# ABSTRACT

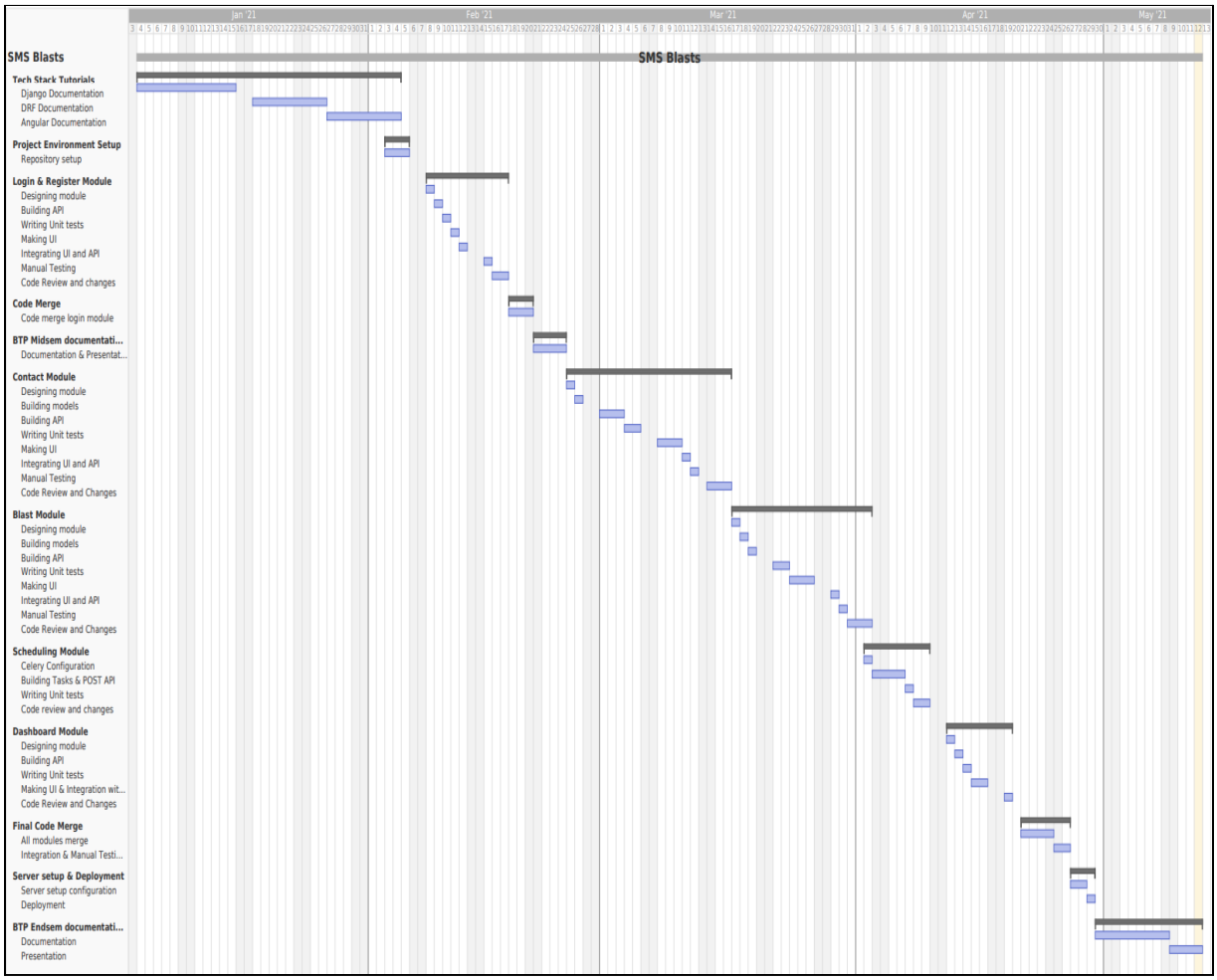
Imagining life without the internet is nearly impossible for us. The multitude of technological solutions and products from online shopping to video conferencing has made it possible for us to connect to any part of the world with just one click. And yet 50% of our country's population, i.e. , roughly 700 million people are still disconnected from the internet [1]. Last mile communication is still a major problem that is not addressed by internet based solutions. We propose a bulk SMS Blast system with an aim to address the Last mile communication issue and serve the underserved communities. The proposed solution let's the registered user upload contacts and schedule bulk SMS after logging in. It also provides insights of reached, unreached and unreachable unique contacts on the dashboard. This solution is a web based solution but it still addresses the last mile communication problem.

# LIST OF FIGURES

# LIST OF FIGURES (CONTINUED)

# LIST OF TABLES

# TIMELINE




For clear visibility, please click on this link to view the timeline.

# TABLE OF CONTENTS

# 1. Introduction

## 1.1 Purpose

This report is a detailed description of the **SMS Blast Web Application**. The purpose of this report is to clearly state all the functional and non-functional requirements of the software, design and testing strategy of the software, implementation details and results.

## 1.2 Scope of the Project

With the penetration of the internet and smartphones in our daily lives, the number of mobile and web solutions for all the facets of life has tremendously increased. Yet these solutions do not fully serve our population. According to a survey conducted by Omidyar Networks, only 5% of the rural population in India uses mobile internet. Another survey done by Statista in 2017 summarised that approximately only 14% of the surveyed farmers use mobile Internet [2]. With time this penetration of the internet is increasing in the rural population but there is still a huge chunk of population that is underserved due to last mile communication barriers.

To address these challenges to last mile communication, we propose to build a SMS Blast Application. This software allows the customer/consumer of this software to upload contacts and schedule a SMS blast and view the statistics of the blast.

The application of this software is in many different sectors like microfinance, agriculture, and education. This software can be used by the companies in this sector to establish a direct connection with their last mile users by sending them educational, informational, or reminder SMS in their vernacular language.

## 1.3 Process Description

For the development of this software, we are following an Agile software development process and Scrum methodology.

The reason we have picked Agile as our software development process is because it is iterative and has a dynamic approach to software development. This allows us to respond to the real need of the users by releasing software more often and quickly.

One of the other benefits of this process is it allows the developers to quickly build software and test it and give a working deliverable at the end of each sprint.

## 1.4 Stakeholders

- **Internal Stakeholder**:
  - AwaazDe: The project definition, resources and the guidance required for the completion of the project is provided by the company to the developer's team.
  - Onsite Supervisor: The guidance and support required for the completion of the project is provided by the onsite supervisor to the developer's team.
  - Internal Supervisor: The guidance required for the successful completion of the project is provided by the internal supervisor to the developer's team.
  - Team of Developers: The development (back-end and front-end coding), as well as the testing of the software, is done by the developer's team.
  - Ahmedabad University: The setting of standards for the project and evaluation of whether software meets the standards is done by the University.
- **External Stakeholder**:
  - Mass SMS Sender: They are the main consumers of the software.
  - Last Mile User: They are the end users of this product, but this product is made to serve them. It is very important to ensure that the software is serving them.

## 1.5 Definitions, Acronyms, and Abbreviations

- Mass SMS Sender: Mass SMS sender term is used to refer to the customer of this software who intends to send bulk SMS.
- Last mile user: This term is used to refer to the customers of the Mass SMS Sender, who do not have access to the internet or do not use the internet.
- Blast: It is a broadcast of a pre-decided message that can be scheduled for multiple contacts.
- Use Case Related Abbreviations:
  - DESC: Description - INC: Includes - EXT: Extends
  - DEP UC: Dependency Use Case - DEP REQ: Dependency Requirements

# 2. Literature Survey

## 2.1 Existing System

The SMS concept and idea was first evolved by Friedhelm Hillebrand and Bernard Ghillebaert in 1984 [3]. It was witnessed that initial popularity and growth for SMS was slow, but by the early 2000s, the total number of messages exchanged indicated an exponential increase [4]. As one-to-one text messaging became popular, many companies started developing softwares for delivering bulk messaging services. Bulk messaging is the process of sending SMS messages to multiple phone numbers. This service is delivered by many software companies and is used by media companies, banks, large organizations, etc for various use-cases like marketing, entertainment, and many more [5]. 'Text Local' is another such platform which provides services like SMS campaigns, instant OTPs, notifications, two-way interactions, and other bulk SMS services [6]. 'SMS Gateway Hub' is another such application which provides bulk messaging services that helps to send promotional, OTP, marketing, bulk SMS gateway and API SMS [7]. All bulk messaging applications ask the user to upload contacts for a particular campaign and schedule the message at a particular time in the future. This literature survey concluded that these applications are efficient and provide many services as well, but most of these systems lack in providing the user with the information of the number of duplicate and invalid contacts present in the file which the user uploads while creating a campaign or group. They also do not provide the functionality of editing the contact details which are once uploaded by the user. These functionalities are provided by our system which gives users more flexibility and information in totality.

Among the new technologies which supported bulk messaging, Electronic mail was one such medium of communication. This facilitated a lot of organizations, companies, and corporations to make use of the email communication to reach to the masses for various purposes. However, the major drawback of this type of communication, especially for a developing country like India, is that this type of communication requires Internet connectivity. Though the Internet has enabled people from all across the world, still data shows that over half of the world's population was disconnected from the Internet in the year 2017 [8]. Statistics also show that, in the year 21018, 80% of India's population did not use the Internet [8], and many of these people live in poor, remote areas. Hence,

internet technologies providing bulk messaging service, no matter how effective and efficient, lack in providing last mile communications to underserved communities.

## 2.2 Proposed System

Awaaz.De Infosystems, an organization situated in Ahmedabad, creates innovative technology to improve the lives of underserved communities across the world. It's main mission is to enable social change through inclusive mobile solutions and services for last-mile communications. Last-mile communication ensures that underserved communities can access information using technology available to them–such as a basic feature phone**.** It's solutions for social impact are used in several sectors such as education, health, finance and agriculture. It provides both bulk sms as well as voice messaging services and has support for different languages so that under-served communities can be benefited through this platform by interacting in their local language [9]. Our system implements a subset of functionalities of Awaaz De's messaging system. It is an end-to-end use case of a Web based Application which provides bulk messaging and analytical services integrated with a third party application.

It is highly important and critical for any individual or enterprise, including the world's poor, to have access to relevant and timely information to make good decisions [10]. Information and communication technologies help improve livelihoods in the whole world, but since the low-income and underserved communities do not have access to digital platforms, smart phones and internet connectivity, there is a risk of leaving low-income communities on the other side of a digital divide [10]. This system is based on an "Internet for the few, mobiles for the many" access model. To solve this problem, this bulk messaging system can be used to reach out to those communities through messages and thus, enabling last-mile communication. One such scenario is that the people from the aforementioned communities often take loans from lenders or Microfinance industries. But, they lack knowledge in financial literacy, and are more vulnerable to exploitation/abuse by loan officers and agents. Our system (similar to Awaaz.De's messaging system), can provide an easily understood communication medium such that sufficient information about loans, and exposure to new, beneficial products and services can be achieved through messages.

This system allows users to upload contacts of various countries which can further be used by the user to schedule bulk messages (called as blast) on a specific date, with a particular message to a selected set of contacts. It also provides a dashboard which consists of the analytics of the messages received, and unreceived messages to the contacts which were selected for blasts. It also provides users with the functionality of downloading reports of individual scheduled blasts consisting of the summary of the status of the contacts (reached, unreached and unreachable).

## 2.3 Feasibility Study

One of the key sectors where this solution is used is the Microfinance Industry. In the microfinance industry, the companies do not have direct connection with their end users. They are connected to their end users via the field agents. Field research in this area has shown that many mal practices have been observed in this arrangement where the end users have been exploited at the hands of field agents and the company too have suffered losses because of the dishonesty of field agents. These observations point towards the need for a more transparent system and direct contact between the Microfin companies and their end users.

With reference to this the feasibility of the proposed solution can be observed as follows:

- Technical and Economical Feasibility: Almost all the softwares used in building this product are open source softwares except the MSG91 API that is used to send messages. Another cost involved is that of infrastructure. Since the deployment is not the scope for the BTP project, we do not require dedicated server infrastructure nor do we have to worry about server maintenance. Therefore the proposed solution is technically and economically feasible.
- Operational Feasibility: The proposed solution provides direct connection between the MFI and it's end users. It also allows information dissemination by allowing the Mass SMS senders to do text campaigns revolving around timely repayment, interest rates, government schemes and many more. This proposed solution provides security to the end users from exploitation at the hands of agents.

# 3. Overall Description

The following section provides an overall description of this project. In particular, it describes the product perspective and defines the system interfaces used by the product. Further, concise and clear objectives of the system and detailed deliverables of this project are included in this section. To conclude the section, hardware and software specifications of this system has been described.

## 3.1 Product Perspective

The software described in this SRS is the software for a bulk messaging and analytical system with third Party Integration. The system has various hardware and software elements which are integrated with external systems to satisfy the requirements. The system merges various hardware and software elements and further interfaces with external systems. Though software elements cover the main functionalities of the system, it still depends on external systems for various specific unhandled tasks.

### 3.1.1 System Interfaces

The SMS Blasts messaging system interfaces with an existing communication platform, known as Msg91, in order to use Msg91 API as third party to streamline the process of communication between the system and its customers by the use of automation by integrating automated messages and planning and triggering SMS notifications to the customers.

## 3.2 Objectives and Deliverables

**Objectives:** Given below are the major objectives of this system is to -

- Allow Users to register into the system.
- Allow Users to login into the system with right credentials.
- Allow Users to add contacts by importing a file (excel or CSV) and edit contacts.
- Allow Users to view and browse the list of contacts added by that user.
- Allow Users to send Blasts to send to specific contacts at a scheduled time.
- Allow Users to download delivered blasts report.
- Allow Users to view details of each of the delivered blasts.

- Allow Users to view the analytics of the delivered Blasts on the Dashboard.

**Deliverables:** Given below are the deliverables of this B.Tech project -

- Working Software that incorporates all the functional requirements, necessary objectives and non-functional requirements.
- Presentation
- Documentation
- Source Code
- User Manual

## 3.2 Hardware and Software Platform

Following is the tech stack for this project:

- **Hardware Platform**:
  - Processor: Intel® Core™ i5-9400F CPU @ 2.90GHz × 6
  - Graphics: NV106
  - Disk: 234.2 GB
  - OS Type: 64 bit
  - Dell Desktop Machine
- **Software Platform**:
  - OS: Debian 10
  - Web Development framework backend: Django(3.0.3) and Django Rest Framework(3.11.0)
  - Web Development framework frontend: Angular (Node 14.15.4 and Angular cli 11.1.2)
  - Database Management Server: PostgreSQL(11.10)
  - Scheduling: Celery(4.0.0)
  - Communication API: MSG91
  - Server side language: Python(3.7.3)
  - UI Framework: Bootstrap

# 4. System Requirements

## 4.1 Functional Requirements

| ID | REQUIREMENTS |
|---|---|
| REQ1 | The system shall require all the new users to sign up to use the webapp. Required information shall include first name, last name, a unique email, unique username, a password and retype password that adheres to guidelines. Upon successful registration, the system will set up an account for the user. |
| REQ2 | After successful registration, the system shall result in a pop up message informing users of successful registration and link to the login page. |
| REQ3 | The system shall allow users to login to the system using registered username and password. |
| REQ4 | The system shall have a link to the registration page at the bottom of the login page. |
| REQ5 | The system shall redirect users to the Dashboard page after successful login. |
| REQ6 | The system shall allow users to log out from the system. |
| REQ7 | The system shall redirect users to the login page once the user logs out. |
| REQ8 | Users can navigate to any page (dashboard, contact, blast) by using the navigation links present at the top of each page (after successful login). |
| REQ9 | The system shall allow users to create contacts by uploading csv/xls/xlsx file (of max 4MB size) containing fields: contact number, first name and last name, in the same order. |
| REQ10 | The system shall return a summary response after the user successfully uploads a contact file. The response contains information about number of valid, invalid and duplicate contacts |
| REQ11 | The system shall allow users to view all the contacts created by the user ordered by most recently created. |
| REQ12 | The system shall allow users to edit contacts, by clicking on the contact (which is to be edited) shown in the contact list page. |
| REQ13 | Contact upload does not overwrite existing numbers, it only creates new ones. |
| REQ14 | The system shall allow users to create a blast by asking them the information: blast name(required), contacts(at least one), message(plain roman character content), and schedule date(has to be future). |
| REQ15 | The system shall allow users to create blast in a single step. |
| REQ16 | After successful creation of a blast, the system shall show a pop-up confirming the creation of the blast and then the user shall be redirected to the blast list page. |

| | |
|---|---|
| REQ17 | The system shall allow users to view all the blasts created by the user ordered by most recently created. |
| REQ18 | The system shall allow users to view blast-specific details by clicking on the blasts shown in the blast list page. |
| REQ19 | The system shall allow users to download blast-specific reports containing information about blast's contacts delivery status. |
| REQ20 | Dashboard shall display a bar graph showing the unique contacts that were reached, unreached and unreachable according to the date range set by the user between which the sms blasts was scheduled. |
| REQ21 | The system should run processes in the background which continuously checks whether the send date of any of the blast is reached or not, and if it is, then should complete the blast at that time by sending out the corresponding message to the contacts of that blast. |
| REQ22 | The system should update the information pertaining to the status of messages received by the contacts of the scheduled blasts. |

Table. 1 Functional Requirements

## 4.2 Non-Functional Requirements

| SrNo | Requirement | Reason |
|---|---|---|
| 1 | Security | It is important that the user login process and overall app is secure so that no misuse of mass messaging takes place. |
| 2 | Maintainability | A low maintenance design is required |
| 3 | Quality | We require a high quality(reliable and low maintenance) application |
| 4 | Readable | It is required that source code is readable as many people work on it. |
| 5 | Testability | The system should be testable to reduce faults. |
| 6 | Scalability | A scalable application is required as the idea is to expand this multi user system to multi tenant. |

Table. 2 Non-Functional Requirements

# 5. Functional Requirements Specification

## 5.1 Use Case Diagram

The main purpose of this diagram is to describe the basic functionalities and show how different types of users will interact with this system. Below is a list of the use cases that will be used in the diagram.

- **U1: Register** ACTOR: Unregistered User
  DESC: Initiated when a user attempts to visit the register module. The user is required to enter these details: first name, last name, username, email, password & re-type password.
  INC: None EXT: None DEP UC: None DEP REQ**:** REQ1
- **U2: Login** ACTOR: Registered User
  DESC: Initiated when a user attempts to do a restricted action in the system. The user is then asked to enter in their registered username and it's corresponding password to log into the system.
  INC: None EXT: None DEP UC: U1 DEP REQ: REQ3
- **U3: Manage Contacts** ACTOR: Registered User
  DESC: Allows users to manage contacts by uploading new contacts, viewing contact list, & editing specific-contacts.
  INC: U4, U6, U7 EXT: None
  DEP UC: U2 DEP REQ**:** REQ9, REQ11, REQ12
- **U4: Import Contacts** ACTOR: Registered User
  DESC: Allows users to upload contacts files (csv, xls, xlsx) in the following column format - phone_number, first_name, last_name.
  INC: U5 EXT: None DEP UC: U2 DEP REQ**:** REQ9
- **U5: View Summary of Uploaded contacts** ACTOR: Registered User
  DESC: Allows users to view the summary of uploaded contacts which have - No of contacts created, No of contacts invalid (due to invalid mobile no.) and No. of duplicate contacts.
  INC: None EXT: None DEP UC: U2, U4 DEP REQ**:** REQ10

- **U6: View Contact List** ACTOR: Registered User

  DESC: Allows the user to view a list of contacts already added by the user. All the contacts are listed on different pages (pagination) with each contact including its first name, last name, Contact, Added on, & Modified on. List should be sorted by the modification date.

  INC: U3 EXT: None DEP UC: U2 DEP REQ: REQ11
- **U7: Edit Contacts** ACTOR: Registered User

  DESC: Allows users to edit contacts first name and last name.

  INC: None EXT: None DEP UC: U2, U4 DEP REQ**:** REQ12
- **U8: Manage Blasts** ACTOR: Registered User

  DESC: Allows users to manage blasts by creating new blasts, browsing the blast list, view blast specific details and downloading reports containing blast details.

  INC: U9, U10, U11, U12 EXT: None

  DEP UC: U2 DEP REQ: RE14, REQ17, REQ18, REQ19
- **U9: Create Blasts** ACTOR: Registered User

  DESC: Allows users to create blasts which includes - Name of the blast, selecting contacts, Message to deliver and scheduled date.

  INC: None EXT: None DEP UC: U2 DEP REQ: REQ14
- **U10: View Blast List** ACTOR: Registered User

  DESC: Allows users to view blast list which includes -Name, Send date, no. of Recipients: total contacts in blast, no. of contacts attempted: reached + not reached, no. of contacts Received: reached, View Details option, Download Report option.

  INC: None EXT: None DEP UC: U2 DEP REQ: REQ17
- **U11: Download Report** ACTOR: Registered User

  DESC: Allows users to download specific blast reports locally.

  INC: None EXT: None DEP UC: U2, U9 DEP REQ: REQ19
- **U12: View Blast-specific details** ACTOR: Registered User

  DESC: Allows users to view blast specific details which includes- blast name, selected contacts, Message to deliver and scheduled date. These fields will be read only.

  INC: None EXT: None DEP UC: U2, U9 DEP REQ: REQ18

- **U13: View Dashboard** ACTOR: Registered User

  DESC: Allows users to view analytics of blasts on dashboard which includes graph consisting of the date filter showing the number of unique contacts reached.

  INC: None EXT: None DEP UC: U2 DEP REQ: REQ20

- **U14: Logout** ACTOR: Registered User

  DESC: Allows users to log out from the system.

  INC: None EXT: None DEP UC: U2 DEP REQ: REQ6

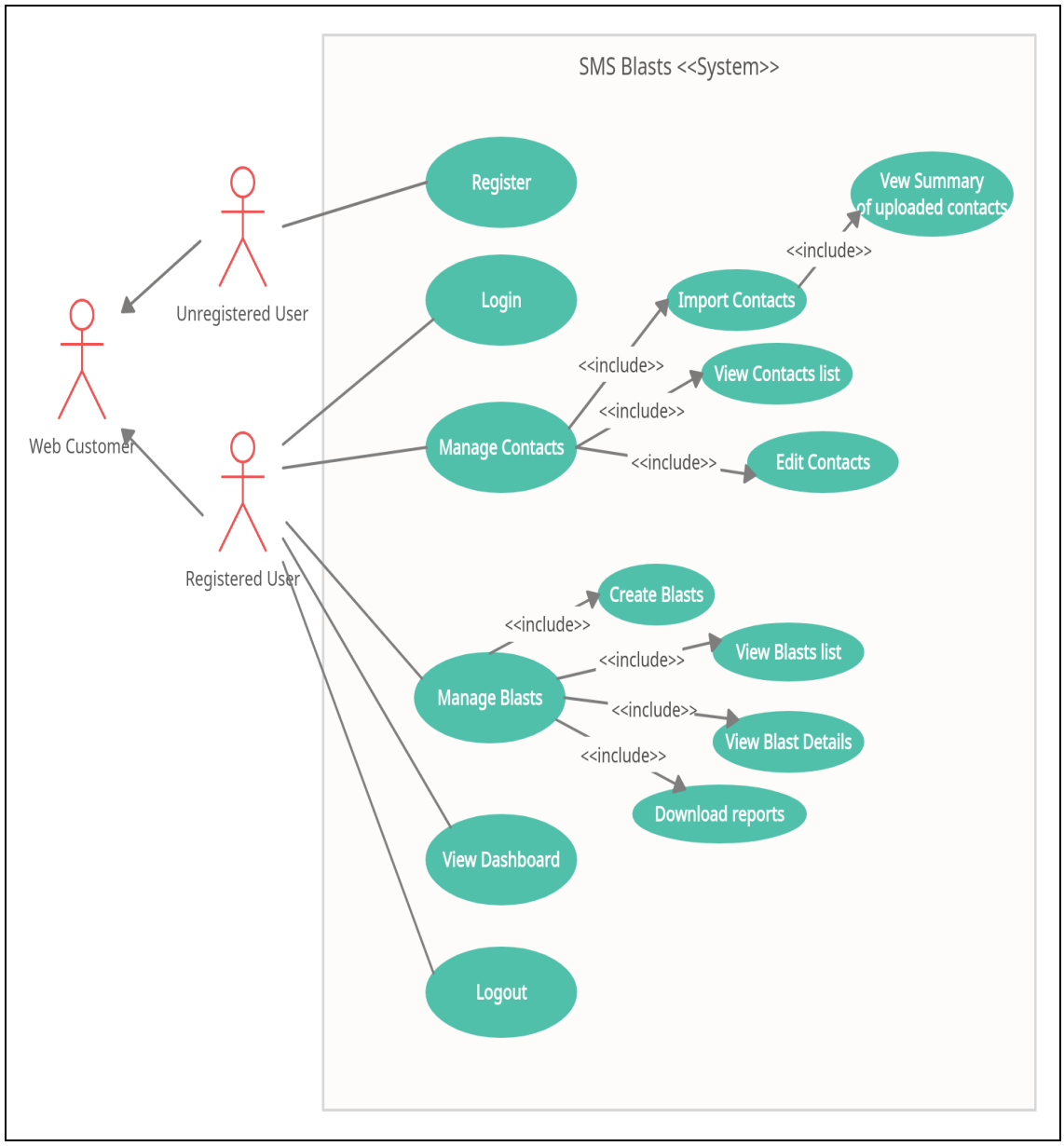



Fig. 1 Use Case Diagram

# 6. System Architecture and System Design

## 6.1 Architecture Diagram

Our architecture is organised in the following manner:

1) Django Application: It is a python web framework that we have used to make our REST APIs. It supports rapid development of the web application. This application also takes care of the communication with database and client frontend. Django has a built-in webserver, but it is clearly written in django documentation that this server should only be used for development purposes. The built-in web server is not designed for security or efficiency.
2) Angular Application: The Angular Application has its own standalone server, but it doesn't have a number of advanced features like load balancing, security, and efficiency.
3) uWSGI: uWSGI is a full fledged http server that implements WSGI spec and converts the http requests into python objects which call the django application as a function. The reason we need WSGI is because it allows any web server to work with any python application
4) NGINX: It is a web server and we are using it as a reverse proxy in our architecture. It provides load-balancing and security to the whole application. Nginx is also very fast with delivering static files. Overall it provides a good performance to the system.

For the purposes of making our application available via the internet without buying the domain name, we have used ngrok.

5) NGROK: It consists of 2 parts - ngrok client and server. It works by using the tunnelling mechanism. The server is situated outside of the firewall, while the client program is installed on the server machine.

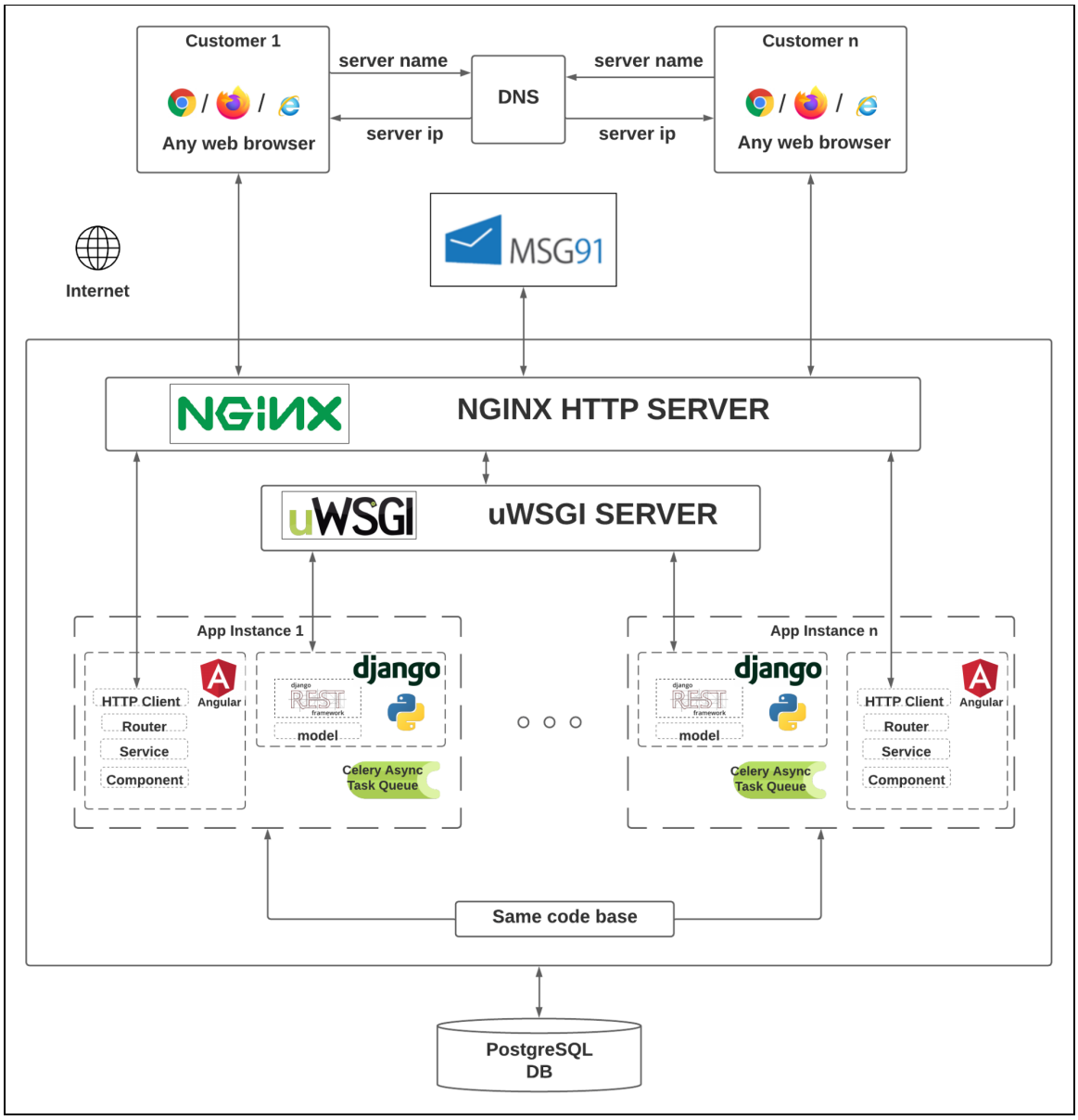



Fig 2: Architecture Diagram

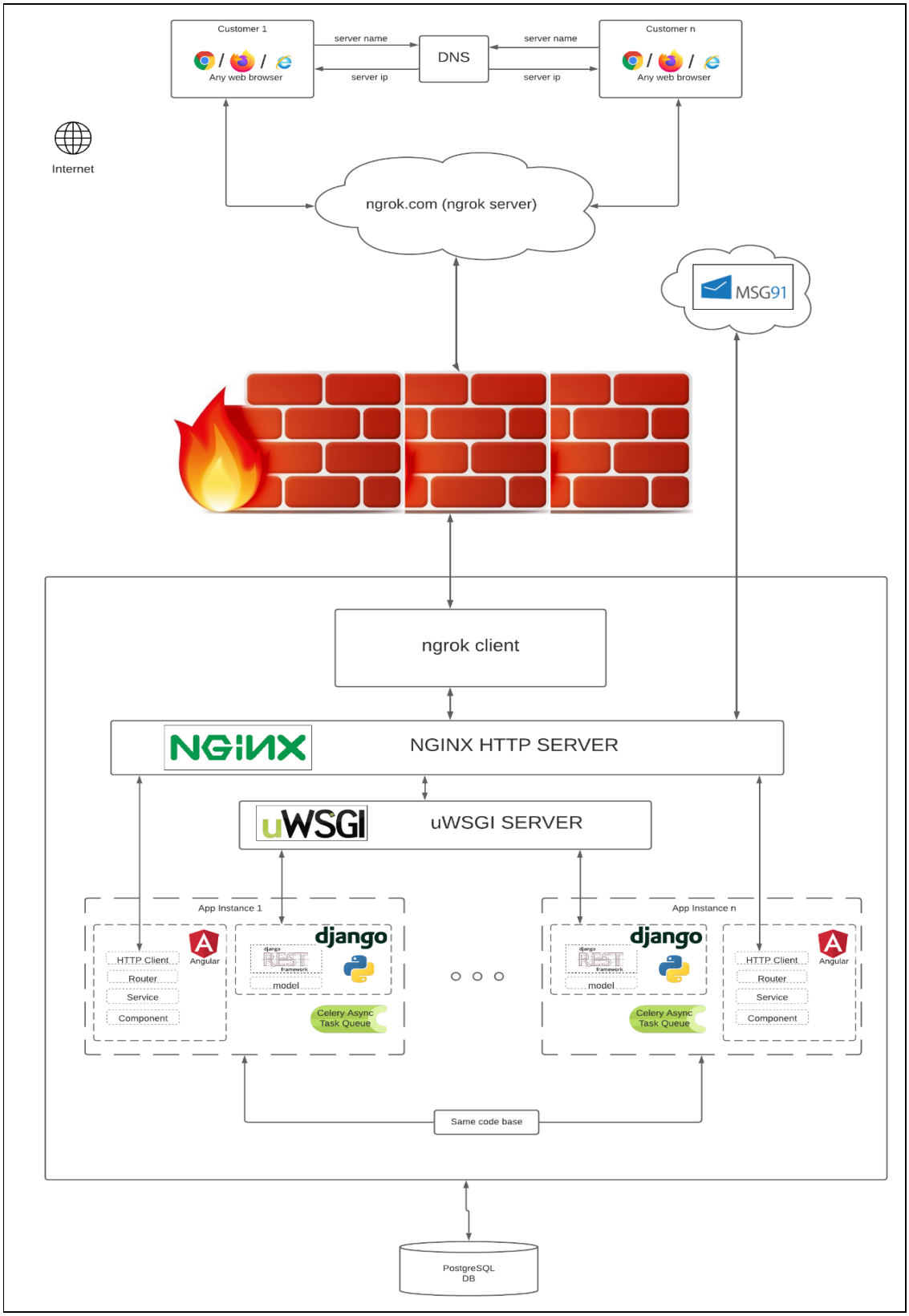



Fig 3: Architecture Diagram with Deployment

## 6.2 System Design diagram

The figure attached below shows the high-level design of all the important components of the system and their connections. The end-user makes a request to the system's web url and the request is passed to the DNS block, which provides the public IP address associated with the domain name in the url. Then, the request is forwarded to the Nginx web server, running as a reverse proxy. Nginx handles all static requests itself, hence it is directly connected to the angular application (frontend framework). However, since it cannot handle non-static requests, NGINX forwards these requests to uWSGI using a unix socket. uWSGI converts the http call into a python object and calls the django application as a function and passes the python object. All the APIs defined for various use-cases interact with the Postgres relational database for fetching, adding, or updating details. Furthermore, this system consists of some periodic tasks which are to be performed by Celery asynchronously using queues and workers. The internal working of Celery can easily be stated as the Producer/Consumer model. Producers place the jobs in a queue, and consumers are ready for them from the queue. In this case, Celery beat service acts as a Producer, and Celery workers act as consumers.  Redis, which is a performant, in memory, key-value data store, is used as a message broker queue and hence, is used to store messages added by the celery beat service. Celery beat service works as a scheduler that adds periodic tasks at their mentioned intervals into the Redis queue. Celery workers are listening to the queue head, so that whenever a task is published, they can consume and execute it. They also interact with the database as well as to perform updation, creation or deletion of objects. Since, this system's primary functionality is to send messages at a specified time, hence, the workers are given tasks at that particular time to send messages to last mile users using the third party application called Msg91. Moreover, Celery Worker service is responsible for creating the configurable number of workers and managing them.

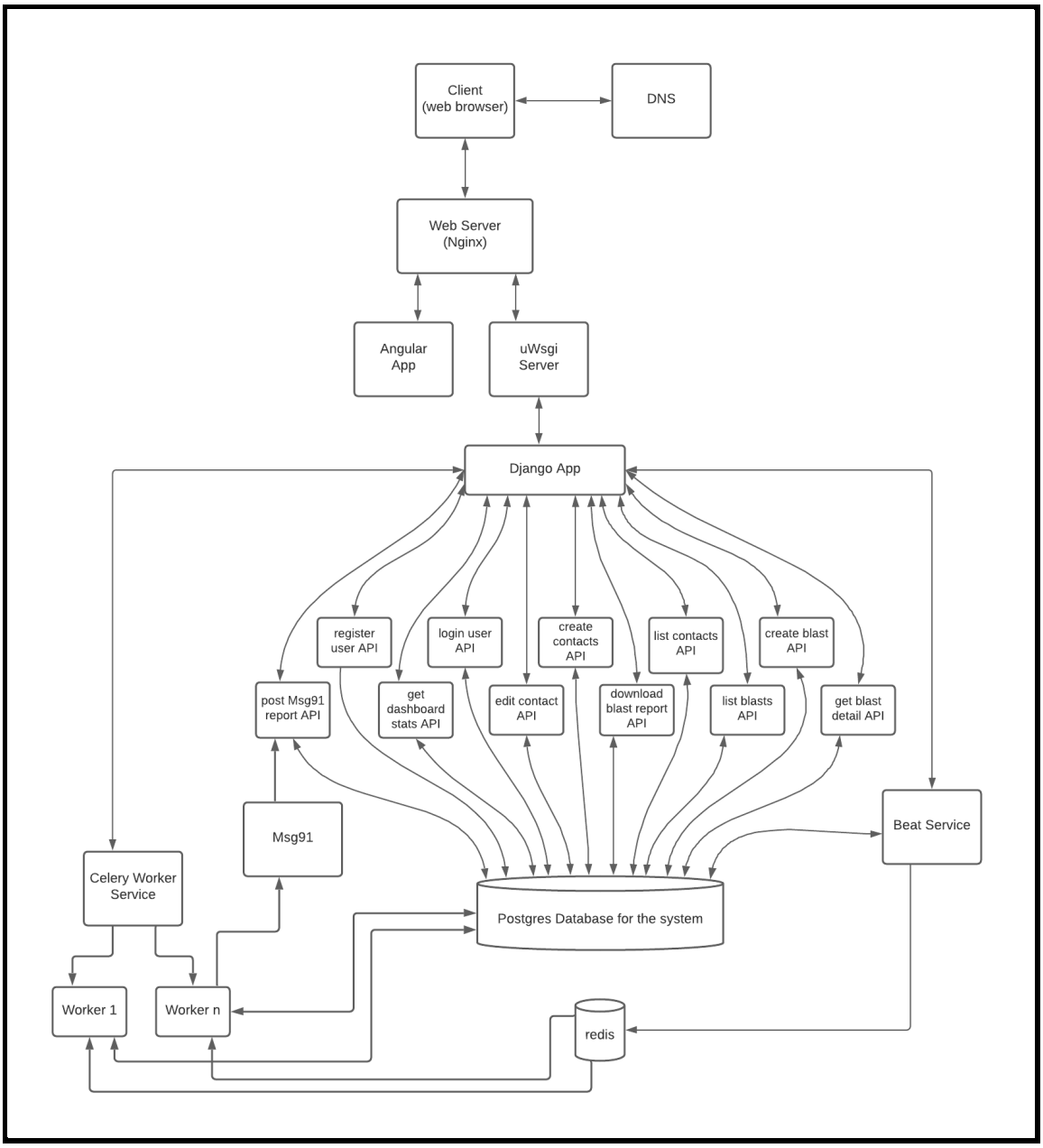



Fig 4: System Design Diagram

## 6.3 Overview of Modules

This project is divided into four major modules, namely, Login/Registration module, Contacts Module, Blasts Module and Dashboard Module.

1. **Login-Registration Module**

**Registration sub module** handles registering a new (unregistered) user into the system. It requires the following fields:

- First name  (condition: required, max 120 characters long)
- Last name  (condition: required, max 120 characters long)
- username  (condition: unique, required, max 120 characters long)
- Email ID (condition: required, Valid and unregistered)
- Password (condition: required, min 8 characters long)
- Confirm Password (condition: required, same as Password field)

Appropriate error messages are displayed if any of the field validations fails.

**Login sub module** handles authenticating a registered user using the following fields-

- Username (condition: username is registered)
- Password (condition: password corresponding to the registered username)

If a user is not logged in, the root URL redirects to the login page where username and password are accepted. Once logged in, the user has the option to logout from the top right nav bar and are redirected to the login screen. If a user fails to login with correct credentials, the user is displayed a corresponding error message.

2. **Contact Module**

Contacts module gives users the flexibility to create or import contact lists in the form of excel or CSV in order to use it for messaging purposes. This module shows a list of already imported contacts ordered according to the latest creation time. The content of contacts include the following fields:

First Name, Last Name, Phone Number, Added On, Last Modified, Edit option.

In addition, the upload contacts functionality only adds new contacts to the existing list instead of overwriting existing contact list. After the user has successfully uploaded contacts, list view refreshes with newly added contacts displayed. In addition, the system also shows a summary response after each upload which includes: no. of created/valid contacts, no. of invalid contacts(due to invalid phone number), no. of duplicate contacts.

3. **Blast Module**

Blasts Module provides the functionality of sending blast messages to user-selected contacts with a user defined message on a scheduled time. This module shows a list of all the blasts ordered by the latest creation time. Content of the blasts list includes the following fields-

Name, Send date, Recipients: total contacts in blast, Attempted: contacts reached + contacts not reached, Received: contacts reached, View Details: displays entered info in read only view, Download Report: should generate CSV synchronously and prompt user to save Locally.

The process of creation of blasts happens in a single session, meaning, there are no drafts of the blasts saved if the blast creation is partially completed. The contents of the creation of blast includes the following fields -

- Name (condition: required field)
- Contact picker to select contacts and text field to display selected contacts as in AD2.
  (condition: At least one contact must be selected, picker should allow selection to be made across pages of contacts in table. )
- Text editor with to compose plain Roman character content (condition: required)
- Schedule for future date option (condition: required)
- Send button (condition: required, On clicking the send button, the user is given confirmation that blast was successfully created and redirected to the page where blasts list is displayed. )

4. **Dashboard Module**

This Module populates an analytics dashboard with blast-specific charts with aggregated results. The graph consists of a date filter which specifies start and end date and this range is used to filter out the analytics of blasts that fall within that range and consists of displaying unique contacts count information within 3 categories - reached, unreached and unreachable. The information is filtered out from Attempt Model which is created while scheduling the blasts.

## 6.3 Block diagrams of modules

### 6.3.1 Account Module Block Diagram

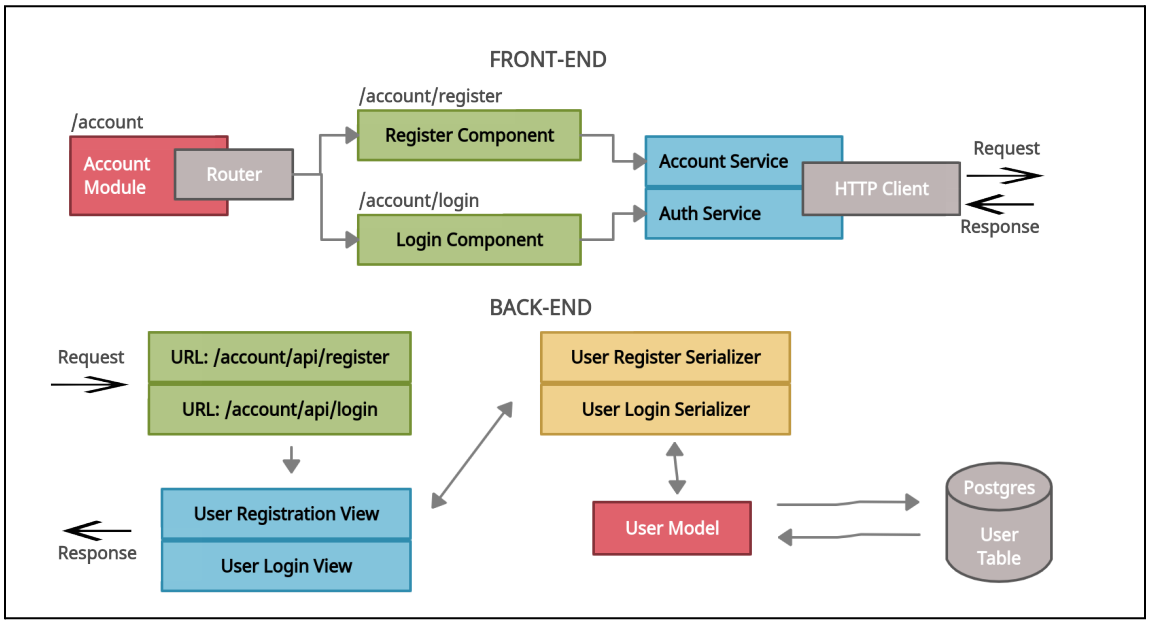



Fig 5: Block Diagram For Account Module

### 6.3.2 Contact Module Block Diagram

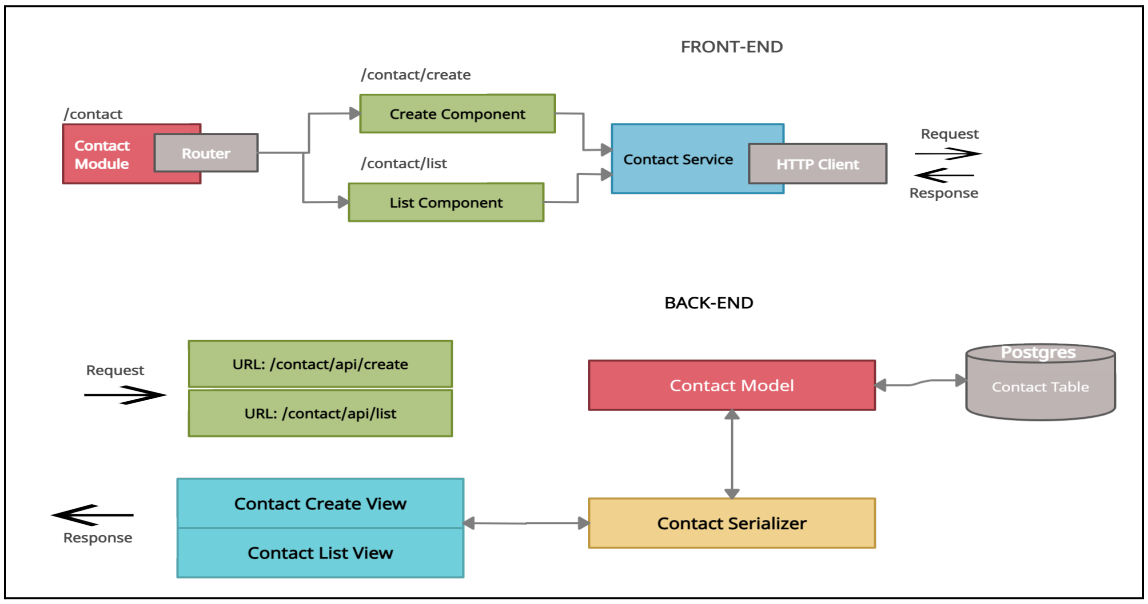



Fig 6: Block Diagram For Contact Module

### 6.3.3 Blast Module Block Diagram

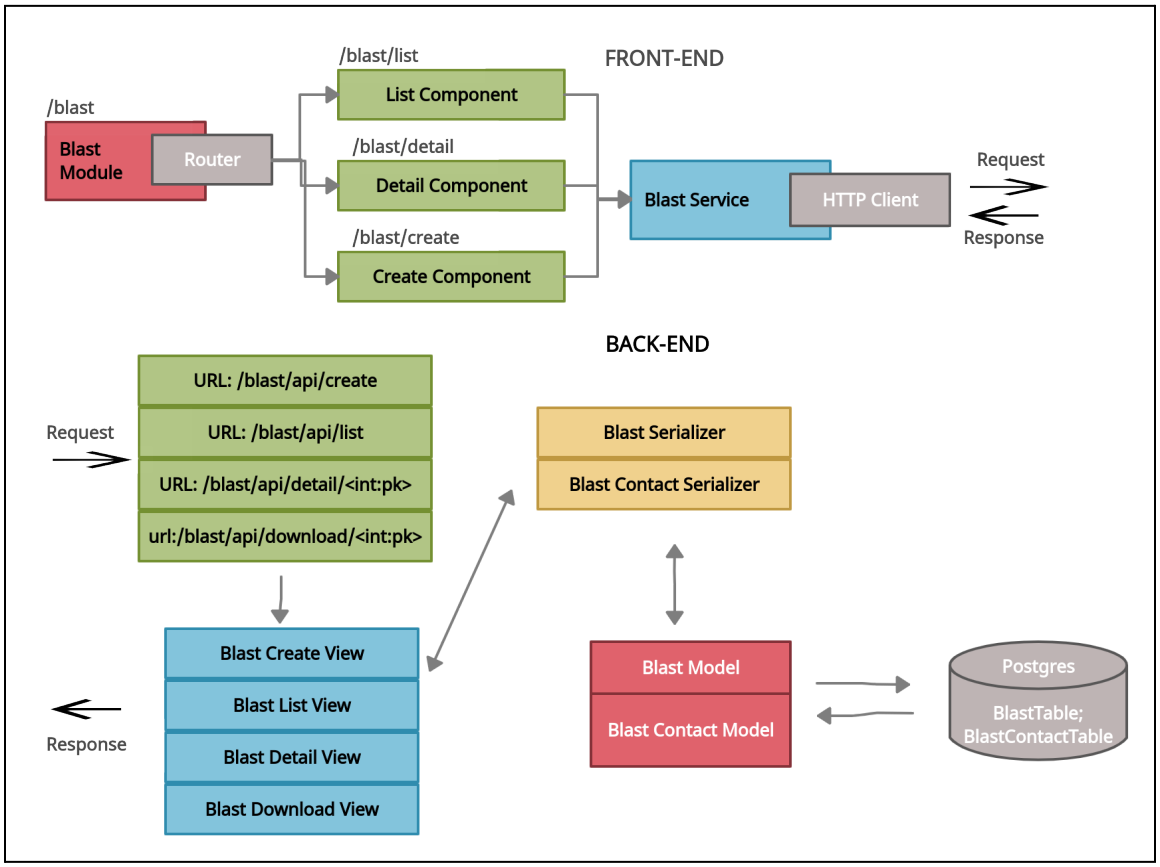



Fig 7: Block Diagram For Blast Module

### 6.3.4 Dashboard Module Block Diagram

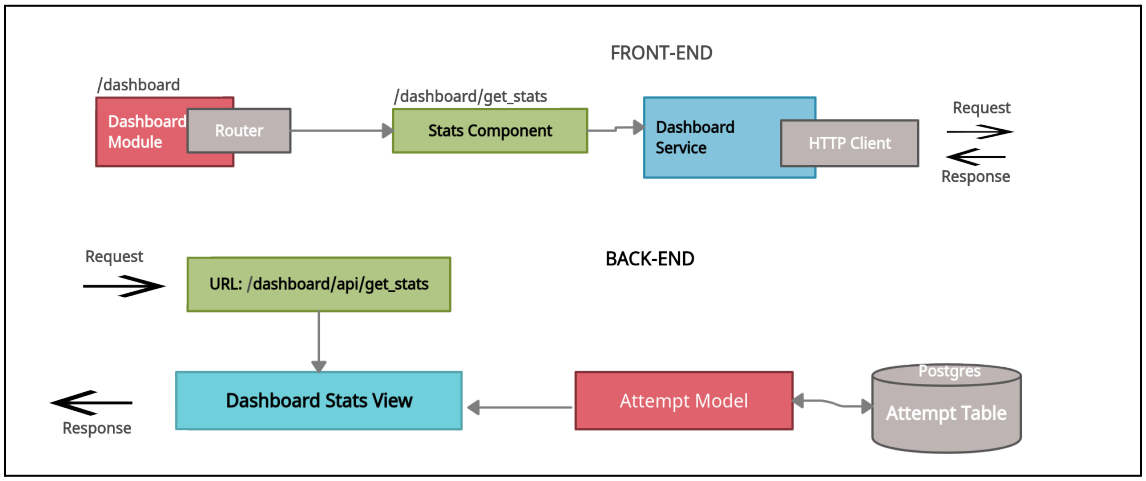



Fig 8: Block Diagram For Dashboard Module

## 6.4 ER Diagram

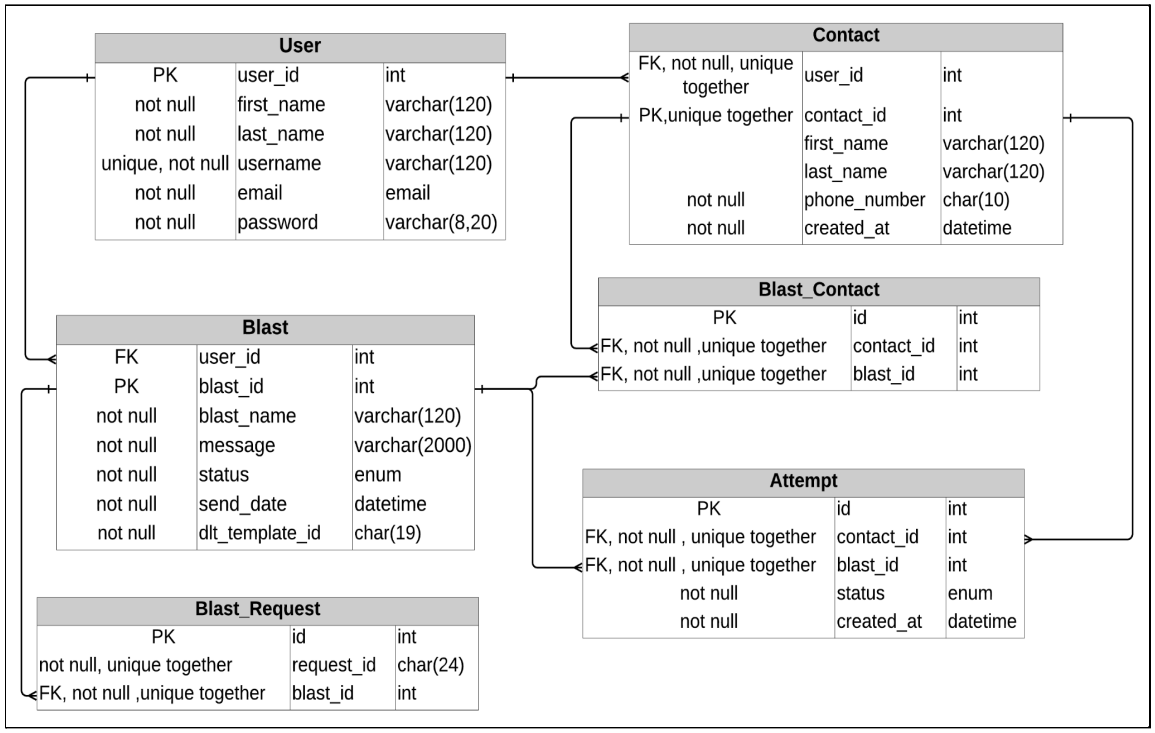



Fig 9: Physical Entity Relationship Diagram

## 6.5 Data Flow Diagrams

Data Dictionary for all the levels of DFDs:

| Data flow | Dictionary | Data flow | Dictionary |
|---|---|---|---|
| login_details | {username, password} | updated_contact_info | {phone_no,first_name,last_name} |
| user_details | {first_name,last_name,email,password,username,retype password} | contact_upload_summary | {valid,invalid,duplicate} |
| contact_upload_files | excel/csv file | blast_list | {count,next,prev,[{blast_obj},...]} |
| parameters_for_analytics | {report_type,start_date,end_date} | blast_details | {name,message,dlt_te_id,send_on,contacts} |
| blast_info | {id,name,message,dlt_te_id,send_on,contacts} | blast_report | csv file |
| blast_id | id | blast_specific_analytics | {reached,unreached,unreachable} |
| blast | Blast_obj = {id,name,message,dlt_te_id,send_on,contacts} | error_msg | invalid_user_details + invalid_login_details + incorrect_credentials + invalid_parameters + invalid_blast_details + is_not_owner + blast_report_ not_ready + contact_file_error |
| contact_list | {count,next,prev,[{phone_no,firat_name,last_name}...]} | | |

### 6.5.1 L0 DFD

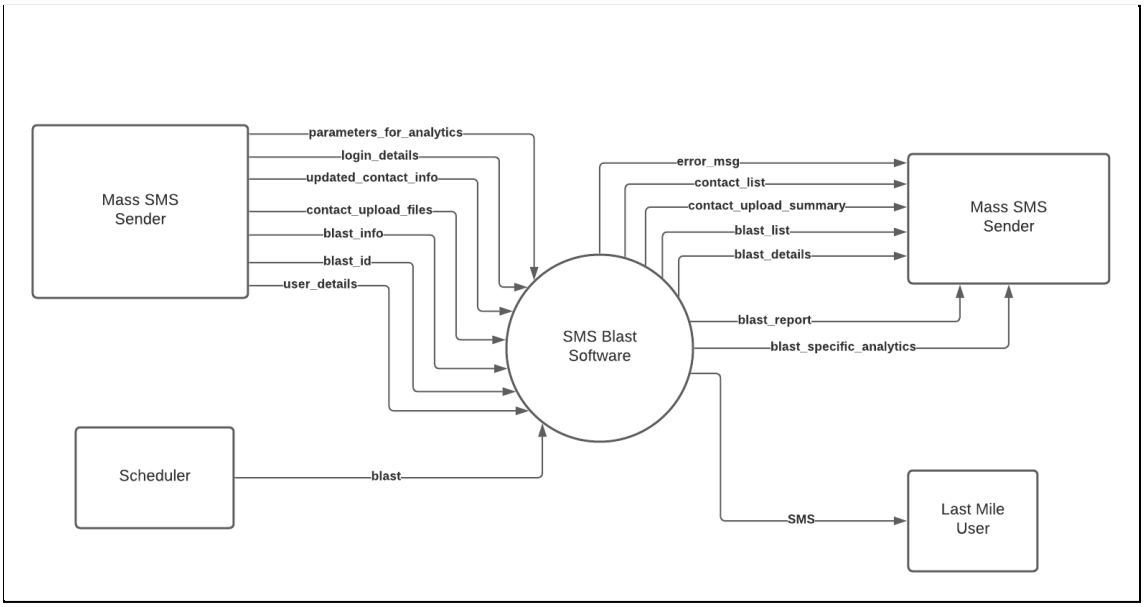



Fig. 10: Level 0 Data Flow Diagram

## 6.5.2 L1 DFD

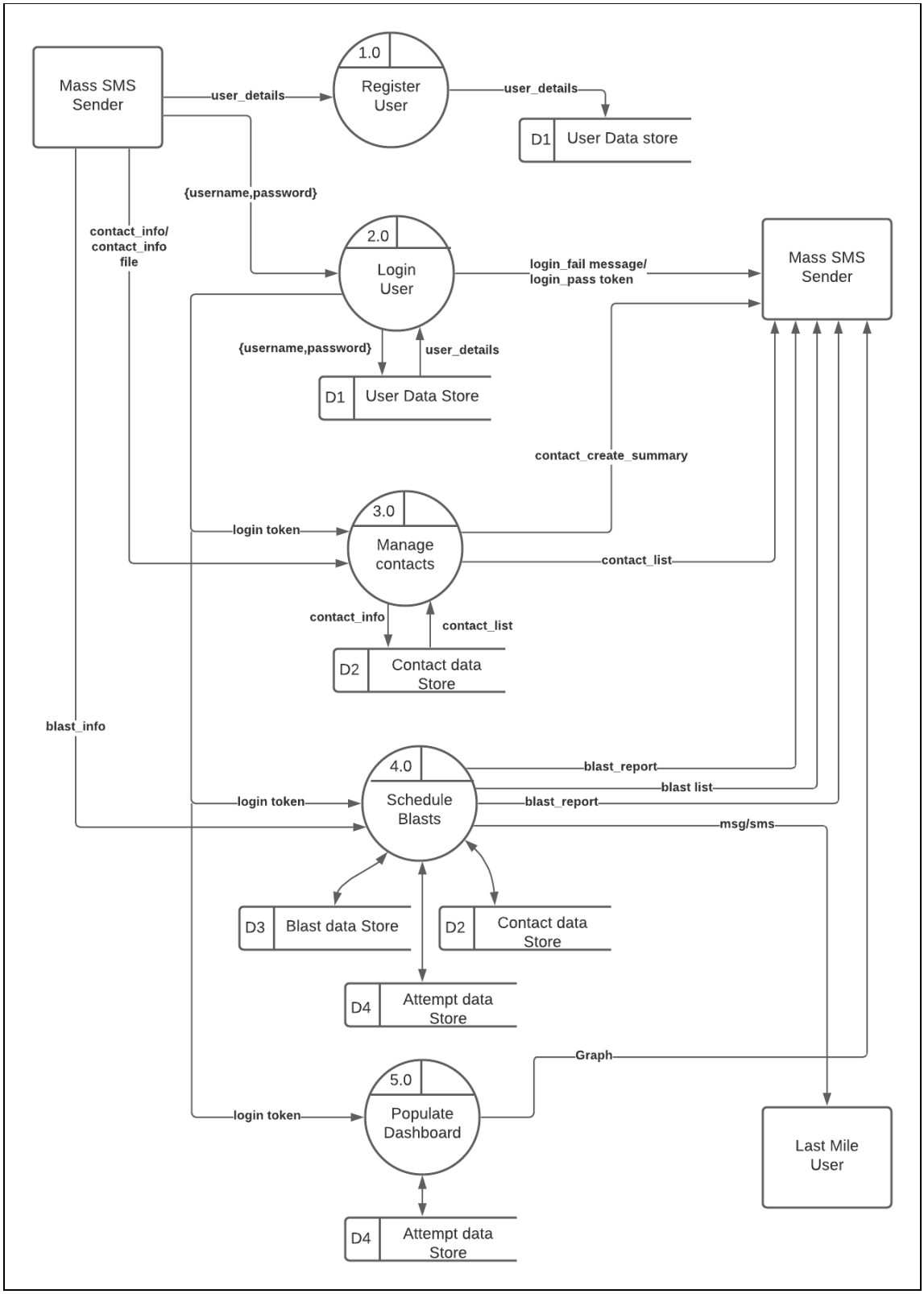



Fig. 11: Level 1 Data Flow Diagram

## 6.5.3 L2 DFD

### Login and Register Module

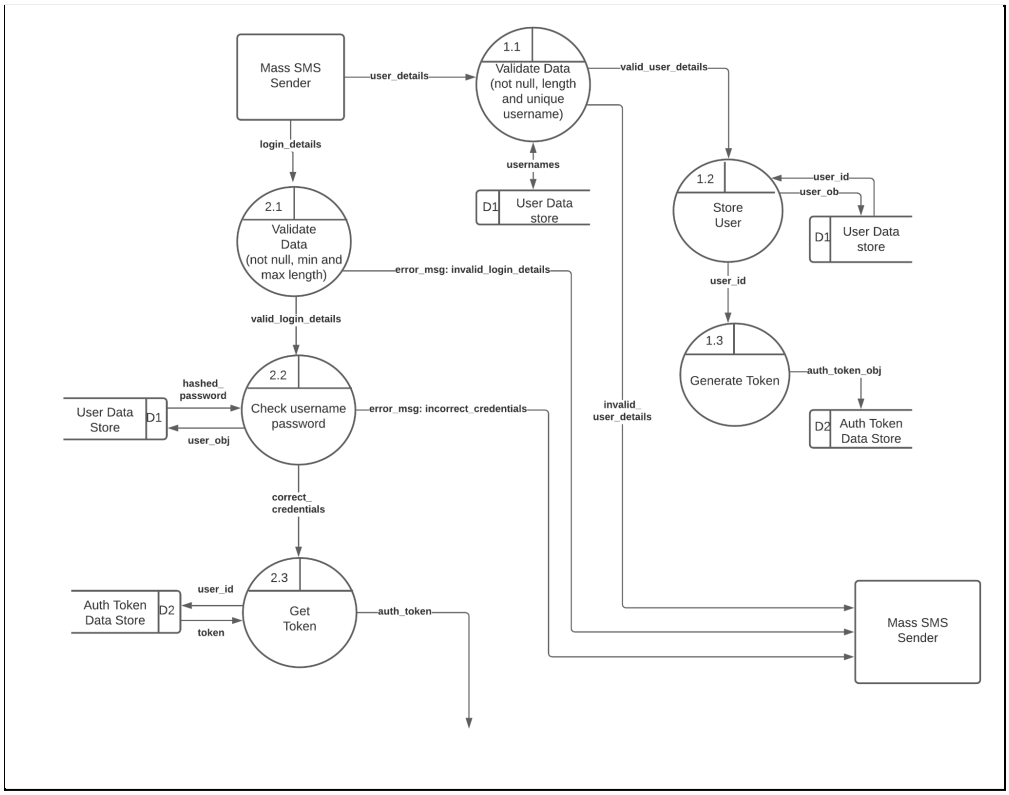



Fig. 12: Level 2 Data Flow Diagram for Login and Register Module

### Scheduling Module

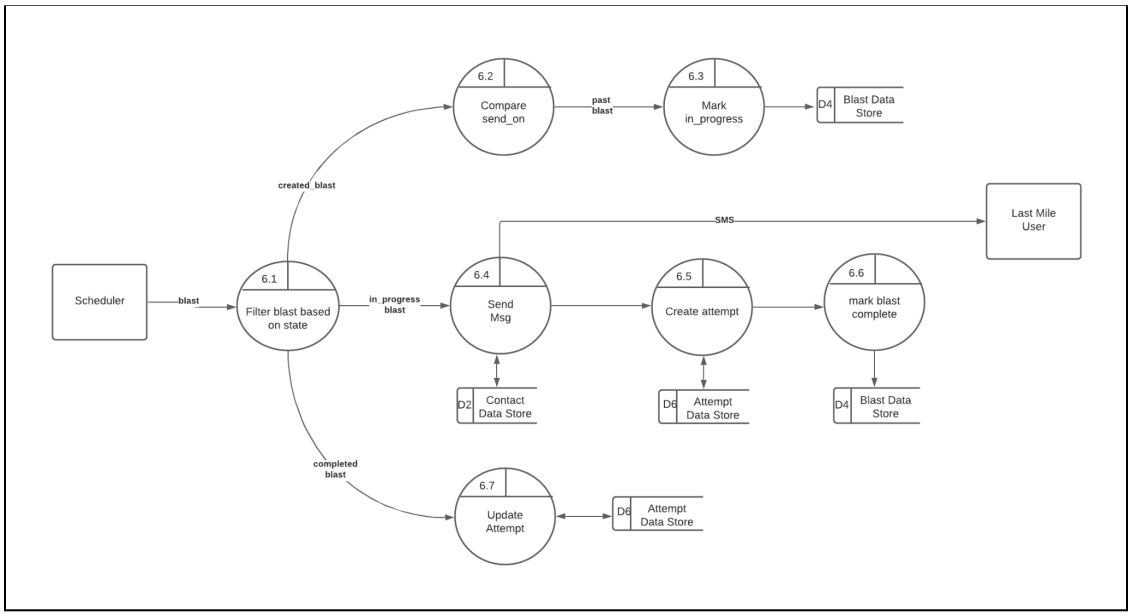



Fig. 13: Level 2 Data Flow Diagram for Scheduling Module

## Contact Module

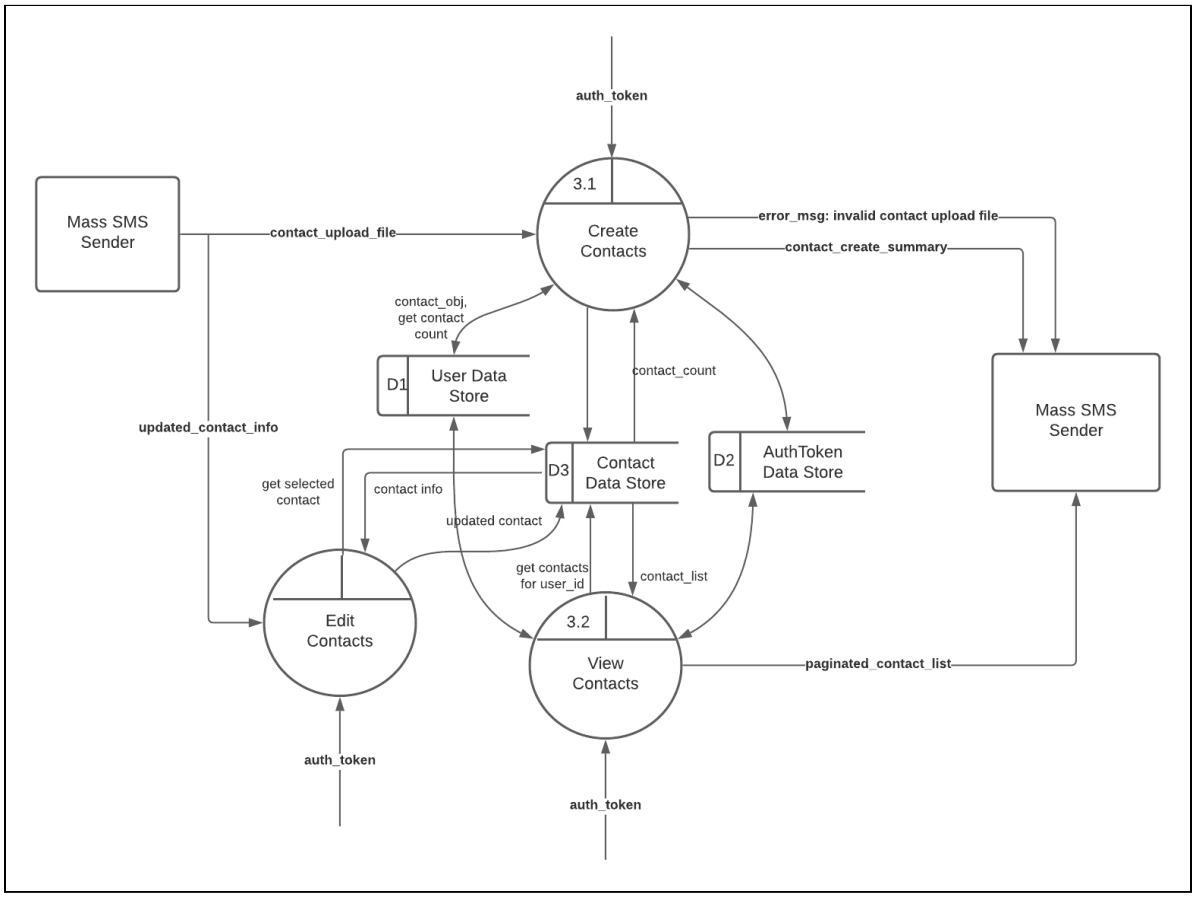



Fig. 14 Level 2 Data Flow Diagram for Contact Module

## Dashboard Module

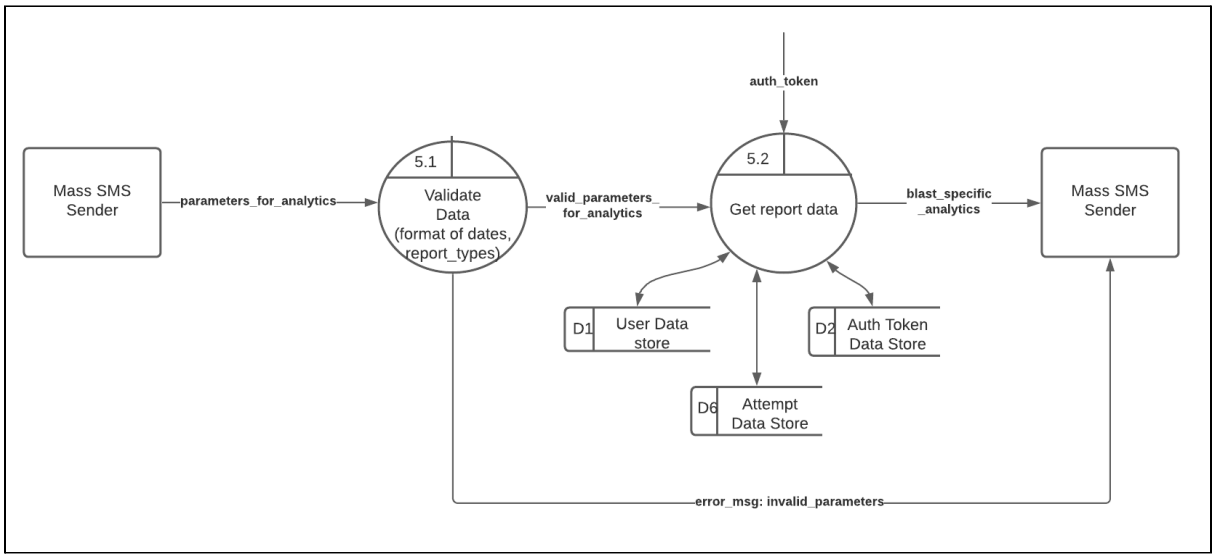



Fig. 15 Level 2 Data Flow Diagram for Dashboard Module

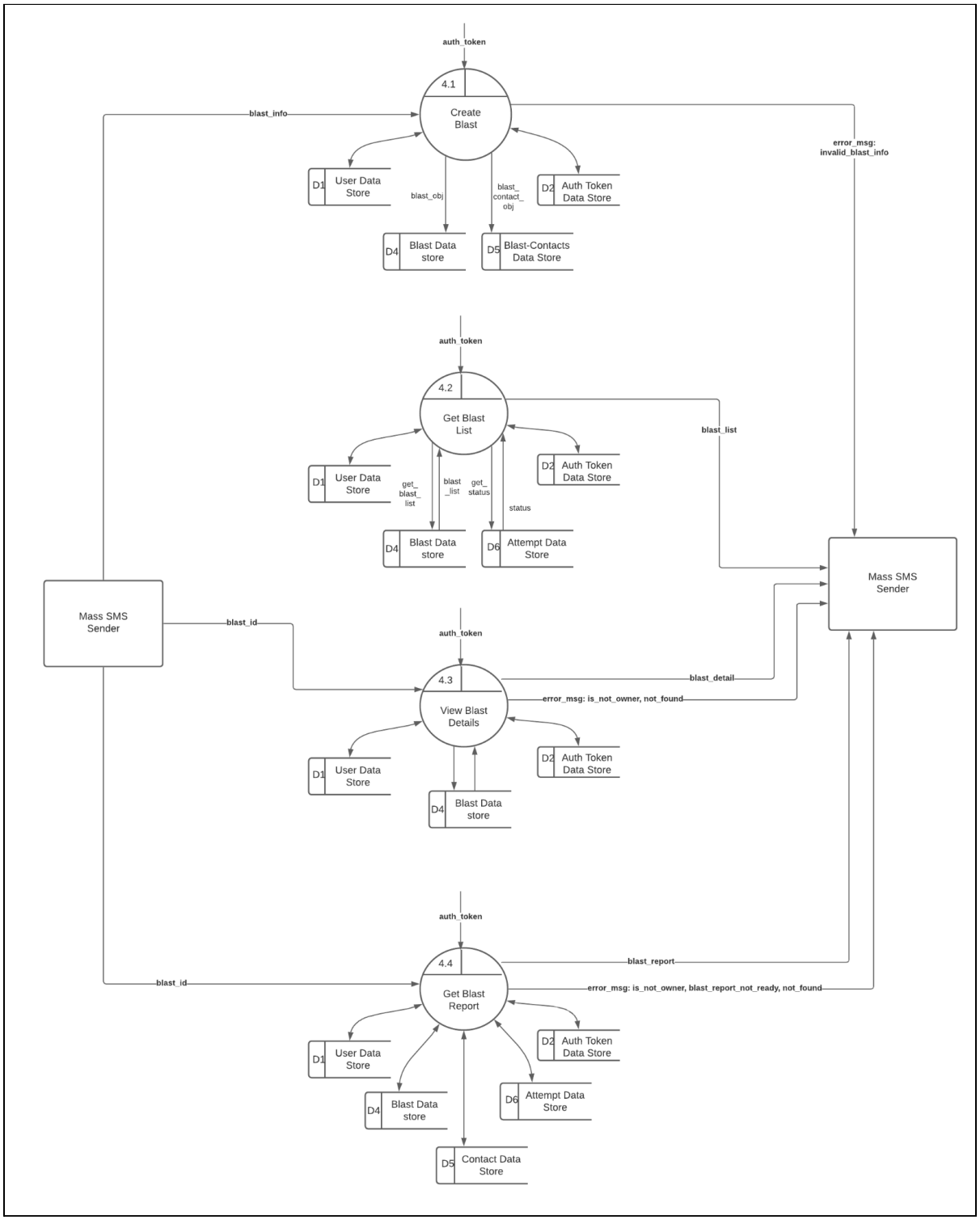



Fig. 16 Level 2 Data Flow Diagram for Blast Module

## 6.5.4 L3 DFD

### Contact Module

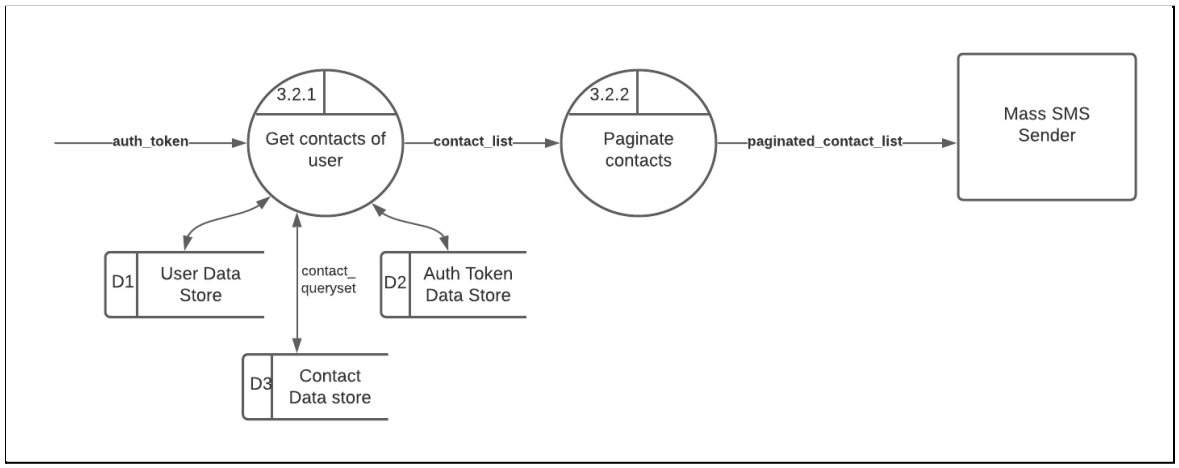



Fig. 17 Level 3 Data Flow Diagram for Contact List

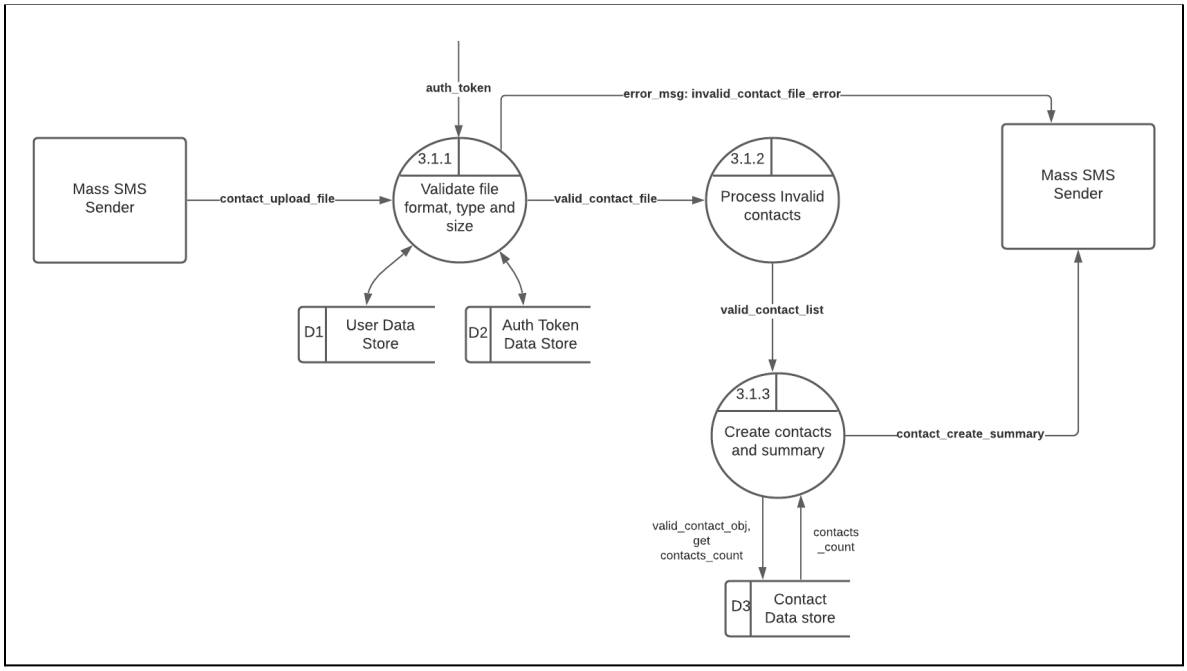



Fig. 18 Level 3 Data Flow Diagram for Contact Create

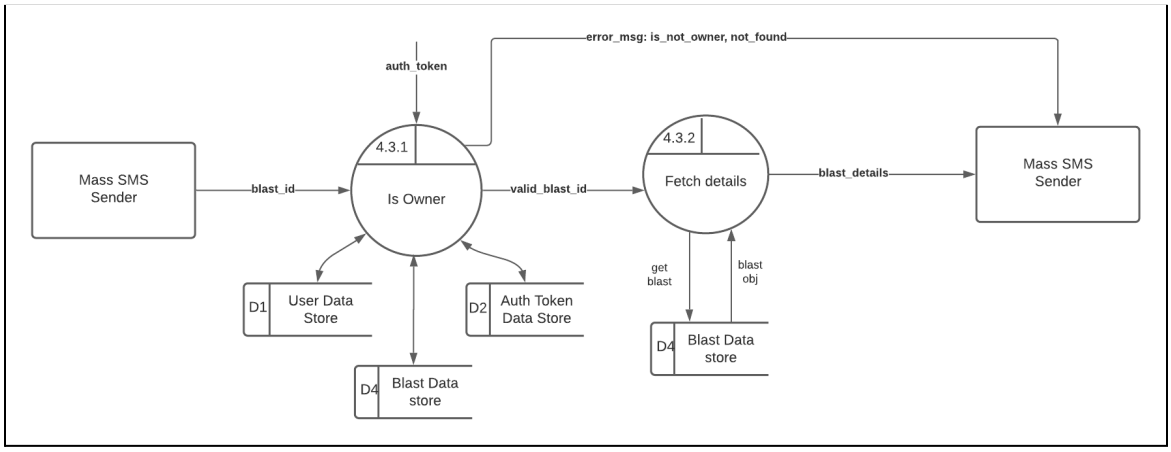



Fig. 19 Level 3 Data Flow Diagram for Blast Detail

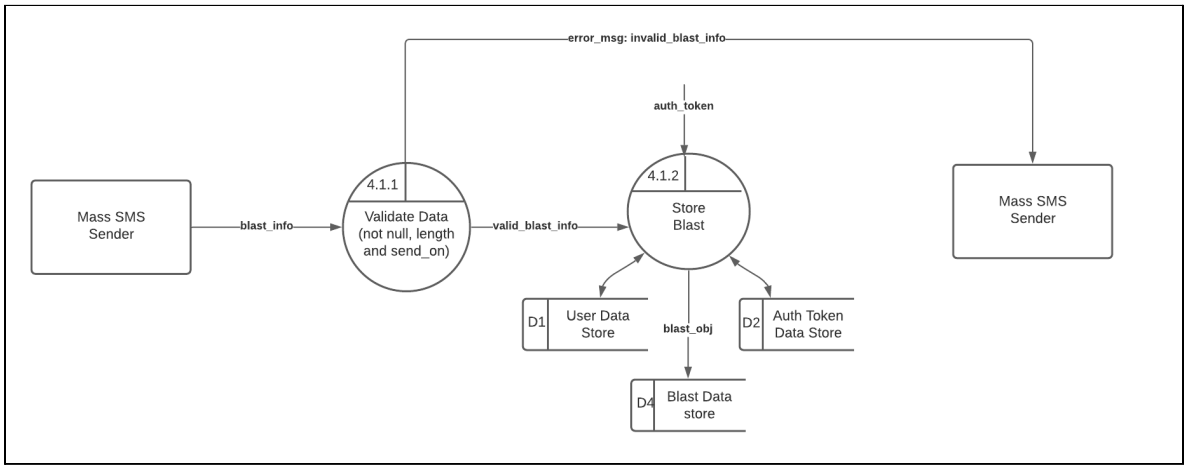



Fig. 20 Level 3 Data Flow Diagram for Blast Create

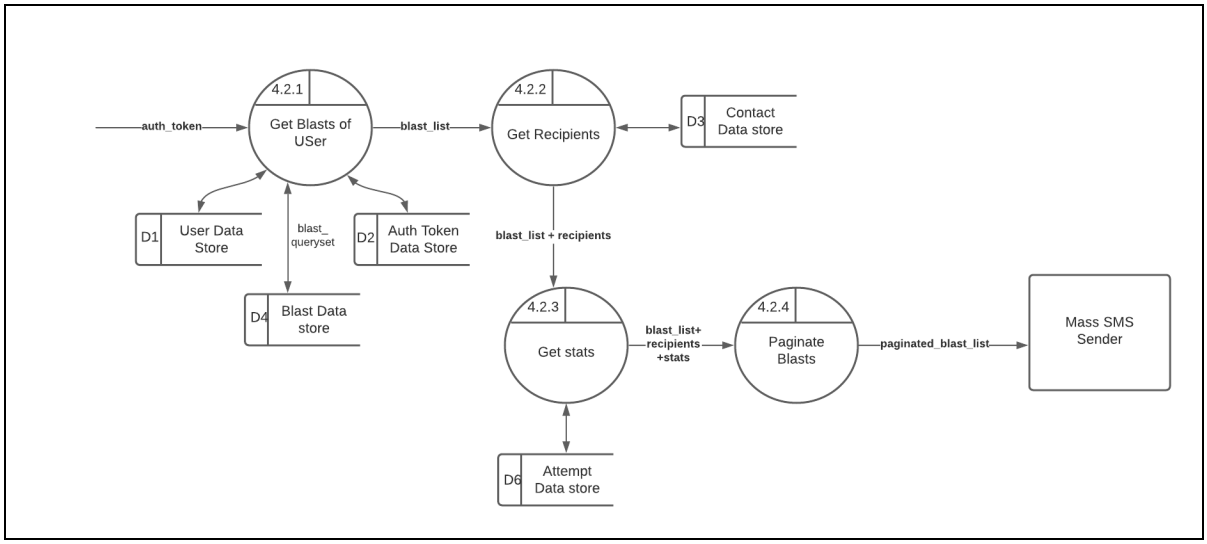



Fig. 21 Level 3 Data Flow Diagram for Blast List

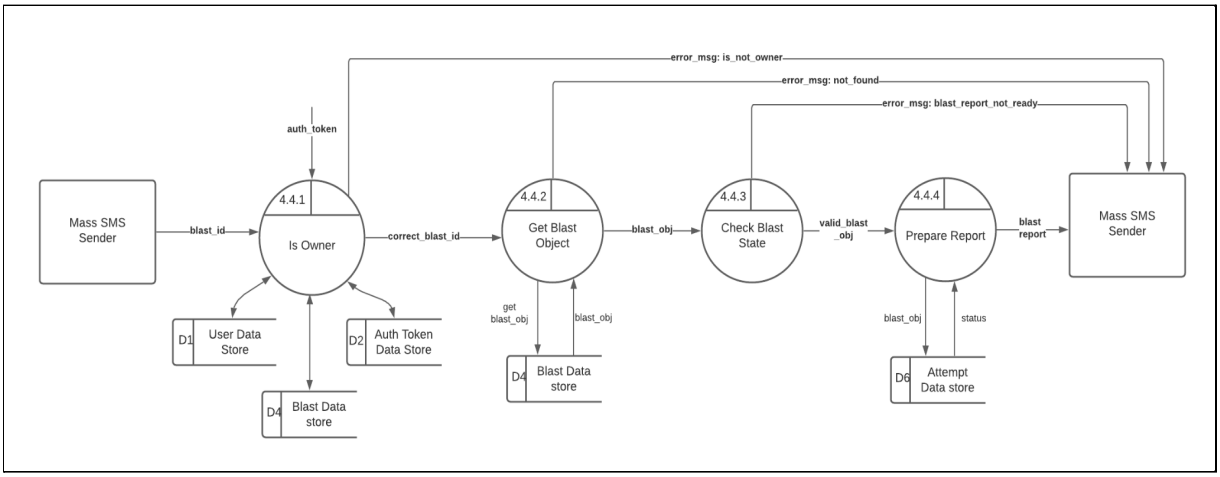



Fig. 22 Level 3 Data Flow Diagram for Blast Report

## 6.6 Sequence Diagrams

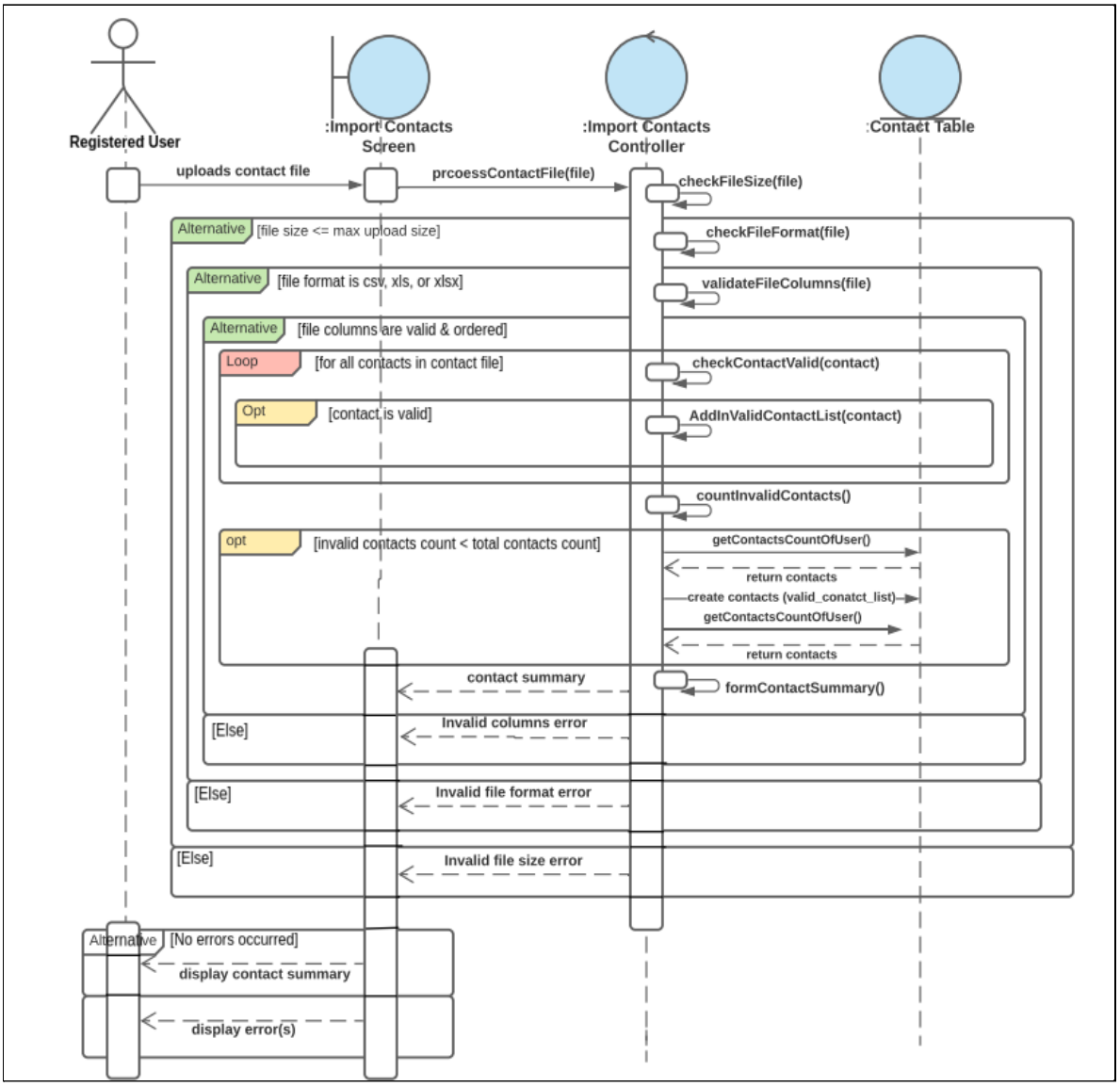



Fig. 23 Sequence Diagram for Imports Contacts

For clear visibility, please click on this link to view the above diagram

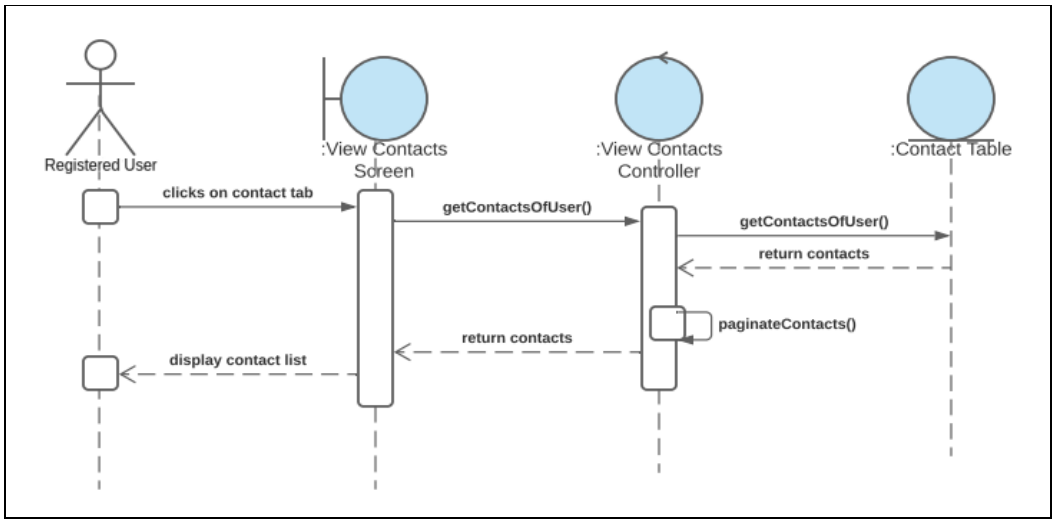



Fig. 24 Sequence Diagram for View Contacts
For clear visibility, please click on this link to view the above diagram

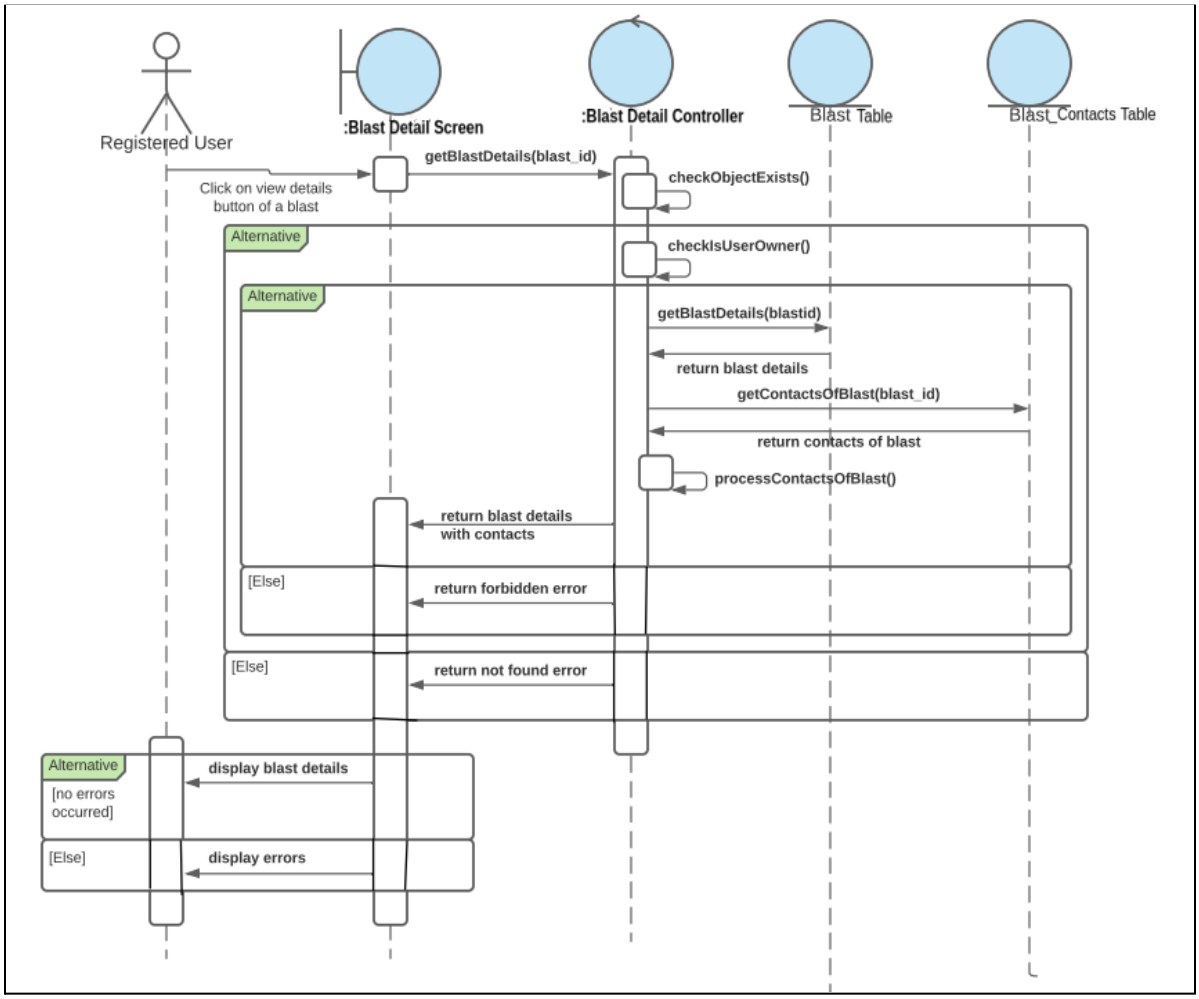



Fig. 25 Sequence Diagram for Blast Details.
For clear visibility, please click on this link to view the above diagram.

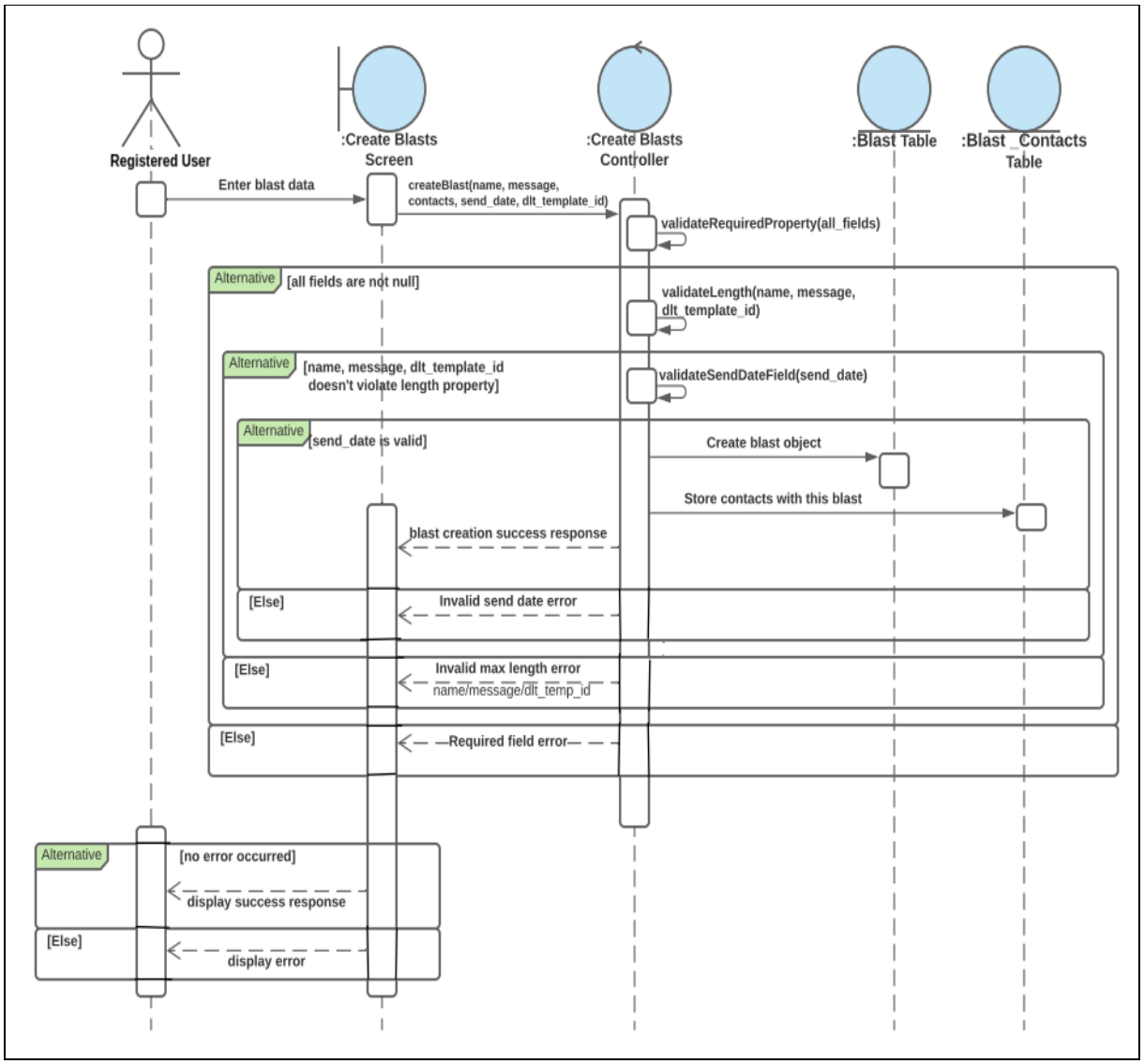



Fig. 26 Sequence Diagram for Blast Create

For clear visibility, please click on this link to view the above diagram

## 6.7 Asynchronous Task Design Content

This project consists of running asynchronous tasks using Celery (with redis server) along with Django. Celery is a python based asynchronous task queue. The need for celery is reflected in the use case where this system starts requesting our provider (which is Msg91) to send attempts to schedule and complete blasts. This poses a need to configure a few asynchronous tasks for scheduling the blasts in celery.

Each Blast goes through 3 main stages which are shown using the finite states below -

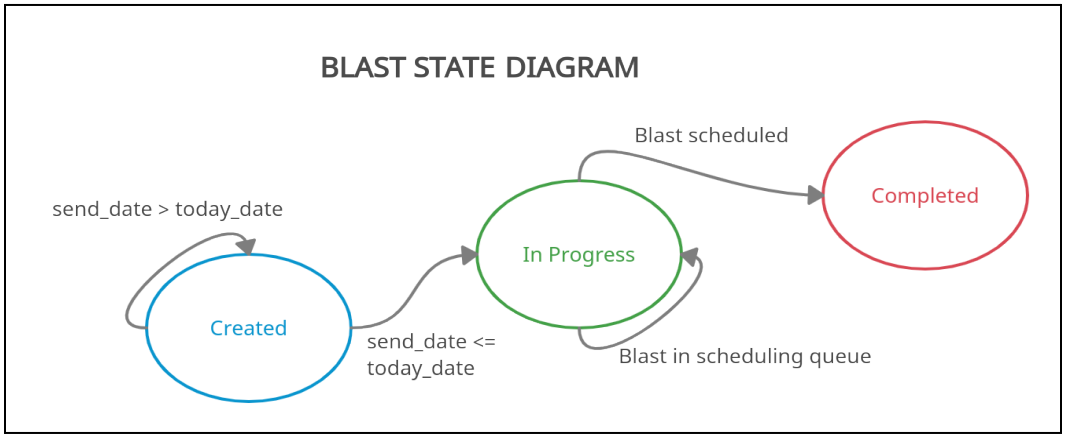



Fig. 27 Blast State Diagram

**Transitions**

**1. Created** to **In Progress** - When the scheduling date of that blast is reached.

**2. In Progress** to **Completed** - When that blast waiting in the scheduling queue is successfully completed by making an attempt to send the particular message to that blast's contacts.

Blast Model has a particular field called 'status' which takes in the above mentioned 3 states with the help of Django's FSMField. Django FSMField is used for managing the state transitions in complex projects with transition decorator.

The 3 major tasks that help in state transitions for blast module which are executed by celery continuously are:

**Task 1:** Prepare Blasts for Scheduling

**Description**: This task filters all blasts whose status is ‘created’ and send date has just passed, and then marks those blast’s status to ‘in progress’ using transition method ‘mark_blast_in_progress’ so that they go in the scheduling queue.

**Task 2:** Schedule Blasts
**Description**: This task filters all blasts whose status is ‘in_progress’ and then, sends the particular blast message to the blast’s contacts using the Msg91 API. Then, it marks those blasts completed using the transition model method ‘mark_blast_completed’. It also creates objects in the Attempt table with these blasts and its corresponding contacts.

**Task 3:** Update stalled attempts
**Description:** This task filters those attempts whose own status is ‘pending’, whose blast status is ‘completed’ & who are created before 6 hrs. These attempts' status is changed to ‘unreachable’.

To receive the reports send by the Msg91 about the delivery status of scheduled blasts, following endpoint has been made:
**Description:** This process filters those blasts whose state is ‘completed’ and collects the Msg91 API response report and updates the Attempt Model’s method ‘status’ which can take 3 values - *reached, unreached, & unreachable*. This state is in correspondence to each of the contacts which is reflected in the response report received by the Msg91 API.

| **Attempt Status** | **Description** |
|---|---|
| Reached | Msg91 report *status* = ‘01’ and *desc* = ‘delivered’ |
| Unreached | Msg91 report *status* = ‘02’ and *desc* = ‘failed’ |
| Unreachable | Msg91 report *status* = ‘16’ and *desc* = ‘rejected’ |

Table. 3 Attempt status Description

Furthermore, the Dashboard module will use this Attempt model to populate the dashboard with charts and graphs showing the unique count of contacts in 3 categories - reached, unreached and unreachable, which belong to those blasts whose send date comes under start date and end date specified by the user.

# 7. Implementation

## 7.1 Implementation Strategy

The implementation strategy for each module had the following steps:

1. Creating new branch for each module development on the sms_blasts git repository
2. Solution design
3. Code Implementation according to the design
4. Test case design and Implementation. This step required that all the test cases passed and manual testing had to be performed for frontend testing.
5. Creating merge request for each module branch into the feature sms_blasts branch, and submitting the MR for Code Review to mentor
6. Multiple iterations of code review and code review changes
7. Merging of module branch into feature branch

## 7.2 API Endpoints Request Response

**1. Register user API (POST):** "/api/account/register"

| **Request body:** | **Response body:** |
|---|---|
| { "first_name": "Jess", "last_name": "Pearson", "username": "jess", "email": "jess@pearson.com", "password": "password" } | { "id": 51, "first_name": "Jess", "last_name": "Pearson", "username": "jess", "email": "jess@pearson.com", "token": "d36ce856e7c9e75c822d6f38b493adeb046b8123" } |

**2. Login user API (POST):** "/api/account/login"

| **Request body:** | **Response body:** |
|---|---|
| { "username":"jess", "Password":"password"} | {"token": "d36ce856e7c9e75c822d6f38b493adeb046b8123" } |

**3. Create Contact API (POST):** "/api/contact/create"

| **Request body:** | **Response body:** |
| --- | --- |
| { file: excel/csv file } | { "valid":10, "invalid":1, "duplicate":0 } |

**4. Edit Contact API (PATCH):** "/api/contact/update/1"

| **Request body:** | **Response body:** |
| --- | --- |
| { "first_name": "Mikey", "last_name": "Ross" } | { "id": 1, "phone_number":"+919879016500", "first_name": "Mikey", "last_name": "Ross" "created": "2021-05-08T07:47:24.010634Z" } |

**5. Contact List API (GET):** "/api/contact"

**Response body**:

```
{ "count": 20,
  "next": "http://localhost:8000/api/blast/list?page=2",
  "previous": null,
  "results": [ {
                "id": 1,
                "phone_number":"+919879016500",
                "first_name": "Mikey",
                "last_name": "Ross"
                "created": "2021-05-08T07:47:24.010634Z"
    }, ..... (truncated 9 contacts) ]
}
```

**6. Create Blast API (POST):** "/api/blast/create"

| **Request body:** | **Response body:** |
|---|---|
| {<br>"blast_name":"Event Announcement",<br>"message": "testing message",<br>"send_on": "2021-05-22 16:15:00",<br>"Contacts":[1,2,3] } | {<br>"response": "Blast Created Successfully!"<br>} |

**7. Blast Detail API (GET):** "/api/blast/list/1"

**Response body:**


{ "id": 1,
"contact_dict": { "+917990682487": "Mike Ross"},
"stats": {"recipients": 1, "attempted": 1, "received": 1},
"blast_name": "testing 1",
"message": "testing message",
"status": 3, "send_on": "2021-04-30T19:37:00Z",
"contacts": [1] }

**8. Blast List API (GET)**: "/api/blast/list"

**Response body**:


```
{  "count": 20,
   "next": "http://localhost:8000/api/blast/list?page=2",
   "previous": null,
   "results": [ {
         "id": 20,
         "contact_dict": { "+917990682487": "Mike Ross"},
         "stats": { "recipients": 1, "attempted": 0, "received": 0 },
         "blast_name": "testing blast",
         "message": "testing message",
         "status": 1,
         "send_on": "2021-05-22T16:15:00Z",
         "contacts": [1]
      }, ..... (truncated 9 blasts) ]
}
```


**9. Dashboard API (GET):** "/api/dashboard/"

**Request Query Parameters:**


```
{
"report_type":"unique_contacts_reached",
"start_date": "11/04/2021",  "end_date": "15/04/2021"  }
```


**Response body:**


```
{    "reached": 6,
   "unreached": 2,
   "unreachable": 1 }
```

## 7.3 Asynchronous tasks flowcharts

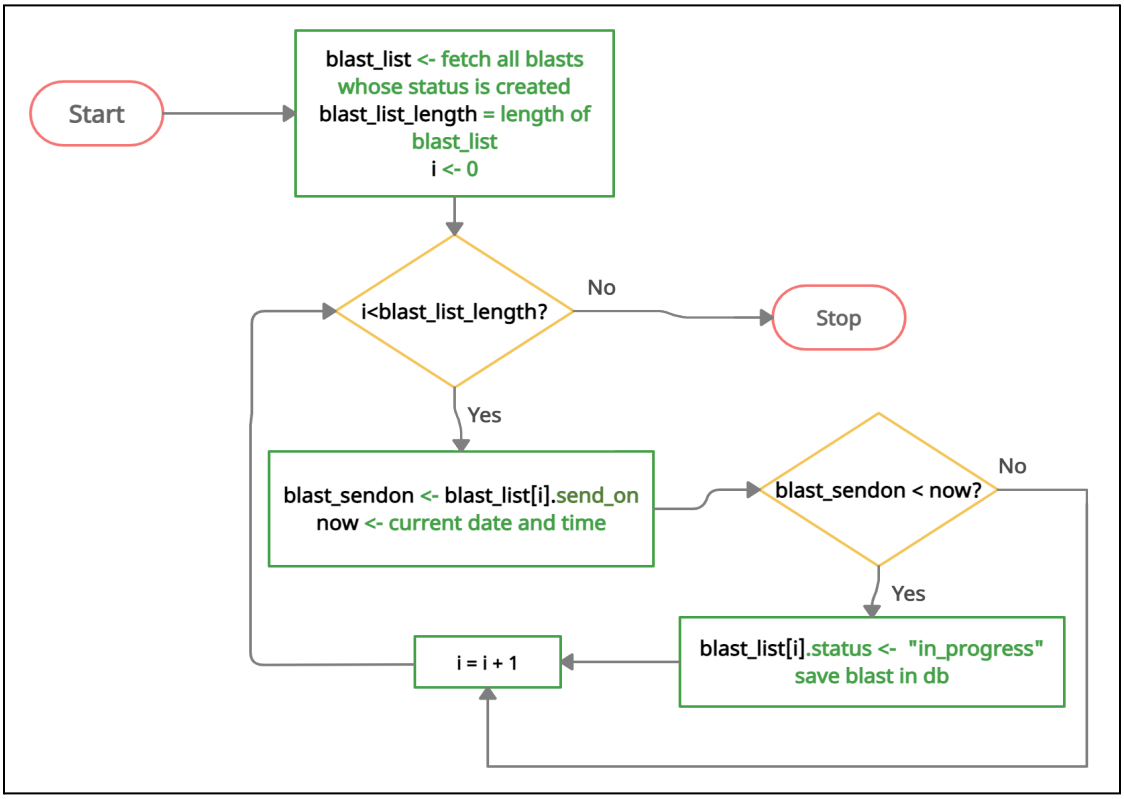



Fig. 28 Asynchronous Task 1 Prepare Blasts for Scheduling Flowchart

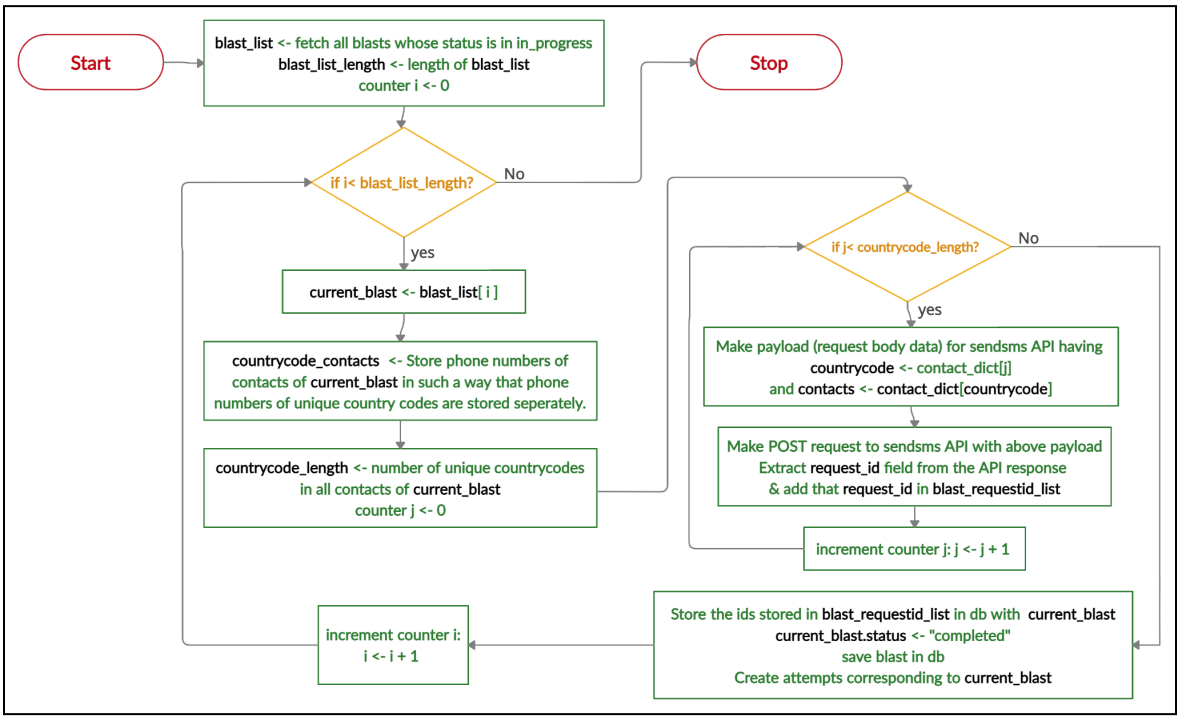



Fig. 29 Asynchronous Task 2 Schedule Blast Flowchart

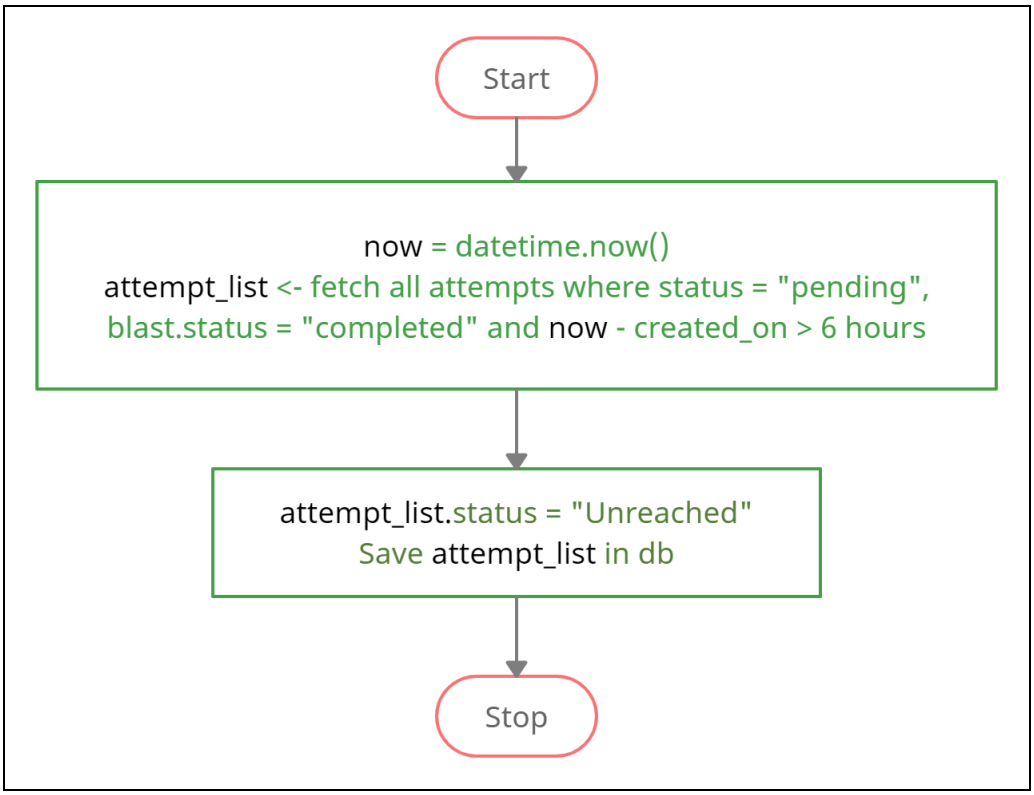



Fig. 30 Asynchronous Task 3 Update Stalled Attempts flowchart

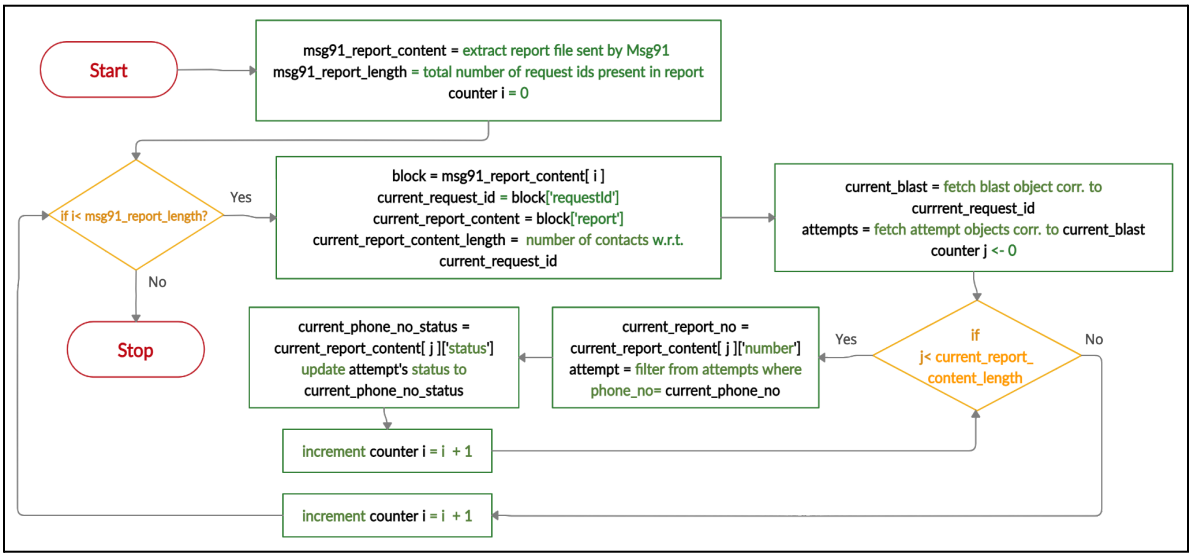



Fig. 31 Msg91 Report API Endpoint Process

## 7.4 Server Setup and Deployment

### 7.4.1 Setting up Nginx and uWSGI

1. Install uwsgi with the following command: `pip install uwsgi`
2. Install Nginx with Following steps

   ```
   sudo apt-get install nginx
   sudo /etc/init.d/nginx start # start nginx
   ```

   To check if your Nginx service is running, go to http://localhost:80/ , you should see the welcome to nginx message.
3. Make a file called “uwsgi_params” in your project directory in settings. Put this content in this file.
4. Next we need to build the frontend application for deployment. We cd into the directory where frontend code lives and run this command:

   ```
   node --max_old_space_size=8192 ./node_modules/@angular/cli/bin/ng build
   ```

   The space defined in the above command may vary from project to project.
5. Next we set up sms_blasts_nginx.conf. This is a configuration file and it tells Nginx how to serve static and media files, proxy pass the django calls to uwsgi and serve requests for angular too. Download this file and edit the paths to your static and media folders, and uwsgi_params file. After that put that file in the following directory - /etc/nginx/sites-available/
6. Symlink to this file from /etc/nginx/sites-enabled and restart nginx:

   ```
   sudo ln -s /etc/nginx/sites-available/sms_blasts_nginx.conf /etc/nginx/sites-enabled/
   sudo /etc/init.d/nginx restart
   ```

7. You need to start uwsgi server with the following command:

   ```
   uwsgi --socket sms_blasts.sock --module config.wsgi --chmod-socket=666
   ```

   The --module parameter defines which uwsgi module to load

8. The whole application will be running on http://localhost:8000.

### 7.4.2 Deployment with Ngrok service

To make our server running on a local machine to be accessible over the internet we have used Ngrok. The following steps were followed for the deployment of this project.

1. Download ngrok from the site: Ngrok download link.
2. Unzip to get the client application:

```
unzip /path/to/ngrok.zip
```

3. Add authtoken to ngrok.yml. Cd into the directory that has ngrok application and run(the auth token will be displayed when you sign in to ngrok account):

```
./ngrok authtoken <auth_token>
```

4. Run this following command to start a HTTP tunnel forwarding to your local port 8000, where nginx is serving the files.

```
./ngrok http 8000 -host-header="localhost:8000"
```

## 7.5 Optimization techniques

### 7.5.1 Database Design

Database design is one of the very important parts of design since the performance of a system is greatly affected by it. While designing a database, we need to keep in mind both space and time complexities.

Normalization is a very common technique which is used during database design. It suggests that if two tables have a many-to-many relationship then we should make another table, such that both the original table have one-to-many relationship with this new table. This optimisation technique helps with reducing space complexity.

Even though this technique is very useful, we need to keep in mind the space-time complexity trade offs.

During our implementation and designing of database tables for ‘Users’ and ‘Contacts’ tables, after analysis and testing we decided to not use normalization techniques and to go

with little high space complexity because it optimised time complexity significantly for us.

### 7.5.2 Database Usage Optimization

In large and complex databases, it is important to reduce the number of database hits using optimisation techniques. The primary reason for this is the cost of database operations in terms of time and resource utilisation. A common problem that we faced while designing and implementing our system was that the normal method of retrieval of all the data of an object made multiple calls to the database to get all the data. This posed an even bigger problem when this retrieval was happening inside a loop. In this section we will focus on two queryset methods provided by django - **prefetch_related()** and **select_related()** that eliminates the need for multiple database calls for the retrieval of all the data related to the object. It should be noted that these methods only increase the performance if later in your code you are trying to access the related field data, otherwise it only adds overhead. [11]

**1. selected_related() method usage and description:**

This method can be used when you are trying to retrieve an object which has fields having either one-to-one or Foreign key relationship. It fetches the related data by performing a complex query once on the database. [11]

One example of the use of select_related() from this project:

```
attempts = Attempt.objects.filter(blast=b).select_related('contact')
for i in report:
    attempt = attempts.get(contact__phone_number=report_number)
```

Attempt model has two foreign key relationships. In the for loop, the phone number field is accessed through the contact object which is a related object in the Attempt table. Since, we have already fetched related object ‘contact’ for all the attempts in the first statement, the database hit does not happen for every iteration of the for loop (while accessing fields of the related contact object). Similarly, select_related() is used in download report functionality, and Msg91 report update attempts Post API.

2. prefetch_related() method usage and description:

This method can retrieve all the related objects for a given lookup. It works for fields having any relationship: one-to-one, Foreign key, one-to-many or many-to-many. The primary difference between the prefetch and select related queries is the way in which they retrieve the objects. Select_related query does the joining in the SQL query, while the prefetch_related query will get all the data and perform join in python. [11]

One example of the use of prefetch_related from this project:

```
queryset =
Blast.objects.filter(owner=self.request.user.id).prefetch_related('contacts')
for blast in queryset:
    for contact_object in blast.contacts.all():
        contact_dict[contact_object.phone_number.__str__()] =
          contact_object.first_name + ' ' + contact_object.last_name
```

Blast model has a many-to-many relationship with contacts Model. In the inner for loop, the fields - first_name, last_name and phone_number are accessed through the contact object which is a related field (many-to-many field) in the Blast table. Since we have used prefetch_related, it collects all the contacts associated to all the blasts in just one extra database hit, and due to this, the database hit does not occur for all the iterations of the inner for loop (while accessing fields of the related contact object). Similarly, prefetch_related() is used in Blast list and Blast detail functionality.

### 7.5.3 Server calls reduction

In complex systems which are created by large organisations with complex business processes and thousands of customers who access the servers at any given single time, it becomes important to reduce HTTP requests to the server to decrease it's load. Increase in HTTP calls increases server load and increases the latency to load the requested resource. Latency has a substantial impact on application performance. Hence, to reduce HTTP requests, we designed our system in such a way that it incorporates client side validations, which allows rudimentary validation of user data without submitting anything to the server. This allows for more interactivity by immediately responding to users' actions and also executes the requests quickly because they do not require a trip to the server.

# 8. Testing

This section thoroughly covers the test plans and the testing strategy. Following types of tests are planned for this project:

- Unit Testing: We will be independently testing the smallest testable parts of the application which are called units(or components of the software).
- Integration Testing: We will be integrating small units and test that they work properly when integrated.

**Testing Strategy**

This software has both a frontend and a backend. The testing strategy employed for different kind of testing and for frontend and backend is different and is summarised below:

- Unit testing backend: Automated tests in django
- Unit testing frontend: Manual tests
- Integration testing: Manual Tests

As the team of developers are full stack developers, the testing will be performed by them. Before implementation of any module, a test case document is prepared. After the implementation, the test cases are implemented and testing is performed. 100% passing of tests is expected.

*** means that the same test case is repeated for all other fields**

| Test Case ID | Test Scenario | Test Case | Pre conditions | Test Steps | Test Data | Expected Result | Actual Result | Status (Pass/Fail) |
|---|---|---|---|---|---|---|---|---|
| TC_REG_01 | Verifying Registering functionality | Testing creation of user with valid details. | - | 1. Enter valid email | test.project@gmail.com | Successful registration response contains status_code 201 (created) and the details of the user. | As expected. | Pass |
| | | | | 2. Enter valid username | test_user | | | |
| | | | | 3. Enter valid first and last name | test_fname, test_lname | | | |
| | | | | 4. Enter same value: password & retype password | test12345, test12345 | | | |
| | | | | 5. Call POST Register API | api_url = "/api/account/register" Body= user_object | | | |
| TC_REG_02* | Verifying Registering functionality | Testing register functionality with invalid data. | - | 1. Enter invalid data for one or more fields | Eg: email: test@user | It results in an unsuccessful registration and response contains status code 400 (i.e. Bad request). It also contains key which has invalid data in the response with value that says Invalid <fieldname>. | As expected | Pass |
| | | | | 2. Enter valid data for other fields | {username: test,first_name:test, last_name:user} | | | |
| | | | | 4. Enter same value: password & retype password | test12345, test12345 | | | |
| | | | | 5. Call POST Register API | api_url = "/api/account/register" Body= user_object | | | |
| TC_REG_03 | Verifying Registering functionality | Testing register functionality with email of an existing user. | - | 1. Enter valid but existing email | test.project@gmail.com | It results in an unsuccessful registration and response contains status code 400 (i.e. Bad request). It also contains "email" key in the response data with value that says Email already exists | As expected | pass |
| | | | | 2. Enter valid username | test_user1 | | | |
| | | | | 3. Enter valid first and last | test_fname, test_lname | | | |
| | | | | 4. Enter same value: password & retype password | test12345, test12345 | | | |
| | | | | 5. Call POST Register API | api_url = "/api/account/register" Body= user_object | | | |
| TC_REG_05 | Verifying Registering functionality | Testing register functionality with different values for password and retype password | - | 1. Enter valid email | test.project1@gmail.com | It results in an unsuccessful registration and response contains status code 400 (i.e. Bad request). It also contains "confirm_password" key in the response data with value that says confirm password should match with password. | As expected | pass |
| | | | | 2. Enter valid username | test_user1 | | | |
| | | | | 3. Enter valid first and last | test_fname, test_lname | | | |
| | | | | 4. Enter different strings for password and retype password field | test12345, test123 | | | |
| | | | | 5. Call POST Register API | api_url = "/api/account/register" Body= user_object | | | |
| TC_REG_07 | Verifying Registering functionality | Testing register functionality with username of an existing. | - | 1. Enter valid email | test.project1@gmail.com | It results in an unsuccessful registration and response contains status code 400 (i.e. Bad request). It also contains "username" key in the response data with value that says username already exists | As expected | pass |
| | | | | 2. Enter existing username | test_user | | | |
| | | | | 3. Enter valid first and last | test_fname, test_lname | | | |
| | | | | 4. Enter same value: password & retype password | test12345, test12345 | | | |
| | | | | 5. Call POST Register API | api_url = "/api/account/register" Body= user_object | | | |
| TC_REG_11 * | Verifying Registering functionality | Testing register functionality with any one of the fields missing. | - | 1. Enter valid email | test.project1@gmail.com | It results in an unsuccessful registration and response contains status code 400 (i.e. Bad request). It also contains "lastname" key in the response data with value that says lastname is required | As expected | pass |
| | | | | 2. Enter valid username | test_user1 | | | |
| | | | | 3. Enter valid first and no last name | test_fname | | | |
| | | | | 4. Enter same value: password & retype password | test12345, test12345 | | | |
| | | | | 5. Call POST Register API | api_url = "/api/account/register" Body= user_object | | | |
| TC_LOGIN_01 | Verifying Logging in functionality | Test login of an existing user with correct credentials | Registered account | 1. Enter valid username | test_user | Successful login with response status code 200 (i.e. OK) and response data contains the login token associated with that user | As expected | Pass |
| | | | | 2. Enter valid password | password | | | |
| | | | | 3. Call POST Login API | api_url = "/api/account/login" Body= username and password | | | |
| TC_LOGIN_02* | Verifying Logging in functionality | Test login with wrong credentials | Registered account | 1. Enter wrong username | test_user123 | Unsuccessful login with response status code 404 (i.e. Not found). It also contains "error" key in the response data with value that says authentication credentials are invalid | As expected | Pass |
| | | | | 2. Enter valid password | password | | | |
| | | | | 3. Call POST Login API | api_url = "/api/account/login" Body= username and password | | | |
| TC_LOGIN_03* | Verifying Logging in functionality | Test login with any one of the fields missing | Registered account | 1. Enter no username | | Unsuccessful login with response status code 400 (i.e. Bad request). It also contains "username" key in the response data with value that says username is required | As expected | Pass |
| | | | | 2. Enter valid password | password | | | |
| | | | | 3. Call POST Login API | api_url = "/api/account/login" Body= username and password | | | |
| TC_BLAST_01 | Verifying Blast functionality | Test blast creation with valid details | User logged in (that gives current user token) | 1. Enter valid blast name | test_blast | Successful blast creation with response status code 200 (i.e. OK) and response data says "Blast successfully created" | As expected | Pass |
| | | | | 2. Enter valid contacts and valid message | <valid_list_of_contacts>, <message acc to dlt_id > | | | |
| | | | | 4. Enter valid send date | 2021-7-30 19:00:00 | | | |
| | | | | 5. Enter valid DLT template id | <registered 24 character DLT template id> | | | |
| | | | | 6. Call POST create Blast API with current user token | *api_url* = "/api/blast/create" *header* = "Authorization: Token <user token>" *Body*: blast_object | | | |
| TC_BLAST_02 | Verifying Blast functionality* | Test blast creation with too long blast name or any other field | User logged in (that gives current user token) | 1. Enter long blast name | [test_blast]*20 | Blast creation unsuccessful with response status code 400 (i.e. Bad request) and response data contains a key "blast_name" and its value says this field should be under 120 characters. | As expected | Pass |
| | | | | 2. Enter valid contacts and valid message | <valid_list_of_contacts>, <message acc to dlt_id > | | | |
| | | | | 4. Enter valid send date | 2021-7-30 19:00:00 | | | |
| | | | | 5. Enter valid DLT template id | <registered 24 character DLT template id> | | | |
| | | | | 6. Call POST create Blast API with current user token | api_url = "/api/blast/create" header = "Authorization: Token <user token>" Body: blast_object | | | |
| TC_BLAST_04 | Verifying Blast functionality | Test blast creation with wrong format blast send date field | User logged in (that gives current user token) | 1. Enter valid blast name | test_blast | Blast creation unsuccessful with response status code 400 (i.e. Bad request) and response data contains a key "send_on" and its value says : "Datetime has wrong format. Use one of these formats instead: YYYY-MM-DDThh:mm[:ss[.uuuuuu]][+HH:MM\|-HH:MM\|Z]." | As expected | Pass |
| | | | | 2. Enter valid contacts | <valid_list_of_contacts> | | | |
| | | | | 3. Enter valid message | <message acc to dlt id format> | | | |
| | | | | 4. Enter valid send date | 2021-1-30 19:00:00 | | | |
| | | | | 5. Enter valid DLT template id | <registered 24 chaacter DLT template id> | | | |
| | | | | 6. Call POST create Blast API with current user token | api_url = "/api/blast/create" header = "Authorization: Token <user token>" Body: blast_object | | | |
| TC_BLAST_05* | Verifying Blast functionality | Test blast creation with any of the fields empty (this case: empty blast name)* | User logged in (that gives current user token) | 1. Enter no blast name | <empty string> | Blast creation unsuccessful with response status code 400 (i.e. Bad request) and response data contains a key "blast_name" and its value says this field may not be blank. | As expected | Pass |
| | | | | 2. Enter valid contacts | <valid_list_of_contacts> | | | |
| | | | | 3. Enter valid message | <message acc to dlt_id> | | | |
| | | | | 4. Enter valid send date | 2021-7-30 19:00:00 | | | |
| | | | | 5. Enter valid DLT template id | <registered 24 chaacter DLT template id> | | | |
| | | | | 6. Call POST create Blast API with current user token | api_url = "/api/blast/create" header = "Authorization: Token <user token>" Body: blast_object | | | |
| | | Test blast creation with any of the fields | | 1. Don't enter blast name field | | Blast creation unsuccessful with | | |
| | | | | 2. Enter valid contacts | <valid_list_of_contacts> | | | |
| | | | | 3. Enter valid message | <message acc to dlt_id> | | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TC_BLAST_06* | Verifying Blast functionality | with any of the fields missing<br><br>(this case: missing blast name)* | User logged in (that gives current user token) | 4. Enter valid send date | 2021-7-30 19:00:00 | response status code 400 (i.e. Bad request) and response data contains a key "blast_name" and its value says this field is required. | As expected | Pass |
| | | | | 5. Enter valid DLT template id | <registered 24 chaacter DLT template id> | | | |
| | | | | 5. Call POST create Blast API with current user token | api_url = "/api/blast/create"<br>header = "Authorization: Token <user token>"<br>Body: blast_object | | | |
| TC_BLAST_07 | Verifying Blast functionality | Test blast creation with no authentication credentials | User not logged in, and hence no login token is there. | 1. Enter valid blast name | test_blast | Blast creation unsuccessful with response status code 403 (i.e. FORBIDDEN) and response data says "Authentication credentials are not provided" | As expected | Pass |
| | | | | 2. Enter valid contacts | <list_of_valid_contacts> | | | |
| | | | | 3. Enter valid message | <message acc to dlt_id> | | | |
| | | | | 4. Enter valid send date | 2021-7-30 19:00:00 | | | |
| | | | | 5. Enter valid DLT template id | <registered 24 chaacter DLT template id> | | | |
| | | | | 5. Call POST create Blast API without any user token | api_url = "/api/blast/create"<br>Body: blast_object | | | |
| TC_BLAST_08 | Verifying Blast functionality | Test blast detail with valid blast id | User logged in (that gives current user token), | 1. Make blast detail API url with valid blast id | blast id: <valid blast id><br>api_url: "/api/blast/list/" + blast_id | Blast details fetched successfully with response status code 200 (i.e. OK) and response data gives details of blast which includes id, blast_name, message, contacts (contact ids), send_on, contact_dict (in the form of phone_number: fullname), and blast statictics | As expected | Pass |
| | | | | 5. Call GET Blast detail API with current user token | api_url: "/api/blast/list/" + blast_id<br>header = "Authorization: Token <user token>" | | | |
| TC_BLAST_09 | Verifying Blast functionality | Test blast detail with blast id of another user. | User logged in (that gives current user token), | 1. Make blast detail API url with blast id of another user. | blast id=<another user's blast id><br>api_url= "/api/blast/list/" + blast_id | Blast details fetching unsuccessful. Response status code 404 (i.e. NOT FOUND) and response data says "Blast not found!" | As expected | Pass |
| | | | | 5. Call GET Blast detail API with current user token | api_url= "/api/blast/list/" + blast_id<br>header = "Authorization: Token <user token>" | | | |
| TC_BLAST_10 | Verifying Blast functionality | Test blast detail with non existent blast id | User logged in (that gives current user token) | 1. Make blast detail API url with non existent blast id. | blast id: <non-existent blast id><br>api_url: "/api/blast/list/" + blast_id | Blast details fetching unsuccessful. Response status code 404 (i.e. NOT FOUND) and response data says "Blast not found!" | As expected | Pass |
| | | | | 5. Call GET Blast detail API with current user token | api_url: "/api/blast/list/" + blast_id<br>header = "Authorization: Token <user token>" | | | |
| TC_BLAST_11 | Verifying Blast functionality | Test blast detail without user authentication | User not logged in, and hence no login token is there | 1. Make blast detail API url with valid blast id | blast id: <valid blast id><br>api_url: "/api/blast/list/" + blast_id | Fetching of Blast detaiils unsuccessful. Response status code 403 (i.e. FORBIDDEN) and response data says "Authentication credentials are not provided" | As expected | Pass |
| | | | | 5. Call GET Blast detail API without any user token | api_url: "/api/blast/list/" + blast_id | | | |
| TC_BLAST_12 | Verifying Blast functionality | Test get blast list | User logged in (that gives current user token) | 1. Call GET Blast List API with current user token | api_url: "/api/blast/list/"<br>header = "Authorization: Token <user token>" | Blast list fetched successfully with response status code 200 (i.e. OK) and response data gives paginated blast list with all blast details | As expected | Pass |
| TC_BLAST_13 | Verifying Blast functionality | Test get blast list withour user authentication | User logged in (that gives current user token) | 1. Call GET Blast List API without any user token | api_url: "/api/blast/list/" | Blast list fetching unsuccessful, with response status code 403 (i.e. FORBIDDEN) and response data says "Authentication credentials are not provided" | As expected | Pass |
| TC_BLAST_14 | Verifying Blast functionality | Test blast download report with blast status "created" | User logged in (that gives current user token), | 1. Make blast download API url with valid blast id whose status is created | blast id: <valid blast id with status=created><br>api_url: "/api/blast/download_report/" + blast_id | Blast download report unsuccessful, with response status code 400 (i.e. BAD REQUEST) and response data says "Blast is not scheduled yet. Please try again after some time!" | As expected | Pass |
| | | | | 2. Call GET Blast Download report API with current user token | *api_url*: "/api/blast/download_report/" + blast_id<br>*header* = "Authorization: Token <user token>" | | | |
| TC_BLAST_15 | Verifying Blast functionality | Test blast download report with blast status "completed" | User logged in (that gives current user token) | 1. Make blast download API url with valid blast id whose status is completed | *blast_id*: <valid blast of another user with status=completed><br>*api_url*: "/api/blast/download_report/" + blast_id | Blast download report successful, with response status code 200 (i.e. OK) and response data contains information of that blast's contacts: Firstname, lastname, phone_number, send_on, delivery_status. | As expected | Pass |
| | | | | 2. Call GET Blast Download report API with current user token | *api_url*: "/api/blast/download_report/" + blast_id<br>*header* = "Authorization: Token <user token>" | | | |
| TC_BLAST_16 | Verifying Blast functionality | Test blast download report with blast of another user | User logged in (that gives current user token) | 1. Make blast download API url with valid blast id of another user | *blast_id*: <valid blast id of another user><br>*api_url*: "/api/blast/download_report/" + blast_id | Blast download report unsuccessful, with response status code 404 (i.e. NOT FOUND), response data says "Blast not found" | As expected | Pass |
| | | | | 2. Call GET Blast Download report API with current user token | *api_url*: "/api/blast/download_report/" + blast_id<br>*header* = "Authorization: Token <user token>" | | | |
| TC_BLAST_17 | Verifying Blast functionality | Test blast download report of a non existent blast | User logged in (that gives current user token) | 1. Make blast download API url with non0existent blast id | *blast_id*: <non-existent blast id><br>*api_url*: "/api/blast/download_report/" + blast_id | Blast download report unsuccessful, with response status code 404 (i.e. NOT FOUND), response data says "Blast not found" | As expected | Pass |
| | | | | 2. Call GET Blast Download report API with current user token | *api_url*: "/api/blast/download_report/" + blast_id<br>*header* = "Authorization: Token <user token>" | | | |
| TC_BLAST_18 | Verifying Blast functionality | Test blast download report of any blast with no user authentication | User not logged in (hence no current user token) | 1. Make blast download API url with any blast id | *blast_id*: <any blast id><br>*api_url*: "/api/blast/download_report/" + blast_id | Blast download report successful, with response status code 403 (i.e. FORBIDDEN) and response data says "Authentication credentials are not provided" | As expected | Pass |
| | | | | 2. Call GET Blast Download report API with no user token | *api_url*: "/api/blast/download_report/" + blast_id | | | |
| TC_SCHEDULING_01 | Verifying Scheduling functionality | Test scheduling tasks for blast having send date in the past | User logged in (that gives current user token) | 1. Create two contacts (ids = 1,2) and then Create blast having send_on=now time with these two valid contacts | Blast object:<br>{ name: "test_blast",<br>contacts: [1,2],<br>message: <acc_to_dlt_id>,<br>dlt_temp_id: <registered_dlt_id>,<br>send_date: <time_now> } | -- | As expected | Pass |
| | | | | 2. Call 1st task: prepare_blast_for_scheduling() | -- | -- | | |
| | | | | 3. Check the blast status | -- | Blast status should be 2 (In_progress) | | |
| | | | | 4. Check Attempt count corresponding to this blast and its status | -- | the attempt status of two contacts of this blast should be 1 (i.e., Pending) | | |
| | | | | 5. Call 2nd task: schedule_blasts() | -- | -- | | |
| | | | | 6. Check the blast's status | -- | Blast status should be 3 (completed) | | |
| | | | | 7. Check status of attempts corresponding to this blast | -- | the attempt status of two contacts of this blast should be corresponding to the report (here, it should be delivered) | | |
| | | | | 1. Create two contacts (ids = 1,2) and then Create blast having send_on=future time with these two valid contacts | Blast object:<br>{ name: "test_blast",<br>contacts: [1,2],<br>message: <acc_to_dlt_id>,<br>dlt_temp_id: <registered_dlt_id>,<br>send_date: <future_time> } | -- | | |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| TC_SCHEDULING_02 | Verifying Scheduling functionality | Test scheduling tasks for blast having send date in the future | User logged in (that gives current user token) | 2. Call 1st task: prepare_blast_for_scheduling()<br>3. Check the blast status<br>4. Check Attempt count corresponding to this blast and its status<br>5. Call 2nd task: schedule_blasts()<br>6. Check the blast's status<br>7. Check status of attempts corresponding to this blast | --<br>--<br>--<br>--<br>--<br>-- | --<br>Blast status should be 1 (created)<br>No attempts should be found for the present blast<br>--<br>Blast status should be 1 (created)<br>No attempts should be found for the present blast | As expected | Pass |
| TC_SCHEDULING_03 | Verifying Scheduling functionality | Test update staled attempts task | User logged in (that gives current user token) | 1. Create two contacts (ids = 1,2) and then Create blast having send_on=current_time with these two valid contacts<br>2. Mark the blast in progress & create 2 attempts for this blast with status 'completed' & 'created_on'= 7 hours before current time<br>3. Call task: update_staled_attempts()<br>4. Check Attempt count corresponding to this blast and their status | Blast object:<br>{ name: "test_blast", contacts: [1,2], message: <acc_to_dlt_id>, dlt_temp_id: <registered_dlt_id>, send_date: <current_time> }<br>--<br>--<br>-- | --<br>--<br>--<br>Both attempts status changed to 'Unreachable' | As expected | Pass |
| TC_DASHBOARD_01 | Verifing Dashboard functionality | Test dashboard API with no authentication | - | 1. Enter valid details<br>2. Call GET dashboard stats without user token | <query_param: report_type, start_date, end_date><br>- | Dashboard stats fetching unsuccessful. Response status code 401 (i.e. Unauthorized) and response data says "Authentication credentials were not provided." | As expected | Pass |
| TC_DASHBOARD_02 | Verifing Dashboard functionality | Test dashboard API with valid details | User logged in (that gives current user token) | 1. Enter valid details<br>2. Call GET dashboard stats with user token | <query_param: report_type, start_date, end_date><br>- | Dashboard stats fetching successful. Response code 200 OK | As expected | Pass |
| TC_DASHBOARD_03* | Verifing Dashboard functionality | Test dashboard API with invalid details | User logged in (that gives current user token) | 1. Enter invalid value for one or more parameters<br>2. Call GET dashboard stats with user token | <query_param: report_type, start_date, end_date><br>- | Dashboard stats fetching unsuccessful. Response code 400_BAD_REQUEST | As expected | Pass |
| TC_DASHBOARD_04 | Verifing Dashboard functionality | Test dashboard API with different date ranges w.r.t start_date and blast send_on | User logged in (that gives current user token) | 1. Set query params value<br>3. Call GET dashboard stats with user token | <query_param: report_type, start_date, end_date><br>- | Dashboard stats fetching is successful. response code is 200_OK | As expected | Pass |
| TC_DASHBOARD_05 | Verifing Dashboard functionality | Test dashboard API with different date ranges w.r.t end_date and blast send_on | User logged in (that gives current user token) | 1. Set query params value<br>2. Create 3 blasts such that it cover three conditions w.r.t end_date<br>3. Call GET dashboard stats with user token | <query_param: report_type, start_date, end_date><br>-<br>- | Dashboard stats fetching is successful. response code is 200_OK | As expected | Pass |
| TC_DASHBOARD_06 | Verifing Dashboard functionality | Test dashboard API with same value for start_date, end_date and blast send_on | User logged in (that gives current user token) | 1. end_date = start_date = blast.send_on<br>2. Call GET dashboard stats with user token | <query_param: report_type, start_date, end_date> | Dashboard stats fetching is successful. response code is 200_OK | As expected | Pass |
| TC_DASHBOARD_07* | Verifing Dashboard functionality | Test dashboard API with missing query_params | User logged in (that gives current user token) | 1. Set query params s.t one or more parameter is missing<br>2. Call GET dashboard stats with user token | <query_param: report_type, end_date> | Dashboard stats fetching unsuccessful. Response code 400_BAD_REQUEST | As expected | Pass |
| TC_DASHBOARD_08 | Verifing Dashboard functionality | Test unique contacts reached filter of dashboard API | User logged in (that gives current user token) | 1. Create 2 blasts having one common contact<br>2. create attempt object s.t that common number's status in both the attempt is 'reached'<br>3. Set query_params<br>4. Call GET dashboard stats with user token | <blast_object><br><attempt_objects><br><query_param: report_type, start_date, end_date><br>- | Dashboard stats fetching successful. The stats value for key "reached sould be 1. The response code is 200_OK | As expected | Pass |
| TC_CONTACTS_01 | Verifying Contact list functionality | Test blast list API | User logged in | 1. Call GET contact list with user token | - | Contact List fetching is successful. The response code is 200_OK | As expected | Pass |
| TC_CONTACTS_02 | Verifying Contact Upload Functionality | Test contact creation with invalid file type/extension | User logged in | 1. Upload contacts file of invalid format<br>2. Call POST contact create API with user token | file<br>- | Contact creation unsuccessful. Response Code 400_BAD_REQUEST | As expected | Pass |
| TC_CONTACTS_03 | Verifying Contact Upload Functionality | Test contacts creation with invalid headers inside file | User logged in | 1. Upload contacts file with invalid format of columns<br>2. Call POST contact create API with user token | file<br>- | Contacts creation unsuccessful. Response Code 400_BAD_REQUEST | As expected | Pass |
| TC_CONTACTS_04 | Verifying Contact Upload Functionality | Test contacts upload with file exceeding MAX_FILE_SIZE | User logged in | 1. Upload contacts file exceeding file size<br>2. Call POST contact create API with user token | file<br>- | Contacts creation unsuccessful. Response Code 400_BAD_REQUEST | As expected | Pass |
| TC_CONTACTS_05 | Verifying Contact Upload Functionality | Test contacts upload with vald file type, headers and size | User logged in | 1. Upload valid contacts file<br>2. Call POST contact create API with user token | file<br>- | Contacts creation successful. Response code 201_CREATED | As expected | Pass |
| TC_CONTACTS_06 | Verifying Contact Edit Functionality | Test contact edit API for contact that user owns | User logged in | 1. Set valid values for body parameters<br>2. Call PATCH contact edit API for contact id that user owns with user token | { first_name: (within 30 characters), last_name: (within 30 characters)} | Contact edit is successful. Response code is 200_OK | As expected | Pass |
| TC_CONTACTS_07 | Verifying Contact Edit Functionality | Test contact edit API for a contact user doesn't own | User logged in | 1. Set valid values for body parameters<br>2. Call PATCH contact edit API for contact id that user does not own with user token | { first_name: (within 30 characters), last_name: (within 30 characters)} | Contact edit is unsuccessful. Response code is 403_FORBIDDEN | As expected | Pass |
| TC_CONTACTS_08* | Verifying Contact Edit Functionality | Test contact edit API for contact that user owns with invalid data | User logged in | 1. Set invalid values for body parameters<br>2. Call PATCH contact edit API for contact id that user owns with user token | { first_name: (exceeds 30 characters), last_name: (exceeds 30 characters)} | Contact edit is unsuccessful. Response code is 400_BAD_REQUEST | As expected | Pass |

# 9. Final Results

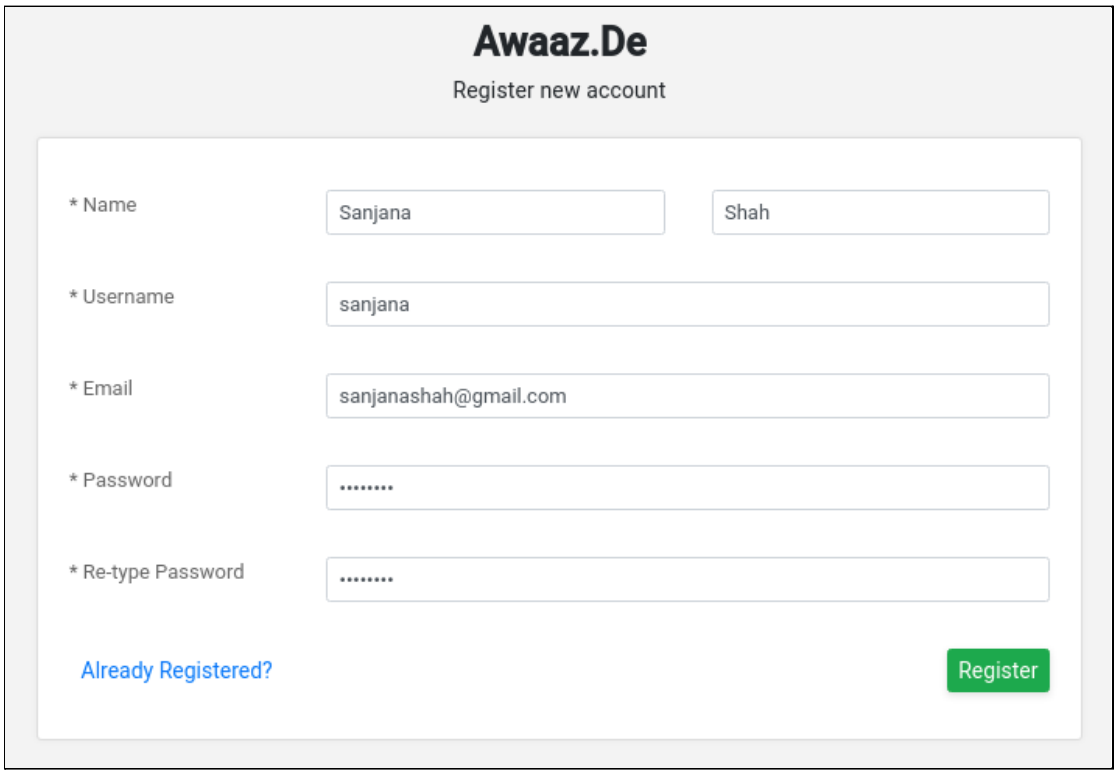



Fig. 32 Results: Register Page

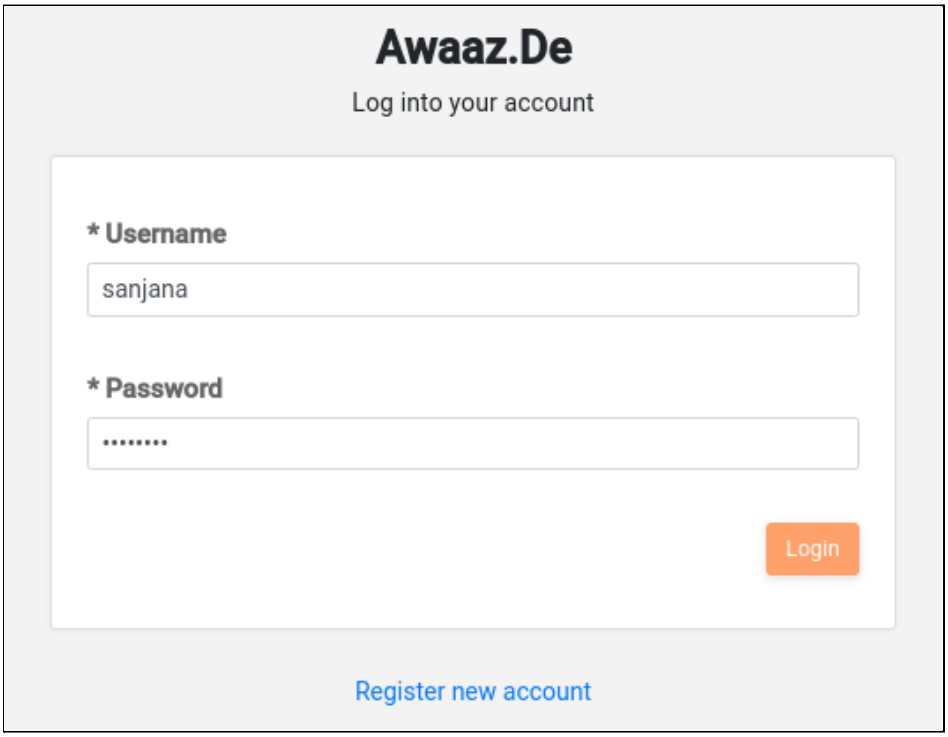



Fig. 33 Results: Login Page

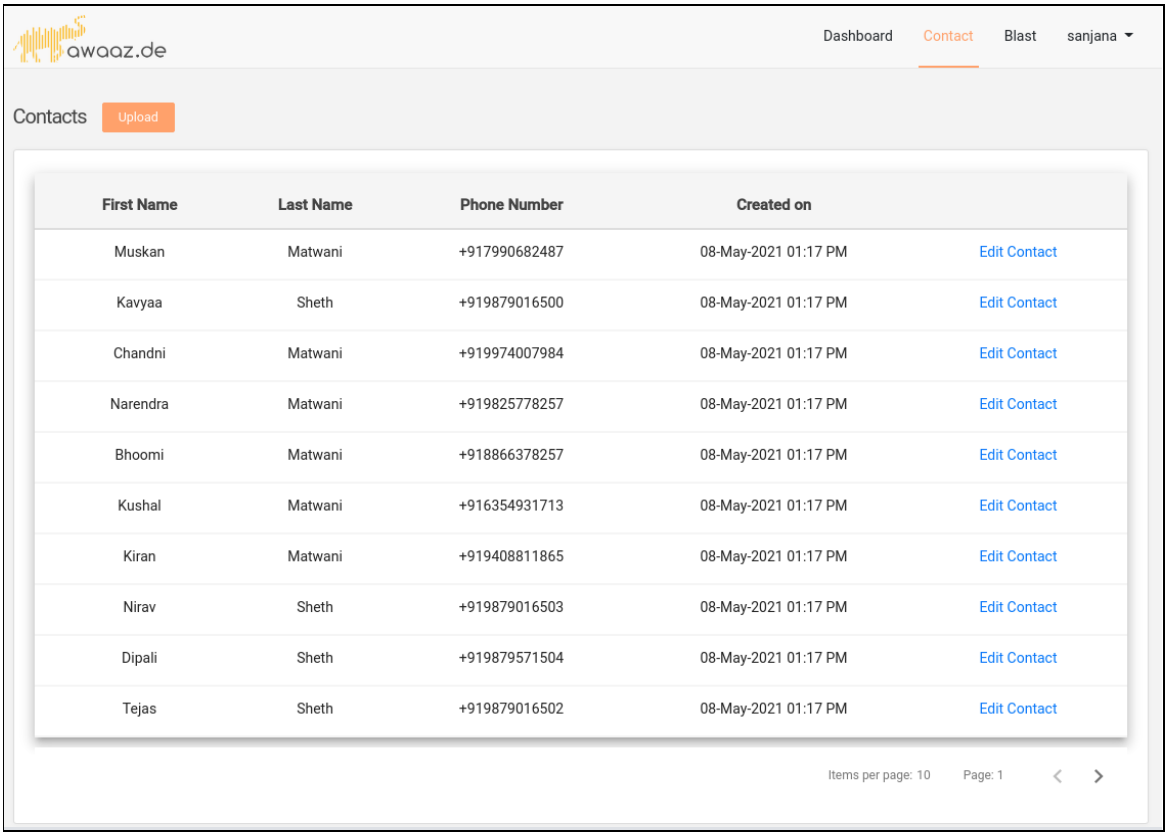



Fig. 34 Results: Contact List Page

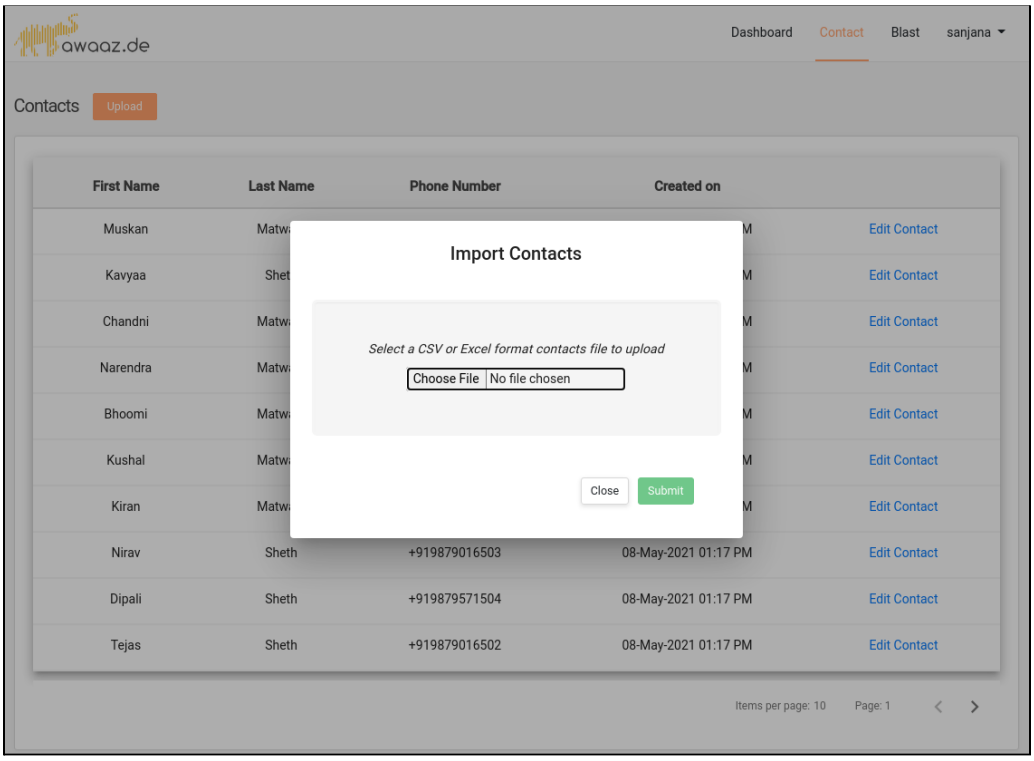



Fig. 35 Results: Contact Upload Page

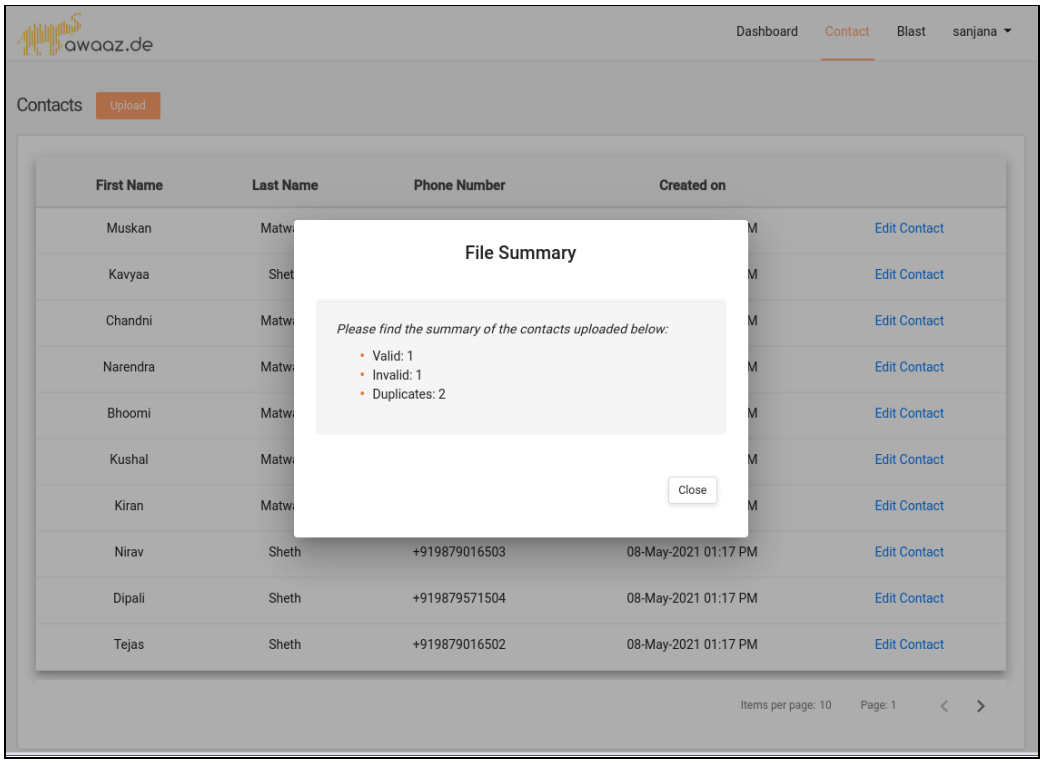



Fig. 36 Results: Contact Summary Page

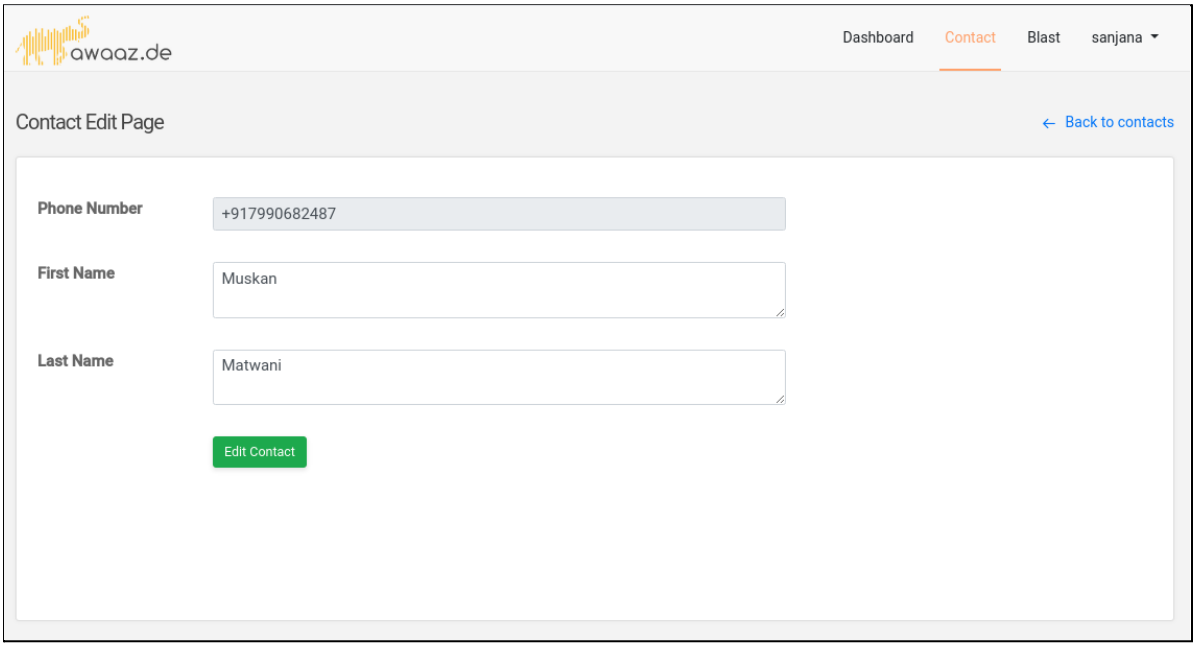



Fig. 37 Results: Contact Edit Page

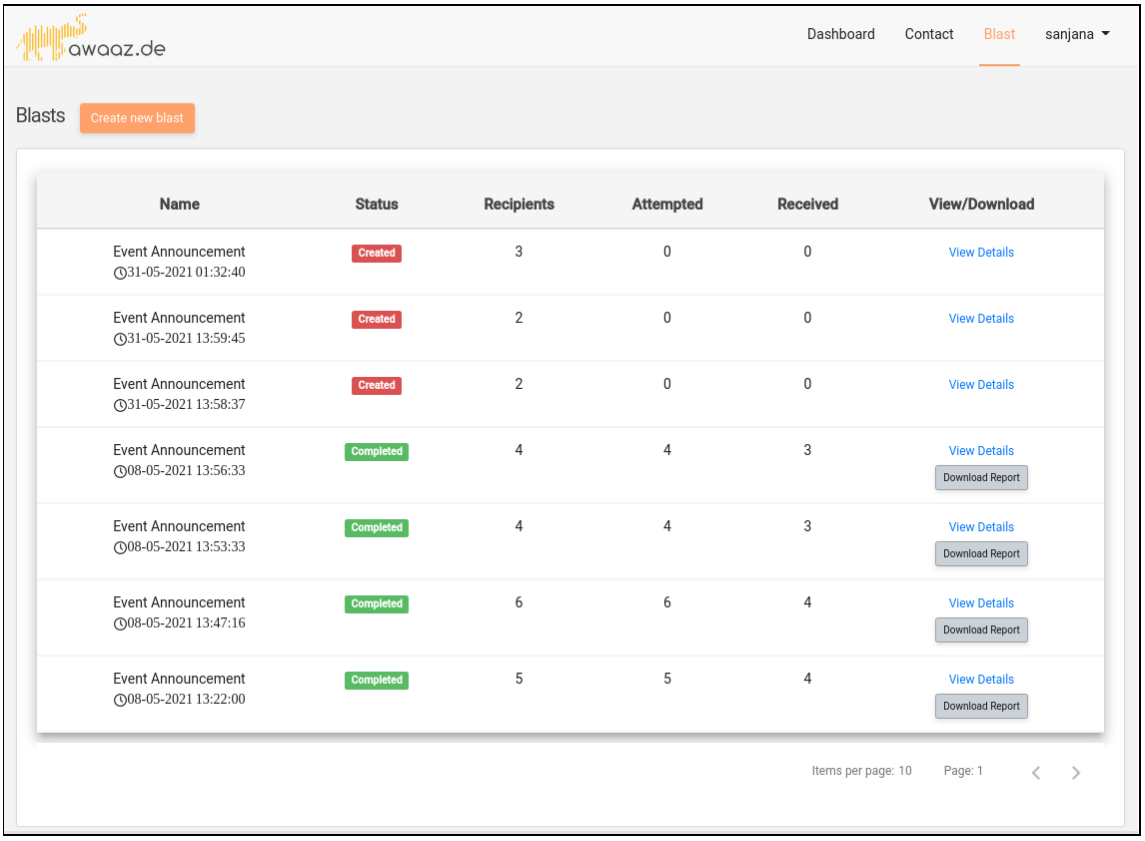



Fig. 38 Results: Blast List Page

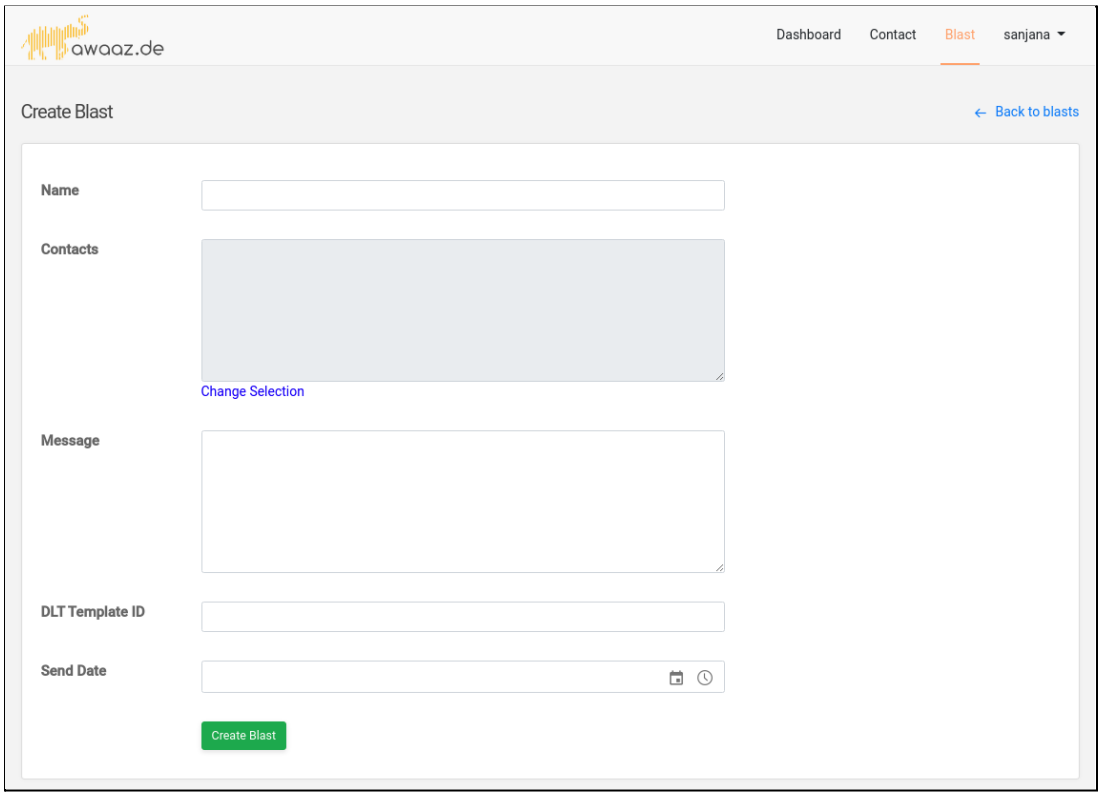



Fig. 39 Results: Blast Create Page

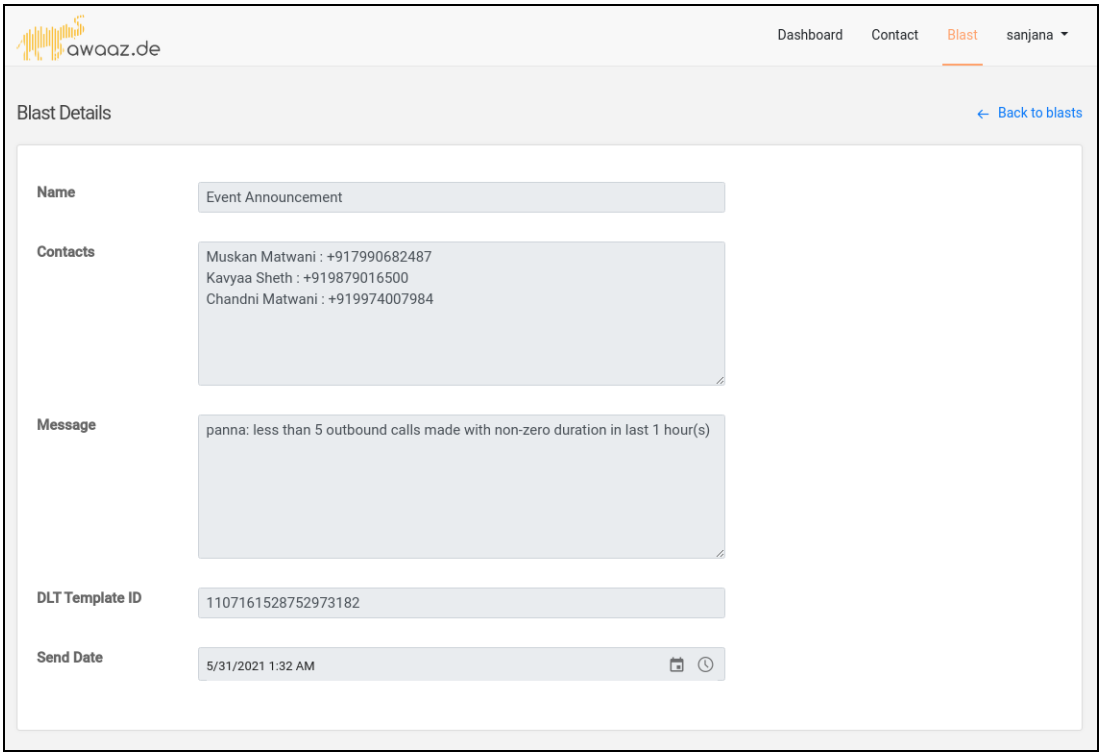



Fig. 40 Results: Blast Detail Page

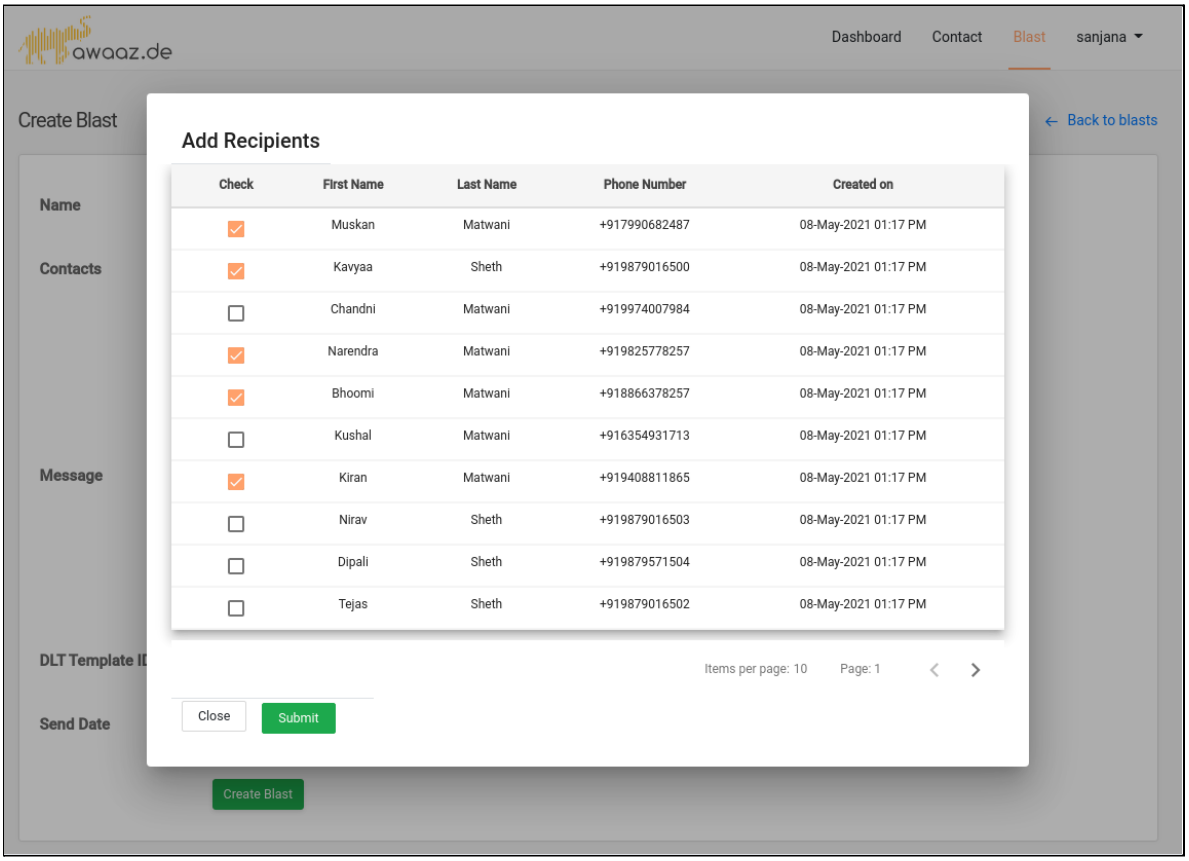



Fig. 41 Results: Blast Contact Picker Page

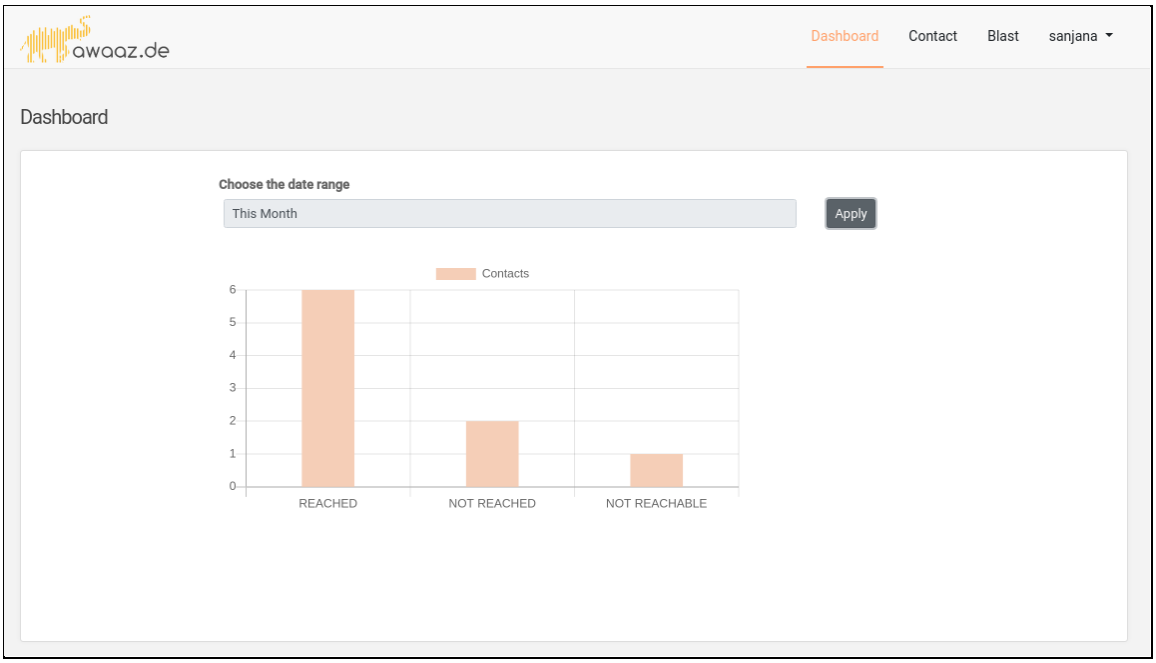



Fig. 42 Results: Dashboard Page

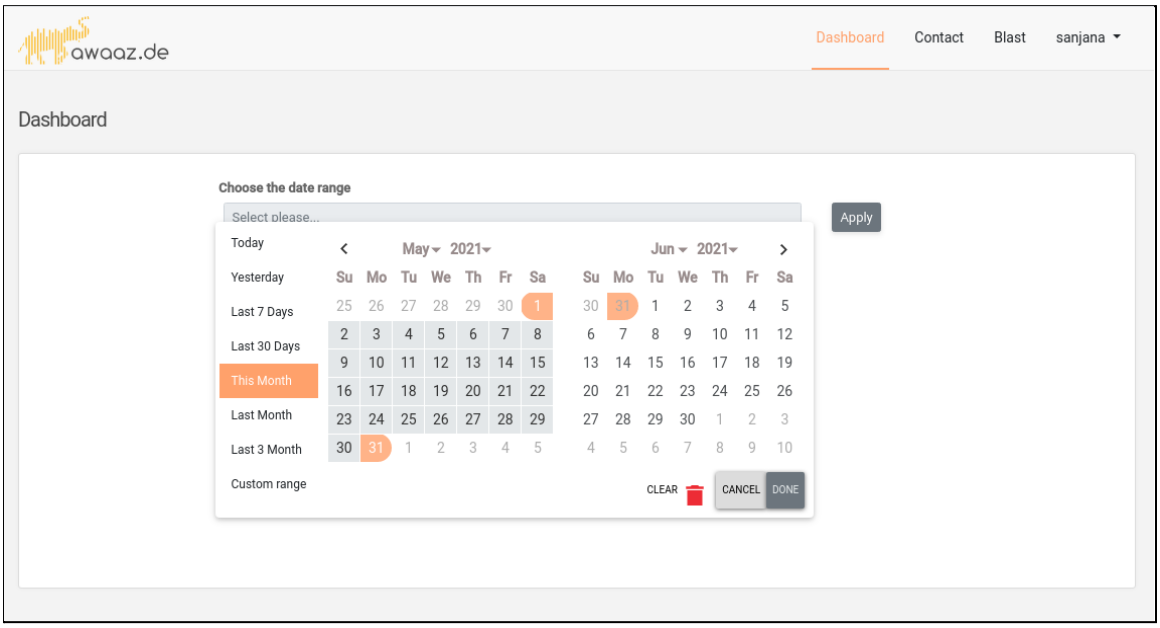



Fig. 43 Results: Dashboard Calendar Page

# 10. Code Distribution*

| Member Names | Login Module | Contact Module | Blast Module | Dashboard Module | Scheduling Module |
|---|---|---|---|---|---|
| **Kavyaa Sheth** | Frontend code | Backend code | Frontend code | Backend code | Celery task 1 + Post API msg91 report |
| **Muskan Matwani** | Backend code | Frontend code | Backend code | Frontend code | Celery task 2 + Celery task 3 |

Table. 5 Code Distribution

**We specifically want to bring to the attention that as per our organization guidelines, we have developed this whole project individually from scratch. Since the BTP guidelines prevented two different projects building the same product, we proposed to work as a team for the B.Tech Project where we will submit the project as a fair integration of our individual code. However, as specified, during this 4 month period, we both have worked on all the modules of this project individually. The final product which was built after merging both of our codes following the above mentioned distribution.*

# 11. Conclusion & Future Direction

We set out on the journey of the SMS Blast project with one primary motivation - to help solve the last mile connectivity issue. We strongly believe that the full power of technology can only be realised if it's inclusive. With the successful completion of SMS Blast Project, we have taken a small but significant step forward towards building a solution that serves the underserved communities. The SMS Blast Project meets all the requirements that we committed to in the project proposal as well as the Functional Requirements specification submitted in the mid sem report.

The barriers to last mile communication for underserved communities are - connectivity, literacy and communication barriers. This project only addresses the connectivity problem. The future direction for this project would be to send text in vernacular languages and incorporate IVR (Interactive Voice Response).

# 12. References


[1] (2020) "These Are The Countries Where Internet Access Is Lowest". World Economic Forum, https://www.weforum.org/agenda/2020/08/internet-users-usage-countries-change-demographics/.

[2] Statista. 2021. *India - internet usage among rural population by occupation 2017 | Statista*. [online] Available at: <https://www.statista.com/statistics/1012507/india-internet-usage-among-rural-population-by-occupation/>

[3] (2021) "SMS - Wikipedia". *En.Wikipedia.Org*, https://en.wikipedia.org/wiki/SMS.

[4] (2012) "A Brief History Of Text Messaging". *Mobivity.Com*, https://www.mobivity.com/mobivity-blog/a-brief-history-of-text-messaging.

[5] (2021) "Bulk Messaging - Wikipedia". *En.Wikipedia.Org*, https://en.wikipedia.org/wiki/Bulk_messaging.

[6] (2021) *Textlocal.In*, https://www.textlocal.in/.

[7] Arora, Gulpreet. (2021) "Bulk SMS Services Provider INDIA - SMSGATEWAYHUB". *Smsgatewayhub.Com*, https://www.smsgatewayhub.com/.

[8]: (2019) "Individuals Using The Internet (% Of Population) | Data". *Data.Worldbank.Org*, https://data.worldbank.org/indicator/IT.NET.USER.ZS?most_recent_year_desc=true.

[9] (2021) "Awaaz.De | Mobile Solutions For Social Impact". *Awaaz.De*, https://awaaz.de/.

[10] Patel, Neil. (2011) "Sharing Information in rural communities through voice interaction", Ph.D, Stanford University, 2011. https://hci.stanford.edu/neilp/pubs/patel-dissertation.pdf.

[11]. (2021) "Queryset API Reference | Django Documentation | Django". *Docs.Djangoproject.Com*, https://docs.djangoproject.com/en/3.2/ref/models/querysets/.

[12] (2021) "Make Python Faster With NGINX: Web Serving & Caching". *NGINX*, https://www.nginx.com/blog/maximizing-python-performance-with-nginx-parti-web-serving-and-caching/

[13] (2021) "DB2 - Schemas - Tutorialspoint". *Tutorialspoint.Com*, https://www.tutorialspoint.com/db2/db2_schemas.htm.

[14] (2021) "Security In Django | Django Documentation | Django". *Docs.Djangoproject.Com*, https://docs.djangoproject.com/en/2.2/topics/security/#sql-injection-protection.

# 13. Appendix

## 13.1 Load Testing

For a web application system like ours, Load testing is one of the most important types of testing as it gives the idea about how many concurrent users the current system handles. This testing enables the development team to determine the bottleneck for their system. The results of the load test can help in scaling the system, reduce the downtime and improve user experience by ensuring reduced response time.

There are various tools available that help in performing load testing. We decided to go with the “Apache bench” tool to load test our system. We have not performed extensive load testing, but our primary testing suggests that we can handle 16,690 requests per second with a concurrency of 8,100. Our findings are very close to the Nginx documentation [12], which suggests that it can make 10,000 simultaneous connections.

The steps we followed for performing load testing are as follows:

1. Install Apache bench

   ```
   sudo apt-get install apache2-utils
   ```

2. Run the following command. This command performs 10,000 requests with 8,100 concurrent users.

   ```
   ab -n 10000 -c 8100 http://localhost:8000/
   ```

   We did not directly try this number, We have tried for different combinations of number of requests and concurrency, and this was the bottleneck for our system.

We get the following output when we run the command mentioned in second point:

```
muskan@dell:~/Desktop/awaazde.sms_blasts/awaazde.sms_blasts$ ab -n 10000 -c 8100 http://localhost:8000/
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
Completed 1000 requests
Completed 2000 requests
Completed 3000 requests
Completed 4000 requests
Completed 5000 requests
Completed 6000 requests
Completed 7000 requests
Completed 8000 requests
Completed 9000 requests
Completed 10000 requests
Finished 10000 requests


Server Software:        nginx/1.14.2
Server Hostname:        localhost
Server Port:            8000

Document Path:          /
Document Length:        3793 bytes

Concurrency Level:      8100
Time taken for tests:   0.599 seconds
Complete requests:      10000
Failed requests:        20638
   (Connect: 0, Receive: 0, Length: 12287, Exceptions: 8351)
Total transferred:      6665258 bytes
HTML transferred:       6254657 bytes
Requests per second:    16690.42 [#/sec] (mean)
Time per request:       485.308 [ms] (mean)
Time per request:       0.060 [ms] (mean, across all concurrent requests)
Transfer rate:          10863.86 [Kbytes/sec] received

Connection Times (ms)
              min  mean[+/-sd] median   max
Connect:        0  155  51.5    152     250
Processing:    29  144  74.9    145     292
Waiting:        0   36  82.3      0     261
Total:        229  299  27.7    300     451

Percentage of the requests served within a certain time (ms)
  50%    300
  66%    306
  75%    309
  80%    311
  90%    336
  95%    356
  98%    367
  99%    370
 100%    451 (longest request)
```

When we try increasing the concurrency to 8200, it gives following output:

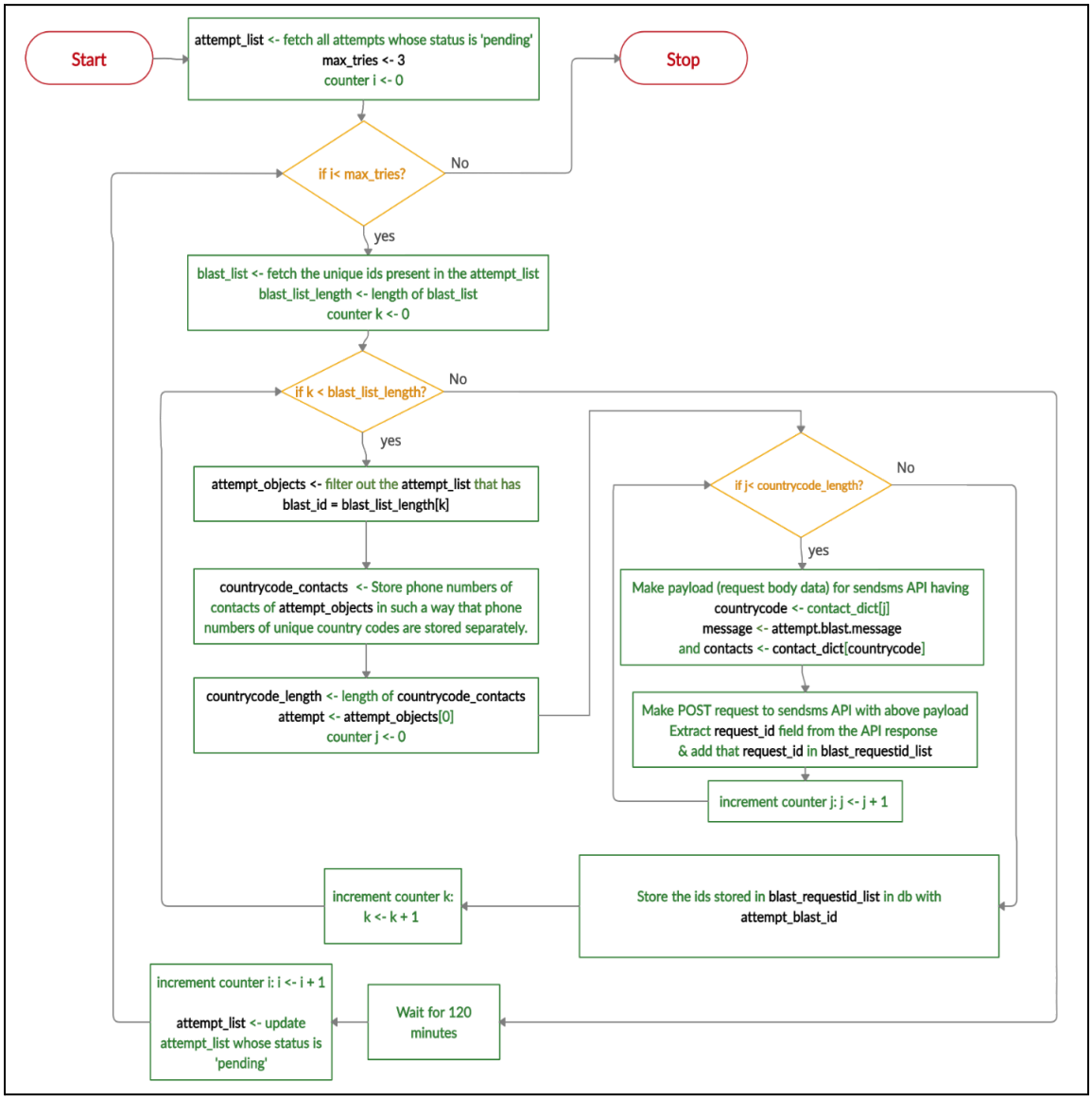
```
muskan@dell:~/Desktop/awaazde.sms_blasts/awaazde.sms_blasts$ ab -n 10000 -c 8200 http://localhost:8000/
This is ApacheBench, Version 2.3 <$Revision: 1843412 $>
Copyright 1996 Adam Twiss, Zeus Technology Ltd, http://www.zeustech.net/
Licensed to The Apache Software Foundation, http://www.apache.org/

Benchmarking localhost (be patient)
socket: Too many open files (24)
```

In future to handle the scenario for increased concurrent users, we need to increase the number of application servers and enable load balancing in Nginx.

## 13.2 Recovery Strategy for Unreachable Attempts

We are using a 3rd party application called Msg91 to send out messages. This API provides the delivery status report on the configured web URL. Sometimes, the API fails to provide a report for certain numbers indicating it has not sent out those messages. Currently in our system we deal with this scenario by marking the status of those attempts as “Unreachable”. The following flowchart shows the task design for improving our recovery policy and making sure that messages get delivered on those contacts too. This task is run every 6 hours since the completion time for this task is 6 hours. Keeping periodicity less than 6 hours will result in poor performance and wrong results.




Flowchart for Asynchronous Task 4

We would like to make a note on the meaning of different status that we set for the Attempt model. The following table summarizes it:

| **Attempt Status** | **Description** | **Meaning** |
|---|---|---|
| Reached | Delivered | Message is successfully delivered |
| Unreached | Failed | Either the number is out of service, switched off or there is a network issue |
| Unreachable | Rejected | Not enough credits left to send out the message |

It should be noted that Msg91 makes multiple tries to send the message on the number it is not able to deliver, it only returns with failed status if on all the attempts it is not able to deliver it. This is an indication that the number is no longer in use and it is not judicious to make even more attempts on the number whose status is “Unreached”. It is an indication of client data not being updated or proper.

## 13.3 Policy for Setting a Strong Password

Our current system has a login-register module, which helps users to register by giving its details which includes first name, last name, username, email, and password. While logging in, the user is asked for a username and password. The password has a validation which says that password must be at least 8 characters long. This is done to make sure that each user’s account is protected and secured and can’t be hacked easily. However, we can include a number of policies such as:

1. Passwords must not contain any of the personal information such as first name, last name, username, etc. This can be checked by doing the following:

```
password = "$password" #user entered password
first_name = "$first_name" #first_name entered by user
last_name = "$last_name" #last_name entered by user
username = "$username" #username entered by user
```

```
if first_name in password:
    return "Password is too similar to first name. Please enter a new password."
elif last_name in password:
    return "Password is too similar to last name. Please enter a new password."
elif username in password:
    return "Password is too similar to username. Please enter a new password."
else:
    return "Password is strong"
```

2. Passwords must contain at least one character, one number, one lowercase alphabet and one uppercase alphabet, with the min length of character being 8.

```
password = "$password" #user entered password
if len(password) < 8:
    return "Password must be at least 8 characters long”
elif re.search('[0-9]',password) is None:
    return "Password must contain at least one number”
elif re.search('[A-Z]',password) is None:
    return "Password must contain at least one uppercase alphabet”
elif re.search('[a-z]',password) is None:
    return "Password must contain at least one lowercase alphabet”
else:
    return "Password is strong"
```

After defining these policies, we can make sure that the password is strong enough so that the security and safety of the user’s account is maintained.

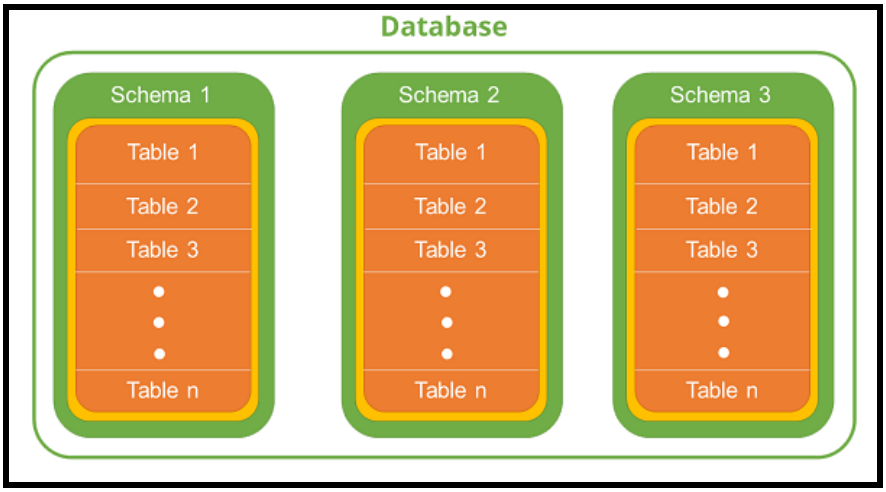
## 13.4 Design for Ensuring Data Privacy

Our current system supports multi-tenancy and stores all the tenant’s (client) information in the same database which has only one schema where all the tables are stored. Each table in this schema has an attribute called ‘user’ which differentiates the data among

different tenants. The application server takes care of filtering the data when a request is made according to the current tenant.
This approach has these advantages:
1. Since all tenant's data is stored in a single database, there is only one connection between the database and the application server. This eliminates the need for the extra configuration changes whenever a new tenant is added.
2. Since all data is stored in a single instance, it behaves as a standalone application which makes all the database management tasks easier.
3. There are some tables which should be accessible by all the tenants, and by this approach, storing shared data is easy as we just need to avoid adding the 'user' column in those shared tables.

However, the disadvantages are:
1. Data privacy is the biggest issue, as there can be scenarios where, for some reason, the 'user' filter is not applied while fetching the data from the database. This is a privacy vulnerability because a single such error can compromise sensitive information.
2. Since there is only one database instance in which all the tables are shared among all tenants, a single tenant can affect the performance of the application. The reason is, there are various scenarios where tables get locked by one tenant request. For example, if a tenant creates many objects in a bulk operation in a table that requires locking the same, then this table will be blocked for all the other tenants and any CRUD operation which accesses that table will have to wait till the bulk operation finishes.

Even though our current system supports multi-tenancy, the disadvantages include data privacy issues and hence, to address the disadvantages of the current (above mentioned) approach, we can have a single database but a separate schema for each tenant.
The design and implementation of this 'Single Database, Multiple Schema' approach which ensures data privacy, is discussed below.




Single Database, Multiple Schema diagram [13]

The above displayed structure, where each schema is associated with a single tenant, ensures data privacy and security. In this case, all the schemas will have the same tables, and the schemas can be accessed through a unique namespace.

To implement this, we have to make use of the package called ‘django-tenant-schemas’ as Django does not provide in-built support for serving multiple tenants. The implementation steps are as follows:

1. First, we need to make changes in the settings file where database configuration is defined.

```
DATABASES = {
    'default': {
        # 'ENGINE': 'django.db.backends.postgresql',
        'ENGINE': 'tenant_schemas.postgresql_backend',
        'NAME': 'sms_blasts',
        'USER': 'awaazde',
        'PASSWORD': 'awaazde',
        'HOST': '127.0.0.1',
        'PORT': '5432',
    }
```

```
}
```

2. Adding TenantSyncRouter to database routers defined in the settings file of the project.

```
DATABASE_ROUTERS = (
    'tenant_schemas.routers.TenantSyncRouter',
)
```

3. Adding TenantMiddleware to middleware classes defined in the settings file of the project.

```
MIDDLEWARE_CLASSES = (
    'tenant_schemas.middleware.TenantMiddleware',
)
```

4. After these configurations, we need to define a model (table in database) which tells about the tenant details. We can create a model like this

```
from django.db import models
from tenant_schemas.models import TenantMixin

# TenantMixin has 2 fields- domain_url, schema_name
class Organization(TenantMixin):
    name = models.CharField(max_length=100)
    auto_create_schema = True
```

And then we should define the tenant model name in the settings file.

```
TENANT_MODEL = "organizations.Organization"
```

5. Then, we need to define apps in our project which are to be shared by all tenants and which are to be tenant-specific. The shared apps are going to be created in the default schema called ‘public’ and the tenant-specific schemas will be created in the separate tenant schema.

After that, we define the configurations of these apps in the settings file.

```
SHARED_APPS = (
    'tenant_schemas',  # mandatory, should always be before any django app
    'organizations', # you must list the app where your tenant model resides in
    'django.contrib.contenttypes',
)

TENANT_APPS = (
    'django.contrib.contenttypes',
    # your tenant-specific apps
    'contact', # contact module app
    'blast',   # blast module app
    'account', # login-register module app
    'scheduling', # scheduling module app
)
```

6. Then, we run migrations (means we create database tables) for the app: ‘organizations’ (which stores the tenant details).

```
python manage.py makemigrations organizations
```

7. Then, we run migrations (means we create database tables) for the apps which are shared among all tenants in ‘public’ schema.

```
python manage.py migrate_schemas --shared
```

8. Then, we run makemigrations command for all the tenant-specific apps.

```
python manage.py makemigrations auth
```

```
python manage.py makemigrations contact
python manage.py makemigrations blast
python manage.py makemigrations scheduling
```

Then, if any organization/user wants to register themselves, they have to first add a unique name that depicts its organization and name of the organization. Then, other registration details. The creation of a new tenant will happen as follows. For example, the name of the organization is ‘tenant1’.

```
tenant = Organization(domain_url='tenant1.my-domain.com', # don't add your port or www here! on a local server you'll want to use localhost here
...                 schema_name='public',
...                 name='IBM',
...                 on_trial=True)
tenant.save()
```

After this, if any request comes to the url tenant1.my-domain.com, then it will directly set the search_path of PostgreSQL DB to tenant1 and public. Any CRUD operation that happens with that request will be done on that tenant’s schema. This way, each tenant will have one separate schema ensuring data privacy and security.

## 13.5 Prevention Against SQL Injection

SQL Injection is a security vulnerability which is done by malicious hackers and/or attackers. They interfere with the SQL queries which are made to the database by the application server. This poses a threat to the organization and people as the attack can reveal sensitive information of the users. There are also scenarios where SQL injection attacks not only retrieves data, but modifies and deletes them from the database. Hence, to maintain security and safety, the developers should take preventive measures while building the application such that SQL injection attacks can’t happen.

Some examples of SQL Injection attacks are listed below:

- Retrieving sensitive and hidden data: This can be achieved by modifying the SQL query to return data including sensitive and hidden information.

- UNION attacks: This is performed to extract information from multiple databases.
- Revealing information of database: This is performed to extract the version and other details of the database (like structure, etc) on which the application is working on.

E.g. Example of modifying SQL query to find sensitive information:
This is a login query with username and password

```
SELECT * FROM users WHERE username = 'mike' AND password = 'mike@123'";
```

The attacker can change the quey to the following:

```
SELECT * FROM users WHERE username = 'mike OR '1'='1' --' AND password = 'mike@123';
```

The above query has "OR '1'='1' which means that this will always be true and the double lines '--' means that the rest of the query is a comment. Hence, this will let the attacker access the information of any user and bypass authentication.

In our project, we have used Django as a backend web-framework. It has by default protection against SQL injection attacks, because it supports ORM, i.e., Object Relational Mapping. ORM helps developers to perform database operations such as creating, retrieving, deleting, and updating databases without having to use SQL queries. It transmits data between the application model and the application's database. It does it by using querysets and mapping Django models to database tables. Since SQL query is not explicitly written in these scenarios, Django is able to protect from SQL Injection attacks. [14]
Django's documentation says that the SQL queries are made from python queries using query parameterization [14]. The parameters of the SQL query are defined somewhere else and are used in the query's SQL code. But, since the parameters may be defined by the user, hence, poses another security risk. This is handled by Django by the underlying database driver. The two techniques supported by Django: parameterization of query and

escaping of the parameters by the database driver ensures security against SQL injection attacks.

However, there are some cases in Django where SQL Injection can be a threat. There are some querysets like raw(), extra() and RawSQL for which the above 2 techniques (i.e., query parameterization and escaping of the parameters by the database driver) are not implemented, which is why these queries are not immune to the attacks. Additionally, Django also provides flexibility for users to write direct SQL queries in python, which can also be a problem. In our project, we have not used any of the above querysets and have not written the SQL queries explicitly which makes our application perfectly secure from the SQL Injection attacks.

## 13.6 Engineering Aspects

To increase the performance and optimize our system we have taken following steps from an engineering perspective:

1. Asynchronous scheduling of blasts using multiprocessing:

   We are not sending out messages in synchronous fashion because it results in a very poor performance. To send out messages we have designed 3 tasks that are detailed in section 7.3 . We are carrying out this asynchronous task outside of the request response loop with the help of Django beat scheduler and Celery workers. Django beat scheduler and Celery follow a Producer/Consumer model where the scheduler acts as a producer by putting the tasks in the redis queue and the Celery worker acts as the consumer by consuming the task and assigning it to one of its child processes to perform the task. Since we only have 3 tasks, we are using only one celery worker that has 4 child processes. For our current scenario this is the optimal setting. But to handle more tasks we can increase the number of celery workers as well as the child processes. However it must be noted that the real bottleneck is the hardware and the number of cores it has. So increasing the number of celery workers will definitely increase the performance to some extent but after that we will have to upgrade the hardware to get more performance.

2. Database design:

Database design is one of the very important parts of design since the performance of a system is greatly affected by it. While designing the relationship between the “Contact” and the “User” model, we were faced with a space-time tradeoff. If we followed the database design principle of normalization for database design, then we ended up with a time complexity of $O(n^2)$ for contact upload API, but not following normalization resulted in a better performance since the time complexity was reduced to O(1). A detailed description of this is provided in section 7.5.1

3. Database usage optimization

   In large and complex databases, it is important to reduce the number of database hits using optimisation techniques. The primary reason for this is the cost of database operations in terms of time and resource utilisation. A common problem that we faced while designing and implementing our system was that the normal method of retrieval of all the data of an object made multiple calls to the database to get all the data. This posed an even bigger problem when this retrieval was happening inside a loop. In this section we will focus on two queryset methods provided by django - prefetch_related() and select_related() that eliminates the need for multiple database calls for the retrieval of all the data related to the object. It should be noted that these methods only increase the performance if later in your code you are trying to access the related field data, otherwise it only adds overhead. [11] A detailed description of selecte_related and prefetch_related methods along with the instances of where we have used it in our code is given in section 7.5.2

4. Server calls reduction:

   In complex systems which are created by large organisations with complex business processes and thousands of customers who access the servers at any given single time, it becomes important to reduce HTTP requests to the server to decrease it’s load. Increase in HTTP calls increases server load and increases the latency to load the requested resource. Latency has a substantial impact on application performance. In our system we had put a lot of validations on our

database field, so whenever any new object is created the respective validations are run for it. If we relied on server side validation every time a user registers or creates a blast it would result in too many server calls because everytime user enters invalid data they would only know after the server has performed validations on it. Hence, to reduce HTTP requests, we designed our system in such a way that it incorporates client side validations, which allows rudimentary validation of user data without submitting anything to the server. This allows for more interactivity by immediately responding to users' actions and also executes the requests quickly because they do not require a trip to the server.

5. Following Agile Software Development Process:
   For the rapid development of our project we have followed the Scrum methodology. The development process was executed in a series of 2-4 week sprints. The sprint steps included design, implementation, testing and integration of the modules or sub-modules taken in that sprint. Code reviews were conducted by mentors after development of each module and code review changes were carried out in the next sprint. Daily scrum meetings and sprint retrospective meetings were also held.