Real-time log monitoring and notification system

Siddharth Baskaran (S3922782)
Deepshi Garg (S4199456)
Konstantina Gkikopouli (S3751309)
Shivam Mutreja (S3926575)

11th January 2020

GitHub Link:
`https://github.com/shivammutrejarug/spring-jms-mongo`

Contents

1 Introduction

Nowadays, even though technology is essential for businesses, it can still be the cause of faults and errors within the business processes. These faults and errors can lead to critical situations that need to be dealt with immediately.

In order to detect and prevent failures, it is important to include a monitoring system in the infrastructure. The main purpose of a monitoring system is to monitor technological tools and check if these tools are operating and performing properly. In addition, the monitoring systems are responsible for detecting errors/faults and alerting the users about them through various means like notifications and text messages. Apart from fault detection, the monitoring systems provide its users with a detailed real-time analysis regarding the state of the components using the monitoring tools to visualize the high-level analysis based on graphs, which are user-friendly and easy to interpret.

Given the importance of the monitoring systems, we aim to create and build a simple monitoring system that provides functionalities such as, analysis in real time, system alerts when an issue is detected, generating notifications and graphs that visualize the current state of the system.

2 System Description

As it is aforementioned, monitoring tools play an important role in the continuous operation of the HW/SW tools, as their main purpose is to prevent and detect when faults occur in the system. Our main purpose for this project is to create a system that consists of three main components. The first component is the client application. This represents an application that is being used by the user, who sends requests to the back-end. The second component is the back-end. The back-end will have a server and its corresponding database. The client interacts with the server through a CRUD API. And finally, the last component will be the monitoring tool, which monitors and checks the operation of client application and its back-end. In case a fault is detected, it will alert and notify the users of the monitoring system. All these components interact with each other and fetch data through a message queue.

3 Functional Requirements

This section of the document provides a list of functional requirements of the monitoring system. The table below consists of three columns. The first column Reference contains a label for each requirement. The second column Priority indicates the importance of each functional requirement and the third column has a small description of the functional requirement.

Reference	Priority	Requirements
FR1	MUST	The client should be able to send requests to the server application through the message queue.
FR2	MUST	Once the server application has received a request, it should be able to process the request and respond back via the message queue.
FR4	MUST	The server application should be able to send the monitored events to the log processor through the message queue.
FR5	MUST	The Log processor should be able to collect, aggregate and process the incoming events through the message queue.
FR7	MUST	The monitoring tool should be able to detect any errors or faults in the system.
FR8	MUST	The monitoring system should alert the users once a fault has been detected.

Table 1: Functional Requirements

4 System Architecture

This section focuses on describing and explaining the architecture of the monitoring system. All the architectural components will be defined and the entire architecture will be visualized in a figure. In addition, we will mention the technological stack that is used for building the monitoring system.

4.1 Components of Monitoring System

The monitoring system consists of the three main components and each of these components is responsible for certain tasks and functionalities.

All the components and their connections are visualized in the figure 1.

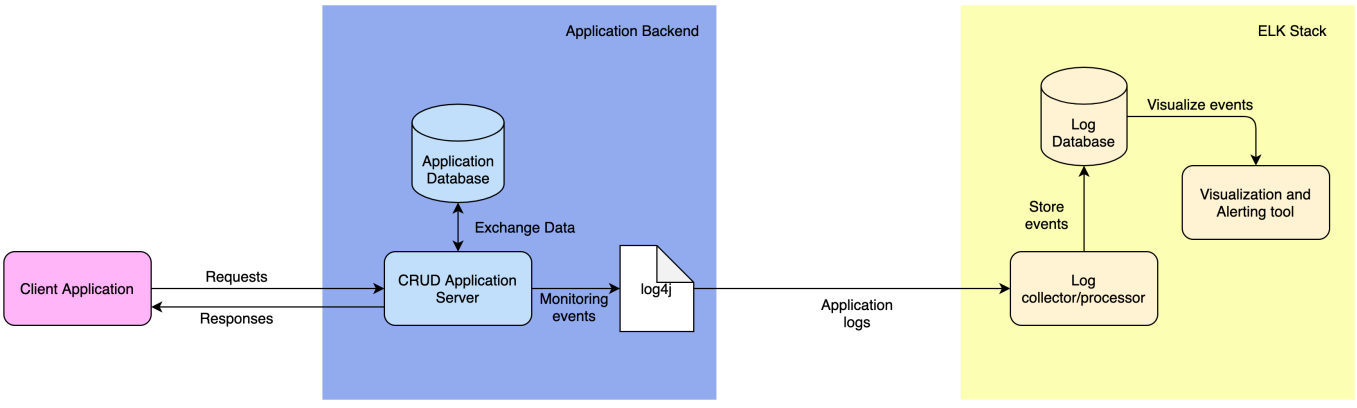

Figure 1: System architecture

Understanding each component in detail:

1. **Client Application:** It is the external client which invokes the server by sending requests, and demanding data back.
2. **Application Back-end:** It is a basic user authentication application in our system, which provides CRUD functionality via APIs for login, logout, fetch, and authenticating users. It stores user data in a database, which is called upon to read or write as per the requests received from the client. Server also records monitoring logs about the kinds of requests received, their responses recorded, and the time duration for each DB call. These logs are written into a `log4j` [3] file.
3. **ELK stack [6]:** It receives application logging events from the application backend, and is responsible for producing a user-friendly analysis based on them. It is a composition of 3 subsystems. First is the Logstash [8], which receives events from the application and stashes it into the search engine. Second is ElasticSearch [5], which is the search engine responsible for analysing the stored data. Third is Kibana [7], which provides pretty visualisations for this analysed data.

4.2 Technology Stack

The technological tools that helped in building the monitoring system are listed below.

- **Backend:** *Java Version 8 [22], with Spring Boot Framework [23]*: We initially considered Python as well, however, finally used Java because it has better community support for JMS implementations. SpringBoot was then chosen because it supports easy integrations of different subcomponents like DB and ActiveMQ, and exposing APIs.
- **Message Broker:** *ActiveMQ [1] with JMS compatibility [2]*: Amongst multiple available message broker options like OpenMQ, RabbitMQ and ActiveMQ, we chose latter because it has better community support and open source clients for JMS integrations.
- **Logging Collector:** *Log4j [3] and Logstash [8]*: Logstash was decided to use as a part of the ELK stack. We could directly collect logs from the server using Logstash. However, we added Log4j dependency in the middle so as to be able to seperate out the two systems (server and log monitor), and connect them via JMS.
- **Database:** *MongoDB [20] for Application Backend, Elasticsearch [5] for logging*: For server backend, we decided to use MongoDB because it is a document database, which means it stores data in JSON-like documents. It provides us flexibility with object structure. For storing logs, Elasticsearch was mainly opted as a part of ELK stack. It provides an efficient search and analysis engine.

- **Message Chanel:** *Point-to-Point [17]*: We use this because according to the current system design, we need only one receiver to receive a message. So, even if we horizontally scale any consumer component, a message is delivered to only one of those instances.
- **Monitoring Visualisations:** *Kibana [7]*: Kibana is an open source visualisation tool, which was chosen to use as a part of the ELK stack. It was selected over other available tools (like Grafana) because of its ease of configuration and available plugins to configure email and slack alerts.

5 Tiered Architecture

This section focuses on describing the tiered architecture of the monitoring system. It provides an overview of the entire system and its components. In addition, the tiered architecture is visualized.

This tiered architecture is visualized in Figure 2

The architecture of our monitoring system is 3-tier as it consists of three layers.

- The first layer, **presentation layer**, is the user authentication web interface. For our system in the presentation layer we also have the Kibana dashboard as it categorises the different status codes received from the server and provides the user with the number of occurrences for each status code. It is meant to be used by system engineers to monitor system performance and receive alerts in case of any configured anomaly. For the project, we did not implement a web interface as it was not a project requirement and we could simulate the behaviour with APIs. Thus, for our implementation we are simulating APIs and error scenarios.
- The second layer, **application layer** consists of the CRUD application, which is responsible for handling all logic of the system such as user login, user authentication, user retrieval, and user logout functionality, log4j, which sends the logs to the queue, ActiveMQ, which is the queue that holds the log messages, and Logstash which gathers logs from the queue, formats the logs.
- The third layer, **database layer** stores the formatted logs received from the **application** layer. In our case, elasticsearch receives the data from logstash and indexes the data for ease of retrieval. It also holds the MongoDB database which stores all the customer info used for authentication.

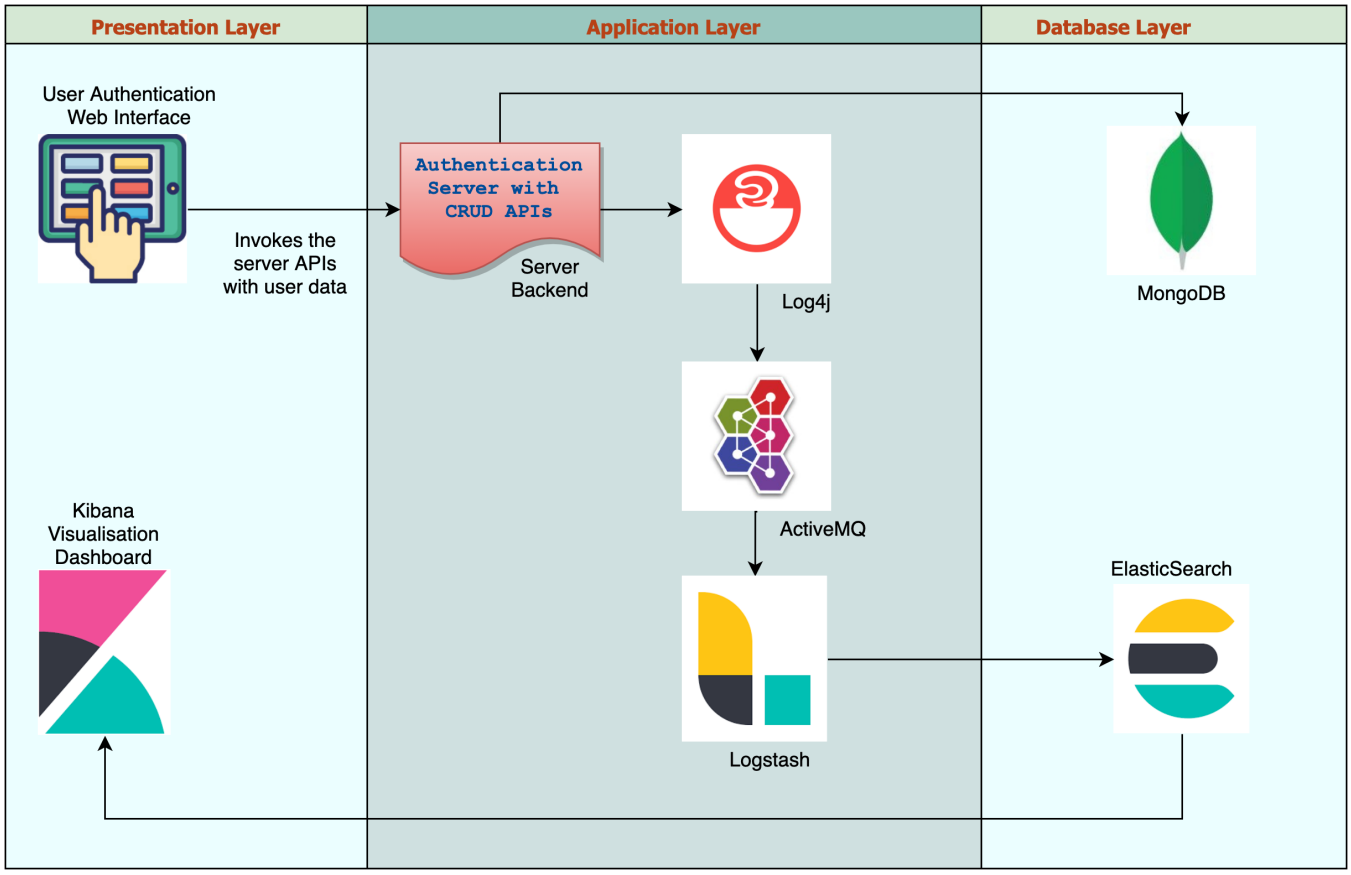

Figure 2: Tiered architecture

6 Integration Patterns

This section of the document focuses on the integration of the components in the monitoring system. Firstly, we define the interactions that occur between the components and then we describe the patterns that are applied for each interaction.

6.1 Interactions and patterns

The monitoring system contains the two major interactions between its components. They are described in the Figure 3

Understanding briefly,

1. The client application interacts with the server application by sending

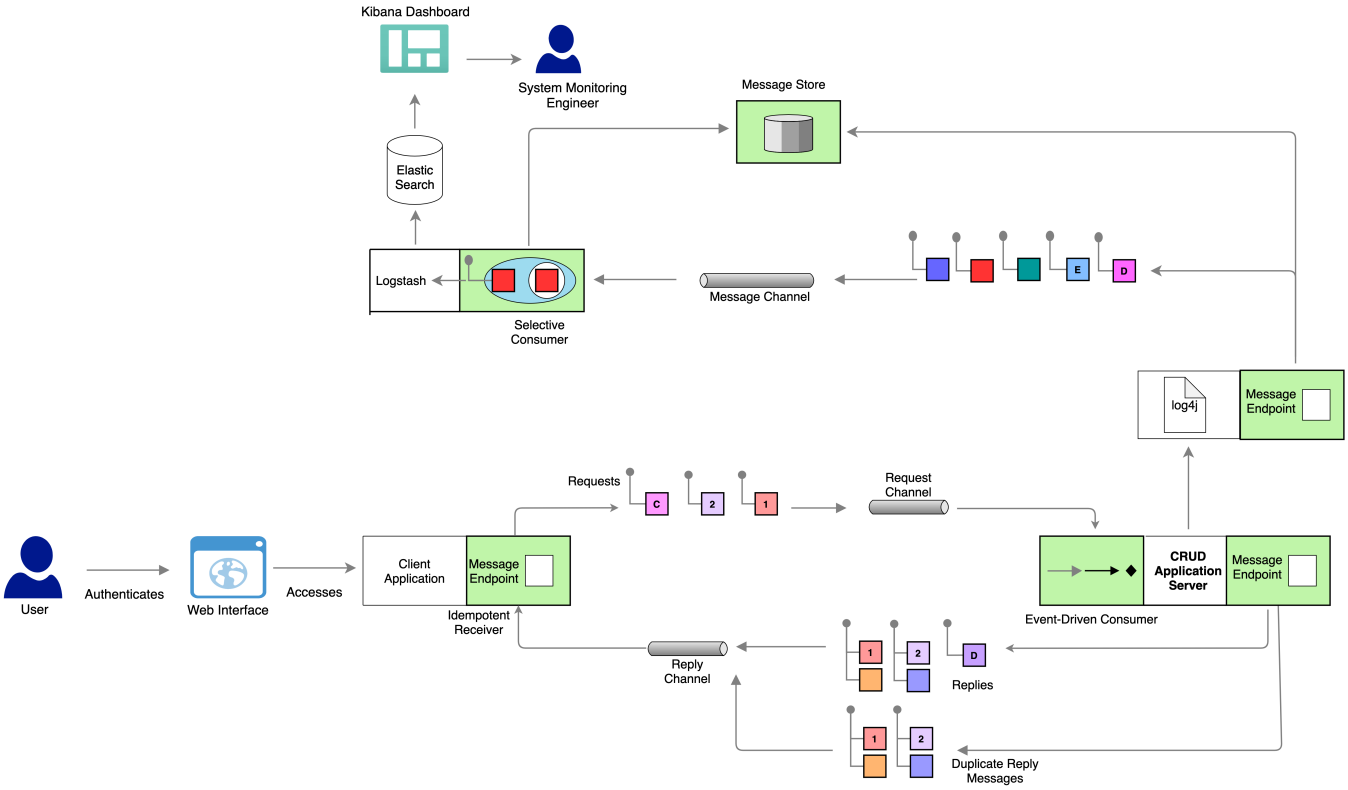

Figure 3: System Architecture with Integration Patterns

requests.

2. Once the client sends a request, the server receives this request, processes it and replies back to the client by a response message.

3. The server application also interacts with the log processor of ELK stack. to send monitoring events to the log processor.

The patterns [4] that are used in each interaction are described below:

6.1.1 Interaction between client-server

The patterns that are applied for this interaction are shown in Figure 4

- **Message Endpoint[15]:** This pattern is used by both the client and server applications to send and receive messages from the message queue. It is implemented using a Java interface class[24] in our system. This class provides methods for configuring both: a sender as well as a receiver. Across the system, this class has been instantiated as per need. We decided

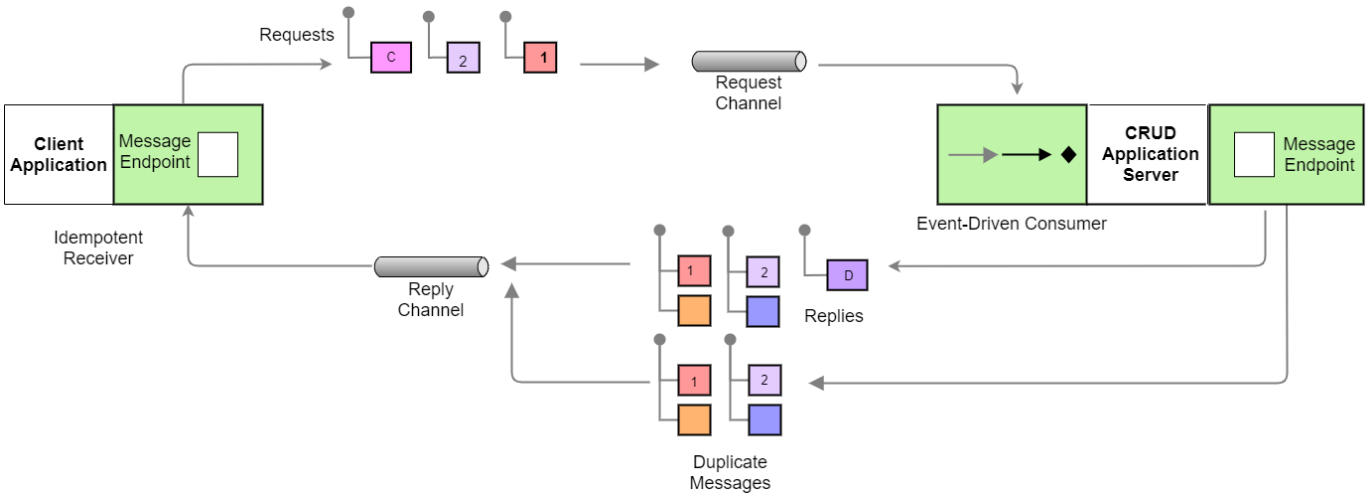

Figure 4: First Interaction

to use this pattern to provide a common method to interact with the message queue, so as to avoid code duplication. This also transfers the responsibility of interacting with the message queue from the service layer to the endpoint layer.

- **Request-Reply[18]**: This pattern is used as the primary mode of communication between the client and the server. Server exposes CRUD APIs over a request message queue (called `server.q` in our system). Client pushes request messages over this queue, and awaits a response on another queue (`client.q`). Server receives the request message, processes it, and sends the appropriate response over the `client.q`. The decision to use this pattern is highly backed by the decision to have asynchronous communication between server and client. This is to let each of them handle their messages at their own pace, while being able to engage in other processing.

- **Event-Driven Consumer[12]:** This pattern is used to make sure that the receivers of all the queues in the system receive the message as and when it is placed in the queue. It is implement by using the Default JMS Listener Connection Factory to establish receiver queue connections. In our system, server acts only upon the messages from client, it makes sense for it to receive these messages as they arrive. Similarly, client has to process the responses sent by the server, and thus, does receives them at the earliest.

- **Correlation Identifier[10]:** We have a field `request-id` in each of the messages passed between server and client, which acts as a correlation identifier. When client sends a request message, it adds a `request-id`

string to it. Server adds this same `request-id` to the corresponding response message. This is needed in our system for the client to be able to track the response messages per request message basis. Although, in the current basic implementation, client does not track or process responses. But if it were to, `request-id` makes it possible.

- **Idempotent Receiver[14]:** In our system, the field `request-id` allows the client to be an idempotent receiver. It means that even if the client receives a duplicate response message from the server, since it would have the same `request-id`, client would not reprocess it. However, server is not idempotent as it does not store the `request-id`s of all the messages received.
- **Command Message[9]:** The request message from client to server is a command message as it invokes its own processing in the server.
- **Document Message[11]:** The response message from server to client is a document message because server just blatantly passes the response data to the client, without caring how client handles it. In the current implementation, client just logs it. However it can process it in multiple ways, without impacting anything on server.

6.1.2 Interaction between server-logs-ELK

The patterns that are applied for this interaction are shown in Figure 5

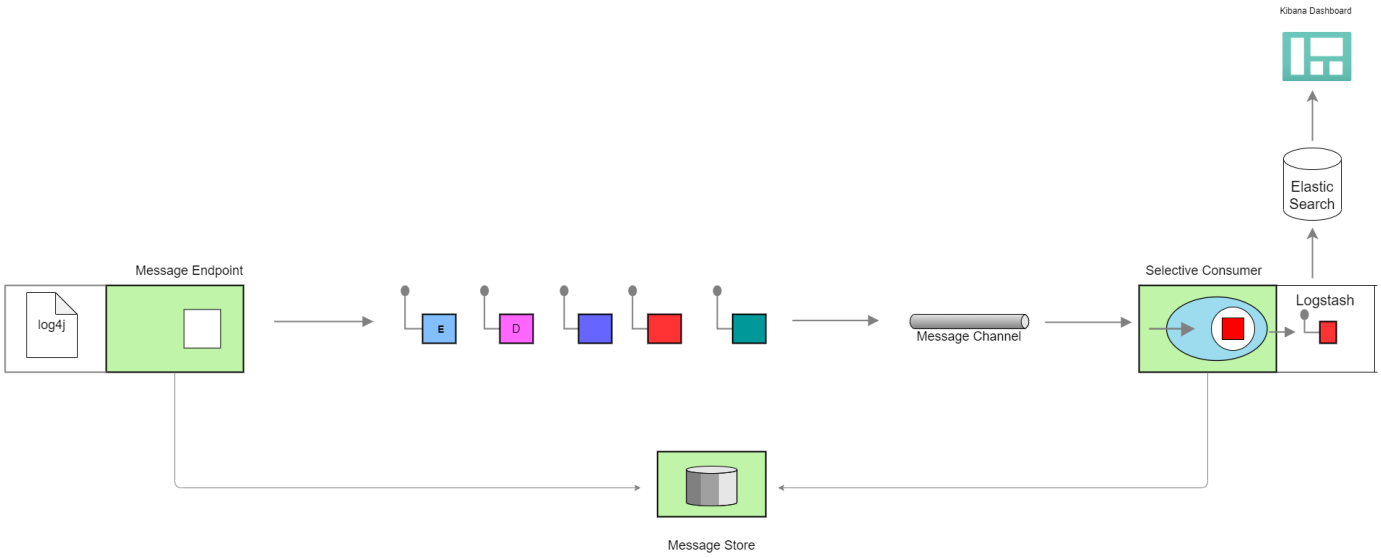

Figure 5: Second Interaction

- **Message Endpoint:[15]** This interaction describes the communication between the server and the logstash. As mentioned earlier, the server sends monitoring events to logstash. In order to enable this interaction, this **message endpoint** pattern has been implemented, which connects both these components through the message queue. In this way, the server and the logstash are able to exchange messages via the message queue. Logstash has an index `logstash-*` which has been configured using `jms-appender` to read from the `server.q` and `client.q` queues.

- **Event Message:[13]** Another pattern that is present in this interaction is the **event message** pattern. Since this interaction focuses on sending monitoring events, it is important to have the event message pattern implemented. This pattern is responsible for notifying the participant components when an event occurs. In our case, the `jms-appender` is triggered every time a new event is sent and this will be processed by the logstash and then visualized in the Kibana dashboard.

- **Document Message[11]**: Document message is an additional pattern present in the communication between the server and the logstash. The pattern is used as a means of transporting the data in form of a document from the server to the logstash.

- **Selective Consumer[19]**: As logstash only filters out ERROR logs and does not show INFO and SUCCESS logs, using a selective consumer we were able to filter out the the log messages and send only the ERROR logs to logstash.

- **Message Store[16]**: As we needed to store historical data for population of data later on, we used a message store which works as a good backup solution to store the logs as a duplicate without impacting performance.

7 Future Work

This project can be elaborated in multiple directions.

- A web interface can be implemented for a naive user to interact with the authentication server.

- Client application can me made more complex. Currently it does not logically process the response messages received from the server. It only logs them. However, it can be extended to perform tasks a follow logical flows based on these responses received from the server.

- Authentication server only provides login, logout, fetch and authenticate APIs. There is no provision to register a new user. Also, there is no idempotency in handling received request messages, or even maintaining user states. An already logged in user, for example, can login again. Work

can be done in these directions, and more complexities can be added as such.

- Kibana visualizations currently classify and visualize the server requests only based on the response status code, and the API triggered. More visualisations regarding the time taken for each request to process, the time taken for DB calls, etc, can be added.

- Kibana is currently configured to send out email and slack alerts only, and those too only for exemplary scenarios. This can be extended to multiple alert channels and multiple alert scenarios as well

In addition, the whole system can be turned into a generically available monitoring system which can work on any client-server combination. Additionally, the authentication server can be developed into an independent module as well, available for being plugged into any bigger system.

8 Team members, role distribution and responsibilities

Konstantina Gkikopouli

- Researched the patterns and decided on a few for the topic
- Wrote the first draft of the report (Introduction, Functional Requirements, System Architecture, Technology Stack, System Description, Enterprise Application Integration Patterns and Tiered Architecture)
- Created client web interface.
- Created configurable client to flood the API endpoints the with multiple requests.
- Created the final pattern diagrams.

Siddharth Baskaran

- Researched various ways to setup the system and finalised on Spring-boot.
- Set up initial spring boot application.
- Implemented initial set up for Log4j.
- Implemented initial set up for Kibana logs.
- Tried to integrate Apache Camel into the system, ran into some errors.
- Setup Kibana Dashboard to filter error logs.
- Fixed Server-logs-ELK Patterns in the report.
- Gave the Final Presentation.

Shivam Mutreja

- Researched about various ways we can use a JMS compliant broker, Apache ActiveMQ with Python, as it was our preferred language.
- Setup the initial working integration between ActiveMQ and Python using the STOMP protocol which was scrapped after the team's discussion.
- Setup the initial working Spring Boot application with ActiveMQ and MongoDB [24].
- Implemented a Python script to generate fake customer data and store it in MongoDB to be used by the authentication system.
- Configured ActiveMQ and set up the queue.

- Setup the ELK stack which included connecting Log4j to the application and the ActiveMQ queue to Logstash.
- Completed Log4j integration with ActiveMQ.
- Completed Camel integration with Spring boot and ActiveMQ.
- Configured Kibana Dashboard to filter error logs.
- Setup Open Distro [21] plugin to upgrade the otherwise limited scope of Connectors and Alerts in the Kibana dashboard for sending out notifications.
- Re-wrote section 5 and the tiered architecture diagram corresponding to our implementation of the application.

Deepshi Garg

- Researched different integration patterns and proposed where to use what.
- Proposed the initial architecture, and put forward the idea for CRUD app and client app as the two components with Request-Reply connectivity
- Created initial architecture diagram, and the diagram with all patterns
- Tried the initial project setup with Java as a Maven project, with a JDBC PostgreSQL driver
- Implemented a basic CRUD application for customer authentication. Exposed REST HTTP APIs for testing
- Added JMS Request-Reply APIs for the CRUD application
- Implemented a basic client app to invoke multiple error scenarios in the server, using the JMS APIs.
- Sent out relevant logs from server and client apps, to be able to visualise system state.
- Rewrote the Sections 4, 6.1, 6.1.1 and 7 with relevant references and explanations.
- Recreated the system architecture diagram and all the interaction diagrams
- Fixed the Bibliography to be able to display the links of online accessed resources

References

[1] Apache. Activemq. Accessed on January 10, 2021, http://activemq.apache.org/h.

[2] Apache. Activemq jms implementation. Accessed on January 10, 2021, https://activemq.apache.org/how-do-i-use-jms-efficiently.

[3] Apache. Apache log4j 2. Accessed on January 10, 2021, https://logging.apache.org/log4j/2.x/.

[4] Apache Camel. Enterprise integration patterns. Accessed on January 9, 2021, https://camel.apache.org/components/latest/eips/enterprise-integration-patterns.html.

[5] Elastic. Elasticsearch. Accessed on January 10, 2021, https://www.elastic.co/elasticsearch/.

[6] Elastic. Elk stack. Accessed on January 10, 2021, https://www.elastic.co/what-is/elk-stack.

[7] Elastic. Kibana. Accessed on January 10, 2021, https://www.elastic.co/kibana.

[8] Elastic. Logstash. Accessed on January 10, 2021, https://www.elastic.co/logstash.

[9] Enterprise Integration Patterns. Command message. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/CommandMessage.html.

[10] Enterprise Integration Patterns. Correlation identifier. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/CorrelationIdentifier.html.

[11] Enterprise Integration Patterns. Document message. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/DocumentMessage.html.

[12] Enterprise Integration Patterns. Event-driven consumer. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/EventDrivenConsumer.html.

[13] Enterprise Integration Patterns. Event message. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/EventMessage.html.

[14] Enterprise Integration Patterns. Idempotent receiver. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/IdempotentReceiver.html.

[15] Enterprise Integration Patterns. Message endpoint. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageEndpoint.html.

[16] Enterprise Integration Patterns. Message store. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageStore.html.

[17] Enterprise Integration Patterns. Point-to-point channel. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/PointToPointChannel.html.

[18] Enterprise Integration Patterns. Request-reply. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/RequestReply.html.

[19] Enterprise Integration Patterns. Selective consumer. Accessed on January 9, 2021, https://www.enterpriseintegrationpatterns.com/patterns/messaging/MessageSelector.html.

[20] MongoDB. Mongodb. Accessed on January 10, 2021, https://www.mongodb.com/.

[21] OpenDistro. Plugins. Accessed on January 9, 2021, https://opendistro.github.io/for-elasticsearch-docs/docs/alerting/.

[22] Oracle. Java 8. Accessed on January 10, 2021, https://www.oracle.com/java/technologies/java8.html.

[23] Spring. Spring boot. Accessed on January 10, 2021, https://spring.io/projects/spring-boot.

[24] Team. Jms implementation. Accessed on January 9, 2021, https://github.com/shivammutrejarug/spring-jms-mongo/tree/main/src/main/java/com/eaiproject/jms.