Entomomachia

Error classification and bug detector

University of Catania, engineering of distributed systems course

Locicero Giorgio

locicero.giorgio@studium.unict.it

01. January 2021

Abstract

With the demand and need for software, the amount of errors that are spawned from the code grows almost linearly with the number of lines written. Most of the times, the errors produced are common for a lot of people, for example a forgotten line to clean the code, or some erroneous logic. In fact, most of the times, when an error comes up for code that seems to be perfect, the first solution is to search in the internet for similar problems and errors, and that is why the problem of finding an error falls most of the times in understanding the context behind and see if other people get the same error. If there are no apparent errors in the code, mutation testing is also a great instrument to use to find if the logic of the code and the tests written is good enough to enhance the overall quality, and mutations are also linked to possible errors in the code(like logic or exceptions). The tool presented in this research addresses the recommendation of solutions and mutations to account for these common problems when writing and improving code.

1 Introduction

The base concept of this project was taken from common linting techniques and software, but It is different because It is directed towards classification and labelling of errors and predictions of solutions. The framework also tries to predict mutation or introducing errors based on an inverse prediction of bugs or failed tests based on correct code (the dataset used to build the model without any shared code is the defect4j-dissection dataset [1], that contains the commits to a repository, so It have the code with some errors and the corrections/additions to the code that were introduced to fix the problem). The project exposes some REST API to get errors and mutations based on code, users or groups. A frontend to interact directly with an interface was not in mind but It is under development (because this project was done by only one person and I do not like front-end design, I am also quite busy). The recommendation are also filtered with data passed to the HTTP request body, and this data is used for the query. Some solutions will be public and accessible from all users, other resources can only be accessed by particular users (maybe because the code is not public or the user doesn't have permissions to see the code), and some types of operations could only be done by administrators. Authentication and Authorization is done via the web interface, but that part of the project is not yet complete so for now, the only available part is the REST API to post and get recommendation code, and to improve the model.

The project is available in my repository at

`https://github.com/josura/Error-classification-ISD`.

2 Preliminary considerations and conventions

The UML classes and definitions of methods are mixed between two types, that is like **Scala** defines methods and variables, and how **Java** defines them, and sometimes the two definitions are mixed because the classes were done very quickly and I have made some mistakes during the documentation of the project.

3 General description and workflow of the software architecture

The project is divided in services that serve different purposes, the Prediction service is divided in smaller micro-services to have a well-defined boundaries on responsibilities among every part of the system. The client interface is also divided in smaller micro-services, in particular the users could interact directly to the REST APIs or to the Web interface.

There are 3-4 main parts for this project:

- **Prediction Framework**: That is composed of a labeller service that implements a **machine learning pipeline** to clean, control and classify the code that It receives
- **ElasticSearch interface**: That implements a **remote facade** to query the ElasticSearch server, It returns a DTO to clients.
- **Broker-orchestrator**: That serve as a central component to the system, It communicates with the ElasticSearch interface via a client, takes the labels predicted from the prediction framework, and write the results to a repository(REDIS transaction repository)
- **REST APIs** that is part of the backend but is also used to interact directly with clients(at least until the web interface is implemented and the boundaries of the functionalities of the project are well-defined)

There are also more parts that are under development or for future extensions of the project, these tasks are:

- UNDER DEVELOPMENT **Web interface**: The web interface will be the primary medium where the clients will operate with the system, It will also implement **Authorization** and **Authentication** with **Spring security** to differentiate users in different groups.
- NOT IMPLEMENTED **Tracing** with OpenTracing and Jaeger, It will monitor transactions and end-to-end interactions.

There are also other parts of the system like **log analysys** and **index analysis** of ElasticSearch with Kibana, along with other services that are not used in the main project, but these parts are not useful to the general user so they will not be described in this documentation.

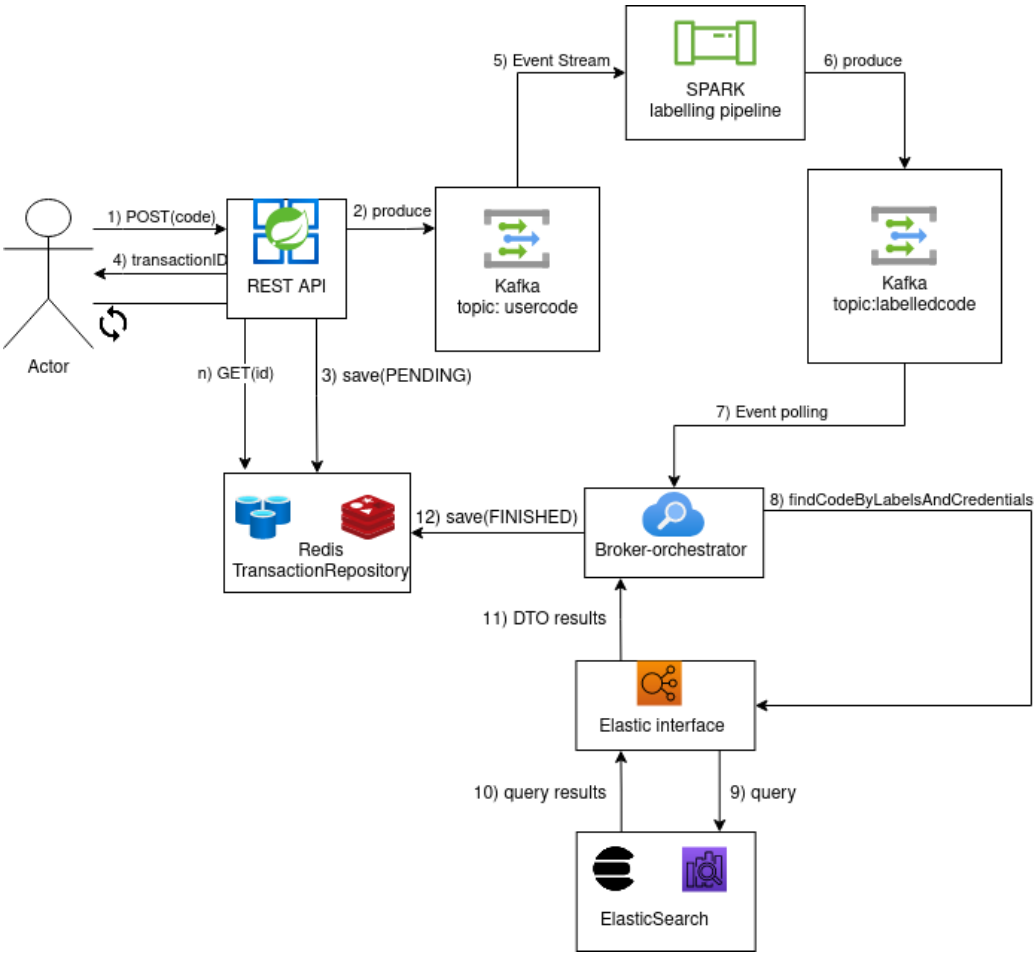

Figure 1: Workflow path through the system

The whole workflow and journey of the data and requests among the services in the project can be seen in figure 1 The workflow for the user who wishes to get solutions and mutations recommendation is the following:

1. the client send some code through an HTTP request to the **server**, in particular a **POST request**, where the body is formed in the following way:

```
{
  "user": "...",
  "group": "...",
  "code": "..."
}
```

2. The requested body gets unmarshalled and controlled/validated, code with comments will not be accepted. After this the marshalled body is sent to a **Kafka topic**(usercode).

3. A transaction is created and saved in **REDIS** as **PENDING**, the **transaction ID** is generated incrementally and is retained by the redis repository as well.

4. The client will receive a transaction status with an ID as one of Its features, this ID will be used to poll the status of the transaction.

5. The client will poll whenever He wants by sending HTTP GET requests to the server, at least until when the transaction status is FINISHED and contains the result or for an ERROR status.

6. The labelling pipeline is activated when new streaming data arrives, the new code will be labelled by the neural network model.

7. The labelled code will be written to Kafka in a topic(the labelled stream will be passed to the topic as a stream as well) or if an error occured in the current stream, all the record in the stream will be marked with **ERROR** in the transaction repository in REDIS.

8. The **Broker-Orchestrator**(or choreographer) will poll for new records in kafka for the **labelledcode** topic, when new data arrives, the service consumes single labelled records and uses the **Elastic interface** to query ElasticSearch for the Solutions and Mutations with the correct labels,users and groups.

9. The **Elastic-interface** queries ElasticSearch with the credentials and labels provided.

10.

11. the results of ElasticSearch are unmarshalled and encapsuled in a DTO

12. The DTO is sent to the client(**Broker-orchestrator**).

13. The results obtained are written to the REDIS repository and the TransactionStatus is changed to **FINISHED**, if some problems occured during the query, the system will change the transaction status to ERROR.

14. The client will poll for the results and will finally get a FINISHED transaction status along with the results, that is a list of **solution codes** and a list of **mutation codes**

To understand how the neural network model used to label the records is built see 4.1.6

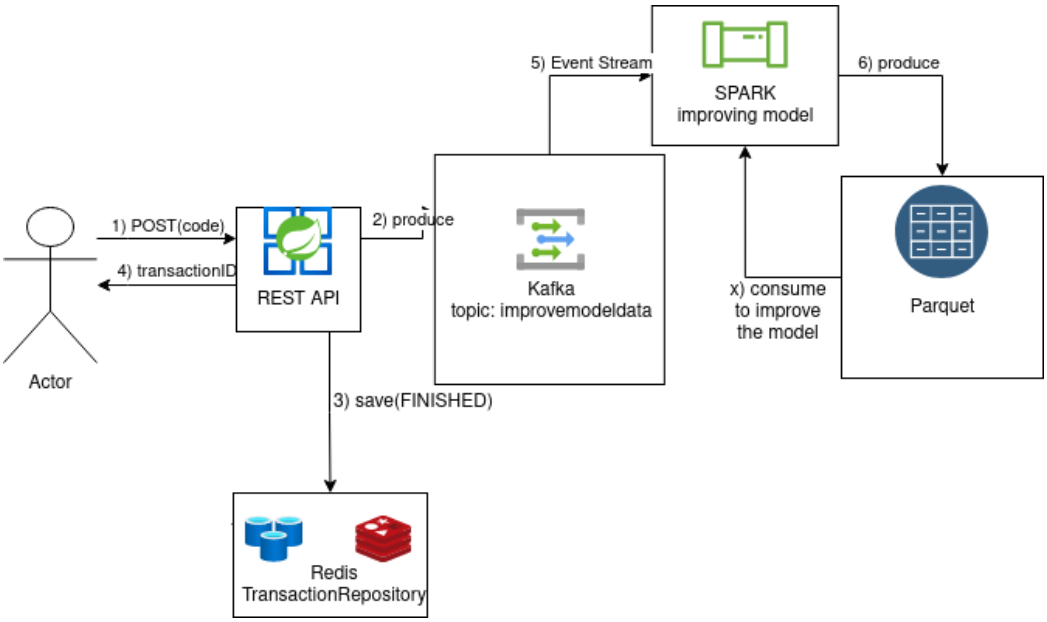

Figure 2: Workflow path through the system to improve the model

The situation when the user shares some code with errors and a solution is not seen in figure 1 but can be seen in figure 2.

The process to improve the model is really simple and is described by the following steps

1. The client send some code through an HTTP request in a PUT request(or a POST request to the */share* route), where the body is formed in the following way:

```
{
  "user": "...",
  "group": "...",
  "code": "...",
  "solution": "..."
}
```

2. The server will unmarshall and validate the data received from the client, and will write to the kafka topic (**improvemodeldata**) the marshalled data.

3. The server will write a transaction to REDIS with the status **FINISHED** and will return the transaction to the user.

4. The **improve-labeller-model** service will read the stream associated with the previous topic and will save the new training data in a Parquet folder.

5. whenever the model needs to build a new model, It reads the data in the parquet file and builds the model with the old data along the data shared by users.

6. and will receive a token ID to poll the status of the transaction.

Consideration on the model and the improvements that could be done with the new data shared by users are discussed in 4.1.6.

No further calls are done, because the client receives an instant **FINISHED** transaction if the REST APIs have sent the code to kafka with success.

The single components of the architecture will be analysed and described in detail to understand the objective and the role of every service in the whole system.

4 infrastructure description

The whole framework is built with a Service-Oriented infrastructure in mind, composed of smaller microservices that, combined, form a single service that implements some functionalities for the system.

From the point of view of the advanced API user, the system offers some APIs to get the predictions from the backend core infrastructure that does the work behind (the requests are done in a REST style and will respect the CRUD practices as much as possible, even though the GET could also take a body).

With GET operations, the user will get the resources it needs(Transaction status and results primarily), with a POST operation, the user will pass the code that will be used for prediction, that is creating a new transaction and passing to the labelling pipeline that will be used to build the final results, with the PUT operation(or a post to the */share* route), the user will update the code with a correction code or a mutation, or pass the whole code with correction or mutation to the system(to improve the model). The DELETE will try to delete the resource, that is deleting results and transaction status.

The development was done in a Test-Driven fashion with no additional scheduling or a defined plan about the development strategy because I was the only one involved in this project and, because of the complexity of the whole system, the whole project has taken most of my free time (the development was in the middle of the semester, so the day schedule was filled almost all day with other projects and research), and my routine is not defined day to day, so I work on the project when I have time (this action have consequences on my development because I find It hard to continue a project after 3-4 months of not even seeing and controlling the code that

I have written). To aid my quest to complete this project, the documentation cover a lead role and I will try to make It as clear as possible.

I will start describing the prediction framework and the operations done to predict errors and mutation(by recommendation of shared code and solutions).

4.1 Prediction

This part of the project is done in Scala with the help of the Spark engine.

The Prediction service takes data sended by users from kafka in a kafka stream and labels It or uses It to improve the model.

The model is created initially by using the training data from the defect4j dataset [1], obviously the data used was previously formatted and cleaned from components that could give the runtime cleaning procedures some problems(that is, all comments were removed, useless parts were deleted, only the code has remained in the final data used).

The operations done by the framework are logged in a file in the */logs* folder at the root of the project.

To build the model, the code with errors and the solutions to the errors are additionally cleaned for words and symbols that are not in the language, this is because the method used to do text mining will construct the vectors, used in the clustering and labelling methods, will create these vectors by aggregating group of sequential words, see if the group of words are common, and build the final array of doubles(the process is more complex, for more information about the construction of the **feature vector** see the original article [2] and the Spark implementation [3]). Most words that are not in the main programming language (like variable names) will be filtered, and the code will be formed only on operators or semantic of the language.

After the creation of the feature vector(around 300 features for every code with errors and solution used independently from one another, that is 600 features for record), these feature vector are used to aggregate and cluster more records, these cluster will be the final labels that will be used to train the neural network. The **expected value** of elements for cluster will be around 3 records.

After the clustering part of the pipeline, the final step is to built the **Multi-layer perceptron model**, the model right now is only formed by a single hidden layer with 200 nodes(the model is really simple, additional improvement could be done afterward), the last layer is the number of cluster that we want, the first layer is formed by the length of the feature vector passed.

For both error and mutant prediction, the code submitted should be with no comments(even though the code submitted will be validated before the whole process behind, if the code has comments, nothing will happen and an alert will be generated) and with good spacing between operations and instructions.

4.1.1 Labeller Code structure

The whole structure of the labeller as an UML diagram can be seen in 3. This structure is used in two services: the **usercode labeller** and the **labeller improver**. The usercode labeller will label the records received from the clients that use the REST APIs to get Solutions or Mutations recommendations, the labeller improver will get the data shared by the user to improve the prediction model and will save the new records in a repository.

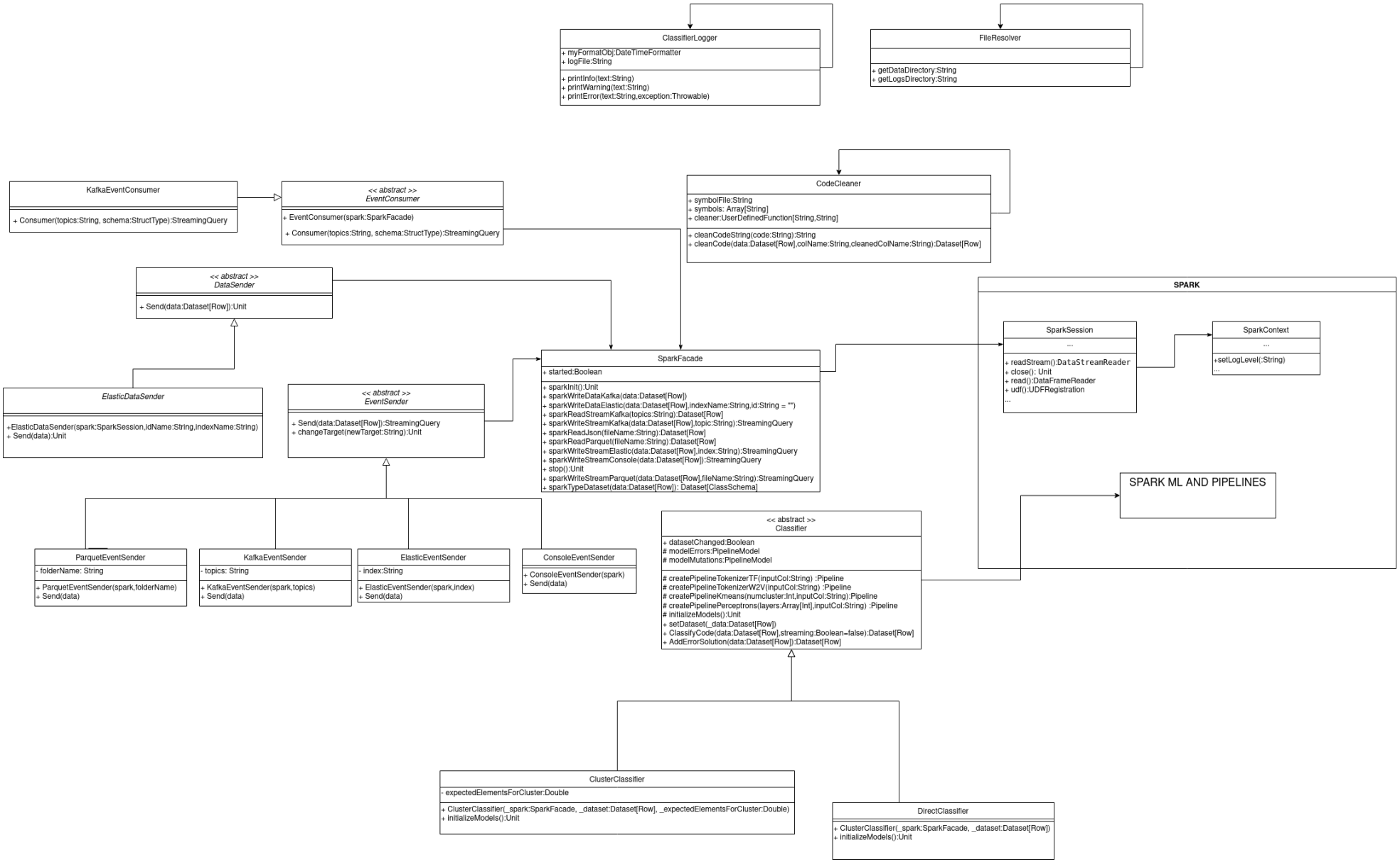

Figure 3: Classifier code structure

The **SparkFacade** class implements a Facade that serve as an interface to the spark framework and its main functions, It also could be seen as a **Proxy**(in some parts, even though It does not implements a common interface with the same methods as a spark session) because the Spark Session is not started until an operation is done, and also because the methods that It implements could not return values when unusual or undesired events occur during execution (to protect access to the framework operations, so It also implements some functionalities of a protection proxy, with only a fixed set of authorization rules for a single user).

The **SparkFacade** encapsulates a lot of external calls to the SPARK framework into single calls to the object, It also provides some bulk calls to the Dataset class (for a Dataset Object passed) because a lot of methods need the *spark.implicit._* functions to work properly.

The other components of the system communicate to Spark through the SparkFacade, this is done by passing the main SparkFacade instance through the constructor of the classes that need to use the Spark APIs (**dependency injection** of the SparkFacade instance). For example, the *KafkaEventConsumer.consume(...)* method uses the *SparkFacade.sparkReadStreamKafka(topics:String)* function to read a stream of a topic in Kafka.

The singletons of the projects are **FileResolver**, **CodeCleaner** and **ClassifierLogger**

The FileResolver singleton is used to differentiate toward services run in the machine as programs or containers, It resolves system paths of logs and data folders.

The CodeCleaner is one of the main components of the labeller architecture, the singleton is used to clean a feature of a dataframe(that is streaming or not). In particular, the code cleaner removes the words that are not in the language that is currently analyzed (only **JAVA** for now).

The ClassifierLogger extends the LazyLogger class from **scalalogging**(*com.typesafe.scalalogging.LazyLo*
and is used to log during execution to console and file, the log file is retained in the **logs folder** under the name of *classifierlog.log*. Future tasks could be to transfer the logs and listen to new record-logs to do some analysis(with the use of Logstash or something like that).

The **ParquetEventSender** and **ConsoleEventSender** are used in local(even though the Parquet sender could be changed to send the files in a distributed file system like **hdfs**). The ParquetEventSender is used by the **improve-labeller-model** service to save the streaming shared code to a parquet folder/file, where these new record will be used to improve the prediction the next time the model is rebuilt (there is an open issue though because the whole model is not currently rebuilt with new data as training but the new data is only used to label **error** and **solution** to be used in ElasticSearch for the queries from the client).

4.1.2 Streaming and data consumption

The data that needs to be labelled or used to build a better model are taken from kafka through topics, for labelling the topic used is **usercode** while the model enhancement and data collection(used for traning) takes new record from the **improvemodelcode** topic.

The data is passed to the classifier model or is used to build a better model by saving the new record(that will be used the next time the model is reloaded to build a better model).

To consume **Events** and **Streaming topics**, the abstract classes **EventConsumer** and **DataConsumer**(not in the final project) are extended.

The subclass created is **KafkaEventConsumer** for the EventConsumer superclass, this is the only Consumer used in the final project because there was no need for other services to be used as communication frameworks, Kafka is enough for the labelling service.

The classes used to consume events and remote data could be seen as **Receivers** from the **Forwarder-receiver** design pattern, even though the original patterns uses a direct connection through the remote devices(with **synchronization**), the concept of communication with events could be implemented in the same way(Asynchronous communication) by polling or waiting for streaming data in a topic.

4.1.3 Streaming and data creation

The labelled data is sent to Kafka and Elastic search to be used by other services.

To create and send **Events** or **data** to Kafka o ElasticSearch, the two abstract classes **EventSender** and **DataSender** are used.

The subclasses created from the EventSender superclass are **KafkaEventSender** that is used to send events to Kafka, **ElasticEventSender** that is used to send events to Elastic, **ParquetEventSender** that is used to send the data used to improve the model to a Parquet folder or remote file system and **ConsoleEventSender**, used to print streaming data that gets consumed.

The abstract class **DataSender** is extended only by the **ElasticDataSender** class, that sends the training and labbeled data to elastic to be used to query for possible errors and mutation in the code shared by users.

The classes used to send events and data could be seen as **Forwarders** from the **Forwarder-receiver** design pattern, the consideration done in 4.1.2 are the same in this part, the original pattern is **Synchronous** but the concept and decoupling introduced by the pattern is useful to make the code more flexible to changes.

4.1.4 Error prediction

Prediction of errors is done by classifying snippets of code with some patterns of errors and solutions associated. When a user sends some code that should be labelled, the model will label the code with the neural network model and one of the label produced will be used to query ElasticSearch with the service described in 4.1.7, after the Solutions are returned by the ElasticSearch interface, the resulting recommendations will be passed to the Transaction associated with the request of the user that has issued the code labelling service along with the Mutations predicted.

4.1.5 Mutant prediction

The method described is the same as the Error Prediction 4.1.4, but the label used to query ElasticSearch will be the one associated to Solutions from the training data. The whole process described in 4.1.4 will be almost the same.

4.1.6 Model creation

The model used for the functionalities previously seen is built from a classifier trained with data from a dataset that contains errors and solutions (`https://github.com/program-repair/defects4j-dissection`). Other than this data, user could share further records of codes with Errors associated with Solutions, this code will be saved in a local repository to be used when the model will be rebuilt.

To build the model, the classes that extends the abstract class *Classifier* are used.

The *DirectClassifier* class associates with every record of the training set a new label(for both errors and solutions) and two feature vectors for the code with errors and for the solutions. These features and the labels created will be used to train a neural network as introduced in 4.1, that is a single hidden layer of 200 nodes, an input layer of 300 nodes(the dimension of the feature vector associated to code with errors and solutions) and the number of labels as the output(for the direct classifier, the number of labels is equal to the number of records used).

The *ClusterClassifier* is almost the same as the *DirectClassifier* but the labels associated to every record in the training set are estimated with a clustering method that for this project is Kmeans but could be also changed at will. The number of output nodes in the final layer of the neural network is the number of cluster that were found during the clustering step(for this

project, the number of clusters are *numRecords/3* as default, so the number of labels and output nodes in the final layer of the neural network is that as well).

At the end of this Pipeline described, the model will be saved in local to speed-up the loading process during development and testing phases, but this option could be turned off(by removing the loading process in DirectClassifier and ClusterClassifier or by removing the saved model from the */model* folder under the */mining* root of the sbt project, further development could deactivate directly this option by an environment variable or something like that).

The model created will be used to label the training data to be sent to ElasticSearch for the recommendation pipeline and incoming streaming data from users to get the recommendations from the previously sent training data in ElasticSearch.

There is also a problem in this moment with the model, the initial data is not enough to create a model that converges, the final model created by the Neural Network(multi-layer perceptrons) is unusable with the initial data. To further improve the model for convergence and the analysis of performance, some metrics could be controlled to find the perfect number of cluster that maximize/minimize this metric (for example the **silhouette** could be used to find the right amount of clusters) but this analysis is out of the scope for this project, as It only serve as an example of a micro-service distributed application.

4.1.7 ElasticSearch interface and the Broker-orchestrator

The ElasticSearch interface implements a **Remote Facade** interface to make calls to the **ElasticSearch** server service with the **index APIs** provided by the server in form of REST HTTP requests, used to query the system to find specific records that match a pattern.

The code structure as a UML diagram of the ElasticSearch interface can be seen in figure 4.

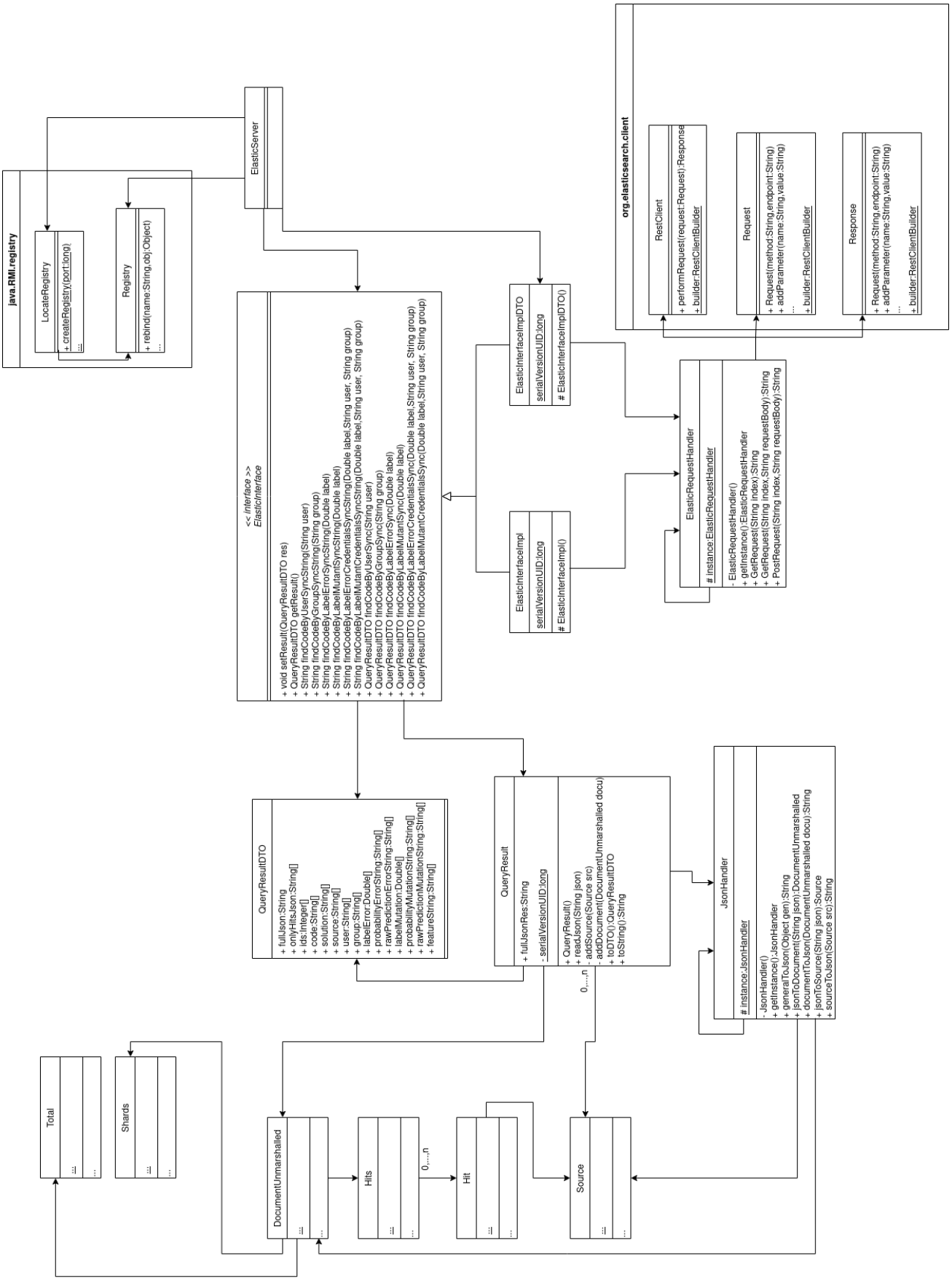

Figure 4: Elastic Interface Server code structure

The **ElasticInterface** is the interface used to communicate with the remote object and is shared between client and server.

The subclasses of ElasticInterface are two, but only one is used in the final project because there were some problems with the serialization of the unmarshalled Object from the queries. The **ElasticInterfaceImplDTO** is the only class used, this class will make the calls to ElasticSearch by using the singleton **ElasticRequestHandler** that implements a client to make the query requests and handling the returned results. The client is implemented in low level, that is an HTTP client with direct calls (the high level client was not working for some reason, probably due to the instability of the APIs and the fact that It is deprecated in the more recent version of the ElasticSearch API).

The returned results are parsed through the class **QueryResult**, that uses the **JsonHandler** singleton to transform the results from the query in an **UnmarshalledDocument**, this class will also provide the functionality to assemble the QueryResultDTO that will be returned to the client, this DTO will provide all the parts that the client could need.

The Unmarshalled document is the object obtained from the ElasticSearch query, the structure of this document is given by the following json:

```
{
  "took": ..., "timed_out": ...,"_shards": {
        "total": ...,"successful": ...,"skipped": ...,"failed":
           ...
    },
    "hits": {
        "total": {
            "value": n,"relation": "eq"
        },
        "max_score": ...,"hits": [
            {
                "_index": "primarydirect","_type": "classified","
                   _id": "...","_score": ...,"_source": {
                    "ids": ...,"code": "...","solution": "...","
                       source": "","user": "...","group": "...","
                       labelError": ...,"probabilityErrorString":
                       "...","rawPredictionErrorString":"...","
                       labelMutation": ...,"
                       probabilityMutationString":"...","
                       rawPredictionMutationString": "...","
                       featureString": "..."
                }
            },
            ...
            {
                ...
            }
        ]
    }
}
```

The useful part of the obtained json and unmarshalled object is the **Hits** list, that contains the list of solutions or mutation obtained. That is why this part will be the one returned to the clients, in particular the **Source** part, that contains the information about the recommended Solution or mutation.

The meaning behind querying ElasticSearch and limiting the results to specifics users and groups is to implement a RBAC where the roles are the combination of groups associated to users(one client could see his code, the global code, and the code of Its organization).

The remote part of this system is implemented via RMI [4], that is done by the following steps:

- The **ElasticInterface** extend the **Remote** object
- The implementation of ElasticInterface, that is **ElasticInterfaceImplDTO**, will implement the **UnicastRemoteObject** to be able to receive remote calls in Unicast(one to one).
- The **QueryResultDTO** needs to be serializable, along Strings and everything passed and returned by the ElasticInterface(the standard types and classes are serializable by default).
- the server needs an RMI registry to be able to create **skeletons** for the remote objects, this RMI registry is created at runtime because if It is already running in the background, a lot of problems with dependencies will occur so It is best practice to create the registry directly at runtime and let java and maven handle the class-paths.
- When the server is up, an ElasticInterface object is created(one of Its implementation) and registered to the RMI registry as a skeleton (the object will be binded by a name that is invocable by the client).

To understand the sequence of operations done in this part, the sequence diagram visible in figure 5 make the whole process clear.

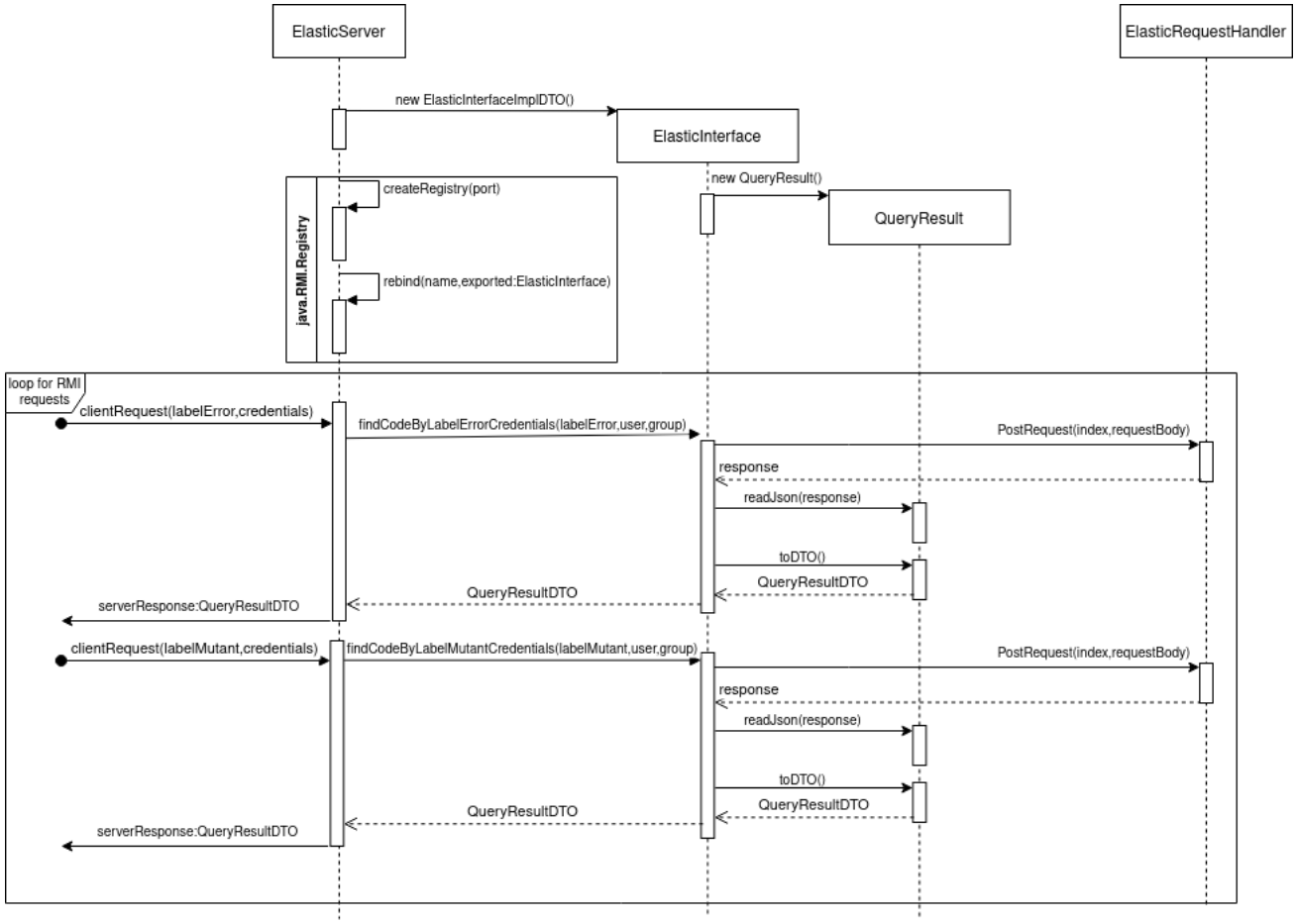

Figure 5: Elastic Interface server sequence diagram

The first thing that the ElasticSearch interface does is creating the RMI registry, creating the local *ElasticInterface* object(one of Its subclasses) and rebinding the object to the RMI registry to create the skeleton that clients will use to communicate with the original object.

The loop for RMI requests does the following things:

1. When a remote method invocation arrives from a client, the server delegates the call to the local object *ElasticInterface*.

2. The local object will use the *ElasticRequesthandler* to do the queries in an ElasticSearch index.

3. When the query finishes, the records are unmarshalled with the *QueryResult* method *readJson*.

4. After the records are inside the QueryResult, the *toDTO* method is called to get a DTO with all the useful information about the query.

5. The DTO is returned to the client.

The code structure of the broker-orchestrator(broker because It forwards the requests to the ElasticSearch-interface, orchestrator because to communicate and react to events in the distributed system, It uses an event-messaging system like Kafka) is visible in figure 6.

Figure 6: Elastic Interface Client code structure

The **ElasticClient** class is the main for the service, It is called ElasticClient because that is

Its main purpose.

The structure is almost the same as the ElasticSearch interface server, but there are some differences due to the fact that this component carries out more tasks and functionalities:

- This service listens to a kafka topic as a consumer, this kafka topic contains new labels along ids and credentials that will be used to query ElasticSearch through the interface previously defined.
- It also serve the role of the final step to get the results of the recommendation, because It writes the results in the Transaction Repository(REDIS) for the end-user to take.

The first step is to get the remote registry by localizing It with the standard library for RMI, that is *LocateRegistry*.

After the registry was found, the broker-orchestrator does a lookup to get the remote object as a stub(that serve as a proxy for the true remote object).

All the operation done to the *ElasticInterface* are carried by a proxy to delegate the calls and augment the level of abstraction, It also serve as a protection proxy for calls that are not permitted and for possible errors that could occur during the remote calls.

The broker waits for new records in the kafka topic. When new records arrive, It consume every record one by one by querying the ElasticSearch interface through the proxy, when the results arrive in the form of DTO, some of the information contained in the DTO are used to write the final product to the REDIS repository through *Jedis*(a redis client), if some error occured during execution, the current transaction status is updated in REDIS as **ERROR** like what was done during the prediction pipeline for the labels.

To enhance and build a more consistent structure for the Senders and Receivers linked to all the parts that use the event-messaging system(that is Kafka), these classes and services could implement a **Builder** pattern to define the connection options when the object is created and decouple the options creation to the communication Class. For example the broker-orchestrator could get the Kafka-consumer from a KafkaConsumerBuilder and delegate the option setting directly to this builder

The sequence of calls done during execution is visible in figure 7.

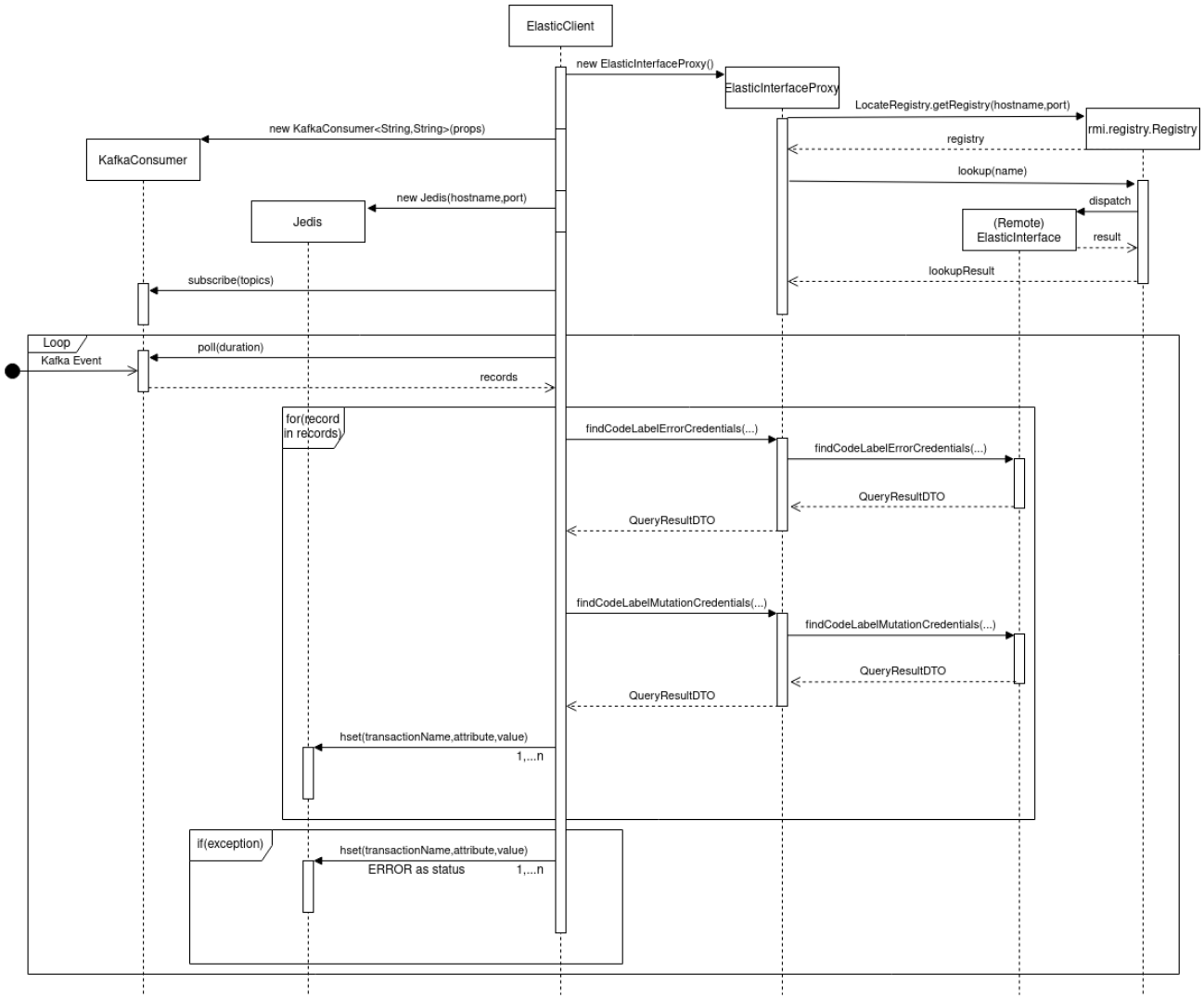

Figure 7: Elastic Interface orchestrator sequence diagram

The steps previously introduced are clearly seen:

1. The broker(ElasticClient) creates the Proxy for the ElasticInterface.
2. The *ElasticInterfaceProxy* locate the registry and does a lookup to get the remote object as a **stub**
3. The broker creates a new *KafkaConsumer* and subscribes to a series of topics(only one in this case).
4. The REDIS client is instantiated, that is *Jedis*.
5. When new records arrive to the Kafka topic, the broker does the following things for every record:
   - 5.1. The broker queries ElasticSearch through the ElasticInterfaceProxy, the proxy calls the methods to get the code for specific label(for errors and mutations) in the stub and get some results(if the calls were successful), the results are returned to the client.
   - 5.2. The broker extrapolates some information from some of the fields of *QueryResultDTO* and writes them in the Transaction Repository along the transaction status(**FINISHED**)

The broker-orchestrator could also be seen as a choreographer because It implements both elements of an orchestrator (because It uses an Event based messaging and communication framework, to wait for new labelled code) and a choreographer(because It uses direct connections to the ElasticInterface as well as the REDIS repository, even though the redis repository is centralized).

4.1.8 The recommendation service workflow

The prediction service is divided in three main components(two main runned in two different containers that use the same code structure 4.1.1, that is *MainClassification* and *MainImprove-Model*, and the broker-orchestrator used to query ElasticSearch and update the Transaction repository with the results obtained).

The **Classification** service labels firstly built the model(or load if available), model creation is described in 4.1.6. After building the model, It sends the data used for training the model in ElasticSearch with the *ElasticDataSender*4.1.1, these data will be those that are recommended to users that want to use the system.

After the model creation and the recommending repository creation in ElasticSearch, the system listen to the topic **usercode** for new data sended by users, when the data arrives in the form of a stream, these data will be cleaned, controlled and labelled by the *Classifier* implementation.

After the model have finished with the data in the stream, the predicted labels are sent to kafka in order to be consumed by the Broker-orchestrator described in 4.1.7.

The **Improve-Model** service will listen to new training data that could be used to improve the overall results and make the system return more accurate results to the user. This component will listen to a kafka topic(improvemodeldata) and when new record arrive, these new data will be saved locally to be used in future model building.

The Broker-orchestrator will subscribe to the topic **labelleddata** and will consume new records that arrive, the records that arrive in that topic are formed by the predicted labels, the Transaction ID used to index the **Redis repository** for status and results of the transaction, the user that has issued the call(the one specified in the REST API http request in the json body) and the group of the user(specified as well in the json body passed in the POST request).

The received record will be used to query ElasticSearch through the Elastic-interface-server service, these service was described in 4.1.7 and implements a Remote Facade that returns DTO that encapsulates the results of several calls to ElasticSearch in a cleaned format.

The results obtained are written to REDIS, if some error has happened during execution, the Transaction Status in REDIS is updated to **ERROR**.

4.1.9 Testing

For **integration testing** in the labelling pipeline service, the only part that was needed to be tested was the **KafkaEventConsumer** because Event and Data senders are not testable.

The *KafkaEventConsumerTest* tests the types, consistency of the methods and returned values of the class *KafkaEventConsumer*. To do these types of test, the *SparkFacade* is mocked with **scalamock** and the mocked object is passed with dependency injection to the KafkaEvent-Consumer.

The tests of *KafkaEventConsumerTest* see if the following conditions are respected:

- If the dataframe returned is not null or empty.
- If the dataframe returned has the right schema.
- If the dataframe returned contains the right number of records.
- If the whole unmarshalling logic behind is right.

For **unit testing** in the labelling pipeline service, the parts tested are the ones related to the classifier and the singletons that do some local work, that is:

- **ClassifierTest** tests if the classifier and Its subclasses is working properly, the dependency of the *SparkFacade* is injected as a mock.
  - if both direct and cluster classifier return non-empty dataframes.
  - if both direct and cluster classifier return the right number of records.
  - if both direct and cluster classifier return the right schema.
- **CodeCleanerTest**
  - See if the String returned is formed only by the right keywords of the programming language(java).

The **broker-orchestrator** is not tested because It implements a simple loop and uses predominantly classes from libraries, so already tested code is used.

The **ElasticSearch-interface** tests the following functionalities:

- **QueryResultTest** that tests for some functionalities provided by the class that also implements a DTO assembler.
- **ElasticInterfaceTest** is not yet fully tested, but in the future It will test the calls and the returned results of the methods that are called by the client, this will be done by mocking the **ElasticRequestHandler** and changing the class a little to pass the request handler as a dependency (with a setter method or with the constructor)

4.2 Service communication

The services communicate in two different ways between them, as It was already stated in previous chapters, the main communication interface between services is Kafka to create events and REDIS as a repository where all the services could write and read for status.

Some services communicate directly with each other like the **broker-orchestrator** and the **ElasticSearch-interface**.

The choice for an architecture more closer to orchestration than choreography was because an event-based interaction is more suited for responsive asynchronous systems and because Kafka is reliable.

4.3 REST APIs

The REST APIs are the secondary interface to the system(where the primary interface is the web interface, but the front-end is not yet implemented), these APIs will be available to developers and authorized clients that needs full access to the system, but for now they are the first access to the system.

As already stated at the start of this main chapter 4, the operations available to the client are GET to get Transaction Statuses, POST to start a transaction to get some Solutions and Mutations to the code passed, PUT to send some Code with errors and a Solution with no Errors to improve the model at the core of the system and, finally, DELETE to delete a transaction.

The code structure that implements the Spring server that takes HTTP request is visible in figure 8

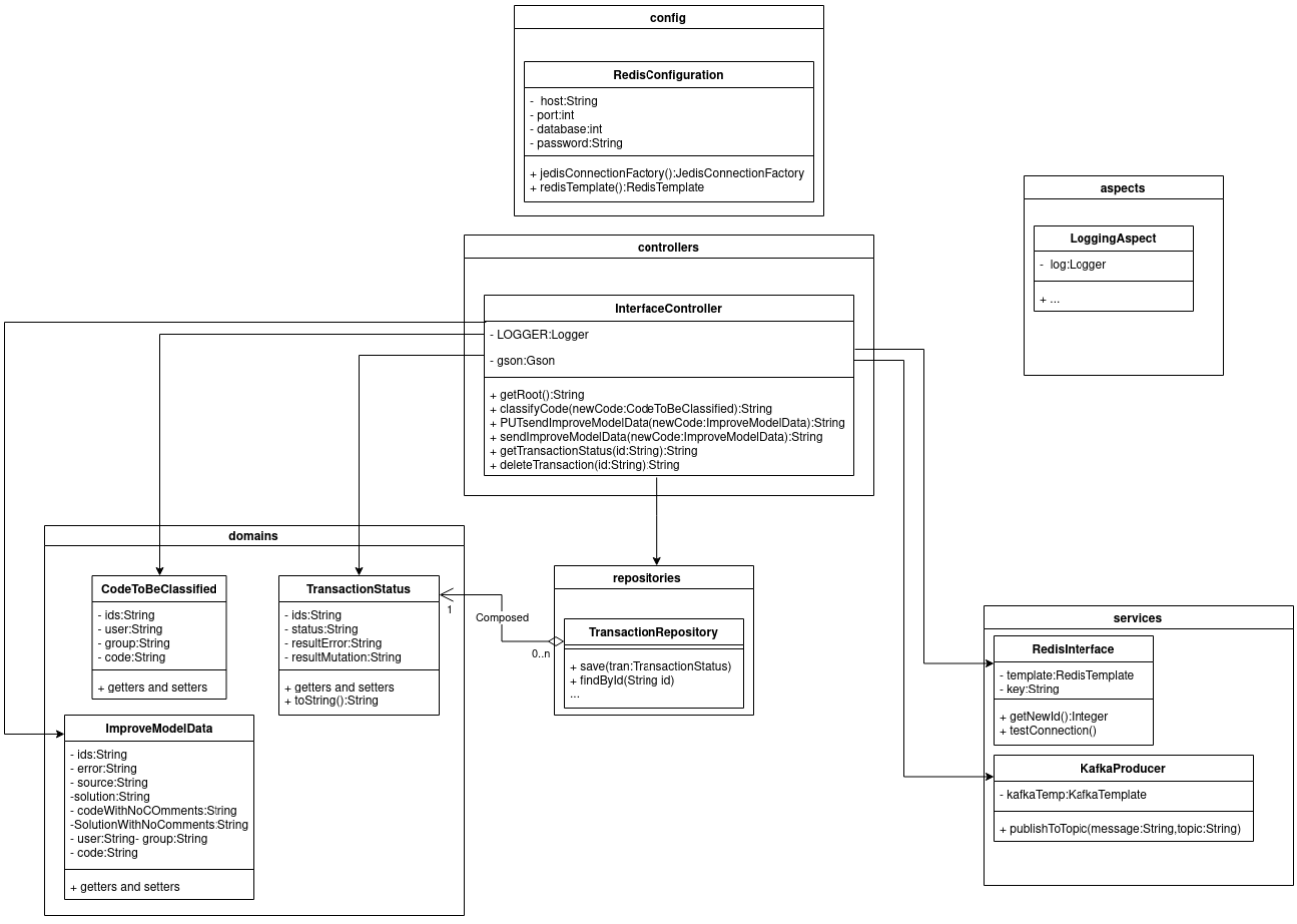

Figure 8: code structure of the Spring REST server

The Code structure respects some common best practices with Spring projects, where different components of Spring are divided in sections/subpackages to make the code easily extensible with other functionalities.

The **domains** package contains the resources and the schemas of the data used in the system that is:

- **CodeToBeClassified** is the code sent with the POST, this code will be labelled and used to recommend Solutions and Mutations.
- **TransactionStatus** is the object that is saved to REDIS(in the form of an hash), this object represent the status of the transaction issued by the user and the future result.
- **ImproveModelData** is the code sent with the PUT, this object contains two snippets of codes, one is the code with errors, the other is the solution that has resolved the problem, the object contains other fields that will not be used to improve the model but will be really useful to the end user that will get more information on the recommended Solution and Mutation.

All the classes in domain are Validated through the **@Valid** keyword in Spring.

The **repositories** package contains the components that make access to a repository, the only class in this package is **TransactionRepository** that access the Redis server to get or save Transactions. This class implements the **CRUDRepository** to get the classic set of operations for a repository.

The **config** package contains classes that will be used to instantiate beans in the Spring environment, the only class in this packate is **RedisConfiguration** used to instantiate the **RedisTemplate** used in the *RedisInterface*. The parameter used to configure and instantiate the RedisTemplate are taken from the Spring properties.

The **services** package contains classes that do some business logic in the code. The two classes contained in this package are:

- **RedisInterface** used to access the Redis server to get new Ids for the transactions and to test for the connection to the Redis server.
- **KafkaProducer** used to send new events to a kafka topic with the *publishToTopic(...)* method

The **controllers** package contains the REST controller, that is **InterfaceController**, that implements the routes and the operations done by the server. The *InterfaceController* has a *Logger* to print minimal information(all the logging is done in the aspect that will be seen shortly), the method visible in the UML implement the routes of the server and the different types of HTTP requests.

The **aspects** package contains the aspects of the system. For this project there is only one aspect, **LoggingAspect** to log the requests from clients and errors that could occur during execution.

The general workflow was introduced in 3, the client does an HTTP request to the server, the server will return to the client the TransactionStatus that contains the ID used to query periodically the server to get the current status of the transaction and the results.

The pattern implemented by this system is **asynchronous request-reply** seen in figure 9, this pattern implements a communication framework for the client that is asynchronous in principle, the client will make a request for a transaction Token that will be used to query the APIs until the transaction has not succeded.

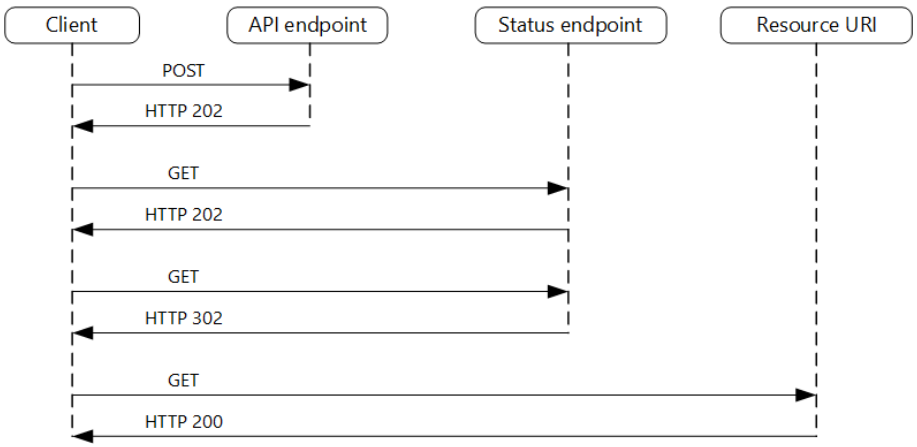

Figure 9: asynchronous Request-response pattern

The pattern sequence of calls done during execution is seen in figure 10.

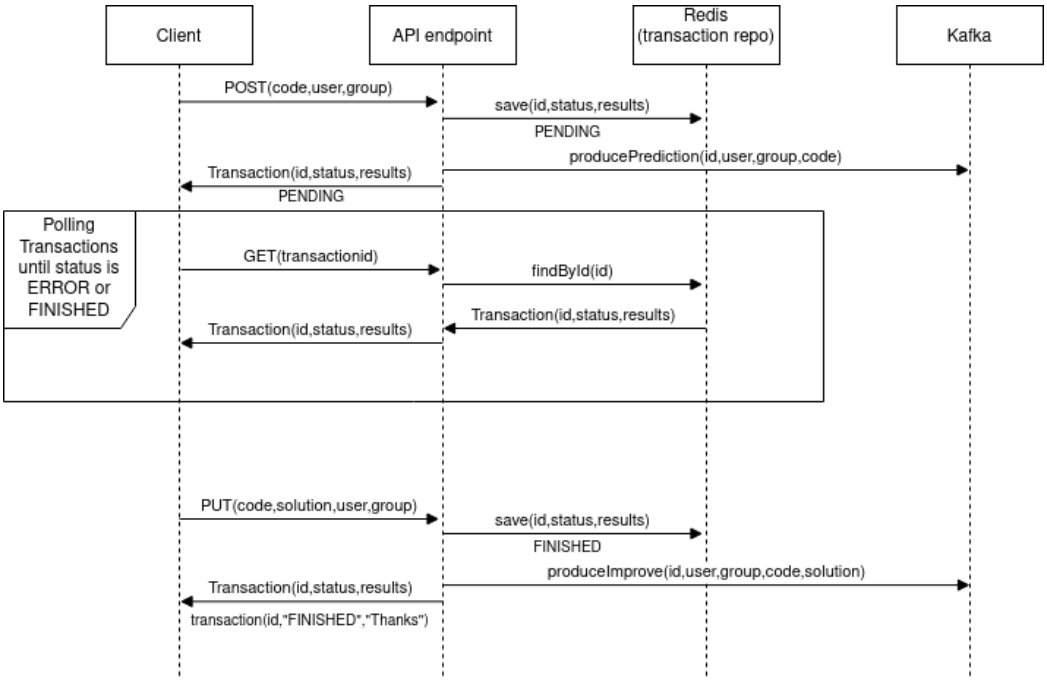

Figure 10: sequence diagram for the REST APIs

The process of the REST interface is described by the following steps:

1. The client will send some code to get some recommendations with a POST request.
2. The REST server will send the code to Kafka to start the streaming pipeline to label the code of the user and will return the Transaction to the user.
3. The user will query periodically the REST server to get the Transaction along with Its status, when the status is FINISHED(or an ERROR has occurred), the user will have a list of recommendations for Solutions and Mutations.
4. The user will share Its code to improve the model with a PUT request.

The REST API could also be used directly by admins or authorized users, for now the project uses this APIs as the main interface for the client to communicate to the whole system, at least

until when the Web interface is implemented alongside the Authorization and Authentication framework.

No controls over what is passed other than simple validation are done, because the controls should be done before by the web interface, for example when the users pass code with some comments, the code must be refused because the comments are not treated well by the system and the labelling pipeline.

4.3.1 Testing

For **unit testing**, the only class tested is **TransactionStatus** and the ToString method, because It is used directly to transform the *TransactionStatus* object in a json that will be passed to the user.

For **integration testing**, the **InterfaceController** is tested along with all the possible HTTP requests to the server. The server is Mocked(as a MockMVC), along with the repositories(*TransactionRepository*) and the services(*RedisInterface* and *KafkaProducer*). Every endpoint is tested for returned value, returned status code, returned response body, possible exceptions and for deletion of nonexistent Transactions.

Other tests could be done, but the other parts of the system are simple clients or components that use library classes, so everything is already tested(theoretically).

4.4 Client web interface

Under development

4.4.1 Authorization

4.4.2 Authentication

4.4.3 Testing

5 Future outlooks

A **Remote Proxy** could be used to control access to the database where all data is stored and maintained, along with information about users of the system. Another level of protection could added along the proxy, with patterns of reference monitoring and protection proxy (implemented with the proxy previously mentioned). The proxy could be used as well to control the access to some resources of the system (probably as a reverse proxy to load balance the requests for the bugs).

All this Authorization logic could be controlled by a (Reference Monitor or RBAC) that is used along the proxies to control and authorize access from users to resources and utilities.

There is no control over the DELETE operation over transactions, so a control could be if the resource is from the user that has done the request, or from an admin.

The model created could be more useful and better in performance, because It does not converge, and the number of training records could be augmented to build a better model.

The **Web interface** should be completed because the REST APIs give complete access to the system, with no Authentication.

References

[1] Dissection. defect4j dissection dataset. `"https://github.com/program-repair/defects4j-dissection"`, last visit 25/10/2021.

[2] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space, 2013.

[3] Spark. Spark word2vec. "https://spark.apache.org/docs/latest/api/java/org/apache/spark/mllib/feature/Word2Vec.html", last visit 25/10/2021.

[4] Oracle. rmi tutorial. https://docs.oracle.com/javase/tutorial/rmi/, last visit 09/11/2021.