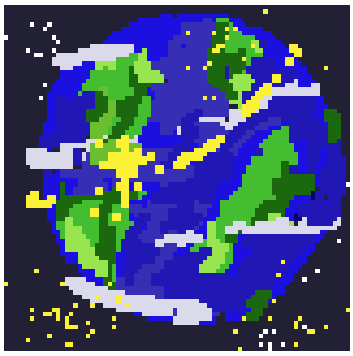
Team HelloWorldByMe



Software Design Document

Version 1

January 17th, 2025

Project Sponsor:

Kevin Daily

Faculty Mentors:

Brian Donnelly

Savannah Chappus

Team Members:

Elizabeth Knight

Joey Banaszak

Jessica Maldonado Olivas

Samantha Madderom

Table of Contents

Introduction

Chronic homelessness is a significant issue, particularly in Tucson, Arizona. Between 2019 and 2020, there was a 300% increase in homelessness in Pima County. When it comes to helping the homeless, many organizations offer services to these people. However, they are often working individually toward the same goal. There should be more collaboration among these entities. Our project aims to provide a method for breaking down the silos of communication that exist between these companies. We will also allow for the compilation of the data collected on the people they serve and interact with for funding and research. Our solution will be focused on Pima County, with the ability to be scaled to larger areas and industries in the future.

Through our solution, nonprofits can employ "Navigators" who go out into the community, gather information on individuals experiencing homelessness due to drug addiction and mental health issues, and connect them with available services. Our project aims to streamline this process by providing a system that allows Navigators to update and access a shared database of the people they interact with. This ensures that, if someone has been previously engaged, their information is readily available to assist in future interactions. Additionally, shelters will be able to update the database with real-time information on available beds, enabling Navigators to secure spots in advance for individuals in need. Shelters can also log details about the services they offer, such as support for overcoming addiction or mental health care. Available services and collected demographic information on those in need will not be segregated by organization, instead, all organizations will be connected and sharing information. In this way, we hope to break down some silos of communication.

Moreover, the data collected will be shared with local government agencies to contribute to existing research and secure funding. By fostering better communication among service providers and supporting data-driven decision-making, we hope this project will lead to improved outcomes for both the organizations and the people they serve.

Our client, Kevin Daily currently works as an Electrical Engineering Manager at CNH Industrial in Tucson, Arizona. In addition to this, he is involved in the community. He, along with Tucson restaurant owner Josh Jacobsen, has volunteered at several nonprofits that target the homeless population. They have seen firsthand the struggles they experience, as well as the unfortunate crime that can arise out of this problem. They noticed that within the industry, there is very little communication between these organizations. In talks with local government officials, as well as leaders in these nonprofit organizations, they decided that something had to be done. This project was created as a result of that need.

Kevin Daily created EKA Labs to hold the patent on technology he had already developed for a previous project. We will be repurposing this concept to allow for ease of communication between these organizations. Since EKA Labs is a small corporation for this specific purpose, it doesn't have a structured workflow. It simply consists of Kevin and Josh meeting with Pima County officials to determine what they want from the solution. They relay that information to us as they get it, and we incorporate it into our planning and design process. The nonprofits have to manually collect and store information, and there is little to no communication between the different organizations.

Implementation Overview

We propose developing a web application that enables users to host an organization or take on a role in an existing organization. This system will allow interactions between users of different roles while providing certain tools assigned to them by their organization's leader. These tools include the ability to take on a 'navigator' role and interact with people suffering from chronic homelessness or drug addiction and route them to the proper facilities to receive specialized help from trained professionals. Additionally, supporting the basic functionality of an organization, and acquiring the necessary information concerning the community being served will be done to a basic degree.

Our solution must address several main functional requirements. There will be an initial login/signup page where a user can create/access their account. The login page will handle log-in and logout auditing to validate their identity. The user interface will have several different tabs, from which the user can access and manage their profile, messages, and roles. A WorldByMe administrator or a specific organization administrator can add more roles.

Our system will collect different types of information on individuals to store a person's history. This information will be available on a single database that will compile all data organizations choose to share. The database will be updated in real-time to ensure accurate information is shared. The database will also store messages from the chat system for up to 30 days, or 11 megabytes. Users can use the chat system to communicate with one another or contact people by phone through the site. Several nonfunctional requirements include enforced password requirements, scalability, personal information security, cross-platform compatibility, and product usability.

For our project, we are utilizing the following common design patterns:

- *Factory Method*
    - Utilizing the factory design pattern would be efficient when defining objects based on conditions.
    - Ex: defining role permissions, or functionalities of the different services.
    - Currently, our user authentication and role-based access control methods follow the factory design pattern since role objects are a key part of the project.
- *Template Method*
    - Having the template design pattern when defining the overall structure of each component could be a good approach.
    - Ex: the process of messaging, account creation, or audit trailing all are the same general process but require different information for each task.
    - Overall, our process of user registration and role assignment demonstrates template logic.

Technologies Being Relied On and Contributions:

PostgreSQL: A SQL-based relational database for storing/managing data

- Need a structured approach to store data about roles, organizations, and the people in need

Django: Developer-friendly backend framework for business logic, role-based access, authentication

- This will allow us to connect database, infrastructure, and frontend services to the user

Apache: Handles requests and applications efficiently

- To be used in tandem with Django to more efficiently route traffic and potentially attach other HTTP-based services

React: Frontend library for making responsive user interface

- Allows for the modular design of the user interface
- Easy to make additions to a dynamic web page

Python: Application logic

- Easy to maintain and highly readable language
- Good choice for cloud-based services where programming language performance is not a requirement

AWS EC2: Hosting application for scalability and performance

- A scalable and extendable compute service hosted by AWS
- Features can be easily extended by connecting other existing AWS services

AWS RDS: Hosting database for reliability and backups

- We will use a managed AWS database hosting service to abstract the challenges of self-hosting infrastructure for a database
- Native auditing and monitoring services that will be utilized throughout the project

Leaflet: Mapping library for visualizing community data

- Allows for an easy-to-use, engaging mapping interface that will be used to connect with people in need
- Chosen because it is a JS library that is easy to integrate within a react ecosystem

Twilio: Messaging API

- The main driver for the messaging feature, our service will communicate with this API to connect individuals
- The API has a free tier and an intuitive interface

To ensure that the functionalities of the roles can be executed effectively, our web application must offer a secure, easy-to-use platform that allows for efficient communication between and within organizations. Achieving this will include developing the following features:

User interface

Our user interface will showcase a modular tab-based design using React, preserving the cross-platform ability. With the messaging system at the forefront of the web application, any updates will be displayed almost in real-time. On the right-hand side, tabs for the profile, navigation, messaging, and role management will be present. The left-hand side will utilize functionalities for messaging, such as deleting messages, sending messages, and drafting messages allowing for seamless communication.

Role-Based Access Control

Each role will be defined in a table in the PostgreSQL database with a personal role being the first official role a user could have. Over time, a user can apply for multiple roles, which would be approved by the WorldByMe admin. Within the PostgreSQL database's permissions abilities, specific roles will be able to see certain "blog posts" and information.

Login/Logout Auditing

The audit trail system will act as an extra layer of user authentication, in which users have to register with a valid email and password that complies with security requirements. With an email verified, the account creation is complete and all future successful and failed login attempts are logged by Django signals, while logouts will trigger a session clear out. Additionally, database modifications are logged by the built-in

PostgreSQL triggers and all audits are stored in a table that will also be stored in Amazon Web Services (AWS) CloudWatch for enhanced monitoring.

Secure Messaging System

Twilio will be the main service for messaging and each user will be able to message other users based on their roles. Attachments will be limited to 4 megabytes with 11 megabytes of storage for messages. Every message over the age of 30 days will be deleted using the AWS storage capabilities.

Data Object/Input Forms

Utilizing HTML/CSS and React, there will be text input portions of the 'navigator' data collection form. Additionally, emails, item searching, and file uploads up to 4 megabytes will be implemented by AWS services and Twilio. Each input form will have input validation to comply with security requirements and will eventually store data that is approved by the Institutional Review Board, a committee that reviews research involving people and ensures that the welfare and rights of the people in need are protected.

Architectural Overview

The diagram below illustrates our system's architecture, which is based on the layered architecture pattern. Layered architecture is known for its clear structure and modularity, which makes it a good choice for systems that require scalability like ours. However, we found that a strict layered architecture didn't offer enough flexibility for our specific use case. Strict layered architecture requires that communications pass through multiple layers, which can become cumbersome when trying to connect certain components. To overcome this, we opted for a variation of the pattern known as the relaxed layered architecture. This design is more adaptable, allowing us to connect components more directly. This ensures a simpler and more efficient flow of information.

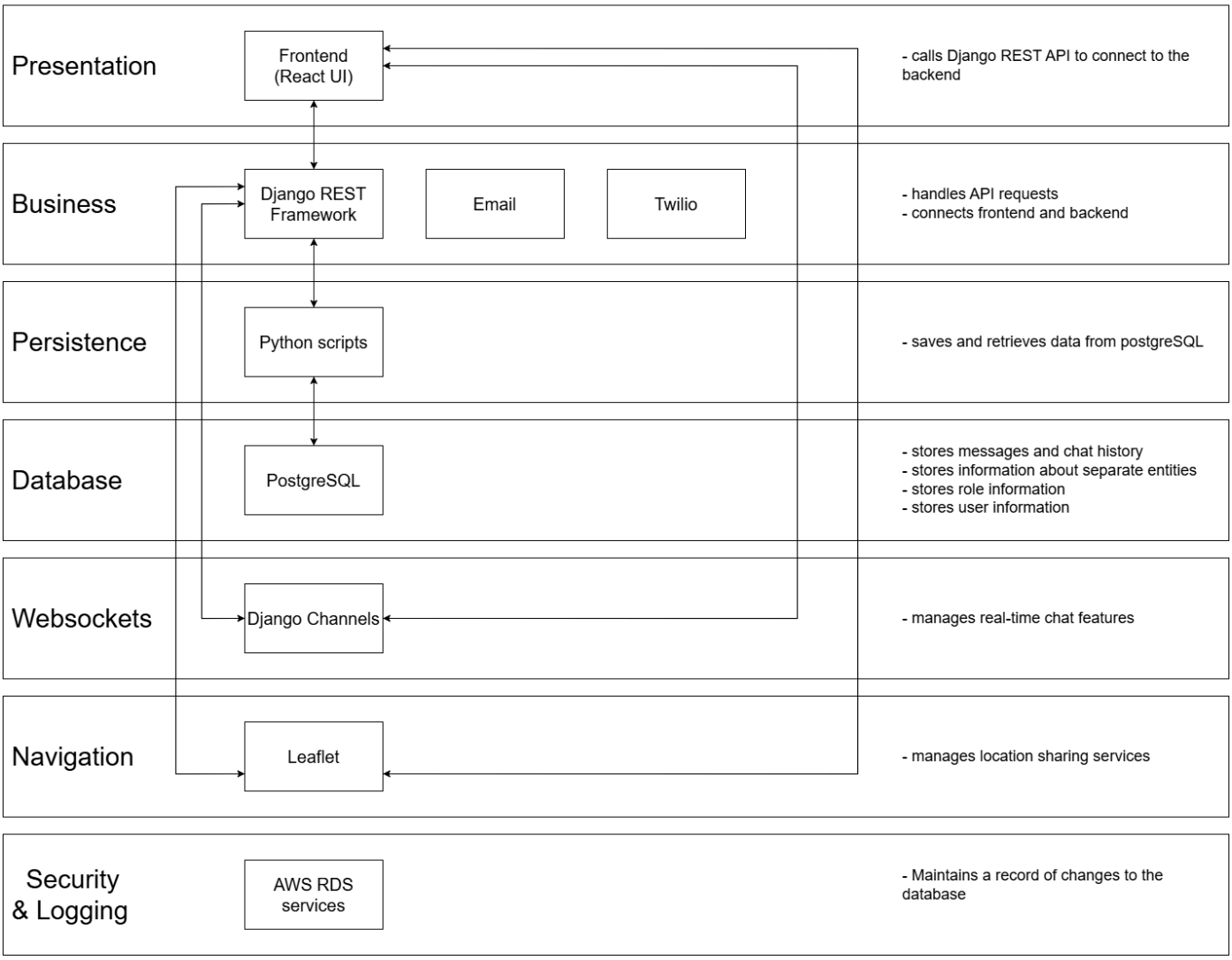


Figure 1: Diagram presenting the architecture of the project

User Interface

The user interface is responsible for facilitating user interaction. It should be easily accessible from desktop, tablet, and mobile devices. Pages should not take excessively long to load. The UI will feature several different tabs to organize the information in a cohesive and intuitive way. The first main page will be a login/sign-up page, from which users can access their accounts. Once logged in, the profile tab will allow users to view their account information as well as which roles they have. There will be a navigation tab for inputting location information. There will also be a messages tab from which the user can create chats, view previous chat history, and delete chats. The system will have a modular design with the specified tabs to the right of the main messaging system. This will allow the user to locate the necessary information from the main

messaging page. Our frontend is written using HTML, CSS, and Javascript. These elements are connected to a main server.js file.

Login/logout auditing

Our system includes a login/logout auditing function, which maintains a record of changes to the database, storing information such as: who made the change, what is their role, when they made the change, and how they made the change. It will use Django signals to alert the system when these changes have been made and store that information.

Messaging

The messaging system is responsible for providing a method of communication between different users. This involves several elements, including the ability to send email and sms messages. The main component, however, is a built-in messaging system that can be accessed from the site. The system will rely on websockets in conjunction with Django Channels to ensure messages are sent, received, and displayed in real-time. Message history will be stored in the database.

Navigation

We will utilize Leaflet for location-sharing services, allowing navigators to mark the last known location of people they come in contact with. This will be embedded into our website using Javascript, and location information that users store will be saved in our database.

Database

The database is a key part of the system and is responsible for storing all information related to the project. This includes but is not limited to: user and user information, roles, data objects associated with certain users, and message history. We are using PostgreSQL for our database which will be hosted on an RDS server through AWS. It can be accessed by different components of the system through the utilization of Python scripts.

Module and Interface Descriptions

User Interface

The front end of our web application is responsible for rendering the user interface, handling user interactions, and facilitating the connections by utilizing HTTP requests. While being built by HTML, CSS, and JavaScript, the management of the UI components, HTTP requests, and the state is done by the JavaScript library, React. The user interface additionally facilitates the sending and receiving of data that is retrieved by React's connection to Django's REST Framework (DRF). As the data is flowing through the application, Django's channels and WebSockets provide real-time messaging, ensuring that communication can be done without constant page refreshes. The navigation system will be interconnected with the UI using Leaflet.js which will provide a way to record proper geolocational data with ease. Communication with the database will happen directly when retrieving and storing data when utilizing the messaging and navigation systems.

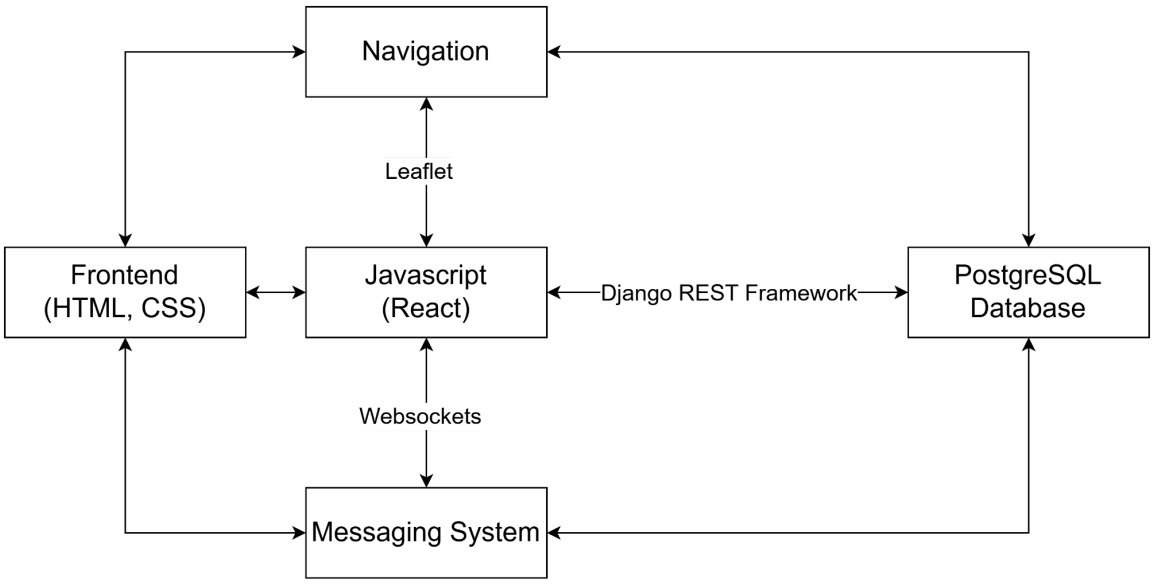
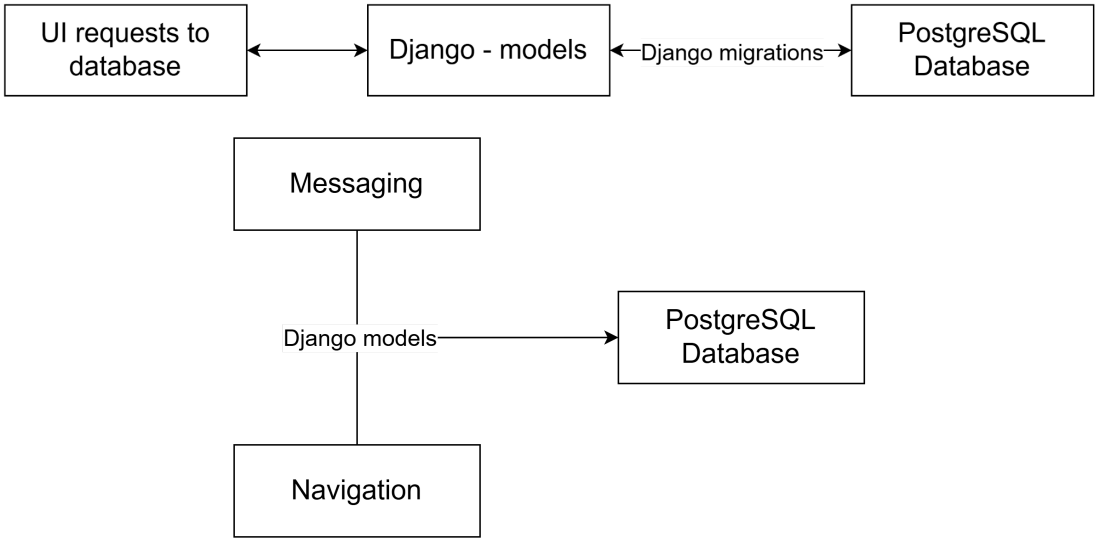


Figure 2. Showing how the UI connects to the rest of the system

The Django architecture that makes up the systems architecture can be split into various major components. First and foremost are the models, which handle database interactions. Next are the Django views which allow us to process requests and responses from the user via the UI. The URLs and middleware can handle both routes to specific pages and authentication requests. The middleware can also be used for security purposes, such as hashing passwords. Django handles all the backend logic, while React manages the frontend user interface. The connection comes from the Django REST framework which uses an API to communicate between the two.

Database

The database is set up using AWS Relational Database Services to host PostgreSQL, which Django can connect to remotely. This allows Django to use the views instance to connect the RDS instance seamlessly. As mentioned before, Django can create middleware that hashes and secures passwords which can connect to the database directly for the security of the users. Each middleware module is responsible for performing a different security function for the software's backend. For instance, all string based queries and items stored in the database are screened to prevent intrusions like SQL injection attacks. The middleware functions are able to take requests and return specific responses, just as the view module Django uses.



Figures 3-4. Showing how the database connects to the system through models

Django manages database changes using migrations and Object-Relational mapping. When defining models using Django, migrations are specifically used to propagate changes made to the database/models. After the RDS instance is connected to the specific backend file, a migration is run to the tables in the RDS instance. Models are then created for each needed value, like companies, users, and roles, depending on the specific value needed to be integrated into the database.

The database will have numerous tables for storing different types of information, including, but not limited to: role definitions, user information, user roles, passwords (hashed), different companies, message history, and chats, as shown in figure 5 below.

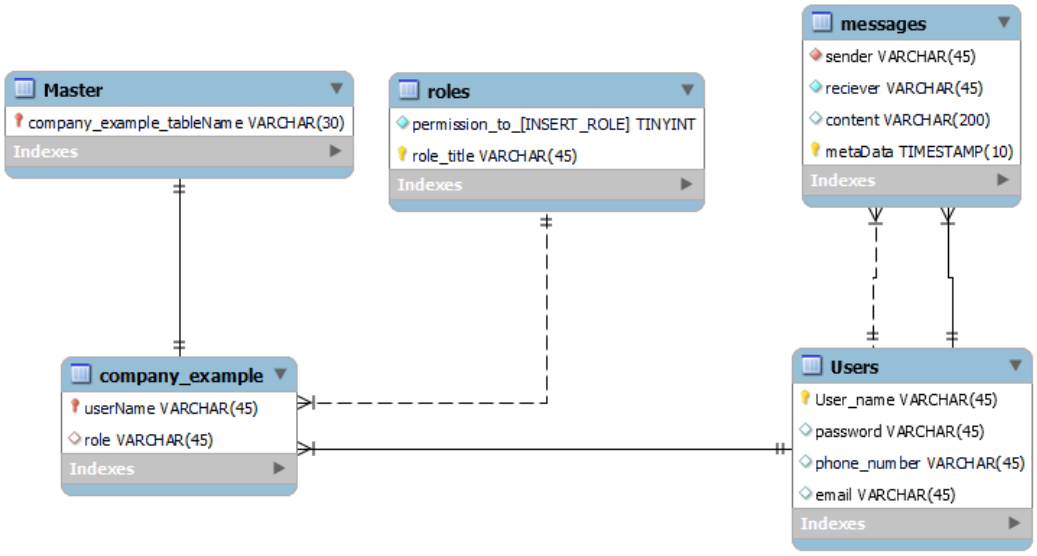


Figure 5.Showing the design of the database

Messaging

Messaging can be broken down into 3 major parts. The first is the main messaging system. This will be used to send messages directly to and from users through our site. It will use a combination of Django Channels and websockets to accomplish this. The UI Javascript files will subscribe to the necessary websockets and will be notified when there is a new message to display, as well as communicate the sending of a message. Django Channels routes the message request to a consumer, which is a special kind of Django view that handles WebSocket events. Django Channels uses Redis to handle real-time communication. Redis keeps track of active WebSocket connections and ensures that messages are properly distributed to the correct chat rooms. The frontend updates the chat log instantly without needing to refresh the page.

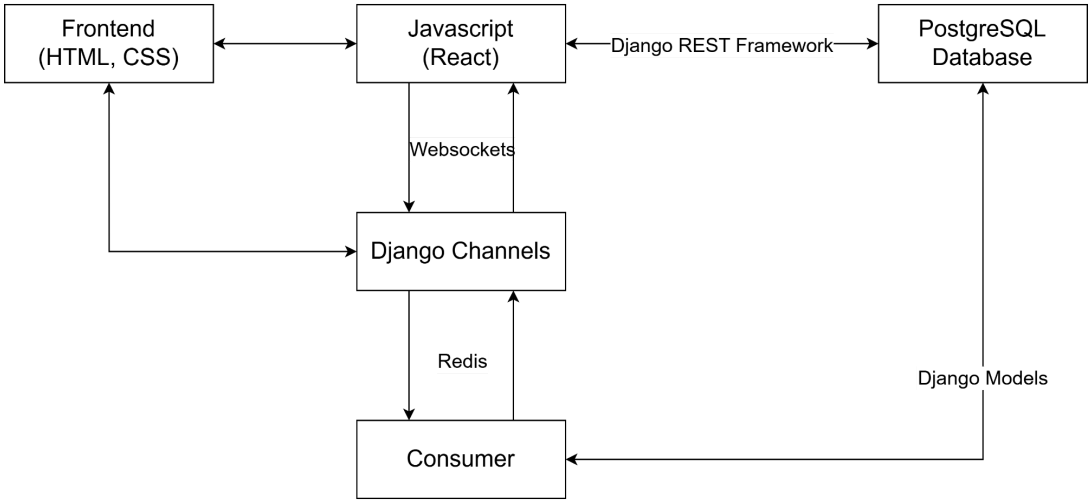


Figure 6. Showing how the messaging system connects to the frontend and database

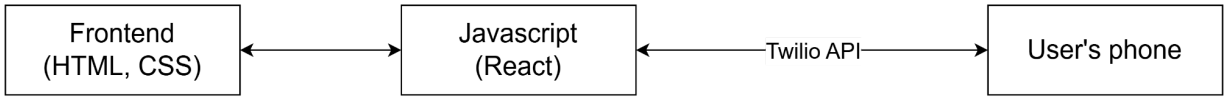


Figure 7. Showing how the Twilio API fits into the system

The next major part of messaging is the option to send SMS messages. For this implementation, we are utilizing Twilio, a paid API, to handle the message sending for us. Possible use cases for this include password recovery and account verification. This will also be used to notify users of available housing and resources. This will be integrated into our already existing Django framework, creating a utility function, and using React to trigger SMS messages from the frontend.

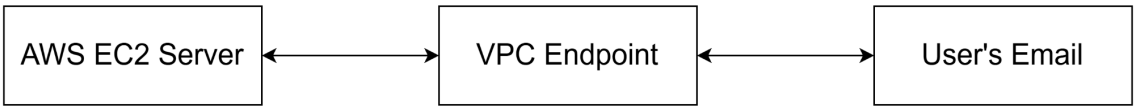


Figure 8. Showing email setup

The third and final element of messaging is the option to send emails. This has a similar use as above. To handle sending emails, we will use the AWS email system, known as Amazon Simple Email Service (SES). We will open a Virtual Private Cloud (VPC) endpoint through which any emails can be sent.

Navigation

The navigation system utilizes the Leaflet JavaScript library to create our interactive mapping interface, focusing on locational-based tracking. It provides the user the ability to put in latitude and longitude coordinates and displays a marker on the map. This system was designed so that the users can visually understand the general whereabouts of individuals in need and which services would be of most help, given the location of the person. The key functionalities of the navigation interface include:

- ***InputFormHandler*:** The `getInputData()` method prompts an individual for general information, such as name, a description of the services they need, and the coordinates of the location.

- ***Map display*:** The map is centered on the city of Tucson and allows users to zoom in and out of the map as they please. The **OpenStreetMap tiles**, **a geodata API's grid system**, are responsible for dynamically updating the map background within Leaflet as the city changes.

- ***AddMarker***: After putting in the manual location and clicking on the 'Add Marker' button, the **AddMarker class** triggers the `AddLocation()` method. Utilizing the Nominatim geolocation API, the coordinates' geocode is processed into a readable address in Tucson, such as the nearest QuikTrip or Bashas'. A marker is then placed on the map and the **PopUpHandler class** attaches a pop-up using the `bindPopUp()` method, displaying relevant information and the coordinates that were given.

- ***ClearForm*:** Right beside the 'Add Marker' button is a 'Clear Marker' button which triggers the `clearMarkers()` function in the **ClearForm class**. The `clearMarkers()` function allows a user to remove all markers from the map.

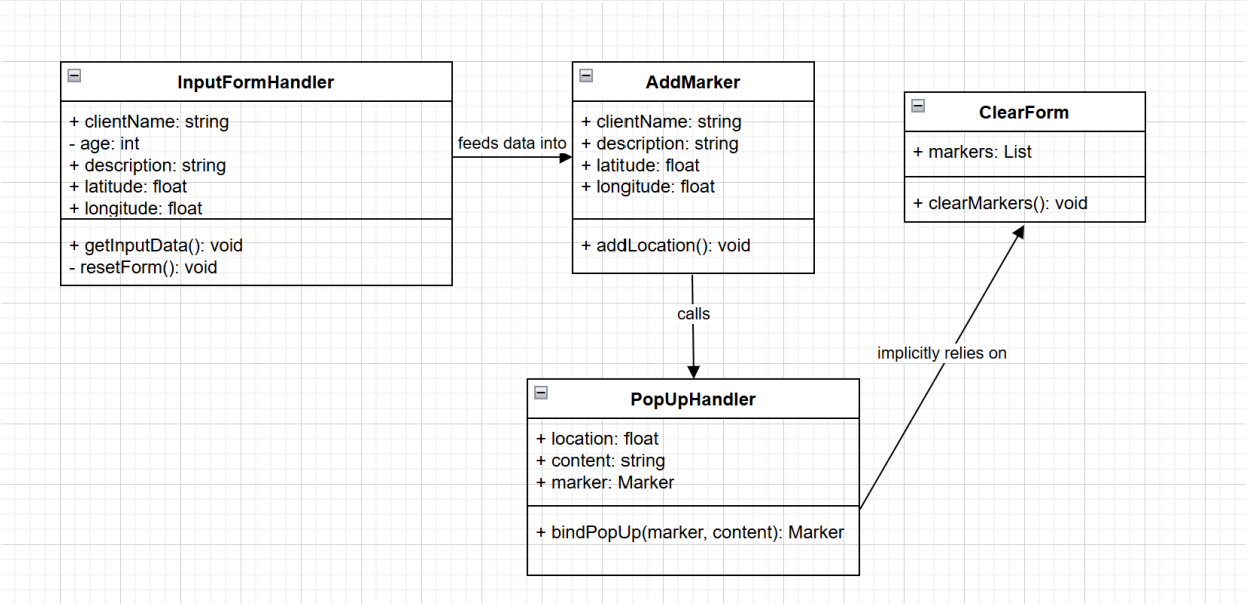


Figure 9. Navigational System Overview

As seen in Figure 9, the **InputFormHandler** is mainly responsible for gathering the responses from the form on the interface. However, it also serves as the source of data for new markers. The key steps of the workflow are included below:

- **InputFormHandler** gathers user responses from the form and feeds the data into the AddMarker function.
- Moving on to the **AddMarker** class, creating a new marker based on the latitude and longitude from the **InputFormHandler**. Figure 9 highlights how **AddMarker** directly relies on the **InputFormHandler** and calls the **PopUpHandler.**
- **PopUpHandler** finishes the marker creation process, binding the additional information to the marker made in **AddMarker**.

- Finally, when a user is done, they're able to clear out all the previous markers using the **ClearForm** class. The clearing component directly relies on the public Markers array to identify and remove markers but also implicitly relies on the **PopUpHandler** to be able to remove all the pop-ups as well.

Implementation Plan

When defining the implementation plan, it was evident that each component needed to be done at a specific pace and worked on in pairs at the very least. Along the way, we intend to approach certain implementation challenges with a more spaced-out timeline to ensure that more help can received in time, collaboration with each other, and in-depth research methods. Considering past work done together, we find it easier to have team members who specialize in certain areas, which are defined by the different color bars below in Figure 10. Each elongated bar represents each component we are to focus on with each of the implementation steps displayed below them. Testing and validation will be done with limited cycles after each iteration of components is completed but will be fully tested throughout the testing and validation phase. With this implementation plan in place, we aim to complete a robust, scalable role-based access system that will provide services to the Tucson community, extending to state and national levels.'

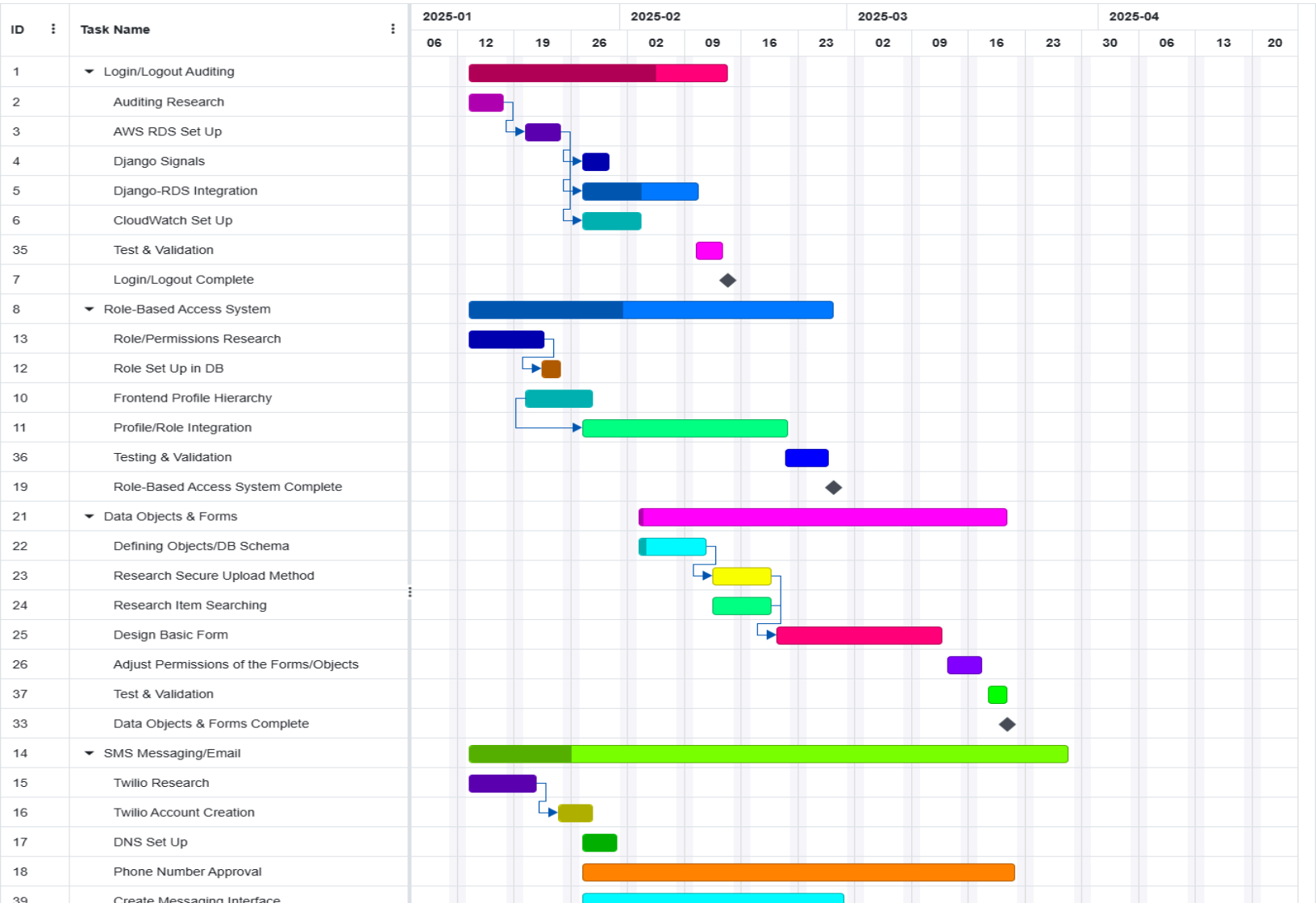


Figure 10: Implementation Plan Gantt Chart

When the state of the overall architecture was revisited, we found that hosting the database on the same instance as the server would reduce the project's scalability. Additionally, it would create various issues when implementing the login/logout auditing, SMS/email messaging, and even the current data objects and forms. The constant manual implementation of each component would reduce the quality of the architecture and the effectiveness of the maintenance it requires. Overlooking the integration of all the components is an essential part of making the final product possible.

Login/Logout Auditing

When setting up our audit trail system, we first had to utilize a more efficient AWS architecture when it came to hosting our application and database. Therefore, our hosting service remains with an AWS EC2 instance and the database resides inside our AWS RDS instance. From there, built-in PostgreSQL event triggers were implemented to record any changes to the database, and Django login signals were developed to track login/logout activities. One remaining part of the login/logout auditing system that is left is connecting the Django signals with the database's audit table. Lastly, CloudWatch, a AWS monitoring service was initialized during the modification of the AWS RDS settings so that the PostgreSQL logs were connected to the CloudWatch logs.

SMS/Email Messaging

Implementing SMS messaging has not been a hard task, as most of the functionality is being paid for through Twilio, a third-party service. We have been working together with the client on setting up the Twilio account but the full implementation won't be completed until we have more development on the frontend and backend aspect. As for the email feature, the most plausible option would be to utilize Twilio Email API and for a backup plan, the AWS SES (AWS Email) could work.

Data Objects and Forms

Developing more specific data objects relies more on the structure of the database and role-based access. As more of the architecture of our project develops, we will be able to implement more enhanced features such as file uploads and item searching. However, we have not made much progress on this yet and our plan for implementation is very general. In the coming weeks, we should have a more concrete idea of how to accomplish

this part. When it comes to the forms, HTML/CSS with React to make the forms more engaging. Input validation will be utilized to ensure consistent and valid data is typed in.

Testing and Validation

Throughout the final stretches of each phase of development, the testing and validation of each component will be a continuous and iterative process. Each component will have at least 2 different kinds of validation to ensure that each workflow is working properly. The main kinds of testing and validation include unit testing, integration testing, database testing, and front-end UI testing. Utilizing manual and automated testing, our approach will make sure that bugs/issues are caught early, components are integrated smoothly, and our project remains effective and reliable.

Conclusion

In summary, there is an epidemic in Tucson, Arizona with homelessness, people suffering from severe mental illness, and chronic drug addiction. There are many different organizations in the area with goals to remedy these issues, but they all operate separately, with little to no communication between them, which bogs down the process of helping these individuals. This problem is not limited to Tucson, however HelloWorldByMe is planning on addressing the issue in Tucson first and being expanded to help communities globally in the future. HelloWorldByMe will allow organizations to join the platform with ease, once added to our database they will then, through leaflet, have access to a multitude of other organizations in their surrounding area. Using Django Channels, users will be able to communicate with other users in their organization or reach out to others who may not be in their organization and would not be able to reach otherwise. Our database will store contact information as well as permissions for each user so if a user is permitted to be communicating with another, they are able to in several different ways such as email, sms messaging, or HelloWorldByMe's native messaging system. We already have a working prototype of all the major components discussed, going forward we need to connect all the pieces together and get feedback from our client after some field testing. Once our client has what he is looking for, HelloWorldByMe can be used by employees, organization leaders, and numerous other entities to better communicate with each other and be able to provide the proper care to their community much faster and more effectively than how they are now.