CS5722 Project

VoD Recommendation System

Team Members

Name	ID
Assad Nawaz	19085842
Reezvee Sikder	15140997
Shravan Chandrashekharaiah	18043038

Table of Contents

Report Checklist

Item		
I.	Table	Check
II.	Researched Patterns	check
III.	Class Diagram(s)	check
IV.	Code Submission	check
V.	Use of GitHub	check
VI.	Added Value	check

Requirements Table

I.	A	Continuation
I.	B	Same team members
I.	C	See description below
I.	C. ii)	Ruby
I.	C. iii)	Factory, Decorator, Singleton, State, Façade, Observer
I.	C. iv)	REST Arch Pattern, ORM, Microservices, Multi-tenant Arch
I.	C. v)	Total Number of packages 5 Total number of classes 40 Total number of files 432 Total number of lines of code 5769

III. Requirements

This project is an extension of the project submitted by the group of CS5721 Software Design, which was based around the idea of multi-tenant Video on Demand platform, wherein any media company will be able to upload videos and create a website of their own brand. Instead of hosting or promoting respective content in YouTube, this will allow you to create your own website and host content to address those particular users in the way they want.

Based on the backbone of the Video on Demand platform, we have built a recommendation system for the platform, which was done keeping in mind that it should not impose any constraints on existing business rules. The added functionality includes giving recommendation to the user about videos using deep learning framework and also let the user know whether this video has a good or bad review.

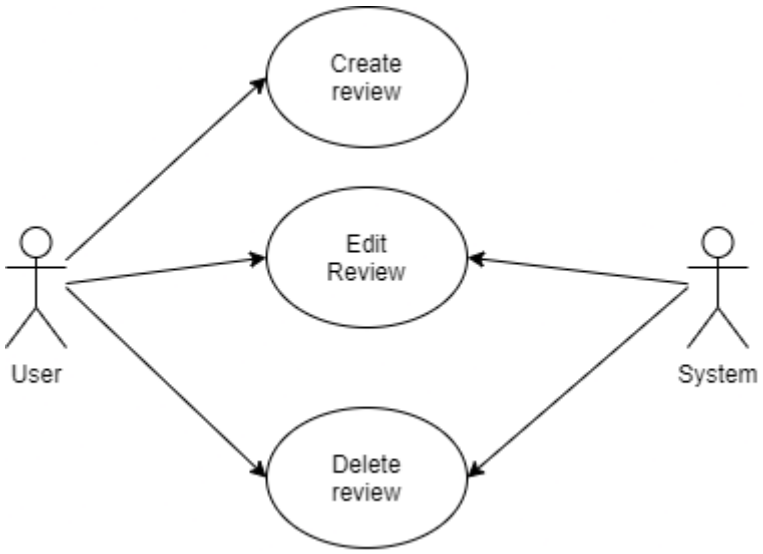
A general outline of the new work undertaken in this project is as follows:

- A recommendation engine was built using the deep learning framework like TensorFlow.
- A sentimental analysis was also performed on the reviews added on the videos to give a general idea to the user about the video.
- The sentimental analysis will also delete the reviews if it has bad language.
- Previously in the CS5721 project VOD platform was directly interacting with the microservices, that has now been refactored and now Interceptor pattern has been incorporated.
- The Template Method was incorporated.
- The Command Design Pattern was incorporated.
- The Visitor Pattern was incorporated.
- The Builder Design pattern was incorporated.
- The Memento Design pattern was incorporated.
- The Factory Design pattern was incorporated.
- The Composite Design pattern was incorporated.

Functional Requirements

- Give recommendation to the user for the videos they might like.
- Add reviews about the videos and perform sentimental analysis and display the result of how many good and bad reviews for that particular video.
- Delete a submitted review if it contains bad language.
- Create a business group.
- Create multiple tenants for a business group.
- Create applications for each tenant.
- Upload videos to the platform.
- Update video information.
- Publish videos to the platform.

Use Case Diagram



Use Case Descriptions

Use Case 1	Make a review
Goal in Context	User wants to create a review for a piece of content
Preconditions	User must have an account and be logged in to the system
Success Postcondition	User successfully creates a review

Failed Postcondition	User has not created a review
Actors	User, System
Description	Step | Action 1 | User logs in 2 | User navigates to a movie 3 | User clicks create review button 4 | User enters review data and submits 5 | System checks review and updates it to the site
Extensions	Step | Branching Action 5 | System deems review content to be inappropriate and deletes review

Use Case 2	Give movie recommendation
Goal in Context	The system gives certain movie recommendation to the user
Preconditions	The user must have an account made and be logged in to see the recommendations
Success Postcondition	User is able to see recommended movies
Failed Postcondition	User sees default movies that weren't recommended to him/her
Actors	User, System
Description	Step | Action 1 | User logs in 2 | User watches some movies 3 | User checks recommended movies based on watch history and ratings
Extensions	Step | Branching Action 3 | User has not seen any movies on the platform so their recommendations are default recommendations

Non-Functional Requirements

Security

- Passwords should not be visible at any time in the system, it should be transmitted to the server side as hash and never as a clear text.
- User account should not be able to access API which they are not supposed to.
- Administrators should have all the rights and access.
- Recommendation of a particular user should be visible to that user and not to any other user.
- Any known vulnerabilities supplied by the third-party frameworks should be monitored.

Portability

- The system should not be constrained to any specific hardware or operating system.

Extensibility

- The system architecture should be built in such a way that any new component can be added to the existing flow without causing any alteration to the logical flow.

Scalability

- The system should be scalable at any point of time.

Tactics Used to Support Quality Attributes

Security

- Perform multiple layers of validation to make sure data integrity is retained.
- Using defensive programming to make sure that the interface is not exposed to any external abuse.

Extensibility

- Any new feature can be added to the system without lot of changes.
- Our system uses the feature of microservices and many design patterns specifically interceptor to make sure that the system is extensible.

- By implementing interceptor, we get to control events proceed across the system, thus improving the extensibility.
- The interceptor just needs to understand the how to interpret and process the events, it does not need to understand the service specifications.
- By using microservice each service is deployed separately and still manage to communicate with each other well-defined interfaces.

Portability

- The recommendation system's framework is developed in python which can be run on any operating system machine by using python interpreter.

Scalability

- The system should be built in such a way that it should be able to make scalable deployments.
- The framework should be able to support scaling on platforms like AWS and GCP.

Quality Assurance:

- In this system we have used TDD in jasmine/karma framework to make sure that quality work is being achieved.
- We have also incorporated SonarQube to take care of the code smells.

IV. Design Patterns

Interceptor Architectural Pattern

The Interceptor architectural pattern allows services to be added transparently to a framework and triggered automatically when certain events occur.

In our project, we have integrated an interceptor design pattern to check the subscription for particular services like recommendation service or review service. And also, to check the integrity of the data before the request is forwarded to the respective service.

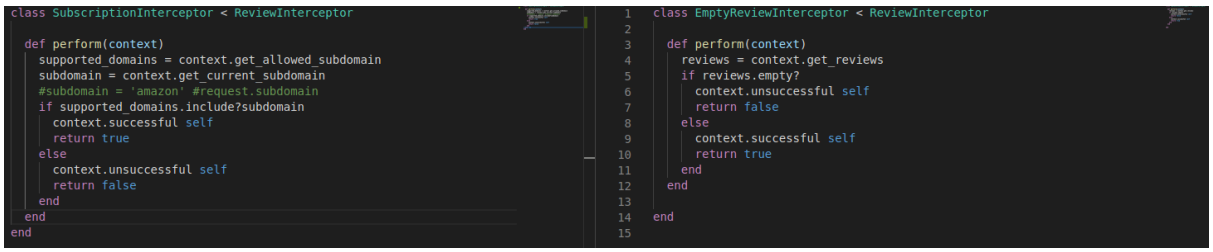
It is done by making the controller action subscribed to the interceptor. When the particular request is received then automatically the dispatcher and its registration with the interceptor will happen. The dispatcher then starts calling the interceptors with the context object which contains all the information that the interceptor would require to perform actions on the request [2].

Context Integration:

```
class SubscriptionContext < ReviewContext

  def initialize(params, request)
    @reviews = params[:review] #params.require(:review).permit([:text])
    @allowed_subdomains = ['amazon', 'hbo']
    @request = request
  end

  def get_allowed_subdomain
   return @allowed_subdomains
  end

  def get_current_subdomain
    @request.subdomains[0]
  end

  def get_reviews
   return @reviews[:text]
  end

  def successful interceptor
    Rails.logger.info("Subscription Interceptor successful--->#{interceptor.class}")
  end

  def unsuccessful interceptor
    Rails.logger.info("Subscription Interceptor unsuccessful--->#{interceptor.class}")
    #render json: {'message': 'Subscription not found for the service'}
    return
  end


end
```

From the above, it is clear that the interceptor would require information about the context, i.e. information on allowed domains and then the review data. Once the interceptor does the required action on the contextual data it will call the successful/unsuccessful method in the context to further control/monitor each interceptor points.

Interceptors:



```ruby
class SubscriptionInterceptor < ReviewInterceptor

  def perform(context)
    supported_domains = context.get_allowed_subdomain
    subdomain = context.get_current_subdomain
    #subdomain = 'amazon' #request.subdomain
    if supported_domains.include?subdomain
      context.successful self
      return true
    else
      context.unsuccessful self
      return false
    end
  end
end
```

```ruby
class EmptyReviewInterceptor < ReviewInterceptor

  def perform(context)
    reviews = context.get_reviews
    if reviews.empty?
      context.unsuccessful self
      return false
    else
      context.successful self
      return true
    end
  end

end

```

Two interceptors' areas above, one to check for subscription and another to check the data integrity. Each interceptor has to implement the perform action like it is mentioned in the interface Review Interceptor.

Dispatcher:

```ruby
class SubscriptionDispatcher < ReviewDispatcher

  def initialize(interceptors)
    @interceptors = interceptors
  end

  def event_triggered context
    @context = context
    dispatch
  end

  private
  def dispatch
    @interceptors.each do |interceptor|
      status = interceptor.perform(@context)
      return status unless status
    end
  end

end
```

From the above, once the event is triggered dispatch method will be called where all the registered interceptors will be called to perform its action.

Factory Design Pattern

Factory Method is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

In our project, we have used it to instantiate the object related to recommendation implementation. In our project, there are two types of recommendation available i.e. content-based recommendation and collaborative recommendation.

We have used to factory method to instantiate either of the recommendation based on the argument passed to the factory method.

```python
from app.helpers.factory.recommendation_engine import RecommendationEngine
from app.helpers.factory.content_based_v1 import ContentBasedV1

class MovieRecommendation(RecommendationEngine):

  def get_engine(self, engine_type):
    if engine_type == 'content_based':
      return ContentBasedV1()
    elif engine_type == 'collaborative':
      return CollaborativeV1()
    elif engine_type == "hybrid":
      return HybridV1()

    return ContentBasedV1()
```

Factory method implementation is as above which takes an argument about what type of recommendation is required and then the factory method will instantiate the concrete product class.

```python
from flask import request, jsonify, render_template

from app.helpers.factory.movie_recommendation import MovieRecommendation

def index():
  engine = MovieRecommendation()
  movies = engine.recommend_movies('content_based')
  return jsonify({'data': movies})


def home():
  return render_template('index.html')
```

The controller is the client here which calls the factory method with the argument. This will then return the recommendation.

Builder Design Pattern

Builder is a creational design pattern that lets you construct complex objects step by step. The pattern allows you to produce different types and representations of an object using the same construction code.

In our project where we have implemented the recommendation engine, it involved various steps to recommend content. We identified various complex objects in this process and used a builder pattern to access these objects with the same instantiation method and using the director we have defined two different methods to get the recommendation.

This Genre Based in the Product class which implements the recommendation based on genres. This allows defining another type of recommendation easily by defining new class and pass this product object to the builder.

```python
from app.helpers.builder.recommender import Recommender
from app.helpers.builder.genre_based import GenreBased

class ContentBasedBuilder(Recommender):

  def __init__(self):
    self.reset()

  def reset(self):
    self._product = GenreBased()

  @property
  def product(self):
    product = self._product
    self.reset()
    return product

  def users_features(self):
    self._product.add( 'users_features' )

  def rank_feature_relevance(self):
    self._product.add( 'rank_feature_relevance' )

  def user_ratings(self):
    self._product.add( 'user_ratings' )

  def normalization(self):
    self._product.add( 'normalization' )

  def recommendation(self):
    self._product.add( 'recommendation' )

  def format_recommendation(self):
    self._product.add( 'format_recommendation' )
```

From the above, it is evident that on changing the product there will be a completely different integration of recommendation.

```python
class Director:
  def __init__(self):
    self._builder = None

  @property
  def builder(self):
    return self._builder

  @builder.setter
  def builder(self, builder):
    self._builder = builder

  def recommendation_with_normalization(self):
    self._builder.users_features()
    self._builder.rank_feature_relevance()
    self._builder.user_ratings()
    self._builder.normalization()
    self._builder.recommendation()
    return self._builder._product.evaluate()
    #return self._builder.format_recommendation()

  def recommendation_without_normalization():
    self._builder.users_features()
    self._builder.rank_feature_relevance()
    self._builder.user_ratings()
    self._builder.recommendation()
    return self._builder.format_recommendation()
```

The Director class is responsible for getting the flow done properly. This will set the builder task and once the evaluation is called on the product it will aggregate all the function and perform recommendation function and return the results.

Composite Design Pattern

Composite is a structural design pattern that lets you compose objects into tree structures and then work with these structures as if they were individual objects.

In our project we have integrated sentiment analysis for the movie reviews, to give user idea about the content they are watching. Apart from the rating, sentiments in the movie are also captured. In this process, there are two process training and then prediction. Training happens when new data is fed to the system from the CMS with the labels on the exiting model. And prediction happens when the user submits a review to the system about the movie. Since the training model can be done in many ways, where there involve many different steps in training the model. And each step can have many different ways to do it. For example, processing the text for different languages could be different, there could many different approaches to building the model and also saving the model to either local or on the cloud.

To tackle the above scenarios, we chose the composite design pattern, where we can construct a tree-like structure, choosing the best possible combination and then execute it accordingly.

```python
from app.helpers.composite.components.component import Component

from app.helpers.memento.originator import Originator
from app.helpers.memento.care_taker import CareTaker

class Composite(Component):

  def __init__(self, message=""):
    self._children  = []
    self.originator = Originator(message)
    self.caretaker  = CareTaker(self.originator)

  def add(self, component):
    self._children.append(component)
    component.parent = None

  def operation(self, obj):

    for child in self._children:
      print(child)
      obj = child.operation( obj )

    return obj
```

The code above explains how to add the branch to the tree. Add function basically adds up the branch and operation will then execute the branch functionality from top to bottom.

```
    self.tree = Composite()

    self.english_branch = Composite()
    self.english_branch.add(English( ))
    self.english_branch.add(KerasTokenizer( ))

    self.model_branch = Composite()
    self.model_branch.add(Simple())
    self.model_branch.add(SentimentPrediction())
    self.model_branch.add(Local())


    self.tree.add(self.english_branch)
    self.tree.add(self.model_branch)
```

The above code shows how to construct the tree structure from various components. This led us to easily add more functions and variations easily to the process of training or prediction.

Command Design Pattern

The command is a behavioural design pattern that turns a request into a stand-alone object that contains all information about the request. This transformation lets you parameterize methods with different requests, delay or queue a request's execution, and support undoable operations.

Whenever a review is submitted to the platform, there are certain rules to be satisfied before it is saved inside the database. And if the rules are not matched an undo operation on the database is done to remove it. We have identified that the command pattern would do this action.

We have identified all the rules and then parameterized it inside the invoker, and then once the invoker is filled with all the information operation is performed on the invoker. To help an invoker we have created a receiver class to help it with certain tasks like removing emoticons and validating profanity.

```
def validate_review data
  invoker = Invoker.new
  invoker.check_charecter_limit = SimpleCheck.new(data)

  receiver = Receiver.new
  invoker.check_abuses = ProfanityCheck.new(receiver, data['text'])

  invoker.perform_validation
end
```

This shows the client code which in this case is model. It initializes the Invoker and fills in all the information it requires to check all the rules for data persistence.

```ruby
class Invoker

  def check_charecter_limit=(command)
    @limit_check = command
  end

  def check_abuses=(command)
    @abuse_check = command
  end

  def perform_validation
    @limit_check.execute if @limit_check

    @abuse_check.execute if @abuse_check
  end
end
```

This shows the implementation of the invoker class. Here perform_validation checks all the rules before it saves. If any of the rules failed then an undo operation is triggered.

Template Design Pattern

Template Method is a behavioural design pattern that defines the skeleton of an algorithm in the superclass but let's subclasses override specific steps of the algorithm without changing its structure.

In the project, we have integrated the template method for maintaining various different tree structures for training and prediction by defining different template methods for performing a specific type of operation. If a new tree structure is identified a new template method can be created and then use that method to access that particular operation.

```python
class Template:
  def __init__(self):
    self.tree = Composite()
    self.branches = []
    self.texts    = None
    self.labels   = None

  def prediction_template(self, texts):
    self.texts = texts

    self.branches.append(self.text_data())
    self.branches.append(self.prediction())

    self.add_branches()
    self.operate()

    return self.response_format()


  def training_template(self, texts, labels):
    self.texts  = texts
    self.labels = labels

    self.branches.append(self.text_data())
    self.branches.append(self.train())

    self.add_branches()
    self.operate()

    return self.response_format()
```

Two template method in the above code constructs two different tree structure using composite design pattern and serve the prediction and training facility to the controller.

```
from app.helpers.template.template import Template

from app.helpers.composite.composite import Composite
from app.helpers.composite.components.languages.english import English
from app.helpers.composite.components.tokenizer.keras   import KerasTokenizer

from app.helpers.composite.components.models.simple           import Simple
from app.helpers.composite.components.predictions.sentiment   import SentimentPrediction
from app.helpers.composite.components.saving.local            import Local

class BasicSentimentPrediction(Template):

  def __init__(self):
    super(BasicSentimentPrediction, self).__init__()

  def text_data(self):
    english_branch = Composite()
    english_branch.add(English( ))
    english_branch.add(KerasTokenizer( ))

    return english_branch


  def prediction(self):
    model_branch = Composite()
    model_branch.add(Simple())
    model_branch.add(SentimentPrediction())
    model_branch.add(Local())

    return model_branch
```

The above code shows the subclass implementation that implements text_data and prediction function which gets overridden from the base class template method.

Memento Design Pattern

Memento is a behavioural design pattern that lets you save and restore the previous state of an object without revealing the details of its implementation.

The sentiment analysis process involves various steps to be performed like pre-processing of text, obtain word embeddings, build model, train model, etc. This can get sensitive to the data that it receives and prone to break with minor variations. In order to keep track of each step we have integrated the memento design pattern to save each step result and keep it for further analysis of the failure process.

```python
from app.helpers.composite.components.component import Component

from app.helpers.memento.originator import Originator
from app.helpers.memento.care_taker import CareTaker

class Composite(Component):

  def __init__(self, message=""):
    self._children  = []
    self.originator = Originator(message)
    self.caretaker  = CareTaker(self.originator)

  def add(self, component):
    self._children.append(component)
    component.parent = None

  def operation(self, obj):

    for child in self._children:
      print(child)
      obj = child.operation( obj )

    return obj


  def save_and_operate(self, obj):

    for child in self._children:
      self.originator.record(child)
      obj = child.operation( obj )
      self.caretaker.backup()

    return obj
```

The above code shows how the memento is used, where composite design pattern integration is the client code for the memento design pattern. Here basically for each execution of the branch method, it is recorded in the memento and once the execution is done, it is stored for further analysis by taking a backup. This way it helps keep a record of the operation of training and prediction

Visitor Design Pattern

The visitor is a behavioural design pattern that allows adding new behaviours to existing class hierarchy without altering any existing code.

In the project, there are situations where a user is required to view all the reviews that were given by the user and perform certain actions to it. In order to give this information, we have identified the visitor design pattern to actually construct the data for this review. The visitor pattern will allow us to expand the new behaviour in this case.

```ruby
def construct_details ratings
  components = [UserComponent.new, ItemComponent.new]

  rating_visitor = RatingVisitor.new
  components.each do |component|
    ratings = component.accept(rating_visitor, ratings)
  end
  return ratings

end
```

From above we can see the client code that is rating model, which will list down all the components required to construct and then pass each component with the visitor object. This will then visit the Visitor class and then as already defined in the visitor class that particular function will be called on the components.

```
class RatingVisitor < Visitor

  def visit_item_component component, args
    return component.item_details(args)
  end

  def visit_user_component component, args
    return component.user_details(args)
  end

end
```

From the above, it shows visitors call the specific function that it requires from the component. A different visitor method could serve different purposes from the component. The behaviour of the component can be expanded by defining more visitor methods [1].

Service Locator

The goal of this pattern is to improve the modularity of your application by removing the dependency between the client and the implementation of an interface.

In our project have integrated service locator pattern to redirect the API request to the actual service location. This design pattern is integrated in the VOD platform which acts as an API gateway. For each API call, the VOD platform receives it will initialize service instantiation where it will take in the details about the controller and the action and then it will dynamically construct the API request and then transfer the call to the particular service location.

```ruby
class Admin::ListsController < ApplicationController
before_action :initialize_service

def index
  data, status = @service.communicate
  render json: data, status: status
end

def show
  data, status = @service.communicate
  render json: data, status: status
end

def create
  data, status = @service.communicate({authorize: true})
  render json: data, status: status
end

def update
  data, status = @service.communicate({authorize: true})
  render json: data, status: status
end

def destroy
  data, status = @service.communicate({authorize: true})
  render json: data, status: status
end

private

def initialize_service
  @service = ListMicroService.instance
  @service.attributes(request, params)
end
```

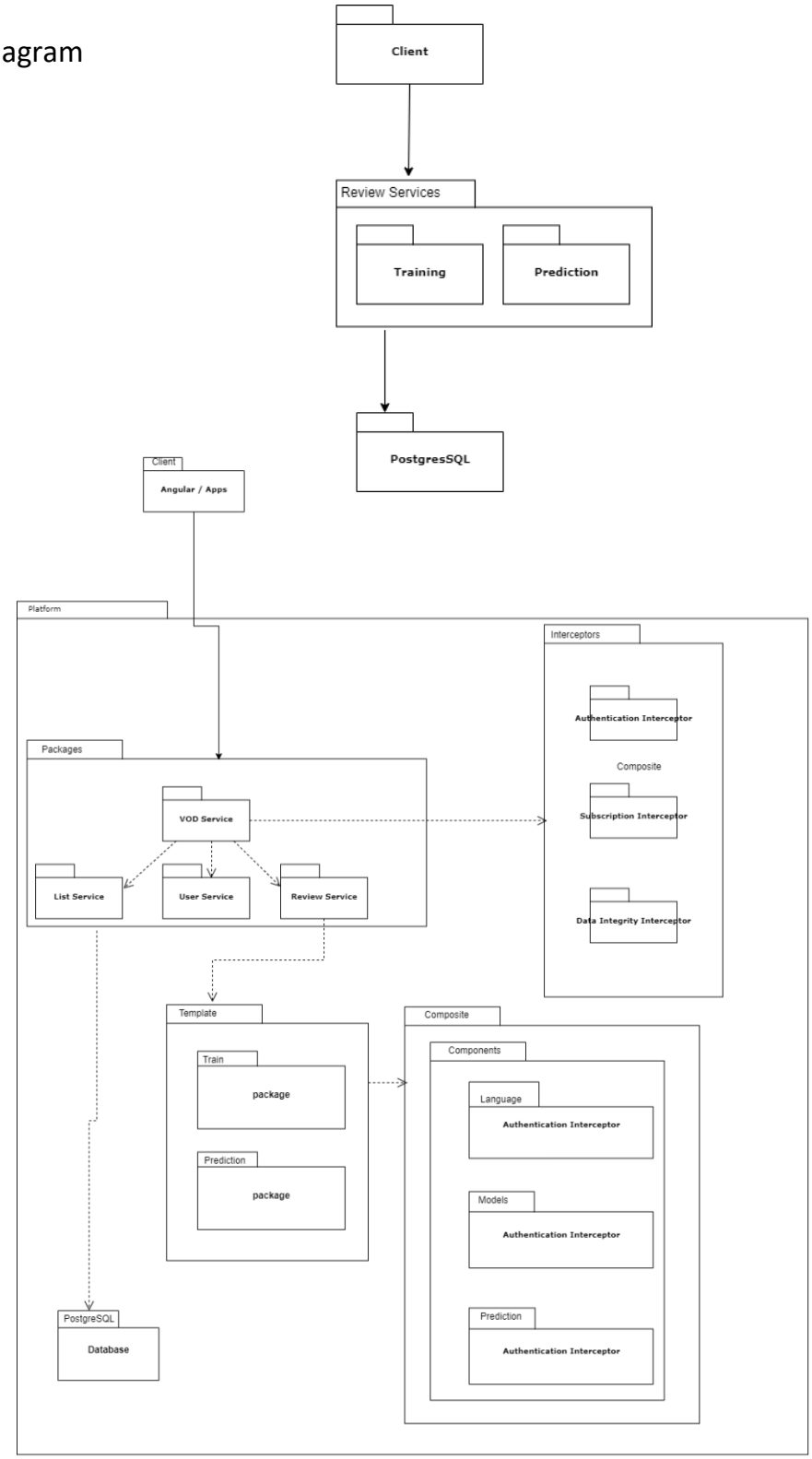
As we can see above before_action is defined for each API request this controller gets. In the before_action service, the class will be initialized, and then based on the request and params it will get the service that it has to redirect and transfer the call to the service location.

```ruby
class MicroService
attr_accessor :access, :data, :resp, :method, :path, :controller, :body_key, :url, :subdomain

def initialize request, params
  @access     = request
  @subdomain  = self.access.subdomain + "." + self.access.domain
  @data       = params.as_json
  @resp       = nil
  @method     = request.method
  @path       = request.path.gsub("/admin", "")
  @controller = self.data["controller"].gsub("admin/", "")
  @body_key   = self.controller.singularize
  @url        = self.access.path.gsub("/admin", "")
  @privilege  = self.method + "_" + self.data["action"]
end

def call
end

def get_response
end

def error_response
  return [ {message: "Request Failed, Please contact Admin"}, :unprocessable_entity]
end

def get path=nil
  if path.nil?
    url = construct_url + "/" + (self.data["id"] || "")
    return HTTParty.get(url, :headers => construct_header).to_json
  else
    url = self.host + "/" + path
    Rails.logger.warn "----#{url}"
    return HTTParty.get(url).to_json
  end
end

def post path=nil
  Rails.logger.warn "****POST call****"
  if path.nil?
    return HTTParty.post(construct_url, :headers => construct_header, :body => construct_body.to_json)
  else
    return HTTParty.post(self.host + "/" + path, :headers => construct_header, :body => construct_body.to_json)
  end
end
```

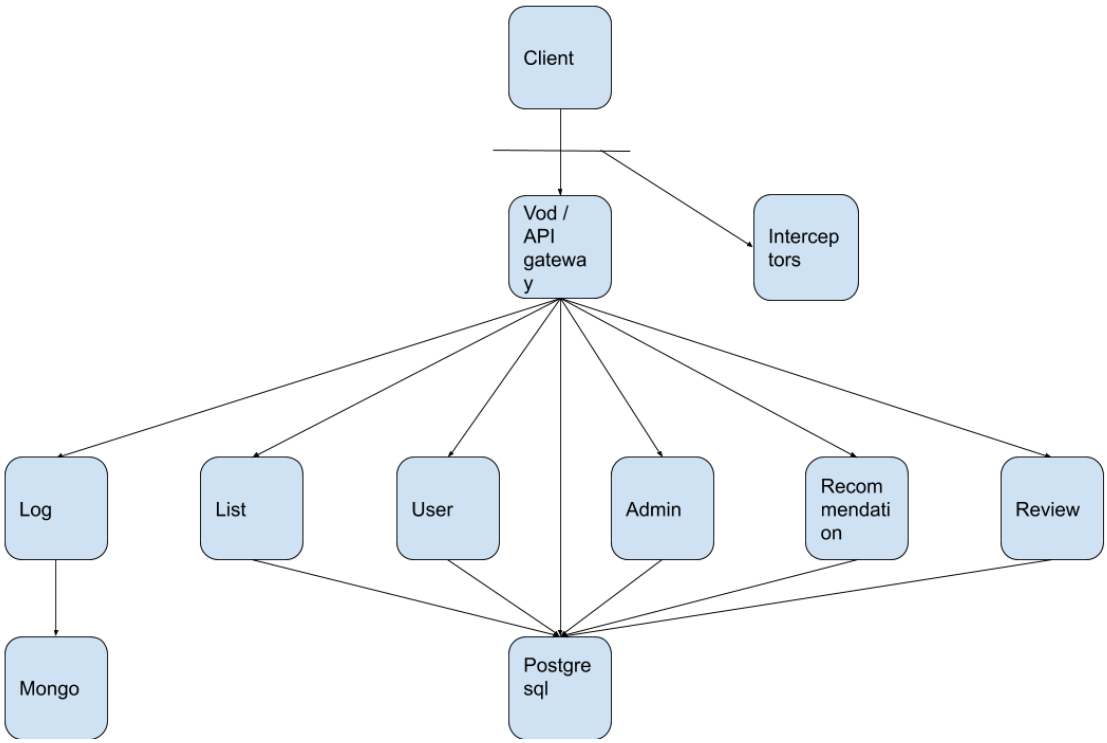
This class basically gets and sets all the information required to forward the API request to the required service. With this approach, it is easy for us to add any new service as most of the work is common and defined in the above class.

V. System Architecture

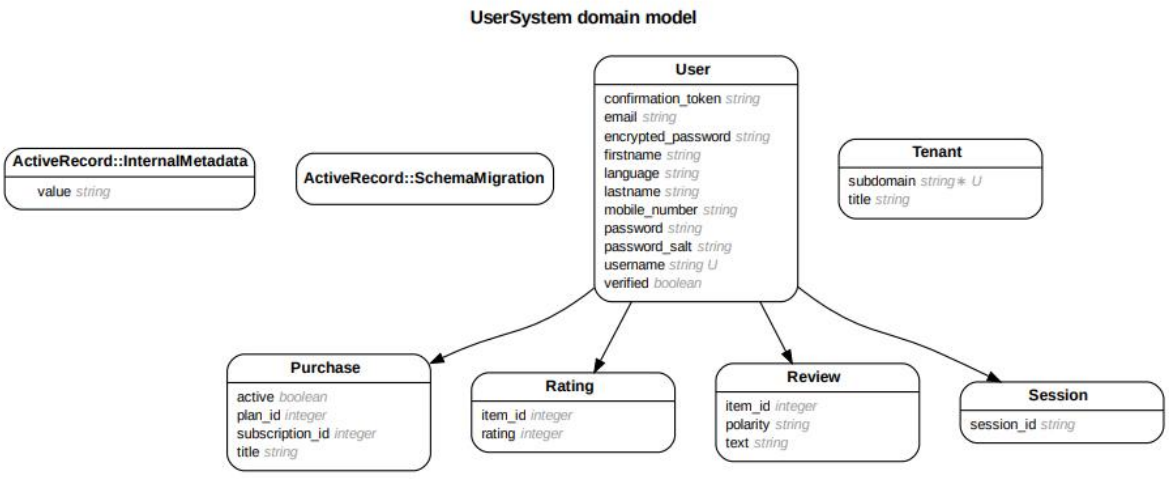
Package Diagram



System Architecture

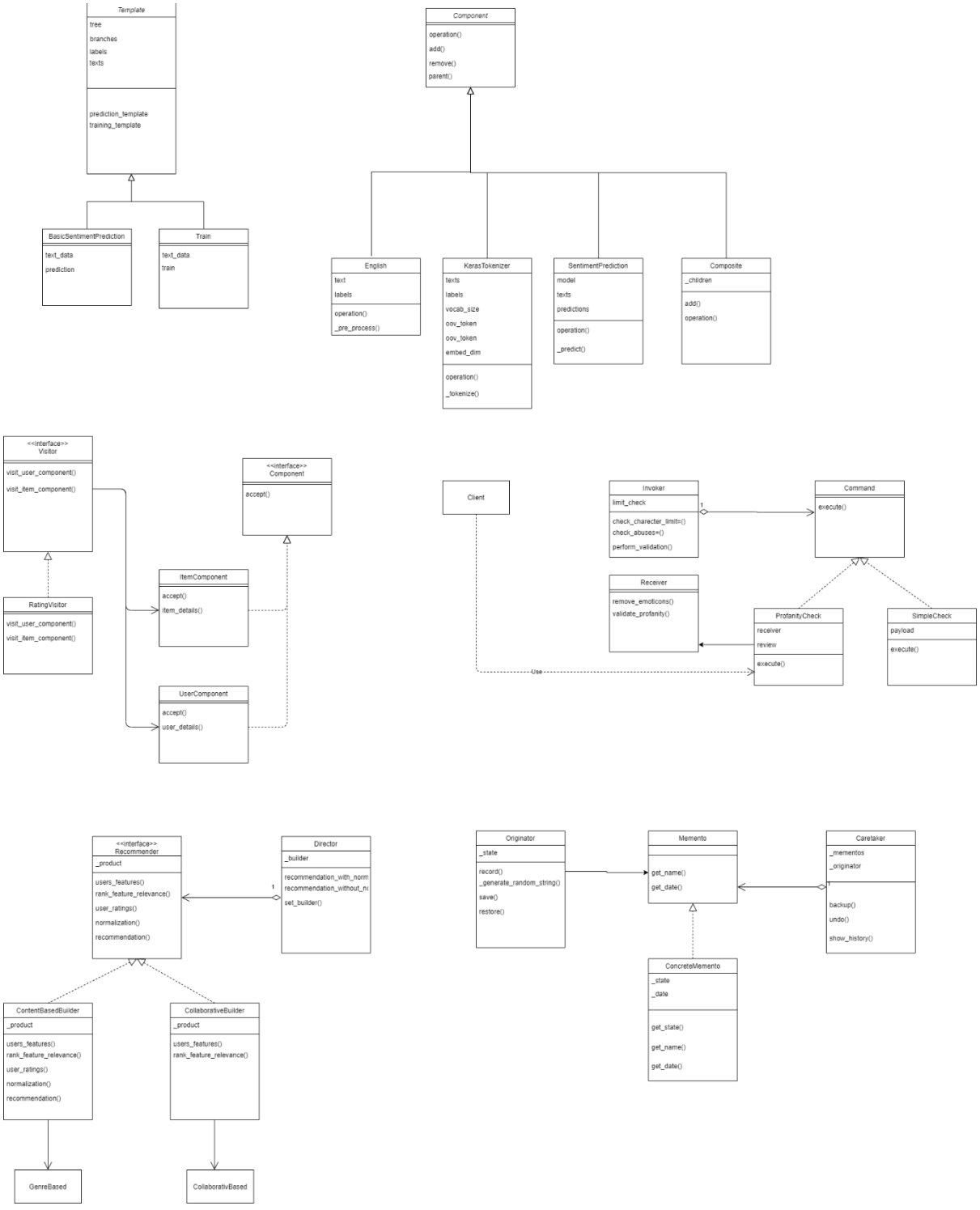


Entity Relational Diagram (ERD)

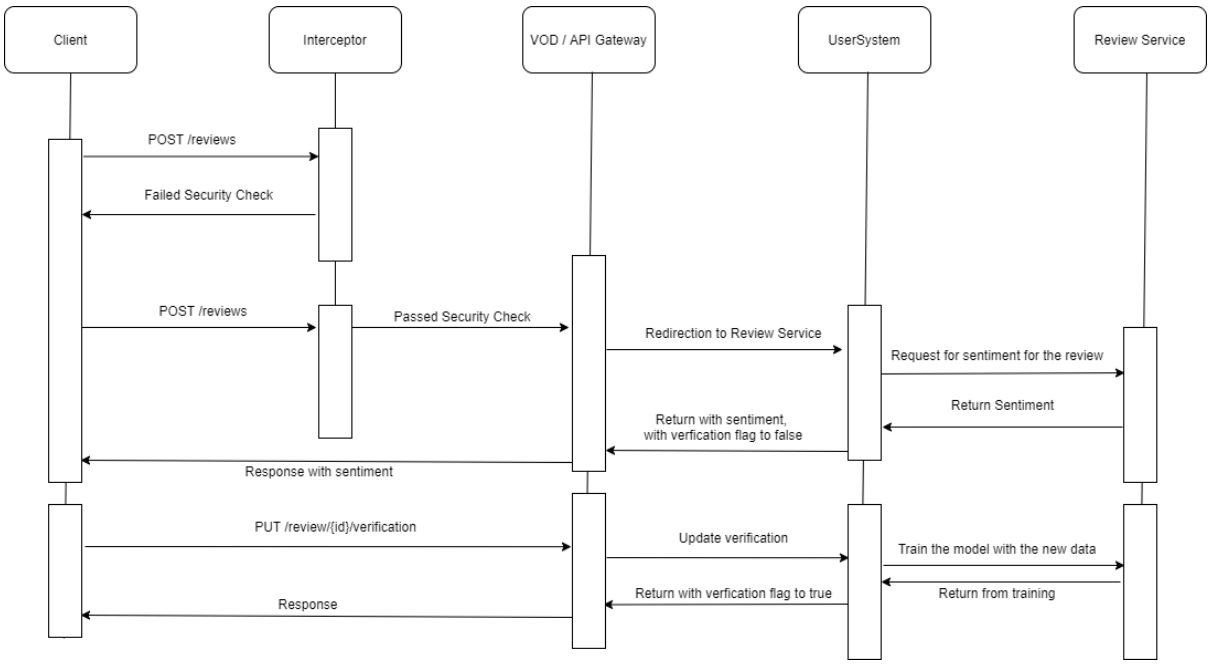


VI. Structural and Behavioural Diagram

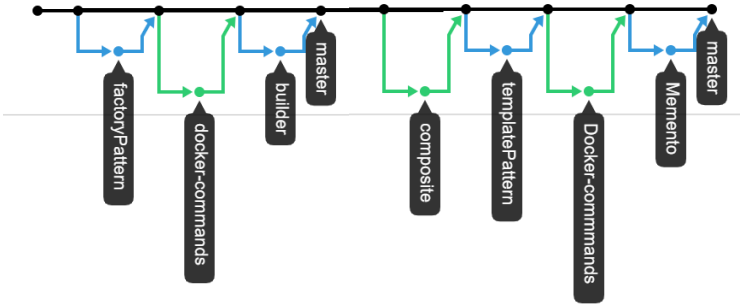
Class Diagram



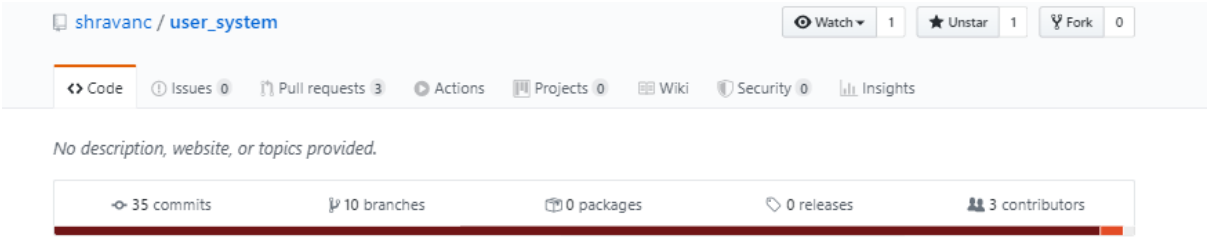
Sequence Diagram



VII. GitHub



GitHub was used as the version control tool for the project. An example of using branching in the project is shown in the above image.



An example of one of the repositories which is a microservice in our project (user_system)

VIII. Added Value

SonarQube

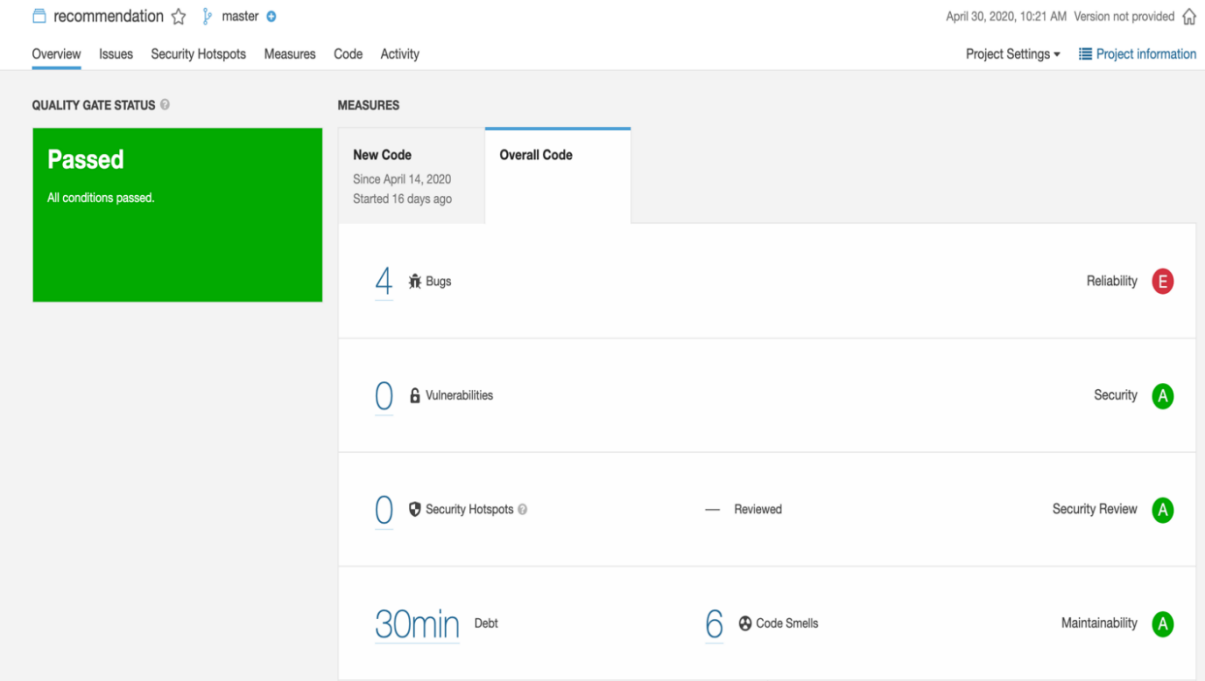
SonarQube gathers and perform analysis on the source code to provide measuring quality reports for the projects. It incorporates static and dynamic analytical tools and enables continuous measurement of quality over the time. SonarQube evaluates everything that affects the code base, from minor styling details to critical errors. It is an open source platform which can access and evaluate the quality of the source code projects on more than 20 programming languages. It evaluates and give reports on code duplication, coding standards, code smells etc.

Once SonarQube is downloaded you can run the script to start the server. The server is now accessible at http://localhost:9000. You can login to the console with admin rights with user name and password being admin:admin. With SonarQube installed and configured, it is ready to analyze the source code. You need to write a sonar-project.properties file inside the folder where you want the SonarQube to perform the analysis [3, 4].

To perform the analysis, we need Sonar-Scanner which helps the SonarQube script in the analysis. Once the scanner has started the output look like:

```
INFO: Load project repositories
INFO: Load project repositories (done) | time=43ms
INFO: Starting rules execution
INFO: 28/28 source files have been analyzed
INFO: 28 source files to be analyzed
INFO: Sensor Python Sensor [python] (done) | time=1125ms
INFO: 28/28 source files have been analyzed
INFO: Sensor Cobertura Sensor for Python coverage [python]
INFO: Sensor Cobertura Sensor for Python coverage [python] (done) | time=13ms
INFO: Sensor PythonXUnitSensor [python]
INFO: Sensor PythonXUnitSensor [python] (done) | time=4ms
INFO: Sensor SonarCSS Rules [cssfamily]
INFO: 1 source files to be analyzed
INFO: 1/1 source files have been analyzed
INFO: Sensor SonarCSS Rules [cssfamily] (done) | time=2246ms
INFO: Sensor JaCoCo XML Report Importer [jacoco]
INFO: Sensor JaCoCo XML Report Importer [jacoco] (done) | time=3ms
INFO: Sensor JavaXmlSensor [java]
INFO: Sensor JavaXmlSensor [java] (done) | time=3ms
INFO: Sensor HTML [web]
INFO: Sensor HTML [web] (done) | time=38ms
INFO: ------------- Run sensors on project
INFO: Sensor Zero Coverage Sensor
INFO: Sensor Zero Coverage Sensor (done) | time=21ms
INFO: SCM Publisher SCM provider for this project is: git
INFO: SCM Publisher 6 source files to be analyzed
INFO: SCM Publisher 6/6 source files have been analyzed (done) | time=125ms
INFO: CPD Executor 13 files had no CPD blocks
INFO: CPD Executor Calculating CPD for 16 files
INFO: CPD Executor CPD calculation finished (done) | time=10ms
INFO: Analysis report generated in 66ms, dir size=122 KB
INFO: Analysis report compressed in 109ms, zip size=60 KB
INFO: Analysis report uploaded in 63ms
INFO: ANALYSIS SUCCESSFUL, you can browse http://localhost:9000/dashboard?id=recommendation
INFO: Note that you will be able to access the updated dashboard once the server has processed the submitted ana
lysis report
INFO: More about the report processing at http://localhost:9000/api/ce/task?id=AXHKZCyyza7ulhXmBSf0
INFO: Analysis total time: 9.159 s
INFO: ------------------------------------------------------------------------
INFO: EXECUTION SUCCESS
```

After the scanner has executed, the results can be seen on the http://localhost:9000



Docker

To help with improving efficiency and speed of development and deployment, we decided as a team to use Docker for our project. Docker is an open-source project which is used in the software industry to automate the process of deployment of software applications inside containers. This is done by providing an additional layer of abstraction and automation of OS level virtualisation on Linux.

Docker is an essential tool that allows for developers and engineers to easily deploy applications in environments called containers. These containers package applications with all of its dependencies into a standardised unit. Docker containers are unlike virtual machines where there is high overhead. Containers provide much more efficient usage of underlying resources. Docker uses the concept of images and containers to run applications. Docker images are the blueprints of the applications. Docker images contain all the necessary dependencies and OS software requirements. Once the image is created, containers can be created from these images [5,6].

A Dockerfile is necessary to create a Docker image. This file contains the list of commands required to create an image. Once this image is created, containers can then be deployed and run.

In our project Docker was an extremely useful tool to help with the running of our application and deployment of the application. Our project consisted of microservices, so having the ability to run a microservice on each container made running the project very simple by running the docker containers. The below Docker files show the efficiency of Docker.

```
FROM python:3

COPY . .
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
# 8888:5000
CMD ["python", "./main.py"]
```

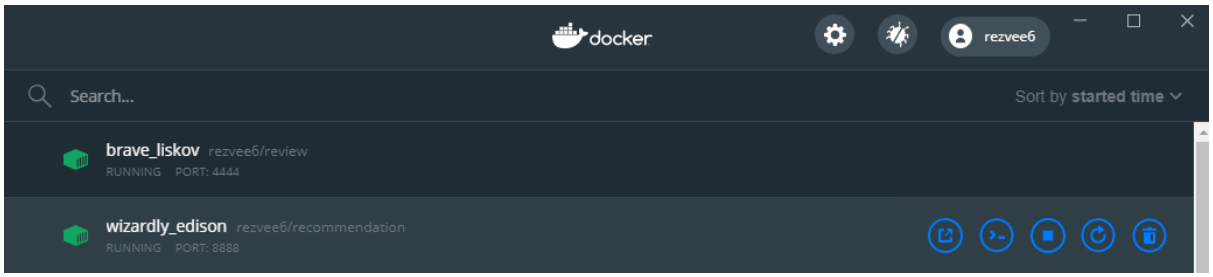
DockerFile for recommendation_service

```
FROM rezvee6/recommendation

COPY . .
RUN pip install --no-cache-dir -r requirements.txt
EXPOSE 5000
# use 4444:5000 in build process
CMD ["python", "./main.py"]

#docker run -p 4444:5000 rezvee6/review
```

Dockerfile for Review_service

From the Dockerfiles above, you can see that a new python3 environment is created in the image for recommendation service. For the second Dockerfile, we can reuse the first environment rather than creating a new one. This example shows us running two microservices, recommendation_service & review_service.

Once the two images are complete, the containers can be run using the docker desktop application using 1 click.



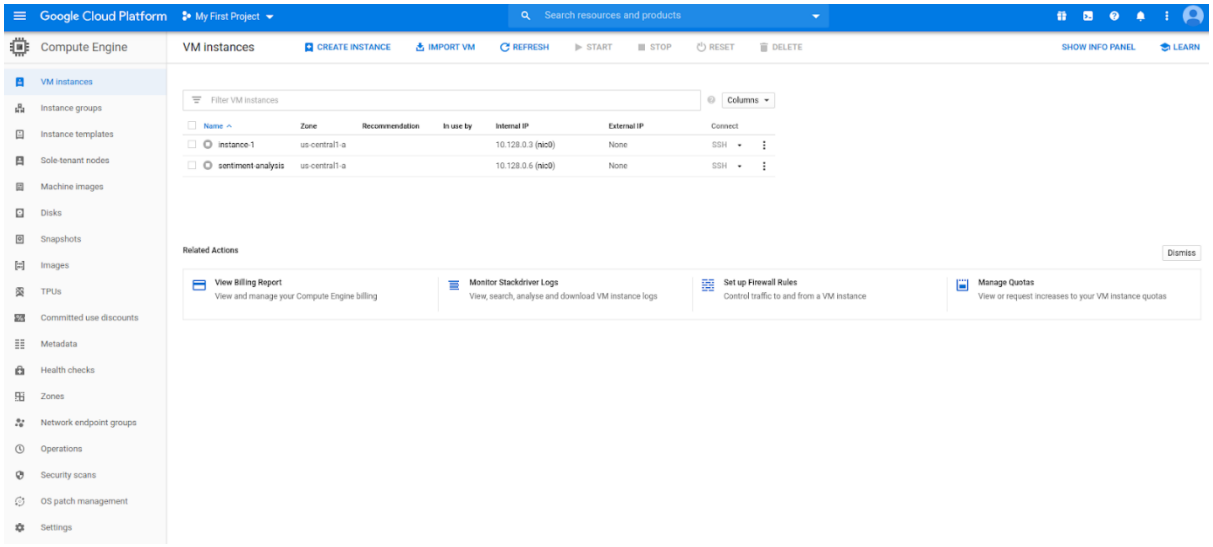
Both containers can be interacted with from the command line.

Cloud Deployment

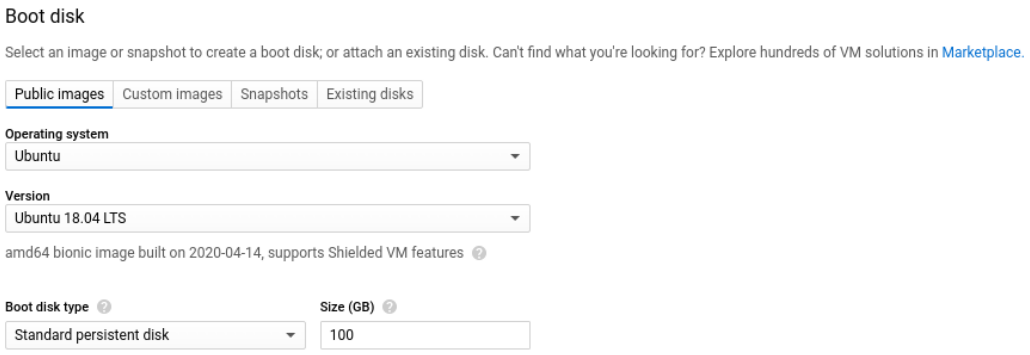
We have used the Google Cloud Platform (GCP) for hosting the platform on the cloud. As stated above we have containerized all the services that are used in our project. This made us very easy to deploy our service into the cloud [7]. We might have to barely install the only docker into the cloud servers and rest docker does the job of installing and setting up and running of the application/service. We will show in details about how it is done in details:

Compute Engine:

GCP provides a compute engine service to launch VM machines for any needs. Here we are showing at the beginning we are not having a VM machine for our project:



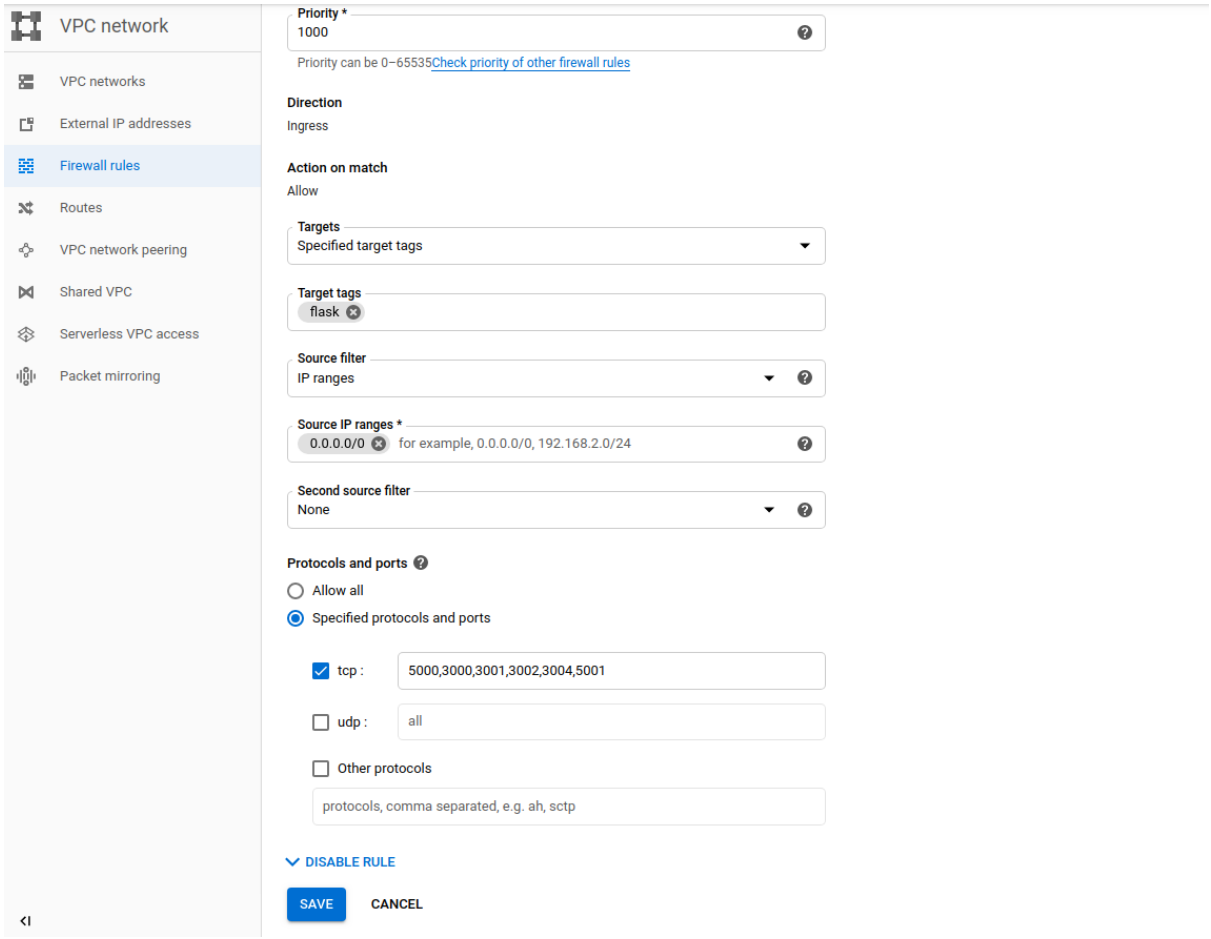
We then created a CM machine for our need, we decided to go with 2CPU and 8 GB ram. Memory might go low for future use; it is to test the deployment. Here are the steps to create:



We chose Ubuntu 18.05 OS as we have developed this application on this OS and this is the latest version available as of now. We chose 100 GB which is more than sufficient.

Network Setting:

We needed multiple port numbers to be opened for multiple services. Hence, we had to open port number for the instance we created. It is done as below:

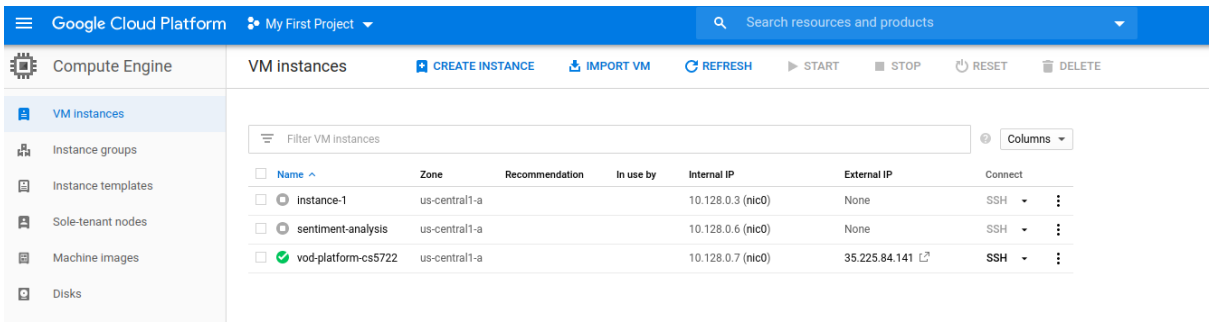


Under Protocols and ports section it is evident that TCP is set to multiple port numbers for each service 3000(vod_platform), 3001(admin_system), 3002(user_system), 3003(list_system), 5000(recommendation_service) and 5001(review_service).

Target tags are the tags that can be specified in the network setting of the compute engine. With this tag, all port will get available to the instance [8].

SSH:

With the instance running as below:



We then configured ssh for our local system by generating RSA password and updating it in the instance. Logging is done as below:

```
shravan@shravan:~$ ssh -i ~/.ssh/cs5722_rsa abraca_data@35.225.84.141
Welcome to Ubuntu 18.04.4 LTS (GNU/Linux 5.0.0-1034-gcp x86_64)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

  System information as of Thu Apr 30 17:50:43 UTC 2020

  System load:  0.0               Users logged in:                0
  Usage of /:   3.4% of 96.75GB   IP address for ens4:            10.128.0.7
  Memory usage: 8%                IP address for docker0:         172.17.0.1
  Swap usage:   0%                IP address for br-4f4b6383c88f: 172.18.0.1
  Processes:    117


28 packages can be updated.
15 updates are security updates.


Last login: Thu Apr 30 17:50:02 2020 from 95.44.101.177
abraca_data@vod-platform-cs5722:~$
```

Once docker is installed in the server will be run with docker commands as below:

```
abraca_data@vod-platform-cs5722:~/user_systen/user_system$ sudo docker-compose up -d
Starting usersystem_db_1 ...
Starting usersystem_db_1 ... done
Starting usersystem_app_1 ...
Starting usersystem_app_1 ... done
abraca_data@vod-platform-cs5722:~/user_systen/user_system$ ls
Dockerfile  Gemfile  Gemfile.lock  README.md  Rakefile  app  bin  config  config.ru  db  docker-compose.yml  entrypoints  erd.pdf  lib  log  public  storage  test  tmp  vendor
```

The same process is repeated for other services with the docker commands.

IX. Testing

We have followed BDD with a rails application we have integrated various minimal test cases on mainly with the return value to be 200 and then checking further on the content type of the response. And mainly the JSON response. The JSON response is the most important thing in API development. Whenever by any means a JSON is modified it will hamper the frontend functionality. So, in our project, we have taking this as an important factor in the development.

```ruby
class UsersControllerTest < ActionDispatch::IntegrationTest

  test "Should give 200 response" do
    get '/users'
    assert_response :success
  end

  test "content type validation" do
    get '/users'
    assert_equal response.content_type, 'application/json'
  end

  test "json validation" do
    jdata = json.parase response.body
    assert_equal 0, jdata['users'].length
  end


end
```

Likewise, we have integrated the test case to the controllers to validate the most import part of the API request. This will enhance to gain the confidence that we do not break the front-end functionality.

X. Evaluation & Critique

To make sure that the code we have developed for this project is efficient and properly optimized we have chosen the design patterns carefully. Some of the previous code was also refactored to make them efficient and also some of the previous design patterns were tweaked a little to make them batter. In our opinion, the design and architectural patterns used in this project support their relevant use cases. The Interceptor which plays an important role in our framework validates any request which comes from the client and if the validation is met it redirects the requests to the proper services. The composite and template patterns are used extensively to handle the sentimental analysis in our framework. Although we have implemented a few design patterns in this project, due to time constraints we did not get the most out of some of the patterns. An example of this is the memento design pattern. In our implementation we did not have time to include the undo operation which would be quite useful in real world scenario where there may be some errors in the training phase of the sentiment analysis phase. Using the memento design pattern has allowed for useful functionality by allowing us to make snapshots of the objects state and storing these states for further use.

In the project, we have introduced two main features which we think will add more value to the project. These two new services are related to machine learning integration to the platform. One service is on building a recommendation engine that will give

recommendations for the user with a certain history of rating the movies they watch. This service provides a reasonable recommendation but it will not encounter the new user's behaviour. We had thought of build two recommendation engines, one to tackle early users and another to tackle existing users. With the design pattern, it is very easy to implement but at the limited amount of time, we were able to build only content-based recommendation which will consider only genres and the rating that user has done on the movies that they have watched. Also, another fact to consider is the delivery of the recommendation, we have not implemented the caching to the system and every time the API request is called, a fresh set of calculations will be done and recommendations will be served. And in this process entire user rating data is required and, on the flow, the calculation is done and will provide the result. This might cause a huge load on the system and might hang on certain requests the server receives. The more and more users get in the system API might take longer time and there is no security integrated into the system to actually cut off the request if it takes too long.

Another important service that we have integrated is sentiment analysis. Here our idea is to give the both the rating user has rated and also an idea of what user thinks about the movie using the review that the user has given into the system. We think that this is a very good feature as also it gives real-time feedback on what the sentiment behind the review is about. The flow goes like this: User enters the review and then submits it to the system, then system returns the sentiment what they are trying to infer and the user has to then return back the feedback if the sentiment is correct. This is an optional step for the user. If the user provides the feedback the system trains back the saved model in the system and saves it with the new data. If the user decided not to send the feedback, then this process has to be done from the CMS. In the database, a state of feedback has to be maintained as to see if the feedback for a review is given or not. In the CMS only the review without feedback is shown and then the model is trained further on the data. In this systematic review could more and might increase. This service required us to maintain a profanity check and also subscription check. All this will add the load up on the server and there is not precaution written in the system to handle the load. As the model grows bigger and bigger, loading and storing become the bigger problem. There is no plan has to how to handle the load and more requests in such cases would hang the system.

Adding an interceptor design pattern to the system allowed us to add any type of security validation to the system with ease and not disturbing the existing application. The maximum effort that could go was in creating an interceptor class and provide it to the dispatcher. This was a great learning for the team to understand how the interceptor works and can be an easy to support further support to the system.

XI. References

[1] Design Patterns, 2019, refactoring.guru, Viewed 10 November, 2019

< https://refactoring.guru/design-patterns>

[2] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael stal, Pattern-Oriented Software Architecture, Wiley, Volume 1.

[3] C-sharpcorner, 2019, Step by step SonarQube Setup and Run

< https://www.c-sharpcorner.com/article/step-by-step-sonarqube-setup-and-run-sonarqube-scanner/>

[4] Medium, SonarQube Setup in 10mins on Ubuntu, Harsh Verma, 2019

< https://medium.com/@harshverma04111989/sonarqube-setup-in-10-mins-on-ubuntu-5d73578f8891>

[5] Docker Curriculum, Introduction to Docker, 2019,

< https://docker-curriculum.com/#introduction>

[6] DigitalOcean, Containerizing ruby on rails, 2019,

< https://www.digitalocean.com/community/tutorials/containerizing-a-ruby-on-rails-application-for-development-with-docker-compose>

[7] GCloud, Google Cloud Docs, 2020,

< https://cloud.google.com/compute/docs/machine-types>

[8] GCloud, Google Cloud Docs, 2020,

< https://cloud.google.com/vpc/docs/firewalls>

XII. Contribution

Report

Name	ID	Contribution
Assad Nawaz	19085842	Requirements, Design Patterns, SonarQube, Eval & Critique, Testing
Reezvee Sikder	15140997	Use Cases, Design Patterns, Docker, Eval & Critique, Github
Shravan Chandrashekharaiah	18043038	Architecture, Design Patterns, Cloud deployment, Eval & Critique, Github, Testing

Table with LoC etc.

Microservice	Package	Contributors	Loc
reviewService		Reezvee/Assad	774
	Template		103
	Memento		107
	Template		70
	schema		230
	Docker		8
	Controllers		45
	models		211
reccomendationService		Reezvee/Assad	556
	Controllers		16
	Builder		210
	Factory		42
	Decorator		84
	model		204
User system		Shravan	240

	Controllers		40
	Models		70
	Command		60
	Visitor		60
	Docker		10
List system		Shravan	60
	controllers		30
	models		20
	Docker		10
Vod platform		Shravan	531
	controllers		35
	models		10
	interceptor		136
	Service locator		340
	Docker		10