West Bengal State University

West Bengal State University
Barasat, N 24 Pgs
लक्ष्यं विश्वमानम्

Railway Reservation System

Project Home Page – [https://apccproject.herokuapp.com]

Deployment – Public

Paper – CMSADSE06P

College – Acharya Prafulla Chandra College

Supervised by Soumi Das Gupta

Submitted by

```
Aniket_Sarkar:
{
        "Roll-No": 6221101-00995,
        "Registration Number": 1011911101046
},
Tusher_Mondal:
{
        "Roll-No": 6221101-00991,
        "Registration Number": 1011911101021
}
```

Status	Complete
Phase	Maintenance

Year - 2022

Acknowledgment

We would like to earnestly acknowledge the sincere efforts and valuable time given by our Project Coordinator (Mrs. Soumi Das Gupta) and respected teacher (Mr. Joydeb Das Biswas). Their valuable guidance, lectures, and feedback have helped us in completing this project. We are also thankful to the authors of the various research contents that helped us proceed through.

Also, We would like to mention the support system and consideration of our parents who have always been there in our life.

Last but not the least, We would like to thank our classmates who have always been there by our side and helped us test this project in different scenarios.

Without them, We could never have completed this Project.

Thanks a lot.

Acharya Prafulla Chandra College

New Barrackpore, Kolkata-700131, West Bengal

Certificate Of Authenticity

This is to certify that the Project entitled **“Railway Reservation System”** is done and maintained by **Aniket Sarkar** and **Tusher Mondal** under the guidance of **Mrs**. **Soumi Das Gupta**, for the partial fulfillment of the degree in Bachelor of Science Honours in Computer Science from **Acharya Prafulla Chandra College** affiliated to **West Bengal State University**.

------------------------   ------------------------

Head of the Department   Project Supervisor

Contents

|| Project Overview ||

Elementary Description

It is a simple software solution with a simple resolution of booking a ticket on a train by providing the required information(i.e Source Station, Destination Station, Preferred Time) and getting back a unique ticket number, by which users can track and manage their reservations.
We tried to compose a system where people can see the timetable by station and view information about booked tickets and available seats.
For the admins there exists a whole another set of functionalities. An admin can manage the whole system, i.e adding a new train to the database or removing an existing train from the database, the same goes for coaches and seats, Checking feeds, collecting ticket information, and rejecting reservations depending on the scenario.

This software provides much more functionality but more on these later.

|| Software Requirement Specification ||

The application should have two different access levels.

- Admin
- User

Admins should have functions like -

- Add a Train
- Remove a Train
- Manage a Train
- Add a Coach
- Remove a Coach
- Manage a Coach
- Add Seat(s)
- Remove Seat(s)
- View booked Ticket(s)
- Cancel a Reservation/Ticket

Users should have functions like -

- View trains by Station
- View available coaches and seats
- Book tickets or make a reservation
- View and track booked seats
- Cancel reservation

Both Admin and Users should have different account portfolios with different levels of system access.

|| Feasibility Study ||

There are some vital points to count while in research if the Project in consideration is feasible or not.

- **Economic** - This is a project to present to our final semester. Economically it will have none to very little offering to the real world. In the process of completion of this project we, the students or the supervisor do not intend to spend any money. We will be spending our time and dedication on this project.
- **Technical** - By the system assessment, we came to conclude that we need the following technical proficiency.
    a) HTML, CSS
    b) Networking
    c) Python Programming
    d) The Database

As responsible students, we will do our best to study every field and within. The department of Computer Science has offered us enough knowledge to assemble such a unique idea and implement it. Furthermore, our honorable supervisor **Ms. Soumi Das Gupta** will help us complete the assessment efficiently.

- **Legal** - As this project's sole purpose is to learn. And by the course of implementation, there is a lot to learn, and new technologies to discover. We will reserve the certificate of authenticity of the project and will not commercially use this software by any means. So there will be little to no legal issues to deal with.

- **Scheduling** - The time allotted for this project is the semester duration. So, as beginner software developers we allot our first one month to system design, two additional months to build and test custom modules, and one month to assemble. Our assumption supports that the system will be online in between 4 to 5 months.
- **Operational** - Although the mass allotted for this system is only two, we can manage to distribute the workload between us as efficiently as we can. While the system is online, apart from the daily maintenance, bug fixes, and redeployment, there is very little to manage. In the meantime, we would be able to implement new features to make our system better and stronger than ever.

|| System Assessment ||

To analyze the primary establishment of the Software, We have to figure out the key points that our system must have. By gathering the system requirement we get that the user needs a UI-like interface to interact with, To process requests a backend service is a must, and To store user and railway data a Database is ideal.

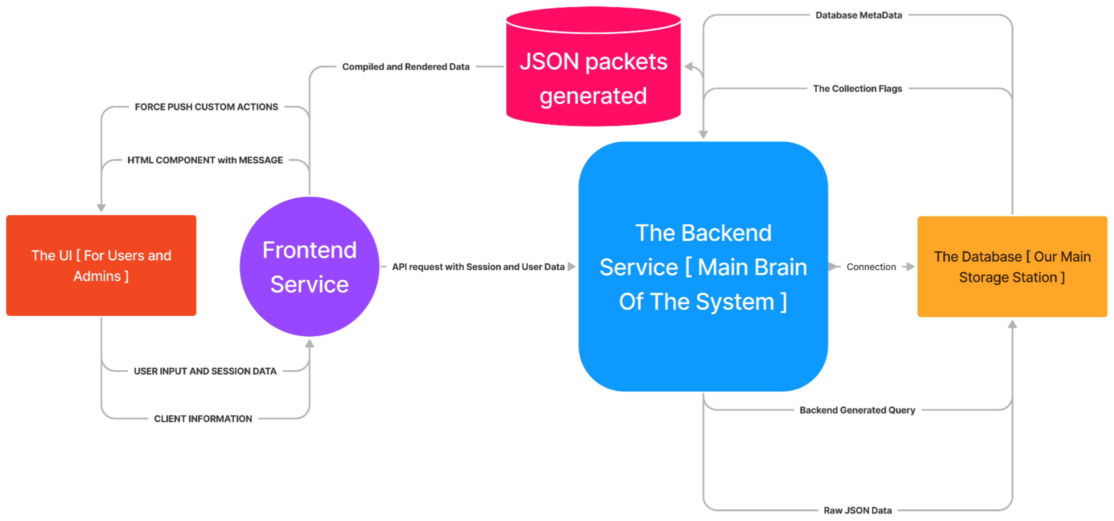
The Project Backbone Elements

As it is a web application. It has mainly three crucial parts of structural design. Those are -

- ➔ The Frontend
- ➔ The Backend
- ➔ The Database

Generalized DFD [Data Flow Diagram]

The simple gateway to user input from server results is much more complicated than that. The following divisions will bring more clarity on how the entire system handles data and delivers results.



JSON packets generated
Database MetaData
Compiled and Rendered Data
The Collection Flags
FORCE PUSH CUSTOM ACTIONS
HTML COMPONENT with MESSAGE
The UI [For Users and Admins]
Frontend Service
API request with Session and User Data
The Backend Service [Main Brain Of The System]
Connection
The Database [Our Main Storage Station]
USER INPUT AND SESSION DATA
CLIENT INFORMATION
Backend Generated Query
Raw JSON Data

Risk assessment

- The project we chose was quite complex but rewarding, Although it was fun to play, along with workflow, a lot of modules and API can be a real hiccup to manage and track sometimes.
- The mass allotted for the project was only two, So the level of responsibility was always higher to count.
- The project is a Non-Profitable asset, We did not spend our money to buy addresses, domains, and system resources. We used free limited resources provided by a third-party company(Heroku) on which our system runs and operates. If the traffic gets higher, there is a high probability that our system will go offline due to the limitation.
- It is an online system, It requires high security and enhanced service delivery. A good security attack or mass request may cause our system to become unstable and unusable for moments.

  But maintenance is a daily routine so new changes are constantly making the system more immune to attacks.

Development approach

There exist a lot of Software development models. But for our System, We chose Agile Software Development.
Because -

- It is a cyclic way to develop and manage a certain software project
- We are continuously taking recommendations from users and making changes to deliver more sustainable performance.
- It became easier to handle the project tasks distributed between the team members.
- During the lifespan of the project development, We continuously integrated new features into the existing ones and we continuously delivered them to the cloud simultaneously, Doing so, our system never became offline, handling jobs that it was meant to handle yet always available to accept newer versions.
- Moreover, We chose Agile because we wanted to make sure customer satisfaction
- It is open to receiving changed requirements.
- Development went easy because the requirements and the design are developed together, rather than separately.

The Software documentation is scattered in three different parts where There is to settle every little sub-project. The partition scheme of the documentation is given below.

- ***The Database***
- ***The Backend***
- ***The Frontend***

|| The Database ||

Description:

A Database is a systematic collection of data. It is a storage station where data is being stored in a particular manner. Data can be accessed, manipulated, and used directly from the database in different applications.

Need for the overall project:

The system we are building is focused on data. User information, Admin information, Rail information, and other additional data are to be stored to be processed and generate valid output for the user.

Data is crucial for the life cycle of the project. To efficiently work with data it needs to be stored in a way so that the developers can access it without any hiccup. That is why we are using a uniform database for our project.

Technology:

There are a lot of database choices available in the market. In the designing phase, we decided to use traditional RDBMS. Later on, in this semester we were introduced to No-SQL and MongoDB in Big Data (A DSE paper of our 6th semester). Distributed systems made us interested to research how to implement this technology. We are using a free python library - “Pymongo” to perform operations on our database as our primary language to build whole backend logic is python.

System Requirements:

The MongoDB database is scattered throughout the cloud-distributed system. The MongoDB Atlas is giving students a free 512MB of Database storage. Our system is not that demanding or heavy. And the maximum utilization is very close to 300MB or so. For the system where the database is alive, we chose the closest Microsoft Azure cloud storage to host our cluster. The replication factor is 3, and the communication channel is shared but the traffic is minimal. Being a sharable cluster it can be slow at times but the efficient system resource utilization of Atlas handles the problem quite efficiently. The system requirements are to be classified as -

Hardware Requirements:

CPU - 800 Mhz x86 CPU or Higher.

RAM - 128 MB of Free Memory

Network Bandwidth - 256 KBPS at least

The host machine should have the aforementioned minimum to smoothly operate on its own.

Software Requirements:

OS - Windows XP SP1 or later

Dependencies - Python[2.7 or latest], Pymongo, bson

Component Design:

Database designing is the most crucial and interesting part for the system to work efficiently.
We created 6 tables or collections for our Railway Database by analyzing the requirements.
For storing Railway information(i.e Train, Coach, Seat) We created two collections -

a) Trains
b) Coaches

For storing level access accounts(Super Admin, Admin, and User), We created three collections -

a) superadmin
b) admin
c) users

For storing Ticket information, We created one more collection -

a) Tickets

Being a No-SQL there will be no joining, no schema, So the whole part of logic programming is mandatory to update every required collection by any particular change.
Furthermore, to maintain consistency, We will not close the connection until all changes have been made properly. The backend programming part will reveal more about database programming.

Collection Design:

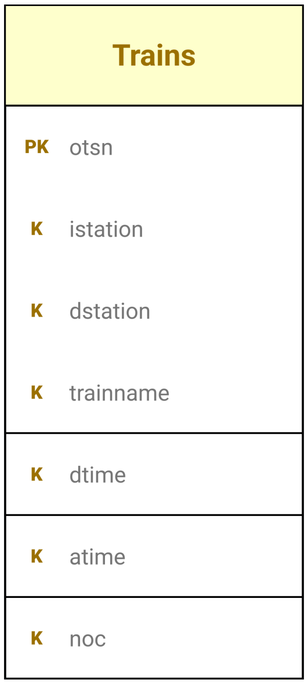
There are a total of 6 collections responsible for holding every data used for our system.

a) Trains: Every document represents individual trains. Each document in this collection has 7 fields. Those fields are -

   i) 'otsn' - Used for storing train serial numbers in incrementing order.
   ii) 'trainname' - Used for storing the name of the train that the admin gives.
   iii) 'istation' - Used for storing the initial station of the train set by admin.
   iv) 'dstation' - Used for storing the destination station of the train set by admin.
   v) 'dtime' - Used for storing the departure time for the train set by admin.
   vi) 'atime' - Used for storing the arrival time for the train set by admin.
   vii) 'noc' - Used for storing the number of coaches on the train.

Example -

```
{
  "_id": {
    "$oid": "[redacted]"
  },
  "otsn": 1,
  "trainname": "BNG TO DD",
  "istation": "BONGAON",
  "dstation": "DUM DUM",
  "dtime": 4,
  "atime": 5,
  "noc": 6
}
```

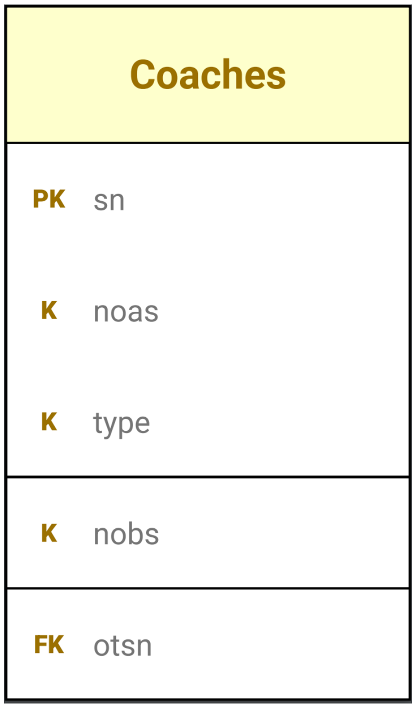


P.S - The "_id" is the primary key of the documents encoded with bson or ObjectID by MongoDB, It is a default feature to create unique documents every time.

b) Coaches: Every document represents individual Coaches. Each document in this collection has 5 fields. Those fields are -
   i) 'sn' - Used for storing serial numbers of coaches in ascending order.
   ii) 'noas' - Used for storing the number of available seats in this coach set by admin.
   iii) 'type' - Used for storing the type of the coach set by the admin.
   iv) 'nobs' - Used for storing the number of booked seats in the coach.
   v) 'otsn' - Used for storing the serial number of the train on which the coach exists.

Example -

```
{
  "_id": {
    "$oid": "[redacted]"
  },
  "sn": 1,
  "noas": 40,
  "type": "NAC",
  "nobs": 0,
  "otsn": 1
}
```

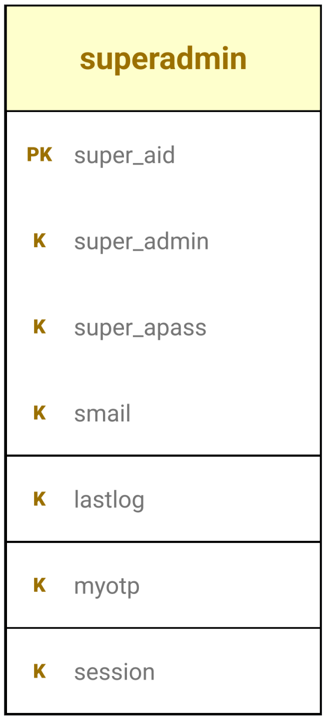


c) superadmin: Every document represents individual Super-Admins. Each document in this collection has 7 fields. Those fields are -

i) 'super_aid' - Used for storing the username of Super Admin set by Database Administrator
ii) 'super_apass' - Used for storing Super Admin password set by Database Administrator.[Encrypted]
iii) 'super_admin' - Used for storing Super Admin name set by Database Administrator
iv) 'lastlog' - Used for storing the last login time of Super Admin
v) 'smail' - Used for storing Super Admin email set by Database Administrator
vi) 'myotp' - Used for storing temporary OTP for authentication
vii) 'session' - Used for storing session variable for authentication

Example -

```
{
  "_id": {
    "$oid": "[redacted]"
  },
  "super_aid": "user@name",
  "super_apass": {
[redacted]},
  "super_admin": "ABC XYZ",
  "lastlog": "2022-06-22 20:46:44.270443+05:30",
  "smail": "apccproject.herokuapp@gmail.com",
  "myotp": "123456",
  "session": "A1B2C3D4A1B2C3D4A1B2"
}
```



d) admin: Every document represents individual Admins. Each document in this collection has 7 fields. Those fields are -

i) ‘aname’ - Used for storing Admin name set by Super-Admin.

ii) ‘aid’ - Used for storing Admin username set by Super-Admin.

iii) ‘apass’ - Used for storing Admin password set by Super-Admin

iv) ‘amail’ - Used for storing Admin email set by Super-Admin

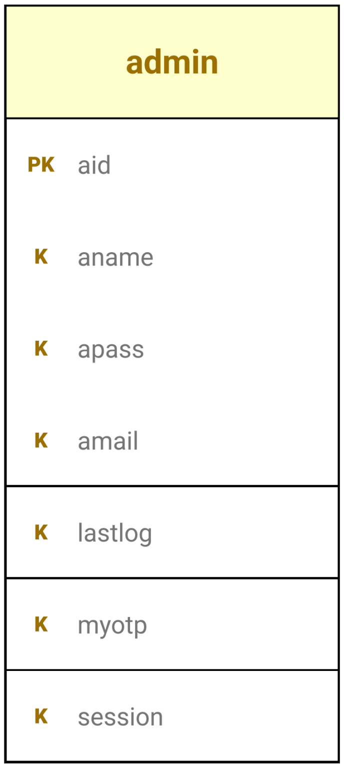
v) ‘lastlog’ - Used for storing the last login time of Admin.

vi) ‘myotp’ - Used for storing temporary OTP for authentication

vii) ‘session’ - Used for storing session variable for authentication

Example -

```
{
  "_id": {
    "$oid": "[redacted]"
  },
  "aname": "ABC XYZ",
  "aid": "user@name",
  "apass": {
    [redacted]
},
  "amail": "apccproject.herokuapp@gmail.com",
  "lastlog": "22/06/2022 20:39:38",
  "myotp": "123456",
  "session": "A1B2C3D4A1B2C3D4A1B2"
}
```



e) users: Every document represents individual Users. Each document in this collection has 7 fields. Those fields are -

i) ‘uname’ - Used for storing User’s name set by them via registration

ii) ‘ph_num’ - Used for storing phone numbers of Users set by them via registration

iii) ‘upass’ - Used for storing User passwords set by them via registration

iv) ‘umail’ - Used for storing the User's email set by them via registration

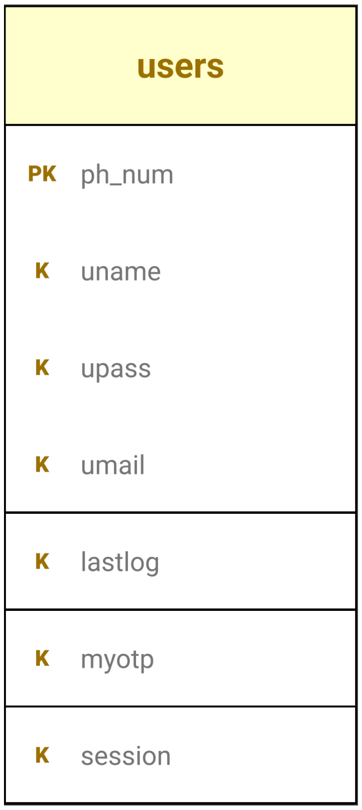
v) ‘lastlog’ - Used for storing the last login information of Users.

vi) ‘myotp’ - Used for storing the temporary OTP for authentication.

vii) ‘session’ - Used for storing the session variable for authentication.

Example -

```
{
  "_id": {
    "$oid": "[redacted]"
  },
  "uname": "ABC XYZ",
  "ph_num": "1234567890",
  "upass": {
    [redacted]},
  "umail": "apccproject.herokuapp@gmail.com",
  "lastlog": "2022-06-22 20:51:48.057123+05:30",
  "myotp": "123456",
  "session": "A1B2C3D4A1B2C3D4A1B2"
}
```



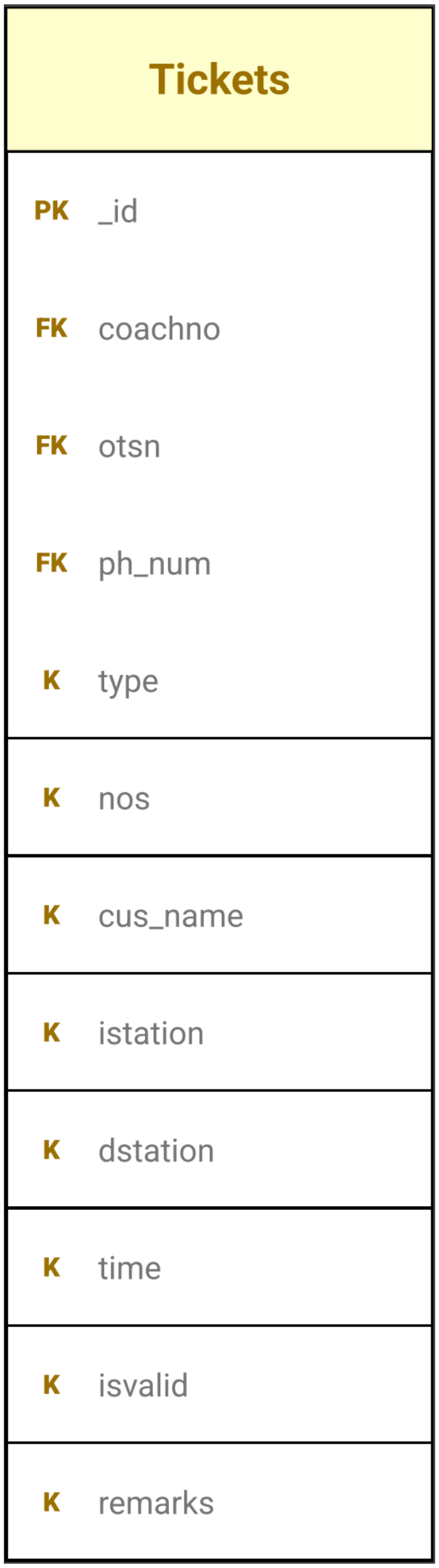
f) Tickets: Every document represents individual Tickets. Each document in this collection has 11 fields. Those fields are -
   i) ‘cus_name’ - Used for storing the customer name
   ii) ‘coachno’ - Used for storing the coach serial number where seat(s) are booked
   iii) ‘type’ - Used for storing the type of the coach
   iv) ‘nos’ - Used for storing the number of seats booked
   v) ‘otsn’ - Used for storing the train serial number where the ticket is booked.
   vi) ‘istation’ - Used for storing the initial station of the train where the ticket is booked.
   vii) ‘dstation’ - Used for storing the destination station of the train where the ticket is booked.
   viii) ‘ph_num’ - Used for storing the user’s phone number who booked the ticket
   ix) ‘time’ - Used for storing the departure time of the train from the initial station
   x) ‘isvalid’ - Used for storing the flag that determines whether the ticket is valid or not
   xi) ‘remarks’ - Used for storing the system message if the ticket is invalid, If the ticket is valid then the remarks field is empty.

Example -

```
{
  "_id": {
    "$oid": "1[redacted]"
  },
  "cus_name": "ABC XYZ",
  "coachno": 19,
  "type": "AC",
  "nos": 2,
  "otsn": 1,
  "istation": "BONGAON",
  "dstation": "DUM DUM",
  "ph_num": {
    "$numberLong": "1234567890"
  },
  "time": "4 O'clock",
  "isvalid": false,
  "remarks": "Cancelled By User."
}

{
  "_id": {
    "$oid": "2[redacted]"
  },
  "cus_name": "ABC XYZ",
  "coachno": 20,
  "type": "AC",
  "nos": 2,
  "otsn": 2,
  "istation": "BONGAON",
  "dstation": "SEALDAH",
  "ph_num": {
    "$numberLong": "1234567890"
  },
  "time": "6.30 O'clock",
  "isvalid": true,
  "remarks": ""
}
```

These examples are not actual data that has been stored in the database



Tickets	
PK	_id
FK	coachno
FK	otsn
FK	ph_num
K	type
K	nos
K	cus_name
K	istation
K	dstation
K	time
K	isvalid
K	remarks

Collection Interrelation:

The interrelated data is everywhere except for the ‘superadmin’ and ‘admin’ collection, Every other collection has interrelated data and interrelation among them. Those interrelations are denoted as -

Ex - collection1[key] == collection2[key]

collection3[key] == collection4[key]

Trains[otsn] == Coaches[otsn]

Trains[istation] == Tickets[istation]

Trains[dstation] == Tickets[dstation]

Trains[dtime] == Tickets[time]

Coaches[sn] == Tickets[coachno]

Coaches[type] == Tickets[type]

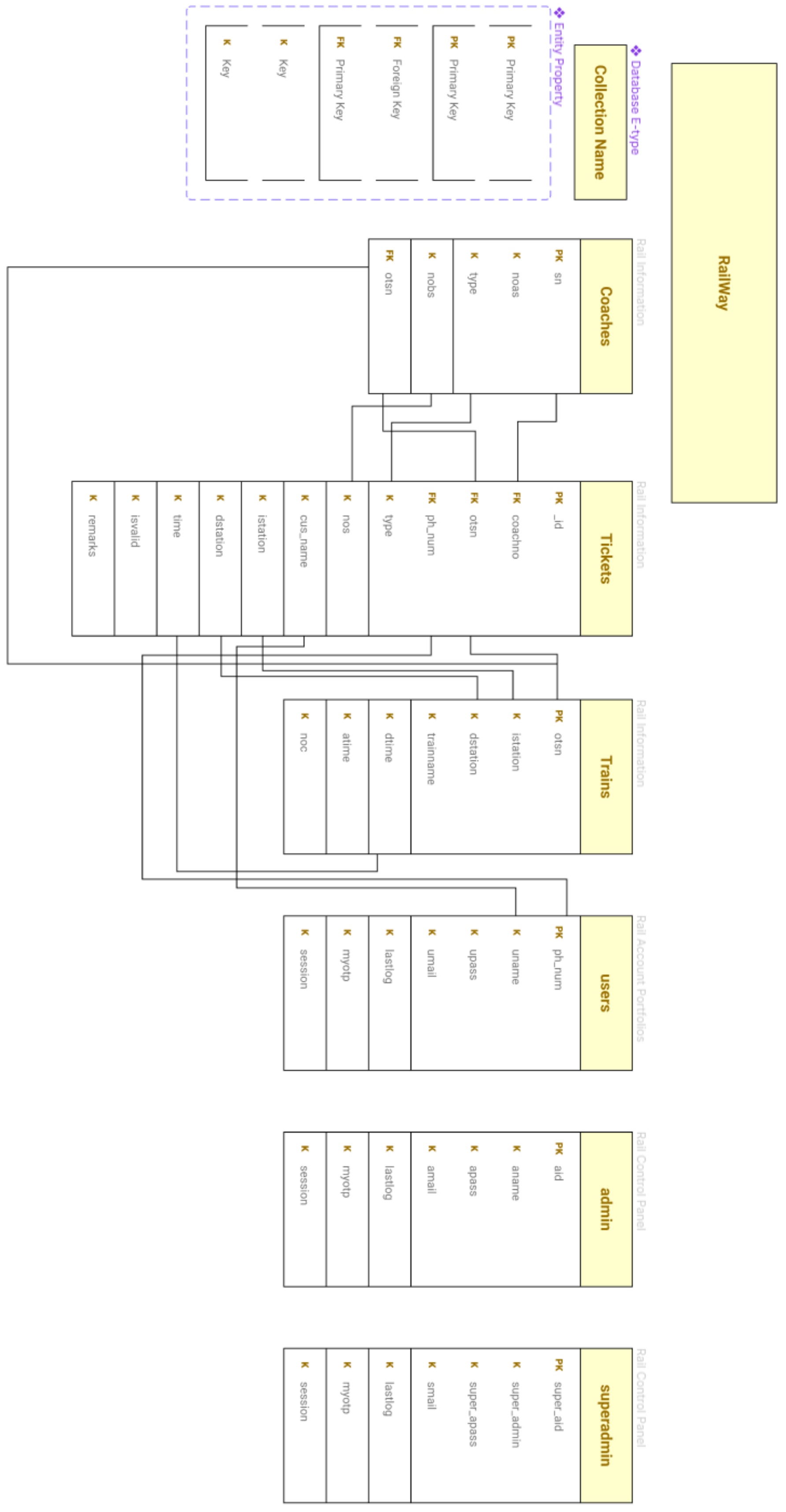
Tickets[otsn] == Trains[otsn]

Tickets[ph_num] == users[ph_num]

Visual Representation of The Database:

The interrelating data makes the database a little complicated. But being a No-SQL database, managing those interrelations is really easy with programming.

The diagram below will demonstrate more about how the database is designed.


RailWay
Database E-type
Collection Name
Entity Property
PK Primary Key
PK Primary Key
FK Foreign Key
FK Primary Key
K Key
K Key
Rail Information
Coaches
PK sn
K noas
K type
K nobs
FK otsn
Rail Information
Tickets
PK _id
FK coachno
FK otsn
FK ph_num
K type
K nos
K cus_name
K istation
K dstation
K time
K isvalid
K remarks
Rail Information
Trains
PK otsn
K istation
K dstation
K trainname
K dtime
K atime
K noc
Rail Account Portfolios
users
PK ph_num
K uname
K upass
K umail
K lastlog
K myotp
K session
Rail Control Panel
admin
PK aid
K aname
K apass
K amail
K lastlog
K myotp
K session
Rail Control Panel
superadmin
PK super_aid
K super_admin
K super_apass
K smail
K lastlog
K myotp
K session

Implementation:

We previously mentioned that we are going to use MongoDB atlas to implement our database.

In the **Implementation** phase, We followed the steps mentioned below.

1. Create a MongoDB Account on the MongoDB Atlas platform
2. Signup for free shared Mongo Cluster
3. Setup Database user
4. Setup Network Access
5. Create a Database and a Collection
6. Inserting test documents
7. Generate connection URL for MongoDB Compass and python script.
8. Surfing

By performing the aforementioned steps, We successfully tested our logical connection across the shared clusters.

The video reference is given here - [https://youtu.be/jU5l37DOo8w]

|| The Backend ||

Description:

A Backend is the logical state of a Program or Service which is completely transparent and serves or makes decisions for the system considering the user arguments. It is responsible for most of the mathematical calculations, secure authentication, and data delivery..

A good backend is responsible for good security and optimized user experience.

Need for the overall project:

Our system needs a medium to make good decisions for clients and to serve data from the database to the front UI. Feeding data without checking the integrity or validating the requests is not a good practice. So we seek the need for an intelligent logic program. Overall, having a backend will allow us to expand our system, add more advanced technologies, and be flexible in accepting new modules. Furthermore, for an intelligent gathering of documents and composing complex HTML, XML, and JSON, we need a good optimized backend solution.

Technology:

As we are to deploy our system to a shared distributed system on Heroku, our application programs in the backend will be language-independent. So by pre-analyzing our needs We, As beginners decided to choose **Python** because of its wide library and functionality. And going with Python made this project easy to manage and more flexible to work with. Because it is a Web application we are using application programming interfaces(API) to define custom functions for single units of requests. We used additional libraries that are described below

Libraries

Pytz - Python Time Zone [World Timezone Definitions for Python]
The Library is used to mark up the exact time stamps while an entity is trying to make a change to the system.
Ex - The time when an Admin logs in.
More detailed info on Pytz can be found here -
[https://pypi.org/project/pytz/]

datetime - Basic date and time types [package where date and time can be operable]
The library is used to store the last change in the system. It helps keep track of the progress, whilst comparing local branches to the master.
More detailed information can be found here -
[https://docs.python.org/3/library/datetime.html]

Flask - Flask is considered more Pythonic than the Django web framework because in common situations the equivalent Flask web application is more explicit. Flask is also easy to get started with as

a beginner because there is little boilerplate code for getting a simple app up and running.
Flask has many configuration values, with sensible defaults, and a few conventions when getting started. By convention, templates and static files are stored in subdirectories within the application's Python source tree, with the names templates and static respectively. While this can be changed, you usually don't have to, especially when getting started.
It is the main frame from where our backend starts.
More detailed information can be found here - [https://flask.palletsprojects.com/en/2.1.x/]

Pymongo - Detailed library of python to connect control and drive a MongoDB asset
By using this library we can perform CRUD operations on our MongoDB database.
It is a secure way to connect a single entity at a singular time to the Database to avoid severe duplication, as many operations can take place at the same time.
More detailed information can be found here - [https://pymongo.readthedocs.io/en/stable/]

bson - Binary Javascript Object Notation
It is a binary-encoded serialization of JSON documents. BSON has been extended to add some optional non-JSON-native data types, like dates and binary data. BSON can be compared to other binary formats, like Protocol Buffers.

We used Bson to decrypt the ObjectId data from ticket collection to serve the number as a ticket because it is unique for every document.
More detailed information can be found here - [https://pymongo.readthedocs.io/en/stable/api/bson/index.html]

Flask-CORS is a cross-origin resource-sharing paradigm of Flask applications.
By applying the CORS policy we made sure that our backend is only available to our frontend requests. And It is responsive to the same origin even multiple times.
More detailed information can be found here - [https://flask-cors.readthedocs.io/en/latest/]

email - EmailMessage dictionary-like interface is indexed by the header names, which must be ASCII values. The values of the dictionary are strings with some extra methods. Headers are stored and returned in case-preserving form, but field names are matched case-insensitively. Unlike a real dictionary, there is an ordering to the keys, and there can be duplicate keys. Additional methods are provided for working with headers that have duplicate keys.
Further, all use this library to send text via the SMTP server.
More detailed information can be found here - [https://docs.python.org/3/library/email.message.html]

smtplib - It is used to make a secure connection to an email server(e.g mail.google.com) via SMTP protocol.
With some direction, our system sends automated OTP alerts, updated to user mail at ease. As the threading is very much adaptable, it is possible now to send multiple emails at the same time.
More detailed information can be found here -
[https://docs.python.org/3/library/smtplib.html]

base64 - Its library is unique encryption and decryption engine on python.
The password and other sensitive details our website captures from users are very valuable and must be kept and stored securely. As a responsible database administrator our approach of using base64 to encrypt passwords for keeping in databases
More detailed information can be found here -
[https://docs.python.org/3/library/base64.html]

Json - JavaScript Object Notation
Using the JSON library we can cripple multiple data in key-value pairs and propagate through different APIs or frontend to backend and vice versa.
JSON is the hashlike data notation where the properties are the same as Hashmap but it has encoders and decoders by default to protect the data for non-Json object receivers.
More detailed information can be found here -
[https://docs.python.org/3/library/json.html]

secrets - This library is just used for generating random strings to use as session data. This helped the authentication to be more secure and flexible.
More detailed information can be found here - [https://docs.python.org/3/library/secrets.html]

System Requirements:

Although the system is web-based, It still needs a considerable amount of processing power and software solutions. The system requirements are to be classified as -

i) By Hardware
ii) By Software

Hardware Requirements:

CPU - 800 Mhz x86 CPU or Higher.
RAM - 512 MB of Free Memory
Storage - 1 GB of free disk space
Network - A Valid Network Card with at least 10 Mbps Speed.

The host machine should have the aforementioned minimum to smoothly operate on its own.

Software Requirements:

OS - Windows XP SP1 or later
Dependencies - Python, Visual C++ Redistributable, Werkzeug

The mentioned dependencies are needed to run the backend server without any error.

Component Design:

The backend is running as a Flask application We made using python.
By component, We mean the microservices that define unit individual tasks. Our goal is to design every unit of a task in the form of Flask APIs.
There are a total of 48 microservice APIs in our system. That is why it is quite impossible to define every one of them in this documentation. Although source code is provided for further understanding and descriptive comments are also added so that the reader can understand all of them. The main working units are described below.

- ☐ Public Queries(Trains, Coaches, Seats, timetable)
- ☐ User Login/Registration
- ☐ User Reservation[Ticket Booking, Inventory, Ticket cancellation]
- ☐ Admin login
- ☐ Admin Manages Trains and Coaches(add, edit, remove)
- ☐ Admin Manages Reservations(view, cancel)
- ☐ Admin Other Queries(timetable, mothertable, documentation)
- ☐ Super Admin Login
- ☐ Super Admin Manage Admins(add, remove, edit)
- ☐ Super Admin Queries(admin information table)

- **Public Queries**: for public queries, we used APIs mentioned below -
  1) *t_view*: We are searching the Trains collection by the istation and dstation provided by the user. From the cursor, we are creating an HTML table element and returning it to the frontend
  2) *c_view*: We are searching the Coaches collection by the istation, dstation, and coach type provided by the user. From the cursor, we are creating an HTML table element and returning it to the frontend. The table also contains seat information.
  3) *ticketview*: We are taking session information and searching through user collection for the logged-in user. Once retrieved we are getting the ph_num of the logged-in user from the document and searching for tickets that match with ph_num. One retrieved from the cursor, We are creating an HTML table element and returning it to the frontend
- **User Login/Registration**: In the authentication part we used APIs mentioned below -
  1) *userlockdb*: Getting the phone number and password from the frontend and searching the *users* collection. In the flagged document creating a session variable and returning to the frontend

2) *registeruser*: Getting phone number, email username, and name from the frontend. This new information is stored in the database as a new document.
3) *userlogout*: While a logout request is being generated from the frontend. The session variable of the particular document is set to empty.
4) *changepass*: an OTP based password changing API in action. A randomly generated 6-string number is sent to user's email as an OTP.

- **User Reservation:** For reservation, the user must log in. In other words, a browser must have a valid session to book, see, and cancel tickets. The APIs are -
  1) *book*: Given the initial station, destination station, departure time, coach type, and the number of seats, The book API point down considering every argument and flags suitable coach for booking. Also updating that particular coach document, this API successfully generated a new ticket document and returns it to the frontend. A mail is also sent to the user.
  2) *ticketview*: This is the same as the public query.
  3) *cancelticket*: Given the ticket number and right password, The *cancelticket* API is called to just change the state of the flagged ticket document from valid to invalid.
- **Admin Login**: In the authentication part we used APIs mentioned below -
  1) *adminlock*: Getting the phone number and password from the frontend and searching the Admin collection. In the

flagged document a randomly generated 6-string number variable is created as an OTP and returned to the frontend as well as the users email.

2) *adminlockotp*: After getting the correct OTP, this API generates a session variable to be stored in the database. A copy of the session variable is also sent to the front for further use.
3) *adminlogout*: When the frontend generates a logout request this API gets called. This API just resets the session variable that is stored in the database.

- **Admin Manages Trains and Coaches**: For this kind of managing, Train and Coaches have almost the same fashion workflow. The APIs that help manage these documents for Admin portfolios are given below.
    1) *t_insert & c_insert*: These APIs take the frontend generated JSON value and depending on the already available documents create a new document.
    2) *t_edit & c_edit*: These APIs take the train number and coach number to flag the document to be edited and sets the new values frontend generated avoiding contradiction. Although this system change if any ticket is booked on the system, then it automatically gets canceled.
    3) *t_delete & c_delete*: These APIs take the train number and coach number as input and flag the document to be deleted. After successful deletion, a success message is generated.

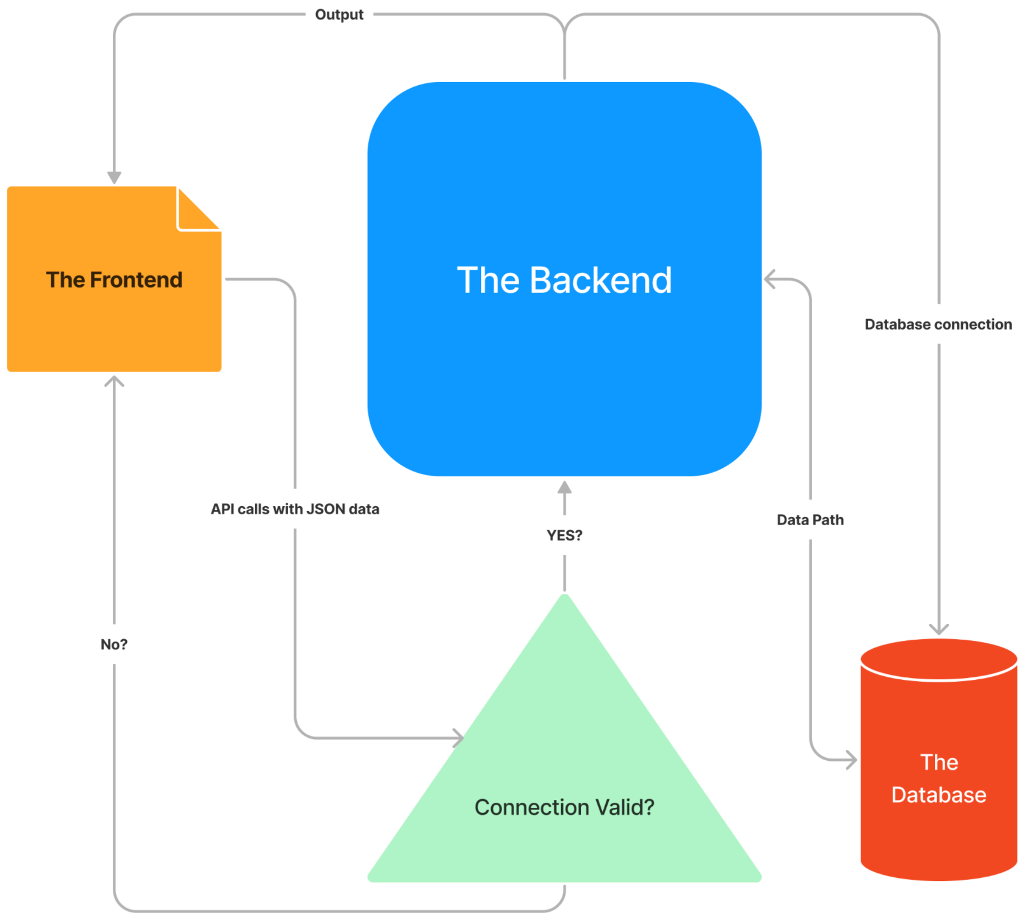
- **Admin Manages Reservations**: For this kind of managing, The APIs given below are used -
    1) *alltickets*: This API collects every documentation the ticket collection and returns it as an HTML table element to the frontend.
    2) *cricket*: This API helps the admin to cancel any ticket on the database. Given the ticket number, the document is flagged to be invalid from valid.
- **Admin Other Queries**:
    1) *ticketview*: This is the same as the public query.
    2) *mothertable*: This API returns a big HTML table element containing every coach, train, and seat information with every required information. This table can be called a Master table.
    3) *documentation*: This API just returns the source of the documentation of this project to the front.
- **Super Admin Login**: It is the same process as Admin Login previously described. The access is very limited as it possesses the ultimate portfolio of the system.
- **Super Admin Manages Admins**: Instead of hardcoding Admin information to the database, it's more efficient to open a portal to control admin documents. For adding, removing, and changing Admin information on the system We use these APIs below -

1) *registeradmin*: Similar to user registration but here, the database administrator creates the Admin account for him/her. The required information like username, password, email, and name is stored as a new document on the admin collection.
2) *adminchange*: This API changes admin faulty information and reset them and sets the new values to the database.
3) *admindelete*: Given the admin username a database administrator calls the API to post a deletion request of the flagged document on the database.

- **Super Admin Queries**: Although there is not much functionality on Super Admin Portal, Super Admin can visit and see Admin status in a tabular fashion.
    1) *alladmins*: This API returns the admin collection to the frontend but with just username and last login information for security purposes.

DFD (Data Flow Diagram):

This DFD will provide a visual representation of how data travels from frontend to backend filtered decision-making to the database.



Implementation:

To implement the backend our primary IDE choice was Visual Studio Code because it provides useful extensions that make complex programming easy.
We created a local server first and tested our APIs using Postman. After testing and debugging every API, we deployed them to the main server which is hosted at Heroku. [irctcbackend.herokuapp.com] As we are using a different server for the frontend, We had to set up cross-origin resource sharing across those servers. After a successful deployment of the local branch, We tested the system with maximum stress and found a lot of impurities while handling a chunk of data. At first, the server was slow, and buggy, user experience was very poor. We fixed it one by one with patience and dedication. It is still unstable, but we are improving the end-user experience day by day. The steps below will give a short brief on how we implemented every API.

Building a Test Server using Flask:

As previously mentioned We will be using Python to create our test Flask application. Our choice of IDE will be Visual Studio Code.

- Installing all dependencies
- Getting started on Python, VS CODE
- Creating a Flask app on Python
- Running the Local server

- Creating an API and connecting to the database
- Sending test data

Every step is well demonstrated in the video - Click Here [https://youtu.be/sGyhf7HNsq8]

APIs with arguments and The Postman:

Postman is a Public API platform for developers to design, build, test, and iterate their APIs. It helps developers make better decision logic for their APIs.

- Building a conditional API with arguments
- Installing Postman
- Making the environment ready for API test
- Setting up server URL and Checking Methods
- Setting JSON data as arguments
- Sending a request to get the result

Every step is well demonstrated in the video - Click Here [https://youtu.be/RLlWMdnTKjY]

Deployment:

Our system is a web application. It can not stay in just local servers. We decided to deploy into the open. We ventured into some open-world deploy station choices. But among them, Heroku was giving us the best deal. So, using git we created our repository and deployed our system online. We followed these steps accordingly -

- ☐ Creating Procfile
- ☐ Creating Web Server Gateway Interface and Importing app
- ☐ Making runtime configuration
- ☐ Making a list of requirements for the server to install
- ☐ Logging in through Heroku-CLI
- ☐ Initiating an empty repository
- ☐ Committing all files to be pushed
- ☐ Creating Heroku Web app
- ☐ Pushing the files to the empty repository
- ☐ Checking the server
- ☐ Fixing if any error persists
- ☐ Sample push demonstration

Every step is well demonstrated in the video - Click Here [https://youtu.be/Hpwe4ulyD5U]

|| The Frontend ||

Description:

The Frontend is the UI or the face of the system with which users and admins interact. This is the part where the data for the backend server is generated. Our frontend point is the most minimalistic ever. We used traditional HTML, CSS, and JS components to construct every page. The data compilation of the frontend is written in plain JavaScript.

Need for the overall project:

Our system needs a medium to interact with the backend. Our backend accepts data structures that we only can provide with API testers or AJAX calls or any XML-HTTP requests. As we are to publish our system for regular users with little to no knowledge of the aforementioned paradigms, We will certainly need a Frontend to be the face of the system. And the frontend server will generate and forward user requests and data to the backend seeking results which will be filtered to a human-friendly format so that users can understand the input and moves on.

Technology:

As we are to deploy our system to a shared distributed system and the Frontend might have a lot of pages to represent, a feasible idea would be of using a similar flask application like the backend. The flask application will render our traditional HTML to a certain call point and maintain a secure connection to the backend. Here just for the sake of data transfer from the cross-server we have to use cross-origin resource sharing provided by Flask-CORS.
Again we will be using python to implement this kind of application. The libraries we require are listed below.

Libraries

Bootstrap - Provides a wide library of design components
Fully dynamic, lightweight, optimized, and easy-to-use library that gives us a refreshed look and feel.
More detailed information on Bootstrap can be found here - [https://pypi.org/project/pytz/]

Flask - Flask is considered more Pythonic than the Django web framework because in common situations the equivalent Flask web application is more explicit. Flask is also easy to get started with as a beginner because there is little boilerplate code for getting a simple app up and running.
Flask has many configuration values, with sensible defaults, and a few conventions when getting started. By convention, templates and static files are stored in subdirectories within the application's Python source tree, with the names templates and static

respectively. While this can be changed, we usually don’t have to, especially when getting started.
It is the main frame from where our front end starts.
More detailed information can be found here -
[https://flask.palletsprojects.com/en/2.1.x/]

AJAX - Asynchronous JavaScript and XML is a method that uses a combination of technologies that allow the content on webpages to update immediately based on a user's action, which may be a click on a page or even a simple mouse movement. Just one or a few parts of the page may be refreshed, instead of reloading or refreshing the entire page. This differentiates AJAX from an HTTP request, during which users must wait for a whole new page to load. AJAX can also access data from external sources even after a webpage has loaded completely. The web handle requests are straight passed to AJAX Jquery to be forwarded to external API calls. It is also responsible for receiving the external data and pass to JavaScript for further operation.
AJAX uses JavaScript for dynamic content display, XHTML for content and XML to receive server data. In addition, it combines many other programming tools
More detailed information can be found here -
[https://api.jquery.com/category/ajax/]

System Requirements:

Although the system is web-based, It still needs a considerable amount of processing power and software solutions. The system requirements are to be classified as -

i) By Hardware

ii) By Software

Hardware Requirements:

CPU - 800 Mhz x86 CPU or Higher.

RAM - 512 MB of Free Memory

Storage - 500MB of free disk space

Network - A Valid Network Card with at least 10 Mbps Speed.

The host machine should have the aforementioned minimum to smoothly operate on its own.

Software Requirements:

OS - Windows XP SP1 or later

Dependencies - Python, VC runtime 2013 or higher. DirectX 9.0 or higher.

The mentioned dependencies are needed to run the backend server without any error.

Component Design:

The frontend is running as a Flask application We made using python.
By component, We mean the that define unit individual tasks. Our goal is to design every unit of a task in the form of Flask APIs.
The Basic components would be -
i) **HTML** - As we are developing Web-based software, It bounds to follow the traditional hypertext format which is HTML.
ii) **JavaScript** - It is a very complicated system where decision-making is a frequent phenomenon. By using JavaScript we can load components dynamically, handle data, communicate with the backend, execute backend pulls, and validate requests.
iii) **CSS**[Cascading Style Sheets] - To give our UI a refreshed yet simplified look, We used style sheets of our own as well as a widely available bootstrap library.
iv) **JSON**[JavaScript Object Notation] - Our system prioritize data. Being a data-driven system, We had to come up with different data structures, Among them we found Hashmap, the hashlike data structure easier to use, Thus we decided to use JSON, where we can send chunks of objects to the backend to be processed. Being in a hashlike fashion, JSON objects are easy to manipulate and can be unpacked in every programming language.

We used traditional HTML to write the skeleton of every page, used CSS to style those pages, and used JS to pack data from HTML and dynamically load pages or HTML components.
v) **jQuery** - jQuery is a fast, small, and feature-rich JavaScript library. It makes things like HTML document traversal and manipulation,

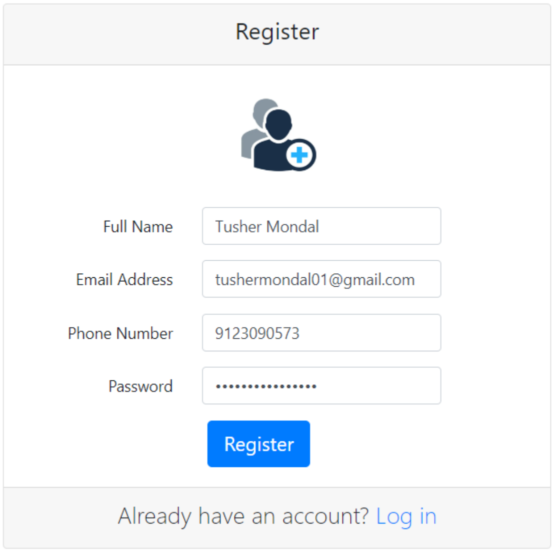
event handling, animation, and Ajax much simpler with an easy-to-use API that works across a multitude of browsers. With a combination of versatility and extensibility, jQuery has changed the way that millions of people write JavaScript

The Frontend has three core components that take the data from the user and packs it and sends the data for the backend to process.

1. **The Receiving Layer:** This layer is closer to the user. The user-given data and preprocessed data from HTML components are a part of that layer. Suppose there is an HTML form element, in that form element the data values on that field are being received. This kind of data is unprocessed and not suitable for the backend to perform operations on.

Example -

```html
<input type="text" id="uname">
<input type="text" id="umail">
<input type="number" id="ph_num">
<input type="password" id="upass">
```



Register

Full Name	Tusher Mondal
Email Address	tushermondal01@gmail.com
Phone Number	9123090573
Password	••••••••••••••••

Already have an account? Log in

2. **The Processing Layer:** This layer is responsible for processing the user-given or HTML-generated data as backend friendly and compact so that it can be sent using minimal bandwidth.

JSON is a commonly used paradigm that helps pack the data into a smaller object that can be passed over the network. JSON defines seven value types: string, number, object, array, true, false, and null. It uses a hashlike structure so accessing it in any programming language is not a problem. Processing means that the data needs to pack in a structure by key-value or array objects. We used traditional JavaScript to pack this data using **stringify** and created packets to be sent for transfer.

Example -

```
    var ph_num = document.getElementById("ph_num").value;
    var upass = document.getElementById("upass").value;
    var uname = document.getElementById("uname").value;
    var umail = document.getElementById("umail").value;
    var jdoc = {"ph_num": ph_num, "upass": upass, "uname": uname,
"umail":umail};
    jdoc = JSON.stringify(jdoc);
```

Here ‘jdoc’ is a JSON object ready to be sent over a network to the backend.

3. **The Connection Layer:** This is the most needed layer for interconnection between servers and data transfer. We used jQuery to access the endpoints of the backend. By making an asynchronous secure connection to the backend it will wait for a 200 status message (i.e 200 means connection successful). After getting a successful acknowledgment it will send the data to the data to the particular endpoint along with browser data. This kind of handshake is needed to avoid discrepancy. Since **Asynchoruns JavaScript and XML** (AJAX) is easy to use

and debug, We chose it to use for the system. The success report is easy to receive since We are using JavaScript no middle port is needed.

Example -

```
$.ajax({
      url:"https://irctcbackend.herokuapp.com/registeruser",
      type:"POST",
      contentType:"application/json",
      data:jdoc,
      async: false,
      success: function(msg)
      {
            document.getElementById("msg").innerHTML = msg;
            if (msg=="Registered Successfully")
            {
                  window.location =
            "https://apccproject.herokuapp.com/userlock";
            }
      }
      })
```

Here the endpoint is -

```
url:"https://irctcbackend.herokuapp.com/registeruser",
```

The data we previously processed into an object is sent to this mentioned endpoint.

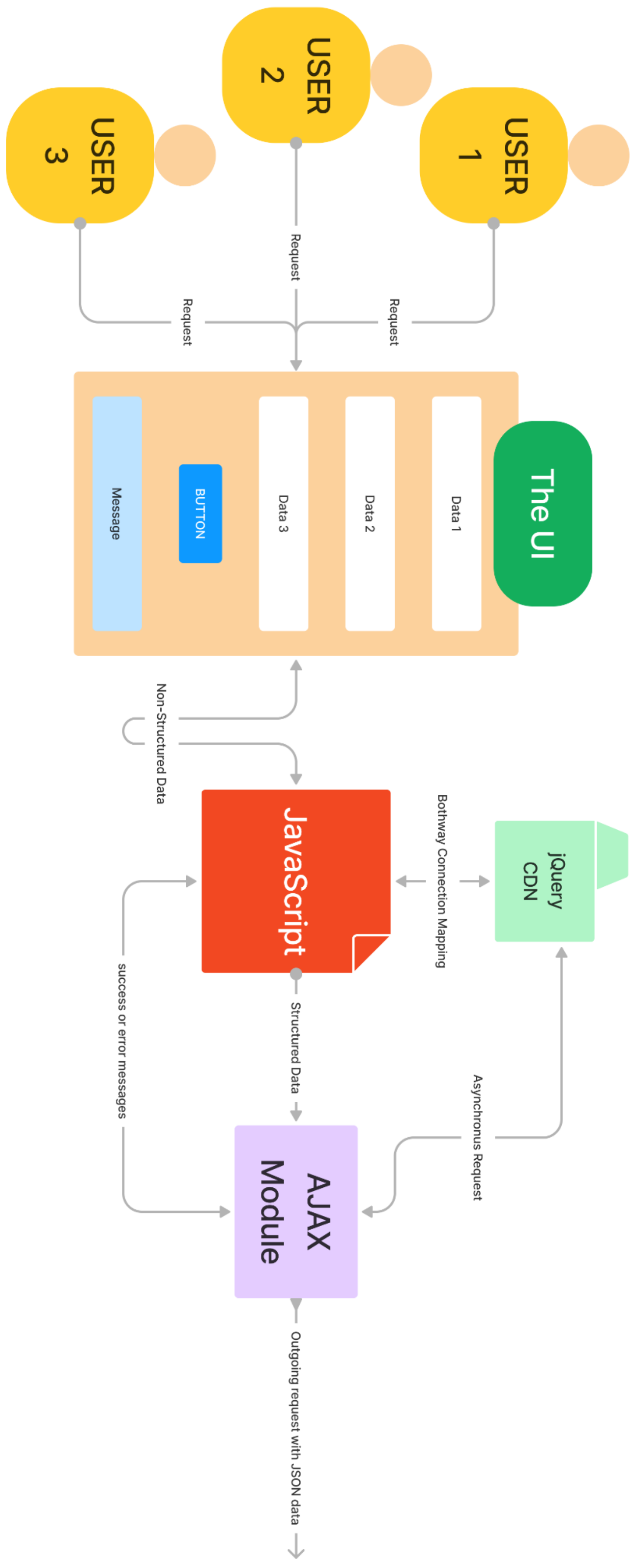
```
data:jdoc,
```

If the connection is successful and the backend endpoint delivers a message or state, the success port will generate JS-friendly data. By the JS readable data, we can point and decide on the frontend.

```
success: function(msg)
```

DFD(Data Flow Diagram):

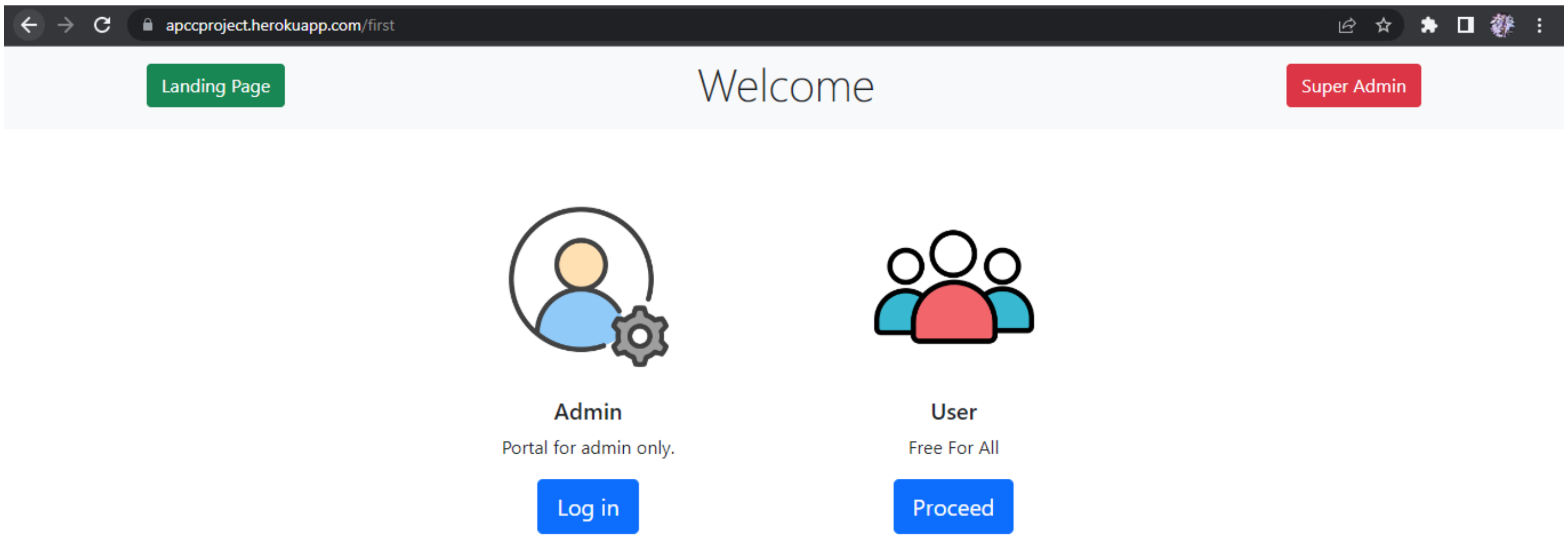
This DFD will provide a visual representation of how data gets dressed from raw input to smaller objects for data transfer to the backend over the network.



Implementation:

To implement the frontend our primary IDE choice was Visual Studio Code because it provides awesome extensions that make programming easy.
Like the backend, we tried to implement the frontend in the local server. There was little to no programming as We are to only render the HTML documents we wrote, in the endpoints.
Example -

```
@app.route('/first')
def first():
    return render_template("index.html")
```



Here ‘first’ is the endpoint and index.html is the document to render.
Considering the system requirement specification, We built every page according to the backend APIs. We used JavaScript for

hassle-free navigation between endpoints. The deployment offers the source code of both the backend and the frontend to admin accounts.

In JavaScript, We took user-given values or Frontend generated values in a suitable format and packed them to a JSON object to transfer with AJAX.

Example -

```
    var c_no = parseInt(document.getElementById("coachno").value);
    var aid = sessionStorage.getItem("aid");
    var session = sessionStorage.getItem("logged");
    const doc = {"coachno":c_no, "aid": aid, "session":session};
    const jdoc = JSON.stringify(doc);
```

And After that considering the method for the backend, we initiate a nonasynchronous file transfer to the particular endpoint.

```
      $.ajax({
        url:"https://irctcbackend.herokuapp.com/admin/coach/delete",
        type:"DELETE", //method
        contentType:"application/json",
        data:jdoc,
        async: false,
        success: function(msg)
                 {
                      document.getElementById("msg").innerHTML = msg;
                      choose(msg);
                 }
           })
```

By following this similar fashion we implemented almost every part of our frontend.

Deployment:

We could deploy our frontend along with the backend flask application. Even though there is too little to code and is just HTML renders, we still decided to go with a different server to balance the workloads. Surfing the system doesn’t use backend resources until provoked. Separating the server uses minimal CPU operation and it is more economical.
We followed similar methods as the backend to deploy our frontend server to Heroku.

|| Compilation ||

Job-Ready, CI/CD Pipeline:

Before deployment, both the servers were running on a local server, In JS and Python sources we had to change the endpoints to the new deployment URL. We are using CI/CD(Continuous Integration and Continuous Deployment) through git. It is much easier to track and build an agile cycle movement without losing screentime.

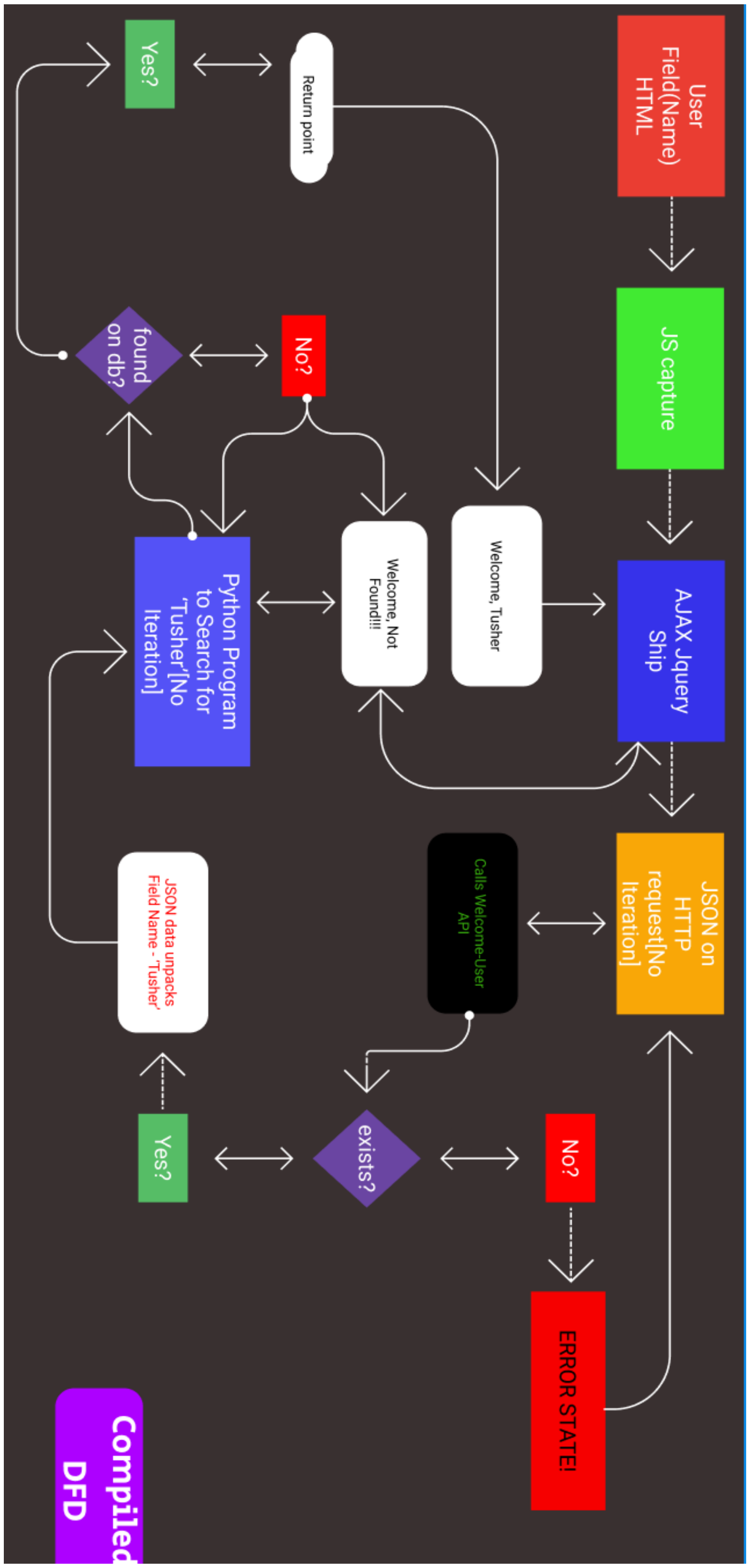
CORS Policy:

Being two different web servers running differently, both servers need cross-origin resource sharing authentication, Unless they will not be able to talk to each other or share data. As previously mentioned we used Flask-CORS to provoke the frontend's URL as the origin in the backend source code and vice versa.

Stability and Reliability:

Heroku provides a very limited resource for free projects. Still, we tested a lot with different environments and data. It became unresponsive several times, The validity of the data was not good. After one month of optimization, The unresponsiveness is not there anymore. Day by day We are implementing new security approaches for the system. It is still not the best, but regular maintenance is the routine to make it more stable day by day.

Compiled DFD(Data Flow Diagram):



|| End-User Documentation ||

Operating System Requirements:

Anything better than Windows XP SP1 is recommended. A browser with HTML 5 support is needed.

How to use this Software/Application:

As this is Web-Based Software, the User needs to open a browser of his/her choice and dial this URL below.

https://apccproject.herokuapp.com

Accounts:

First of all, Please do not try to log in to the Super Admin section. This section is intended for Database Administrators only.
If you are an Admin, Login using the username and password the Super Admin/Database Administrator provides. You should have access to your email. An OTP will be sent to your email for authentication. For quick navigation go to this URL below.

https://apccproject.herokuapp.com/admin

If you are returning User, Login using the username and password you have set during the registration process.

https://apccproject.herokuapp.com/userlock

If you are a new User, Proceed through the registration process by the link given below, then try to log in.

https://apccproject.herokuapp.com/register

If you forgot your password you can visit this link. In this case, you should have access to your email id.

https://apccproject.herokuapp.com/user/forgotpassword

Please visit our Web site with caution. Do not use real-life data on it. It could lead to privacy breaches.

For further questioning, one can contact us

1. Tusher Mondal
   a. Email - tusher9073mondal@outlook.com
2. Aniket Sarkar
   a. Email - aniketkolkata24@gmail.com