CSE 546 - Project-2 Report

Monil Nisar
Samip Thakkar
Siddharth Sheladiya

1. Introduction

The greatest effect of the internet is the establishment of connections among people around the world. There might be times when a person gets an excited idea about building an innovative product. But the idea does not get implemented as per the plan as the project might need a team of members with different skills. As a result, many ideas are left undeveloped due to lack of financial or technical support. The problems may arise due to unavailability of resources, insufficient contacts to make the idea possible, lack of skills to implement the idea or lack of funds. We try to bridge this gap from idea to implementation by building a web based application named “Pitcher”. “Pitcher” provides a platform for pitchers, contributors and investors to connect over a single project and implement it. A pitcher pitches an idea with a video and description, and the investors and contributors can join the project team. The students and professionals who want to contribute to the projects and expand their skills. Investors can select the project he is interested to fund for. They can select the projects as per their interests and chat with the pitcher and form a team. Thus, Pitchers is a cloud-based web application that connects pitchers, contributors and the investors to a single platform.

2. Background

With the advent of cloud technologies, creating and scaling applications have been much easier. Developers do not have to worry about the hardware and network related specifications, rather they can focus more on the core logic of their application. Specifically PaaS(Platform as a Service) services provided by the cloud providers are in much use due to its ease in use and efficiency. PaaS services tend to scale automatically according to the traffic flow on the application, which makes the application elastic without the developers being bothered. We are using Google Cloud Platform for our application where our main web application is deployed on the Google App Engine(GAE) which is Google's scalable PaaS service for hosting web applications. The traffic on the application is unpredictable, hence it is very important to scale the application if the traffic increases which in our case is taken care of by the Google App Engine. Along with that, we are using Google Firebase's Realtime Database to support the CRUD operations of the application. Realtime Database captures the updates on data in real time which becomes very useful in our case when different users want to interact in real time which would be tough to implement using the traditional database due to its latency. There are certain event triggered scripts which the application requires to run when certain changes happen in the database. These scripts are deployed on separate cloud functions which continuously checks for the event to trigger and perform the assigned functions accordingly. These serverless scripts come in handy to perform independent functions.

2.1 Why this problem is important

- It often happens that someone has a great idea for an innovative project or potential skills, but they can't complete that project all by themselves. Students, in particular, face this problem more commonly, as they constantly look to work on some projects to test and expand their skill set. On the other end, there are people struggling to pitch their ideas as most of the available platform only features your idea if either you have a prototype, or are experienced enough.
- The side-projects are the best way to learn new skills for the students or someone who wants to learn new skills. It will help them expand the knowledge and give practical experience.
- Many of the great ideas are buried because they don't get proper reach, either financially or to skillful people to implement the project.

- The problems introduced to investors are mainly developed to some level, such that sometimes, investors can not ask for fundamental changes and as a result, he has less options to invest.
- The main problem is that currently, there is no platform available where the pitcher, investor and contributors can form a team right at the start of the project and work from scratch. Pitcher tries to bridge this gap.

2.2 How our solution is different than current solutions

One of the similar applications available is globalpitch.com. It allows you to sign-up as a startup, where you will pitch your idea online to the investors, incubators, media and corporates. You need to apply once, and your data will be visible to all. The other sign-up method is to register as an investor/ corporate to fund the idea. However, this website focuses more on the competitions between start-ups rather than creating a single community. There is no facility to sign up as a contributor and work on their idea.

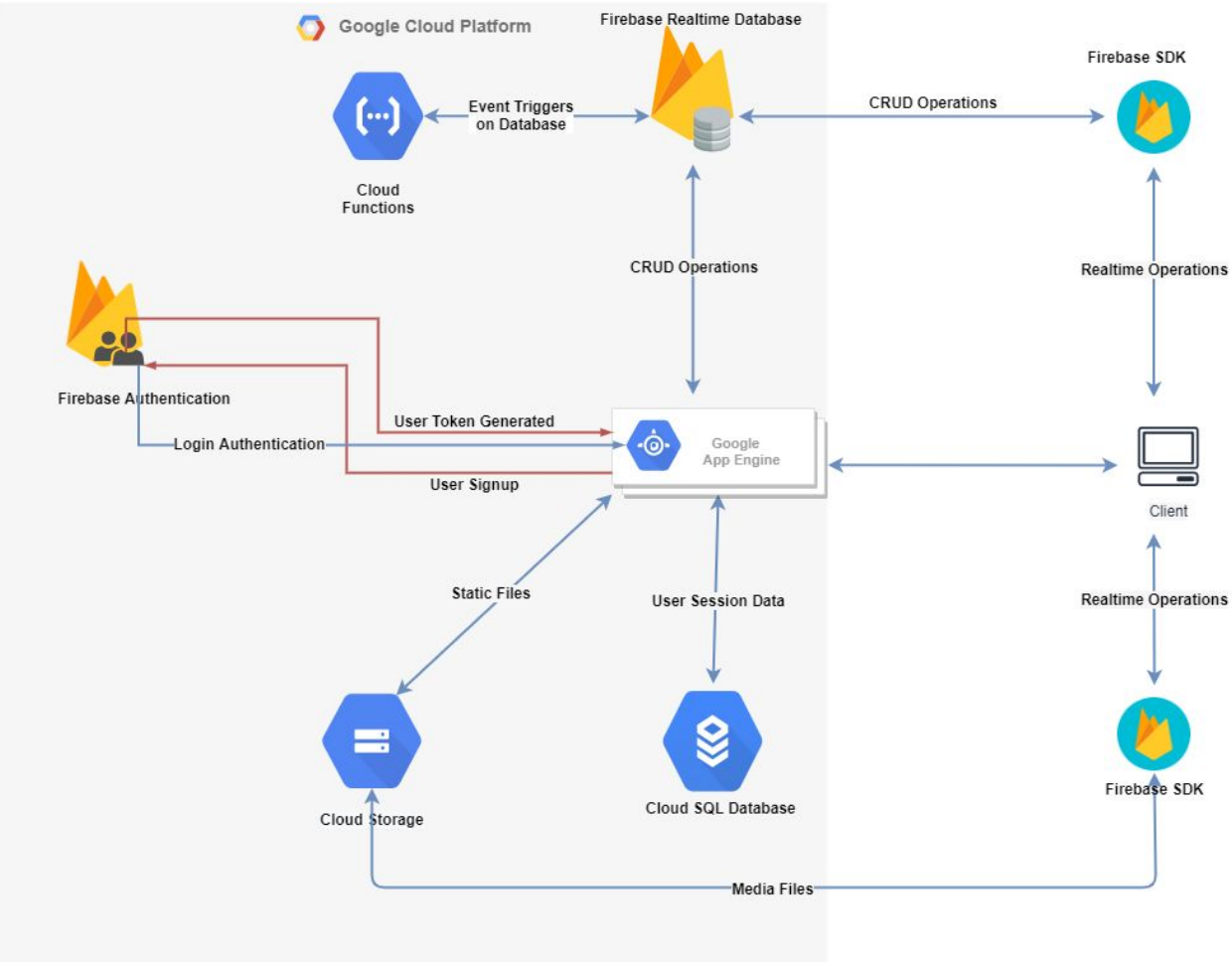
Fiverr.com, upwork.com and aquent and many other websites provide freelancing, but the method is they will set you up and you need work experience to get selected. They are not completely open-source platforms where any student can join and communicate with someone who has an idea but not all the skills to complete the project. The main feature of almost all such websites is they serve like a job board. There is not much room for someone who wants to expand the skills or side-work on a project.

However, Pitcher is an open portal, where anyone can submit their idea and contributors/ investors can connect and discuss through chat rooms. Anyone who has an idea, who wants to contribute to an open project and who wants to take the idea to commercial level can join and use the platform.

3. Design and Implementation

We have designed a web application using the Django framework, hosted on Google App Engine with Firebase Realtime Database as the database and cloud storage as a bucket to store media. There are three different roles a user can have: pitcher, investor and contributor. Pitcher delivers the idea in form of textual description and a video, a pitch. The investor and contributor can view the pitch and can show interest in it. On showing interest, a chat room will be created where they can communicate and discuss the idea. We have also kept event listeners that perform specific tasks, like notifying the pitcher when the pitch he/she posted seems plagiarized or notifying the investor whenever a pitcher posts a new pitch or deletes one.

3.1 Architecture



figure(3.1.1) - Component Level Architecture

The Architecture is made up of six main components. These six components share the work by taking up different aspects of the application. The application is hosted on the Google App Engine which handles the incoming traffic by balancing the requests among the instances. The user first makes the request to the website and the home page is loaded. All the static pages are stored in the cloud storage bucket which is fetched by the application when a request is made. The user signs up by providing credentials in the sign up page. From this data, the credentials(i.e email address and password) are transported to the firebase authentication system using the firebase SDK. The application uses pyrebase, which is a wrapup around the python's firebase SDK for simpler implementation. After being transmitted to the authentication system, firebase stores the credentials of the user and creates a unique user key i.e User UID for every user. After storing the credentials, the application fetches this unique user UID from firebase and stores the other user data such as firstname, last name in the firebase realtime database. This data is stored as a key value pair where the key is each user's unique user UID. Thus each user's data is mapped with the user UID which makes other database operations much easier to perform. A Cloud Function which is running independently, is triggered when a new user is signed up and sends a welcome mail to the user. Later the user can log in with credentials he provided during the sign-up process.

After the user is logged in, the user can perform CRUD operations on the realtime database according to the role. The user session is created in the Cloud SQL Database when a user logs in and the session is deleted every time after the user is logged out. A Pitcher uploads a new pitch which contains a title, a description and a short video of the pitch. This data is sent to the real time database using firebase's SDK. This event again triggers a cloud function which computes the pitch's similarity with other pitches and notifies the user accordingly. The cloud function also notifies all the investors when a new pitch is available using an email. As it is inefficient to store the video in the realtime database, the video is stored in the cloud storage and its link is attached to the data stored in the database to synchronize the data while reading. The reading operation is handled in a similar way. Based on the query and the role of the user, the data is displayed using the SDK to the users from the database. Thus the firebase realtime database handles all the CRUD operations using the user's UID generated by the Firebase authentication system and the firebase SDK.

The investors and contributors can show interest in the pitch by a toggle event. When an interest is shown, a chatroom is created between the pitcher and the interested

user (i.e an investor or contributor). The users can discuss the pitch in the chatroom in real time. The chatroom is implemented by firebase realtime datastore and javascript. Both the user's messages are stored in the database under a unique chatroom ID. The messages are fetched in real time by the client side javascript code which deals with sending and receiving the messages in the database. Javascript is used for this operation to reduce the end-to-end message latency and provide the live updates to both the parties.

- **Google App Engine**
  The main web application is deployed on the Google App Engine(GAE). It is a PaaS service developed by google to deploy applications without bearing any trouble of hardware specifications. Google App Engine automatically handles the traffic flow on the application by deploying more instances for support. So it can scale up and down without any intervention of the developers, that is the reason we decided to use this service as our deployment platform. Another benefit of using GAE is that it automatically balances the among the instances and can handle a bundle of web requests without any flow and developers does not have to handle it programmatically.

- **Firebase Authentication**
  The application uses Firebase Authentication System rather than the traditional authentication of Django. Firebase provides the whole backend authentication service which can be implemented by the firebase SDK. The user can first provide the credentials to the application which is passed to the firebase authentication system using the firebase SDK where its authentication system verifies the user credentials and grants the user access to the data and any related firebase service with which the user is attached to. Our application uses email and password as the valid credentials. Firebase assigns a unique 28 digit user key to each user, which the application uses to map user data with the user key. Thus each user is restricted to only a certain level of data read and write permissions with the help of this unique key.

- **Google Cloud Storage**
  Google App Engine uses the gunicorn server for hosting the django applications on the cloud. Gunicorn does not support the static files, that is why we use Google Cloud Storage to store the static files of the application[1].GAE handles the application while all the static files of the application are stored in the Google DataStore bucket. The application is synced with the google datastore to fetch the static files whenever requests are made by the user. Thus all the frontend html, css

and javascript files are stored in the bucket. We specify the storage bucket path in the static path of settings.py file of the application.

- **Google Cloud SQL**
  Google Cloud SQL Database is used by the application to store the user session details. Whenever a user logs in, a session is created which is stored in the database. All the CRUD operations the user performs is done under this session to map the data belonging to the user in the realtime database.

- **Firebase Realtime Database**
  The application being dynamic, the user has rights to perform CRUD(Create, Read, Update, Delete) operations upto certain level according to their role. There are certain operations in the applications where these transactions must be realtime to provide the users with minimum latency such as live chat. That is where the Google Firebase's realtime database comes into picture. The application uses real time datastore to cater all these transaction needs of the application. Firebase's realtime datastore is an unstructured database where every entry is stored in the form of a key-value pair, thus providing better flexibility and performance in comparison to traditional structured databases.

- **Google Functions**
  Google cloud functions is a serverless hosting platform where the event triggered scripts are deployed. These functions run independently from the main application server-lessly and are called when a certain specified event is activated. We have used the cloud functions to implement multiple functionalities in the application. The cloud functions are waiting for the event at the realtime database and firebase authentication. One function is triggered when a new user is signed up using the firebase authentication and sends a personalised welcome email at the user's email address. Similarly, another function is triggered when a user deletes the account and sends a goodbye mail at the email address. The third cloud function is triggered when a user uploads a new pitch. The function eventually captures the pitch description and compares with the other pitches in the database to check whether it is similar to any of the already present pitches. The similarity is calculated by tokenizing the description and then calculating the cosine similarity with each of the present pitches. The pitcher is notified via an email if the similarity score is above a certain threshold, which in our case is kept at **0.50**.

- **Autoscaling**

  Autoscaling is an important aspect to keep in mind while designing the architecture of the application. Inefficient auto scaling method of an application can result in crashing the application when there is an increase in traffic within a short period of time and the traditional system cannot handle or balance the traffic, hence resulting in crashing the system. As our application is hosted on a public domain, forecasting the traffic on the application is impossible and hence efficient auto scaling architecture is needed to cater the incoming requests and balance the load. This process is handled by the Google App Engine in our case. As GAE provides a platform to host our application, the instance management is handled by GAE itself. The Google App Engine automatically launches and shuts the instance when there is a fluctuation in traffic.[2] This whole process is done automatically hence at any time the application can be running on multiple instances and can be scaled according to the incoming requests.

3.2 Django Framework Architecture

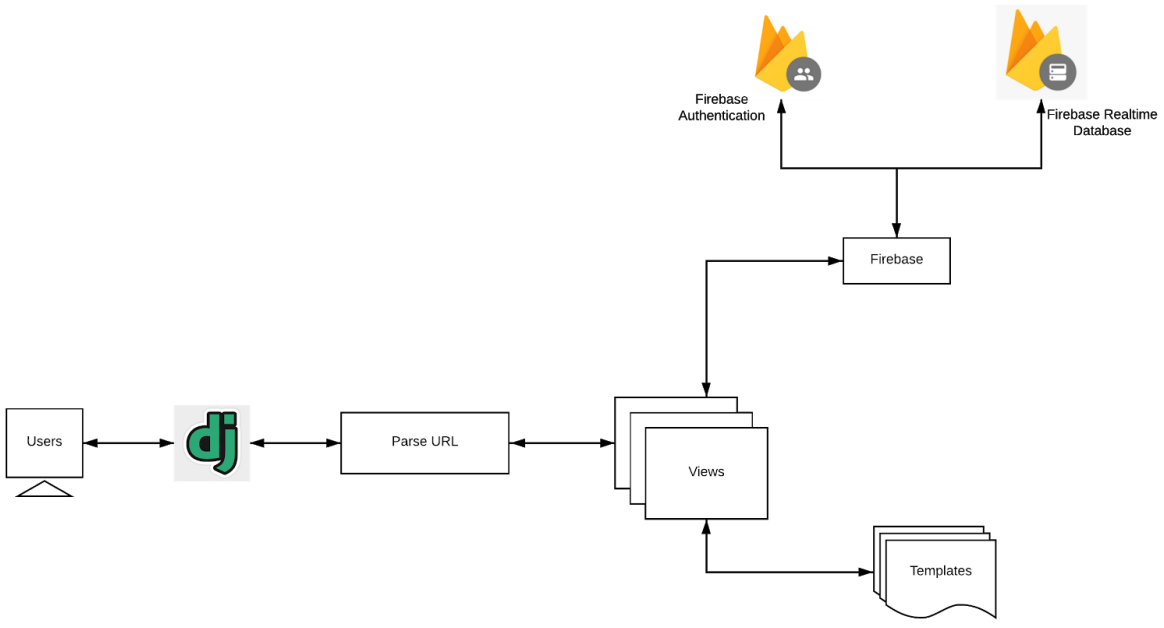


Figure (3.1.2). Django Framework Architecture

We are using Python's Django framework to build our application. Django follows a MVT(Model-View-Template) architecture where Model part deals with the database operations for the operation, View deals with the backlogic of the application and Template deals with rendering the data into html files to provide user responses. Here, as we are dealing with the unstructured database, we have tweaked the model part of the architecture, and instead the view parts handle the database operations using the firebase's SDK. So when the request is made by the user, the urls.py file redirects the request to the view part which decides how to deal with the request according to its type i.e, if it requires an operation related to the database then firebase queries are fired and data id provided to the user by rendering the data using the assigned template.

3.3 Proposed Solution

Our application aims to provide a platform where individuals with certain business ideas can propose those ideas and people could invest or contribute in development of those ideas. We have three different roles: pitcher, investor and contributor. The pitcher is the one who presents the idea, with a description of idea and a short video(pitch). The investors show interest in the pitch and could talk with the pitcher to invest. Contributors are the technical people, who would like to play a role in development of the project. They too show interest and can communicate with the pitcher for participation.

Our solution differs from the state-of-the-art in many ways. Our application uses many cloud services to accomplish different tasks. We used Google App Engine to host the application, Firebase Realtime Database as our database, Cloud Storage to store extra files, Cloud Functions for handling events and Firebase authentication for user management. Our solution has many functionalities, whenever a new pitch is presented, it extracts information from it and notifies the investors about the new pitch. Another function is the chat. It is done using chat rooms which are updated in real-time using Firebase Realtime Database. We have used pyrebase, a wrapper for firebase in python, to query the database. For some real time updates directly from the client, we used firebase

JavaScript SDK to modify the database. To maintain the server's internal information, cloud SQL database is used which is tightly coupled with the django server.

So all these functionalities help the pitcher to showcase the idea, get people to invest in the idea and also connect with skilled professionals who can work for development of the idea. The whole experience of users is improved with the help of supplementing functions like chat and notification.

4. Testing and Evaluation

The testing process of the application was done in multiple phases which covered different components of the architecture. This component level testing started with testing the Google App Engine. As the autoscaling for this application is handled by the Google App Engine automatically, we carried out stress testing on the instances and recorded latency and number of instances launched to handle the requests. Google Api was used to monitor the number of instances in use along with the App Engine Dashboard as it displayed the stats in a more detailed manner. Initially, no instances were created and the instances list fetched via Google Api reported 0 listed items.

```
(base) C:\Users\siddh\Desktop\deployement\Pitcher>gcloud app instances list -v 20200510t015626
Listed 0 items.
```

We later carried out multiple testings using a hardcoded selenium script which fired multiple requests simultaneously using multi-threading. First test script made 31 requests which were completed by the application in around 20 seconds apart from the initial boot-up time. Total 3 instances were launched by the Google App Engine where the first instance handled 24 requests and other two handled very few requests.

00c61b117c29d6670933bd9e86e7c6fe0d439dd6558d33cd979ab34575e005aab416ea	0.4	1,124 ms	24	0	180.5 MB	May 10, 2020, 2:00:26 AM	Dynamic
00c61b117c255f7765c83d4b5d7a063e104255a4a5b840bccb03a5b2be209d04f47c80	0.067	4,554 ms	4	0	174.5 MB	May 10, 2020, 2:00:30 AM	Dynamic
00c61b117c976a15421a9c7726e7a7980bf872c204eb1930c62981ed8890c398a3acbb	0.053	3,414 ms	3	0	150.1 MB	May 10, 2020, 2:00:35 AM	Dynamic

Figure(4.1) - Instance list for 31 requests

Later we tested the App Engine by deploying the test scripts on multiple VMs which were synchronized to fire the requests simultaneously to the application. At the end, the application handled around 1,036 requests in approximately 210 seconds and a total 3 instances were used by the Google App Engine. When looking at the utilization charts

from the App Engine Dashboard it was inferred that a new instance was launched if the average CPU utilization of the current instances increased above 50%.

Apart from the autoscaling part the application was tested for all links and redirections which were included in the project. Loading and latency of all the static files of the application from Google Cloud Storage was tested, making sure that all the paths are provided accurately. All the CRUD operations were tested making sure that all the transactions are reliable and are performed in real time. The media quality was tested for any quality degradation or discrepancies.

5. Code

5.1 Functionality of every program

- **app.yaml**
  - The configuration of the application in the App engine is done by this app.yaml file. It contains all the general settings, URLs and versions for the application deployment. It specifies the runtime environment for the application to run which in our case is python 3.7.
- **urls.py**
  - This python file is used to redirect the user's request to a specific view. It is also used to switch between webapps. It points to the view of that webapp and loads up a particular function.
- **admin.py**
  - This file is used for administrative purposes. Its main purpose is to register models that are used within applications. The url ending with “/admin” points to this file, but for security reasons, it is not always used in this way.
- **main.py**
  - This python file points the application in pithcher_app/wsgi.py is the app that is to be deployed. The App Engine looks for this file first while running the application.
- **wsgi.py**

    - When the WSGI server loads your application, Django needs to import the settings module — that's where your entire application is defined.[7]
- **manage.py**
    - It is used to provide command line utility and you can interact with your django project using this utility. The file point to the application settings related to the application is mentioned. Manage.py is created automatically when you create your django project.
- **settings.py**
    - The settings.py file is the core of the entire application as it binds all the configuration of the application. It contains important information related to which database it used along with its username and password, the path to the static files of the application and other important API configuration used in the application. It also has information about the different apps created within the project which helps django while hosting the application.

- **Templates**
    - The templates folder in application/templates contains the front-end html files of the project. The files are separated folder wise according to each django component. The Application has 4 components and there are certain files in these components which perform the same task but must be replicated in each component. They are:
        - **base.html**: It contains the general front-end layout which has to be common for all the pages like header, footer and title bar. The base.html is imported in the required pages of the app.
        - **dashboard.html**: The function of this html file is to display pitches from the views.py file when a user logs in according to the role of the user. This file is present in templates/pitchers, templates/contributors and templates/investors where the pitches have to be displayed.
        - **chat_window.html:** The chat_window.html is used to provide the chat box layout to the users. It is present in all the templates folder except for the users/. It renders the incoming messages and displays the sent messages for either pitcher-to-investor or pitcher-to-contributor conversations.
        - **users/home.html**: It is the first page that loads up when the URL is hit. It displays the title bar which contains several options like login, sign-up and home, and clicking those buttons redirects the users to other pages accordingly.

        - **users/login.html**: It renders a page to take login credentials and role from the user for the authentication purpose. The user credentials are sent to the backend python code via “post” method for further processing. The page also provides a forgot password button which redirects users to reset_password.html for resetting the user password.
        - **users/reset_password.html**: This page is used to take user email ID to reset the password. The user is redirected to another page to reset the password using Firebase authentication system.
        - **users/signup.html**: This page is rendered when the user selects the sign-up option on the top menu bar. The user is asked to fill the details like first name, last name, role, email and password, and all these details are sent to back-end python code to store the details in Firebase Database using the Firebase SDK. The page also imports a JavaScript file using which the current form is validated for any blank entry before submission.

- **Backend Django Files**

    Just like how templates are divided according to the Components, top back in Python files are also divided in a similar manner. Each component has its own set of python files. The control is redirected from one component to another using the urls.py file.

    - **urls.py:** Each application module in the backend has a urls.py file which contains a list of URL patterns. Each element in the list is a mapping that maps each action in the front-end html pages to its respective backend methods in views.py file. Thus the controllers know where to transfer the data on the basis of this urls.py file.

    - **apps.py**: Each application module has an apps.py file which defines a config class inherited from django.apps’ AppConfig class. This file basically registers each module to the main settings.py file so that django knows to include this application every time a server is run.

    - **users/views.py**
        - **home:** This function renders the landing page for the website. Its main purpose is to present the user with information about the app and options for login and signup.

- **login:** This function renders the login form and also it does the user verification. It uses the firebase authentication system to do it.
- **signup:** This function renders the signup form to the user and also collects the user data to sign up new users. It uses firebase's authentication system to register users.
- **reset_password:** This function deals with password reset form. In case the user forgets the password or wants to change it, this form is used.

- **pitcher/functs.py**
  - **summarizer:** This function extracts the summary of the description given by the pitcher in the pitch. It depends on nltk to perform its sentence tokenization and term frequency. It does sentence tokenization and then by using TF-IDF (term frequency–inverse document frequency) generate a summary.

- **pitcher/views.py**
  - **dashboard**: This function deals with the pitcher's dashboard. It is like the homepage for a pitcher. It shows the current pitches, the chat box with active chats and many other options. It uses pyrebase, which is a wrapper of firebase SDK for python. The code is mainly divided into two sections, one to get the current chatrooms where the pitcher is present and another is to get all the pitches that he/she presented. The application uses pyrebase library to perform the firebase related operations in python. All the data related to the current user is fetched using pyrebase's **get()** method. Queries are made to fetch related pitches and chatrooms of the user from the database. This data is filtered using the user's unique UID key generated during the signup process.

  - **new_pitch:** This function allows the pitcher to add a new pitch. It either renders the HTML for new pitch or it gets data from HTML and puts everything into the database.The data is fetched from the html template using the **request** object via its **get()** method. Pyrebases's **push()** method is used to store the data in the database in key-value form using a unique automatically generated key.

- **edit_pitch:** This function lets the user edit the pitch. It uses the pyrebase wrapper to get the information from current pitch so that it would be easy for pitchers to edit it. The updates are made to the database using the pyrebase's **update()** method in a similar manner to that of push and get operations.

- **delete_pitch:** This function deals with removing the pitch details and its references within the database. It takes the pitch_id as reference and deletes the pitch and its references within the database. It also makes sure that the database remains in a consistent state.

- **chat_window:** It renders the HTML for the chat room page. It gets all chat rooms for that pitcher, and displays it. This HTML page uses Javascript to get live data from Firebase Realtime Database.

- **logout:** It is for server side processing to make sure that the pitcher is logged out and server knows about it.

- **delete_account:** This function is used for removing the pitcher's account from the database. For deleting a pitcher, all the links related to pitchers in investors and contributors have to be deleted. Entris of the given pitcher is deleted from the chatrooms, pitches, investors, contributors and finally pitchers. All these operations are done using pyrebase's **remove()** method. At the end, the user id was deleted from the authentication system using **remove_user_account()**.

- **contributor/views.py**
  - **dashboard**: This function deals with the contributor's dashboard. It is like the homepage for the contributor. It shows all the new pitches by all pitchers, the chat box with active chats and other options. It uses pyrebase, which is a wrapper of firebase for python. The code is mainly divided into two sections, one to get the current chatrooms where the contributor has shown their interest and another is to get all the pitches. Similar to the pitcher's function, here also the **get()** method from pyrebase gets all the pitches from all the pitchers. Same is for the chat. It looks into all

chat rooms and finds related ones by comparing **chat_id** with the ID of the current contributor.

- **Current_project:** This function works very similar to the dashboard, but it filters out only the pitches the contributor is interested in. There is a chat rendering same as the one for dashboard.
- **chat_window:** It renders the HTML for the chat room page. It gets all chat rooms for that pitcher, and displays it. This HTML page uses Javascript to get live data from Firebase Realtime Database.
- **logout:** It is for server side processing to make sure that the contributor is logged out and server knows about it.
- **delete_account:** This function is used for removing the contributor's account from the database. It removes all the chat rooms and linked contributors and investors.

- **investor/view.py**
  - **dashboard**: This function deals with the investor's dashboard. It is like the homepage for the investor. It shows all the new pitches by all pitchers, the chat box with active chats and other options. It uses pyrebase, which is a wrapper of firebase for python. The code is mainly divided into two sections, one to get the current chatrooms where the investor has shown their interest and another is to get all the pitches. It is very similar to a pitcher's dashboard. It uses pyrebase's **get()** method to fetch all the data. Same goes for the chats. It filters the chat rooms with the **investor_Id**.
  - **Current_project:** It shows all the projects that the investor has shown interest. The chat window on right shows the current chat rooms.
  - **chat_window:** It renders the HTML for the chat room page. It gets all chat rooms for that pitcher, and displays it. This HTML page uses Javascript to get live data from Firebase Realtime Database.
  - **logout:** It is for server side processing to make sure that the investor is logged out and server knows about it.
  - **delete_account:** This function is used for removing the investor's account from the database. It removes all the chat rooms and linked contributors and investors.

- **Google Cloud Functions**
  - **funtion_1.py**
    - This function sends a welcome mail to a new user via python's smtplib library. The **hello_auth()** function is called when an event is triggered at the firebase realtime database. This function takes two parameters (event, context) where event is the event payload that is generated while triggered and context is the metadata for that event. This function calls another function **send_emails()** which uses smtplib library to send the email. It first creates a server at a defined port and then logs in the user's credentials provided by the developer to send the mail. The email's body and subject is predefined and the email id of the receiver is retrieved from the event variable.

  - **function_2.py**
    - This function sends a goodbye mail to the user who is deleting the account. This function performs in the same manner as that of the functions_1, just the message body and subject is changed in the code base while trigger is changed using the Google Cloud Console.

  - **function_3.py**
    - This function notifies all the investors when a new pitch is available. This is done by setting the event on the firebase realtime database and the event is triggered when a new pitch is added to the database. The email ids of all the investors are retrieved using the **get()** method of the pyrebase library, from the firebase database. The emails are sent using the smtplib library like function_1 and functtion_2.

  - **function_4.py**
    - This function is used to find similarity of a new pitch from the already existing pitches in the database. This is calculated using the Cosine Similarity. The **tokenize()** function takes the description and first tokenizes the description then converts all the words in the lower case, then removes all the stopwords in the present in the english language and then returns each token count. Python's **nltk** library is used for this function.

        - The **calculateSimilarity()** calculated the similarity between two documents by converting the word dictionaries into vectors and the calling **cosine_sim()** function which returns the cosine similarity between two vectors using numpy modules.
        - The description of all pitches is fetched using the pyrebase library's **get()** method. The description of current pitch is fetched from the event variable and is compared with all the fetched descriptions. If the similarity is greater than 0.50 then a mail is sent to the pitcher regarding this issue using the smtplib library.
- **Testing script**
    - **testing.py**: the testing script creates multiple threads, where each thread has a flow to run. First, it will create a browser element and send a request to access our home page. It will then wait for 5 seconds and then try to login and wait for another 5 seconds. Then it would send dummy refresh requests twice every 5 seconds and then logout and close the browser. This will create 5 requests per thread. We can modify the number of threads to scale the testing. We ran it on multiple machines to generate a tremendous amount of traffic.

5.2 Steps to install and run the application

- Create a project on **Google Cloud Platform** with project_id for your choice.
- Create a **Google Cloud SQL** instance under this project.
- Create a project in **Firebase** under this project.
- Under the Firebase Authentication, **enable** the sign-in providers status for **Email/Password**.
- Create a bucket in **Cloud Storage**.
- Replace all firebase configurations from the application to your project's configurations.
- Change the **Firebase Realtime database rules** as follows.

```
{
  "rules": {
    ".read": true,
    ".write": true
  }
}
```

Fig(5.2.1). Firebase realtime database rules

- Change the **Firebase Storage rules** as follows.

```
rules_version = '2';
service firebase.storage {
  match /b/{bucket}/o {
    match /{allPaths=**} {
      allow read, write;
    }
  }
}
```

Fig(5.2.2). Firebase storage rules

- Download the **Google Cloud SDK**.

- Create a **Cloud Function** from the gcloud console with the following parameters.
    - Name: <you_choice>
    - Memory_allocated: 128MiB
    - Trigger: **Firebase Authentication**
    - Event-Type: **Create**
    - Runtime: Python
    - Copy the contents of **functions/function1.py** into the code panel.

- Create a **Cloud Function** from the gcloud console with the following parameters.
    - Name: <you_choice>
    - Memory_allocated: 128MiB
    - Trigger: **Firebase Authentication**
    - Event-Type: **Delete**
    - Runtime: Python
    - Copy the contents of **functions/function2.py** into the code panel.

- Create a **Cloud Function** from the gcloud console with the following parameters.
    - Name: <you_choice>
    - Memory_allocated: 256MiB
    - Trigger: **Firebase Realtime Database**
    - Event_Type: **Write**
    - Database: <Your_realtime_database_name>
    - Path: **user/pitches**
    - Runtime: Python
    - Copy the contents of **functions/function3.py** into the code panel.

- Create a **Cloud Function** from the gcloud console with the following parameters.
    - Name: <you_choice>
    - Memory_allocated: 256MiB
    - Trigger: **Firebase Realtime Database**
    - Event_Type: **Write**
    - Database: <Your_realtime_database_name>
    - Path: **user/pitches**
    - Runtime: Python
    - Copy the contents of **functions/function4.py** into the code panel.

- Install Python 3.7 and libraries mentioned in requirements.txt.
  $:~ pip install requirements.txt

- Copy the contents of the **/application/** to a folder of your choice.
- Download the **Google Cloud SQL proxy** for migrating the local django database.
- Run the following code in Google Cloud SDK Shell and fetch the **Connection_Name**:
  $:~ gcloud sql instance describe [YOUR_INSTANCE_NAME]

- Initialize the Google Cloud Proxy using the **Connection_Name** as follows:

```
cloud_sql_proxy.exe-instances="[YOUR_INSTANCE_CONNECTION_NAME]"=tcp:5432
```

- Change the **settings.py** in the application folder and replace the DATABASES section as below:

```
DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.mysql',
            'HOST': '/cloudsql/[your_connection_name]',
            'USER': '[user_name]',
            'PASSWORD': '[your_password]',
            'NAME': '[your_database_name]',
    }
}
```

- Run the following commands to migrate the django database to the Cloud SQL Instance.
  application\$:~ python manage.py makemigrations
  application\$:~ python manage.py makemigrations [application_name]
  application\$:~ python manage.py migrate

- Collect the static content of the application using the below command.
  application\$:~ **python manage.py collectstatic**

- Set the cloud storage bucket read operation as public.
  $:~ gsutil defacl set public-read gs://<your-gcs-bucket>

- Uploading the static content to the bucket. Write the below line in google cloud SDK command line.
  application\$:~ gsutil rsync -R static/ gs://<your-gcs-bucket>/static

- Change the **STATIC_URL** in settings.py in the application folder to the following:
  http://storage.googleapis.com/<your-gcs-bucket>/static

- Deploy the application.
  application\$:~ gcloud app deploy

- Open the Application URL
  application\$:~ gcloud app browse

6. Conclusion

With this project, we created a full fledged multi-user elastic application using a number of cloud services. We understood the importance of designing an efficient architecture. We moved our application from basic non-scalable architecture to an efficient autoscaled architecture using the cloud services. Importance of cloud services, their uses to solve modern world problems were taken into consideration. We learnt and used a wide array of technologies like Django, Firebase Realtime Database, Google App Engine, Firebase Authentication and Cloud Storage. We worked with real time data to provide flawless user experience even with a greater load.

6.1 Future Work

We have some interesting ideas we would like to implement in the application in near future. On the basis of the pitch description, we would recommend the pitcher how to pitch his/her idea to put high impact on the investors. We would try to implement a system to detect the legitimacy of an investor before signing up in the application. Along with this, creating a recommendation engine to suggest pitches to the investors/contributors on the basis of their interests. For boosting the user interface, we would like to migrate the front-end of the application to more flexible technologies like React.js and Angular which would help making the User Interface more interactive. We would like to add a file storage where all the files can be shared. We can process the videos and make the notification system for the chat room.

7. Individual Contribution

7.1 Individual Contribution (Monil Nisar)

- **Design**
  - Design of the overall user interface of the application, transaction of users depending on role and credentials was done by me. The database structure and design was designed by me.
- **Implementation**
  - Implemented the views part for all the roles in the application using page rendering in Django.
  - Merging the code from teammates into the main application and solving many merge errors.
  - Created HTML templates for each user, their dashboard pages, chat rooms and many other pages.
  - The Javascript code for handling events was made by me, along with the CSS required to make the application look aesthetic.
  - Written code for creating the summary from the description of the pitch added by the pitcher.
  - Integrated the chat functionality into the browser to let two users chat. Used Javascript and Firebase realtime database to do it.
- **Testing**
  - A thorough testing of the web app was done manually by me. Making sure all the components work fine and updates/modifications does not make the database inconsistent.
  - Making sure that chat room functionality is working properly and consistency is maintained.
  - Made a script to generate 1000 requests to the web application hosted on the Google App Engine, using selenium. Noted the performance and auto-scaling of instances done by GAE.

7.2 Individual Contribution (Siddharth Sheladiya)

- **Design**
  - Designed the overall component-level architecture of the application which includes all the cloud services.
  - Replaced the traditional django Models with firebase realtime database connectivity to perform all database related operations on the cloud.

- **Implementation**
  - Implemented the backend functionalities of the application firebase connectivity and media connectivity.
  - Shifted the application's authentication system from django to firebase authentication system.
  - Implemented the database connectivity to firebase realtime database using pyrebase library.
  - Migrated the user session data from the local SQL server to Google Cloud SQL instances.
  - Implemented the functionality that lets two users chat in real time, using javascript and firebase realtime database.
  - Deployed multiple cloud functions to send welcome and goodbye mails to the investors.
  - Deployed cloud function to compare the current pitch description to the pitches present in the database using cosine similarity.
  - Deployed the whole django application on the Google App Engine platform including static file synchronization and Cloud SQL data connectivity.

- **Testing**
  - Implemented application testing along with other teammates and evaluated the testing results for autoscaling for over 1000 requests in a short interval.

7.3 Individual Contribution (Samip Thakkar)

- **Design**
  - Contributed in the designing the dashboards of the contributor, investors and pitchers.

- **Implementation**
  - Implemented the application and database connectivity in firebase.
  - Implemented the Machine Learning part of the project like tag generation and phrase generation from the description.
  - Built a recommender system for recommending pitches as per the cosine-similarity.
  - Implemented the independent functionalities of the part of the project like login- logout functions and test them.
  - Integrate different functionalities which were developed and tested by other team mates.
  - Implemented the chat service and dashboard services in Django.
  - Perform error handling for errors during the individual functionality testing.

- **Testing**
  - Implemented application testing along with other teammates and evaluated the testing results for autoscaling for over 1000 requests in a short interval.

8. References

1. Running Django in the App Engine flexible Environment: https://cloud.google.com/python/django/flexible-environment

2. Google Cloud- How instances are managed: https://cloud.google.com/appengine/docs/standard/python/how-instances-are-managed

3. Crowdsourcing: Leveraging Innovation through Online Idea Competitions by Fiona Maria Schweitzer,Walter Buchinger,Oliver Gassmann https://www.tandfonline.com/doi/abs/10.5437/08956308X5503055

4. Teaching the techno-pitch by Charlsye Smith Diaz: https://ieeexplore.ieee.org/abstract/document/5208685

5. Pyrebase:A simple python wrapper for the Firebase API. https://github.com/thisbejim/Pyrebase

6. Firebase Documentation: https://firebase.google.com/docs

7. How to deploy with WSGI: https://docs.djangoproject.com/en/3.0/howto/deployment/wsgi/