

UNIVERSITY *of* LIMERICK

O L L S C O I L L U I M N I G H

Multi-Tenant Video on Demand Platform

Software Design Project
CS5721
Lecturer: JJ Collins

Submitted by:

Assad Nawaz, 19085842

Reezvee Sikder, 15140997

Shravan Chandrashekharaiyah, 18043038

1. Table of Contents

1. Table of Contents.....	2
2. Project Description.....	4
2.1 Overview.....	4
2.2 Business Requirements.....	4
2.3 CASE Tools.....	4
3. Software Development Process.....	5
3.1 Overview.....	5
3.2 Software development and Software Lifecycle Methodology.....	5
3.3 Kanban Process.....	6
4. Project Plan.....	7
4.1 Role Assignment.....	8
4.2 Project Plan.....	8
5. Requirements.....	9
5.1 Functional Requirements.....	9
5.2 Non-Functional Requirements.....	9
5.3 Use Case Scenarios.....	11
5.3.1 Use Case Diagrams.....	11
5.3.2 Use Case Descriptions.....	15
6. User Interface Design.....	23
7. System Architecture.....	25
7.1 Architecture/Package Diagram.....	25
7.2 Technology Pipeline.....	26
8. System Architecture Patterns.....	28
8.1 MVC.....	28
8.2 Additional Architectural Pattern: Microservices Architectural Pattern....	28
8.3 Multi-Tenant Architecture.....	29
9. Design Patterns.....	30
9.3 Factory Pattern.....	30
9.2 Decorator Design Pattern.....	30
9.3 Singleton.....	31
9.4 Chain of Responsibility.....	31
9.5 State Pattern.....	31
9.6 Facade.....	31
9.7 Observer Pattern.....	32
9.8 Active Record Pattern.....	32
10. Object Orientated Analysis.....	33
10.1 Analysis Class Diagram.....	33
10.2 ERD Diagram.....	34
10.3 Architecture/Package Diagram.....	35
10.4 Sequence Diagram.....	36
10.5 Communication Diagram.....	37

10.6 Activity Diagram.....	37
11. Recovered Design & Architectural Blueprints.....	38
11.1 Recovered Architectural Diagram.....	38
11.2 Recovered Class Diagram.....	39
11.3 State Diagram.....	40
11.4 Component Diagram.....	40
11.5 Deployment Diagram.....	41
12. Test Cases.....	41
12.1 Example Test 1.....	42
13. Added Value.....	43
13.1 REST Architectural Pattern.....	43
13.2 Object Relational Mappers (ORM).....	43
13.3 Use of Microservices.....	44
13.4 Framework.....	44
13.5 GitHub Actions.....	44
13.6 Multi-Tenant Architecture.....	44
14. Critiques.....	45
15. Appendices.....	47
15.1 References.....	52

2. Project Description

2.1 Overview

KISS Solutions Plc. have been commissioned by a client to develop Video on Demand Platform. The platform will allow users to consume videos of many genres and types with the ability to like playlist and share videos.

2.2 Business Requirements

Video on Demand platform is a multi-tenant product, 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.

In this platform, each company/business group/company will first create an account in the CMS content management system. Once the account is obtained, they will be able to create multiple tenants for their system. Multi-tenancy gives segregation of content to the various tenant-specific users at the same time having all the content under the same business group. Once the tenant is created, admin is allowed to create applications for the service. Here application could be Android/IOS/Windows/Android TV/Apple TV/Roku etc. This gives further segregation and control over each device. Each application will be given an authentication token. Using which each application will communicate with the servers. The server will understand which application has made the call and serve accordingly. It will check if the content is available to the particular application making the request.

At the user/consumer end, registration/login and various other facilities will be provided. Here user once logged in will be able to access the content uploaded by the provider. Based on the user history of watching content, the recommendation will be given on the videos that a particular user could watch, this is done based on the genres of the content that is watched by the user. Users will also be able to share the video and like and create a playlist of their choice.

2.3 CASE Tools

A variety of software development tools were utilized during the development of the project. The tools are named below:

Visual Paradigm / Draw.io

Visual Paradigm 10.0 was utilised during the early stages of the project to create and edit the UML diagrams which were designed for the project. Visual Paradigm contains all the necessary functionalities to aid with the development of the project. During the latter stages of the project, Draw.io was used to create the remaining diagrams.

Visual Studio Code/ Atom

The software engineering team had the freedom of choosing their own development environments for developing the code. Visual Studio Code was one of the choices as it had built in GitHub support.

GitHub/Version Control

As part of the requirements, the development of the project was done by using GitHub as the version control system. The Project was hosted online on GitHub and the team worked on individual branches and a master branch for the project.

3. Software Development Process

3.1 Overview

It was decided during the initial stages of the project it was decided that the group would try to incorporate Agile methodology during the course of our project. This Agile methodology was modified to suit our needs. General principles such as changing requirements as the project progresses. Sprints were much shorter than typical sprints in an agile workflow.

The primary measurement of our work progress would be to tick off the required aspects of the project.

3.2 Software development and Software Lifecycle Methodology

For this project multiple GitHub repositories were created for the different microservices that the application would contain. The repos consisted of a master branch and couple of leave branches according to the different features. The master branch was maintained properly and considered to be stable and ready to deploy branch. The developer will work on a dev branch while developing a new feature and then push the changes to the master branch on the GitHub.

For the development of this project we have used Agile/ Kanban software development lifecycle. A Kanban board was created where tasks were mentioned. Each developer will first take a deep understanding of the tasks and then pick it up for implementation.

3.3 Kanban Process

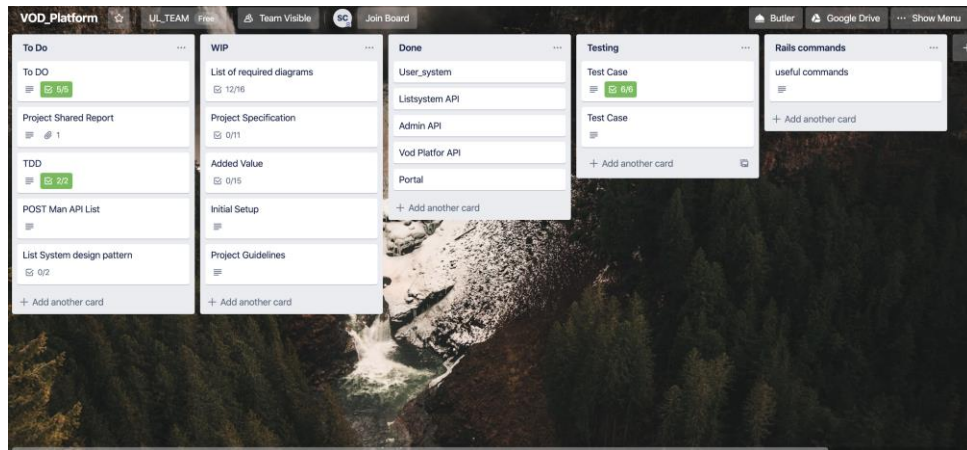


Figure 1. Trello Board

The tasks or features were finalised by the project manager with continuous communication and feedback from the developers. Once a task was decided upon, it was kept in a story bucket for developers to pick up. Each developer will first understand the task/requirement properly and then pick up the tasks from the bucket and move it to work-in-progress and once completed, will move it for testing and from there completed/ready to move to master branch.

4. Project Plan

4.1 Role Assignment

Assad Nawaz, 19085842 - AN

Reezvee Sikder, 15140997 - RS

Shravan Chandrashekharaiiah, 18043038 - SC

	Role	Description	Designated Team Member
1	Project Manager	<ul style="list-style-type: none">- Setting up group meeting.- Planning the project- Tracking the delivery of the Project	SC
2	Technical Lead	<ul style="list-style-type: none">- Guide and Leads the technical development phase.- Tracking the technical debts- Tracking the quality of technical deliverables	AN, RS, SC
3	Business analyst	<ul style="list-style-type: none">- Responsible for requirement gathering- Supporting the project manager in keeping track of product backlog	RS
5	Architect	<ul style="list-style-type: none">- Making high level design and technical standard choices- Defining System Architecture	AN, RS, SC
6	Documentation Manager	<ul style="list-style-type: none">- Aggregating all the supporting documents collected from developers in to report	RS
7	Designer	<ul style="list-style-type: none">- Responsible for recovering design time blueprints from implementation	AN
8	Programmers	<ul style="list-style-type: none">- to develop packages (at least 1) described in the system architecture	AN, RS, SC
9	Tester	<ul style="list-style-type: none">- Testing each component	AN, RS, SC

4.2 Project Plan

	Week	Description
Project Outline	3	<ul style="list-style-type: none"> - Team meet/setup - Decide on project outlines - Decide on meetings going forward.
Requirement Elicitation	4	<ul style="list-style-type: none"> - Discuss on business logic. - Discuss functional and non-functional requirement
Requirements	5	<ul style="list-style-type: none"> - Create use case diagrams - Implement use case diagrams - Create the rest of the use cases
System Architecture	6-7	<ul style="list-style-type: none"> - Decide on technology pipeline (which language and framework to use) - Design architecture of system - Create architectural diagrams
Architectural Patterns	8	<ul style="list-style-type: none"> - Decide and discuss which architectural pattern will be useful for the project
Analysis Phase	9	<ul style="list-style-type: none"> - Create class diagrams using use case diagrams - Create state chart diagrams - Create communication diagram - Design entity relationship diagram - Create sequence diagram - Create activity diagram
Design Phase		<ul style="list-style-type: none"> - Factory - Decorator - Observer - Facade - Singleton - Chain of Responsibility - Active Record
Component & Deployment Diagrams	10	<ul style="list-style-type: none"> - Create component diagram - Create deployment diagram
Iterative Software Development Phase	8-12	Each member will start implementing the task chosen by them according to the system requirement

Critiques of the Project	13	Discuss project critiques with team members
Project Completion	13	Interview

5. Requirements

5.1 Functional Requirements

User

- Each user will be identifiable in the backend using their own user ID on the platform
- The user will have personal information such as username, name and email address.
- User information will be represented as string of characters etc.

CMS/Content Management System

- Another admin like user on the platform that can:
- 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
- Create Listing in the portal
- Ordering or updating the List
- Adding items to list
- Updating the items or the ordering inside the list

5.2 Non-Functional Requirements

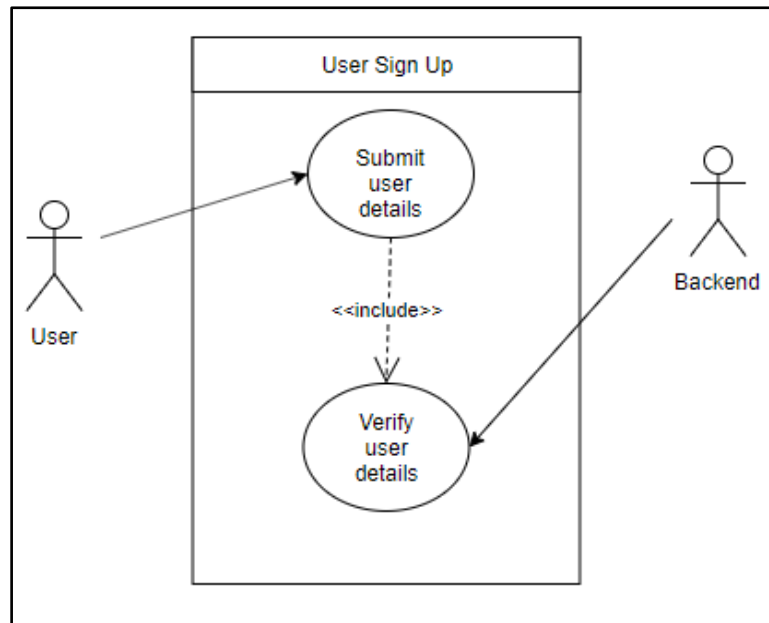
- The user interface of the application should be responsive.
- The user should be able to search the content easily and results of the search should be relevant.

- Depending upon user's previous searches, the search option should give back results in that order.
- The default quality of the videos should be high and the quality of the videos should automatically change depending upon the speed of the user's internet connection.
- The information provided with the content/video should be relevant and easy to understand.
- The error messages displayed to the user should be easy to understand.
- The user should verify his account after registering before he/she can login to access the content
- The system should be able to distinguish between user and admin and should allow only the admin to make the changes.
- Once the user deactivates his/her account the system should delete all the information from the database.
- The system should be scalable, the system should be flexible enough so that new features can be added easily to the existing without compromising the security.
- The system currently communicates with the user via mail, but in the future feature to communicate through text messages should be implemented.

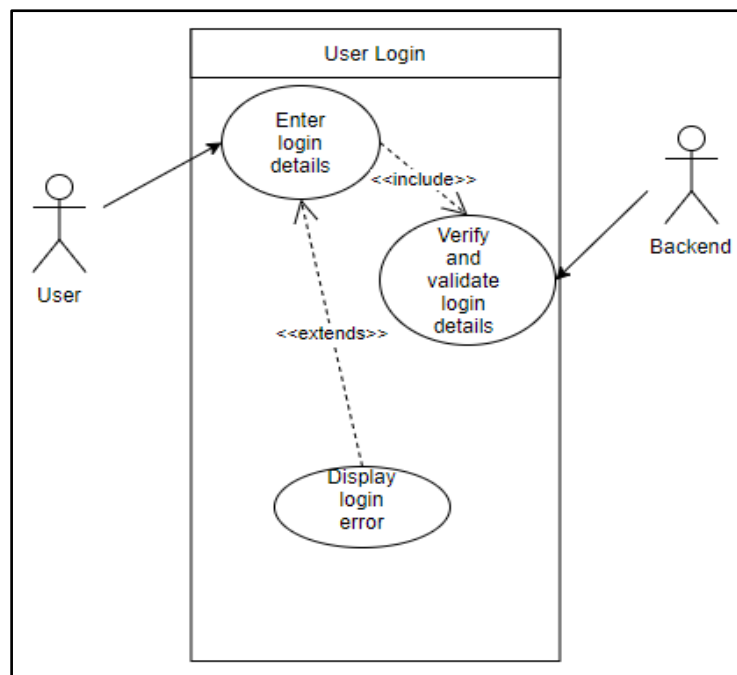
5.3 Use Case Scenarios

5.3.1 Use Case Diagrams

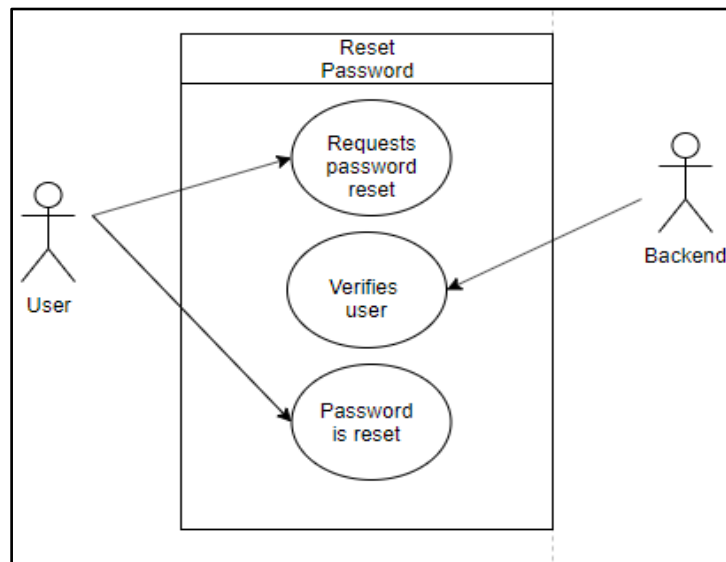
Use Case 1



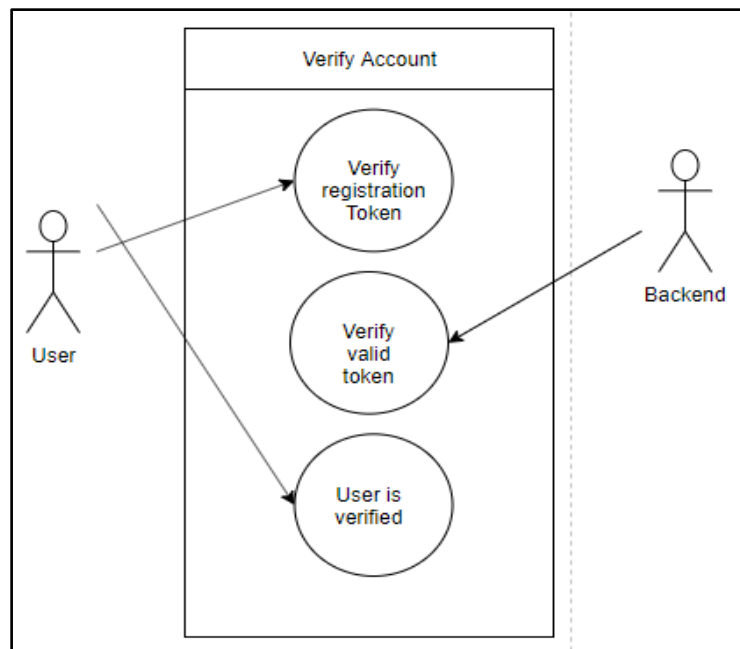
Use Case 2.



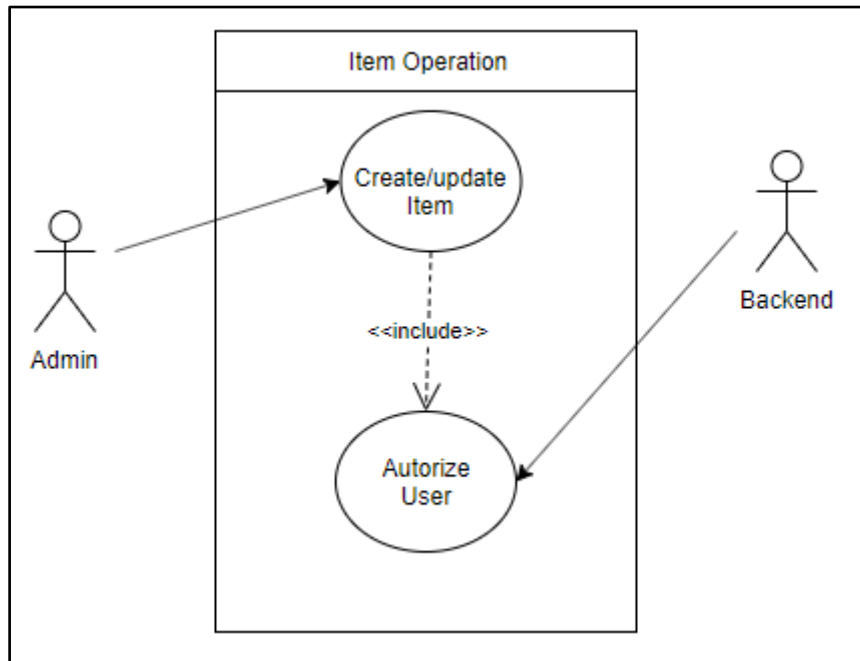
Use Case 3.



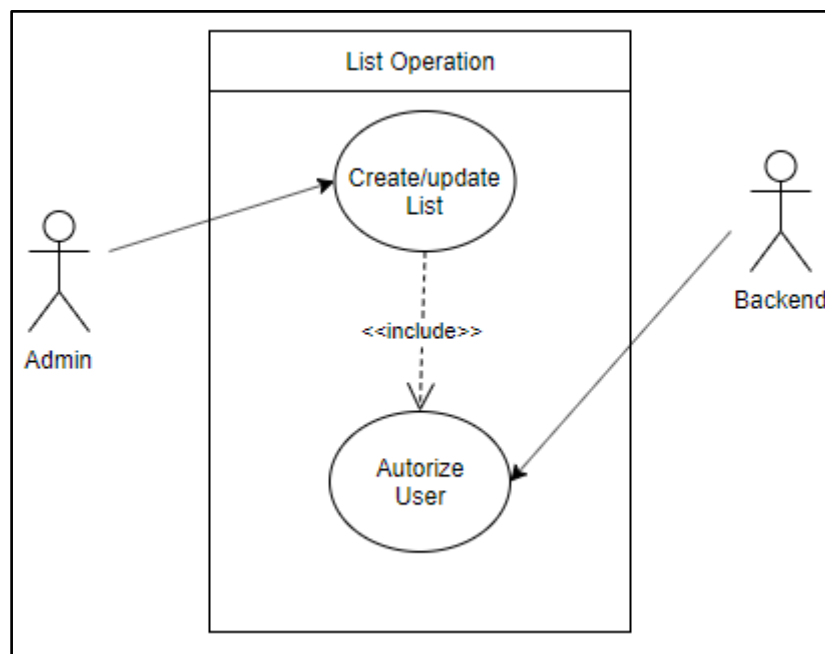
Use Case 4.



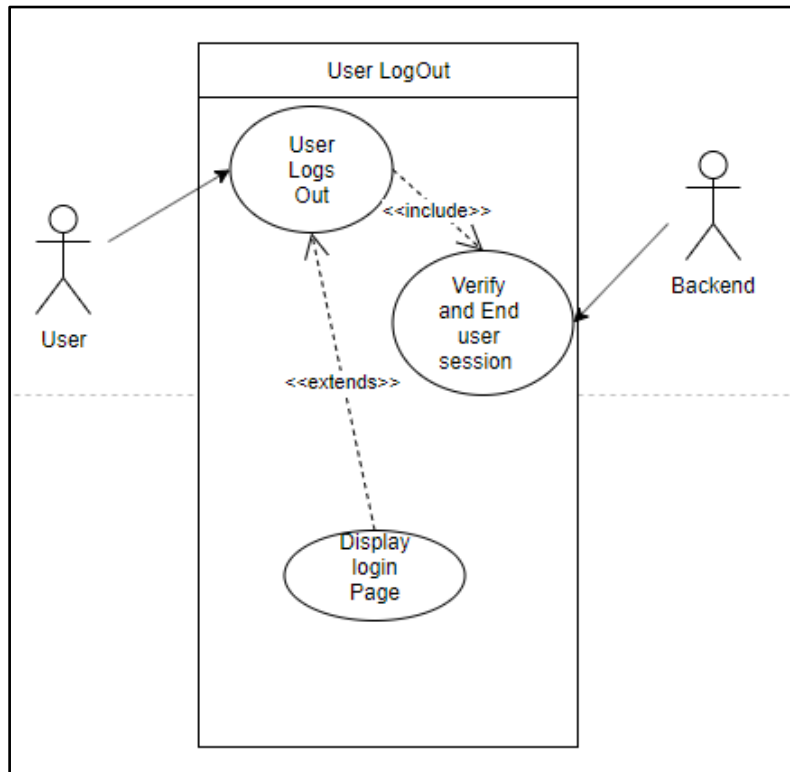
Use Case 5.



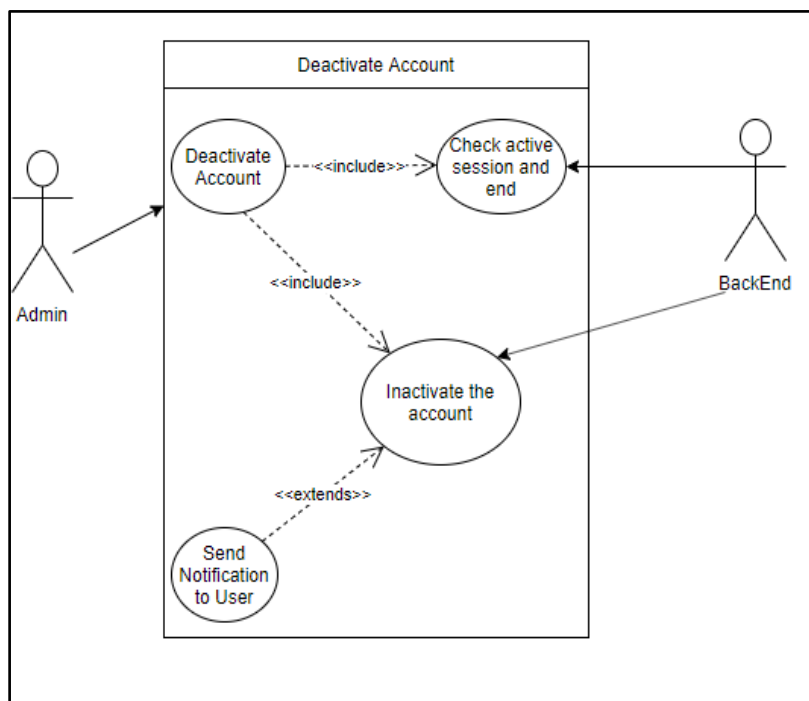
Use Case 6.



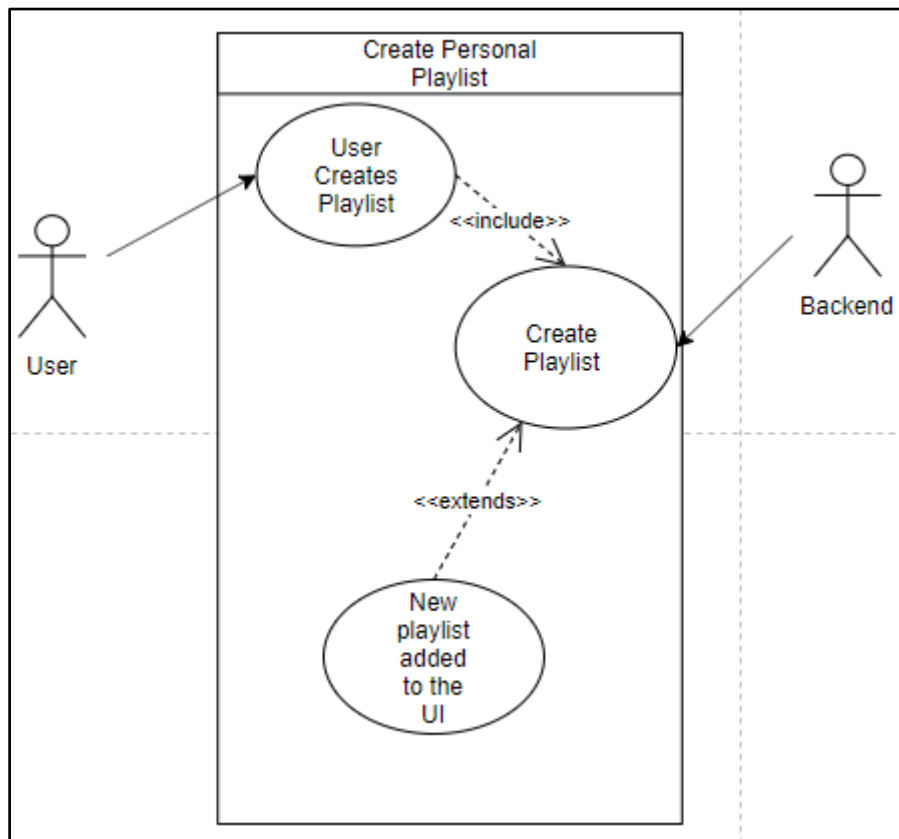
Use Case 7.



Use Case 8.



Use Case 9.



5.3.1 Use Case Descriptions

USE CASE 1	Create User Account
Goal in Context	User creates an account
Scope & Level	System, Primary Task.
Preconditions	User navigated to the create user page
Success End Conditions	The account has been created.
Failed End Conditions	The account has not been created. The user is not able to sign in. User redirected to signup page

Primary, Secondary, Actors	User, Computer	
Trigger	Register request comes in	
DESCRIPTION	Step	Action
	1	User fills in create account form
	2	User submits the form'
	3	Server verifies if the fields in the form are valid or not if valid the account is created successfully
	4	Servers saves account in db.
EXTENSIONS	Step	Branching Action
	3a	If information in the fields are not valid. 1a1. System notifies user.

USE CASE 2	Login	
Goal in Context	User logs in the system	
Scope & Level	System, Primary Task.	
Preconditions	User must have created an account first	
Success End Conditions	The user is able to log in successfully	
Failed End Conditions	The user is not able to Log in.	
Primary, Secondary, Actors	User, Computer	
Trigger	Login request comes in	
DESCRIPTION	Step	Action

	1	User enter username and password
	2	User submits the form
	3	Server verifies if the user is valid or not, if valid successful login
EXTENSIONS	Step	Branching Action
	3a	User name not valid. 1a1. System notifies user.

USE CASE 3	Reset password	
Goal in Context	The user wants to reset his password.	
Scope & Level	System, Primary Task.	
Preconditions	The user has an account.	
Success End Conditions	The password was changed.	
Failed End Conditions	The password wasn't changed.	
Primary, Secondary, Actors	User, Computer	
Trigger	Reset password request comes in.	
DESCRIPTION	Step	Action
	1	The user enters email address. The system verifies it.
	2	The user is prompted by the system to enter a new password
	3	User enters a new password
	4	The system updates the new password in DB
EXTENSIONS	Step	Branching Action

	1a	Email address is not verified. 1a1. System notifies user.
--	----	--

USE CASE 4	Verify Account	
Goal in Context	User verifies the account on his mail	
Scope & Level	System, Primary Task.	
Preconditions	User should have created an account	
Success End Conditions	Account verified	
Failed End Conditions	The account has not been verified. The user is not able to sign in.	
Primary, Secondary, Actors	User, Computer	
Trigger	Register request comes in	
DESCRIPTION	Step	Action
	1	User verifies the link on his email
	2	Server marks the user verified
	3	The user is able to login
EXTENSIONS	Step	Branching Action
	1a	User doesn't verify, the user will not be able to login 1a1. System notifies user.

USE CASE 5	Item Operation / Content Adding	
Goal in Context	User can perform CRUD operation on Items	
Scope & Level	System, Primary Task.	

Preconditions	User should have admin rights	
Success End Conditions	User performs CRUD operation on items successfully	
Failed End Conditions	The user is not able to perform CRUD operations	
Primary, Secondary, Actors	User, System	
Trigger	CRUD request comes in	
DESCRIPTION	Step	Action
	1	System checks whether user has admin rights
	2	User fills in content data/metadata
	3	User submits the form
EXTENSIONS	Step	Branching Action
	1a	User does not have admin rights 1a1. System notifies user.

USE CASE 6	List Operations	
Goal in Context	User can perform CRUD operation on Lists	
Scope & Level	System, Primary Task.	
Preconditions	User should have admin rights	
Success End Conditions	User performs CRUD operation on lists successfully	
Failed End Conditions	The user is not able to perform CRUD operations	
Primary, Secondary, Actors	User, System	
Trigger	CRUD request comes in	

DESCRIPTION	Step	Action
	1	System checks whether user has admin rights
	2	User fills in Lists metadata
	3	User submits the form
EXTENSIONS	Step	Branching Action
	1a	User does not have admin rights 1a1. System notifies user.

USE CASE 7	User Log Out	
Goal in Context	The user wants to logout of the session	
Scope & Level	System, Primary Task.	
Preconditions	The user is already logged in	
Success End Conditions	The user has been logged out	
Failed End Conditions	The user is not logged out	
Primary, Secondary, Actors	User, System	
Trigger	Logout request comes in.	
DESCRIPTION	Step	Action
	1	The user clicks Log Out
	2	The system checks user active session and ends the session
	3	User is displayed login page
EXTENSIONS	Step	Branching Action

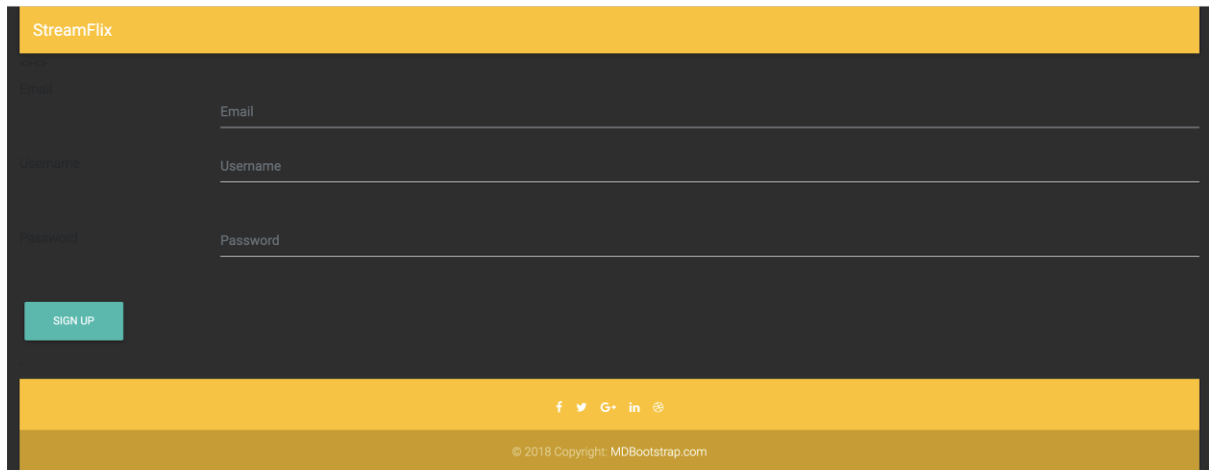
	1a	Email address is not verified. 1a1. System notifies user.
--	----	--

USE CASE 8	Deactivate Account	
Goal in Context	The Admin wants to deactivate an account	
Scope & Level	System, Primary Task.	
Preconditions	The account is present in the system	
Success End Conditions	Account deactivated and notification sent to user	
Failed End Conditions	The account is not deactivated	
Primary, Secondary, Actors	User, System	
Trigger	Deactivate account triggers comes in.	
DESCRIPTION	Step	Action
	1	The admin clicks de-active
	2	The system checks user's active session and ends the session
	3	Inactivate the account
	4	Send Notification to user
EXTENSIONS	Step	Branching Action
	3a	Account already inactivated 3a1. System notifies admin.

USE CASE 9	Create personal playlist
Goal in Context	The user wants to create personal playlist

Scope & Level	System, Primary Task.	
Preconditions	The user should be logged in	
Success End Conditions	Playlist created and added to the UI	
Failed End Conditions	Playlist is not created	
Primary, Secondary, Actors	User, System	
Trigger	Create playlist triggers comes in.	
DESCRIPTION	Step	Action
	1	The user creates a new playlist
	2	The system creates the playlist in the database
	3	New Playlist is added for the user
EXTENSIONS	Step	Branching Action
	1a	Playlist name already present 1a1. System notifies user.

6. User Interface Design



The Sign Up form features a yellow header with the 'StreamFlix' logo. The form area has a dark grey background with labels for Email, Username, and Password on the left, and corresponding input fields on the right. A teal 'SIGN UP' button is positioned below the fields. The footer includes social media icons and a copyright notice.

StreamFlix

Email

Username

Password

SIGN UP

f t G+ in

© 2018 Copyright: MDBootstrap.com

Figure 2. Sign Up Form



The Login page features a yellow header with the 'StreamFlix' logo. The form area has a dark grey background with labels for Email, Username, and Password on the left, and corresponding input fields on the right. A teal 'SIGN IN' button is positioned below the fields. The footer includes social media icons and a copyright notice.

StreamFlix

Email

Username

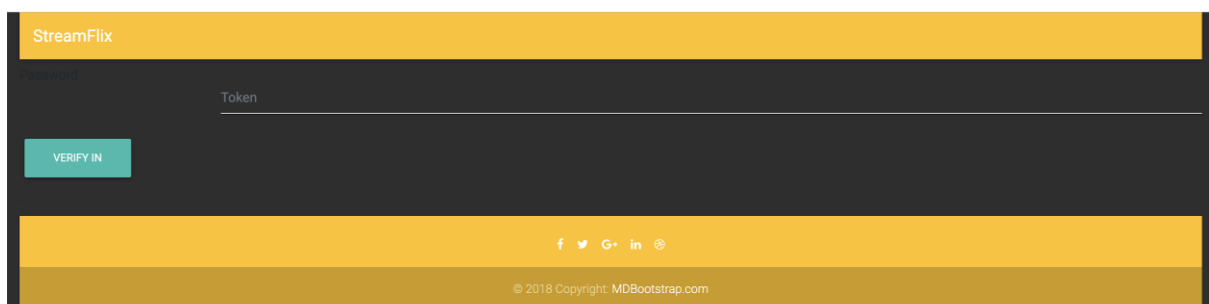
Password

SIGN IN

f t G+ in

© 2018 Copyright: MDBootstrap.com

Figure 2. Login Page



The Token Verification page features a yellow header with the 'StreamFlix' logo. The form area has a dark grey background with labels for Password and Token on the left, and corresponding input fields on the right. A teal 'VERIFY IN' button is positioned below the fields. The footer includes social media icons and a copyright notice.

StreamFlix

Password

Token

VERIFY IN

f t G+ in

© 2018 Copyright: MDBootstrap.com

Figure 3. Token Verification Page

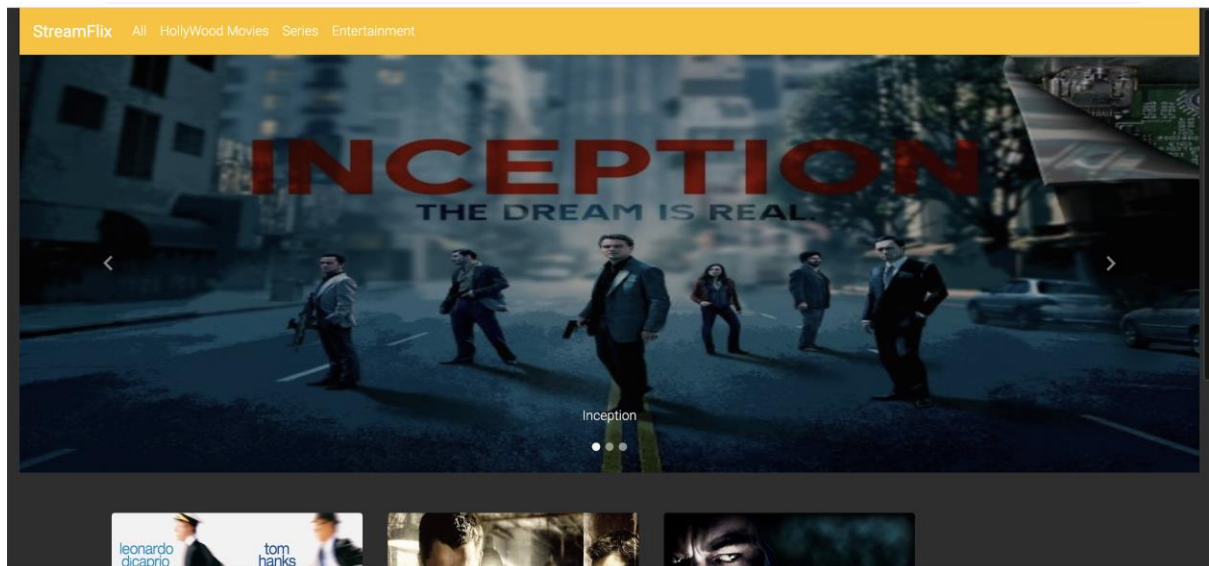


Figure 4. Main Page

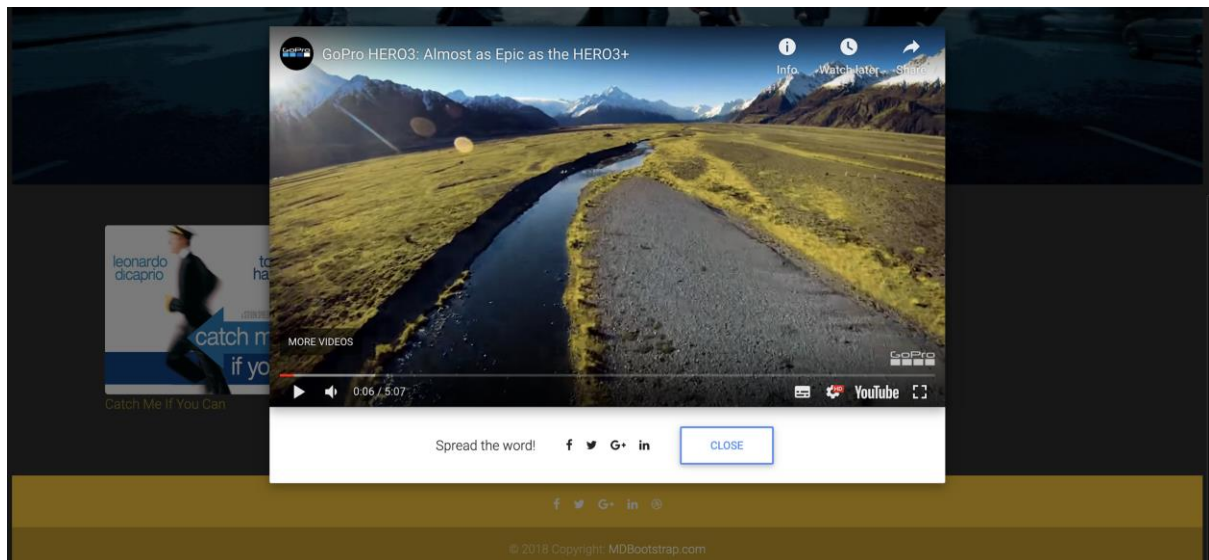
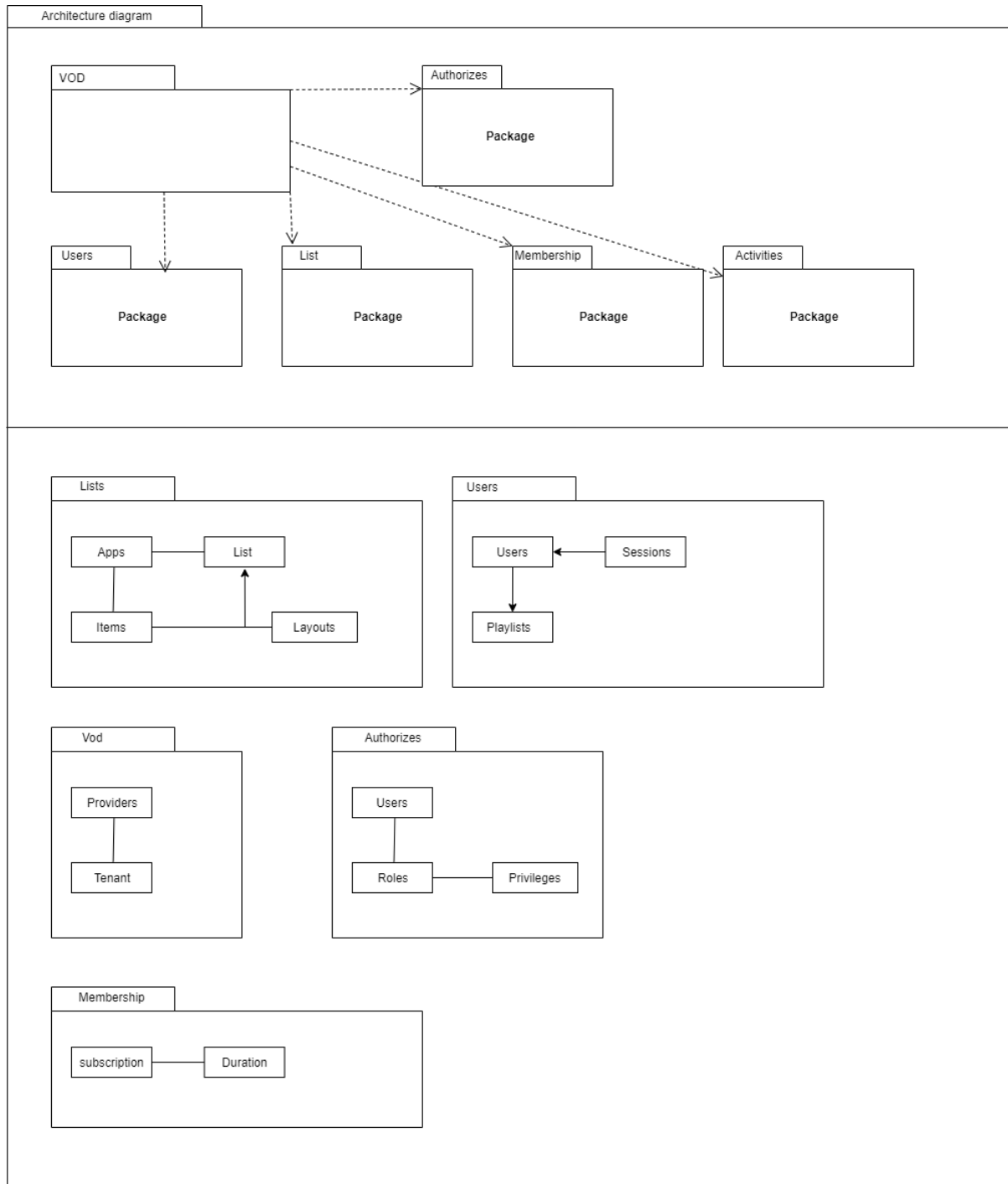


Figure 5. Page for Accessing Content

7. System Architecture

7.1 Package Diagram



7.2 Technology Pipeline

Puma

Puma was built for speed and parallelism. Puma is a small library that provides a very fast and concurrent HTTP 1.1 server for Ruby web applications. It is designed for running Rack apps only. What makes Puma so fast is the careful use of a Rake extension to provide fast, accurate

HTTP 1.1 protocol parsing. This makes the server scream without too many portability issues.

PostgreSQL [8]

PostgreSQL is a powerful, open-source object-relational database system that uses and extends the SQL language combined with many features that safely store and scale the most complicated data workloads. The origins of PostgreSQL date back to 1986 as part of the POSTGRES project at the University of California at Berkeley and has more than 30 years of active development on the core platform.

PostgreSQL has earned a strong reputation for its proven architecture, reliability, data integrity, robust feature set, extensibility, and the dedication of the open source community behind the software to consistently deliver performant and innovative solutions. PostgreSQL runs on all major operating systems, has been ACID-compliant since 2001, and has powerful add-ons such as the popular PostGIS geospatial database extender. It is no surprise that PostgreSQL has become the open source relational database of choice for many people and organisations.

Below is an inexhaustive list of various features found in PostgreSQL, with more being added in every major release:

Data Types

Primitives: Integer, Numeric, String, Boolean

Structured: Date/Time, Array, Range, UUID

Document: JSON/JSONB, XML, Key-value (Hstore)

Geometry: Point, Line, Circle, Polygon

Customizations: Composite, Custom Types

Data Integrity

UNIQUE, NOT NULL

Primary Keys

Foreign Keys

Exclusion Constraints

Explicit Locks, Advisory Locks

Concurrency, Performance

Robust access-control system
Column and row-level security
Multi-factor authentication with certificates and an additional method
Extensibility
Stored functions and procedures
Procedural Languages: PL/PGSQL, Perl, Python (and many more)
SQL/JSON path expressions
Foreign data wrappers: connect to other databases or streams with a standard SQL interface
Internationalisation, Text Search
Case-insensitive and accent-insensitive collations
Full-text search

Bundler:

Bundler provides a consistent environment for Ruby projects by tracking and installing the exact gems and versions that are needed. Bundler is an exit from dependency hell, and ensures that the gems you need are present in development, staging, and production. Starting work on a project is as simple as `bundle install`.

Rails:

Ruby on Rails, or Rails, is a server-side web application framework written in Ruby under the MIT License. Rails is a model–view–controller (MVC) framework, providing default structures for a database, a web service, and web pages. It encourages and facilitates the use of web standards such as JSON or XML for data transfer, HTML, CSS and JavaScript for user interfacing. In addition to MVC, Rails emphasizes the use of other well-known software engineering patterns and paradigms, including convention over configuration (CoC), don't repeat yourself (DRY), and the active record pattern.

Angular:

Angular (commonly referred to as "Angular 2+" or "Angular v2 and above") is a TypeScript-based open-source web application framework led by the Angular Team at Google and by a community of individuals and corporations. Use modern web platform capabilities to deliver app-like experiences. High performance, offline, and zero-step installation. Build native mobile apps with strategies from Cordova, Ionic, or NativeScript. Create desktop-installed apps across Mac, Windows, and Linux using the same Angular methods you've learned for the web plus the ability to access native OS APIs.

Bootstrap

Build responsive, mobile-first projects on the web with the world's most popular front-end component library.

Bootstrap is an open-source toolkit for developing with HTML, CSS, and JS. Quickly prototype your ideas or build your entire app with our Sass variables and mixins, responsive grid system, extensive pre built components, and powerful plugins built on jQuery.

8. System Architecture Patterns

8.1 MVC

Model-View-Controller known as MVC is an architectural design pattern, which divides a particular system into three different components. Each component serves its responsibility and by interacting with each component at the same time. Model component is responsible for Data in the system. It is responsible for the data integrity by serving the constraints required to the system. Most of the frameworks provide an ORM to deal with the Model. For example: "Activerecord" is the ORM provided by the Rails. Controller component focus on the business logic that is required to be served for each request. It communicates with the Model get the data or manipulate the data and then it formats the data so that it can then be utilised by the views. View is responsible for the presentation of the data of the system. Views hands over the request to specific controller and then get the response formatted and then renders it to the presentation layer the way it is designed.

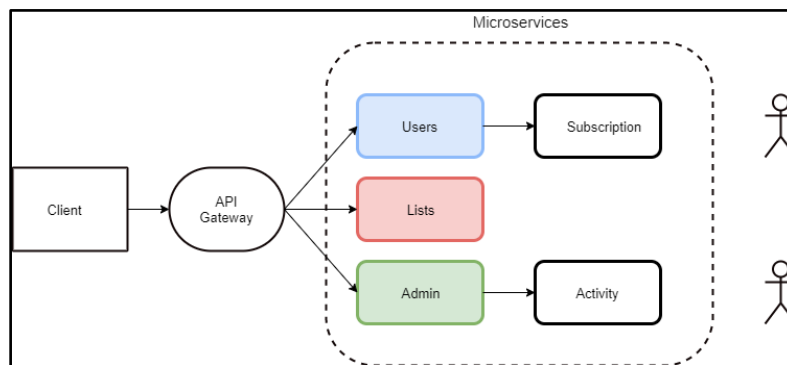
MVC framework is extremely beneficial to an agile environment as it allows for parallel development. It is to say that one person can be responsible for working on models defining the constraints and another person focusing on the business logic and at last another person can focus on the user interface. This decomposition allows for back-end and front-end developers to develop their respective code simultaneously.

8.2 Additional Architectural Pattern: Microservices Architectural Pattern

Micro Service Architecture approach deals with identifying the parts or the component in a complete system that can be segregated, programmed and maintained independently. In our project of design, a VoD platform it was obvious for us to segregate User, Lists, Admin and VoD Platform separately. Each service does its work and maintained separately. Each service is very independent of choosing the stack that needs to be built upon. It allows the liberty of choosing the technology for the implementation. Since it allows independent deployment strategy it can be scaled up easily just adding another service for only a particular service depending upon its usage. Also, it makes development to carry on independently. Developers don't have

to wait for any dependency but can work on their own service and then connect it in the end. There is no limitation in selecting the database as well for the service. Any service can choose its required technology very focused on the tasks it performs. This makes micro service our first choice. Each and every API developed in our system is based on REST CRUD operations. So, it makes it easy across all service and common understanding of the response is maintained.

Figure 6. Microservices Architecture diagram



8.3 Multi-Tenant Architecture

Multi-tenant architecture is a way of supporting different set of users on the same platform. In multi tenancy you can add multiple customers in our case its providers without changing anything in the running software. Several customers can use the software without knowing on how the data segregation is done in the backend. In this architecture data will be maintained separately in either different database or different schema in a database. In our VoD platform we are using PostgreSQL schema to separate the data from each tenant. Each customer will access the data with a specified subdomain to access that particular tenant data. Based on the subdomain database schema is switched to provide those particular data.

9 Design Patterns

9.1 Factory Pattern

In Factory pattern, we create object without exposing the creation logic to the client and refer to newly created object using a common interface.

The related class and interfaces are as below:

- *UserMicroService*: Based on the requests that has reached it will trigger the API request to its respective service. Using the configuration and specification related to that particular service.
- *ListMicroService*: It is responsible for handling the service related to content. It redirects the call to respective port number that handles the contents.
- *AdminMicroService*: It handles all the authorization of creating/updating and deleting the content.

Based on the requests, Microservice will handle/redirects the call to respective services handling the data. It identifies the resource trying to access and initialise the respective microservice and the port number that is required in the context. In case there is a need to create a new service to handle the subscription in the VoD platform, need to define a new class to it and then handles it the same way as it is handled for other services. So, in this way it is easy to add new services to the system.

9.2 Decorator Design Pattern

Decorator is structural Design Pattern which allows us to define/modify/create new functionality without affecting core structure. This pattern extends its functionality by wrapping it with helper class. In our VOD platform we have defined decorators for index/show API calls. Index and Show requests are API where contents are retrieved from the database. Index call will fetch all entries from the database whereas Show call will fetch particular record. Response body often changes with requirement. We have used design pattern to accommodate the changes that will occur.

Below is the participant:

CompositeListDecorator

ConciseListDecorator

```
86   def index_decorator
87     ic = IndexComponent.new
88     cl = CompositeListDecorator.new(ic)
89   end
90
91   def show_decorator
92     ic = ShowComponent.new
93     sh = ConciseListDecorator.new(ic)
94   end
95
```

Figure 7. Decorator participants

9.3 Singleton

Singleton is a creational design pattern that lets you ensure that a class has only one instance, while providing a global access point to this instance.

In our VoD platform, singleton pattern is used to make sure we only have one micro service instance is instance is created. Memoisation is used for instantiation. Since there in one instance required for each request it is perfect fit for the singleton pattern to be used.

9.4 Chain of Responsibility

Chain of responsibility is a pattern where set of handlers are defined and each request is handled one after the other. Defining the workflow before the work is being executed. In our VoD platform we have implemented chain of responsibility for each user registration. Whenever a user is created in the CMS, set of responsibility is carried out. User password has to be encrypted and before saving email notification has to be sent out. In this case we are setting the responsibility chain for each creation and for each user registration we are calling the responsibility one after the other.

```
def fetch_handler
  encrypt = EncryptPassword.new
  notify = SendNotification.new
  encrypt.next_handler(notify)
  encrypt
end
```

Figure. 8. Chain of Responsibility

9.5 State Pattern

In State pattern a class behaviour changes based on its state. This type of design pattern comes under behaviour pattern. In State pattern, we create objects which represent various states and a context object whose behaviour varies as its state object changes. In our VoD platform state of the website changes based on the item/content state. Based on the state of the item, it will appear on the front end. If the state is published then it will appear in the front end and transcoding to various profile will happen. Whereas if the state is unpublished it will not appear in the front end and keeps waiting for the approval to be published.

```
def state_changed
  c = Context.new
  ps = ItemState(c)
  ps.published_state
end
```

Figure 9. State Pattern

9.6 Facade

Facade is a structural design pattern that provides a simplified interface to a library, a framework, or any other complex set of classes. It is used to aggregate set of operation and perform at once. In our VoD platform, any particular resource might also need append the resource information of the other resource. We are using facade design pattern for each resource on what it requires. In this way it is easy to segregate the data requirement from the data manipulation logic. It is also easier to add more facade classes on the various requirements.

```
8   def associated_model_attributes
9     m = MediumAttributes.new
10    l = LayoutAttributes.new
11    i = ItemAttributes.new
12    s = SublistAttributes.new
13    return IndexFacade.new(m, l, i, s).index_attributes
14  end
15
```

Figure 10. Façade

9.7 Observer Pattern

Observer pattern lets us define a subscription mechanism in order to communicate with multiple objects about any events that occurred to the object that is being observed. In our VOD platform, always user activities are observed. In CMS it is always observed that particular user can do particular set of activities. If a user tries to perform unauthorised actions it has to be notified to the admin. Whenever system detects an unauthorised action performed by the user, then email is sent and it is logged in the analytics data for further analysis on the actions.

```
def validate_privilege
  subject = UnauthorisedAccesss.new
  email_observer = EmailObserver.new
  subject.attach(email_observer)
  recorder_observer = RecorderObserver.new
  subject.attach(recorder_observer)

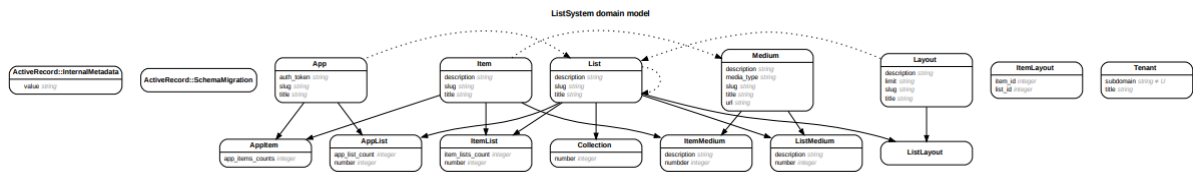
  privilege = params[:authorize_action] # + '_' + params[:controller]
  privileges = params[:session].user.role.privileges.map(&:title)
  |
  if privileges.include? privilege
    render json: {message: "Valid User"}, status: :ok
  else
    subject.alaram_operation privilege
    render json: {message: "Invalid User"}, status: :unprocessable_entity
  end
end

end
```

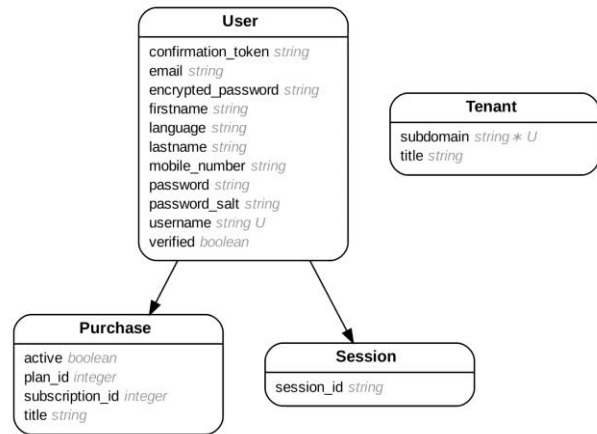
Figure 11. Observer pattern

10.2 ERD Diagram

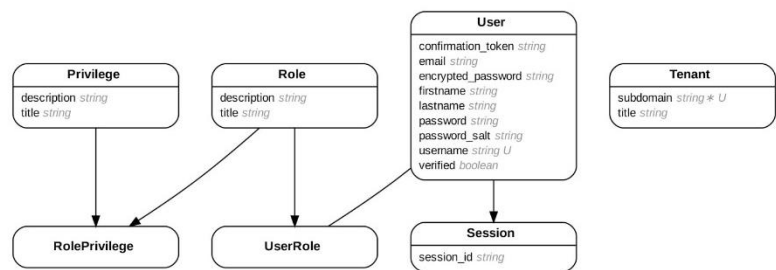
List System ERD



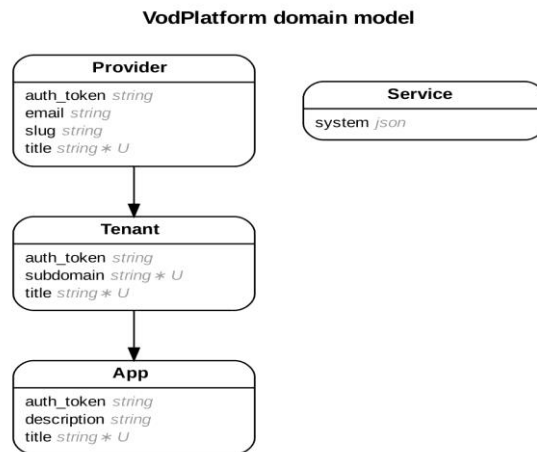
User System Domain Model



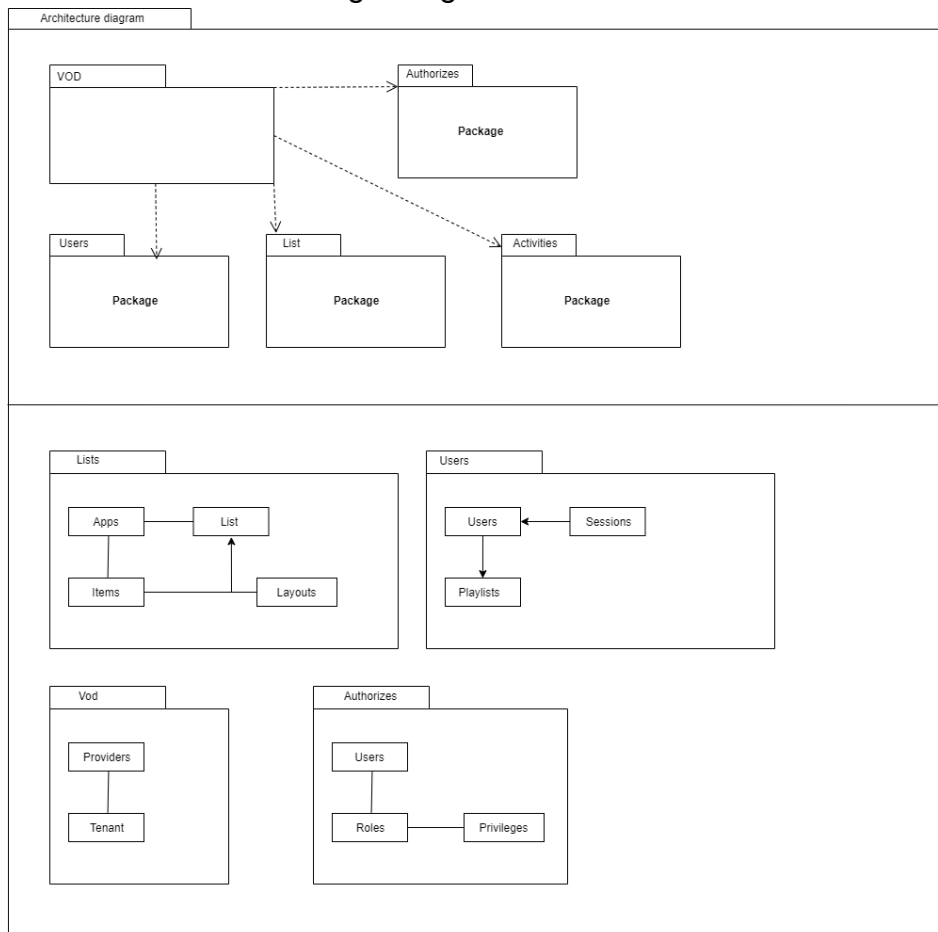
Admin System ERD



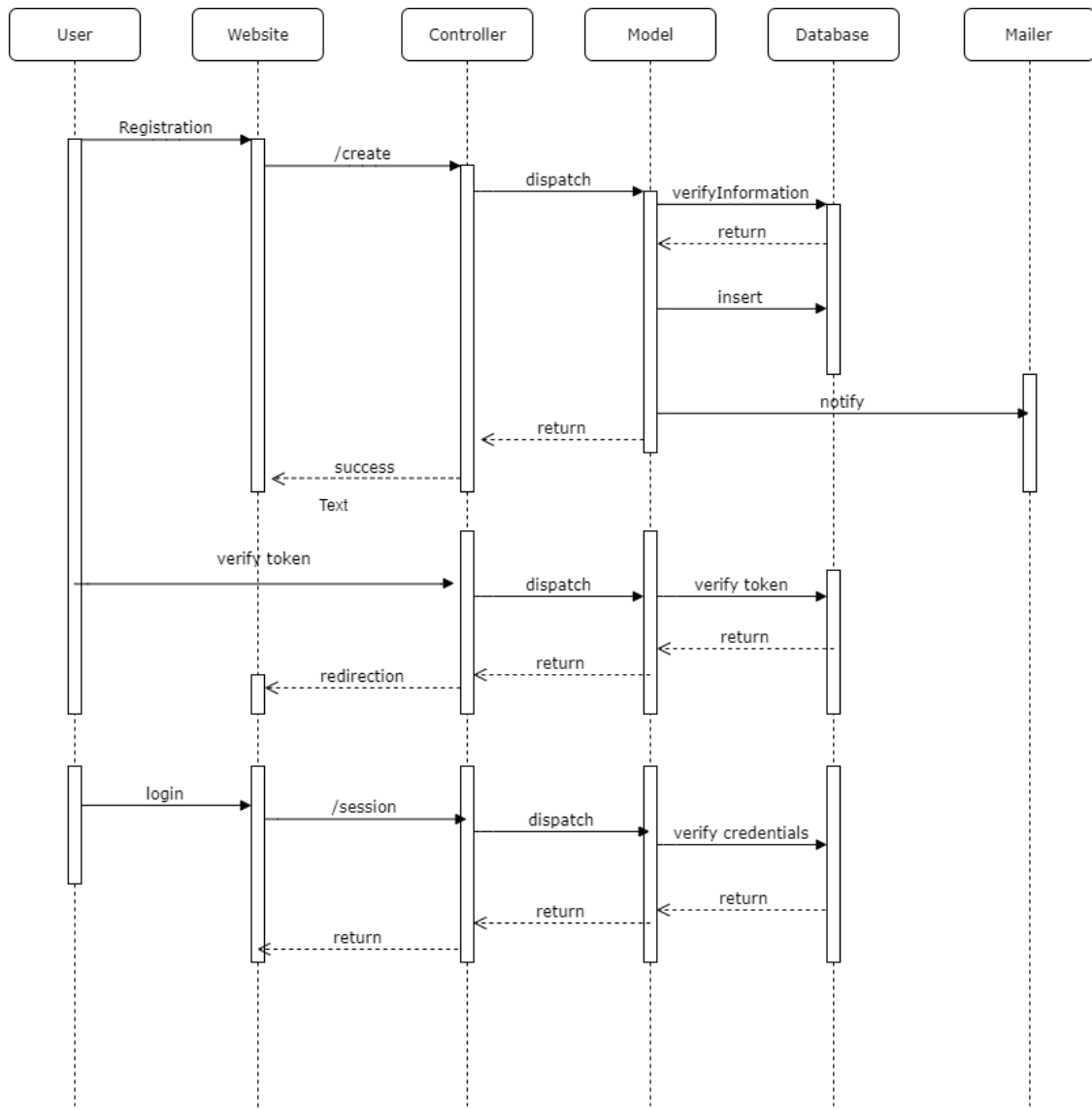
VoD Platform Domain Model



10.4 Architecture/Package Diagram

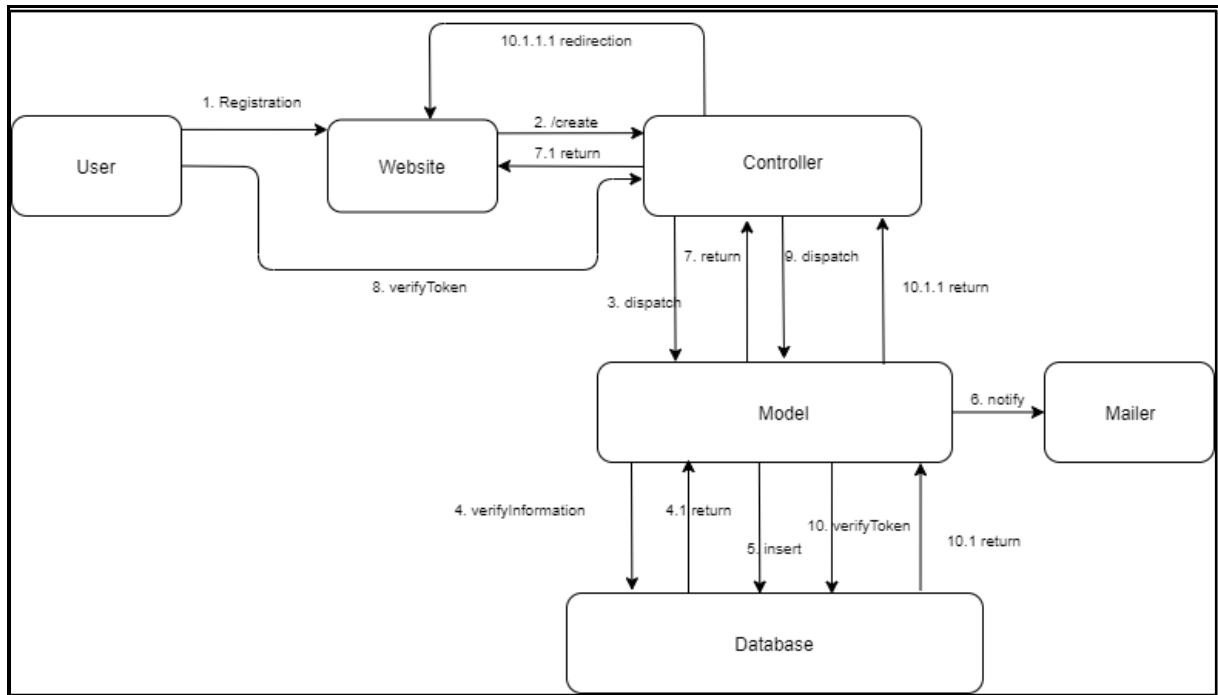


10.4 Sequence Diagram Registration & Login

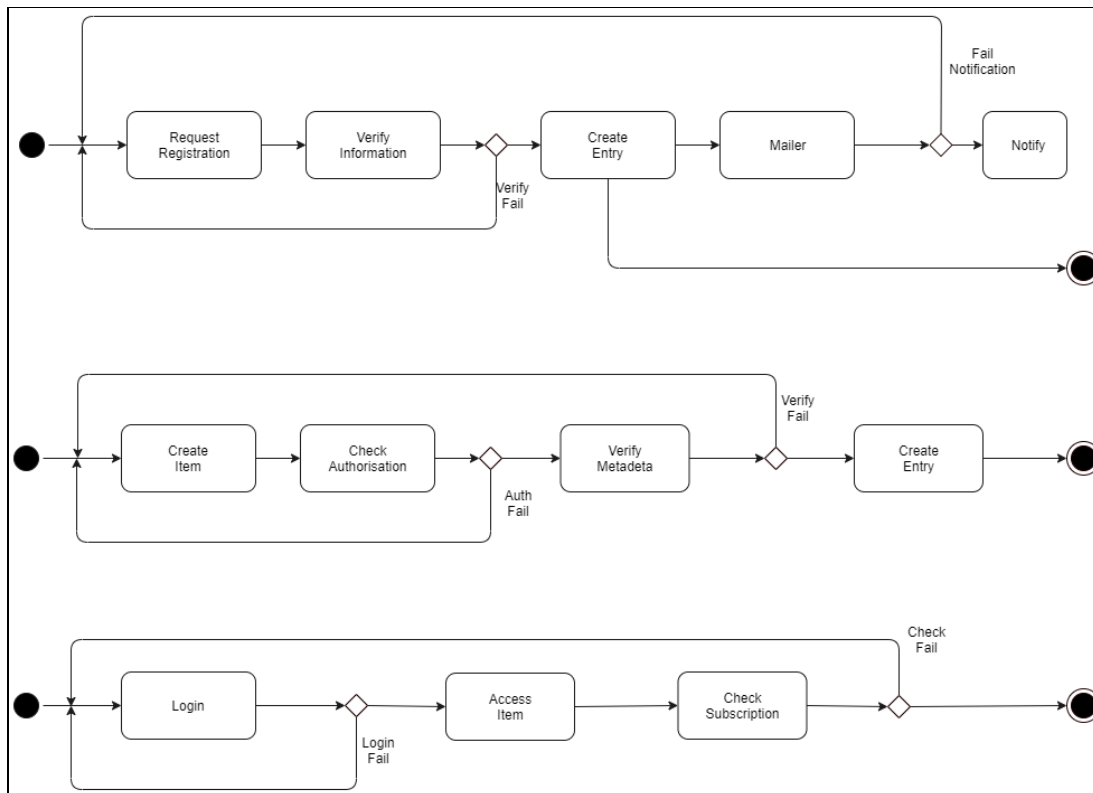


10.5 Communication Diagram

Registration

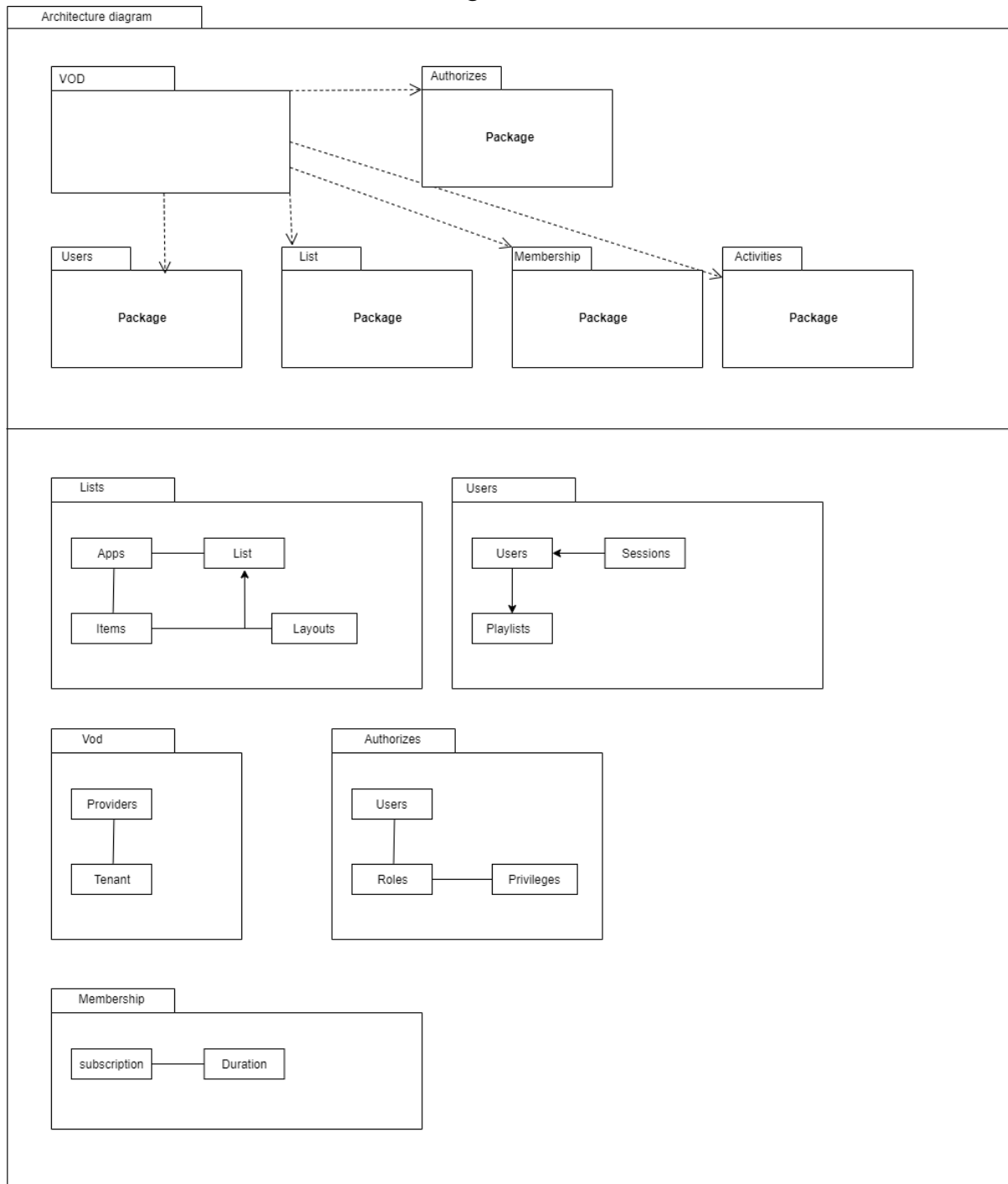


10.6 Activity Diagram

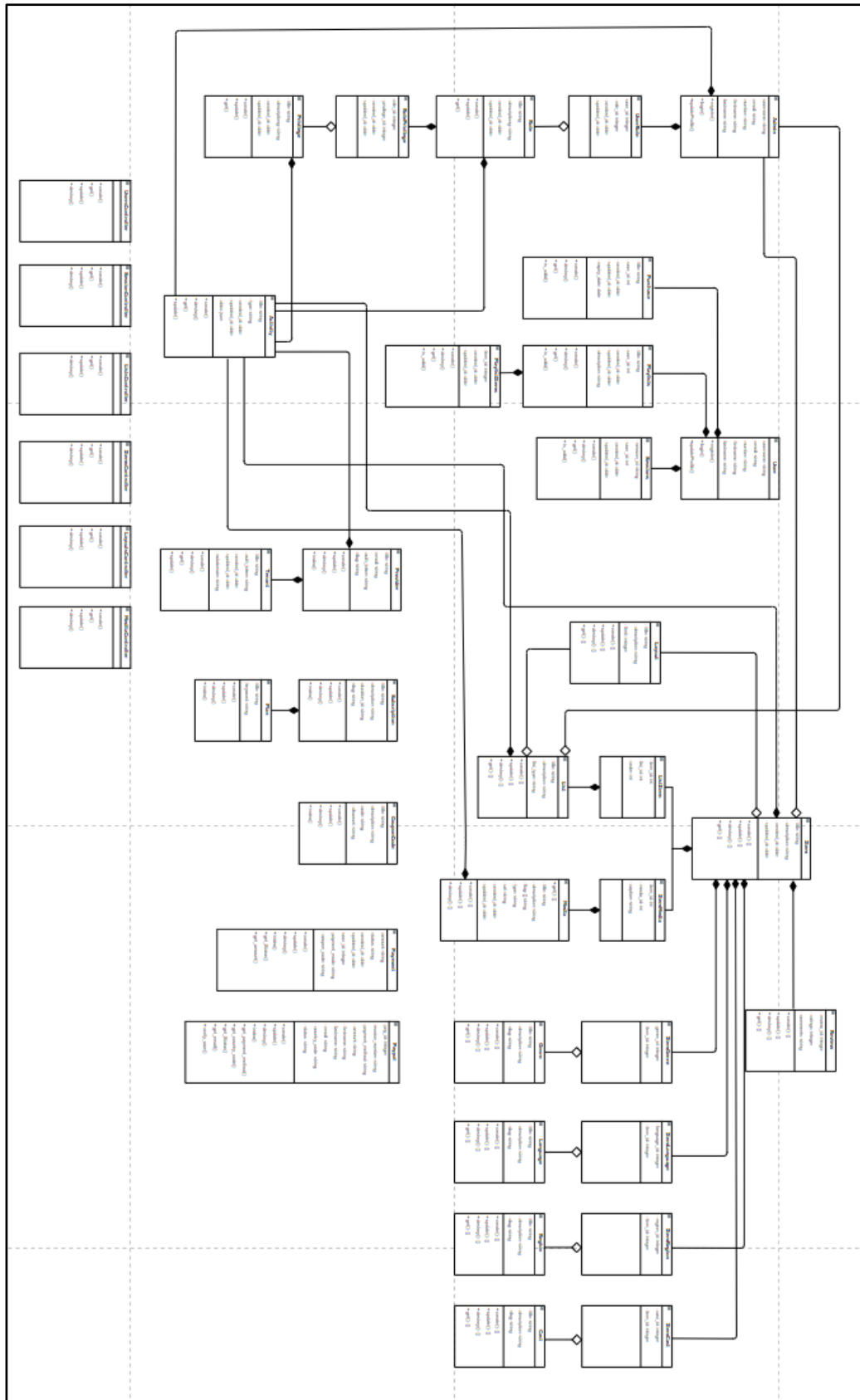


11. Recovered Design & Architectural Blueprints

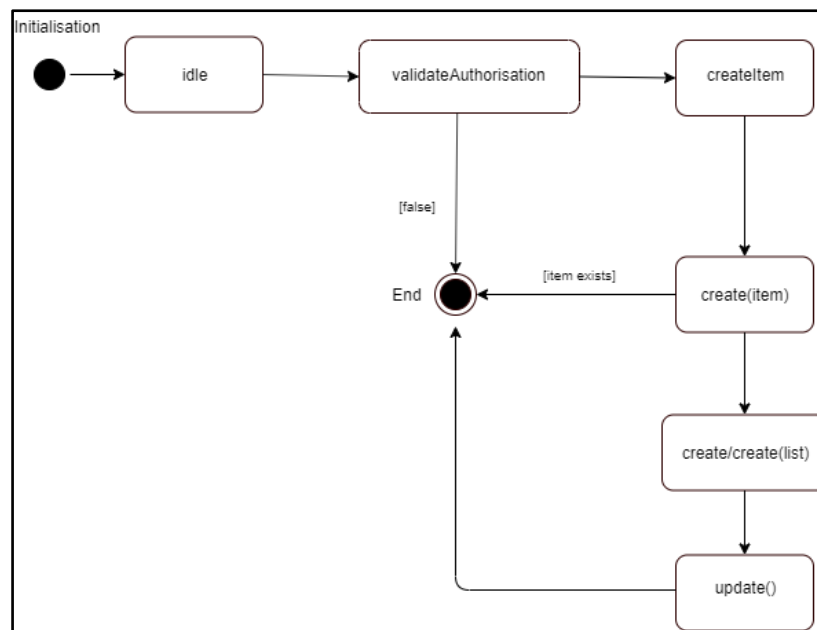
11.1 Recovered Architectural Diagram



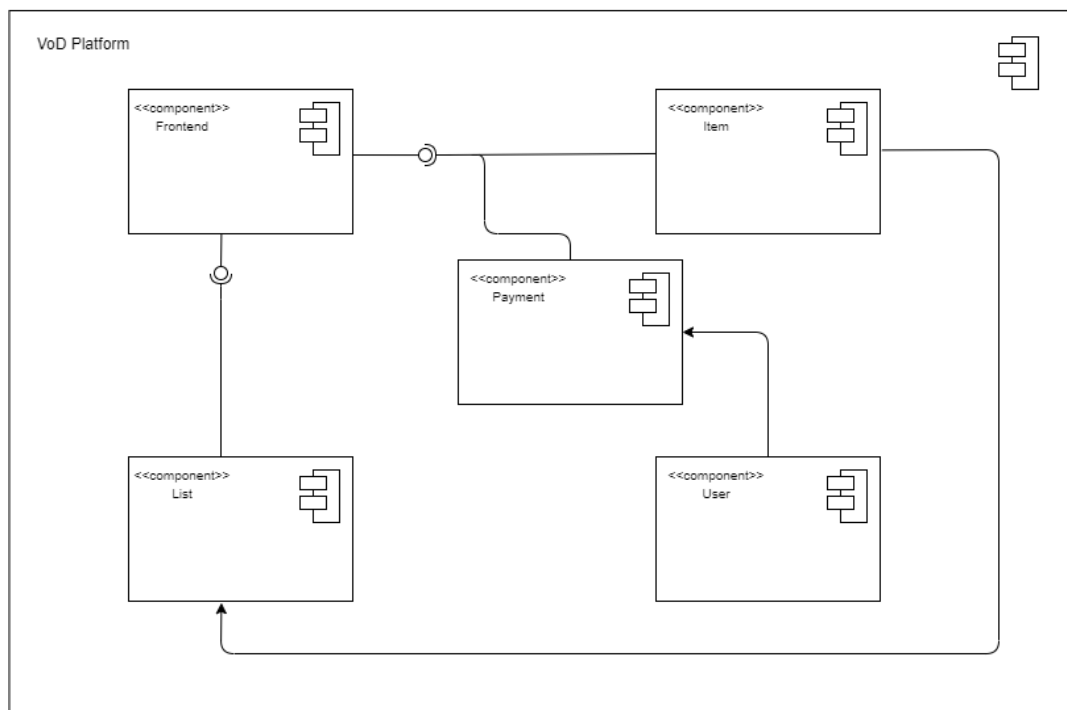
11.2 Recovered Class Diagram



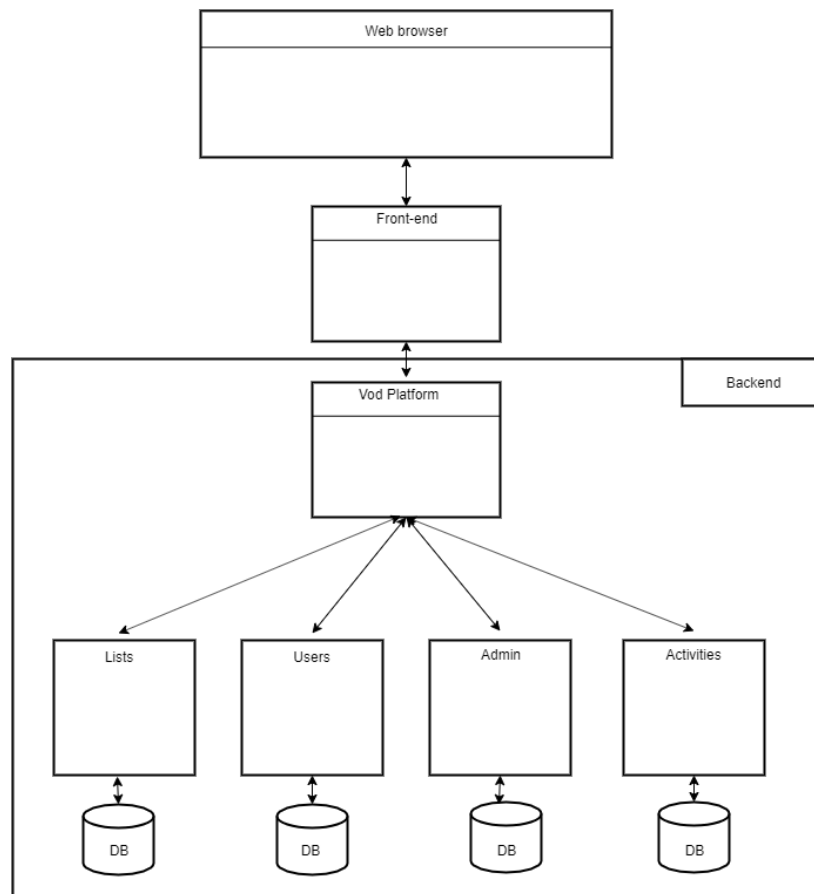
11.3 State Diagram



11.4 Component Diagram



11.5 Deployment Diagram



12. Test Cases

Test cases are developed using Rails built in Minitest. Which makes it easy to implement testing before the API is completely developed. Rails framework generates a test files for each controller and models. By writing test cases we are ensuring our code adheres to the required functionality even after code refactoring. Rails tests simulates browser requests and hence can test application's response without having to test it through browser.

Figure 12. Test cases

```
1 | require 'test_helper'
2 | require 'json'
3 |
4 | class ProvidersControllerTest < ActionDispatch::IntegrationTest
5 |
6 |   test "Should get valid list of providers" do
7 |     get '/providers?authToken=abc'
8 |     assert_response :success
9 |     assert_equal response.content_type, 'application/json'
10 |    jdata = JSON.parse response.body
11 |    assert_equal 2, jdata['providers'].length
12 |    assert_equal jdata['providers'][0]['title'], 'Netflix'
13 |  end
14 |
15 |   test "Should get valid provider data" do
16 |
```

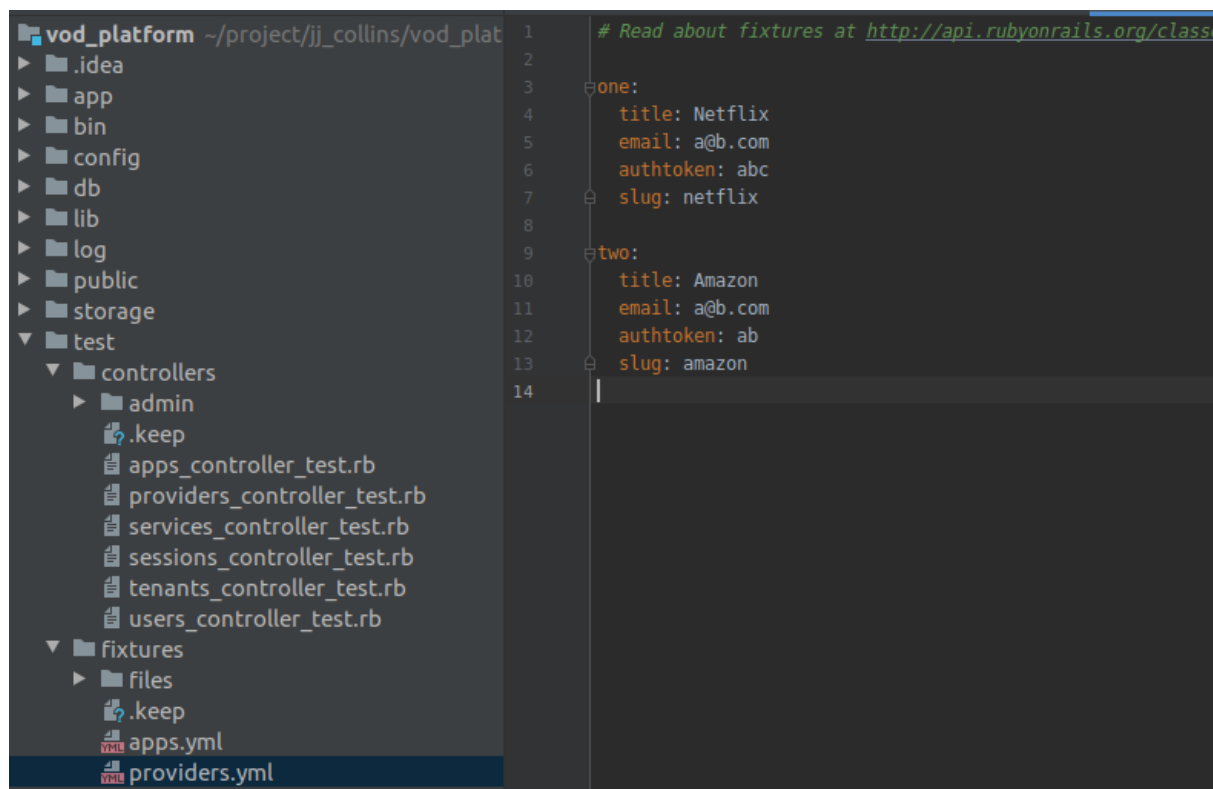
The screenshot shows the IntelliJ IDEA interface. On the left, the project structure is visible, including folders like .idea, app, bin, config, db, lib, log, public, storage, and test. The test directory is expanded, showing controllers, admin, .keep, apps_controller_test.rb, and providers_controller_test.rb. The main editor displays the content of providers_controller_test.rb, which includes test cases for the ProvidersController. The first test case is "Should get valid list of providers", which sends a GET request to /providers?authToken=abc and asserts that the response is successful, the content type is application/json, and the response body contains 2 providers, with the first provider having the title 'Netflix'.

12.1 Example Test 1

Before running the tests, cases fixtures are created for the tests to verify against. Rails will have separate test database where all the data will be created and will flushed once testing is done.

Example features are as below:

Figure 13. Test database



All test cases are tested for 200 response and expected response. Below are the sample test cases:

```
1 require 'test_helper'
2 require 'json'
3
4 class ProvidersControllerTest < ActionDispatch::IntegrationTest
5
6   test "Should get valid list of providers" do
7     get '/providers?authToken=abc'
8     assert_response :success
9     assert_equal response.content_type, 'application/json'
10    jdata = JSON.parse response.body
11    assert_equal 2, jdata['providers'].length
12    assert_equal jdata['providers'][0]['title'], 'Netflix'
13  end
14
15
16   test "Should get valid provider data" do
17     provider = providers('one')
18     get "/providers/#{provider.id}?authToken=abc" #, params: { id: provider.id }
19     assert_response :success
20     jdata = JSON.parse response.body
21     puts jdata
22     assert_equal provider.id, jdata['id']
23     assert_equal provider.email, jdata['email']
24   end
25
26   test "Should get JSON:API error block when requesting provider data with invalid ID" do
27     get "/providers/invalid?authToken=abc"
28     assert_response 404
29     jdata = JSON.parse response.body
30     assert_equal "Wrong ID provided", jdata['errors'][0]['detail']
31     assert_equal '/data/attributes/id', jdata['errors'][0]['source']['pointer']
32   end
33
34 end
```

Figure 14. Test Cases

13. Added Value

13.1 REST Architectural Pattern

We decided to go API only platform. RESTful applications use HTTP requests to POST data to create any resource, GET data to retrieve the data created in the system, UPDATE data to update the existing data, DELETE data to destroy the data in the system. We have implemented CRUD for all the resources in each micro service.

13.2 Object Relational Mappers (ORM)

Active Record is the M in MVC - the model. In rails each model is represented by class. Each Class represents table inside a database. It follows active record pattern for accessing data in a database. After creation of an object, a new row is added to the table upon save.

13.3 Use of Microservice

We have implemented our VoD platform with micro-service architecture in mind. Resources which are closely associated with each other are clubbed and a microservice CRUD operations are created for the same. API gateway is available in the system which communicates with all other microservice making it simpler for the front-end application.

13.4 Framework:

We have used Rails framework for implementation of Each micro service. Rails follows MVC architecture pattern. Each micro service renders json API, Each API is required to send an auth_token to authenticate the request. We have implemented API Gateway to communicate with other microservice, the front end communicates with microservice through API gateway.

13.5 GitHub Actions:

We have integrated Continuous integration to GitHub via GitHub Actions. It is configured to run the test for each push operation. When a new feature is added and pushed GitHub action will get trigger and runs all tests.

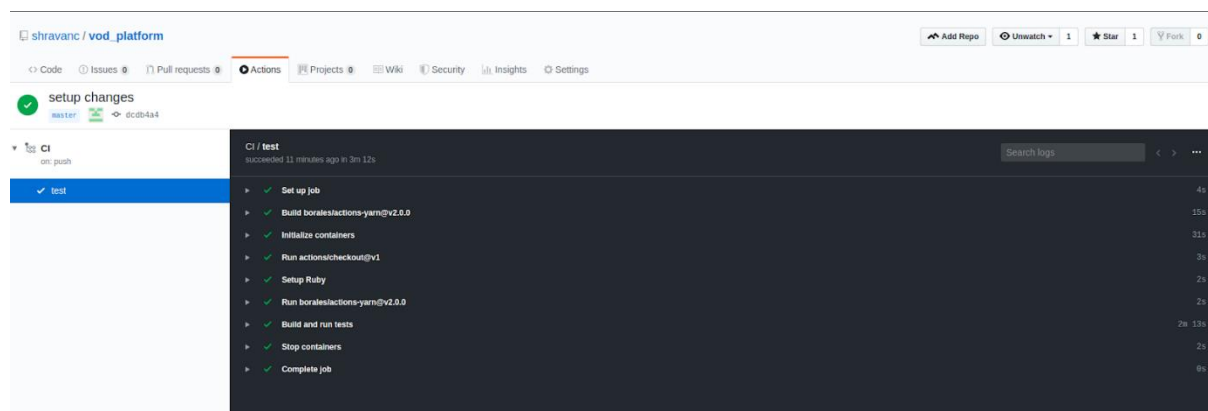


Figure 15. GitHub Actions

13.6 Multi-Tenant Architecture:

Multi-tenant architecture allows to create multiple customers dynamically without changing any existing code. This will expand the usability of the platform and can reach out too many different customers focusing on different set of users. Just by using different subdomain for accessing the data whole separate data is rendered.

14. Critiques

After going through many design patterns, we chose to go with multi-tenant microservice architecture [1][2]. But we are not having much idea about implementing the multi tenancy which should dynamically change the schema underneath the application. For this we need to do research on this and finally came up with the logic of handling it with PostgreSQL [4][8]. This helped us seamlessly switch between the schema underneath. And then when implementing the microservices, all the services endpoints were exposed to the public. And it became difficult at the front end to remember all hosts for all services, then after doing research on this topic. We came up with API gateway concept [3].

Once this is decided to go with API gateway, we needed a proper way of identifying and call respective services from the API gateway, for this we integrated Factory design pattern [5][6]. This helped us manage micro service very easily. Adding another micro service became very easy from this implementation.

When it comes to managing the content, we need an authorization and a way to keep track of unauthorised access, so we had to notify all such activity. For this we came up with Observer design pattern [7] to notify all the existing observer for such activity. Then as we move on with the integration of the system, we got many ideas about implementation of design patterns into existing system. This is our learning from this project, when there is a problem to solve there is a design pattern waiting to accept it. All members contributed equally in coming up with the ideas and discussing it to be accurate and suitable enough to any problem/issue. Also, Trello was used by all members as a documentation/progress purpose.

15. Appendices

Table 1.

Package	File	Author(s)	LoC
VoD platform		Shravan	1188
	providers_controller.rb		
	sessions_controller.rb		
	tenants_controller.rb		
	users_controller.rb		
	items_controller.rb		
	layouts_controller.rb		
	lists_controller.rb		
	media_controller.rb		
	roles_controller.rb		
	sessions_controller.rb		
	users_controller.rb		
	provider.rb		
	service.rb		
	tenant.rb		
	services.rb		
	micro_service.rb		
List System		Assad Nawaz	1199
	application_controller.rb		
	apps_controller.rb		
	items_controller.rb		
	layouts_controller.rb		
	lists_controller.rb		

	media_controller.rb		
	tenants_controller.rb		
	app.rb		
	app_item.rb		
	app_list.rb		
	application_record.rb		
	collection.rb		
	item.rb		
	item_layout.rb		
	item_list.rb		
	item_medium.rb		
	layout.rb		
	list.rb		
	list_layout.rb		
	list_medium.rb		
	medium.rb		
	tenant.rb		
	component.rb		
	composite_list_decorator.rb		
	concise_list_decorator.rb		
	decorator.rb		
	index_component.rb		
	list_decorator.rb		
	show_component.rb		
	index_facade.rb		

	item_attributes.rb		
	layout_attributes.rb		
	medium_attributes.rb		
	show_facade.rb		
	sublist_attributes.rb		
	context.rb		
	item_state.rb		
	list_state.rb		
	state.rb		
User System	channel.rb	Reezvee S	687
	connection.rb		
	application_controller.rb		
	privileges_controller.rb		
	purchases_controller.rb		
	roles_controller.rb		
	sessions_controller.rb		
	tenants_controller.rb		
	users_controller..rb		
	application_job.rb		
	application_mailer.rb		
	user_mailer.rb		
	purchase.rb		
	application_record.rb		
	session.rb		
	tenant.rb		
	user.rb		
	mailer.html.erb		

	mailer.text.erb		
	forgot_password.html.erb		
	welcom_email.html.erb		
	welcome_email.text.erb		
Activity	channel.rb	Reezvee S	211
	connnection.rb		
	actions_controller.rb		
	applications_controller.rb		
	application_job.rb		
	appliacion_mailer.rb		
	actions.rb		
	mailer.html.erb		
	mailer.text.erb		
Portal		Assad/Reezvee	1566
	app.module.ts		
	app-routing.module.ts		
	app.component.ts		
	list-service.service.ts		
	carousal-one-layouts.component.ts		
	navigation.component.ts		
	navigation.component.html		
	register.component.ts		
	register.component.html		
	login.component.ts		
	login.component.html		
	verify.component.html		
	verify.component.ts		
	carousal-one-layouts.components.html		

	carousal-one-layouts.components.ts		
	footer.component.ts		
	footer.component.html		
	gallery-one.component.ts		
	gallery-one.component.html		
	website-body.component.ts		
	website-body.component.html		
Admin		Shravan	918
	privileges_controller.rb		
	roles_controller.rb		
	sessions_controller.rb		
	tenants_controller.rb		
	users_controller.rb		
	privilege.rb		
	role_privilege.rb		
	role.rb		
	session.rb		
	tenant.rb		
	user.rb		
	user_role.rb		
	unauthorised_access.rb		
	subject.rb		
	observer.rb		
	recorder_observer.rb		
	email_observer.rb		
	send_notification.rb		
	encrypt_password.rb		
	abstract_handler.rb		
	handler.rb		

Table 2.

Student Name	Lines of Code
Reezvee S	1681
Assad Nawaz	1982
Shravan C	2106

Table 2.

Total Number of packages	5
Total number of classes	40
Total number of files	432
Total number of lines of code	5769

15.1 References

- [1] API Gateway pattern, 2018, *Microservices.io*. Viewed 20 November 2019
<<https://microservices.io/patterns/apigateway.html/>>
- [2] API gateways in microservices, 2018, *Smartbear.com*, Viewed 15 November 2019
<<https://smartbear.com/learn/api-design/api-gateways-in-microservices/>>
- [3] Microservices, 2019, *Microsoft*, Viewed 15 November 2019
<<https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices/>>
- [4] What is multi-tenant architecture, 2019, *Datamation*, Viewed 18 November 2019
<<https://www.datamation.com/cloud-computing/what-is-multi-tenant-architecture.htm/>>
- [5] Design Patterns, 2019, *refactoring.guru*, Viewed 10 November, 2019
<<https://refactoring.guru/design-patterns/>>
- [6] Factory design pattern in ruby, 2016, *jyaasa*, Viewed 5 November, 2019
<<http://jyaasa.com/blog/factory-design-pattern-in-ruby/>>
- [7] Design patterns in ruby, 2016, *bogdanvliv.com*, Viewed 17 November 2019,
<<https://bogdanvliv.com/posts/ruby/patterns/design-patterns-in-ruby.html/>>
- [8] PostgreSQL, Viewed 16 November, <<https://www.postgresql.org/about/>>