

Design and development of a highly configurable solution to manage and supervise batch executions

OUERGHI Yassine

To all ...

Contents

Abstract	5
1 General Frame of the Project	7
1.1 Presentation of the host organization	7
1.2 Project Background	7
1.3 Proposed Solution	8
1.3.1 RESTful APIs calls	9
1.3.2 Rule Engine	10
1.3.3 JAR Implementation	10
2 Analysis and Specification	13
2.1 Adopted Development Methodology	13
2.2 Functional Requirements	14
2.3 Non-Functional Requirements	14
2.4 Requirements' Modeling	15
2.4.1 General Use Case Diagram	15
2.4.2 Detailed Use Case diagrams	16
3 Design	21
3.1 Architecture	21
3.2 Class Diagram	22
3.3 Sequence Diagram	24
3.4 Activity Diagram	26
4 Implementation	27
4.1 Development Environment	27
4.2 Solution	28
4.2.1 Spring Batch Steps	28
4.2.2 Asynchronous Communication	30
4.2.3 Configuration	31
4.2.4 Report templating & generation	32
4.2.5 Logs management	33
4.3 Overview of the achieved work	34
5 TODO	43
5.1 Kanban artifacts	43
5.1.1 Cumulative Flow Diagram	43

5.1.2	Control Chart	44
5.1.3	Backlog	44
5.2	Perspective	46
General Conclusion		47
Bibliography		49

List of Figures

1.1	Simple Batch Job Execution Components	8
1.2	RESTful APIs Calls Solution Diagram	9
1.3	JAR Implementation Solution Diagram	11
2.1	Main use case diagram	16
2.2	Manage Job Configuration Use Case Diagram	17
2.3	Manage Job Executions Use Case Diagram	18
2.4	Manage Report Use Case Diagram	19
3.1	Architecture	22
3.2	Class Diagram	23
3.3	Sequence Diagram	25
3.4	Activity Diagram Job Life Cycle	26
4.1	Graphical Representation of the Solution	29
4.2	Kafka Performance vs others [iT16]	30
4.3	Reporting Engine	33
4.4	Login interface	34
4.5	Dashboard interface	35
4.6	Creating a Job Configuration 1	36
4.7	Creating a Job Configuration 2	36
4.8	Job Batch Monitoring	37
4.9	Report List	37
4.10	List of Generated Specific Reports	38
4.11	List of Report Template	38
4.12	General Report Editor	39
4.13	Specific Report Editor	39
4.14	Example of a Generated Report	40
4.15	Job History	40
4.16	Managing Report Parameters Category	41
4.17	Managing Report Parameters	41
4.18	A Sample Kibana Dashboard	42
5.1	Cumulative Flow Diagram	43
5.2	Control Chart	44

List of Tables

2.1	Manage Job Configuration Use Case Description	17
2.2	Manage Job Executions Use Case Description	18
2.3	Manage Report Use Case Description	19

Abstract

TODO: Abstract/Summary

1 General Frame of the Project

Introduction

In this chapter, I will start by presenting the host company, along with a presentation of the problem I am trying to solve, afterward I will introduce my proposed solution.

1.1 Presentation of the host organization

Human Resources field is a vast fertile ground for innovation, this is where Advyteam chose to operate, by creating an HR Solution named “Accretio”, suited for all kind of enterprises, it spans different areas of expertise, divided in several modules which the client can choose from. Modules such Core HR functionalities, Talent Management, Leave Management, and others.

Currently, Accretio is an install-based application, however, the company is actively porting the application to a SaaS (Software as a Service) based one, with support of multi-tenancy, elasticity, etc. Accretio uses the microservice architecture, which makes creating new services, and plugin in new services relatively easy.

Advyteam is a young Tunisian company, established 2011 by three human resources specialists, its headquarters is situated in Paris, France.

Besides its Human resource management system Accretio, Advyteam also provides certification programs in Human resources through its Advyteam Academy and offers training and coaching in almost all of the human resources fields.

1.2 Project Background

With its effort to port the application to a SaaS software model, the company’s engineers have faced several problems, one of these problems is how to give the ability an application user to define, launch and monitor batch jobs.

A batch job is a computer program or set of programs processed in batch mode. This means that a sequence of commands to be executed by the operating system and submitted for execution as a single unit. [UNI18]

Currently, the application has several built-in batch jobs, such as calculating the employee's payrolls, leaves, evaluations, etc. However, those batches are neither extendable nor configurable.

Say we have two clients, they both need to generate their employee's payrolls, yet they both have different business rules to calculate the salary, in this case we may end up writing two separate batches for each client.

Furthermore, each time a client comes up with a new use case for batch processing, or a new business rule to be executed in a batch job, he must revert to the Advyteam developers to write a new specific batch, which could be very tedious in a SaaS architecture with tens or hundreds of clients.

In the next section, I will present the different solutions that we came up to try to solve the above problem.

1.3 Proposed Solution

The most challenging aspect of this project is how to provide to the client, a mean to insert his own logic into the application, with little or no help from the Advyteam team.

Before we dig into the different solutions, we need to have a very basic knowledge of how a batch work, we will dive deeper on how a batch work in future chapters.

A batch job is mainly composed of at least one step, each step itself is divided into three other steps;

- A reading step, from which we will read the data that will be processed, this “data” is the list of the registration numbers of the employees that would be processed.
- A processing step, where the data that has been read in the previous step be will processed, this step will hold the client business logic.
- A writing step, where you can write the changes that you have made in a datasource, generate reports... etc.

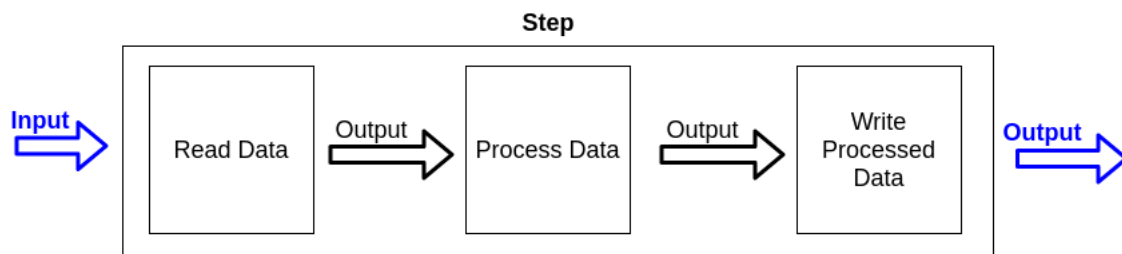


Figure 1.1: Simple Batch Job Execution Components

We need to find out a way where the client can enter his own logic for all the three steps mentioned above, then somehow insert and execute them into our application. I will detail in the next subsection the solutions that we came up with and the choice we made;

1.3.1 RESTful APIs calls

The first solution we came up with, is to create a single batch job that consumes a RESTful API for each step, where we can call for the reading, processing and writing operations.

However, this solution mandates that the client expose these RESTful APIs, and host it somewhere on his own servers, we cannot guarantee that all our clients would be able to host a RESTful APIs.

Another drawback is that this solution would be very inefficient, we would bombard the client servers by our calls, and the processing would be done on their servers as-well.

This solution is quite simple but presents lots of inefficiencies.

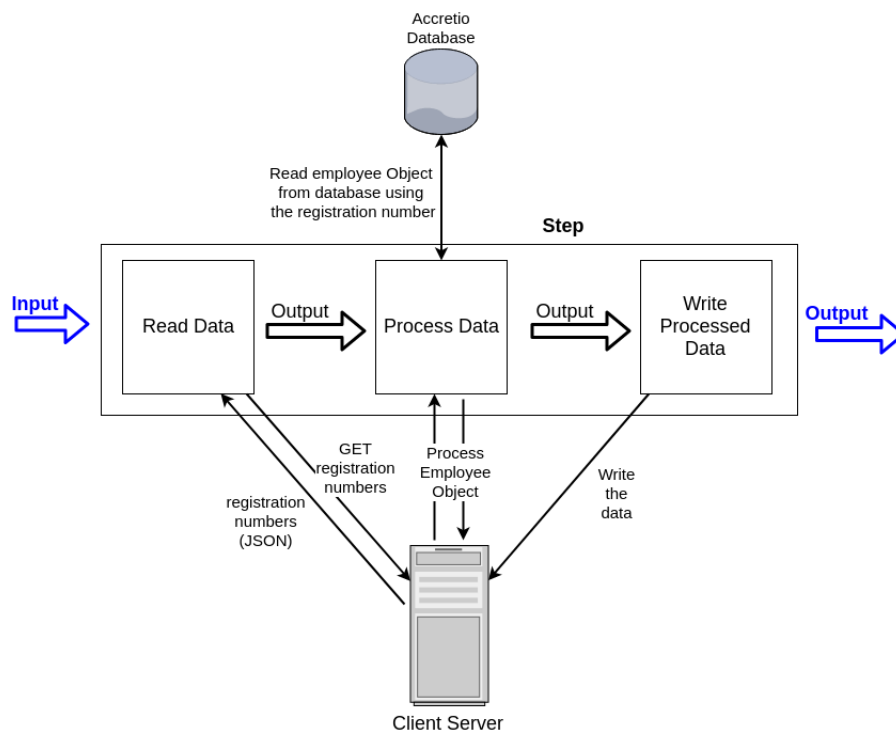


Figure 1.2: RESTful APIs Calls Solution Diagram

1.3.2 Rule Engine

We could also use a rule engine (such as drools), give the client the ability to define the rules that he would like to apply, and execute those rules when selecting the employees population (reading step), and in the processing step as well.

As one can notice, this solution is very limited to the number of possibilities that the client can choose from (finite list of rules) and would be able to define very simple and basic operations, simple filtering in the reading step, and basic math operations in the processing step.

The client would have no control over the reading and writing step, which is another huge limitation, nor over the data structure to be used.

1.3.3 JAR Implementation

We decided to go Java's way, by defining a specification, and let the vendor (client in our case) implement this specification [Ora]. The solution we went for is to create two separate jars:

- Specifications Jar: Also called "Template Jar", which will be handed to the client, where he must implement it in order to write his own business logic. This jar will contain a set of interfaces and abstract classes, for each of the three steps mentioned above. The classes that the client wrote, would then be autowired in a runner jar.
- Runner jar: Will hold the batch steps definitions and configurations and would execute the methods that has been wrote by the client in our environment, by autowiring/injecting the classes that have been implemented.

After implementing the Template Jar, the client only needs to upload the generated Jar, through the frontend application form, and specifies a set of options, such the datasource that would be used in the reading step, whether the job should be scheduled, number of retries ..., he can then manage those jobs, by starting them, stopping, monitoring, view generated reports, and so on.

We will go through the job configuration options in details in later chapters.

Figure 1.3 below, shows a visual representation of the solution, note that the squares marked in red, represents where the client's business logic will be injected, and executed.

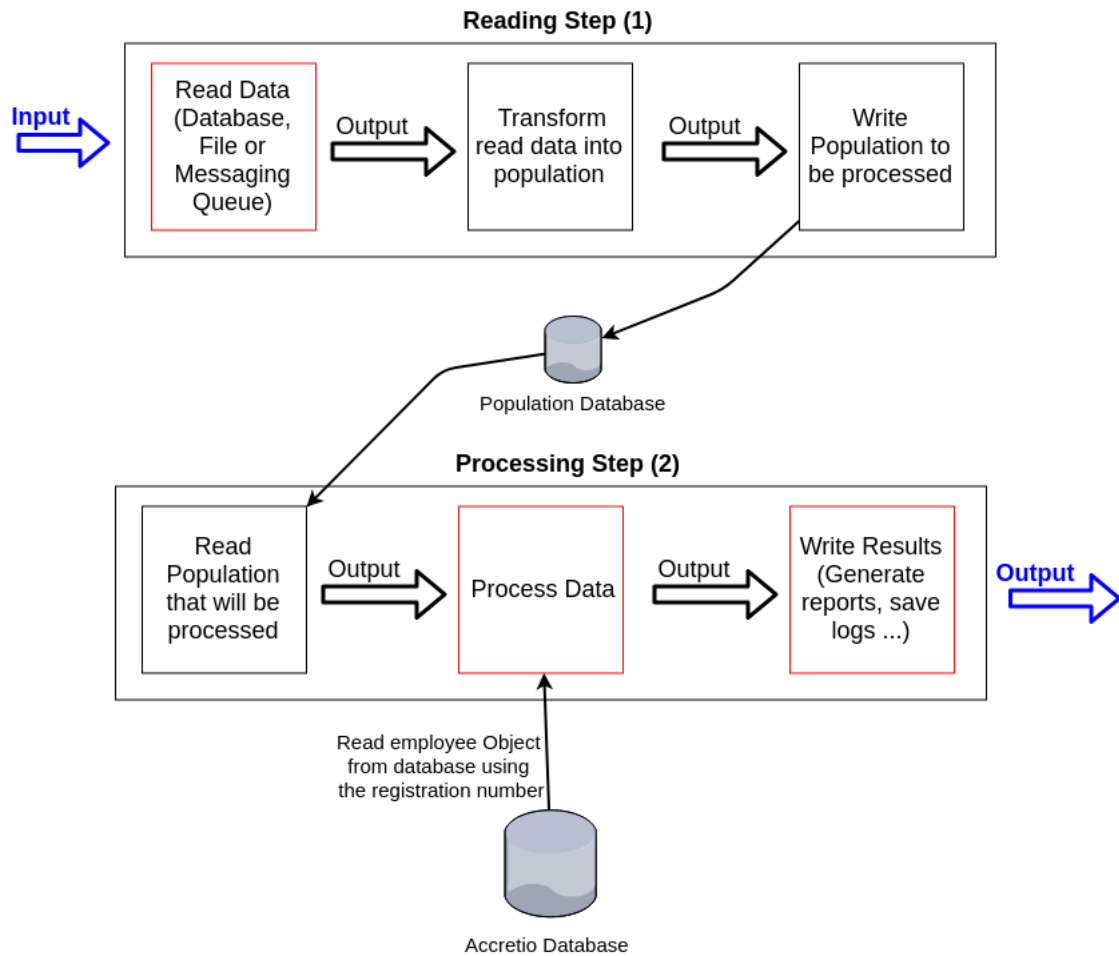


Figure 1.3: JAR Implementation Solution Diagram

Conclusion

In this chapter, I presented the frame on which this project is based on, by introducing the host company as well as the needs behind the project. Finally, I ended this chapter by presenting the different proposed solutions, and the one that I went with. The following chapter will be devoted to the analysis and the design of the application.

2 Analysis and Specification

Introduction

The first prerequisite you need to fulfill before beginning construction (development) is a clear statement of the problem that the system is supposed to solve. [McC04]

In this chapter, I will describe the adopted methodology along with the functional requirements and non-functional requirements. Lastly, I will present the system actors and use cases.

2.1 Adopted Development Methodology

A system development methodology refers to the framework that is used to structure, plan, and control the process of developing an information system. One system development methodology is not necessarily suitable for use by all projects. Each of the available methodologies is best suited to specific kinds of projects, based on various technical, organizational, project and team considerations. [fMMSCOoIS08]

For this project I have opted to use the Kanban method, the main reason for this choice is that this project requires a lot for researching, and many factors are not known beforehand, such which solution will be used, what messaging system will be used, and so on.

The tasks change frequently, unplanned features or use cases would be added on the fly if a valid Proof of Concept was introduced. Kanban is suited for these kinds of projects.

Kanban provides a method to continually adapt in order to smooth out kinks in the arrival of new development work. In this way, it allows the organization to avoid crises and respond more quickly and easily to issues that do arise. [ALR10]

In the last chapter I will be introducing the different Kanban artifacts, such as the backlog, however for the next section, I will be discussing the functional requirements.

2.2 Functional Requirements

Functional Requirements describe what a software is supposed to do and offer to its users, they are the first step toward a solution. We have a single actor in this particular application, which the administrator. The following, represents the requirements that the system must offer to its users:

- Manage Job Configurations:
 - Create a Job Configuration by uploading jar file (that contains the client business logic) and setting the running configurations.
 - Edit/Delete job configuration.
 - Start a Job Batch using a Job Configuration
 - Stop a running Job Batch
 - Restart a stopped Job Batch
 - Retry a failed/locked Job Batch
 - View a Job Configuration launching history
 - Download a Job Configuration launch log
 - Download a Job Configuration launch report (if report generating is enabled when creating the job configuration)
- Manage Reports:
 - Create Job Configuration reports template
 - Create Specific reports template
 - Manage Report Template Parameters
 - Generate Report using a Template and parse parameters
- Dashboard summarizing jobs executions and states:
 - Number of job executions per date
 - Job executions status (completed, on progress, stopped, locked, failed)
 - Number of reports generate per report template
 - Visualize all job executions logs, with the ability to filter on the results

2.3 Non-Functional Requirements

Non-functional requirements detail constraints, targets or control mechanisms for a given system. They describe how, how well or to what standard a function should be provided. For example, levels of required service such as response times; security

and access requirements; technical constraints; required interfacing with users' and other systems. [Com10]

In addition to the features and functions that the system will provide, this application needs to satisfy various Non-Functional Requirements:

- **Security:**
Unauthorized access to the system and its data is not allowed. Only authenticated users can perform an operation, such as creating a new job, launching a job, or viewing job execution.
- **Scalability**
The application is part of a SAAS platform, thus scalability is a must. This module is built with a microservice architecture in mind, so scalability came predefined.
- **Performance**
Batch jobs requires some important amounts of computational power, execution should be seamless, response time of the application must not bother the user in any way.
- **Event-driven**
Job executions might take too long to finish, so executions should be event-driven, using the Asynchronous pattern to manage response time and continuous availability when long-running job executions are required to participate in the application process.

2.4 Requirements' Modeling

For a better understanding of the specification of the required needs of the application, I opted for the realization of some Use Cases Diagrams to have a better understanding of the needs.

In our solution, we only have one actor, which is the administrator that will launch and monitor all the job batches.

2.4.1 General Use Case Diagram

In this section, I will introduce the main functions of our microservice through a use case diagram. The use case diagram defines the expected activities from the administrator with regard to the application.

Figure-2.1 below illustrates the main use case diagram.

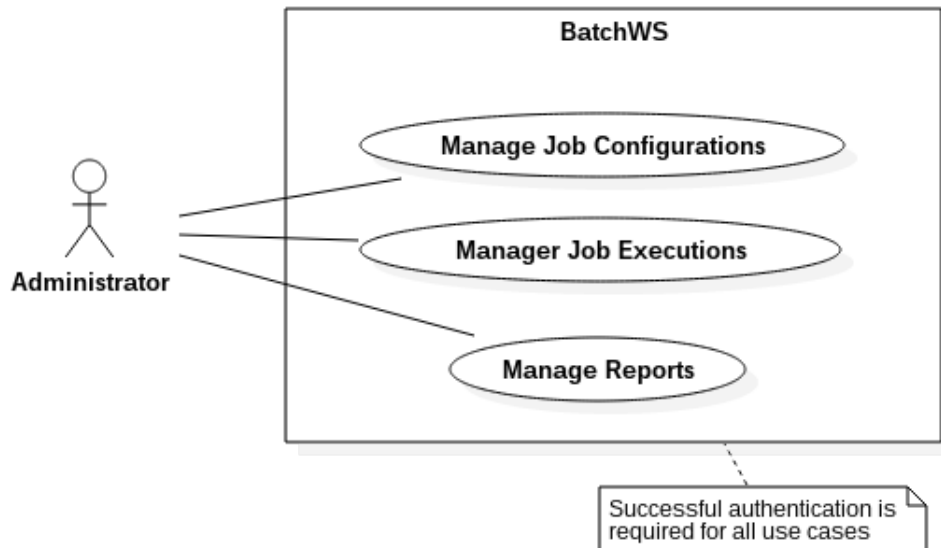


Figure 2.1: Main use case diagram

Description In this paragraph I will define each use case written in the diagram above, later in this chapter, I will go through each use case in a detailed diagram. The administrator has to be logged in, in order to execute any action.

- **Manage Job Configurations:**
Managing job configurations consist of creating configurations, by uploading the business jar, specifying different job batch configurations such as the data source, number of retries, etc.
- **Manage Job Executions:**
The administrator can also manage the job executions, by launching, stopping, restarting jobs, check job histories, download job executions logs, etc.
- **Manage Reports:**
After a job has been finished, the administrator can view the job execution in a detailed downloadable report, that he creates its template. He can also check the reports that has been generated during the writing step, if he chooses to during the creating of the job configuration. The administrator is the one responsible of creating the report templates and can even specify the template parameters for the detailed reports.

2.4.2 Detailed Use Case diagrams

As stated in the above paragraph, this section will detail the complex use cases, for a better understanding of the system.

2.4.2.1 Manage Job Configurations

Figure-2.1 and *Table-2.1* below represent respectively, the detailed diagram of the Job Configurations Management use case, and its description.

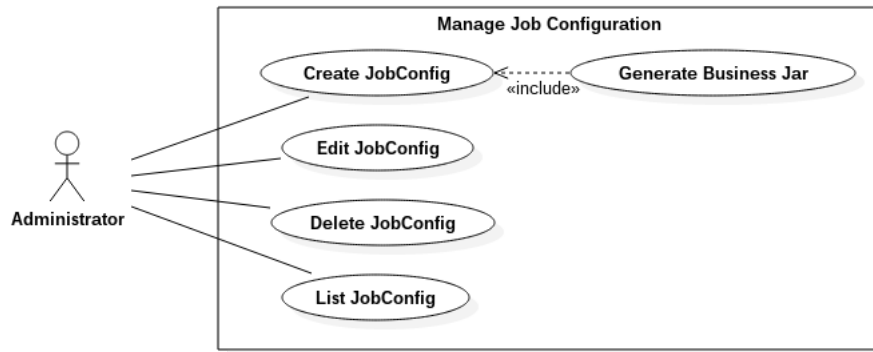


Figure 2.2: Manage Job Configuration Use Case Diagram

Table 2.1: Manage Job Configuration Use Case Description

Title	Manage Job Configuration Use Case.
Intention	Handle the Job Configuration management.
Actors	Administrator.
Pre-Condition	Successful authentication.
Sequences Definition	After the administrator is logged in, and after successfully implementing the template jar with his business logic, he can then upload this jar into the platform, when creating a job configuration, where he can also specify several other parameters that would be used when launching a job. He can edit or delete a job configuration at any given time.

2.4.2.2 Manage Job Executions

Figure-2.3 and *Table-2.2* below represent respectively, the detailed diagram of the Job Executions Management use case, and its description.

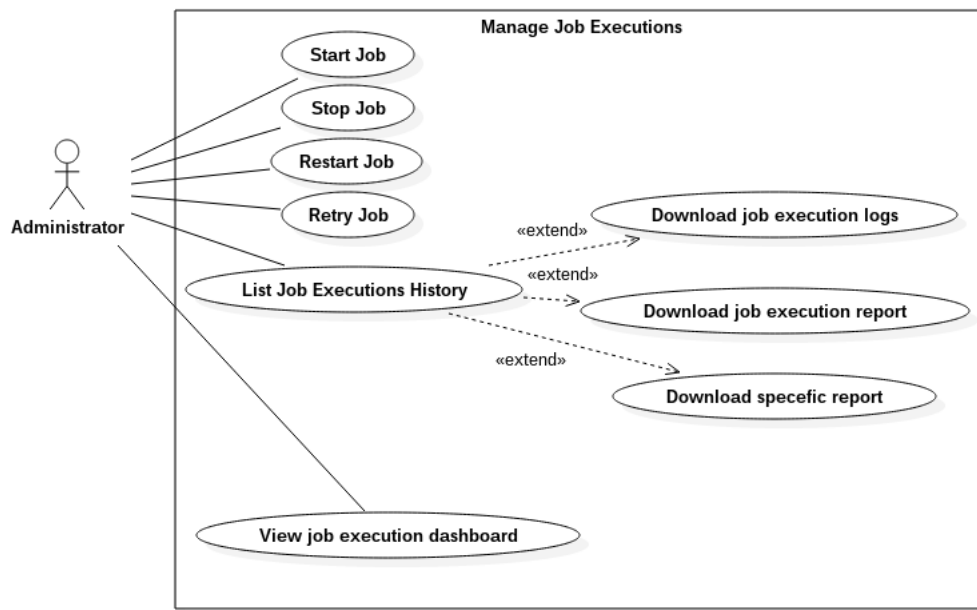


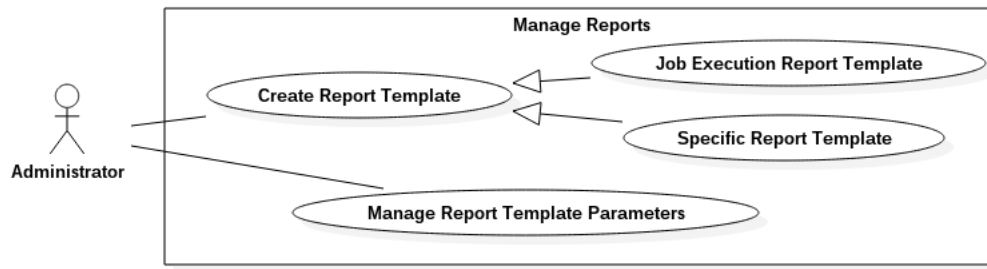
Figure 2.3: Manage Job Executions Use Case Diagram

Table 2.2: Manage Job Executions Use Case Description

Title	Manage Job Executions Use Case.
Intention	Handle the Job Execution management.
Actors	Administrator.
Pre-Condition	Successful authentication.
Sequences Definition	After creating a job configuration, the administrator can then use this newly created configuration to launch jobs. He can also monitor and manage these job executions at any given time. Managing consist of starting, stopping, restarting a job, a job can fail for any reason, in this case the administrator has the ability to retry this failed job. Results of a job execution can be acquired in several ways, from a dashboard containing some important stats to logs and report generation.

2.4.2.3 Manage Reports

Figure-2.4 and *Table-2.3* below represent respectively, the detailed diagram of the Manage Reports use case, and its description.

**Figure 2.4:** Manage Report Use Case Diagram**Table 2.3:** Manage Report Use Case Description

Title	Manage Report Use Case.
Intention	Handle the Report management.
Actors	Administrator.
Pre-Condition	Successful authentication.
Sequences Definition	For a better understanding of a job execution, an administrator can generate a report, that he created its template. There are two types of reports, Job Execution reports, and specific reports (generated during the writing step). For the latter, the administrator can specify the list of the parameters that would be injected, in contrast to the job execution report, which have a pre-defined parameters list.

Conclusion

In this chapter I went through the functional and non-functional requirements. Moreover, I made an analysis of these requirements by identifying drawing the global Use Case diagram and its description, along with the detailed diagram for each use case. In the next chapter, I'm going to build on top of this analysis, and specify the design of the application.

3 Design

Introduction

The phrase “software design” means the conception, invention, or contrivance of a scheme for turning a specification for computer software into operational software. Design is the activity that links requirements to coding and debugging. [McC04]

After a detailed specification of the requirements, we are hereafter capable of elaborating the complete design of the adopted solution. I clarify in this chapter the architecture adopted, followed by a global view of the class diagram. Subsequently, I will draw a sequence diagram showing the optimal execution of our scenario. Eventually, I will end this chapter by presenting the database representation.

3.1 Architecture

The goal of software architecture is to minimize the human resources required to build and maintain the required system. [Mar18] A well-thought-out architecture is the foundation stone of a successful software, hence the importance of architecture. Accretio architects decided that the Microservices approach is best suited for their needs, thus, the application I am building will be a piece in the Accretio puzzle.

Martin Fowler (a pioneer in software development and design related topics, and microservices specifically) defines the microservices architecture as “an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery.” [Fow18]

Figure 3.1 below represents the overall architecture, with some Accretio microservices, the one outlined in red is the microservice that I oversee, basically, most of the communication are done through HTTP, except for the communication between the running job, and the BatchWS microservice, which is done in socket to keep the BatchWS updated in real-time whenever the status of the job is changed.

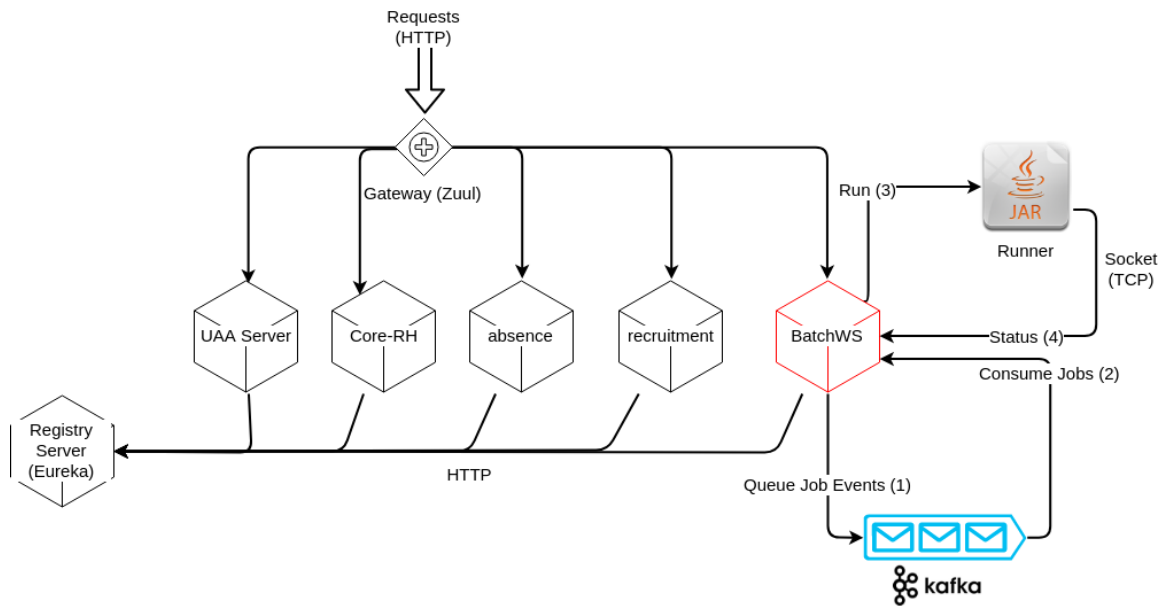


Figure 3.1: Architecture

3.2 Class Diagram

Class diagrams are one of the most fundamental diagram types in UML. They are used to capture the static relationships of your software; in other words, how things are put together. Class diagrams use classes and interfaces to capture details about the entities that make up your system and the static relationships between them. [BDP05]

Figure 3.2 below, represents the structural data of the application, in a class diagram form.

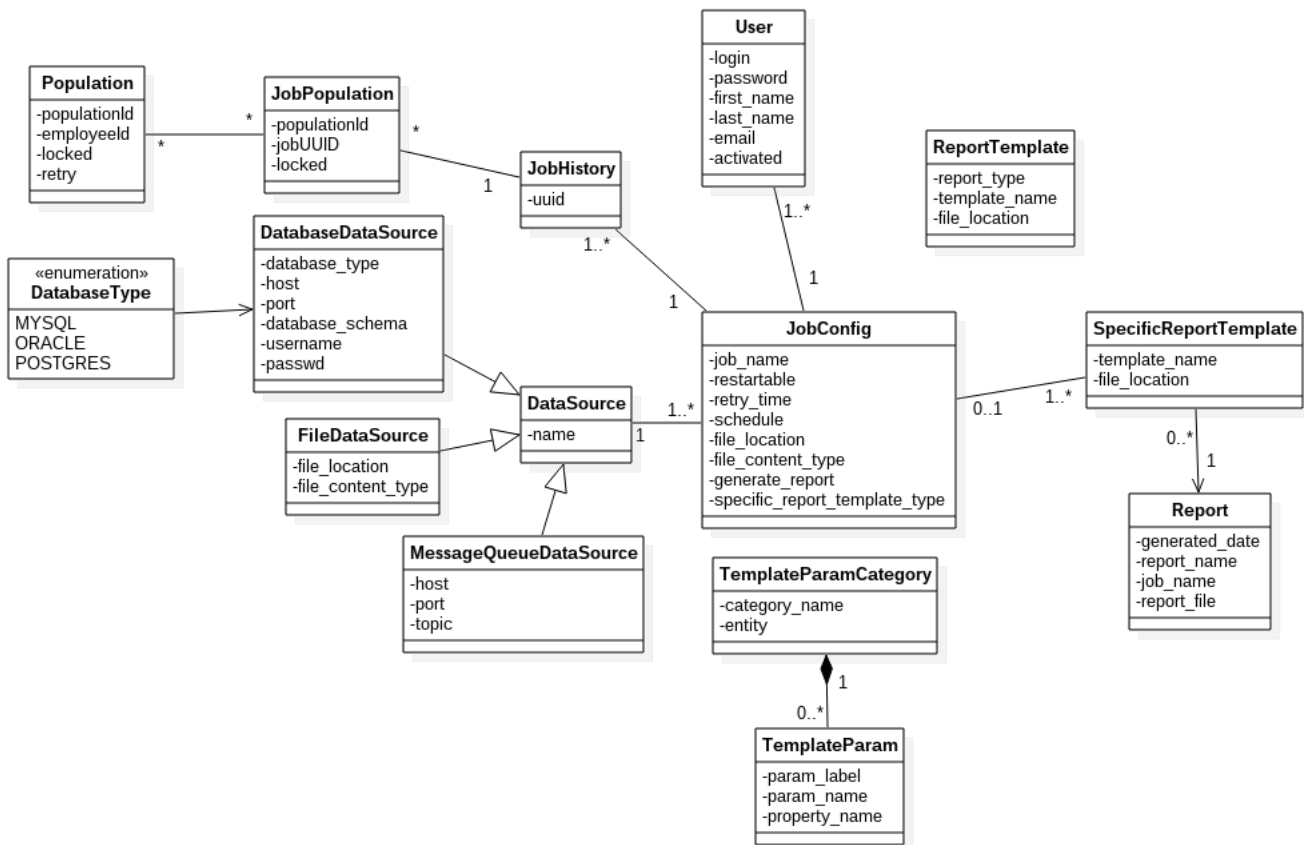


Figure 3.2: Class Diagram

Description This class diagram is composed of the following classes:

- **Job Config**
This is the main class, it holds the job batch configuration, and is used to launch the job.
- **DataSource**
This class defines the data source that will be using in a job, it can be either file, message queue or a database source, each variation have a different set of properties.
- **Job History**
Whenever a new job is launched, a UUID is generated to define this job execution, it is then stored in the job history, in order to track its history, to load the population for that specific job execution, or to generate and download a report.
- **Population**
Each job execution has its own population (Population in Accretio term, is

the combination of the employees that would go through the job batch process step), this population is populated during the reading step of the job batch. There should be only one job processing an employee (population) at a time, therefore, the “locked” flag is set to true, whenever a job started processing the employee, and set it to false when it’s done.

- **Job Population**
This class holds the list of populations for a particular job, a whole list of population can also be locked.
- **Report**
The job can generate for each processed employee a report (in case the administrator has stated so during the job config creation), using the Specific Report Template and the Template Param. Reports such as an employee payroll, annual evaluation reports, etc.
- **Specific Report Template**
This class holds a definition for the template that will be used to generate reports for the population, this template is wrote by the administrator, where he can put the template parameters as a placeholder, and these placeholders will be replaced by their real values during report generation.
- **Report Template**
Along to the Specific Report, the administrator can also generate reports for a whole job execution, however this report is not auto-generated, rather, the administrator can generate the report whenever he wishes to.

3.3 Sequence Diagram

Use cases allow your model to describe what your system must be able to do; classes allow your model to describe the different types of parts that make up your system’s structure. There’s one large piece that’s missing from this jigsaw; with use cases and classes alone, you can’t yet model how your system is actually going to its job. This is where interaction diagrams, and specifically sequence diagrams, come into play. [KH06]

Composite structures such as class diagrams, show how and what objects fit together to fulfill a particular requirement, while interaction diagrams such as sequences diagrams show exactly how those objects will realize it.

Figure 3.3 below represent the sequence diagram of the main functionality of the system, in an error-free scenario.

3.3 Sequence Diagram

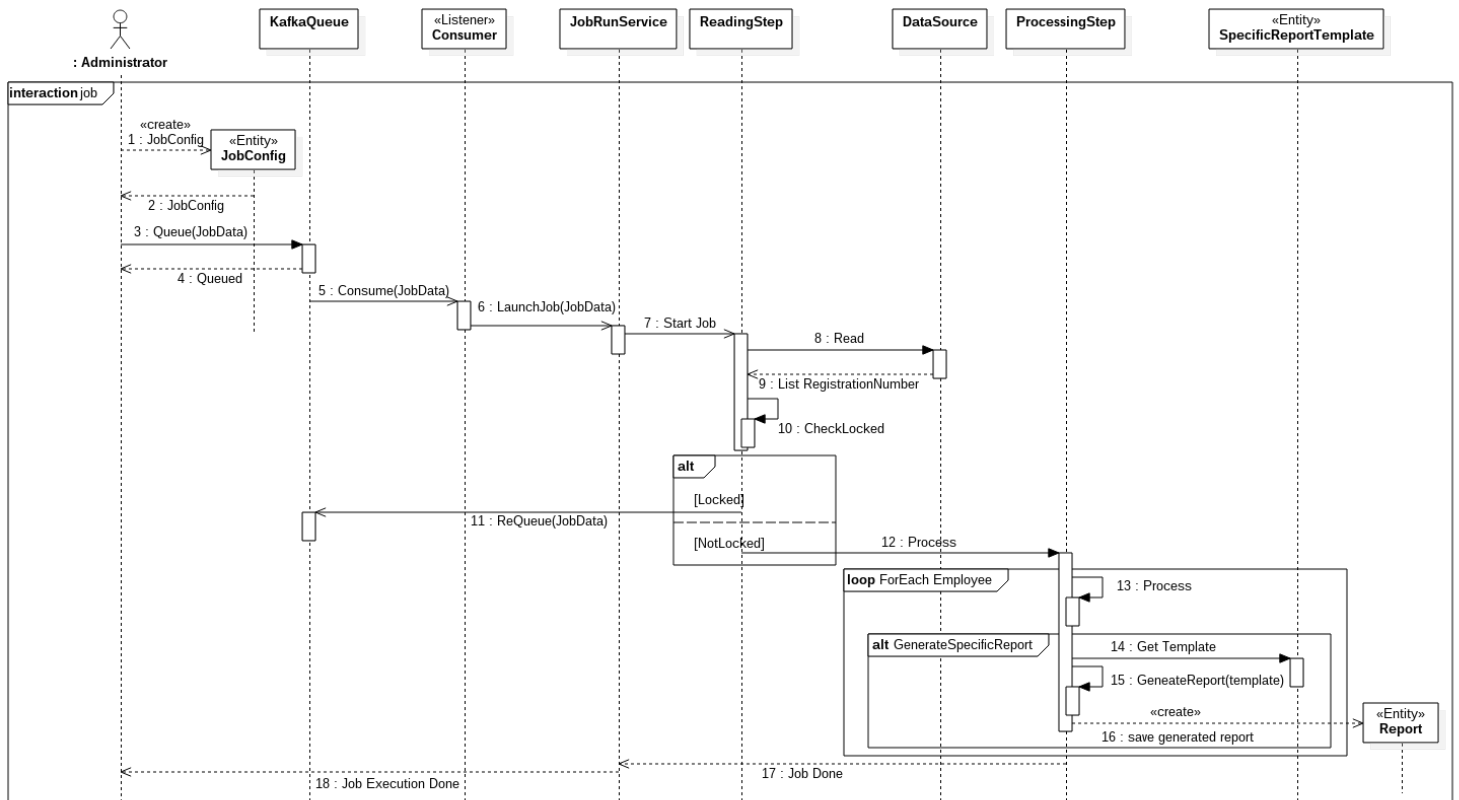


Figure 3.3: Sequence Diagram

3.4 Activity Diagram

To illustrate the life cycle of a job, I have opted in *Figure 3.4* below, to use an activity diagram that shows the different status a job can have.

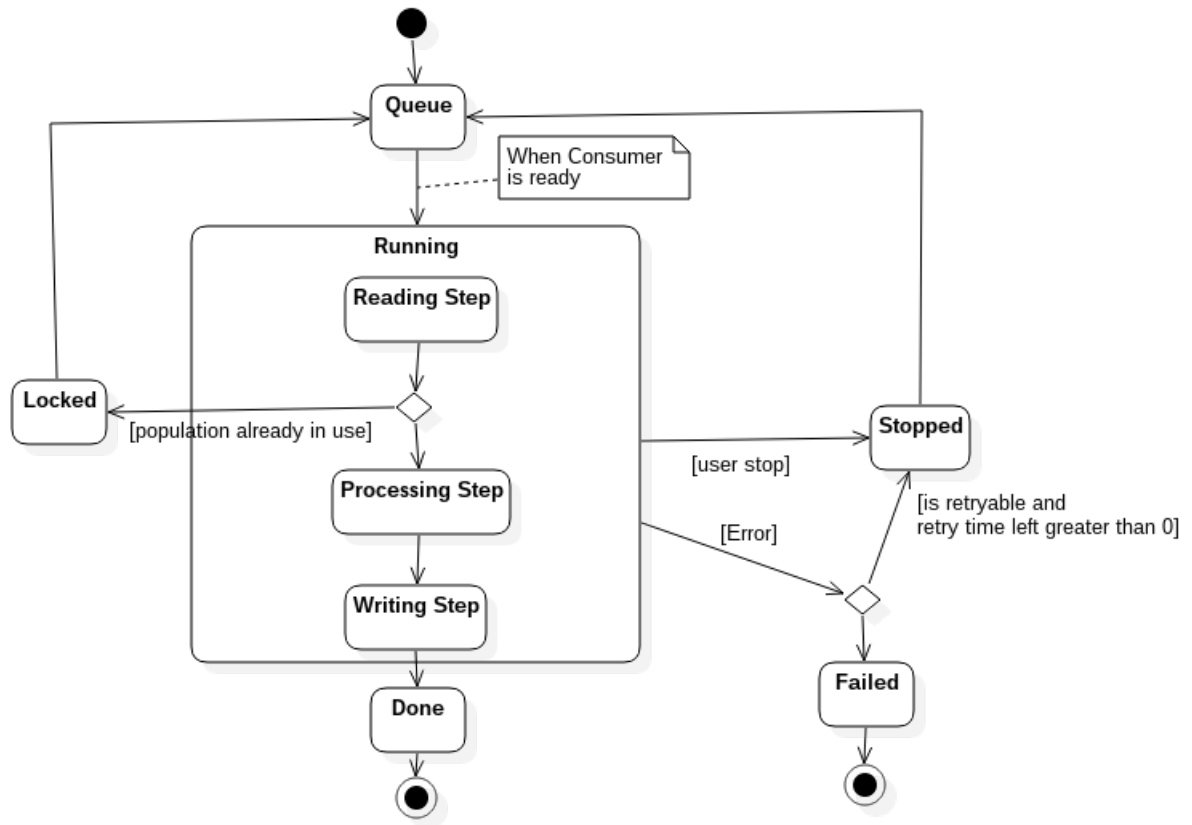


Figure 3.4: Activity Diagram Job Life Cycle

Conclusion

In this chapter I have went through the application architecture, along with a detailed representation of the solution, using class, sequence and activity diagram.

4 Implementation

Introduction

This chapter discusses the implementation of the microservice components. I will begin by presenting the development environment on which this project was developed, presenting the technologies used, finally I will present key aspects of the solution done alongside some screenshots of the main parts of the application.

4.1 Development Environment

Numerous Technologies have been used in order to successfully achieve this work, in this section I will go briefly on the most important ones;

IntelliJ IDEA have been chosen as the Integrated Development Environment (IDE) for its remarkable support for both Java and Angular framework, which in turn boosts productivity.

As for the database, I had to use both *MongoDB*, and *MySQL*. *Accretio* is built upon *MongoDB*, all the employees population is stored in *MongoDB* instance, in addition, *Spring Batch* does not support non-relational databases, thus the use of *MySQL*, along with *MongoDB*.

For the backend, I have used *JHipster* to generate the microservice skeleton, which generates a *Spring Boot Application*. The *Spring Cloud* framework makes it very easy to deal with a microservices application, that with the powerful *Spring Framework* for Dependency Injection, *Spring Security* for authentication and authorization, *Spring MVC* for RESTful APIs, *Spring Data JPA/Mongo* to deal with entities, *Spring Integration* for a reliable communication between the Job execution and the main microservice, and of course *Spring Batch*. In addition to several other Spring projects, but overall, the whole backend technology stack is based on *Spring* technologies.

As for the frontend, *Angular 4* was used, with the rich UI components library *PrimeNG*, that provided several out-of-the-box UI components.

In order to maintain an asynchronous communication, *Kafka* was used as the broker for our asynchronous messages, along with the *Avro* serialization framework.

Finally, *ELK* stack (Elasticsearch, Logstash, Kibana) had been setup as a mean to go through and visualize the logs that has been generated by both the microservice, and the executed batch jobs.

4.2 Solution

The following section will contain detailed information about the adopted solution, starting by spring batch integration, listing the different configuration possible, a walk-through of the reporting engine and logs management. Finally, I'll take a few screenshots representing the final outcome.

4.2.1 Spring Batch Steps

As described in the first chapter, the solution is composed of several steps, some steps are defined by the customer, while others are pre-coded. I have divided the solution into two steps:

- A step for reading the employees population, and store them into a temporary database, this step is itself divided into 3 other phases following the ETL (Extract, Transform and Load) style:
 - Reading phase (Extracting): This is where the client business logic (for reading) is injected, when configuring the job, he can choose from three datasource types: JDBC, File (Excel, CSV, ...) and a messaging queue. Upon executing the reading method, the datasource would be built (using the configuration that the client has setup) and given to the client to be used in his implementation. The expected output of this phase is a list of registration numbers for the employees that would be processed by the batch.
 - Processing Phase (Transform): This phase is already written, giving the output of the previous phase, a new object (population) is constructed for each registration number.
 - Writing Phase (Load): At the end, we need to store the processed population objects, into a temporary database, for two reasons, the first is to pass this data to the next step, and the second is a to lock that population so that future jobs cannot access it and also to check if it is already locked. In that case, the whole job would be marked as Locked, and is re-queued until to lock is released.
- A step for processing the employees that have been read by the first step, same as the previous one, it is divided into 3 phases:
 - Reading phase: Reads the registration numbers that would be process off of the temporary database.

- Processing phase: This is the most essential part of the batch implementation. This is where the processing is done and where the client processing business logic is injected. The input that has been read in the previous phase is used as an input, while the output is totally set and controlled by the client.
- Writing phase: This phase is also handled by the client, after the processing phase is finished, the client can choose to manipulate the data as he wishes to, such as storing it into file system, save logs, etc. In addition to that, the client is provided with a rich templating engine, where he can generate for each processed employee a report, with a template of his choosing. This report would be save into our system and can be viewed/downloaded at demand.

Figure 4.1 below (taken from *chapter 1*) provide more insight to the solution and detailing the work flow of the inputs and outputs.

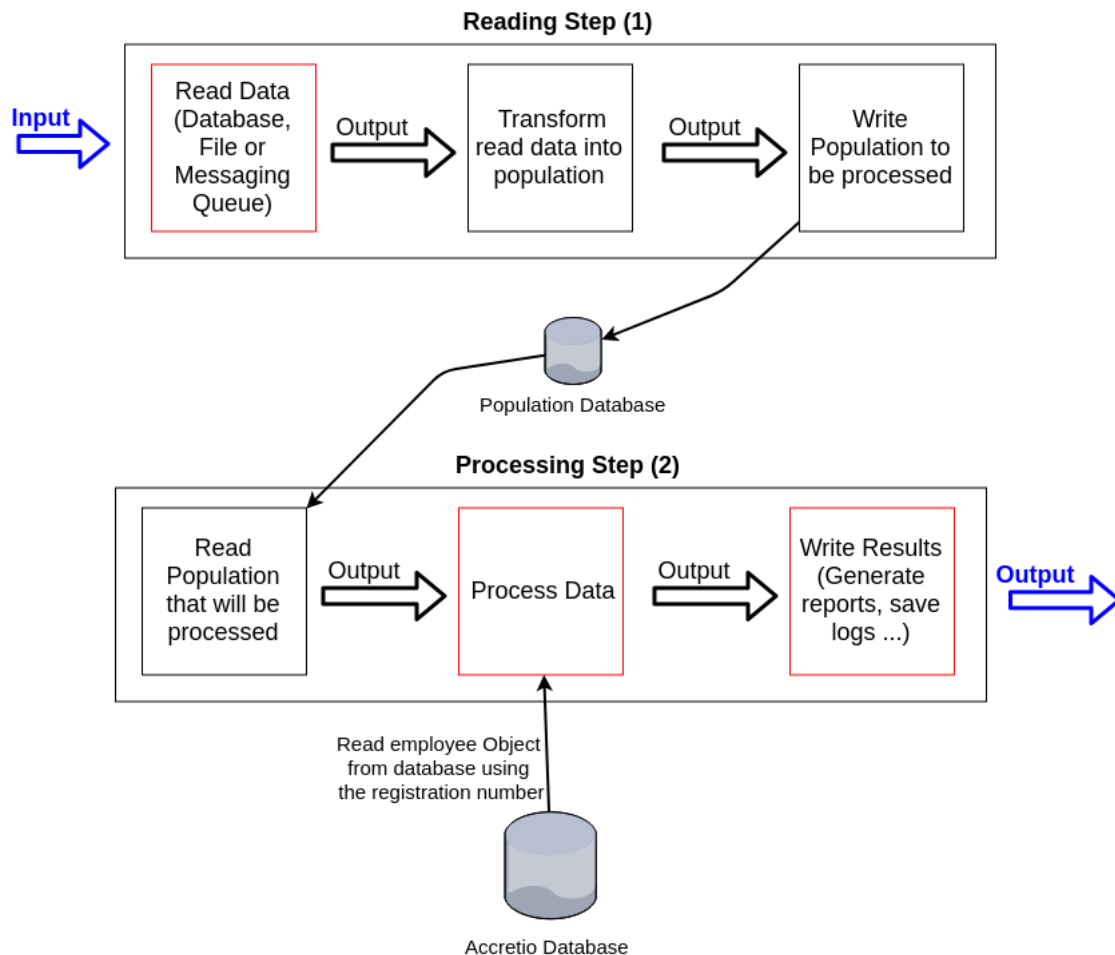


Figure 4.1: Graphical Representation of the Solution

4.2.2 Asynchronous Communication

There is a limit on how many Job Batches that can run simultaneously, limitations are both technical and functional. Batches are resource demanding tasks, running too many jobs would result in a very slow system response time, or even system failure. Functionally, jobs that share the same population cannot run concurrently. Therefore, I used a queue (Kafka), to queue the job execution demands, I also control how many concurrent job the system could run, by specifying the number of Kafka consumer threads. Job executions that are queued are of two types: new execution and restarting a stopped, failed or locked execution.

As stated in the first section of this chapter, I am using Avro serialization system with Kafka, to serialize and deserialize the queued events.

Apache Kafka was chosen for its highly distributed, partitioned and replicated architecture, which would fit best with our SaaS model. Kafka was designed with distribution in mind, unlike other Asynchronous Communication systems, which makes it scalable, elastic and resilient.

Figure 4.2 below, shows a comparison between Apache Kafka and some other well-known brokers. As represented in the figure, Kafka can handle up to 35000 messages per second which is almost the double capacity of the HornetQ.

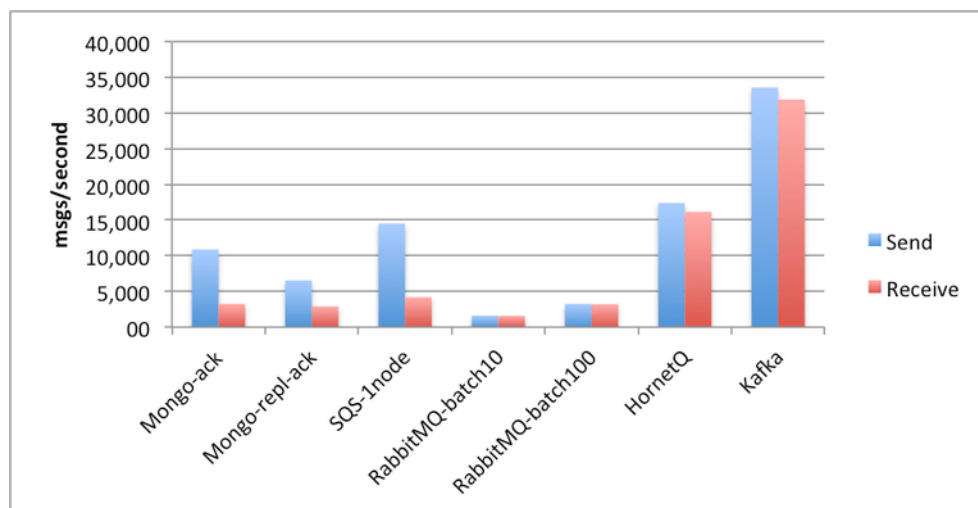


Figure 4.2: Kafka Performance vs others [iT16]

Kafka was also used for its great support with Spring, works on JVM, and for its maturity, Kafka is 6 years old, and used by major IT companies.

As for the Avro, it was chosen for mainly two aspects (the serialized object that would be sent to Kafka is called datum):

- **Schema Evolution:** When Serializing or Deserializing, Avro uses a pre-defined schema, however, Avro support schema evolution (missing, extra or modified fields)

- **Untagged Data:** The format of an Avro datum is binary, unlike other format such as JSON or XML, the datum is written without overhead describing each field length or type, instead this information is extracted from the schema. This process renders the size of the datum extremely small.

4.2.3 Configuration

The requirement was to externalize as much configuration as possible, without having to change the code and re-compile the whole microservices. Consequently, I have divided the configuration into two types, “System” configuration which controls several variables of the microservice, and “Batch” configuration that defines the variables required to run a job. Next, I will list all the properties for each configuration type:

- **System**
 - **uploaddir:** Directory where all user uploaded files reside (Reports, Data files, etc)
 - **dirjar:** Directory where all user uploaded jars reside
 - **dirbatchrunner:** Location of the Batch Runner jar
 - **launcherthreads:** Number of threads for the Kafka consumer
 - **logsdire:** Places where all the batch logs are saved to.
 - **Kafka:**
 - * **topic:** The name of the Kafka topic to be used
 - * **host:** The hostname of the Kafka broker
 - * **port:** The port for the Kafka broker
- **Batch**
 - **job**
 - * **name:** A unique identifier to represents the name of the job
 - * **scheduled:** boolean flag to state whether this job would be scheduled or not
 - * **schedule-time:** if the scheduled flag is set to true, this parameter states the time of the execution
 - * **execution-id:** when restarting a job, this parameter identifies the previous job execution that has been stopped.
 - * **chunk-size:** Chunk size to pass to the writer
 - **populationId:** the ID of the population (in the temporary database) to be used, case of a job restart

- retrieable: states if in case of a failure (in any of the steps), the job should retry or not.
- maxretry: number of time to retry until failure
- datasource
 - * type: Enum represents the type of the datasource (JDBC, File, Message)
 - * driver-class-name: if the type is JDBC, this parameter represent the database driver, currently, we are supporting MySQL
 - * url: url of the database (hostname, port and schema)
 - * username: Username of the database
 - * password: Password of the database
 - * file: if the type is File, this represents the location of the file to be read
 - * message: if the type is Message:
 - host: Represent the hostname of the Kafka broker
 - port: Port of the Kafka broker that is listening to
 - topic: Name of the topic to read the data from
- specific-report-template-id: The id of the template that would be used to generate the specific report for each processed employee
- specific-report-template-type: The type of the document that would be generated (PDF, DOC, HTML)

4.2.4 Report templating & generation

Another important part of the project is template managing and report generation. Basically, we have two types of reports, a global summary report for a job execution, and a specific report that is generated in the final writing phase, for each processed employee.

The user of the application has total control of what fields, text or even images that would be written in the report, using a powerful templating system. With the help of a document designer, the user can either write or design the report in the web application, or upload a pre-written document to the application, however the uploaded document has to be of ODT format.

The document designer is an open source component, based on the WebODF library (Open Document Format), and is called “Wodo.TextEditor”. WebODF is a JavaScript library that makes it easy to add Open Document Format (ODF) support to your website and to your mobile or desktop application. It uses HTML and CSS to display ODF documents. [Gmb]

I had several options to choose from for a Rich Text Editor such as QuillJS or CKEditor, however, I ended up choosing Wodo.TextEditor for its great support for office document, and the user-friendly UI that resembles to the popular editors such as Google Docs and Microsoft Office.

Another drawback of the editors like CKEditor, is that they produce HTML and CSS output, the resulting styles and formatting are never consistent, whereas ODT is a standard, and the output is always the same.

After finishing the design of the report template, the user can then add which fields he would like to come up when generating the report. A placeholder is placed in the template, indicating the location of field, during the generation phase, the report is scanned for all the placeholder, and they are automatically replaced with their respected values, as described in *figure 4.3* below.

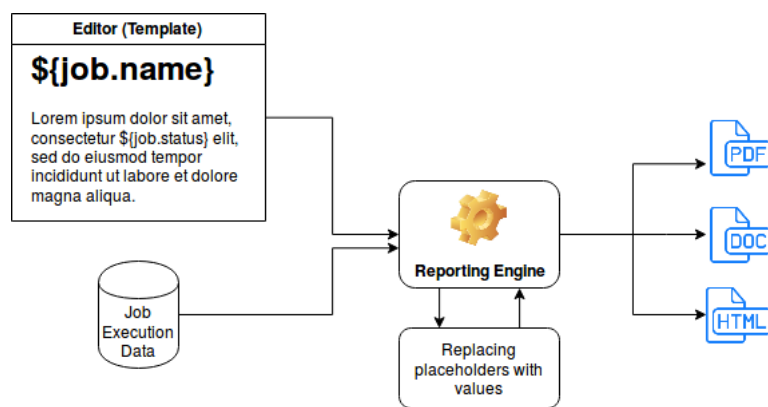


Figure 4.3: Reporting Engine

The key difference between a general report and a specific report, is the parameters/fields that are available to use, while in the general report they are known beforehand and listed next to the editor, upon clicking on any variable on that list, it would be placed in the template. However, the specific report does not have a pre-defined list, and the client have to write them up himself. The mapping of the variable to their respective values, is done at the writing phase, where the client must provide an implementation for the “getTemplateParamMapping” method, which should return a map of String (name of the variable) and String (name of the field in the object that have been returned from the processing phase).

I will present screenshots in the section that presents the two editors for the two types of the reports.

4.2.5 Logs management

Businesses must review their logs daily to search for errors, anomalies, or suspicious activity that deviates from the norm. [Glo16] Thus we need to provide a way to the

client and to our system administrator, to review and check logs. However, we have two types of logs, logs related to the microservice, and the job execution logs.

For the system administrator, I have opted to setup and configure an ELK (Elasticsearch, Logstash, Kibana) stack. The microservice sends the logs to be stored in the Elasticsearch storage system using sockets through Logstash, while the job execution logs are first stored in the file system, afterwards, Logstash reads those logs and store them in Elasticsearch. This way we can retrieve the log files for each job execution for the client, whenever a request is made to view a log file for a particular job execution.

Some Kibana dashboards have been setup for the administrator for fast log monitoring.

4.3 Overview of the achieved work

In this section, I am going to examine the achieved work through several screenshots of the application, starting with the login page.

Once a user has opened the application, he will be prompt with a login modal, Figure 4.4 below represents the login interface.

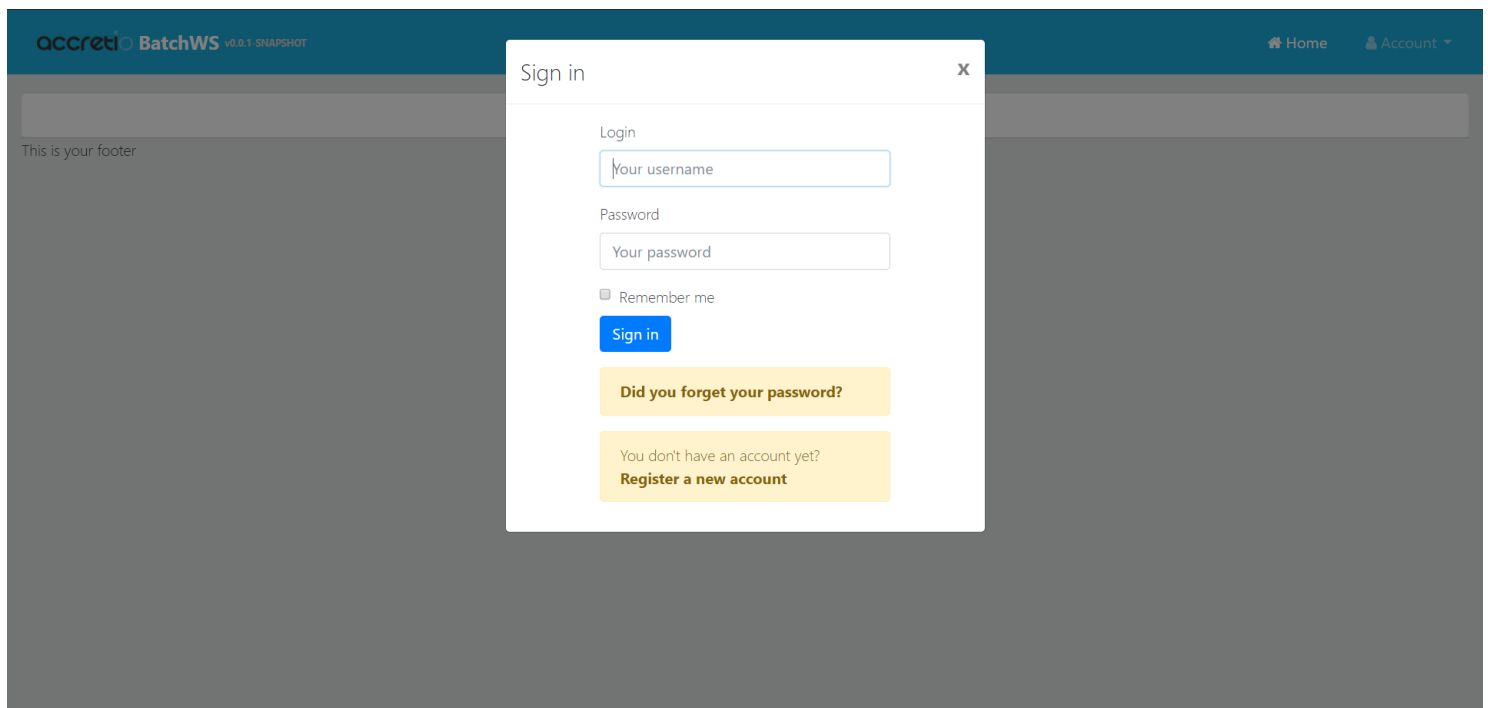


Figure 4.4: Login interface

4.3 Overview of the achieved work

Upon successful login, the user would then be redirected to the dashboard, where he can view a set of charts, going from top to right, the number of job executions per month, a pie chart representing job status and their count, and the number of report generated for each template.

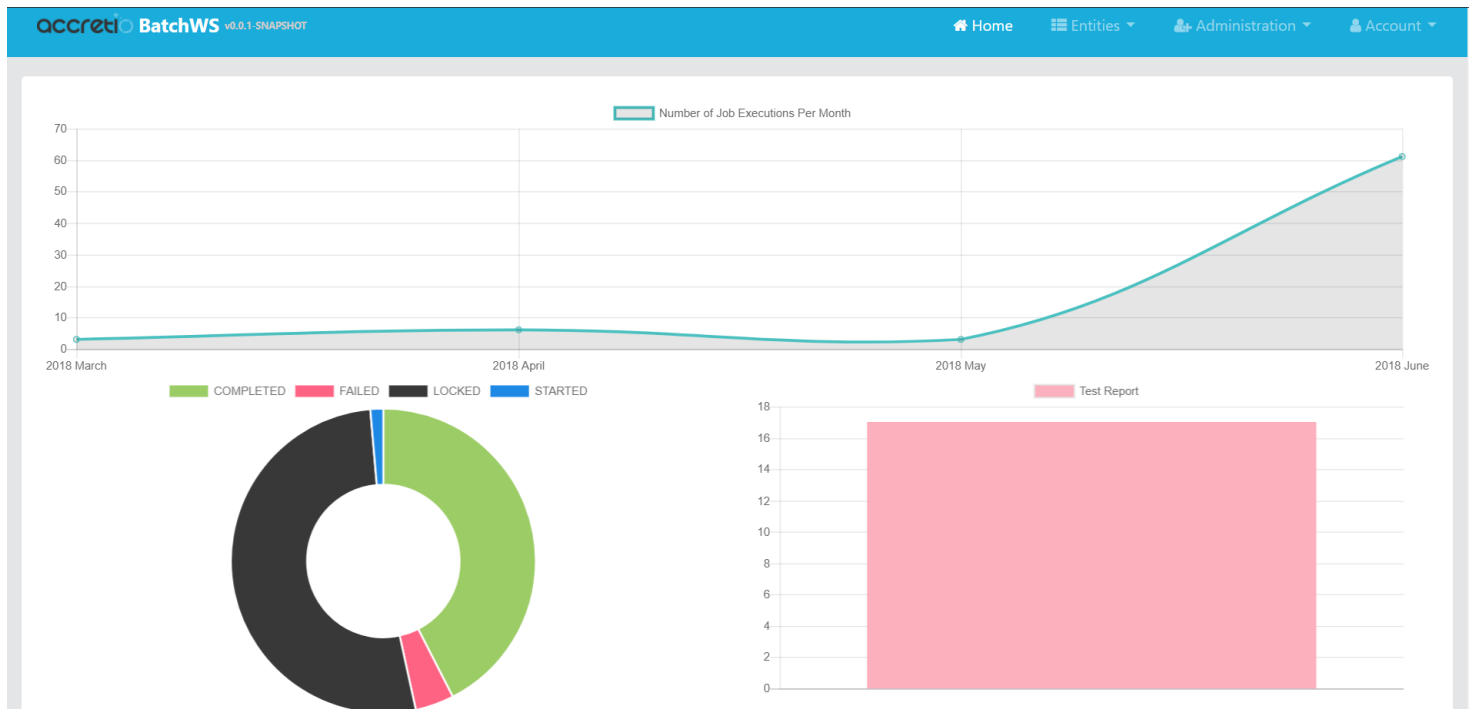


Figure 4.5: Dashboard interface

The primary objective of the application is to configure and monitor job execution, so in the figure and figure below, I represent the job configuration interface, where the user can enter, the job configuration name, whether the job should attempt a retry in case of failure, and for how many times. Another option is to schedule the job to be executed at a certain time. The user can also instruct the job batch to generate a specific report and select which template will be used, and the report type. Afterwards, the user has to upload the Jar which contains his business logic, and set the datasource.

accrati BatchWS v0.0.1 SNAPSHOT

Create or edit a Job Config

Job Name
Kafka

☒ Retry Time
5

☐ Scheduled
mm/dd/yyyy --:-- --

☒ Generate Report
Test Report

Report type
PDF

Jar File
KafkaClientBusiness-1.0-SNAPSHOT.jar, 7 139 bytes

Choose File KafkaClientB...NAPSHOT.jar

Use Old Datasource

Data Source

Job Configs

	Job Name	Restartable
🔍	jdbc	false
🔍	csv	false
🔍	csv2	false

This is your footer

+ Create new Job Config

Launch Job View Edit Delete

Launch Job View Edit Delete

Launch Job View Edit Delete

Figure 4.6: Creating a Job Configuration 1

KafkaClientBusiness-1.0-SNAPSHOT.jar, 7 139 bytes

Choose File KafkaClientB...NAPSHOT.jar

Use Old Datasource

Data Source

Data Source Type: Database File Messaging Queue

Datasource Name
Kafka

Host
localhost

Port
9021

Topic
kafka-topic

Cancel Save

Job Configs

	Job Name	Restartable
🔍	jdbc	false
🔍	csv	false
🔍	csv2	false

This is your footer

+ Create new Job Config

Launch Job View Edit Delete

Launch Job View Edit Delete

Launch Job View Edit Delete

Figure 4.7: Creating a Job Configuration 2

4.3 Overview of the achieved work

After creating the Job Configuration, the user can now launch and monitor jobs, he can also control the job by either, stopping it, restart or retry the job in case a failure has occurred. Another feature is he can also download the logs for that job execution.

The screenshot shows the 'Job Configs' page in the BatchWS application. The top navigation bar includes 'Home', 'Entities', 'Administration', and 'Account'. A '+ Create new Job Config' button is in the top right. The main table lists job configurations with columns: Job Name, Restartable, Retry Times, Schedule, File, Generate Report, and Data Source. Below this, a detailed view of a job execution is shown with columns: status, Created Time, Start Time, End Time, and Exit Status. The 'Exit Status' column contains a dropdown menu with options: Report, Stop, Restart, Retry, and Logs. The 'Report' option is selected, showing a list of reports with columns: status, Created Time, Start Time, End Time, and Exit Status. The 'Exit Status' column contains a dropdown menu with options: Word, PDF, and HTML.

Job Name	Restartable	Retry Times	Schedule	File	Generate Report	Data Source	
jdbc	false			1528056133JdbcClientBusiness-1.0-SNAPSHOT.jar	false	jdbc	Launch Job View Edit Delete
csv	false			1528056149CsvClientBusiness-1.0-SNAPSHOT.jar	false	my first csv	Launch Job View Edit Delete
csv2	false			1528056445CsvClientBusiness-1.0-SNAPSHOT.jar	true	my first csv	Launch Job View Edit Delete

status	Created Time	Start Time	End Time	Exit Status	
COMPLETED	Apr 10, 2018, 9:34:25 PM	Jun 3, 2018, 9:34:25 PM	Jun 3, 2018, 9:34:31 PM	COMPLETED	Report Stop Restart Retry Logs
COMPLETED	Apr 16, 2018, 9:07:35 PM	Jun 3, 2018, 9:07:35 PM	Jun 3, 2018, 9:07:40 PM	COMPLETED	Report Stop Restart Retry Logs

Job Name	Restartable	Retry Times	Schedule	File	Generate Report	Data Source	
Kafka	true	5		1528071411KafkaClientBusiness-1.0-SNAPSHOT.jar	true	Kafka	Launch Job View Edit Delete

Figure 4.8: Job Batch Monitoring

The user can also download or view the general and specific report, from the same interface, which is represented in figure 4.9 below.

The screenshot shows the 'Report List' page in the BatchWS application. The top navigation bar includes 'Home', 'Entities', 'Administration', and 'Account'. A '+ Create new Job Config' button is in the top right. The main table lists reports with columns: Job Name, Restartable, Retry Times, Schedule, File, Generate Report, and Data Source. Below this, a detailed view of a report is shown with columns: status, Created Time, Start Time, End Time, and Exit Status. The 'Exit Status' column contains a dropdown menu with options: Report, Stop, Restart, Retry, and Logs. The 'Report' option is selected, showing a list of reports with columns: status, Created Time, Start Time, End Time, and Exit Status. The 'Exit Status' column contains a dropdown menu with options: Word, PDF, and HTML.

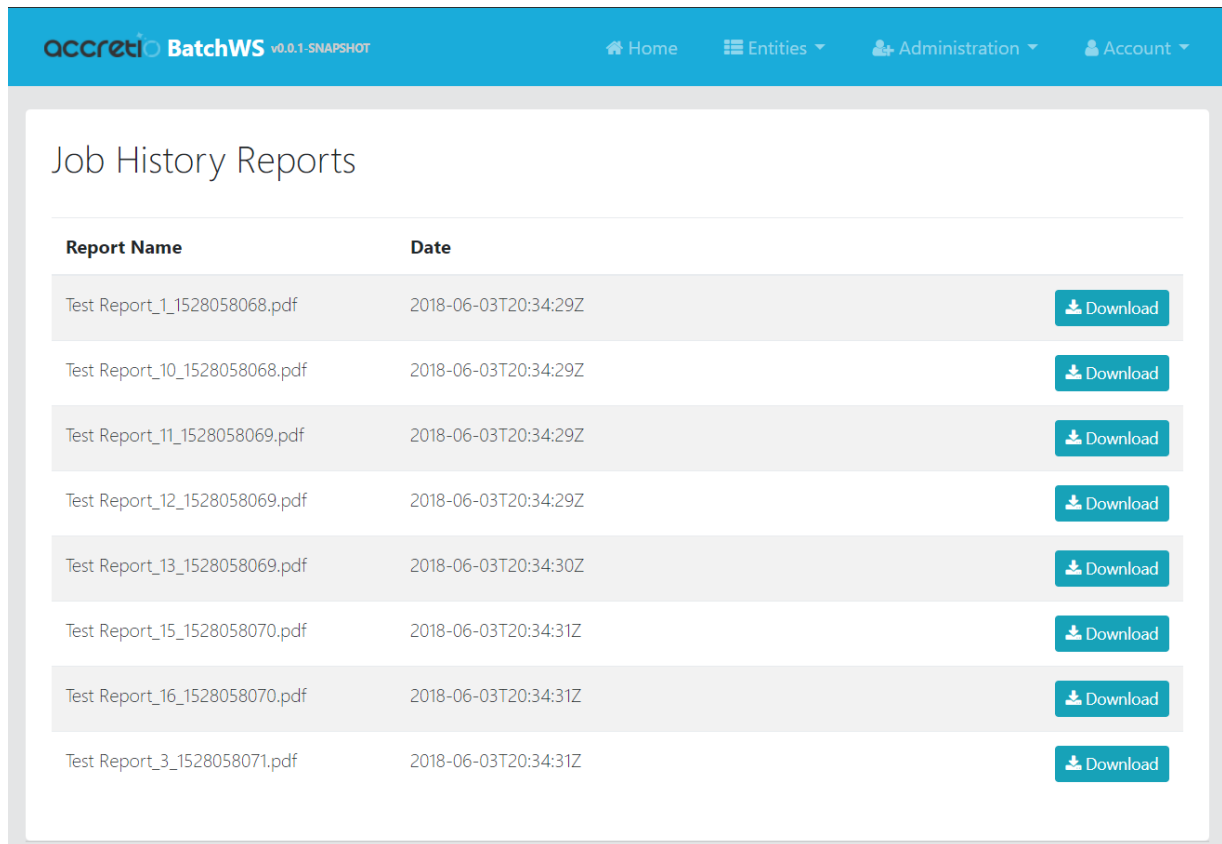
Job Name	Restartable	Retry Times	Schedule	File	Generate Report	Data Source	
my first csv	true					my first csv	Launch Job View Edit Delete

status	Created Time	Start Time	End Time	Exit Status	
COMPLETED	1 PM			COMPLETED	Report Stop Restart Retry Logs
COMPLETED	0 PM			COMPLETED	Report Stop Restart Retry Logs

Job Name	Restartable	Retry Times	Schedule	File	Generate Report	Data Source	
Kafka	true					Kafka	Launch Job View Edit Delete

Figure 4.9: Report List

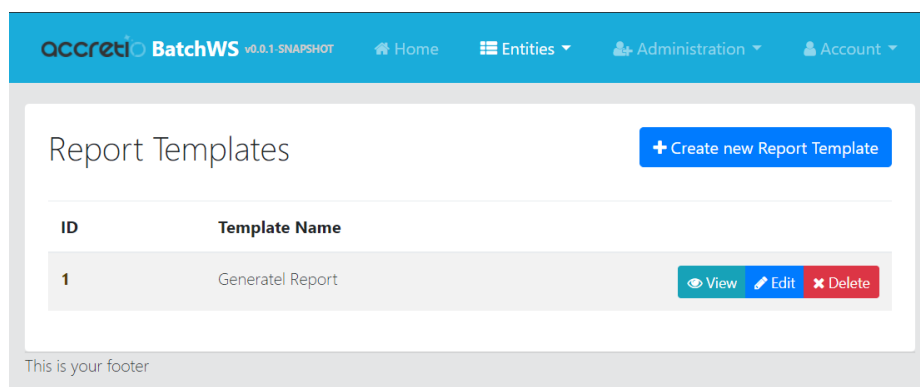
When selecting the “Specific Report” option, the user will be forwarded to the list of reports that have been generated, for each processed employee, with the ability to download each report, as seen in Figure 4.10 below.



Report Name	Date	
Test Report_1_1528058068.pdf	2018-06-03T20:34:29Z	Download
Test Report_10_1528058068.pdf	2018-06-03T20:34:29Z	Download
Test Report_11_1528058069.pdf	2018-06-03T20:34:29Z	Download
Test Report_12_1528058069.pdf	2018-06-03T20:34:29Z	Download
Test Report_13_1528058069.pdf	2018-06-03T20:34:30Z	Download
Test Report_15_1528058070.pdf	2018-06-03T20:34:31Z	Download
Test Report_16_1528058070.pdf	2018-06-03T20:34:31Z	Download
Test Report_3_1528058071.pdf	2018-06-03T20:34:31Z	Download

Figure 4.10: List of Generated Specific Reports

The use can also view the list of general templates that he have created, the same thing applies for the specific report template, as shown in Figure 4.11.



ID	Template Name	
1	Generatel Report	View Edit Delete

This is your footer

Figure 4.11: List of Report Template

4.3 Overview of the achieved work

Figure 4.12 details how a report template is designed by the user, where he can select which fields would be injected into the generated report, note that the list of parameters can also be edited.

The screenshot shows the 'General Report Editor' interface. At the top, there's a blue header with 'accretio BatchWS v0.0.1-SNAPSHOT' and navigation links: Home, Entities, Administration, and Account. Below the header, the interface is divided into several sections. On the left, there's a sidebar with 'Available Parameters List' containing two expandable sections: 'Job Config' and 'Job History'. The 'Job Config' section is currently expanded, showing parameters like Job Name, Retryable, Retry Time, Schedule, Generate Report, Start Time, Created Time, Ended Time, Last Updated, Status, ExitStatus, and File Location. The 'Job History' section is also expanded, showing parameters like Start Time, Created Time, Ended Time, Last Updated, Status, ExitStatus, and File Location. In the center, there's a 'Template Name' field with the value 'My General Report'. To the right of this field are 'Save' and 'Cancel' buttons. Below the 'Template Name' field, there's a rich text editor with a toolbar. The editor contains the following text: 'Job Name: \${job_name}' followed by a table with four rows: 'Started: \${start_time}', 'Ended: \${end_time}', 'Status: \${status}', and 'Exit Status: \${exit_status}'. The accretio logo is visible at the bottom center of the editor area.

Figure 4.12: General Report Editor

The specific report template is almost identical, except that there is no available parameter list, however, the user has to write his own parameters, which he should specific during the creating of the Jar.

The screenshot shows the 'Specific Report Editor' interface. At the top, there's a blue header with 'accretio BatchWS v0.0.1-SNAPSHOT' and navigation links: Home, Entities, Administration, and Account. Below the header, the interface is divided into several sections. On the left, there's a sidebar with 'Insert a Parameter:' and a text input field containing 'custom_salary'. Below this field is a green '+ Add' button. In the center, there's a 'Template Name' field with the value 'Salary Report'. To the right of this field are 'Save' and 'Cancel' buttons. Below the 'Template Name' field, there's a rich text editor with a toolbar. The editor contains the following text: 'Salary Report' followed by a paragraph: 'This is to certify that Mr. / Miss/ Mrs. \${custom_param_name} (Name of Employee) \${custom_param_id} (Employee #) is working with our esteem organization / company under the title of \${custom_position} (Title of employee) has earned [redacted] (salary). We found this gentleman fully committed to his/her job and totally sincere toward this organization / company.' followed by another paragraph: 'We are issuing this letter on the specific request of our employee without accepting any liability on behalf of this letter or part of this letter on our organization / company.'

Figure 4.13: Specific Report Editor

The resulting report of the previous template when generated looks like the following figure:

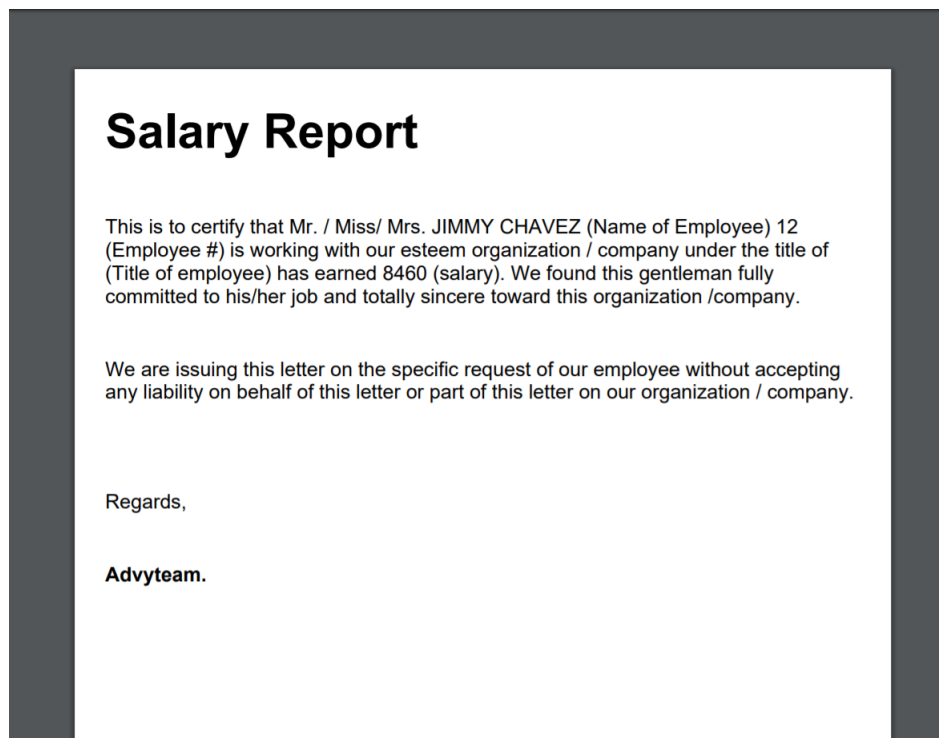


Figure 4.14: Example of a Generated Report

Furthermore, the user can view the job history in a separate window, with a more details.

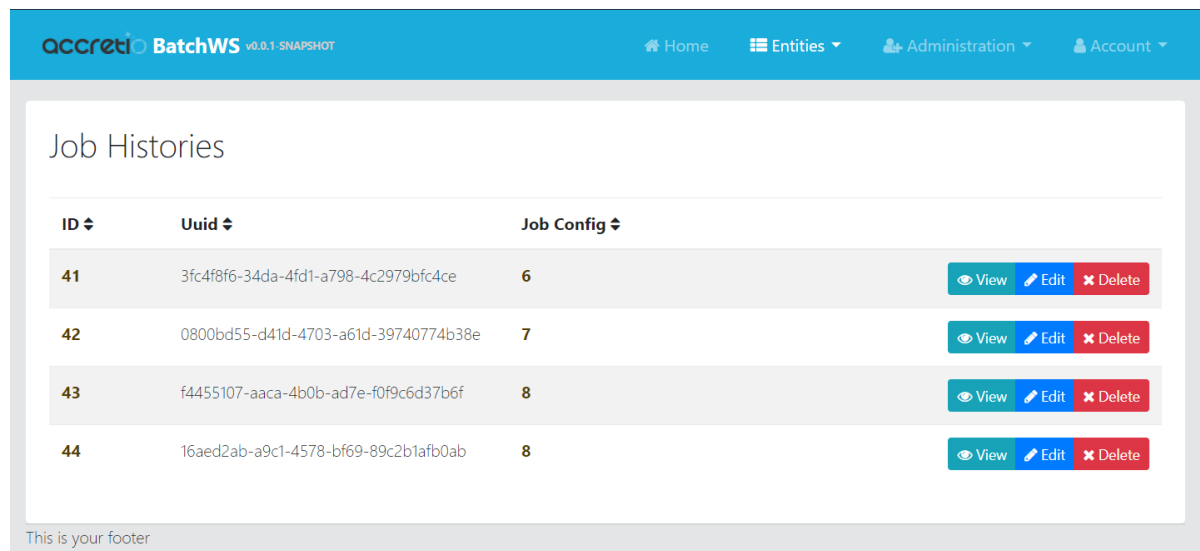


Figure 4.15: Job History

4.3 Overview of the achieved work

The parameters list of the General Report Template can be configured, by editing two collections, the Parameters Category, and The Parameters, which are shown respectively the figures below.

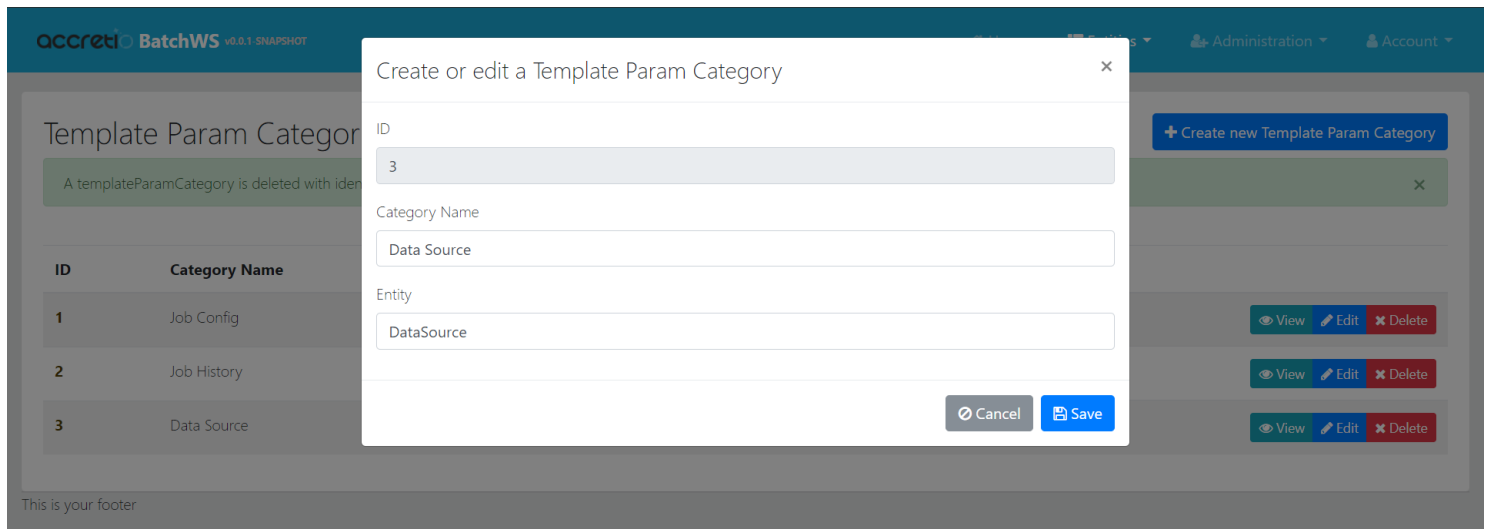


Figure 4.16: Managing Report Parameters Category

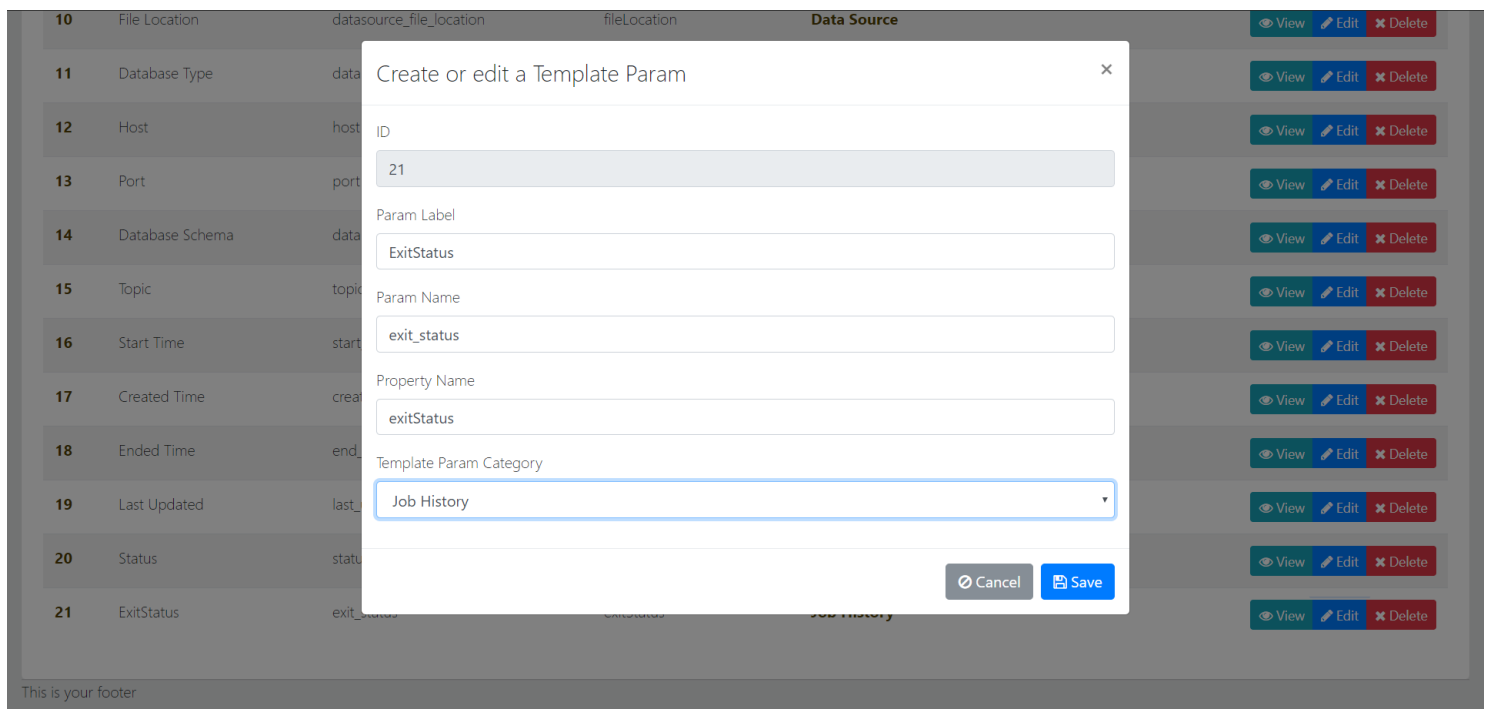


Figure 4.17: Managing Report Parameters

Finally, we end this section with a Kibana dashboard that represents different logs information.

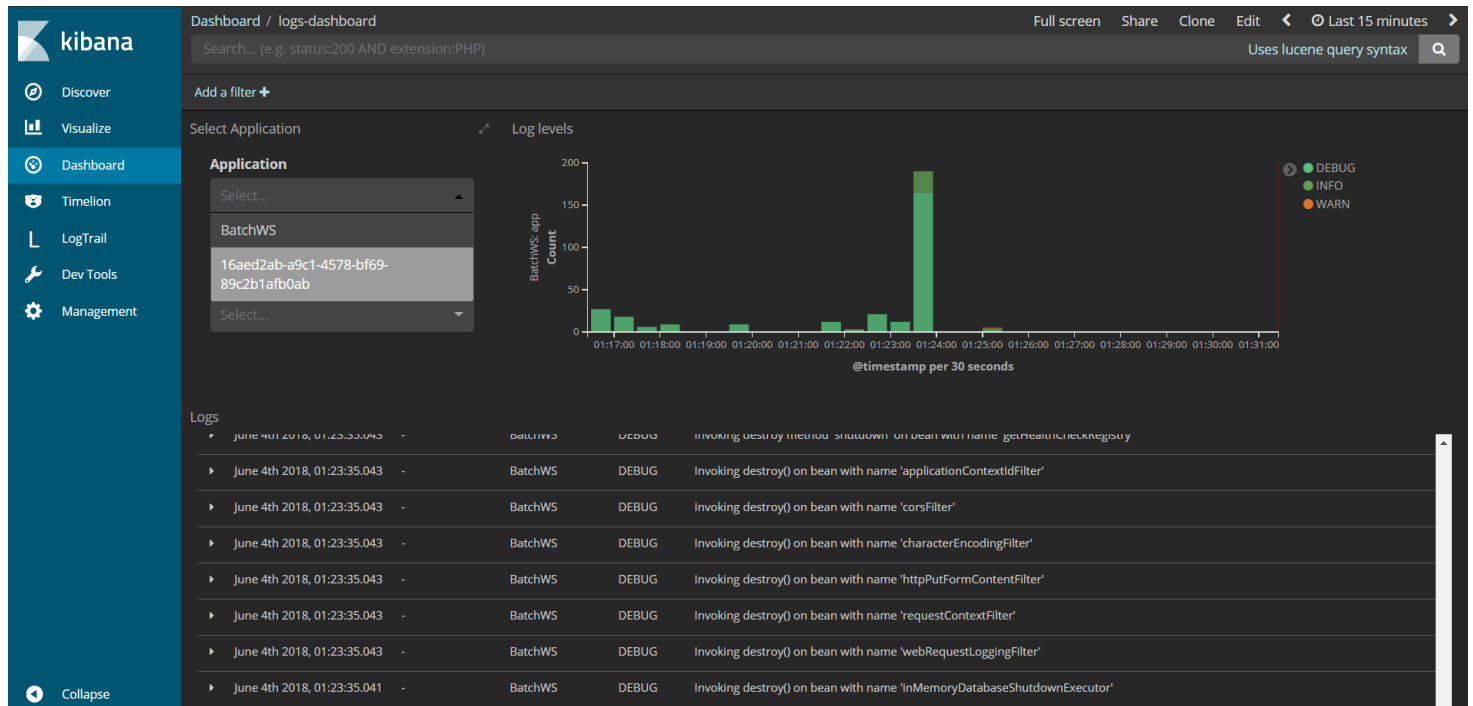


Figure 4.18: A Sample Kibana Dashboard

Conclusion

In this chapter I gave an overview of the Development Environment, following a detailed description of the solution, and finished the chapter by presenting the work done using several screenshots

5 TODO

Introduction

In the chapter we will present some Kanban artifacts, the project backlog, and end with a list of possible enchantments.

5.1 Kanban artifacts

Kanban have several artifacts, however, with the help of the issue tracking tool “Jira”, I was able to track my work, and generate some useful charts.

5.1.1 Cumulative Flow Diagram

The Cumulative Flow Diagram show the statues of tasks over time, this indicate the work done during period of time, and show potential bottlenecks that have faced me.

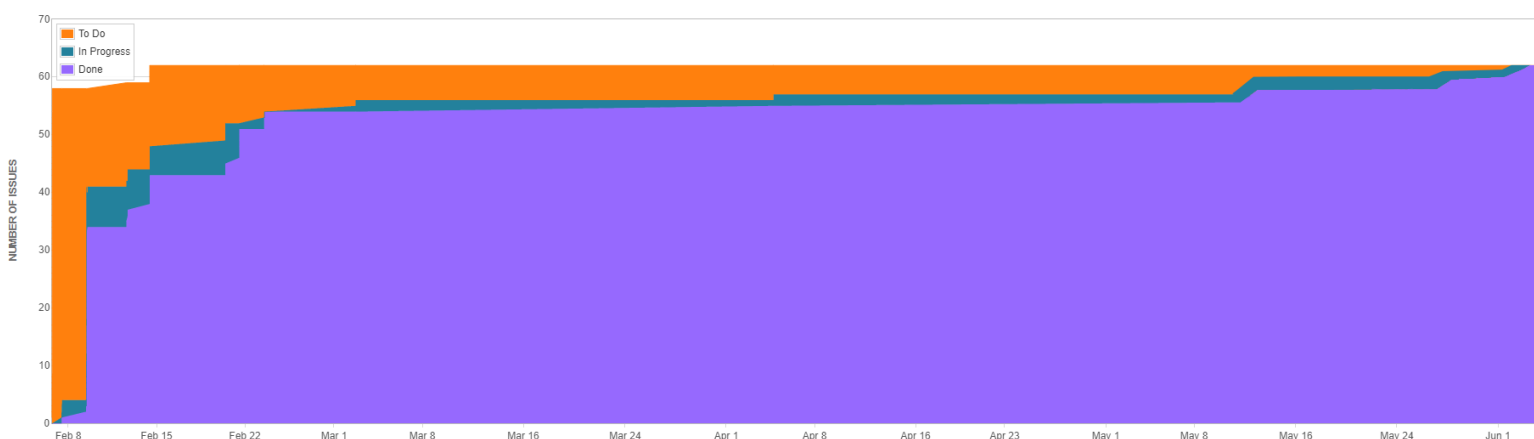


Figure 5.1: Cumulative Flow Diagram

5.1.2 Control Chart

The control chart is used to study the progress of changes over time.

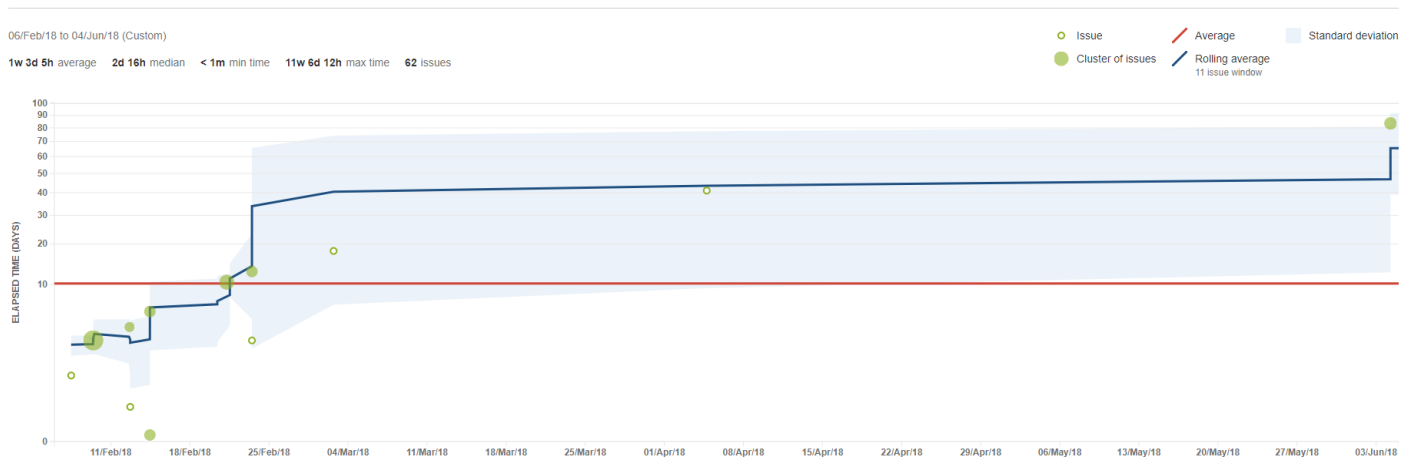


Figure 5.2: Control Chart

5.1.3 Backlog

The following table lists all the tasks in the product backlog.

Issue Type	Issue Key	Summary	Priority
Story	PFE18-1	POC Execute a Job Batch out of two jars	High
Sub-task	PFE18-2	POC Execute a Job Batch out of two jars	High
Sub-task	PFE18-3	Tests	Low
Story	PFE18-4	Job Batch MS creation	High
Sub-task	PFE18-5	JHipster generation	High
Sub-task	PFE18-6	Configure Microservice	High
Sub-task	PFE18-7	Initialize Project on GIT	Low
Sub-task	PFE18-8	Update Jenkins file	Low
Sub-task	PFE18-9	Update Dockerfile and docker-compose	Low
Story	PFE18-10	Setup job batch configuration (back-end)	High
Sub-task	PFE18-11	Design the ConfigBatch Model	Highest
Sub-task	PFE18-12	Generate JDL schema	High
Sub-task	PFE18-13	Update the generated CRUD operations	High
Sub-task	PFE18-14	Update Unit and Integration tests	Low

Story	PFE18-15	Integrate configuration on front-end	High
Sub-task	PFE18-16	Integrate ConfigBatch creation	Highest
Sub-task	PFE18-17	Integrate ConfigBatch update	High
Sub-task	PFE18-18	Integrate ConfigBatch Find/FindAll	High
Sub-task	PFE18-19	Integrate ConfigBatch deletion	High
Sub-task	PFE18-20	Unit and Integration Tests	Low
Story	PFE18-21	MS Creation and Job Batch jars	High
Sub-task	PFE18-22	Template Jar creation	Highest
Sub-task	PFE18-23	Client jar creation (Template Implementation Test) - db	High
Sub-task	PFE18-24	Client jar creation (Template Implementation Test) - csv file	High
Sub-task	PFE18-25	Client jar creation (Template Implementation Test) - Kafka	High
Sub-task	PFE18-26	Runner jar creation (Batch runner)	Highest
Sub-task	PFE18-27	Runner: 1st Step (Reader)	Highest
Sub-task	PFE18-28	Runner: 2 Step (Process, Writer)	Highest
Sub-task	PFE18-29	Unit tests	Low
Story	PFE18-30	Setting up an asynchronous messaging system using Kafka	High
Sub-task	PFE18-31	Create Avro model and Kafka producer	Highest
Sub-task	PFE18-32	Create Avro model and Kafka consumer	Highest
Sub-task	PFE18-33	MS and service runner communication (using socket)	Highest
Sub-task	PFE18-34	Update job status using Web sockets	High
Sub-task	PFE18-35	Create start/restart service	Highest
Sub-task	PFE18-36	Create stop service	Highest
Sub-task	PFE18-37	Schedule job	High
Sub-task	PFE18-38	Manage job execution history	High
Sub-task	PFE18-39	. Unit tests	Low
Story	PFE18-40	Logging	High
Sub-task	PFE18-41	Logstash configuration	Low
Sub-task	PFE18-42	Recover job batches execution traces (jar)	High
Sub-task	PFE18-43	Recover Microservices traces	High
Sub-task	PFE18-44	Preview/Download logs	High
Story	PFE18-45	Report Management	High
Sub-task	PFE18-46	Report generation library choice	Low
Sub-task	PFE18-47	Report Template Creation (General Batch Status)	Highest

Sub-task	PFE18-48	Report Generation (General Batch Status)	Highest
Sub-task	PFE18-49	Report Template Creation (Specific)	Highest
Sub-task	PFE18-50	Report Generation (Runner: Writer Step)	Highest
Sub-task	PFE18-59	View/Download Specific reports	High
Story	PFE18-51	Dashboard	High
Sub-task	PFE18-52	Home page Dashbord	Low
Sub-task	PFE18-60	Show number of jobs executed by date (month/year)	Low
Sub-task	PFE18-61	Show jobs status by exist codes	Low
Sub-task	PFE18-62	Show number of specific job reports generated	Low

5.2 Perspective

A lot of improvement and new feature can be introduced in this project, however, due to time constraint, I only implemented the most important ones, nevertheless, here is a list of possible features that can be done:

- Create a test suite for the developer how is going to implement the template jar, currently it is a bit hard to test. A spring boot starter can be used to help the developer during testing.
- Add a notification system, to notify the user of upon job status change event, using Email or phone notification.
- Add support for the DOC/DOCX format in the report editor.

Conclusion

In this last chapter, I have introduced two Kanban charts exported from Jira system, listed the backlog tasks, and set the spot on some future improvements.

General Conclusion

The purpose of the project was to develop a microservice that manages Job Batches and enable Advyteam clients to introduce their business logic into the application, with little or no help from Advyteam support team. In addition, generate different reports to help the client to have a broader overview of the batch status, and easy the create of employees files, such as payroll, evaluation cards, etc.

During this report I have tried to summarize the work done for 16 weeks while working on the application. I began the report by presenting the host company under which the project has been done. An overview of the project background was then presented, and finished the chapter with the different proposed solutions, and the reasons for the chosen solution.

The second phase was the analysis and the specification. Actually, this was fulfilled by detailing the functional and non-functional requirements. Thereafter a formal representation of the needs was demonstrated with the help and some use case diagrams.

After determining the application necessities, I began the design chapter, which contained an overview of the architecture, and the way the components communicated with each other, and the system state overtime.

Following, I described in the implementation chapter the solution in great detail and presented the achieved work with the help of some screenshots.

In the end, I presented the Kanban artifacts, with the help of a Cumulative Flow Diagram, Control Chart and the Backlog, and finished the chapter with a project perspective.

Personally, this project was very challenging to me, as most the technologies I have used was new to me and had to process a great amount of information in such short time, however, it was a very enriching technical experience, as much as it was a useful and rewarding exercise of the microservice and asynchronous communication worlds. Another great advantage of this internship is that in addition to the technical skills I have acquired, I have also learnt a great deal about the human resources systems, with a pioneer company such Advyteam in this field.

Bibliography

- [ALR10] David J. Anderson and Janice Linden-Reed. Getting started with kanban for software development. <https://dzone.com/refcardz/getting-started-kanban>, August 2010.
- [BDP05] Neil Pitman By Dan Pilone. *UML 2.0 in a Nutshell*. O'Reilly, 2005.
- [Com10] Higher National Computing. D77f 35: Systems development: Structured design methods, September 2010.
- [fMMSCOoIS08] Centers for Medicare & Medicaid Services (CMS) Office of Information Service. Selecting a development approach. <https://www.cms.gov/Research-Statistics-Data-and-Systems/CMS-Information-Technology/XLC/Downloads/SelectingDevelopmentApproach.pdf>, March 2008.
- [Fow18] Martin Fowler. Microservices a definition of this new architectural term. <https://martinfowler.com/articles/microservices.html>, 2018.
- [Glo16] Gary Glover. The importance of log management. <http://blog.securitymetrics.com/2015/08/importance-of-log-management.html>, August 2016.
- [Gmb] KO GmbH. Webodf. <http://webodf.org/>.
- [iT16] Moin in Technologies. Apache kafka v/s rabbitmq - message queue comparison. <http://www.cloudhack.in/2016/02/29/apache-kafka-vs-rabbitmq/>, February 2016.
- [KH06] Russell Miles Kim Hamilton. *Learning UML 2.0*. O Reilly, 2006.
- [Mar18] Robert C. Martin. *Clean Architecture A CRAFTSMANS GUIDE TO SOFTWARE STRUCTURE AND DESIGN*. prentice hall, 2018.
- [McC04] Steve McConnell. *Code Complete 2nd Edition*. Microsoft Press; 2nd edition, 2004.
- [Ora] Oracle. Jsrs: Java specification requests. <https://jcp.org/en/jsr/overview>.

- [UNI18] INDIANA UNIVERSITY. What is a batch job?
<https://kb.iu.edu/d/afrx>, February 2018.