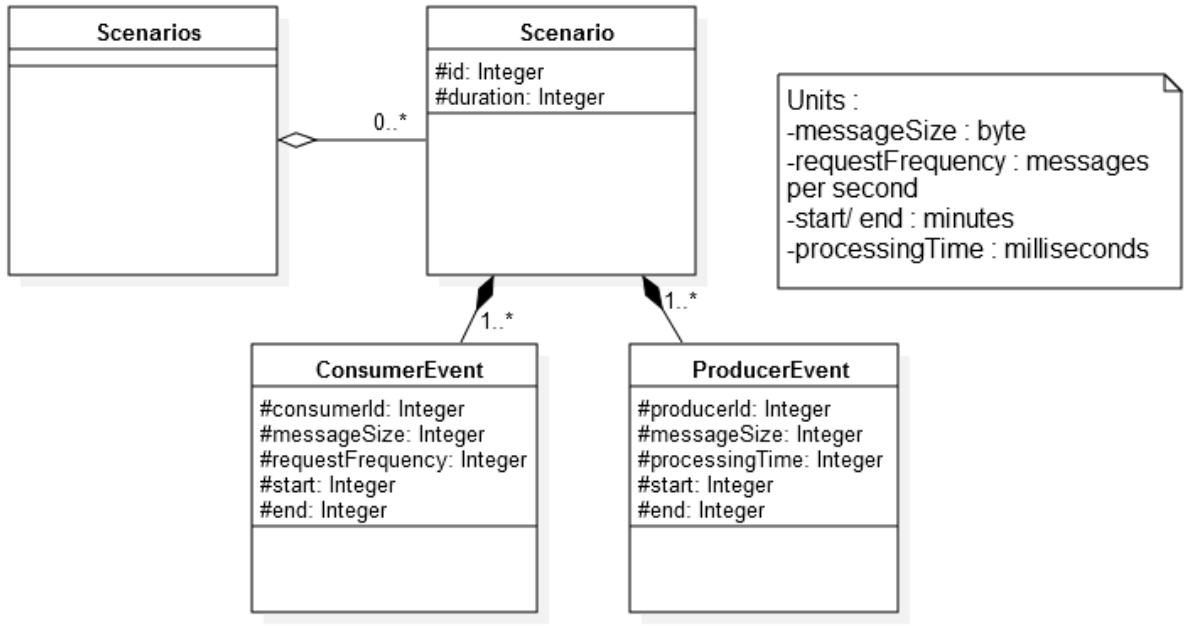
**Model :** The DAO is our model. It will implement the communication between the controller and the database.

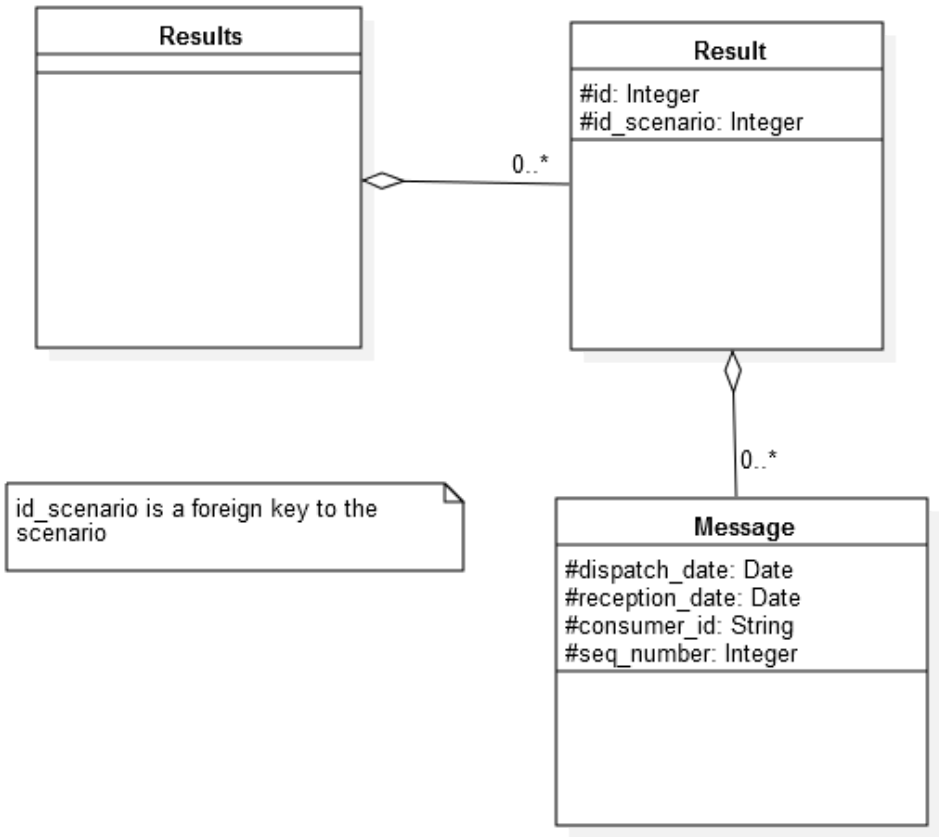
The deployer is used to deploy new consumers and producers, It will include a Java REST client. This will allow us to control Proxmox to deploy/remove nodes from the datacenter

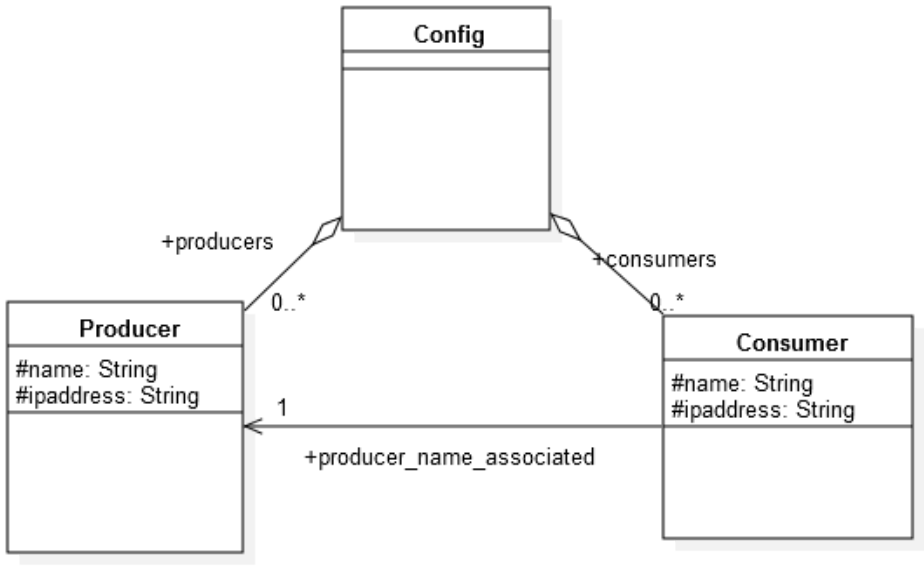
The messaging gateway is useful to wrap and unwrap the messages in a specific way to send and receive message easily.

For the deployment of the producers and consumers, we have access to an OVH server with Proxmox. For each new producer and consumer we will create a new container and deploy the producer or the consumer on it.

**Database :** We chose to store our data as XML files to allow the user to edit the data without using the GUI. We have three different types of data to store : scenarios, results & configuration. Here are the different class diagrams :


Scenarios
Scenario
#id: Integer
#duration: Integer
0..*
Units :
-messageSize : byte
-requestFrequency : messages per second
-start/ end : minutes
-processingTime : milliseconds
1..*
1..*
ConsumerEvent
#consumerId: Integer
#messageSize: Integer
#requestFrequency: Integer
#start: Integer
#end: Integer
ProducerEvent
#producerId: Integer
#messageSize: Integer
#processingTime: Integer
#start: Integer
#end: Integer

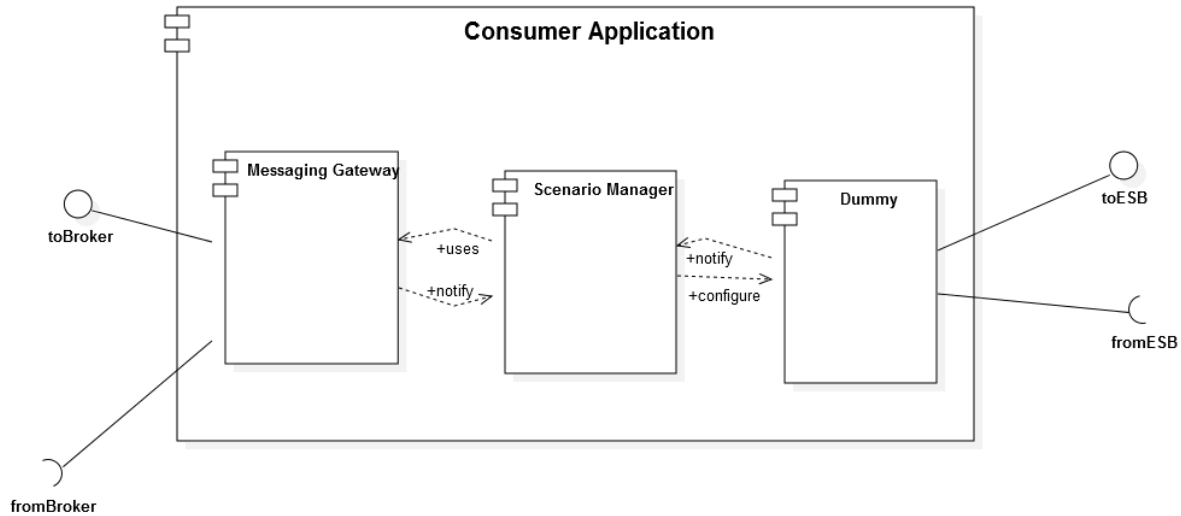

Results
Result
#id: Integer
#id_scenario: Integer
0..*
0..*
id_scenario is a foreign key to the scenario
Message
#dispatch_date: Date
#reception_date: Date
#consumer_id: String
#seq_number: Integer




### 3.2.2. Message Broker

To communicate between the different components of the application we will use ActiveMQ. We will use publish/subscribe or point to point messaging.

### 3.2.3. Consumer

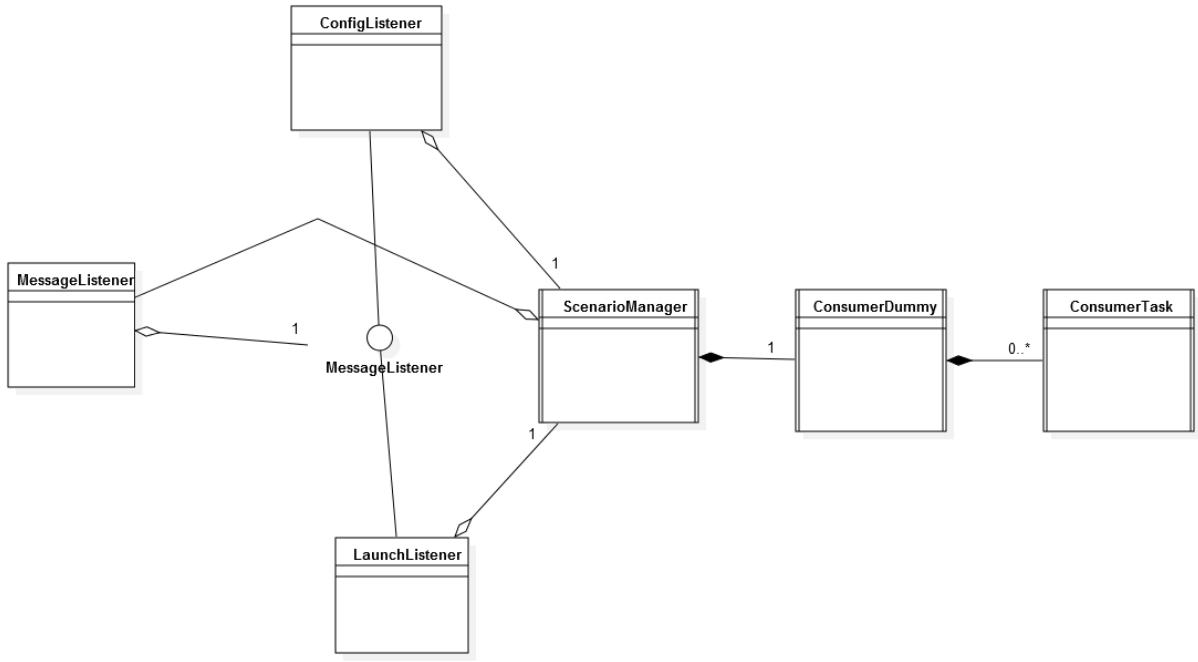



The consumer application is composed of a messaging gateway, a scenario manager and a dummy.

The messaging gateway serves the same purpose as in the main application side, because we need to wrap and unwrap the message on both sides.

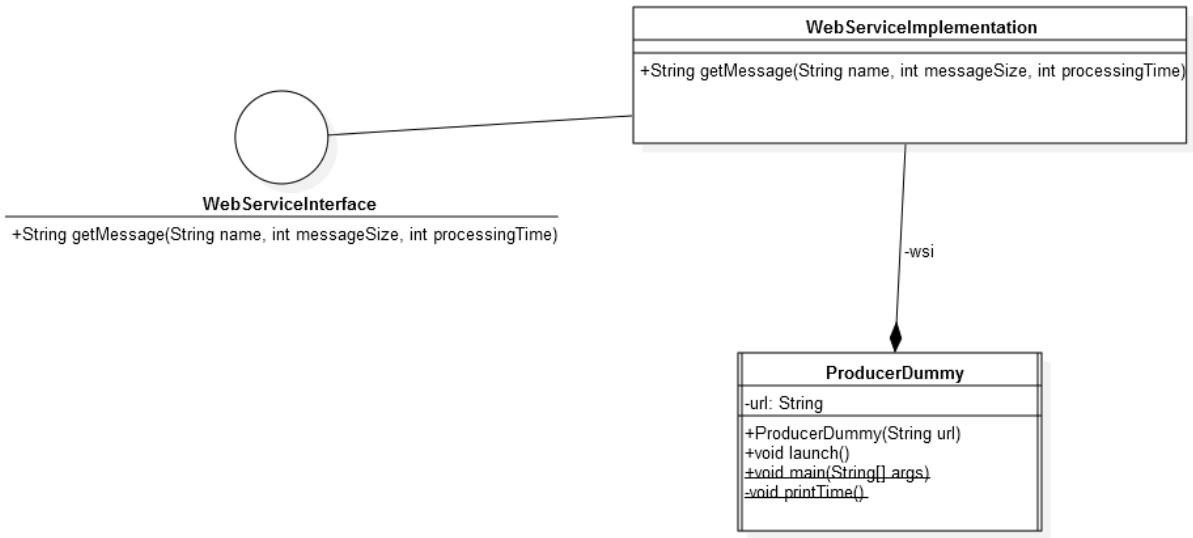
The scenario manager will read the scenario from the xml file and can configure the dummy with a specific scenario.

The dummy will be configured, enabling it to communicate with producers through the ESB.




### 3.2.4. Producer

The producer is a very basic application. It provides a web service and adapts its behaviour with the data it receives. Here is the associate class diagram:




### 3.2.5. ESB

We will use Mule ESB to test our different scenarios. The ESB is useful to make the link between producers and consumers. Our goal is to test this ESB to see if it has good performances in function of the number of producers and consumers, the number of messages and the frequency of messages.

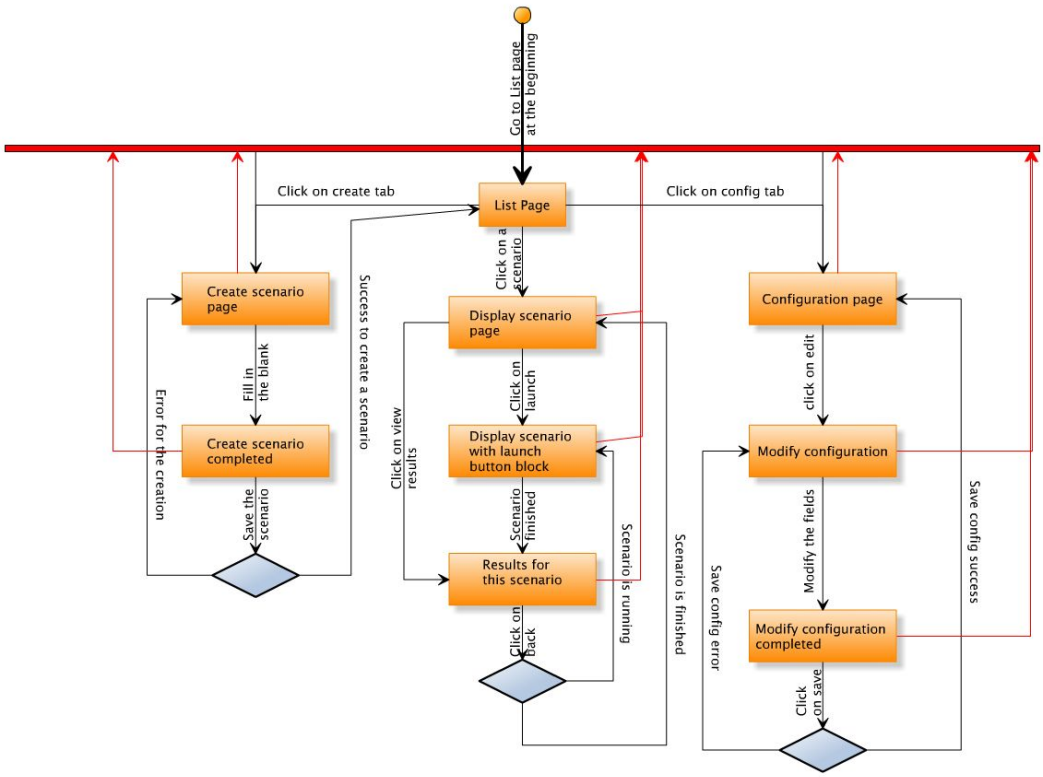
# 4. EXTERNAL INTERFACES

## *4.1. Introduction*

The goal of this section of the document is to precisely describe how the system interacts with internal or external actors like the user or the message broker. This will help the developers to understand what they need to implement for the communication.

## *4.2. User interfaces*

Here you can find the activity diagram for the user interfaces.




The user interface will allow the user to see the different scenarios already implemented in the database and see the results for a specific scenario if the scenario has already been launched. He can also create new scenarios with a specific configuration, configure existing producers and consumers, and launch scenarios. When a scenario is running, the user must wait to launch another scenario until the active scenario has finished its task.

### *4.3. Other system interfaces*

#### 4.3.1. Message Broker

The message broker will communicate asynchronously using the JMS publish/subscribe messaging domain or a simple messaging queue.

- **Main Application to Broker**

The Main application will send the scenario to the nodes application using a serialized Scenario object. Once the message is received by the Nodes, the Scenario can be launched.

- **Node Application to Broker**

The Node Application will send the results to the main application using a serialized Result object.

#### 4.3.2. ESB

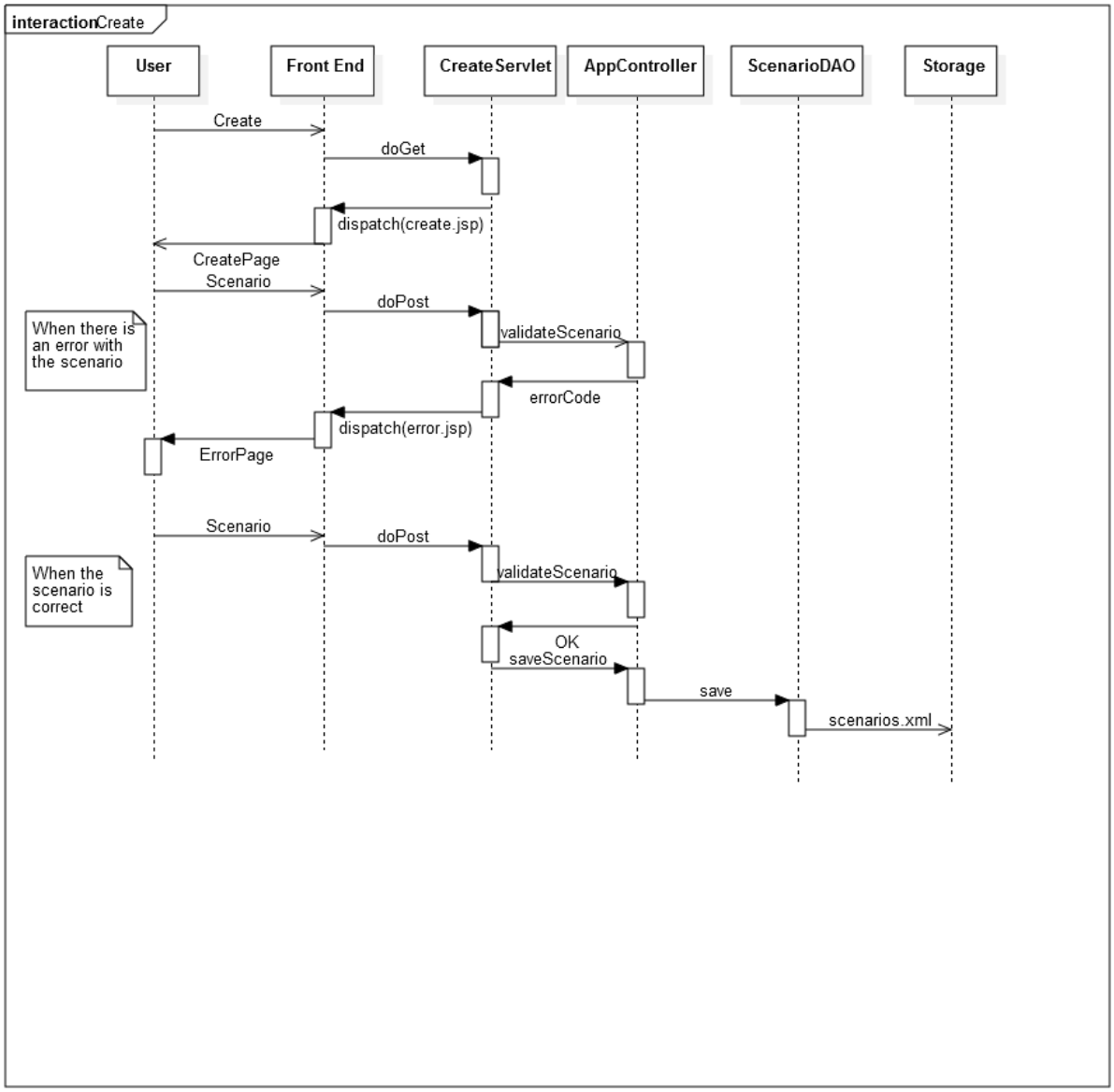
The consumer and producer will communicate with each other via web service requests through the ESB. The consumer and producer both implement a WebServiceInterface, which has a single “getMessage” method that accepts parameters for “name,” “messageSize,” and “processingTime” and returns a string.

# 5. Detailed design

## 5.1. Introduction

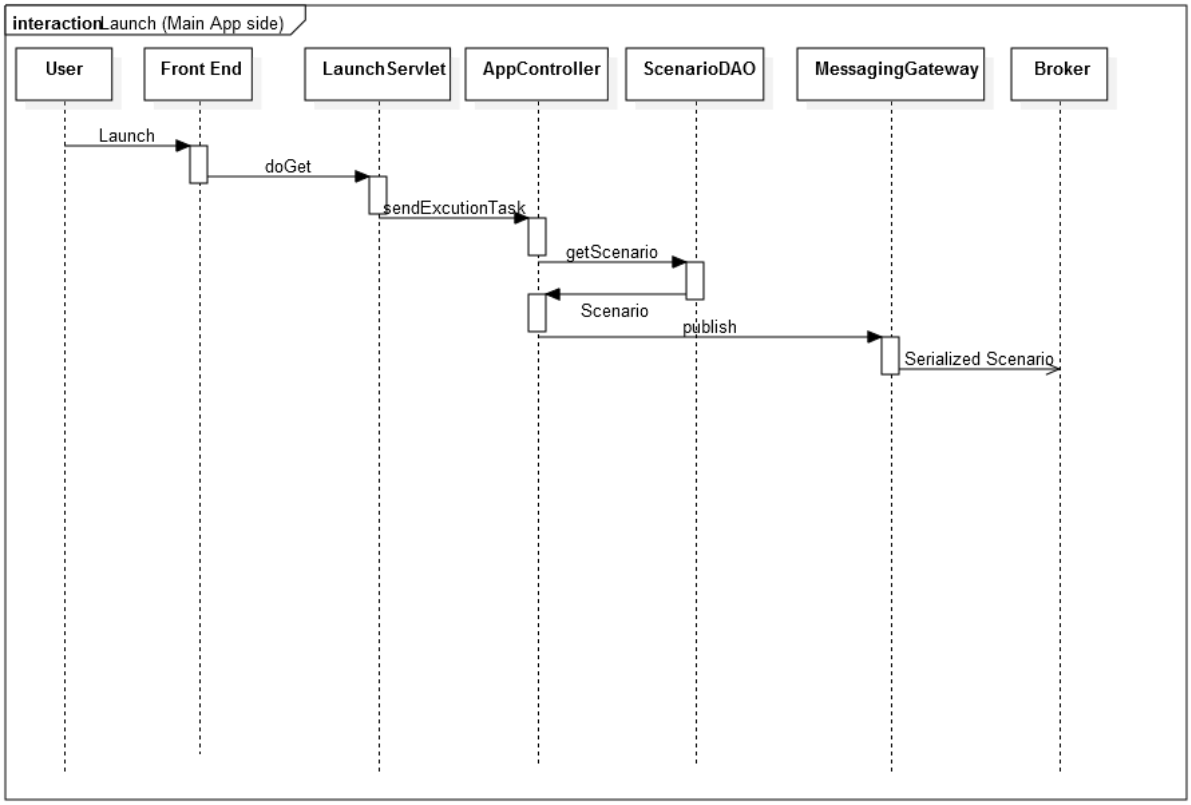
This section explains the internal interaction between the components for the different use case implementations. We provided sequence diagrams for each of them.

## 5.2. Create

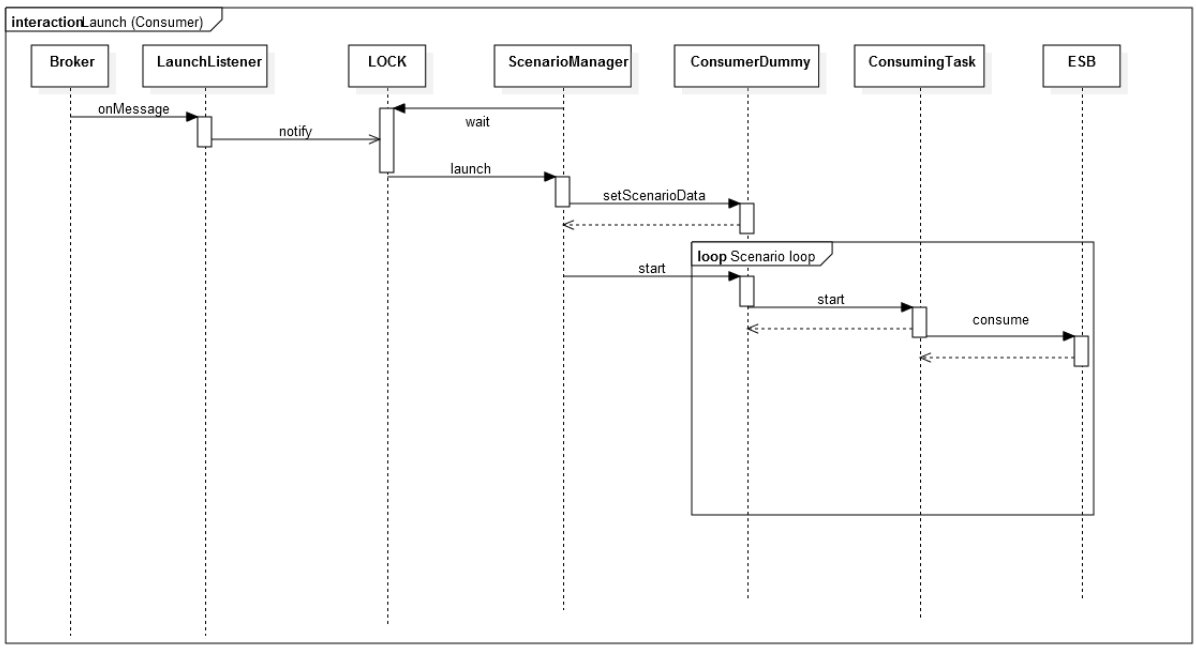



## 5.3. Launch

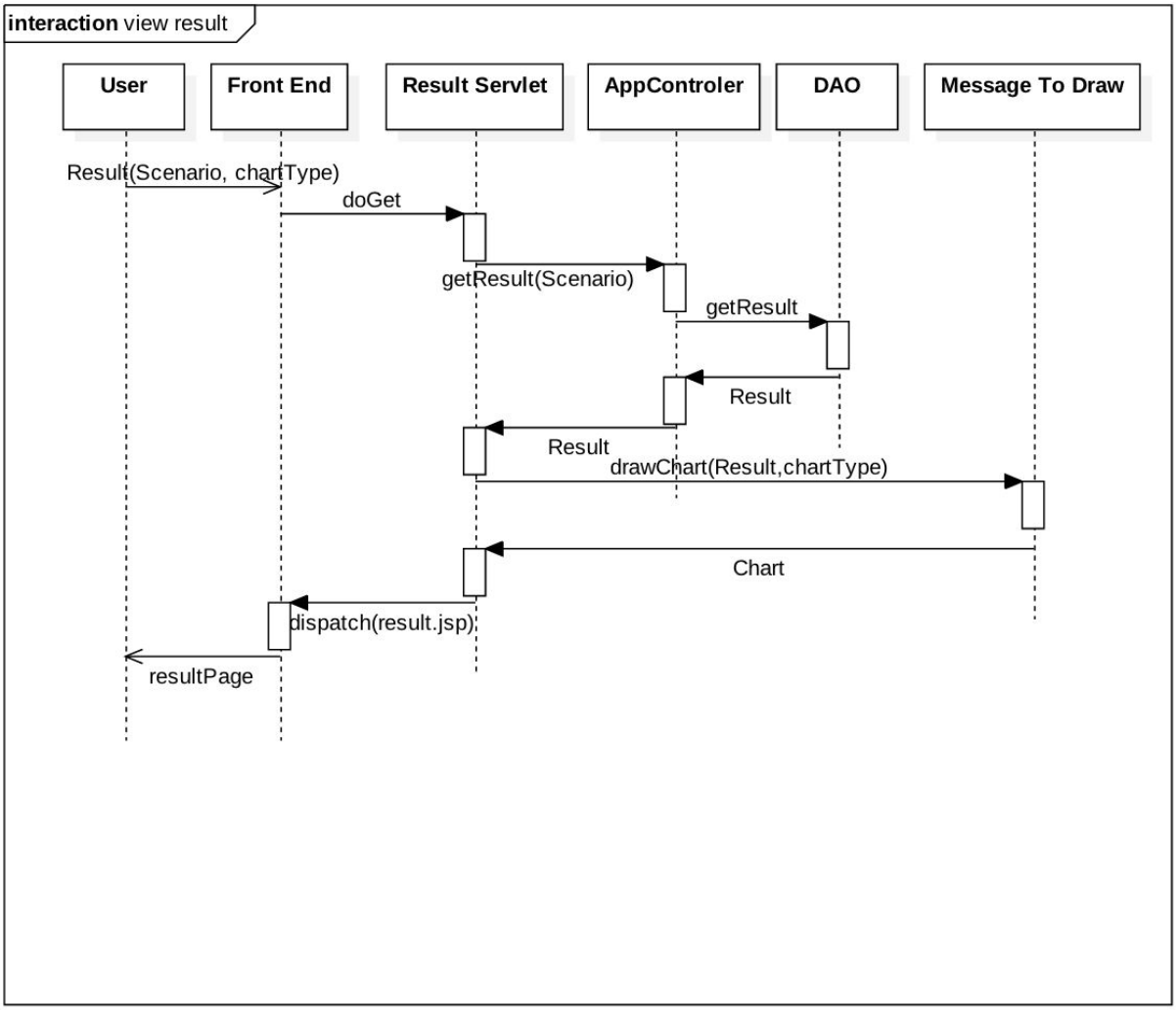
### 5.3.1. Main App side




### 5.3.2. Consumer side

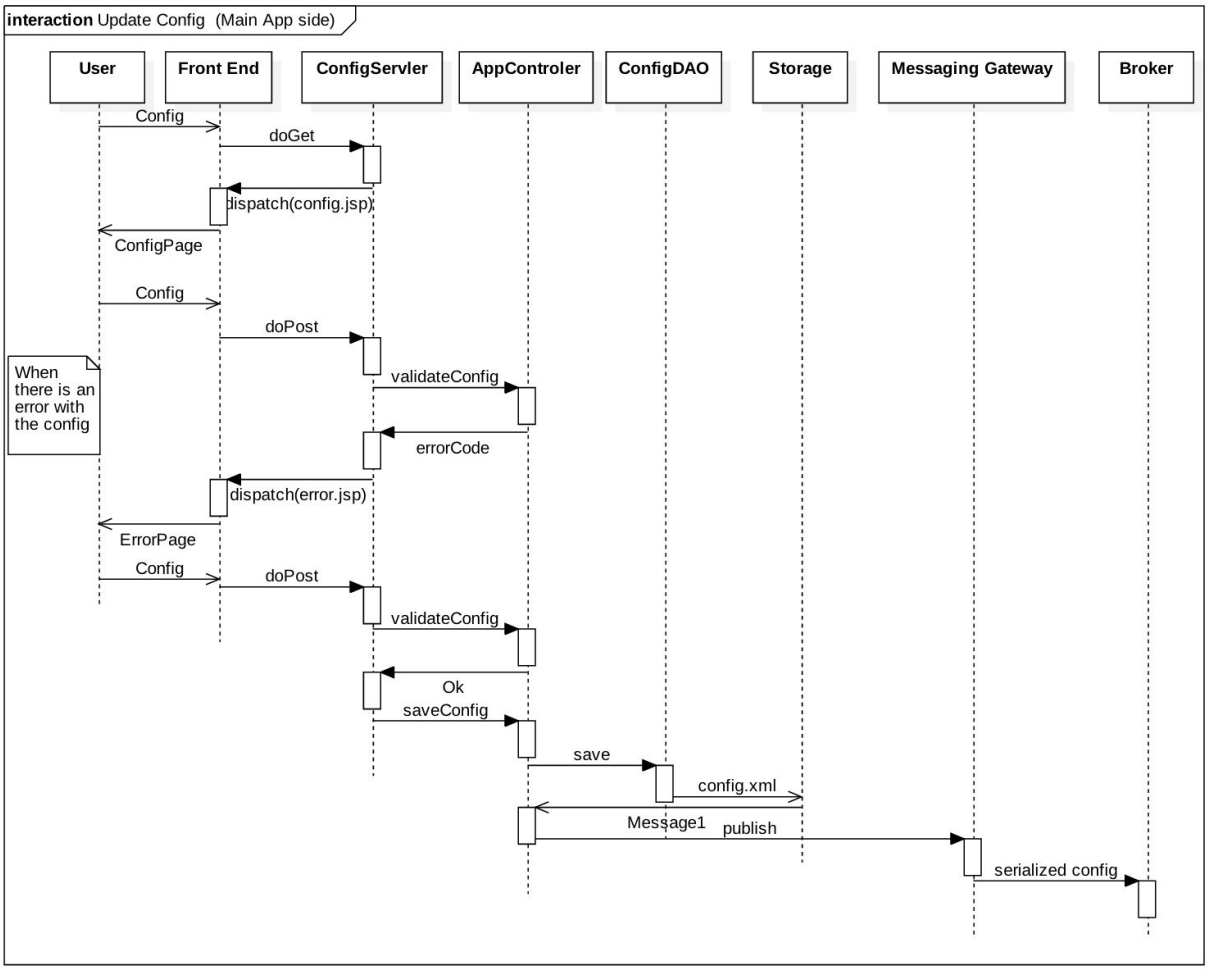



## 5.4. View Results

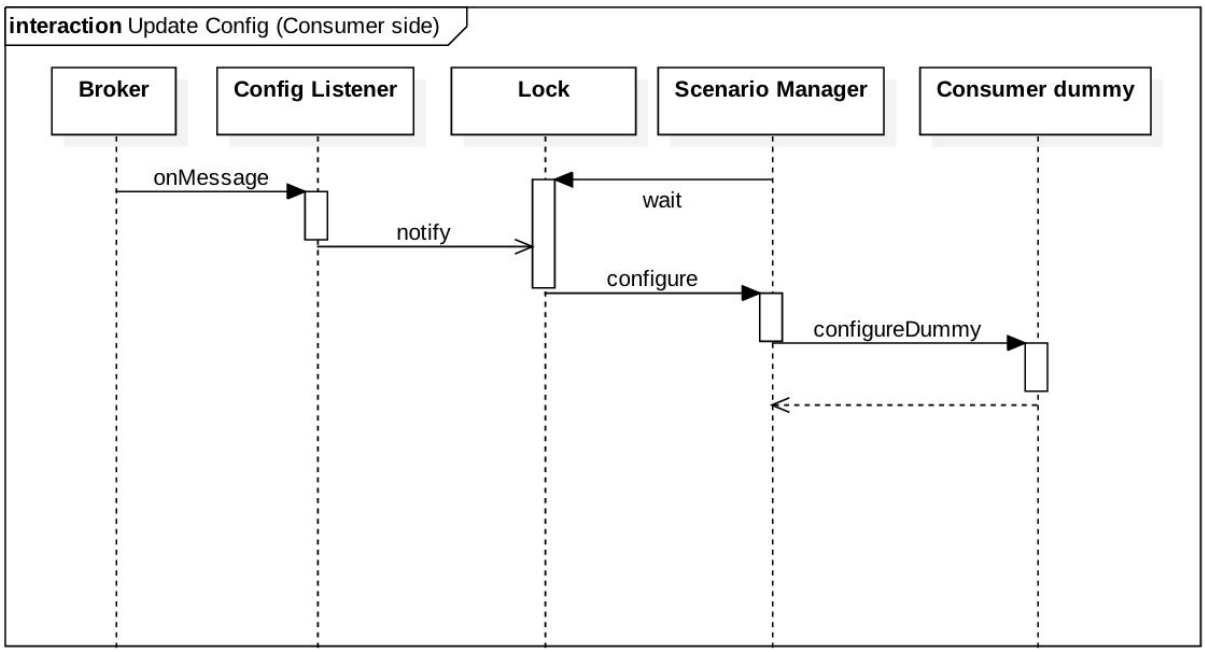



## 5.5. Update Config

### 5.5.1. Main App side




### 5.5.2. Consumer side




# 6. Annexes

### *Traceability*

For traceability, we used the IBM software Rational DOORS. A package is linked in the documentation of the project.