GEOGRAPHIC LOCAL ROUTING FOR SOCIAL CONNECTION:

A NOVEL APPLICATION FOR THE INTEGRATION OF ROUTING TECHNOLOGY AND MULTI-USER ENVIRONMENTS

by

Moises Herrera

A Thesis Presented to the
FACULTY OF THE USC DORNSIFE COLLEGE OF LETTERS, ARTS AND SCIENCES
UNIVERSITY OF SOUTHERN CALIFORNIA
In Partial Fulfillment of the
Requirements for the Degree
MASTER OF SCIENCE
(GEOGRAPHIC INFORMATION SCIENCE AND TECHNOLOGY)

May 2024

Table of Contents

List of Figures

Abbreviations

CLI	Command line interface
CORS	Cross-origin resource sharing
GIS	Geographic information system
GLRSC	Geographic local routing for social connection
GUI	Graphical user interface
IDE	Integrated development environment
NSASC	National Strategy to Advance Social Connection
ONN	Online neighborhood networks
ORM	Object relational mapping
OSNN	Online social neighborhood networks
URL	Uniform resource locator

Abstract

Geographic Local Routing for Social Connection (GLRSC) is an innovative and novel application which integrates road routing algorithms and multi-user digital environments to facilitate immediate and thematic social networking opportunities for users within a walkable reach and a 30-minute period. In 2023, the U.S. Department of Health and Human Services declared loneliness as a public health epidemic; the declaration demands research and innovative action to help Americans connect with one another through face-to-face contact. This thesis develops GLRSC-System-1, a deployed full-stack system that integrates GLRSC and introduces the application online. GLRSC-System-1 helps users increase their opportunities for social encounters by providing a meeting request system where users can request thematic meetings with other users within a 2km neighborhood. The system calculates an optimal midpoint for active participants based on their geolocation and synchronously routes the users to meet at an optimal midpoint within a 15-minute period. The thesis introduces the essential components required of any GLRSC system through the phase planning methodology of GLRSC-System-1. The essential components/phases for a base-level GLRSC system are a multi-user environment, system deployment, a meeting service, and a routing service. This project demonstrates the feasibility of implementing GLRSC in online systems and serves as a guide for the future development of GLRSC technologies.

Chapter 1 Introduction

According to an advisory report by U.S. Surgeon General, Vivek Murthy (2023), the U.S population is experiencing an epidemic of loneliness that poses grave risks to individuals and their communities. There has been neither investment, innovation, nor research investigating how to build technology that can increase social connection with reliable methods to measure effectiveness and safety. However, new technologies have emerged that provide experiences for social connection within a user's walkable reach. For example, location-based games and dating applications provide experiences for social connection but are not intentionally built to address the loneliness epidemic. In fact, the epidemic has persisted despite the advances in location-based social media technology.

Additionally, new community building platforms, such as Nextdoor, Nebanan, and Mapbuzz, use geographic data and a user's location to place individuals in online social networking groups overlayed onto the user's actual neighborhood. However, these platforms have not reliable measured the effectiveness of their platforms in quantifiable or theoretical ways. The loneliness epidemic is ongoing with no viable evidence that suggests that our present digital social environments have alleviated its pernicious effects. Nevertheless, geographic applications such as PokemonGo, Tinder, Uber, and Nextdoor help demonstrate how software developers can apply the user's geographic information, spatial analyses tools, and design new software architectures, to create new systems that drive face-to-face social connection and experiences.

This thesis present Geographic Local Routing for Social Connection (GLRSC) as a novel application which introduces the integration of routing technologies and multi-user environments for facilitating immediate and thematic instances of community engagement within walking distance of the user's location.

GLRSC-System-1 was developed using a phase planning methodology. Software development is divided into four phases: (1) multi-user environment, (2) deployment, (3) meeting service, (4) routing service. Phase planning helps to establish timelines, deadlines, and ultimately optimize work in both team and individual settings (Biondi 2021). For the case of this thesis, phase planning also introduces essential GLRSC components required in any base-level GLRSC system.

The primary goal for this thesis is to introduce GLRSC as a novel application in the integration of routing services with multi-user environments that can potentially increase the user's opportunities for social encounters on desired subjects. This thesis also aims to introduce necessary GLRSC system components through its methodology so that the future readers and developers can more easily build GLRSC-like systems. The tertiary goal is to deploy GLRSC-System-1 online as a learning resource, methodology, and example for building GLRSC technologies.

1.1 Motivation

Social connection is a necessary human action, event, and state for sustaining personal and collective mental, physical, and social health; this fact is backed by both sociological and empirical perspectives. From a sociological perspective, humans evolved within tribes which coordinate efforts to improve their chances of survival (Kaspersen 2008). If a human experiences social disconnection or loneliness there is both an impact on the individual, who experiences a shortage in support from the tribe, and an impact on the tribe, which loses an active contributor in the affairs of survival. From an empirical perspective, researchers and institutions have established loneliness as a risk factor for physical and mental illness (Masi et al. 2011). The health hazard for an individual experiencing loneliness is analogous to that of an individual who smokes up to 15 cigarettes a day (Holt-Lunstad 2017). Loneliness increases the risk of heart disease by 29% and the risk of a stroke by 32 % (Valtorta 2016). On the collective level, several studies link communities who have poor social capital with a decreased ability to face public health outbreaks, respond to natural hazards, create jobs, and establish representative government (NCOC 2011; Kim 2019; Makridis 2021)

Social connection is an urgent topic that requires immediate awareness and action. In 2023, the Office of the U.S. Surgeon General published an advisory that declared loneliness as a public health epidemic (U.S Department of Health and Human Services 2023). The advisory lists a wide range of social impacts and worsening U.S. connectivity metrics. For instance, the advisory reports that the epidemic has at least an economic cost of $6.7 billion in excess Medicare spending annually due to increased hospital and nursing facility spending for socially isolated elders (Shaw et al. 2017). From 2003 to 2023, time spent alone has increased by 11.7% and time spent on in-person social contact has decreased by 200% (Kanaan 2023). Additionally, studies suggest a half of American adults experience loneliness (Cigna 2021). These statistics highlight a deteriorating state of social capital in the U.S. However, despite the overwhelming amount of evidence that documents that human social relationships are a top health determinant, and that healthy human social relationships are in a decline, there is a slow process to acknowledge the epidemic (Holt-Lunstad 2017).

Nevertheless, loneliness presents both challenges and opportunities. While loneliness is linked to adverse health outcomes for the self and the community, social connection, the anti-thesis of loneliness, is a determinant of population health, community resilience, community safety, economic prosperity, and representative government (U.S Department of Health and Human Services 2023). Therefore, addressing loneliness is not just a personal health challenge, addressing and curing loneliness by improving social connection is an opportunity for collective social advancement. Thankfully, steps are being taken to address the epidemic on a national scale. The U.S Surgeon General's Advisory on loneliness establishes the National Strategy to Advance Social Connection (NSASC) which lists six pillars to advance social connection and provides a list of recommended action for specific stakeholder groups including governments, health workers, public health departments, research institutions, philanthropy, academic departments, workplaces, community-based organizations, technology companies, media industries, guardians, and individuals.

The first pillar calls for strong social infrastructure in local communities; it includes supporting pro-connection urban design, community programs, and local institutions. The second pillar calls for pro-connection public policies; it includes viewing social connection as a cross-departmental issue that can be incorporated into all policies across various levels of government. The third pillar calls for mobilization in the health sector; it includes training providers, supporting patients, and increasing surveillance and interventions. The fourth pillar calls for a reformation of digital environments; it includes supporting pro-connection technologies, integrating data transparency, and ensuring user safety (U.S Department of Health and Human Services 2023). The fifth pillar calls for public education on social connection; it includes developing a national research agenda, accelerating research funding, and increasing public awareness. The sixth pillar calls for a culture of connection; it includes cultivating values that increase connection, modelling the values in one's personal life, and expanding conversations on social connection.

This thesis is concerned with advancing and contributing to Pillar Four, reforming digital environments. The call for a reformation of digital environments can be attributed to evidence suggesting that some online social media platforms can produce several social negativities that deepen loneliness (U.S Department of Health and Human Services 2023). The U.S. Surgeon General's Advisory claims that current digital environments may be displacing in-person engagement, monopolizing the population's attention, reducing the quality of human interactions, diminishing self-esteem, and undercutting face-to-face engagement (Duke 2017; Uhls et al. 2017).

Technology insiders have also commented on online platforms negative social impact. For instance, ex-president of Facebook, Sean Parker, claims that social media sites integrate addictive social validation feedback loops and violate user privacy (Solon 2016; Vaidhyanathan 2019). Social media tech-experts, Justin Rosenstein and Nir Eyal, agree that social media sites are created to be addictive. Furthermore, they agree that social media rewards addictive behavior, exploits psychological vulnerabilities and lowers the cognitive ability of focus (Brichter 2017). Former Google employee, Tristan Harris (2017), comments, "It's changing our democracy, and it's changing our ability to have the conversations and relationships that we want with each other." Further evidence shows that social media use is positively correlated with an increase in loneliness (Hunt 2018).

Asserting the truth of claims regarding social media is beyond the scope of this study. However, there is evidence that suggests that digital environments can be improved to support positive social connection. NSASC's Pillar 4 outlines three key actions to help align digital environments with public health needs. The pillar calls for individuals and institutions to support the development of pro-connection technology, integrate data transparency, and ensure user safety (U.S Department of Health and Human Services 2023). This thesis aims to support the development of pro-connection technology through the development of GLRSC-System-1. The system integrates some features that promote data transparency such as features that communicate algorithmic processes. It is also designed to increase the instances of social encounters for users. At the time of this thesis, GLRSC-System-1 is not released to the public but must be made safe for users.

The platform audience is the research community. GLRSC-System-1 is made to engage geospatial professionals, software architects, social connection coalitions, public health experts, and routing experts in a conversation on how to harness digital routing services with multi-user environments to improve social connection in local communities. This paper includes topics such as geographic information systems, software engineering, routing algorithms and social connection advocacy. Some familiarity on the topics previously mentioned is useful.

1.2 Study Area

GLRSC-System-1 relies on Wi-Fi internet service to obtain the user's location. The user's location is then used to access road network data from OpenStreetMap. OpenStreetMap holds data on a large percentage of the Earth's regions. Therefore, the system is theoretically functional across the globe, but its performance across regions of the Earth is not examined in this study. This study provides a study area for running the system's services which is seen in Figure 1.

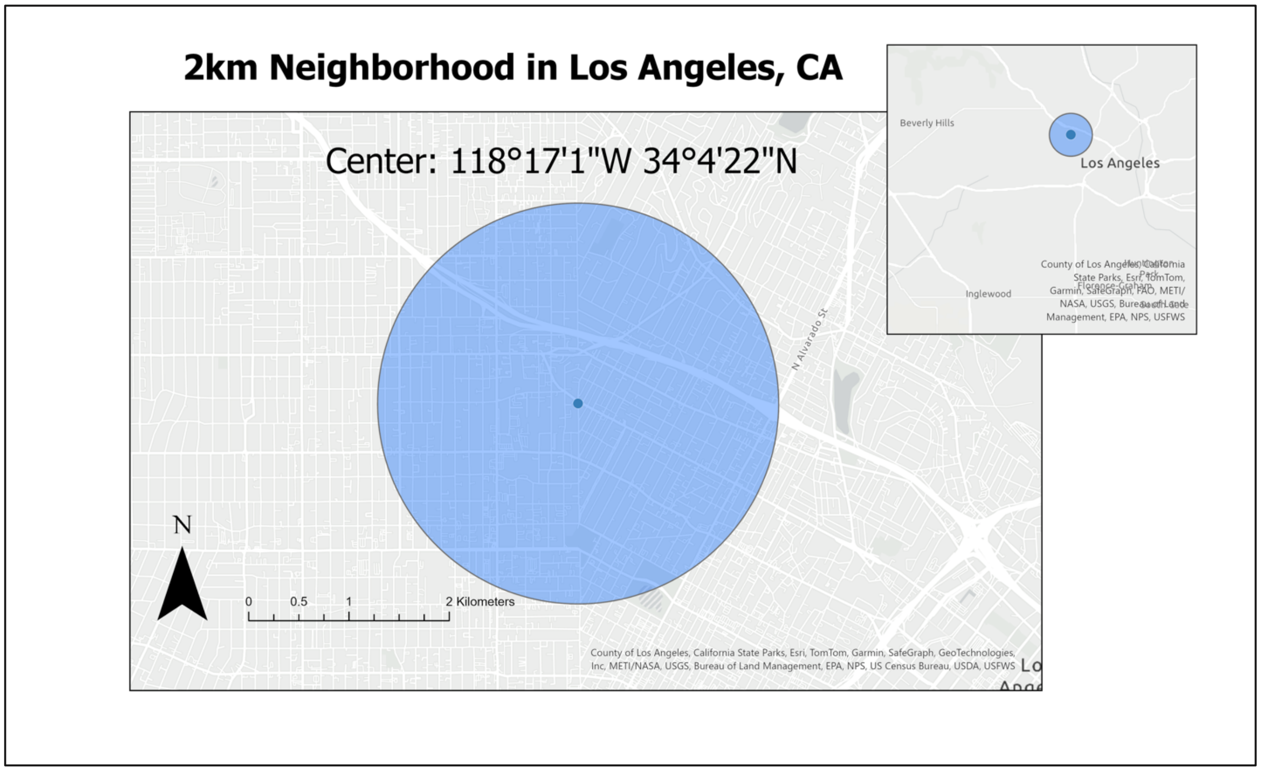

2km Neighborhood in Los Angeles, CA
Center: 118°17'1"W 34°4'22"N
N Alvarado St
N
0
0.5
1
2 Kilometers
County of Los Angeles, California State Parks, Esri, TomTom, Garmin, SafeGraph, GeoTechnologies, Inc, METI/NASA, USGS, Bureau of Land Management, EPA, NPS, US Census Bureau, USDA, USFWS
Beverly Hills
Los Angeles
Inglewood

Figure 1. Los Angeles 2km neighborhood centered at 34.0422 N, -118.171 W

The spatial context is around the area known as Los Angeles, CA, United States, 34.0422 N, -118.171 W. System usage is restricted to a 2km radius area from the user. The results chapter includes images of the study areas, regions close or near 34.0422 N, -118.171 W, used in system testing.

1.3 Data

The sole data source used for this project is road networks from OpenStreetMap. OpenStreetMap is used as both a software and a data source because it is an open-source web application that has a database that contains road network data from around the world. Because OpenStreetMap is open-sourced this project may use its data freely. The data itself is continuously updated, and a timestamp is not found to address the temporal scope of the data.

Finally, Python and JavaScript scripting were used make API calls to OpenStreetMap for data processing.

1.4 Methods Overview

In this thesis, GLRSC is presented in its prototype implementation GLRSC-System-1; GLRSC-System-1 is a desktop web system that provides three distinct services to users: an authentication and multi-user environment, a requestor/responder meeting service, and a route visualization service. The system processes the following actions in order: allow the user to generate an account and login, allow the user to request and/or respond to a meeting request to/from nearby users within 2km, and provide the user with a route visualization from their location to an optimal meeting midpoint along the road network. Through the sequence of actions, the system provides the user with an opportunity to experience face-to-face contact and meet residents in their neighborhood, defined within the system as a 2km radius area from the user's location. Facilitating face-to-face contact is a known strategy to improve social connection and reduce social isolation and loneliness (Masi et al. 2011, 219). Therefore, GLRSC-System-1 attempts to alleviate loneliness users and increase their feeling of being socially connected.

GLRSC-System-1 is built using open-source technologies such as PostgreSQL, React, Flask, Github, and the free version of IntelliJ IDEA, an integrated development environment (IDE). In this thesis, open source is defined as freely accessible and derivable technology (Maurya et al. 2015). The system also relies on open-source data such as OpenStreetMap. OpenStreetMap data is processed through the system's back-end architecture which also relies on spatial data support systems such as PostGIS, a spatial extension for PostgreSQL for storing spatial data. The system uses other supporting open-source spatial technologies such as

Geoalchemy2, OSMnx, and Mapbox, which are incorporated and described throughout the course of the thesis.

1.5 Document Overview

This first chapter has described the motivations, objectives and data for this research. The second chapter discusses related work on online neighborhood networks, routing algorithms, and existing routing products. The third chapter describes the project requirements which includes a section describing the wireflow developed for the system. It also covers use case scenarios, a specialized routing algorithm, and software requirements. Chapter Four describes the four phases of development. Chapter Five shows the resulting GLRSC-System-1 with screenshots of a successful implementation. Chapter Six discusses future research targets, shortcomings, and research implications.

Chapter 2 Related Work

This chapter reviews four groups of related work. Section 2.1 reviews online neighborhood networks (ONN) which are a set of online social media platforms that are also a type of geographic information system (GIS) geared towards improving social connection at the neighborhood level. Section 2.2 introduces scholarly work optimizing routing algorithm by describing the historical evolution of routing algorithms and the reasons why optimization is crucial for its applications. Section 2.3 introduces two web routing applications described in graduate theses projects that are technically and architecturally comparable to GLRSC-System-1. Section 2.4 reviews commercial web/mobile routing products such as Uber and Instacart; they constitute exemplary commercial work and highlight the impact that web routing applications can have on society. Section 2.4 also addresses long term considerations and goals that should be made in the development of successful routing technology.

2.1 Online Neighborhood Networks

Online neighborhood networks (ONN) are online social networking and media platforms that aim to build community at the neighborhood scale. They are important to this work because they provide a view into the current state of the art in applying GIS to increase social connection. Some common names include Nextdoor, Nebanan, and Hoplr.

There is evidence that ONN are consciously designed to address the issue of loneliness and the need to reform digital environments to improve social connection. For instance, company websites, such as those for Nextdoor and Nebanan, include research on loneliness and social connection (Nextdoor, n.d.; Vollman 2018). Therefore, ONN serve as great examples of consciously designed GIS that work towards addressing the loneliness epidemic.

The term ONN refers to the ecosystem of online social media platforms that encourage digital interactions amongst users that all reside within a local geographically defined area. De Meulenaere (2021) coins the term ONN to refer to self-initiated online communities where individuals can interact with nearby residents, such as a city-based Facebook group or a local WhatsApp group. However, Vogel (2021) advances the topic further and delineates a more comprehensive taxonomy for ONN which includes company-initiated platforms as well as self-initiated platforms. This research paper is primarily focused on company-initiated platforms because of the architectural and engineering work that is required to build them. While this paper uses the definition laid out by Vogel, it does not use Vogel's modified term, online social neighborhood networks (OSNN). ONN is used instead because it is more compact and equally descriptive as OSNN.

While ONN promises to build community for users, their strategy, it seems, mostly consists of garnering online user engagement. The appeal to many users is that even though the relationships and conversations begin or remain online, the users found in ONN are the actual people existing in their spatial surroundings which promotes community relevancy. In some cases, one can imagine that encounters in ONN lead to live face-to-face contact. However, there have not been reports nor regulatory actions that measure the efficiency of ONN to build offline face-to-face communities. No reports or studies have been produced or released from ONN nor the public on ONN effectiveness.

Despite the absence of concrete data that documents ONN in improving social connection, ONN have been viewed optimistically as next generation social media platforms. ONN are sometimes seen as a natural progression for internet communities because they place online communities in tangible local environments (Sachdev 2020). They have gained international appeal; popular ONN have emerged in various countries. For instance, Nebanan is a popular ONN in Germany; Nextdoor, in the United States and Spain; Hoplr, in the Netherlands; and IamHere or Simply Local in India. Cataloguing a complete list of existing ONN at an international level is beyond the scope of this study. However, the Appendix provides a table listing existing ONN platforms.

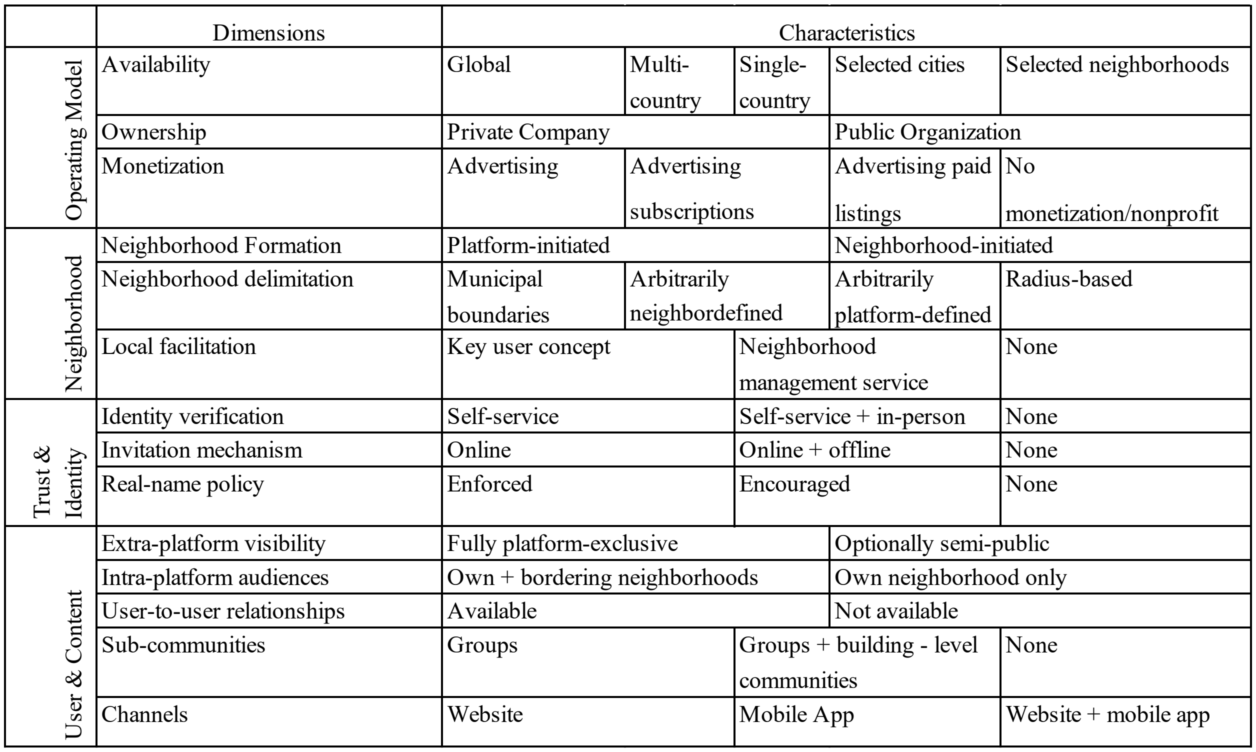
More than just highlighting the emergence of ONN in the digital ecosystem of inventions, the Appendix helps show how various institutions see a value in online neighborhood or hyperlocal networking. ONN platforms commonly use the term hyperlocal to describe a new method of digital networking that focuses on the user's immediate spatial surroundings, the largest spatial scale that includes a user and the small subset population that resides around them. ONN sometimes define hyperlocal as a 1-mile radius or sometimes even use government-housed data to demarcate actual political neighborhood boundaries within a city.

The platform descriptions provided in the Appendix, next to the ONN name, highlight their potential public value; the descriptions were gathered from the platforms themselves. The captions help show the enthusiasm behind these platforms, and it is quite common for ONN founders to feel enthusiastic about the value that their platforms bring. For example, Founder of Simply Local, Nikhil Bapna, is excited about the future of ONN, and claims that the future of social networking is within local micro-communities (Sachdev 2020). Micro-communities being yet another term that has been coined within the domain of ONN to refer to a hyperlocal geographically bound community of people where only distance matters in defining a social bond. He also mentions that streamlining communication at the local level makes communication more efficient. The official website for Nextdoor mentions that their platform is an opportunity to build stronger, more vibrant, and more resilient neighborhoods (Nextdoor

2023). While Michael Vollman, founder of Nebanan, mentions that ONN can help users build social capital nearby (Vollman 2018).

Similarly, ONN founders or proponents find that ONN offer a new potential to improve more popular social networking and media sites. Vollman (2018) critiques social media networks, such as Facebook or Twitter, as platforms plagued by ego-centric self-presentation, narcissism, and anonymous populism. Conversely, he comments that ONN are free from these characteristics. Such a claim is not one made in this thesis; in fact, ONN have engendered a ton of critique including racial profiling (Lambright 2019). However, it is a fact that proponents of ONN, including some ONN founders, believe that healthier social networking should exist and have worked to create digital social networking modalities that are healthier for society than existing modalities.

While ONN offer an alternative and bright future for online social networking, there are some embedded complexities that make ONN effectiveness difficult to measure. The first issue that ONN present is their infrastructural complexity. ONN are complex platforms that appear in various forms and can aim for a wide range of objectives. ONN are not uniform; They exist across different companies and across the globe. Vogel (2021) documents a complete taxonomy of ONN in his 2021 Dissertation, “Designing Openness-Infusing Socio Technical Artifacts”. Figure 2 display’s Vogel’s taxonomy for ONN which helps illuminate the complex design choices that go into developing these platforms.



	Dimensions	Characteristics				
Operating Model	Availability	Global	Multi-country	Single-country	Selected cities	Selected neighborhoods
	Ownership	Private Company			Public Organization	
	Monetization	Advertising	Advertising subscriptions		Advertising paid listings	No monetization/nonprofit
Neighborhood	Neighborhood Formation	Platform-initiated			Neighborhood-initiated	
	Neighborhood delimitation	Municipal boundaries	Arbitrarily neighbordefined		Arbitrarily platform-defined	Radius-based
	Local facilitation	Key user concept		Neighborhood management service		None
Trust & Identity	Identity verification	Self-service		Self-service + in-person		None
	Invitation mechanism	Online		Online + offline		None
	Real-name policy	Enforced		Encouraged		None
User & Content	Extra-platform visibility	Fully platform-exclusive			Optionally semi-public	
	Intra-platform audiences	Own + bordering neighborhoods			Own neighborhood only	
	User-to-user relationships	Available			Not available	
	Sub-communities	Groups		Groups + building - level communities		None
	Channels	Website		Mobile App		Website + mobile app

Figure 2. Vogel's ONN taxonomy

Vogel catalogues 14 total dimensions, across four categories, that are considered when designing an ONN. Each dimension is defined by at least two characteristics if not several more. Each ONN must have its own combination of dimensions with its specified characteristics. Some examples of the dimension seen in Vogel's taxonomy are neighborhood delimitation, neighborhood formation, identity verification, and user-to-user relationships. The total number of distinct combinations help demonstrate how the ONN domain is expansive and in construction. The ONN ecosystem is an enormous system of varied products, each trying to find their niche.

Apart from infrastructural complexity, another issue present that inhibits an objective measure of ONN effectiveness is that there is an absence of standardization and focused research on ONN; even platforms like Nextdoor, which are conscious about the issues of social connection facing the United States, have yet to release reports on their platform's effectiveness in improving social connection within neighborhoods.

ONN should be optimized to ensure that the platforms improve social connection in neighborhoods. But before they can be optimized, effectiveness must first be assessed. Unfortunately, assessments are not being made neither from the platforms themselves nor from the public. And it is even unclear who should make the assessments. However, it is ironic that platforms like Nextdoor can claim to improve social connection in neighborhoods without providing metrics and public confirmations to verify their claims; their confirmation, it seems, is simply online user engagement which is not an intervention strategy to improve social connection or reduce loneliness according to public health literature (Mast et al. 2011).

Nevertheless, ONN have made the significant contribution of opening a discussion on how GIS can be used to improve social connection. The ONN ecosystem is helping the public reimagine what social networks are, where networking can take place, and how software architects can design information systems that increase community development. They have also brought back the idea that the immediate community surrounding individual is a key part of one's livelihood. They help the public acknowledge operations, events, and actions at the micro-level.

ONN also help visualize how web software infrastructure can embed GIS to create an application service that matches online users to their actual geographic dimension, creating an online neighborhood mapped to the actual physical neighborhood where users reside; This technique is novel and has the potential to improve community information networks.

Lastly, ONN usage testifies to a broad community of millions of users interested in creating communities within a walking distance. Although the effectiveness of ONN has not truly been measured, the existence of ONN and their global set of users proves that there is a public demand to experience social connection at the most immediate spatial scale, the neighborhood.

2.2 Scholarly Work Optimizing Routing Algorithms

Routing algorithms are a set of instructions that determine an optimal path within a graph-based model such as a street network (Yildirim 2023). They are derived from path planning in graph theory, which examines various methods for determining a path within a de-contextualized graph. Graphs are non-linear data structures defined as sets of edges and vertices with some special properties.

Routing algorithms are derived from graph theory theorems. For instance, Edsger Dijkstra's shortest path algorithm (1959) was first presented as a decontextualized pure mathematics theorem in his essay "A Note on Two Problems in Connexion with Graphs". In this essay, Dijkstra proves that within any simple or complex graph there exists a shortest path from point A to point B and logically verifies the strategy used to find the path.

Industries have adopted Dijkstra's algorithm within their systems. For instance, Google Maps uses Dijkstra's shortest path algorithm to help users find easy, navigable, and efficient paths to their destinations (Singh 2023). Dijkstra's theorem is a highly important piece of scholarly work that brought benefits to society in the form of modern day road network routing.

To better understand Dijkstra's algorithm, it is first necessary to examine what a graph is. A graph is both a data structure within computer science and a mathematical subject of study. In mathematics, they are abstract noncontextualized diagrams containing edges and vertices. Graphs hold various characteristics. For instance, when a graph is complete it means that each vertex is connected by an edge to every other vertex. Graphs can hold weighted edges which are edges assigned a number to represent a cost, distance or other metric that can be used to compare edges. The graph edges may be directed or undirected. Figure 3 provides an image of a graph along with some relevant graph properties that help introduce the concept to learners.

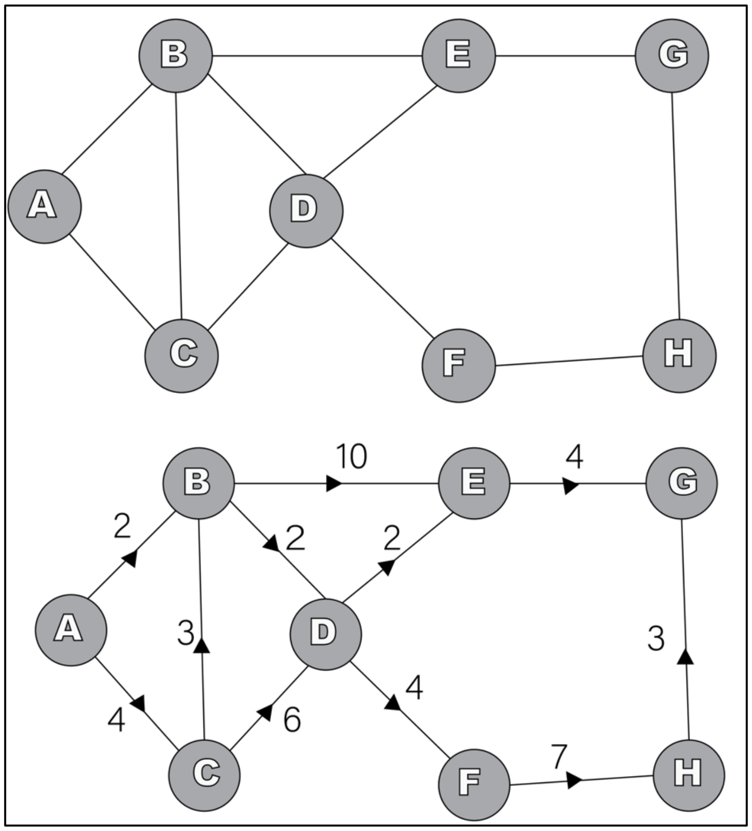


Figure 3. Graph with and without properties

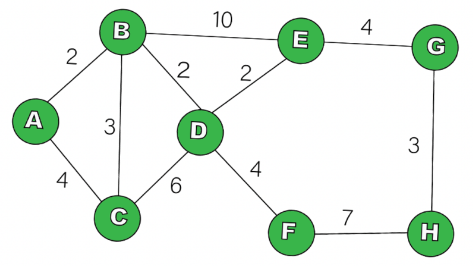
The figure above shows two graphs. On the top, there is an undirected and unweighted graph. On the bottom, there is a directed and weighted graph. Directionality and weight are graph characteristics with clear correspondences in real world networks. For example, street networks may have streets with weights, a number that can denote a type of measurement which researchers can use to decide between different paths. For instance, the weight, of any street within a street network, may be the length of the street in meters or a number denoting the level of traffic present in that street. Someone deciding between two paths will choose the street with the lowest weight if the only objective is to reach the destination as soon as possible.

A path within a street network is an exact combination of connected streets from point A to arbitrary point Z. In computer science, distance and traffic levels can be used as weights to decide on an optimal path; typically, the path with a lower weight total will be more favorable. Computers are especially important for optimization because they can quickly examine thousands of different paths with different weight totals.

In the street routing literature, graph properties are impactful in determining the most optimal path because some street network characteristics may impede travel altogether. Directionality is a property that may restrict some streets altogether from being considered when calculating an optimal path. Streets may be directed edges or one-way streets, which allow for travel in only one direction and completely restrict travel in the other direction. This is especially true for vehicle subjects. Again, computer programs are useful because they can analyze a street network and consider the restrictions in all streets, whereas a human would not be capable of considering restrictions in such an automatic fashion.

In short, graph characteristics may be important when identifying optimal paths in any type of network according to their correspondences to real world laws, phenomenon, and methods of travel. When computing an algorithm, every factor that may impact the way an individual decides on the best optimal path must be considered. Take for instance, the different factors that apply to a moving vehicle vs a walking subject. The walking subject is not restrained by traffic flow in the same way that a vehicle is. But a computer algorithm must calculate every factor if it is to be useful for optimizing operations.

The most relevant routing algorithm in this study is Dijsktra's algorithm. Dijsktra's algorithm is used in the routing algorithm developed for GLRSC-System-1. Figure 4 helps address how Dijsktra's algorithm works. Figure 4 takes Dijkstra's written algorithm (1959), published in his seminal work, and converts it into a diagrammatic form that helps visualize the processes involved.



1.

Set A	Set B	Set C	Set I	Set II	Set III
		A, B, C, D, E, F, G, H			AB, AC AD, AE AF, AG AH

2.

Set A	Set B	Set C	Set I	Set II	Set III
A	B, C	D, E, F, G, H		AB (2) AC (4)	AD, AE AF, AG AH

3.

Set A	Set B	Set C	Set I	Set II	Set III
A, B, C	D, E	F, G, H	AB (2) AC (4)	AD (4) AE(12)	AF, AG AH

4.

Set A	Set B	Set C	Set I	Set II	Set III
A, B, C, D	E, F	G, H	AB (2), AC (4), AD (4) via B, AE (12) via B	AE (6) via D AF (8) via D	AG AH

5.

Set A	Set B	Set C	Set I	Set II	Set III
A, B, C, D, E, F	G, H		AB (2), AC (4), AD (4) via B, AE (6) via D, AF (8) via D	AG (10) via E AH (15) via F	

6.

Set A	Set B	Set C	Set I	Set II	Set III
A, B, C, D, E, F, G, H			AB (2), AC (4), AD (4) via B, AE (6) via D, AF (8) via D, AG (10) via E, AH (15) via F		

Figure 4. A diagram of Dijkstra's algorithm

Dijkstra's algorithm finds the shortest path between two distinct nodes within a graph-based model. In the diagram above, there is a graph-based model with various nodes. The diagram examines how to find the shortest path from Node A to Node G. While finding the shortest path from two distinct nodes in the graph, Dijkstra's algorithm also concurrently finds the shortest paths between Node A to every other node along the way. The algorithm then decides the shortest path from Node A to destination Node G by first finding the shortest path from Node A to the node prior to node G. The chain is created from the very first node, Node A, to the node that follows A, which in Figure 4 is Node B or Node C.

First, the algorithm creates six different sets, or data structures, to store information, then repeats an execution until the path from Node A to Node G is found. During each execution, optimal paths from Node A to the nodes are found. To begin the algorithm, one must first create six sets. Set A is the set used to record which nodes have been examined. Set B stores the nodes to be examined in the next iteration. Set C stores nodes left to be examined. Set I stores all shortest paths ordered from shortest to longest. Set II stores the paths to be examined in the next iteration. Set III stores the paths not yet examined. All paths begin with Node A. Because there are seven nodes apart from Node A, there are seven shortest paths. Therefore, it is expected that there will be seven paths in Set I upon completion of the algorithm. The algorithm begins with all nodes in Set C and all paths in Set III; Iteration 1 is by default setting the algorithm for execution.

In the second iteration, Node A is examined. Node A is connected to Node B and Node C through Edges AB and AC. Therefore, Node B and Node C are placed in Set B and removed from Set C. Paths AB and AC are added to Set II and removed from Set III. Node A, as the first node which requires no optimal path from itself to itself is placed in Set A.

In the third iteration, the algorithm compares the weights of each edge connected to the current node, Node A. Paths AB and AC with their associated weights are placed in Set I in order of shortest to longest. AB is placed in Set I with a weight of 2. AC is placed in set I with a weight of 4. The next iteration is prepared by selecting a new current node. The new current node for the next iteration is the node belonging to the shortest path determined in the current iteration. Therefore, Node B is assigned as the current node for the next iteration, iteration 4, because AB has a weight of 2 compared with AC's weight of 4.

Because Node D and E are connected to Node B, the algorithm places them in Set B, and the algorithm examines paths AD and AE through Node B. Path AD through Node B has a weight of 4. Path AE through Node B has a weight of 12. Paths AD and AE through Node B are added to Set II and removed from Set III. They are placed in Set I in the table of the next iteration if and only if there does not exist a path with same origin and destination nodes with a lower weight total. For example, in iteration 5, a new Path AE through Node D has a weight of 6 which replaces the Path AE through Node B with a weight of 12 in the table of iteration 4; the algorithm is effective because it ensures that only the shortest paths make the final set. The next iteration begins and repeats the process until all shortest paths are found and stored in Set I including the path from Node A to Node G.

Popular well-known applications have capitalized on algorithms such as Dijkstra's algorithm. ArcGIS Pro utilizes routing algorithms and routing analysis via their Network Analyst tool, which helps determine optimal paths and plan projects (ArcGIS Network Analyst, n.d.). Apart from Google Maps, Google also uses network analysis through their Page Rank algorithm which decides the order in which content across the World Wide Web should be listed in searches (Yoon et al. 2011, 96). Facebook uses graph data structures in social network analyses which help determine friend recommendations (Iniguez 2022). Any type of traversal through a graph data structure implies routing in an abstract sense and therefore, these technologies are made possible partially because of scholarly work optimizing routing algorithms.

Researchers not only improve the algorithm for road network routing purposes but also for other applications. Yildirim (2023) investigates Bellmanford's shortest path algorithm and

Dijkstra’s shortest path algorithm in cryptographic tools. Scholarly work optimizing routing algorithms are important because they allow the continuous application and adoption of routing algorithms into practical use cases. Improvements, as such, are made across broad communities of scholars, which help bring benefits to society.

Domain improvements and advancements are critical for algorithm usage. For example, Fan (2010) improves Dijkstra’s algorithm within the field of road route planning by introducing more optimal data storage structures and techniques to minimize the search space of large complex road networks. Fan’s improvements are important because when a computer analyzes real world street networks, there can be up to hundreds if not thousands of streets, or edges, to choose from. As the distance from the origin point increases, the number of streets to examine also increases which results in longer execution time. Imagine Figure 4, used to demonstrate the processes in Dijkstra’s algorithm, but instead of the seven nodes that were analyzed, there were 20,000 nodes; it is typical that a computer processes the shortest path with a street network of that size, but a human could not. A computer can analyze the road network so quickly because it contains optimized algorithms that account for road network complexity through improved data storage structures and instructions on how to minimize search space. Furthermore, vehicle navigation systems or other routing software must also consider traffic flow on each street, directionality, length, and other factors including weather and/or road barriers.

Moreover, in present day, Dijsktra’s algorithm is used through computer programs and interfaces to serve very large audiences. Companies and other routing services take full advantage of path planning by designing code scripts using Python, Java, or another programming language, that can be reused by multiple users within a user interface. However, the downside to this, is that there must be sufficient computing capacity to handle the request from thousands, if not millions of users. The systems must also account for multiple factors such as road lengths, barriers, hazards, traffic, and weather. To meet the demand of scalability and high performance, there is a requirement to produce the most optimized algorithm possible. The better the algorithm, the more requests can be handled. Therefore, work that optimize routing algorithms are critical components of production level success.

Scholarly work optimizing routing algorithms provides, discovers, improves, and contextualizes various optimization options and therefore are an important area of study not only for this paper but for all work where routing services are required. For example, Zhang's work (2011) uses a GIS-based framework to examine and optimize a bus route network in Wuhan, China; the research methodology and findings may be used by city transportation departments investigating optimal bus routes in their city. Likewise, Yongmei's work uses an Ant Colony Routing Algorithm (2015) to find more efficient paths for peach product distribution; distributors may find Yongmei's research useful for optimizing their operations.

As demonstrated by the two examples, scholarly work optimizing routing algorithms are often contextual which results in various methodologies. The research presented in this paper also has its own context; it introduces an approach for using routing algorithms and digital multi-user environments to determine optimal midpoints between users. The literature for finding optimal midpoints and routes to the midpoints is discussed below.

2.2.1 Scholarly Work Optimizing Midpoint Routes

Finding optimal midpoints along routes is a small subset of the routing literature. Routing literature examining optimal algorithms for finding meeting midpoints between two or more origin points cite Weber's industrial location theory as an intellectual predecessor (Faron 2002; Wang et al. 2018; Yan et al. 2011). Weber had a theory that the optimal site to place a company was at the center of a location triangle consisting of two locations of raw materials and one market location. However, contemporary practices have largely divorced from Weber's theory and methods.

Nowadays, network distance and routing algorithms are used to find optimal midpoints (Wang et al. 2018; Yan et al. 2011). Hakami (1965) determined that the vertices of a network can be taken as a set of candidates for the optimal midpoint. The multi-user digital routing strategy incorporated in GLRSC-System-1 takes from Hakami's knowledge by also taking the vertices of a street network as potential candidates for a meeting midpoint. However, different contexts or variations may emerge that do not use street intersections as ideal meeting places and/or midpoint. Some variations may be more favorable for community engagement. For instance, nearby green spaces or friendly commercial locations can also be used as optimal meeting locations or close enough midpoint; these are topics for further investigation that have not been addressed in the routing literature. In other words, research that uses the road network vertices as candidates for finding an optimal midpoint have not included a favorability ranking.

2.2.2 Wang et al. Method for Finding Meeting Midpoints Along Road Networks

Wang et al. (2018) catalogue trip planning, carpooling services, collaborative interaction, and logistics management as situations where finding a meeting midpoint is helpful. Scholarly work optimizing routing algorithms, for finding meeting midpoints along road networks, has made significant key contributions to the work examined in this thesis.
For instance, the key contributions to GLRSC-System-1 are algorithm design considerations, and techniques for improving algorithm efficiency such as improving data storage structures, restricting the search space, and improving time complexity.

Wang et al. provide a baseline algorithm for finding the midpoint using SuperMap and ArcGIS software. The baseline algorithm provides a great reference for developing other context-based algorithms. Figure 5 renders a slightly modified version of Wang et al. algorithm structure.

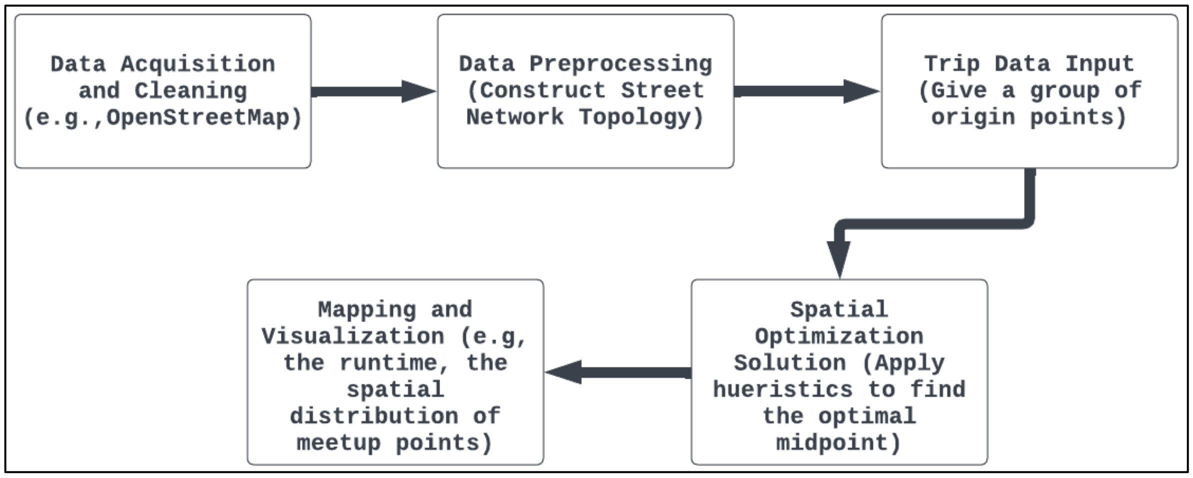

Data Acquisition and Cleaning (e.g.,OpenStreetMap)
Data Preprocessing (Construct Street Network Topology)
Trip Data Input (Give a group of origin points)
Spatial Optimization Solution (Apply hueristics to find the optimal midpoint)
Mapping and Visualization (e.g, the runtime, the spatial distribution of meetup points)

Figure 5. Wang et al. modified algorithm

The first step in the algorithm is to acquire a map with embedded line and point data from an external source such as OpenStreetMap. Secondly, process the map into a network topology using network analysis tools. Third, input the origins, and/or destination points of study. The fourth step is to apply geoprocessing heuristics or the main substance of instructions to find optimal routes and midpoints. Lastly, the fifth step is to visualize the optimal routes from the origin points to the midpoint. This algorithm is mostly copied in this thesis with some key differences.

Figure 5 excludes a step included in Wang et al. algorithm which integrates context input weights, such as traffic information, weather, road barriers, etc. For this research input weights are negligible because the primary user is a walking human subject within a neighborhood context not an automobile traversing a complex urban area. Context input weights are typically relevant when creating routes for vehicle navigation systems but are negligible for this thesis project.

2.2.3 Fan's Major Contributions

The Fan's major contribution (2010) is the knowledge on how to improve algorithms for finding midpoints. Fan lists three useful strategies for improving routing midpoint algorithms: improve data structure storage, reduce the search space, and reduce time complexity. The three strategies should be kept in consideration when developing a system that integrates a routing algorithm because small improvements in efficiency can result in more robust and scalable web services. But for the purposes of this research, the exact methods of reducing data structure storage and improving time complexity are ignored. They are advanced methods examined in the scholarly work but are beyond the scope of this study. Reducing search space is the only strategy that is addressed and used in this study.

The reduction of a search space is an action considered and implemented in GLRSC-System-1. A search space in graph theory is the total network of nodes and edges that is considered when computing an optimized route. For instance, in the Google Maps routing service, when a user asks for a route from point A to point B, the software constructs a network of nodes and edges. The software must also restrict the size of the network to be efficient. Otherwise, the software processes could endlessly search through the ever-expanding street network. Search spaces are used in routing algorithms to ensure that only the closest streets are examined when trying to find the most optimal route (Fan 2010).

2.3 Thesis Level Web Routing Applications

Web routing applications developed in graduate level theses are relevant to this thesis and have helped inform the thesis methodology. Reviewing architectural and thematic similarities and differences of related software can be informative in the development process. The projects in this group provide several tool-choice validations including the decision to use React.js as a front-end framework and Flask as a back-end framework.

This section is primarily concerned with the work of two different development research projects. One of the projects develops a single-user web routing application to find optimal meeting places (Petit 2020). The second project develops a single-user web routing application to route optimal bike paths (Hruby 2021). Both applications are small scale applications meaning that they have a low scale technical infrastructure; they are not industry or enterprise-level products.

While the applications may not be industry-level, they serve as great examples of how to plan a web development project that integrates routing algorithms along with cartographic interfaces. Petit's web application for finding optimal meeting places also provides an analysis on the efficiency of different routing algorithms. Meanwhile, Hruby's project showcases how technical components such as a Flask back-end and a React front-end come together in a web routing web application.

Petit provides sound reasoning for a routing algorithm that finds an optimal meeting place between two or more subjects. The two main strategies that Petit (2020) analyzes are the geographic mean technique and the path traversal technique. The geographic mean is the simplest approach and takes the sum of the coordinates and divides it by the number of total coordinates. The next step in the algorithm is to find the nearest street intersection from the geographic mean; the coordinates of the street intersection become the optimal meeting place. Finding the street intersection is important because it ensures uniformity. For instance, a geographic mean can be positioned on any sort of structure such as on top of a building, within someone's yard, in a lake, etc. Making all use cases use a street intersection as a meeting point prevents potential problems in location. Petit uses OSMnx, a python programming package, to identify the nearest street intersection from the geographic mean (Boeing 2017). All nodes in a graph street network retrieved from OSMnx correspond with real world street intersections (Boeing 2017).

Petit also provides an analysis on the geographic path traversal technique which helped verify the decision to exclude it from the thesis and system implementation. The geographic path traversal technique also finds the coordinates of the street intersection via the geographic mean. It then finds a shortest path for each origin point to the street intersection using Dijkstra's algorithm. Using these paths, the algorithm checks each consecutive node from each path's origin node. If by chance, the next node of one path is the next node of another path, it recomputes the geographic mean node using the new coordinates from each node, which produces a new street intersection as the meeting place. The algorithm then re-computes Dijkstra's algorithm again for each node. This process is continued till there are no nodes left in all paths. The results are a meeting place that accounts for shared paths. It is computationally heavier and more complex than the geographic mean technique, but the results render a meeting place that minimizes distance for the parties involved.

While the geographic path traversal technique accounts for shared paths, this thesis research uses the geographic mean technique. Petit's provides a computational analysis of both techniques; the results of his analysis influenced my decision to use the geographic mean within

GLRSC-System-1. According to Petit, the geographic mean technique is simpler and has lower computational costs than the path traversal technique in most cases (Petit 2020).

Petit also runs an analysis on NetworkX's and OSMnx's shortest path function, which uses the Dijsktra's algorithm previously mentioned, and finds that Dijsktra's shortest path algorithm is much more efficient than other shortest path algorithms. NetworkX is also a python programming library that helps create street network maps and runs useful function's such as finding a shortest-path or a nearest node (Hagber et al. 2008). Petit makes a significant contribution to this work by positively validating the decision to use Dijkstra's algorithm when finding shortest paths.

Furthermore, both Hruby and Petit provide information on how to further improve the efficiency of routing algorithms by incorporating contraction hierarchies (Hruby 2021). But contraction hierarchies are not included in this research to limit the scope of the project. Nevertheless, contraction hierarchies can be considered for future research to optimize the routing algorithms developed for GLRSC-System-1.

Both projects also present their own back-end engines using either Flask or Django as back-end python-based frameworks. Petit uses Django while Hruby uses Flask. GLRSC-System-1 uses Flask, but both are equally viable options for developing small- or large-scale web routing applications. Petit and Hruby's projects testify to the feasibility of using python as a language for programming a back-end API of a routing web application.

Both Hruby and Petit import third party python-based geospatial libraries to help process spatial data from OpenStreetMap. Petit uses NetworkX and OSMnx while Hruby chooses Libosmium. Lastly, Hruby uses PostgreSQL and PostGIS for back-end storage, which provides some validation for choosing the technology in this thesis.

Furthermore, both projects have digital user interfaces backed by JavaScript, HTML, and CSS. Petit sticks to standard JavaScript web development without employing a framework. Hruby utilizes React.js as their front-end framework. Hruby's decision to use React as a front-end framework in a web routing application adds validation to my decision for choosing React for GLRSC-System-1.

Similarly, Petit provides positive validation for the use cases provided in this work. Like GLRSC-System-1, Petit's web application provides a service for computing an optimal midpoint given the coordinates of two or more human subjects. However, Petit does not employ a spatial scale nor encourages connections in local communities. Petit's application is vehicle-centric and is a single-user environment rather than a multi-user environment. Petit's work also has an expansive spatial scale without a limit. Therefore, a single user could use coordinate points 10 or more kilometers away in distance to compute an optimal midpoint. In contrast, this thesis research restricts the spatial scale to a 2km radius, which encourages walking and engaging with a local community. Furthermore, Petit's system is a single-user environment and is not designed to provide a communication channel between two users using the system synchronously. Lastly, Petit also utilizes context input weights which indicates that Petit's work targets a user audience who own and use vehicles. In contrast, GLRSC-System-1 is designed for a walking subject to connect with peers who live within 2km from the user.

2.4 Commercial Web/Mobile Routing Products

Routing applications have emerged as powerful platforms that are changing multiple facets of society including transportation, consumerism, and work. Applications such as Uber, Instacart, and DoorDash are not hypothetical routing services, but rather tangible large-scale routing products with real-world impacts and consequences. They are also a testament to the adaptiveness of routing algorithms and their utility in shaping a new and efficient world. Beyond the criticism targeted towards these platforms, it is a fact that they have changed how some people travel, consume and work. Advances in web routing applications may continue to catalyze social changes. Commercial routing products are included in this section because they are exemplary work that can offer practical insights on how to produce and engineer a routing product that scales. Learning from exemplary commercial routing products can help produce robust and efficient routing web applications.

One factor that should not be ignored for this group is that they are all highly developed, highly complex, as well as highly resourced ecosystems. Each of the applications mentioned in this section rely on a complex, meticulously engineered bundle of web infrastructure which makes their systems so robust. For example, Uber has publicized their technological stack. Some of the key tools that they incorporate in their products are hybrid cloud models, globally distributed data centers, caching systems, logging systems, app provisioning, routing and service discovery, customized front-end engines, and in-house visualization libraries (Lozinski 2016). This list of course only includes some of the components listed on their publication and just stratches the surface of the product ecosystem. In a similar manner, Instacart lists connected data pipelines, third party data providers, machine learning models, real-time data streams, consumer tracking services, payment systems, scalable databases, optimized routing algorithms with traffic context metrics, and rule-based formulas (Rao 2020). For the purposes of this thesis, GLRSC-System-1 does not attempt to mimic these infrastructural systems; that is beyond the scope of the study. However, applications like Uber and Instacart are models of how impactful routing services can be beyond navigation. GLRSC-System-1 attempts to highlight this truth. This research provides a very small-scale infrastructure that serves as a resource for developers creating their own routing services. This research does not attempt to reach application deployment nor production level robustness.

The main contribution that commercial level routing products have made to this thesis research is highlighting the correlation between the availability of a routing multi-user environment and social impact. Commercial web/mobile routing products all incorporate multi-user environments to achieve success. Multi-user environments ensure that a single user can create a profile and demand a service from another, perhaps unknown, user. A multi-user environment provides a communication channel for nearby users to facilitate various types of exchanges that become real-world actions and services such as those seen on the Uber platform, which provides an instant carpooling service, and those seen on the Instacart platform, which provides an instant grocery delivery service. In short, multi-user environments secure the facilitation of human-to-human contact, interaction, and exchange. The construction of a multi-user environment is a major priority for GLRSC-System-1 for this reason.

Both Uber and Instacart have publicly released sources cataloguing the various components that make up their platform's technical infrastructure. For individuals and teams that hope to build large scalable systems, these resources serve as invaluable information (Lozinski 2016; Rao 2020, 36). Individuals or teams who are building routing applications can reference Uber and Instacart documents to identify the different infrastructural components that may be needed when scaling a product. GLRSC system research and development may benefit from these resources in future work.

2.5 Summary

In this chapter, ONN were introduced as a complex understudied set of online GIS platforms with the aim of helping users engage with their local neighborhood communities.

Compared with the other related work groups, ONN differ because ONN do not employ routing algorithms. Conversely, they are comparable to GLRSC-System-1 because ONN provide users with a method to network with nearby peers in a spatially defined neighborhood using geographic information tools.

Scholarly work optimizing routing algorithms was defined as an important historical undertaking that is responsible for much of the advancement in modern digital routing services. Dijkstra's algorithm is an example of an optimized shortest path routing algorithm that is used in this work. Furthermore, not only are optimal routing algorithms relevant to the work conducted in this project but critical and used in all routing services. Scholarly work optimizing routing algorithm help discover new applications by finding solutions to optimization challenges.

Web routing applications found in graduate level thesis typically comprise of small-scale independent efforts that document a development process. These projects integrate specific routing algorithms and serve as a guide to develop routing systems. This group is ideal for technical beginners who need an example of how to proceed with development. They introduce various helpful frameworks and libraries that independent researchers can use when developing relatively simple or small-scale web routing applications.

Commercial web/mobile routing products are large scale industry level applications with complex technical architecture and are highly resourced. They are exemplary applications that successfully integrate routing algorithms, efficient cartographic designs, and robust technical infrastructure. The high level social, cultural, and economic impacts of their work is a testament to their success and the impact that web routing products, beyond navigation services, have on society. They are helpful as exemplary pieces of work from where to garner sound considerations, practices, and aspirations.

Chapter 3 System Requirements and Planning

For GLRSC-System-1, ten necessary planning tasks were executed: acquisition of background knowledge, development of the user scenario, construction of the wireflow, design of the user interface, development of the routing algorithm, design of the software architecture, selection of the supporting software, design of the spatial database, and plan the system dependencies.

3.1 Acquisition of Background Knowledge

Software development and geographic information systems/science (GIS) are two related yet separate fields whose practice/theory were both used to build GLRSC-System-1. Software engineering guidebooks and training material describe the best practices and methods for building data-related computer systems using an arrange of tools and frameworks. GIS deals with the storage, analysis, management, and presentation of geographic data (ESRI, n.d.). Knowledge from both fields was helpful to conceptualize and actualize GLRSC-System-1.

This thesis project builds GLRSC-System-1 as a simple CRUD API. In the field of software development, create, read, update, and delete (CRUD) functions are emphasized as foundational computer functions that should be relatively easy to implement in a back-end database system. Software engineering also introduces the concept of an application programming interface (API), which is defined as a connected system of computer programs or systems. For example, a full-stack application is an API consisting of a database, a back-end framework, a front-end framework, a server, a website, and potentially many more systems and sub-systems. GRLSC-System-1 is an API, albeit a small and closed one. One must review software engineering books that discuss CRUD and API. The concepts can be investigated via the internet, university curriculums, and/or private company programs – commonly known as coding bootcamps.

GLRSC-System-1 draws most fully from GIS related principles such as cartography, spatial data accuracy, spatial data models, spatial topology, and projected coordinate systems. For example, GLRSC-System-1 presents a geographic route on a cartographic display which is designed with cartographic best practices in mind. GRLSC-System-1 ensures that it stores an accurate measurement of the user's location by using Wi-Fi positioning. GLRSC-System-1 uses computer points and polygons to represent real life objects such as people and their neighborhoods, respectively. Lastly, GLRSC-System-1 addresses the topic of spatial topology, by acknowledging and applying people's presence within spatial neighborhoods; The system uses spatial topology knowledge to decide which users can meet and where they should meet. One should reference reliable sources to gain correct information regarding GIS and its topics.

Developing GLRSC-System-1 has been an effort in both GIS engineering and software engineering. GIS engineering requires a comprehension of GIScience. And the ability to practice both disciplines was imperative to build GLRSC-System-1; this thesis requires an integrative GIS engineering – software engineering approach.

3.2 Development of the User Scenario

GLRSC-System-1 has the objective of facilitating community engagement by routing nearby users to meet at a nearby midpoint, within 2km, along the road network. In other words, the general use case is face-to-face social networking within a walking distance. GLRSC-System-1 is successful when it has facilitated face to face contact with a nearby user; this is designed to be part of the user experience. Figure 6 displays a general use case between two nearby hypothetical users, one as the requestor and the other as the responder.

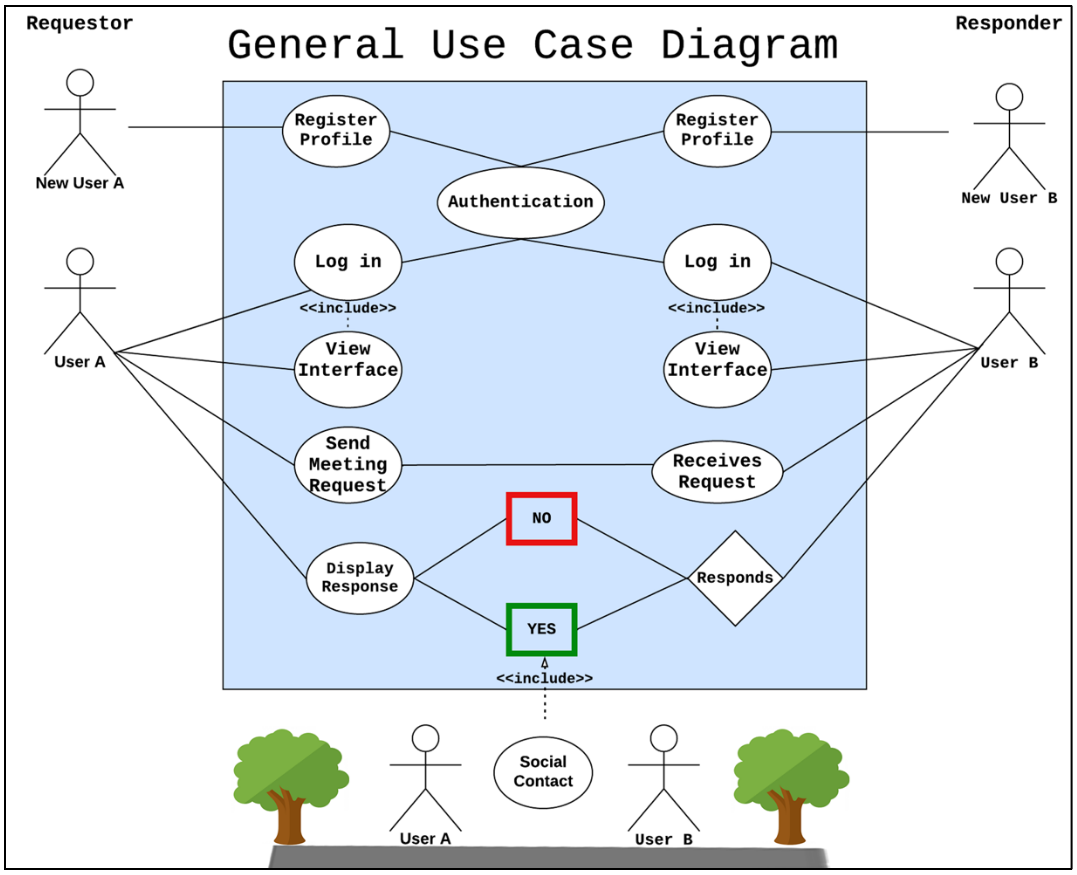


Figure 6. General use case diagram

From top to bottom, Figure 6 describes a system whose goal is to combine a computer environment with the natural environment under one abstract operation: user-to-user social contact. The exact reason for social contact is not explained in Figure 6. But the exclusion of a specific reason for social contact gives more focus to the desired outcome, social contact.

The very top of the diagram demonstrates that when a user interacts with GLRSC-System-1 the first action must be profile or account registration. The second action, from the user's perspective, is to log in, which is followed by viewing the system's interface.

GLRSC-System-1 has two types of user designations: responder and requestor. The user type is important not only in understanding how the system works but also in designing and engineering the system. Within the interface, one can act as a requestor or a responder. The requestor sends a request to meet while the responder responds affirmatively or negatively. If the responder declines, the responder denies meeting in the physical environment and GLRSC-System-1 has failed to facilitate contact for the requestor. However, if the responder accepts the request, GLRSC-System-1 guides the users to meet at a nearby point along the road network, ensuring a successful use case.

User-to-user social contact is an abstract operation because it could result in many outcomes. However, GLRSC-System-1 has a way to make the outcome more specific and predictable. GLRSC-System-1 features a meeting request message which serves as a method to specify the general use case and/or provide a theme to the meeting. In the meeting request message, the requestor can specify a reason to meet using a 75-character limit input. Every unique meeting request message creates a specific use case that extends the general use case. The feature also benefits the responder because it provides them with an additional reason to meet. Some specific use cases are shown in Figure 7; They highlight the potential specific use cases achieved through a custom request message. The specific use cases reduce loneliness and enhance social connection.

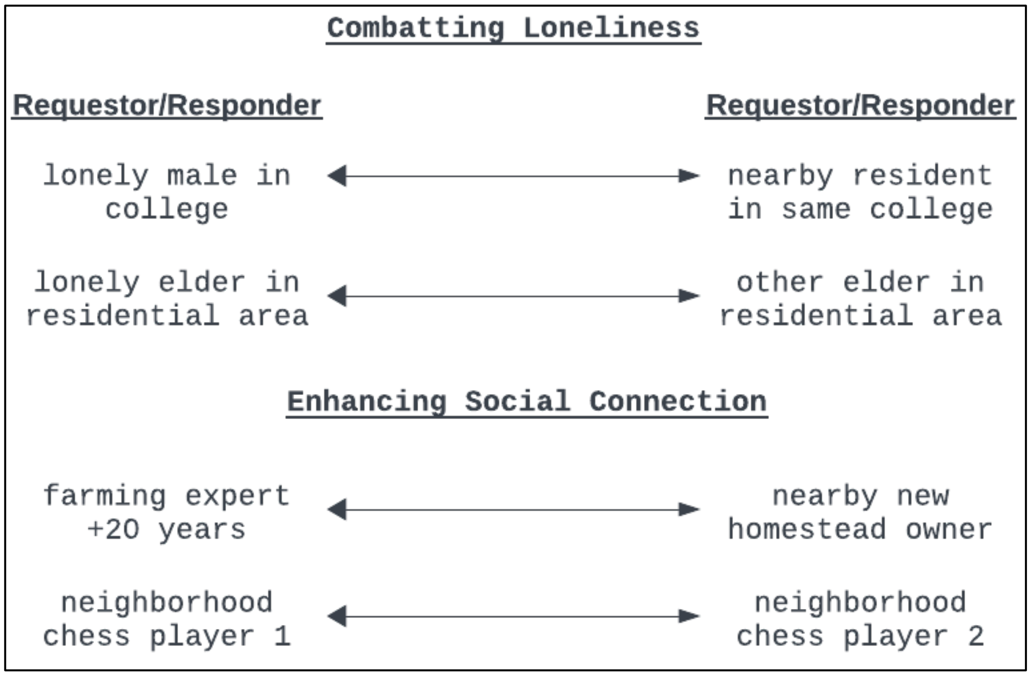

Combatting Loneliness
Requestor/Responder
Requestor/Responder
lonely male in college
nearby resident in same college
lonely elder in residential area
other elder in residential area
Enhancing Social Connection
farming expert +20 years
nearby new homestead owner
neighborhood chess player 1
neighborhood chess player 2

Figure 7. Specific use case examples

The specific use cases shown in Figure 7 help highlight the specific applications for GLRSC-System-1. Applications can range from an individual searching for a friend to someone who wants to play chess. By allowing users to specify a theme for the meeting, GLRSC-System-1 expands the potential use cases and/or gives the users the ability to define the use cases for themselves.

3.3 Construction of the Wireflow

A wireflow is a set of two or connected wireframes that depicts both the organizational structure of the web page but also how the pages lead to one another (Angeles 2024). Building the wireflow is a necessary action to determine the organizational layout not only of the front-end user interface but also of the entire codebase. Highly organized and modular code is necessary to build software that can scale and remain maintainable (Meruliya 2022). Building the wireframes within the wireflow helps adhere to the principle of modularity because it provides a developer with an opportunity to divide the user interface into organized components.

GLRSC-System-1 uses a layout of three columns. The system renders two columns, each containing three boxes aligned vertically. The last column contains two boxes aligned vertically. Each box in the wireframe corresponds with a front-end functional component assigned to a system function. For instance, one box may be assigned with the functions to generate an account while another may print messages to the user. The separation of concerns on the front-end UI, reflected on an original wireframe, lays the foundation for how the code is organized not only on the front-end but also on the back-end.

Building the wireflow presented in Figure 8 was an immensely helpful task that determined software design choices later in the development process. The wireflow makes it possible to approach development using a piece-by-piece, component-by-component, mentality.

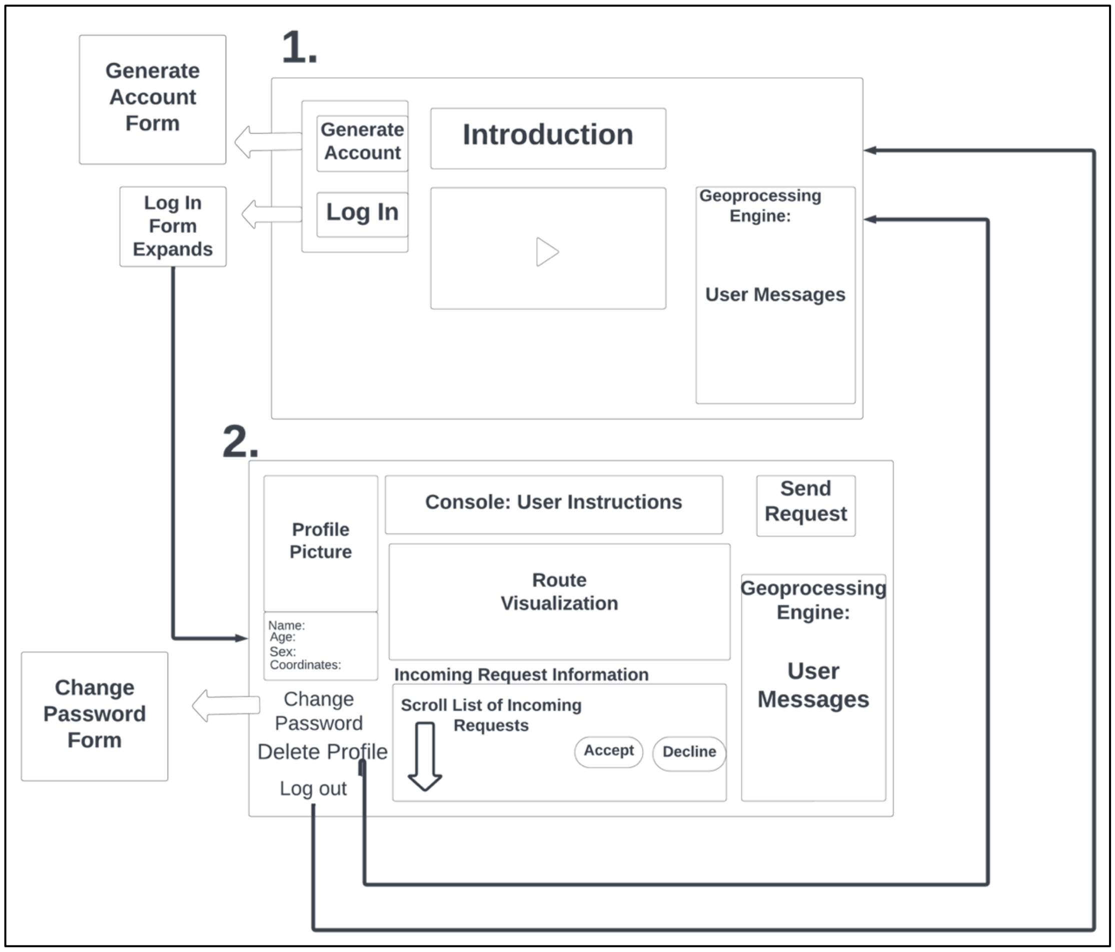


Figure 8. GLRSC-System-1 wireflow

Figure 8 shows the application divided into two separate views. The decision to incorporate only two views was encouraged by concepts of simplicity and minimalism which not only make the application easier to develop but also make the application easier to navigate.

Both views are composed of components. In Figure 8, the components are identified by black border boxes. Each component is a separate functionality that renders on top of a single web page. Therefore, GLRSC-System-1 has two views with independent components that render conditionally.

For instance, the components in view one are responsible for generating accounts and enabling log ins. Two of these components give textual and audio introduction to the system, respectively. There is also a component that processes and displays system messages to users; this component is called the geoprocessing engine and promotes data or algorithmic transparency.

The wireflow contains a logical flow that demonstrates how a user transitions between view one and view two. In Figure 8, there is an arrow connecting the log in form from view one to view two. The arrow demonstrates that to access view two, the user must first log in.

The second view shares the same black border boxes as view one, but separate functional components are present. For instance, in place of view one’s generate account and log in buttons, view two shows account information.

Different functional components appear in view two because of conditional programming. Upon login, the user should have access to GLRSC-System-1’s services. The user should also have access to authentication and account services that would not make sense to include on a landing page where users who are not logged in should not have access to functions such as change password, delete profile, and log out functionalities.

View two has four additional components. The user console is like the geoprocessing engine in that it displays system generated messages to the user. However, the user console is concerned with user instructions rather than algorithmic and data transparency. View two also has a component that displays an optimal route, a component that allows users to accept or decline a request, and a component that allows users to send a meeting request with a custom, 75-character limit message.

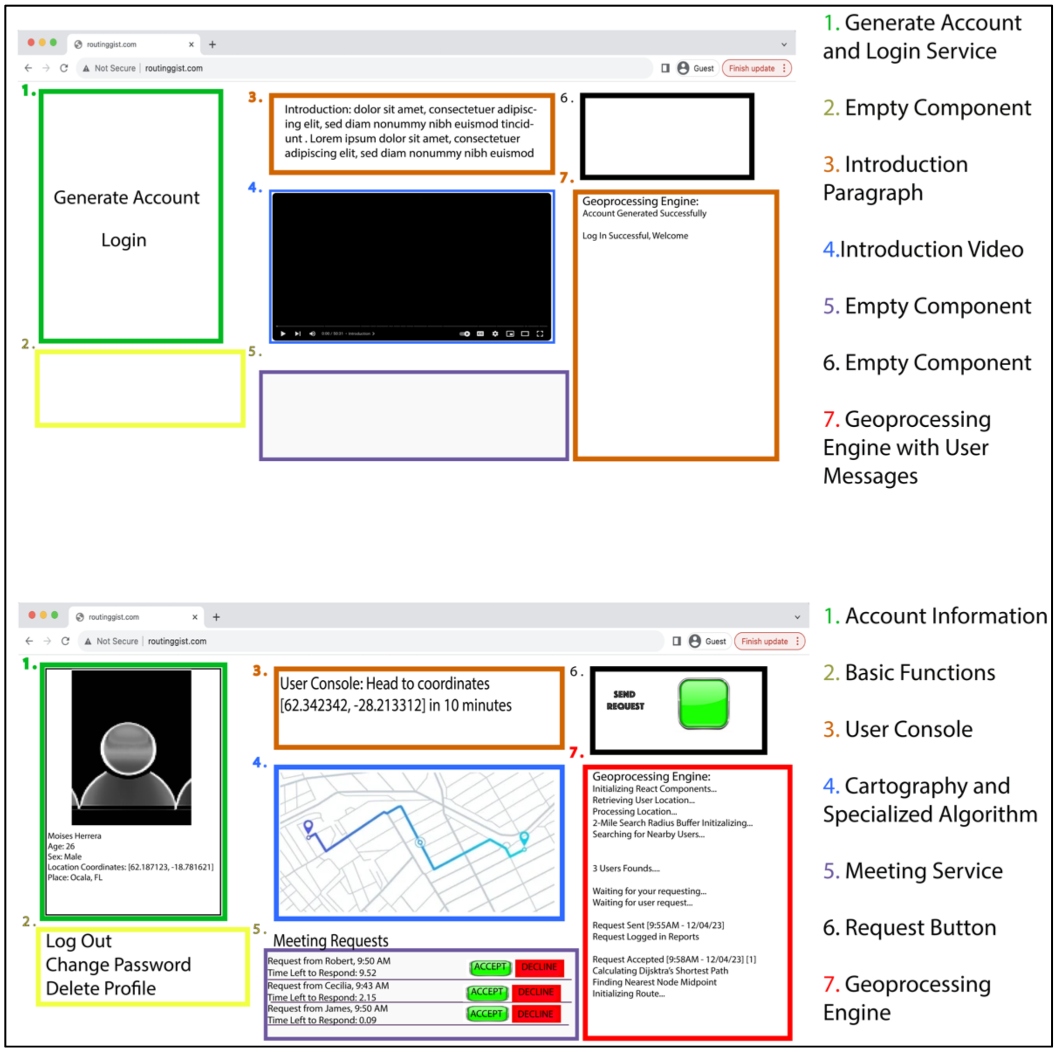
Returning to Figure 8, an arrow connects view two’s delete account and log out buttons to view one. The connected arrow indicates that the user can access view one by using either button.

In conclusion, the wireflow helps define system organization and user flow. Therefore, it was given considerable thought and attention before developing GLRSC-System-1. Designing the wireflow for GLRSC-System-1 was an iterative process. The wireflow in Figure 8 changed to reflect updated decisions and what the system should do and be. Well documented design iterations can help communicate the development process. For concision's sake, only the final iteration is presented. The diagram in this section was created using LucidCharts, a web application for creating wireframes and other prototype visual models (Lucid, n.d.).

3.4 Design of the User Interface and Experience

The objective for the user interface is to not only to provide the users with a unique routing service that allows them to facilitate a quick meeting along the road network within a walkable distance, but also to incorporate key informational elements that provide system knowledge. The audiences are the scientific community and the public because the system hopes to present a new idea and be useful for the everyday person.

Figure 9 depicts the user interface for GLRSC-System-1, followed by a description of the key elements of the interface and how they facilitate a specific user experience.



Introduction: dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod tincidunt . Lorem ipsum dolor sit amet, consectetuer adipiscing elit, sed diam nonummy nibh euismod

Geoprocessing Engine:
Account Generated Successfully

Log In Successful, Welcome

Moises Herrera
Age: 26
Sex: Male
Location Coordinates: [62.187123, -18.781621]
Place: Ocala, FL

User Console: Head to coordinates [62.342342, -28.213312] in 10 minutes

Request from Robert, 9:50 AM
Time Left to Respond: 9.52
Request from Cecilia, 9:43 AM
Time Left to Respond: 2.15
Request from James, 9:50 AM
Time Left to Respond: 0.09

Geoprocessing Engine:
Initializing React Components...
Retrieving User Location...
Processing Location...
2-Mile Search Radius Buffer Initizalizing...
Searching for Nearby Users...

3 Users Founds....

Waiting for your requesting...
Waiting for user request...

Request Sent [9:55AM - 12/04/23]
Request Logged in Reports

Request Accepted [9:58AM - 12/04/23] [1]
Calculating Dijsktra's Shortest Path
Finding Nearest Node Midpoint
Initializing Route...

Figure 9. GLRSC-System-1 interface

Figure 9 uses the organizational layout from Figure 8. In Figure 9, view one is defined by four active components: generate account and login service, introduction paragraph, introduction video, and the geoprocessing engine. View two is defined by six active components: account information, basic functions, user console, cartography and routing component, meeting service, request feature, and the geoprocessing engine. Like Figure 8, Figure 9 is divided into two views, view one where the user is logged out, and view two where the user is logged in.

Figure 9 shows the user experience characterized by five experiential elements: an authentication system, a meeting service, a routing service, user instructions and informational, and data transparency. The elements are reinforced throughout the system and not consolidated into just one component. The authentication system creates a multi-user environment where each independent user has their own private account that holds required functions such as log in, change password, log out, and delete profile; this provides the user with a sense of personal space and control over their user experience. The authentication system is most visible in component one of view one and component one and two of view two.

The meeting service is visible in component four and five of view two. The service provides users with the ability to send a custom meeting request and to receive request from others. The user is given the power to decline or accept any request. All requests provide some information from the requestor account. The information that is provided is the same that is on component one in view two, account details. The system displays requestor account details within all meeting requests to help improve user safety; each user has the right to know who they meet. The meeting service provides an engaging and exciting experience where users may feel intrigued by the ability to send and respond to requests from nearby peers for a face-to-face meeting.

The routing service is consolidated to component four of view two. Component four of view two presents the route from the user's location to a meeting midpoint once a meeting has been accepted. Component four provides a cartographic display of the route on a fixed non-adjustable neighborhood map. The cartographic display helps communicate the objective of the system and helps guide people to the meeting point location. The routing service and component creates a scientific space which bolsters spatial cognition. It provides users with geographic data and the route from the user's location to the midpoint makes the experience very personal and real.

User instructions and informational material is visible in components three and four of view one and component three of view two. User instructions and informational material are important to inform the user on GLRSC-System-1. This increases the learning experienced and gained using the system. Nevertheless, user instructions are likely required for any system practicing GLRSC. Users need to know where to meet, when to meet, and with whom they are meeting; any secure and safe system should address these concerns and communicate them with a user.

Data transparency is visible in component seven of view one and components seven, three and one of view two. Component seven of view one and view two are the same component. The geoprocessing engine reveals to the user how their data is being used. This helps gain user trust and promotes learning by describing the Flask back-end operations and algorithms. Flask, version 3.0.0, is used for this project as the back-end framework (Flask, n.d.). This development is influenced by ArcGIS and other scientific software which include some transparency on behind-the-scenes data functions. Data transparency is incorporated into the system interface to demonstrate the scientific foundations of the system.

The choice of the front-end programming language and front-end framework were validated during the development of the wireframe and user interface. React, version 18.2.0, was chosen as a front-end development framework for this thesis because it makes the development process easier by organizing front-end code into individualized components that corresponds to components on the user interface (React, n.d.). React is programmed in JavaScript, ES6, and therefore, I used JavaScript, ES6, as the front-end programming language by necessity (React, n.d.). The diagrams in this section were created using Adobe Photoshop and Illustrator 2024.

3.5 Development of the Routing Algorithm

GLRSC-System-1 relies on a specific implementation of a routing algorithm that requires the consent of two or more parties. Before programming the system, the algorithm was tested to ensure feasibility. Testing was done through Jupyter Notebooks, version 7.0.6, whereby a script was written to simulate a production level system (Jupyter 2024). The exercise of pre-writing the script in Jupyter Notebooks was helpful to increase confidence in project feasibility.

The routing algorithm for this research is a step-by-step process and pipeline used to convert the coordinates of two or more users into a useful optimal route that leads to a meeting along nearby road networks. Once the optimal route and meeting midpoint are calculated, they are exported to the front-end for visualization. Figure 10 displays the steps in the algorithm which run in the Flask back-end and are spread out across the back-end source files.

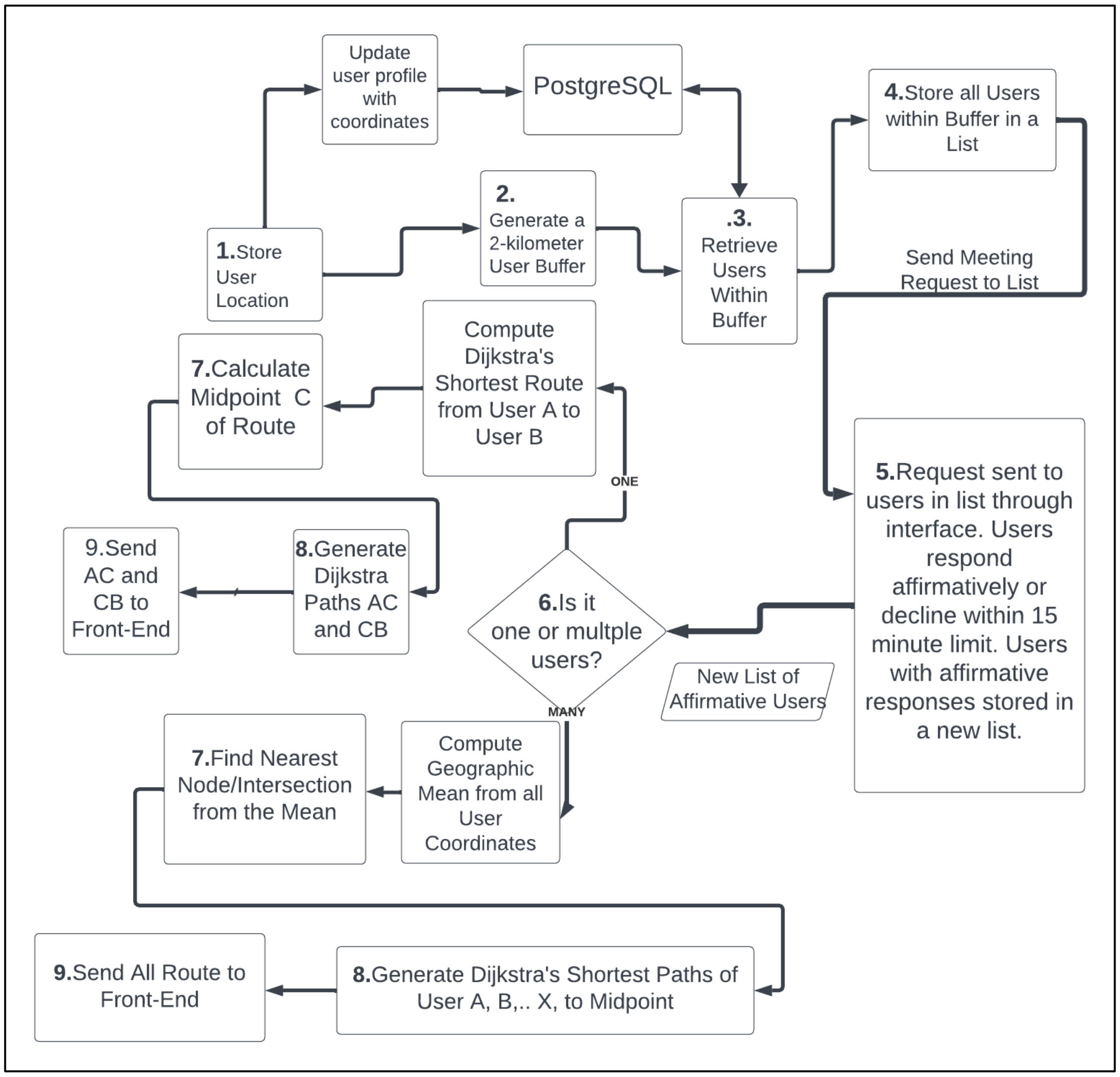

Update user profile with coordinates
PostgreSQL
4.Store all Users within Buffer in a List
2. Generate a 2-kilometer User Buffer
.3. Retrieve Users Within Buffer
1.Store User Location
Send Meeting Request to List
Compute Dijkstra's Shortest Route from User A to User B
7.Calculate Midpoint C of Route
ONE
5.Request sent to users in list through interface. Users respond affirmatively or decline within 15 minute limit. Users with affirmative responses stored in a new list.
9.Send AC and CB to Front-End
8.Generate Dijkstra Paths AC and CB
6.Is it one or multple users?
New List of Affirmative Users
MANY
7.Find Nearest Node/Intersection from the Mean
Compute Geographic Mean from all User Coordinates
9.Send All Route to Front-End
8.Generate Dijkstra's Shortest Paths of User A, B,.. X, to Midpoint

Figure 10. GLRSC-System-1 routing algorithm

The routing algorithm can be summarized in nine steps. The scenario where there is only one other user willing to meet, a one-to-one meet, is described first. The first step is to store the user location in a PostgreSQL database as a Geometry Point. The second, third, fourth steps are done efficiently using a Spatial SQL function that generates a 2km buffer, checks the users that exist within the buffer, and returns a list of the users. The fifth step is to send a request to the list of users who lie within the buffer. This request appears on the interface through a registered and logged in account. Users either accept or decline. The users who accept are placed into a new list of participants. The sixth step is to program a conditional that runs separate heuristic processes depending on whether there are one or more than one participant. In the case that there is only one participant, python code runs Dijkstra shortest path algorithm to produce a route from user A to user B. The seventh step is to find the median node of the Dijkstra route and designate it as the meeting midpoint. Dijkstra routes are calculated as a series of nodes, each of which correspond to real-world street intersections. It is easy to find the median node by determining the number of nodes, street intersections, in the route and dividing by two. The eighth step is to execute Dijkstra shortest path algorithm two times more, once for user A to the midpoint and again for user B to the midpoint. The result is two routes AC and CB. The final ninth step is to send the routes and midpoint information to correspondent user profiles via the user interface.

In the case that there is more than one participant, the GLRSC routing algorithm first calculates the geographic mean of the participants' coordinates and then find the nearest node, in the 2km buffer area of step three, to the geographic mean. Using the OSMnx, version 1.9.1, python package, one can find the nearest node from a pair of latitude and longitude coordinates and a defined street network area (Boeing 2017). The nearest node to the geographic mean is assigned as the meeting midpoint. For each user, GLRSC-System-1 routing algorithm creates routes to their locations to the midpoint location. GLRSC-System-1 routing algorithm then sends the route and midpoint information to the front-end for further processing and visualization.

The choice of database, back-end programming language, back-end framework and graph processing packages were validated during the development of the routing algorithm. PostgreSQL, version 15.7, was selected because it is open-source, free to work with, and has many years of supporting spatial extensions like PostGIS, version 3.4.2 (PostgreSQL, n.d.; PostGIS, n.d.).

Python, version 3.9.0, was selected for the back-end programming language not only because it is a highly popular and developed language, but it also has several geospatial libraries that support geospatial web development (Sharma 2023; Python, n.d). OSMnx was selected because it proved to be useful in calculating the midpoint and its resources are more than adequate for this project. Flask was chosen because it is one of two back-end frameworks that uses Python. A back-end framework that uses Python is essential for this project because the back-end must be able to use OSMnx. Therefore, Flask was selected. It was selected over Django because it has a higher level of customizability and less rules to follow.

3.6 Design of the Software Architecture

GLRSC-System-1 contains a highly customized and conceptual function that is not available in any known system. To build the web application, it was assembled piece by piece with different software building tools.

GLRSC is a niche concept that is not supported by modern GIS systems or software. To develop such an idea into a functional piece of software, it was best to have maximum control and develop the application piece by piece. It is also ideal to not be restrained by costs in the development process. Therefore, free open-source systems, software, and data were necessary to build it. There are many definitions for open-source tools. For this thesis, open source is defined as technology or data that is freely distributed and derivable (Maurya et al. 2015).

There are many high functioning and highly developed tools that work with routing algorithm. For instance, ArcGIS Pro has a Network Analyst tool and modern GIS may be able to compute midpoints including geographic means or median nodes. However, traditional scientific GIS has separate priorities in concern with network science such as analyzing traffic congestion or identifying optimal paths. Present day GIS have not integrated GLRSC. They also do not incorporate multi-user environments where users directly pass data, and consent, to one another such as accepting meeting requests. Therefore, the technology that constitutes GLRSC-System-1 is a novel application and invention but it was assembled through existing lower-level components. The basic components necessary for building a GLRSC system are a database, a back-end framework, a front-end and a server. All four components are systems themselves with existing lower-level tools, libraries, and extensions.

For the database, PostgreSQL is required because it is scalable, free to use, and comes with PostGIS, a required spatial extension that enables storing geometry features, creating spatial indexes, and conducting geography related queries. A Flask back-end framework is required because Flask is free to use, has few rules for how to build a system, and is python-based. Limited rules are ideal because less rules allow for a higher level of control and customizability. Python is required because it is a highly supported language in the geospatial community and by using it one can be confident that they have all the necessary functions required to complete the project (Sharma 2023); there are various geospatial libraries that add support to geospatial development processes such as Geoalchemy, version 0.14.3, a python package, which provides Flask application code with geospatial classes in which to create spatial objects. Geoalchemy maps the Flask spatial objects to a supporting spatial SQL table which is a required functionality for this development effort (Geoalchemy, n.d.).

Lastly, a front-end framework that supports routing visualization libraries is required. React is used as an open-source front-end framework. It was selected because it is free to use, supports Mapbox, a map visualization service, and has strong organizational rules (React, n.d.). Because of the highly organized layout of GLRSC-System-1, as seen in Figure 8, it is necessary to use a framework that has organizational rules that make it easy to design the layout. React uses component logic which means it requires developers to separate files for each component on the page and render each component separately. React's component logic is perfect for GLRSC-System-1. The servers used for GLRSC-System-1 include Github and Heroku cloud servers (Github, n.d.; Heroku, n.d.) Github and Heroku are web applications accessible online. While they must have versioning for their platforms, software hosted on their platforms are unaware of their versions and do not need to keep track of their versions. Heroku and Github were selected because they are mostly free to use and automate processes for a simplified development experience. Github was selected because it works well with Git, a version control system. Lastly, Heroku was selected because it has storage resources more than adequate for this project.

Figure 11 depicts a deployed version of GLRSC-System-1 architecture. The figure was created before development as a tool that helps guide development.

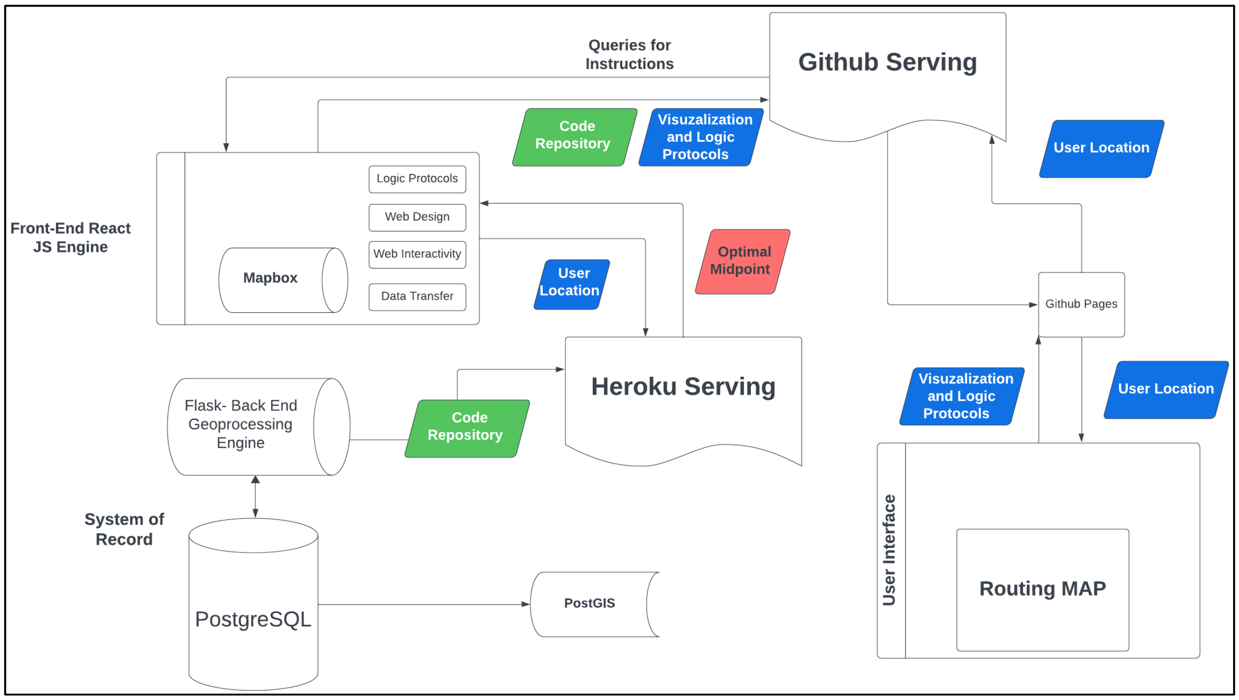


Figure 11. GLRSC-System-1 architecture

The figure is divided by major groups. The major groups that determine the functionality of the application are the front-end React engine, the back-end Flask engine, the PostgreSQL database, and the user interface. These major groups incorporate or include extended functionality. For instance, the React engine includes Mapbox, version 3.2.0, for routing visualization (Mapbox, n.d.). The PostgreSQL database includes a spatial data extension, PostGIS. All four major groups are connected to cloud servers, Github and Heroku, that allow access to the connected system via the World Wide Web. Data and instructions are passed amongst major groups through the cloud providers.

Furthermore, all software development was done using an integrated development environment, IDE. IDE are software building toolsets for writing code and running projects. The IDE that was chosen for this project was IntelliJ IDEA version 2023.2.5 for its number of features and comprehensive abilities for software development (IntelliJ IDEA, n.d.). For example, IntelliJ IDEA has windows where a user can define the Python SDK or examine system dependencies. IntelliJ IDEA also has a great user experience that makes it easy to navigate the complexities of software development.

3.7 Design of the Spatial Database

GLRSC-System-1 must keep a record of data objects that are essential to the functionality of its services. For instance, for a user to create and access account at their leisure requires a system that has the capacity to store that information for later use. The data objects essential to system performance are recorded in a database. A record of past transactions essential to the system's performance are kept in a PostgreSQL database.

To successfully complete this project, a database with spatial data capabilities was used because spatial data is essential to the system's service and design. Spatial properties measure the location of the object in geographic space and its spatial dimensions. Because of the spatial properties associated with the data objects, GLRSC-System-1 requires a spatial database which is a database with an extension for supporting spatial data and a schema that holds objects with spatial properties. To satisfy this requirement, PostgreSQL is used with a PostGIS extension. Figure 12 shows the spatial database design for the system.

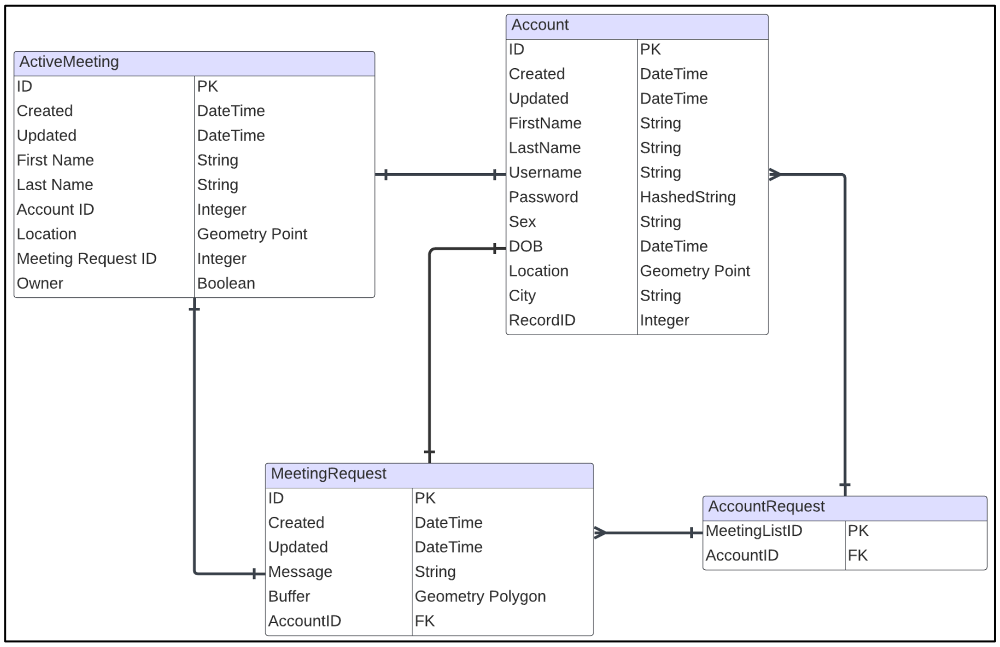


Figure 12. GLRSC-System-1 spatial database design

Figure 12 is a very simple design that consists of only four data objects. The simplicity of the design reflects the simplicity of GLRSC-System-1's functionality which concerns itself with the most minimal services required to achieve its objective. The database design has an account table, a meeting request table, an active meeting table, and an account meeting (potential participant) table.

Account objects are created when the user generates an account. Once the user logs in, the user can then give consent to share their location with the system. The user's location is stored in the account table as a geometry point. The meeting request object is created when a user sends a request; it holds a spatial property called buffer which is a geometry polygon. This buffer is the 2km buffer zone which GLRSC-System-1 uses to search for nearby potential participants. The account request object is created immediately after the meeting request object is created; it stores the account id numbers to record the accounts that lie within the buffer zone. The accounts that fall within the buffer zone are potential participants. Lastly, an active meeting record is created when a user accepts a request. Users within this table have confirmed that they would like to participate in a meeting and are, therefore, active participants in an active meeting.

Furthermore, PgAdmin version 4.8.1 was used to connect to a PostgreSQL database and verify the success of object relational mapping capabilities (PgAdmin, n.d.). PgAdmin provides a superior experience in database management and production. Using PgAdmin, developers can query the database, create a spatial index, examine available functions, and search for records.

Chapter 4 Methodology

Phase planning is a methodology to complete projects. For this thesis, phase planning was used to organize development related tasks and establish deadlines which helped with initiation and termination of chunks of essential duties. During research and development, four key phases were identified: multi-user environment, deployment, meeting service, and route visualization service. The four phases were built separately and chronologically. The phases also correspond to the essential components required to build any GLRSC system. Because this is an innovative system that has not been developed before, the methodology is also new. The work that most resembles comes from Petit (2020). Nevertheless, this methodology is new because it not only computes optimal midpoints using routing algorithms to arrange meetups, like Petit, but it also provides a multi-user environment that serves as a communication channel among nearby users. Furthermore, the methodology presented in this thesis presents a system designed for pedestrians rather than automobile users.

4.1 Phase 1 Multi-User Web Environment

In phase one, a local multi-user system was created, to allow users to make an account, log in, change their password, log out, delete their account, and store their location. Figure 13 shows the steps required to complete phase one.

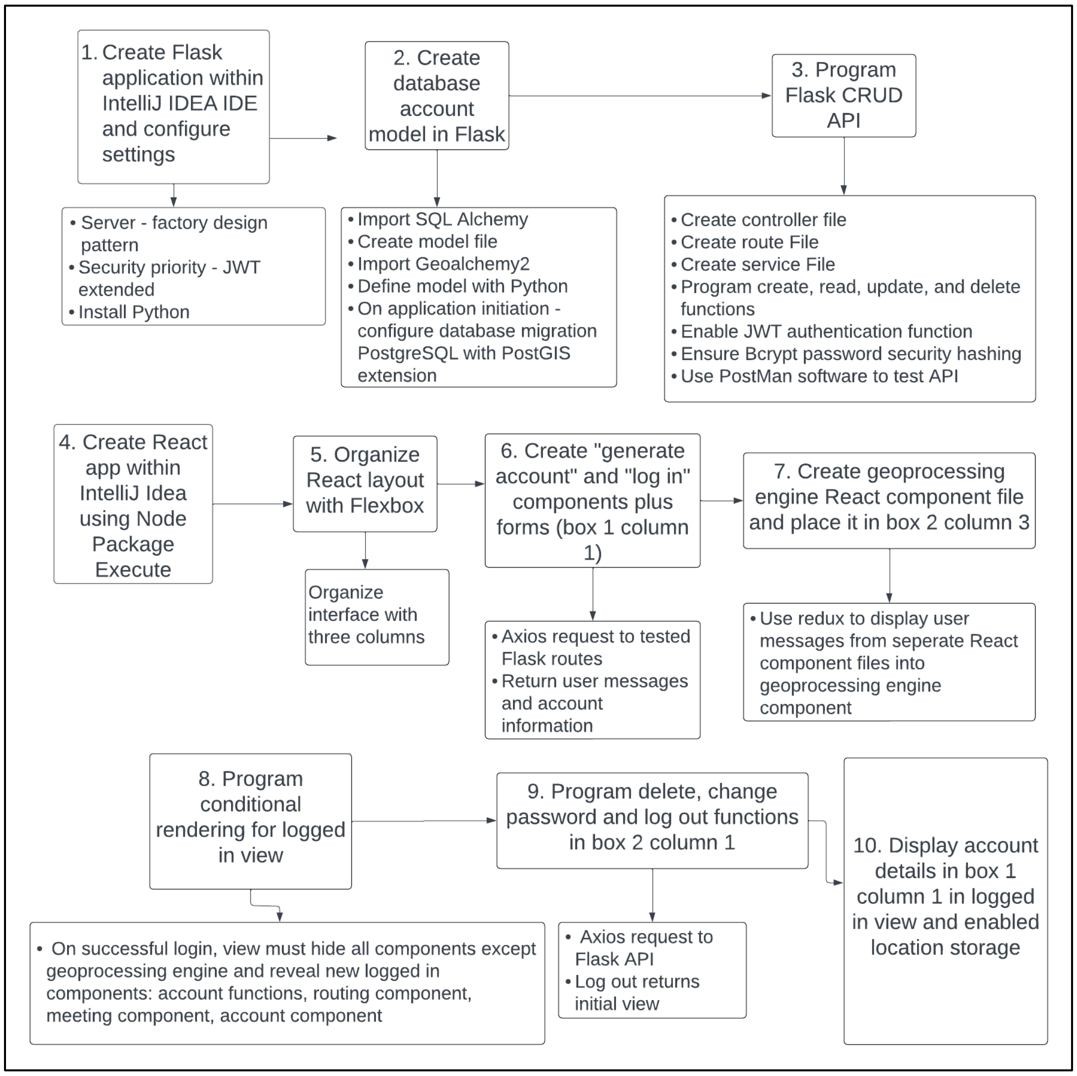

1. Create Flask application within IntelliJ IDEA IDE and configure settings
• Server - factory design pattern
• Security priority - JWT extended
• Install Python
2. Create database account model in Flask
• Import SQL Alchemy
• Create model file
• Import Geoalchemy2
• Define model with Python
• On application initiation - configure database migration PostgreSQL with PostGIS extension
3. Program Flask CRUD API
• Create controller file
• Create route File
• Create service File
• Program create, read, update, and delete functions
• Enable JWT authentication function
• Ensure Bcrypt password security hashing
• Use PostMan software to test API
4. Create React app within IntelliJ Idea using Node Package Execute
5. Organize React layout with Flexbox
Organize interface with three columns
6. Create "generate account" and "log in" components plus forms (box 1 column 1)
• Axios request to tested Flask routes
• Return user messages and account information
7. Create geoprocessing engine React component file and place it in box 2 column 3
• Use redux to display user messages from seperate React component files into geoprocessing engine component
8. Program conditional rendering for logged in view
• On successful login, view must hide all components except geoprocessing engine and reveal new logged in components: account functions, routing component, meeting component, account component
9. Program delete, change password and log out functions in box 2 column 1
• Axios request to Flask API
• Log out returns initial view
10. Display account details in box 1 column 1 in logged in view and enabled location storage

Figure 13. Phase 1 workflow

There are ten steps required to complete phase one. Figure 13 divides steps into sub-steps which it displays in bulleted lists. Within each step, different files or imports are made. To examine the exact composition of each file and the overarching file structure, the source code is made available at: https://github.com/spatial-moi.

4.1.1 Creation and Configuring of Flask Application

This section creates, configures, and initiates a production level system. An IDE is a software application that helps developers build software by providing editing, building, testing, and packaging capabilities (AWS 2024). In the case of this research, IntelliJ IDEA was used as the only IDE. After IntelliJ IDEA is installed, the official Flask website guide is referenced to initialize the application. Because Flask is python-based, a python installation is required. I referenced the official IntelliJ IDEA guide to configure a Python SDK (IDEA, n.d.). It is recommended best practice to activate a virtual environment for a Python SDK configuration. I ran a script that activates a virtual environment on the IDE terminal using the official Flask instructions (Flask, n.d.). I also referenced the Flask page to create a file that holds a minimal Flask application.

I created a file named server.py. The .py extension is one used for all Python files. The file was stored within a folder named src. I imported the Flask library within the server.py file. I then installed Flask through the IDE terminal using the script, “pip install flask” or through the IDE branch File>Project Structure>SDKS>Python 3.9>Install Flask. Once installed, the minimal flask application runs and executes successfully. IntelliJ IDEA has a run execute button on the top right corner of its interface that is used to run the back-end application. The application can only be run if the current file is set to server.py.

The paragraph above describes how to build the most minimal flask application in a single file. Theoretically, all application code can fit into this single file. However, it is best practice to build an application that is modular, maintainable, and scalable (Shyamal 2014). Building an entire application into a single file is not best practice. Therefore, the next task is to think ahead and configure the application so that it is production ready. To configure the application, within the src folder I created a config.py and an .env file. I created the config file in accordance with the official Flask configuration rules under the heading development/production (Flask, n.d.). I programmed a .env file which holds a variable called CONFIG_MODE which I can manually change its value to one of four options: development, testing, staging, or production. In the .env file, I create URL variables for each configuration mode and set them to a PostgreSQL URL. At this stage of development, I use the standard localhost PostgreSQL URL, “postgresql+psycopg2://postgres:postgres@localhost/postgres”. Later in the methodology, I change the production configuration to a production cloud PostgreSQL URL.

In a separate file, called config.py, I created python classes for each configuration mode and assigned to them three properties: a self-referencing Boolean that determines their configuration mode, a Boolean that determines if the application is in debug mode, and a database URL which calls on the .env file to retrieve the respective URL for each mode. Before I import the .env variable in server.py function through one of the classes in config.py. I first created an __init__.py file where I defined a function called create_app which takes as an argument one of the configuration python classes. The create_app function in the __init__.py file determines the setting of the application and then returns the application. The server.py file calls on the __init__.py function create_app and receives the Flask application; this design pattern is called an application factory and is required for production level applications. The __init__.py file is created according to the official Flask application factory standards (Flask, n.d.). Lastly, within the server.py file, a call to create_app includes a string named “CONFIG_MODE” which points to the .env file. I defined the configuration setting that the Flask application uses by manipulating the .env string called CONFIG_MODE to one of the four initial values: development, testing, staging, or production.

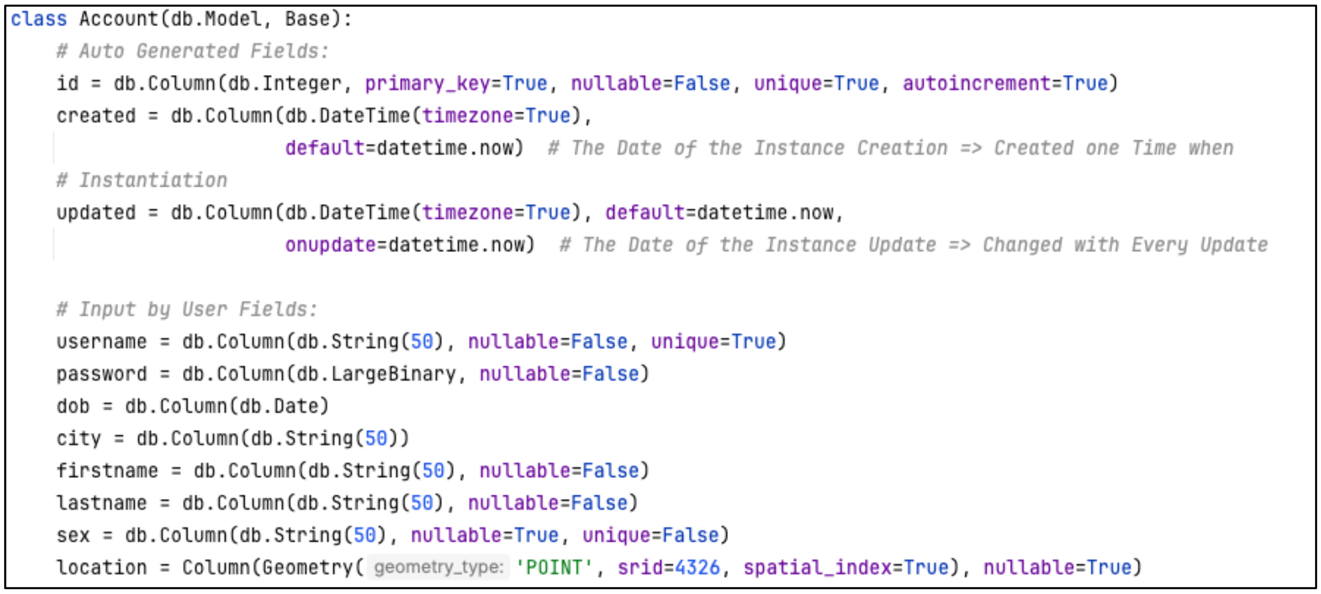
To conclude, flask_jwt_extended version 4.6.0, a python package, was installed and imported into the new __init__.py file in src folder. Additionally, the flask_bcrypt python package version 1.0.1 was also installed and imported into the __init__.py file. Respective JSON

Web Token (JWT) and Bcrypt objects are instantiated in __init__.py. These packages and their respective objects ensure that the system has open access to authentication services and security capabilities. The finishing application file structure for this step is a src folder with a server.py file, an __init__.py file, a config.py file, and a .env file. The application is configured for switching between development and production environment while also setting up the presets for required application services such as authentication and password encryption through flask_jwt_extended and flask_bcrypt.

4.1.2 Creation of Account Model

The Account Model is programmed to have eleven fields. These eleven fields are ID, created, updated, username, password, dob, city, firstname, lastname, sex, and location. The first step is to create a model.py file and store it within the src folder. Then, I referenced development the work of Yahia Qous (2023) who provides a reliable blueprint to build the account model. For the next step, I copied and pasted the model from Quos' tutorial on building a CRUD API using Python Flask and SQL Alchemy ORM with PostgreSQL. The instructions from this tutorial were followed including their instructions on how to install PostgreSQL and SQL Alchemy. Once PostgreSQL install is complete, PostGIS is enabled according to the official PostGIS install instructions (PostGIS, n.d.).

To summarize, I completed Qous' tutorial up to and including the model portion. Once complete, the model file is customized for the needs of GLRSC-System-1. The tutorial model fields are replaced with the eleven account fields mentioned previously. Then, to enable spatial python classes, I installed Geoalchemy2, version 0.14.3, within the IDE and imported the library into the model.py file, which enabled support for the Geometry Python type. Figure 14 shows the model including the location field which requires support from Geoalchemy2.



```python
class Account(db.Model, Base):
    # Auto Generated Fields:
    id = db.Column(db.Integer, primary_key=True, nullable=False, unique=True, autoincrement=True)
    created = db.Column(db.DateTime(timezone=True),
                        default=datetime.now)  # The Date of the Instance Creation => Created one Time when
    # Instantiation
    updated = db.Column(db.DateTime(timezone=True), default=datetime.now,
                        onupdate=datetime.now)  # The Date of the Instance Update => Changed with Every Update

    # Input by User Fields:
    username = db.Column(db.String(50), nullable=False, unique=True)
    password = db.Column(db.LargeBinary, nullable=False)
    dob = db.Column(db.Date)
    city = db.Column(db.String(50))
    firstname = db.Column(db.String(50), nullable=False)
    lastname = db.Column(db.String(50), nullable=False)
    sex = db.Column(db.String(50), nullable=True, unique=False)
    location = Column(Geometry( geometry_type: 'POINT', srid=4326, spatial_index=True), nullable=True)
```

Figure 14. Account model with spatial property

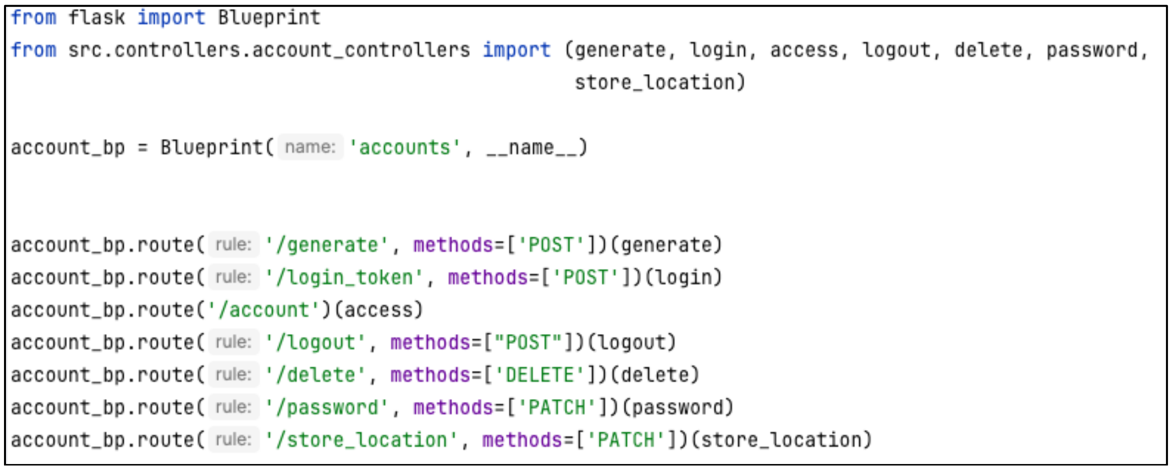
The code snippet shows the account model with its properties. ID, created, and updated are auto-generated fields which means that no code, other than the one written in Figure 14, is required to handle these data fields; the logic is automatic. Below the three auto-generated fields are the remaining eight fields. One of these is the location field. With the Geoalchemy2 import, Flask supports spatial classes (Geoalchemy, n.d.). For the location field, the corresponding PostgreSQL column is defined as one that stores point geometries. The spatial reference system is defined as 4326 which maps to the WGS 84 spatial reference system. The SRID ensures that the PostgreSQL database can position the location data into a geographic map. Lastly, the spatial index property is set as true which automatically ensures that a spatial index is created for the spatial data table on the PostgreSQL database (Geoalchemy, n.d.). The spatial index ensures that the system can conduct rapid spatial queries that are required for phase three (PostGIS, n.d.).

Figure 14 also shows that the Account class takes as input a Python database model and a Python base object, which are created using the SQLAlchemy library. SQLAlchemy, version 2.0.25, is a python package that supports object relational mapping (ORM), between the back-end and the database. The database object within the model.py file can be instantiated by writing db = SQLAlchemy () and the base can be instantiated by declaring Base = orm.declarative_base(). These are requirements for the Account model.

Once the model is completely defined, the db object needs to be imported into the __init__.py file which hold the Flask application. Quos (2023) is referenced for the changes that are required in the __init__.py file to update its capabilities to register the database object which makes a connection with the PostgreSQL database. The changes ensure that the ORM occurs during Flask startup; Flask startup creates a data table in the database as defined in the model.

4.1.3 Use of Flask CRUD API

In this step, I add three more files to the application file structure to continue with best practices of modularity and efficient organization. To begin, I create a file called routes.py in the src folder. This file routes incoming HTTP API calls. GLRSC-System-1 utilizes a Flask routing method known as blueprints which helps modularize code. Blueprints are incorporated as described in the official Flask blueprint documentation (Flask, n.d.). Figure 15 shows the blueprint routes needed for phase one.



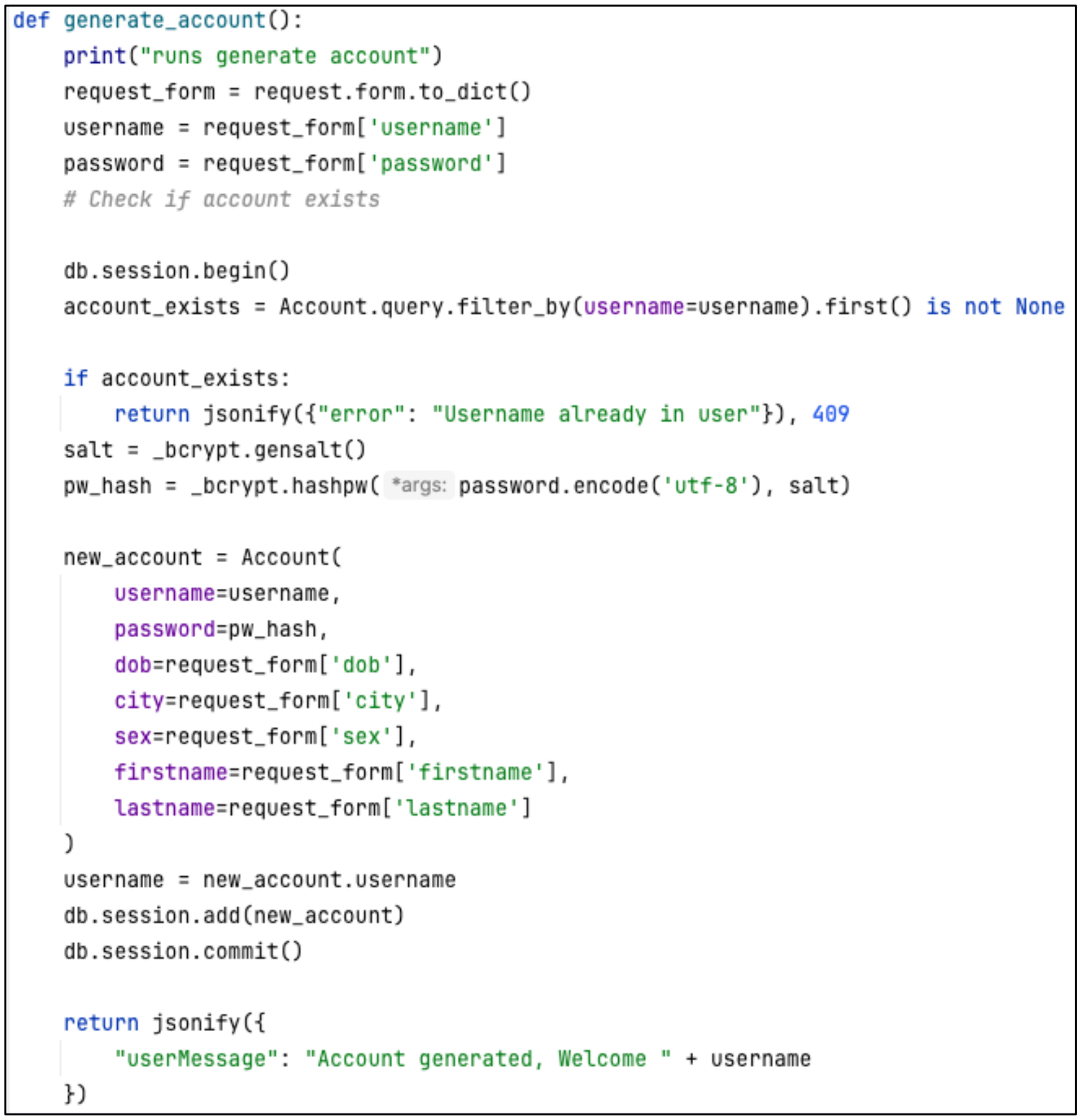
```
from flask import Blueprint
from src.controllers.account_controllers import (generate, login, access, logout, delete, password,
                                                 store_location)

account_bp = Blueprint( name: 'accounts', __name__)


account_bp.route( rule: '/generate', methods=['POST'])(generate)
account_bp.route( rule: '/login_token', methods=['POST'])(login)
account_bp.route('/account')(access)
account_bp.route( rule: '/logout', methods=["POST"])(logout)
account_bp.route( rule: '/delete', methods=['DELETE'])(delete)
account_bp.route( rule: '/password', methods=['PATCH'])(password)
account_bp.route( rule: '/store_location', methods=['PATCH'])(store_location)
```

Figure 15. CRUD routes using Flask blueprint in routes.py

CRUD stands for generic create, read, update, and delete commands. Most actions, in any system, can be defined as CRUD. Figure 15 shows six core actions required in most multi-user environments. A user must be able to create, or generate, an account. A user must be able to login. A user must be able to access their account. A user must be able to log out of their account. A user must be able to delete their account. A user must be able to change their password. These routes are important parts of the development effort that become useful when developing front-end API calls. Notice that in Figure 15, the six functions are imported from a file called account_controllers.

I created two new folders: one called controllers and the other called services. In the controller folder, I created a file called account_controllers.py; this file is used to define the exact functions that are executed per each route. In the services folder, I created a file called account_service.py, which describes and contains advanced protocols and rules for each function according to the required specifications. The account_service.py is separated from account_controllers.py to make the application code more organized.

Account_service.py contains the code that utilizes the authentication and security capabilities made possible in the __init__.py file. The functions need to be created in the order listed in Figure 15 because they are dependent on one another. For instance, a user cannot log in if there is no account. A user cannot change a password if they are not identified and logged in. Figure 16 demonstrates the first function that is created and highlights how each function provisions the next.



```python
def generate_account():
    print("runs generate account")
    request_form = request.form.to_dict()
    username = request_form['username']
    password = request_form['password']
    # Check if account exists

    db.session.begin()
    account_exists = Account.query.filter_by(username=username).first() is not None

    if account_exists:
        return jsonify({"error": "Username already in user"}), 409
    salt = _bcrypt.gensalt()
    pw_hash = _bcrypt.hashpw( *args: password.encode('utf-8'), salt)

    new_account = Account(
        username=username,
        password=pw_hash,
        dob=request_form['dob'],
        city=request_form['city'],
        sex=request_form['sex'],
        firstname=request_form['firstname'],
        lastname=request_form['lastname']
    )
    username = new_account.username
    db.session.add(new_account)
    db.session.commit()

    return jsonify({
        "userMessage": "Account generated, Welcome " + username
    })
```

Figure 16. Generate account function in account_services

The generate account function acquires information from an HTTP call and then checks to see if an account already exists with the information given. If an account does exist, an error is thrown. However, if an account does not exist, steps are taken to generate the account. The first action that the script takes is to ensure user security by encrypting the user's chosen password so that it is inaccessible to developers. Then an account object is generated and is passed to a database function which maps, adds, and commits the object into a data table within the PostgreSQL database. Finally, a success message is returned.

Each of the six functions contain their own set of instructions but rely on one another. For instance, the log in function decrypts the user's password and produces a JSON Web Token (JWT) to provide more security. JWTs ensure that users do not need to send their password information across servers more than once. JWTs are implemented according to standards defined by the official Flask JWT package (Flask, n.d.). Furthermore, each of the following four functions, delete, log out, change password, and access account, require that a JWT be attached as a header in any HTTP call.

Once I defined all the functions, I used Postman software, version 10.24.16, to test the back-end routes (Postman, n.d.). It is important to test the routes at this stage of development to ensure that they run successfully. Once they have run successfully, I proceeded to the next step in GLRSC-System-1 development process.

4.1.4 Creation of React Application

My next step was to create a React application using IntelliJ IDEA IDE. I created a new project gave it the name of the project. Next, using the official React website as a reference, I used the IDE terminal to run the command "npx create-next-app@latest" in the project's directory (React, n.d.). This script creates a react app in the project folder. To start the app, I used the command "npm start" this command opens a page on the web browser with a default React homepage that can be removed later (React, n.d.).

4.1.5 Organization of User Interface with Flexbox

To organize the user interface only two files are needed. App.js and App.css are default files that are provided with every React project. Therefore, I did not create any additional files in this step. The App.js file contains a return function that contains the HTML for the default React homepage. I deleted the default information and kept only the main parent wrapper div with a class name of “App”. A div is an HTML element used to organize web page content (Mdn web docs 2023). I programmed the “App” parent div to have three div children each with a class name of “column” plus their number in consecutive order. For instance, the first child div has a class name of “column-1”. The second child div has a class name of “column-2”. The third child div has a class name of “column-3”.

The first column has two separate child div with the naming rule “column number” then “box number”. For instance, the first column has two child div. The first child div has a class name of “column1-box1”. The second child div has a class name of “column1-box2”. This format proceeds for each column. The second column has three child div. The third column has two child div.

Once I wrote the HTML within the return function of App.js, I applied CSS to the elements in the App.css file. I used a CSS tool called Flexbox to create an organized box layout as seen in Figure 17.

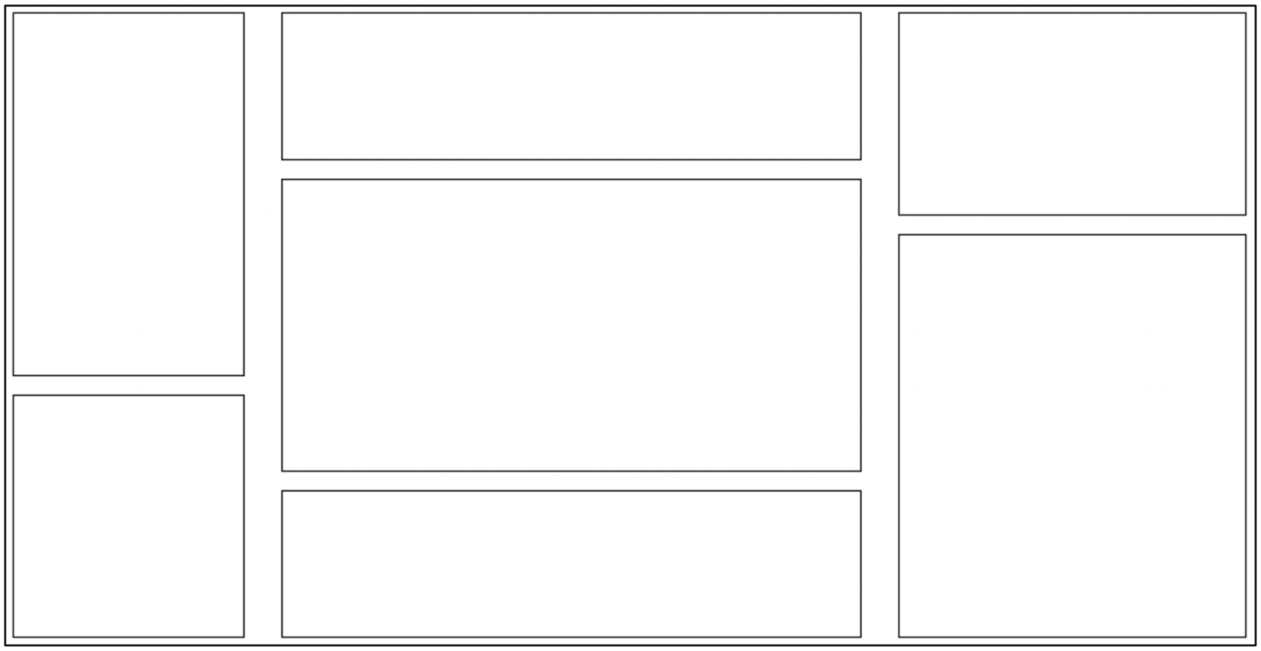


Figure 17. GLRSC-System-1 interface setup

I referenced the official Flexbox page to learn how to apply stylistic rules and methods (Mdn web docs 2023). The class names that were given for each div are referenced within the App.css file. Using CSS, I gave black borders to each box and defined their sizes and spacing. The result is GLRSC-System-1 layout which provides a minimalistic single-page application environment where each black box loads different interface requirements.

4.1.6 Development of Introduction, GenerateAccount, and LogIn Components

In a React application, HTML is divided into components which are stored in separate files (React, n.d.). Not only do these components store and produce HTML, which is then seen on the user interface, but they also store application logic that corresponds to the HTML that they display. Once each component is securely defined to produce the logic and HTML that is desired, they are brought in by name to the App.js file and their name is written within one of the respective HTML div that were defined in 4.1.5. This is the steps I took to program all React components and have them display on the interface.

To program the Introduction component, I created a folder called detailer within the default src folder. I then created and programmed a file named Introduction.js inside the detailer folder. The introduction component renders a paragraph text that informs the reader about GLRSC-System-1. In the introduction there is no data manipulation or HTTP call to the back-end API. Therefore, this was a very simple step to accomplish because I only needed to create the React component and place HTML in the default return function. I then imported the component into App.js and placed it within a div.

I gave each div in App.js a className, which is a reference that React uses to apply CSS. A class is a type of object that is named according to what the object type does within the context of the system. For instance, an object that stores the user's name, age, password, and email would likely be responsible for dealing with that user's account. Therefore, the class is given a className of account. Each black box in the layout is given a column and box numbers as a className to distinguish the position within the interface; this className protocol is used for GLRSC-System-1 but that does not mean that all React applications use the same className conventions. Typically, developers get to decide the class name conventions according to what makes sense for them and their systems. Once a React component is imported into the layout by writing its name in the div, it renders on the user interface unless a conditional is programmed to hide the component.

To generate an account in the database from the front end, there needs to be both a button that accesses the back-end routes and a front-end user facing form that can be completed which includes the account fields defined in the account model. The generate account ability is of particular importance because it set the stage and organizational flow for how I programmed the remaining CRUD actions React. I copied, pasted, and adjusted the JavaScript programming code in the GenerateAccount.js and GenerateAccountModal.js files to make React components for the remaining CRUD actions. Figure 18 shows a file called GenerateAccount.js. This file serves as a template for other CRUD actions which require both a button and a pop-up modal form.

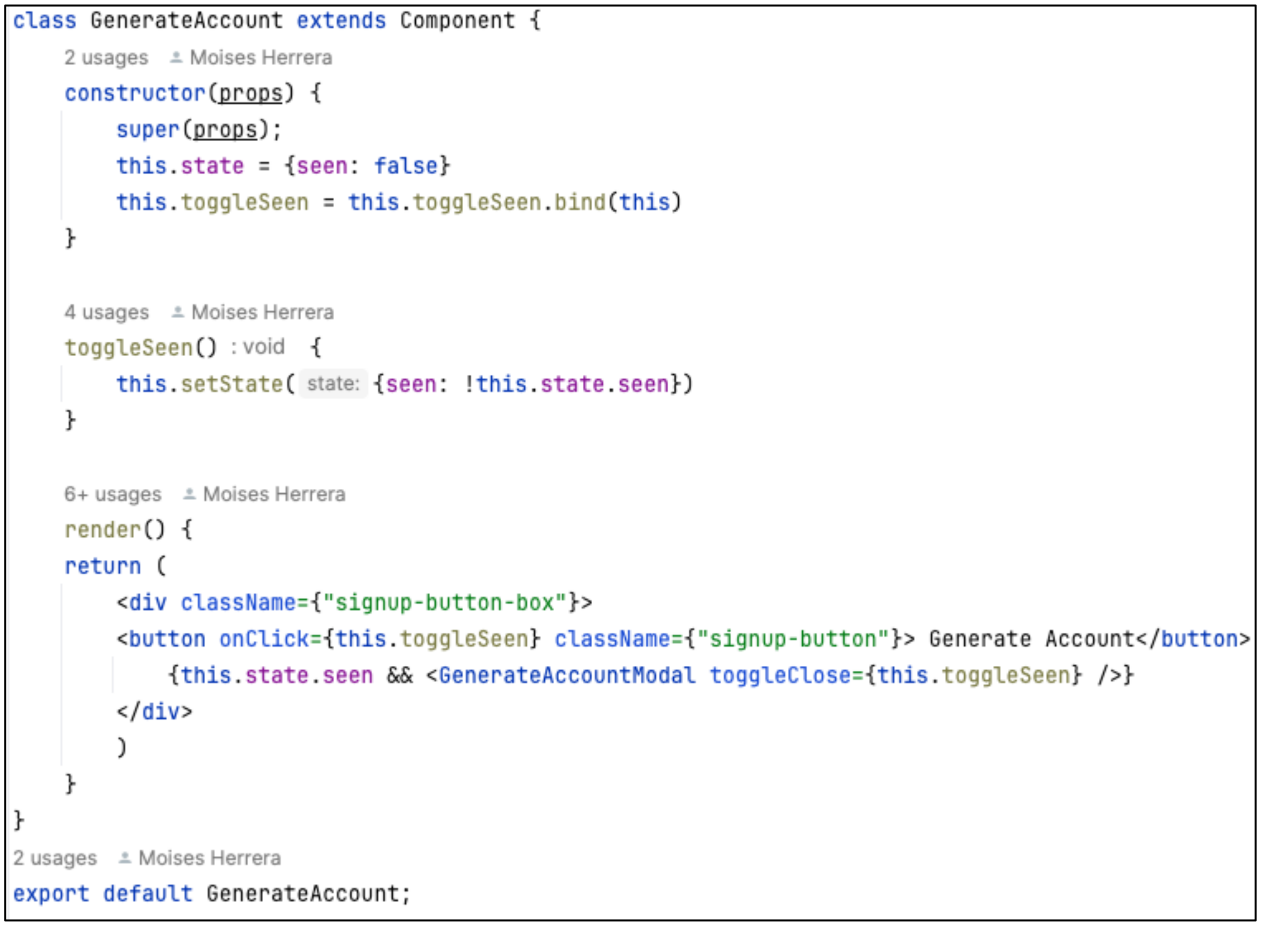

```
class GenerateAccount extends Component {
    2 usages  Moises Herrera
    constructor(props) {
        super(props);
        this.state = {seen: false}
        this.toggleSeen = this.toggleSeen.bind(this)
    }

    4 usages  Moises Herrera
    toggleSeen() : void  {
        this.setState( state: {seen: !this.state.seen})
    }

    6+ usages  Moises Herrera
    render() {
    return (
        <div className={"signup-button-box"}>
        <button onClick={this.toggleSeen} className={"signup-button"}> Generate Account</button>
            {this.state.seen && <GenerateAccountModal toggleClose={this.toggleSeen} />}
        </div>
        )
    }
}
2 usages  Moises Herrera
export default GenerateAccount;
```

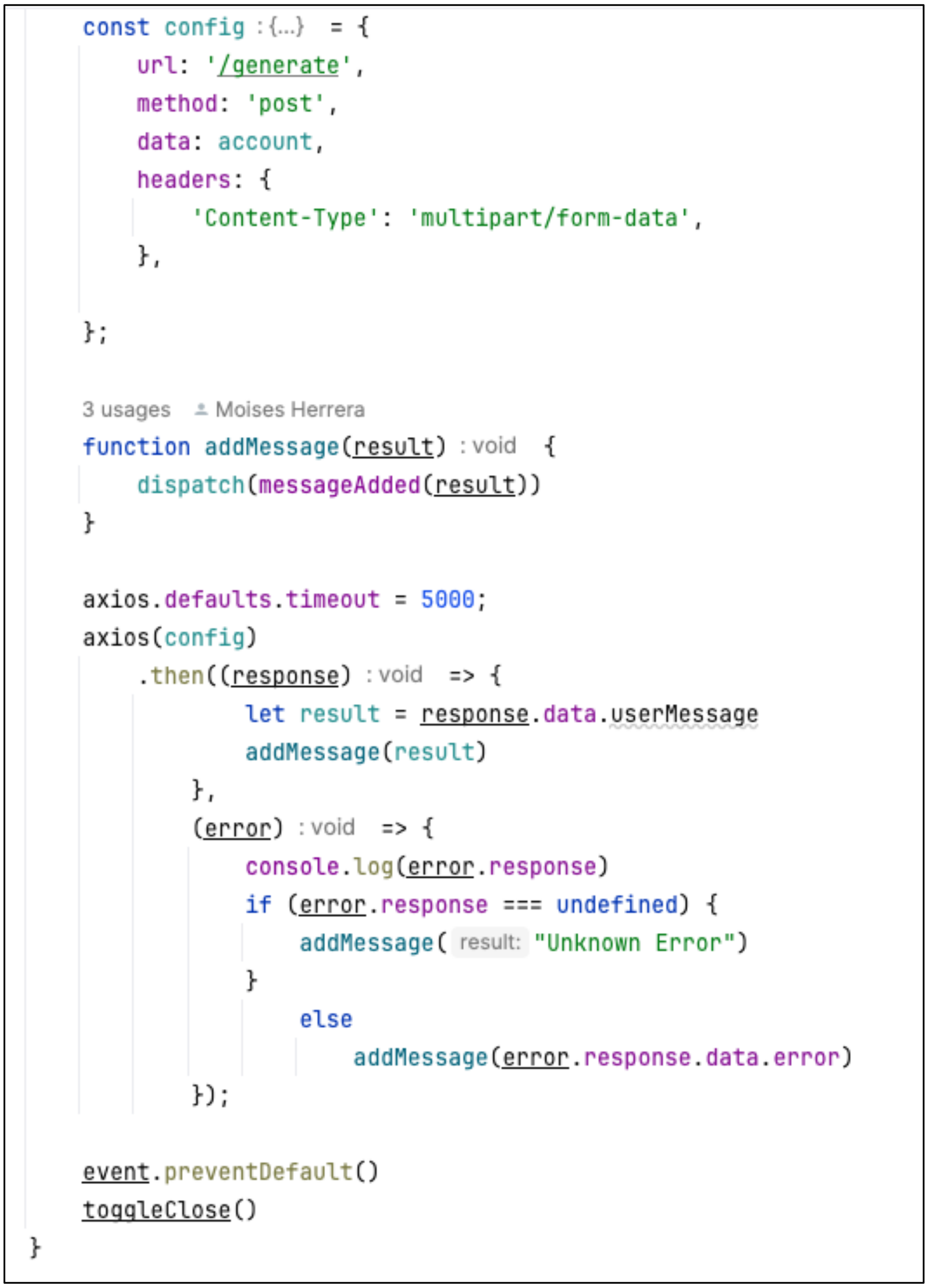
Figure 18. GenerateAccount component

The first function in the GenerateAccount class is the constructor function. The constructor function is required to initialize a GenerateAccount global object with the defined properties within its brackets. The properties include a Boolean variable named “seen” and a function named “toggleSeen” which changes the state of “seen” to true rather than false.

The return function returns an HTML button element wrapped in a separate div. The wrapper div is used for stylistic purposes only and is not relevant to the application logic. Within the HTML button element, a click event point to the “toggleSeen” function indicates that the “seen” variable is switched to true when the button is clicked. The last thing I programmed in this file is a conditional below the HTML button element. The syntax of brackets within the return statement are React’s version of enabling conditional rendering. The bracket code states that if the “seen” variable is set to true, show a component named GenerateAccountModal stored in a file named GenerateAccountModal.js. Lastly, I imported the GenerateAccount component into the App.js file inside the first column-first box div to ensure that the user has access to the generate account feature.

The file GenerateAccountModal.js is the second file that is used to program the generate account functionality. The GenerateAccountModal component returns an HTML form where users can input information for the following account model variables: username, password, first name, last name, date of birth, sex, and city. The HTML form also has a submit button. The submit button triggers a function within GenerateAccountModal called generateAccount which creates an account object named account with the information that the user submitted.

The account information is stored within a JavaScript dictionary named config which stores key value pairs with information that is required to make a request to the back-end. For this application, I used axios as an HTTP client package. Axios allows the application to make HTTP requests to a back-end server. Figure 19 depicts how the config dictionary is used within an axios POST request to generate an account.



```
    const config :{...}  = {
        url: '/generate',
        method: 'post',
        data: account,
        headers: {
            'Content-Type': 'multipart/form-data',
        },

    };

    3 usages   Moises Herrera
    function addMessage(result) : void  {
        dispatch(messageAdded(result))
    }

    axios.defaults.timeout = 5000;
    axios(config)
        .then((response) : void  => {
                let result = response.data.userMessage
                addMessage(result)
            },
            (error) : void  => {
                console.log(error.response)
                if (error.response === undefined) {
                    addMessage( result: "Unknown Error")
                }
                    else
                        addMessage(error.response.data.error)
            });

    event.preventDefault()
    toggleClose()
}
```

Figure 19. Axios post request to generate account

The config dictionary stores the account object with the form information. It also holds a method key with the value of post which indicates that the function is meant to create data, one of the four CRUD actions. The url key holds the route url seen in routes.py file within the Flask back-end. The headers key describes the type of data transfer standard that is used to transfer account data.

A couple lines below the config dictionary is the axios request. The config dictionary is passed as an argument. The axios function with the config argument sends the request. Then the .then(response) call captures a response from the server. Finally, at the very end, a toggleClose function closes the modal.

The log in functionality follows the same structure and strategy in LogIn.js component. The only difference is that the config data key in LogIn.js holds different information. The config data only holds username and password as input. In the LogIn.js component, the server returns a JWT access_token. LogIn.js component uses local storage to store the JWT access_token for future account actions. The code in GenerateAccount.js and LogIn.js can be used interchangeably and updated according to program needs. The LogIn.js component is also imported into App.js column1-box1 and its name is written inside the div.

4.1.7 Use of Redux to Program User Messages in the Geoprocessing Engine

To uphold the projects commitment to data transparency, GLRSC-System-1 required a dedicated interface element that is responsible for printing messages to the user. This interface element is called the Geoprocessing Engine or Geoprocessing_Engine.js in the React codebase. The messages that the Geoprocessing Engine prints ranges according to the functionality of the system. For instance, when a user generates an account, the Geoprocessing Engine prints a message that the API call was successful. When a user decides to give the system permission to their location, the Geoprocessing Engine informs the user that their location data is stored and what is done with it. The goal for the Geoprocessing Engine is that it communicates algorithmic processes, such as routing processes. This helps make the system informative and scientific as well as educational. The Geoprocessing Engine facilitates a user experience that is rich in knowledge and openness.

To create such a panel, all React components, such as GenerateAccount and LogIn need access to a centralized storage unit where they can send messages to, upon successful or erroneous completion. As a solution, Redux, version 4.2.1, is a predictable centralized state container, that I used for storing variables that are needed across the application file structure (Redux, n.d.). I imported a redux store into individual React components and then added messages into the store within independent file contexts.

Then I created a folder called geoprocessing_engine. Then within the folder, I created a file called Geoprocessing_Engine.js. I followed the instructions from the official Redux installation page. Then created a folder called redux under src. Within the redux folder I created two files, geMessageListSlice.js and store.js. Then, I referenced the redux installation instructions to access the code that should be placed in both files (Redux, n.d.). Within geMessageListSlice.js and store.js, I created a variable and made it accessible to all parts of the application. The global redux variable is accessed within React components that require global storage capabilities. Within the .then function of an axios CRUD request, global redux variables are accessed, called, modified, and stored again in Redux.

For example, once a user logs in, a redux variable is accessed and modified. Simultaneously within the GeoprocessingEngine.js file, the modified redux variable is read, and the modified variable is printed onto an HTML div. In App.js, column3-box2 is designated as the panel for the Geoprocessing Engine. The GeoprocessingEngine.js component is imported into App.js and its name is written inside column3-box2. Redux set up is done within the file structure and no external programs or websites need to be used. The tutorial is very comprehensive, and it is easy to adjust it to the needs of GLRSC-System-1 or any other system.

4.1.8 Development of Conditional Rendering for Logged in View

Redux variables can also be used to program conditional rendering in App.js. In GLRSC-System-1, the user should expect an initial landing page where they can either generate an account or log in. Once logged in, the generate account and log in buttons should disappear from the user interface. In their place, GLRSC-System-1 should render new components that are required for the system's service.

In other words, the system itself must know if the user is logged in or out, and this information must be updated at the exact moment that a user successfully logs in. I created a global variable called, 'loggedIn' following the same instructions from the official redux page (Redux, n.d.). The new variable's default value is "false". In the LogIn.js component the redux variable is modified and set to true. In App.js, the updated stored variable is re-accessed and read. The system applies conditional rendering using the loggedIn variable to hide components and show others. This is done by pairing the redux variable with the React component using the format "{reduxVariable && </ReactComponent.js>}" within the corresponding div in App.js.

4.1.9 Development of Change Password, Delete, and Log Out Functions

Change password functionality is divided into two files: ChangePassword.js and ChangePasswordModal.js. Both are made by using GenerateAccount.js and GenerateAccountModal.js as templates. The difference is that the data key in the config dictionary changes to include only the data that is necessary. For the change password there are only two variables stored in config, the user’s current password and the new password. The route changes from “/generate” to “/password”. The method changes from “post” to “patch”. Patch is an update function from the four CRUD functions.

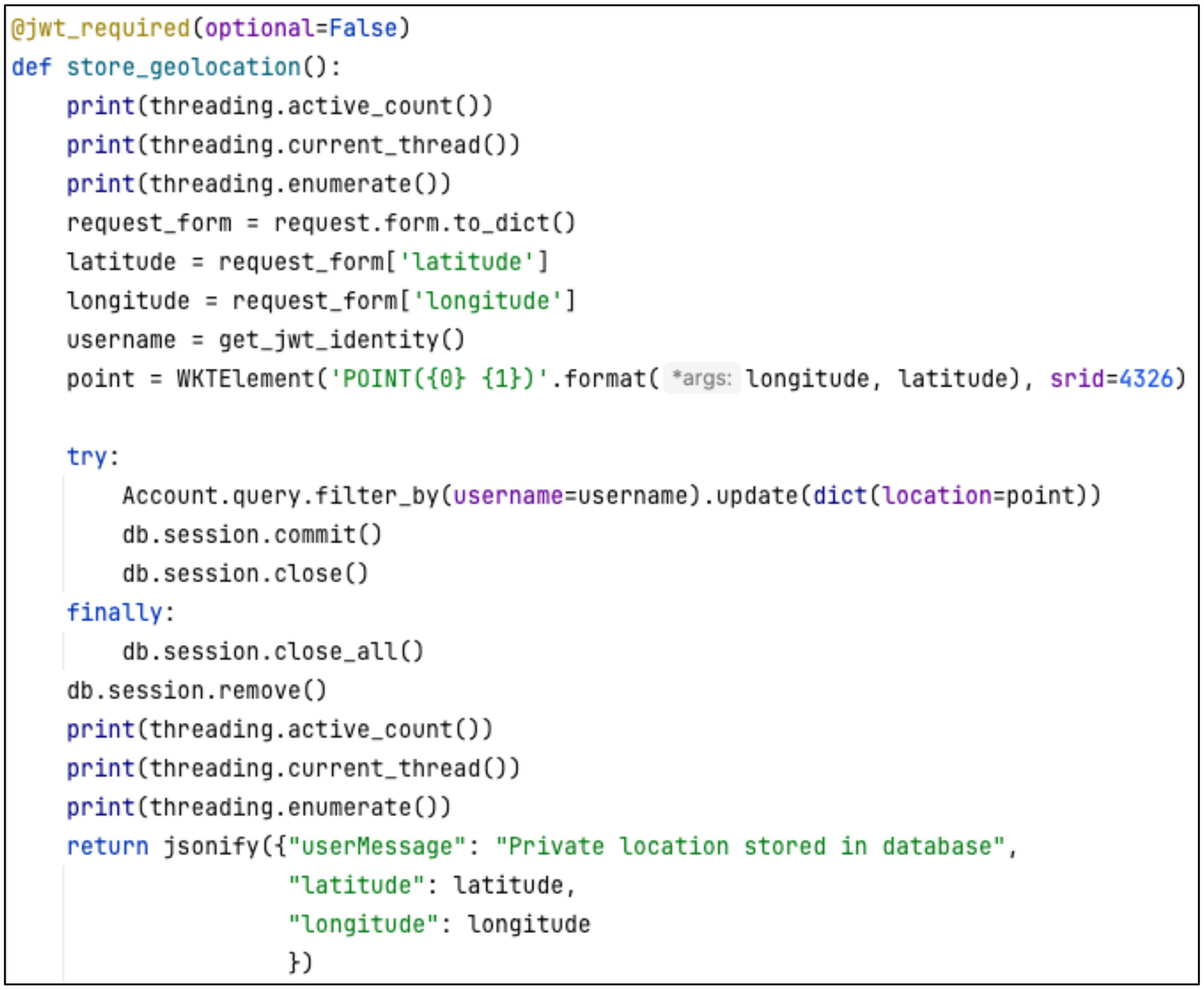
In ChangePasswordModal.js, Delete.js, and LogOut.js, I included an authorization JWT header key in the config dictionary; this is done to add advanced security and password authentication capabilities. At the time of logging, the system back-end generates and returns a JWT access token which is then stored in the system’s front-end local storage. I programmed the system to require authorization for change password, delete, and logout HTTP calls. Therefore, the system calls on local storage to access the JWT access token which are then placed as headers in HTTP calls.

I reduced delete and log out functionality to one file each: Delete.js and LogOut.js. There is no need to include a modal because neither require the user to submit information. Both Delete.js and LogOut.js make direct axios calls without including data.

4.1.10 Display of Account Details and Enable Location Storage

As mentioned in Section 4.1.8, when the user logs in, the GenerateAccount.js and LogIn.js buttons disappear. In their place, account details are presented. To program account details, I created a new file called Account.js. The Account.js component also makes an HTTP request, but it does not pass data like LogOut.js and Delete.js. Instead, it makes a “Get” request to the “/account” url in the Flask back-end which returns a list of account details such as first name, last name, sex, dob, and city.

Finally, I programmed a button with location storage capabilities and included it within the Account.js component. The location button is tied to a click function called storeLocation within Account.js. The store location button raises a prompt to the user to allow or reject location sharing. If the user gives the prompt permission, the user's location is stored in a React variable and the system triggers an axios HTTP request to update the location field in the account model. Figure 20 shows the axios request's destination code.



```python
@jwt_required(optional=False)
def store_geolocation():
    print(threading.active_count())
    print(threading.current_thread())
    print(threading.enumerate())
    request_form = request.form.to_dict()
    latitude = request_form['latitude']
    longitude = request_form['longitude']
    username = get_jwt_identity()
    point = WKTElement('POINT({0} {1})'.format( *args: longitude, latitude), srid=4326)

    try:
        Account.query.filter_by(username=username).update(dict(location=point))
        db.session.commit()
        db.session.close()
    finally:
        db.session.close_all()
    db.session.remove()
    print(threading.active_count())
    print(threading.current_thread())
    print(threading.enumerate())
    return jsonify({"userMessage": "Private location stored in database",
                    "latitude": latitude,
                    "longitude": longitude
                    })
```

Figure 20. Store geolocation python function

In the store_geolocation function, the system stores integer values of latitude and longitude from the axios data. However, PostgreSQL requires the WKTElement to process spatial data. Therefore, the system first converts the integer latitude and longitude coordinates into a WKTElement. Then the system makes an update command using SQLAlchemy syntax. If the action is successful, the system returns a success message that informs the user that their location has been stored. The system displays the message in GeoprocessingEngine.js.

4.2 Phase 2: System Deployment

GLRSC-System-1 deployment is achieved using Github and Heroku which are remote application repositories that can serve code to clients. Using Github-Heroku for deployment simplifies the process by automating server setup, network administration and database tuning (Pattan 2023). It is a pay to use bundle where developers pay according to HTTP traffic to their hosted API. Furthermore, the strategy that is used to ensure continuous user access to updated versions of the application is known as continuous deployment and it can be done using the Heroku-Github pair. Continuous deployment emphasizes minor changes to the application with each deployment. By not bundling all updates into a single version, application bugs can be more swiftly identified. For each deployment, Heroku runs automated checks to ensure that each update is viable for production. Heroku checks the code for viability and then pushes the changes it to a live environment. HTTP requests can be sent from a live user to access API resources and computations. This project stores the system files in a Github repository and uses Github Pages for hosting its domain making the system accessible to users. Github Pages is a service within Github that provides automated and quick domain name assignment which ensures that the user has a custom URL to access the system interface.

This section covers the first deployments for the back-end and the front-end. The section uses the previously discussed phase 1 code as the first deployment unit. The section begins with back-end deployment using Heroku and Github. It is followed by a section on front-end deployment using Github and Github Pages. The section ends with a discussion on maintenance.

The maintenance section discusses relevant information needed to successfully execute continuous deployment for this project.

4.2.1 Back-End Deployment using Heroku-Github

First, I created a Heroku account by visiting the website and following the graphical user interface (GUI) instructions. After I created an account, I navigated to the dashboard, and selected create application. I assigned the application a name and then clicked create. The name used for this project was “GLRSC-System-1”.

After I created the application, I navigated to the dashboard with the application name which holds seven tabs: overview, resources, deploy, metrics, activity, access, and settings. In the resources tab there is a horizontal bar to configure add ins where I selected the Heroku Postgres add in. The Heroku Postgres add in is used to create and manage a PostgreSQL database. I followed the GUI instructions to create the Heroku PostgreSQL instance for production while keeping the local PostgreSQL instance for development. A PostgreSQL database is required because that is the database that was used in phase 1 and GLRSC-System-1 is configured to use PostgreSQL. To use the Heroku PostgreSQL service, it is also required to enter a payment method in Account Settings > Billing.

The next step I took was to install the Heroku command line interface (CLI). The CLI provides functionality and commands that are required to deploy GLRSC-System-1. The official Heroku guide is used to install the CLI (Heroku, n.d.). Brew is used as the command line install option.

Once installed, I typed “heroku login” in the CLI and clicked enter. This allowed me to connect to the Heroku application from my computer and enable needed commands. Once logged in, I connected to the PostgreSQL database instance using the command “heroku pg:psql". Once connected to the PostgreSQL instance, I ran the command, "CREATE EXTENSION postgis". PostGIS is a required extension. In Phase 1, the model was created with a geometry point. If PostGIS is not installed at this point, deployment fails because Heroku cannot store geometry points without the PostGIS extension.

On the Heroku GUI, I navigated to the application settings. There is a section titled, "Config Vars" where there is a variable named DATABASE_URL. The key string for DATABASE_URL is the URL that is used to connect to the PostgreSQL instance. However, Heroku requires a slightly modified string. Therefore, I clicked the add variable button, and made a new variable named DATABASE_URI. I copied and pasted the string from DATABASE_URL and assigned it to DATABASE_URI. I then modified the string by adding "ql" to the prefix "postgres". The DATABASE_URI string is identical to the DATABASE_URL string except for the distinct prefix "postgresql" rather than "postgres". I then copied and pasted the DATABASE_URI string to the production configuration PostgreSQL URL in the .env file in the Flask back-end. I also changed the CONFIG_MODE variable to "production" rather than "development". In config.py, I updated the variable DATABASE_URI in the ProductionConfig python class to match the DATABASE_URI variable described in the previous paragraph. These specific changes are made to configure the application for production.

Next, I configured the application to allow cross origin resources. To do this, I modified the __init__.py file by adding an array variable called cors_origin that holds different URL sites that can access the Flask back-end routes. Secondly, I used the CORS python package version 4.0.0 to define a CORS object that allows from cross-origin resource transfer. CORS stands for "Cross-Origin-Resource-Sharing". The CORS object also takes the cors_origin as an argument.

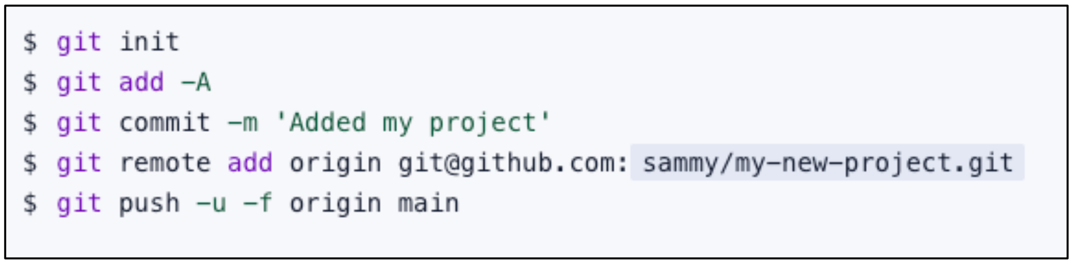
Next, I ran the command “pip3 freeze > requirements.txt” in the Flask repository in the IDE terminal. This command created a file called requirements.txt in the Flask file structure. This file is required for Heroku deployment. The command created a file with project dependencies and defined version numbers. This process is automatic. However, the result is a list of roughly 400 dependencies. Most of the dependencies are extraneous. I removed many of them and only kept the ones that I explicitly imported into Flask python files. Heroku installs other needed dependencies automatically upon deployment.

Trimming the dependencies is important because doing so reduces the slug size. There is a maximum slug size of 500MB for standard Heroku deployment. Keeping the 400 original dependencies guarantees a slug size of over 500MB and halt deployment. Furthermore, more dependencies may be added in Phase 3 and 4 so it is best to plan and optimize slug size.

Next, I created a file labeled “Procfile”. This file is another Heroku requirement. In the file, I typed “web: gunicorn server:app” on one line. In deployment, Heroku reads the Procfile to identify the Python application.

Next, I created a file called “runtime.txt”. In it, I typed 3.9.6. This number indicates the Python version used to develop the application. Heroku identifies the runtime.txt and makes a deployment using the specified Python version. As a necessary pair, I changed the Heroku Stacks to 20. Heroku Stacks is the version number that supports different programming languages and version. By default, Heroku Stacks is set to 22. However, Heroku Stacks 22 is not compatible with Python 3.9.6. To configure GLRSC-System-1 to use Heroku Stacks 20, I run “heroku stack:set heroku-20” in the Heroku CLI.

The application was then ready for deployment. For Heroku to run checks on the application, the application must first be stored in a cloud server. I used Github to store the application. It is required to create a Github account and then create a repository on that account. For this project, the repository for the back-end is called, “GLRSC_S1_FLASK”. S1 stands for system 1. This project uses a tutorial by Cerminara (2022) for instructions on back-end deployment. Figure 21 summarizes the processes involved according to Cerminara.



```
$ git init
$ git add -A
$ git commit -m 'Added my project'
$ git remote add origin git@github.com: sammy/my-new-project.git
$ git push -u -f origin main
```

Figure 21. Deploy local repository to Github

The commands in Figure 21 are run from the MacOS terminal at the head of the project directory. The first command, “git init”, initializes a git repository; it enables version control. The next command, “git add -A”, creates an update object and adds all files in their current state to the update. The “git commit” command commits the changes. It bundles the update and prepares it to be pushed remotely to the Github repository. The quotations following the command should give a description of that update in a couple of words. The next command, “git remote add origin”, is used to pair the local repository to the Github repository created with the name, “GLRSC_S1_FLASK”. Finally, “git push -u -f origin main” executes deployment to Github. Once I ran the code above, the repository was made available on Github.

To deploy from Heroku, on the GUI, I navigated to the application created previously in this section. In the deploy tab, under deployment method, I selected Github, typed “GLRSC_S1_FLASK”, selected the master branch and clicked connect. The application then deploys successfully. I then viewed the deploy back-end webpage through a link that Heroku provides. To access HTTP requests and logs that are sent to the API, I ran the command, "heroku logs –tail –app=heroku-app-name", from the Heroku CLI which served as a great resource to debug and to monitor the API.

4.2.2 Front-End Deployment using Github-Github Pages

Within the React front-end system, the first course of action, was to modify the HTTP request URL to point to the Heroku deployed back-end. During the end of the Heroku build, right before successful deploy, the Heroku GUI provided the URL that is used to make HTTP requests. This URL is https://glrsc-system-1-27a1742ceb52.herokuapp.com.

I had to change all HTTP requests that pointed to localhost:5000 to instead point to https://glrsc-system-1-27a1742ceb52.herokuapp.com. However, while I developed Phase 3 and Phase 4, I reverted the URL paths back to localhost:5000 so that I can test the system without needing to run continuous deployment.

I used environment variables to replace the localhost URL. I first created a .env file at the root of the React directory. Within this file, I typed the variable REACT_APP_API_URL. When using environment variable in React, it is necessary to prefix the variable with REACT_APP. Any modifications without the prefix do not work. Next, I assigned the variable to the deployed Heroku URL. Assignment is easily done by placing an equal mark between the variable and the URL.

I made each HTTP request within the React app by creating a new variable called "path" in each of the necessary files. I assigned the "path" variable to REACT_APP_API_URL. Then in the config variable, for each Axios HTTP call, I prefixed the route with "path".

At this point of development, I updated the HTTP request structure to match in the following files: GenerateAccountModal.js, LogInModal.js, ChangePassword.js, DeleteProfile.js, LogOut.js, and Account.js.

I deployed the project to Github using the instructions from Figure 21. The final step is to use Github-Pages to host the site. Hosting on Github-Pages allows for a quick domain name assignment that simplifies the deployment process. Within the terminal from the React project root directory, I ran the commands: npm install, gh-pages –save-dev, npm run build, and gh-pages -d build. And then committed the changes using git and push to Github using the same syntax from Figure 21.

On Github, I navigated to settings and then pages. Under the source heading, I made sure that source pointed to gh-pages branch. Once source is set to gh-pages, deployment is automatic. The initial deployment takes only a few minutes. Github provided a link to the deployed site within settings > pages, right above the source heading.

4.2.3 Maintenance

For Phase 3 and Phase 4, testing is done in production as well as development. It is necessary to create different branches for different work settings. For both the Flask back-end and React front-end, I used git within the terminal to create different branches. Beginning from the root directory of each project, I ran "git checkout -b 'development" and then checkout to master using git where I ran "git checkout -b 'production'". These commands prepared and created the development and production branches. All local work is done within development and all production-level work done in production. This set up provided me with an organized workflow.

4.3 Phase 3: Meeting Service

GLRSC-System-1 required a meeting service that enables and processes meeting requests, executes necessary front and back-end spatial functions, and delivers vital meeting information to the user interface. The meeting service objective was to schedule an immediate meeting, within 30 minutes, and provide the latitude, longitude coordinates of the meeting place to all self-registered participants.

To achieve the meeting service objective, GLRSC-System-1 integrates a data flow which mimics the procedures of the GLRSC routing algorithm described in Figure 10. Figure 22 diagrams the GLRSC routing algorithm and data flow within the context of a deployed multi-user system and displays how the system meets the meeting service objective.

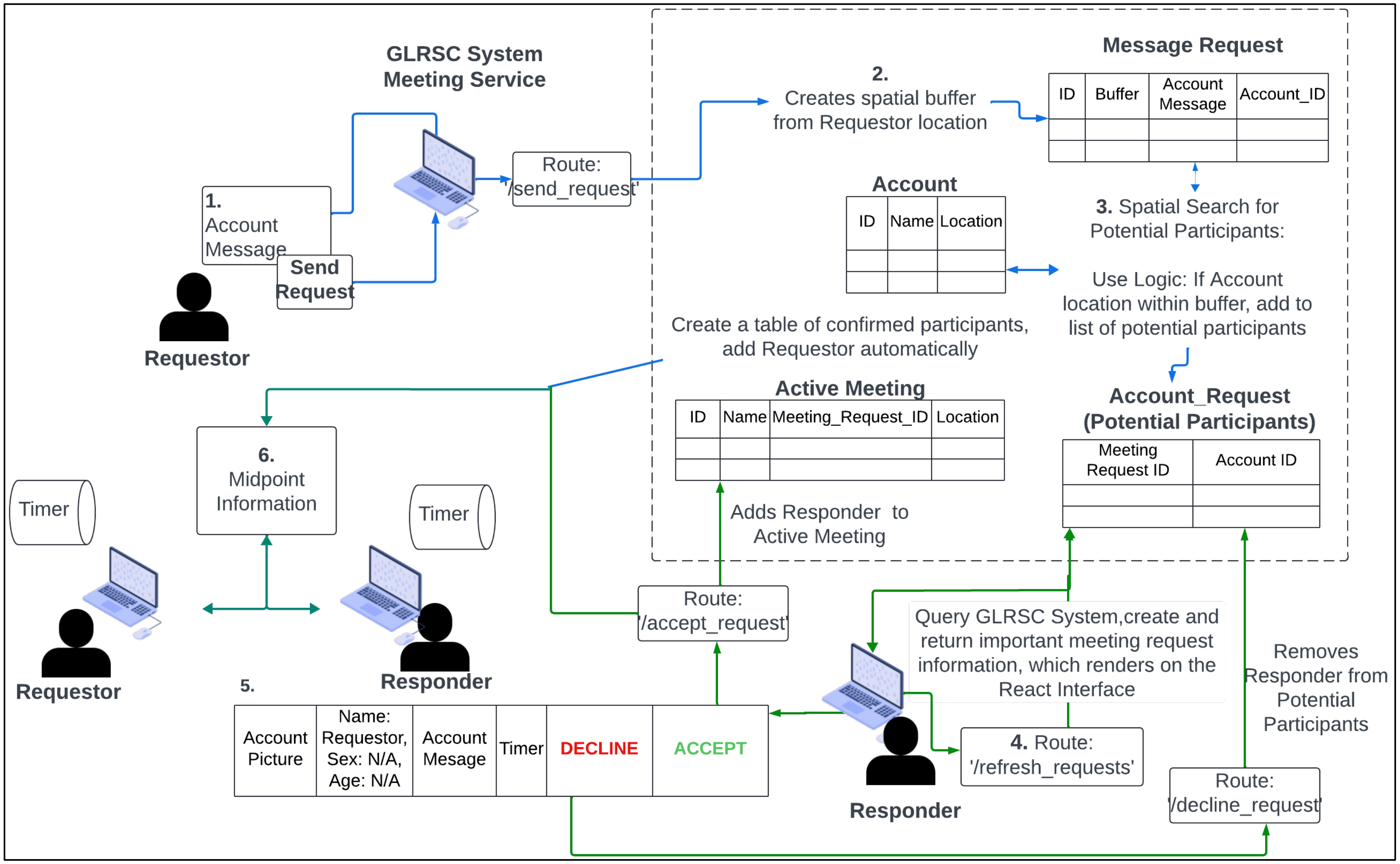

GLRSC System Meeting Service
1. Account Message
Send Request
Requestor
Route: '/send_request'
2. Creates spatial buffer from Requestor location
Message Request
ID | Buffer | Account Message | Account_ID
Account
ID | Name | Location
3. Spatial Search for Potential Participants:
Use Logic: If Account location within buffer, add to list of potential participants
Create a table of confirmed participants, add Requestor automatically
Active Meeting
ID | Name | Meeting_Request_ID | Location
Account_Request (Potential Participants)
Meeting Request ID | Account ID
Adds Responder to Active Meeting
6. Midpoint Information
Timer
Timer
Requestor
Responder
Route: '/accept_request'
Query GLRSC System,create and return important meeting request information, which renders on the React Interface
Removes Responder from Potential Participants
5.
Account Picture | Name: Requestor, Sex: N/A, Age: N/A | Account Mesage | Timer | DECLINE | ACCEPT
4. Route: '/refresh_requests'
Responder
Route: '/decline_request'

Figure 22. GLRSC-System-1 meeting service user flow

Figure 22 is a diagram that simplifies and summarizes the steps required to achieve the meeting service's objective. A user achieves the first step, while playing the role of a requestor, when they send a request. The send request button triggers an HTTP call to GLRSC-System-1 back-end which matches the call to a specified route called 'send_request'. In the request call, the system creates a Meeting Request object.

The second step the system takes is to define the buffer geometry polygon, in the Meeting Request object, representing the search area or neighborhood. The system's geospatial functions process and convert the requestor's location into a geometry polygon and then store the Meeting Request in the database along with its buffer as an attribute.

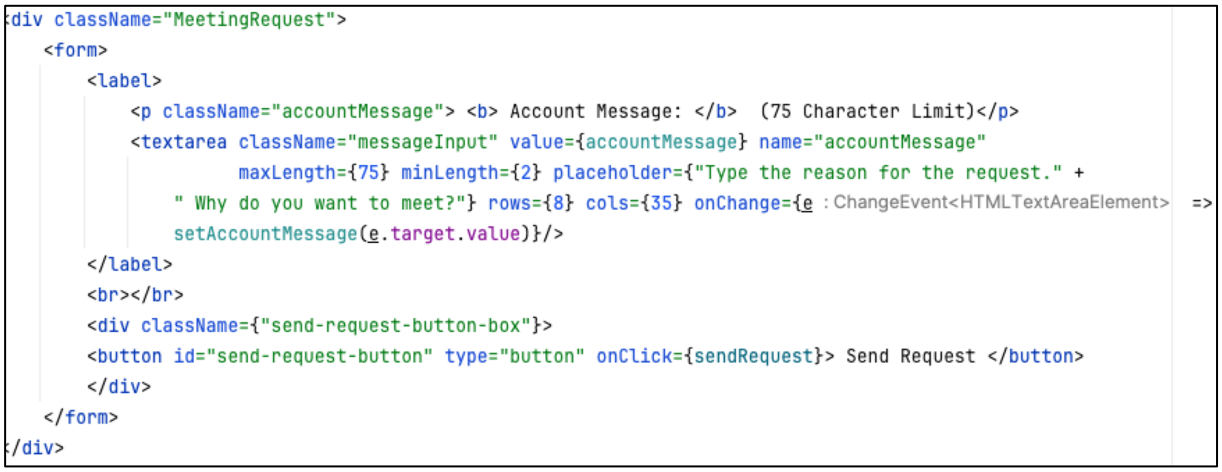
The third step in the system is to retrieve a list of potential participants; it is achieved by running a geospatial function facilitated by PostGIS. In the function, the code checks for accounts whose location lies within the buffer geometry polygon. Those accounts that are within the buffer are then referenced in a table called Account_Request, which is a table for potential participants.

The fourth step in the system requires a separate account, a Responder, who chooses to access meeting requests by clicking an interface button called, "refresh requests" which sends an HTTP call to the system's back-end which matches the call to a specified route called, "/refresh_requests". The route queries the back-end across the Account table and the Meeting_Request table to produce an object that has attributes from both tables. The system returns the custom object to the Responder's front-end engine. The front-end engine then renders the custom object as a record in a table of meeting requests. Each meeting request on the React front-end interface provides important key details of a meeting such as the name of the person who sent it, their age, their sex, and the custom message that sets the theme for the meeting. Each meeting request also allows the Responder to decline or accept the request. In the case of a decline, the Responder's information is simply deleted from Account_Request table. However, in the case that the Responder accepts the request, the system stores the Responder's account information in a new record in the Active_Meeting table which indicates that the Responder is now an active participant in the meeting.

Step six processes all account references stored in the Active_Meeting table. The system uses the accounts' locations and third-party python libraries to calculate an optimal meeting place along the road network. After the system is finished determining the midpoint, the system sends the information back to each registered account referenced in Active_Meeting. The process is done synchronously so that each participant receives a notification of the meeting place latitude and longitude coordinates at the same time. Section 4.3 reviews each of the six key steps in detail and documents important code blocks.

4.3.1 Meeting Request Functionality

To position the meeting request form on the user interface, I first created a folder called Meetings under the src folder in the React repository. I then created a file called MeetingRequest.js in the Meetings folder. I exported the React component and included it within the App.js file within the div with the className of "column3-box1". Within MeetingRequest.js, I defined an HTML div that determines the component's appearance and responsiveness with attributes such as the width of the text box and the limitation of characters. I also defined the MeetingRequest.js component to make an axios call to the route "send_request". In the configuration dictionary, I defined a variable named message and programmed the HTML text box to set the message variable as the user's text in the text box. The div from Figure 23 sets up the functionality for the MeetingRequest.js component.



```
div className="MeetingRequest">
   <form>
       <label>
           <p className="accountMessage"> <b> Account Message: </b>  (75 Character Limit)</p>
           <textarea className="messageInput" value={accountMessage} name="accountMessage"
                     maxLength={75} minLength={2} placeholder={"Type the reason for the request." +
               " Why do you want to meet?"} rows={8} cols={35} onChange={e : ChangeEvent<HTMLTextAreaElement>  =>
               setAccountMessage(e.target.value)}/>
       </label>
       <br></br>
       <div className={"send-request-button-box"}>
       <button id="send-request-button" type="button" onClick={sendRequest}> Send Request </button>
       </div>
   </form>
/div>
```

Figure 23. Message request HTML form

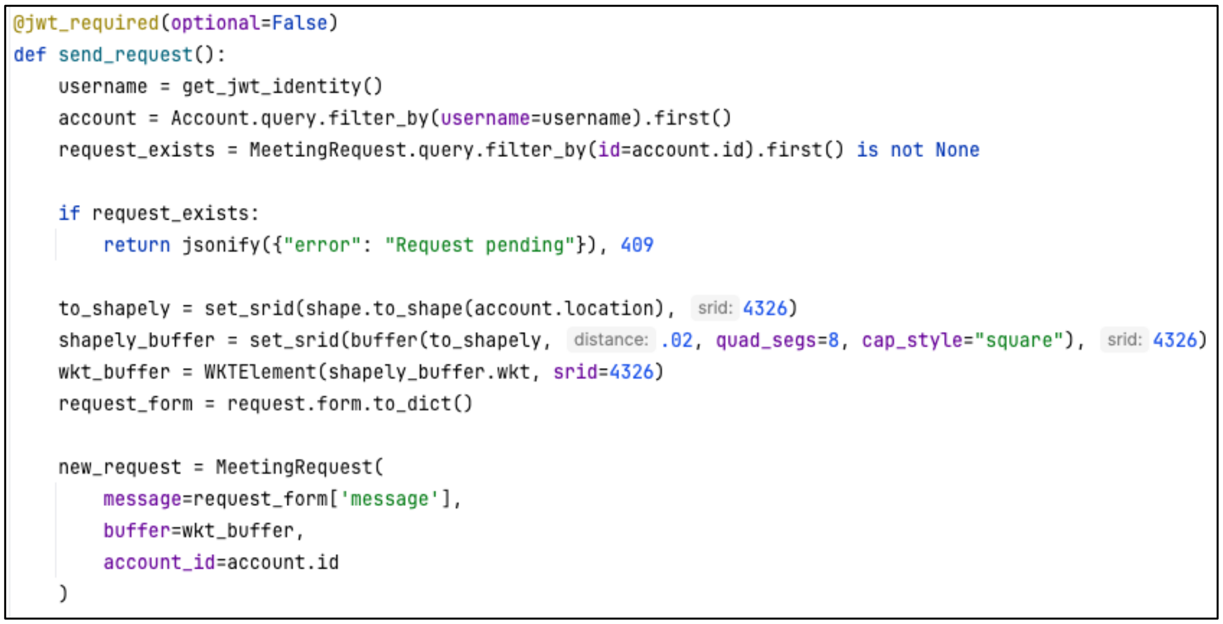
Figure 23 defines how the form is displayed with tags such as maxLength, minLength, number of rows, columns, and placeholder text. Within the textarea tag, there is a property called onChange that sets a variable called message to whatever is typed inside the text area. The HTML includes a button element which is tied to an onClick event. The onClick event triggers an axios function called sendRequest in the React file.

The variable message is used to define the topic of the meeting. For instance, a user can type, “Let’s plant seeds” or “Cookout! Bring your own goods”. Once the user clicks the send request button, the system sends the message and registers a Meeting Request object via an axios call using the format of Figure 19.

Lastly, to enable the meeting request feature, I created a new redux variable named updateRequest using the methods from 4.1.7. In the axios request, I updated another redux variable known as ucMessages with a message of, “Wait 15 min for user response”. The system displays the message in a component known as User Console which is like Geoprocessing Engine, but it has a separate position in the HTML grid layout. User Console is placed in “column2-box1” and delivers instructions and a midpoint, consisting of longitude and latitude coordinates, to the user. Furthermore, the Meeting Request div renders onto the interface when a redux variable called isLocated changes to true using the conditional rendering methods from 4.1.8. The user changes the redux variable isLocated to true when they click on the location button in account details.

4.3.2 Spatial Buffer

On the back-end, the system calls to a route named send_request. The first step that the system takes is to create a buffer search area from the requestor’s location. Figure 24 displays the code that creates the buffer location.



```python
@jwt_required(optional=False)
def send_request():
    username = get_jwt_identity()
    account = Account.query.filter_by(username=username).first()
    request_exists = MeetingRequest.query.filter_by(id=account.id).first() is not None

    if request_exists:
        return jsonify({"error": "Request pending"}), 409

    to_shapely = set_srid(shape.to_shape(account.location), srid: 4326)
    shapely_buffer = set_srid(buffer(to_shapely, distance: .02, quad_segs=8, cap_style="square"), srid: 4326)
    wkt_buffer = WKTElement(shapely_buffer.wkt, srid=4326)
    request_form = request.form.to_dict()

    new_request = MeetingRequest(
        message=request_form['message'],
        buffer=wkt_buffer,
        account_id=account.id
    )
```

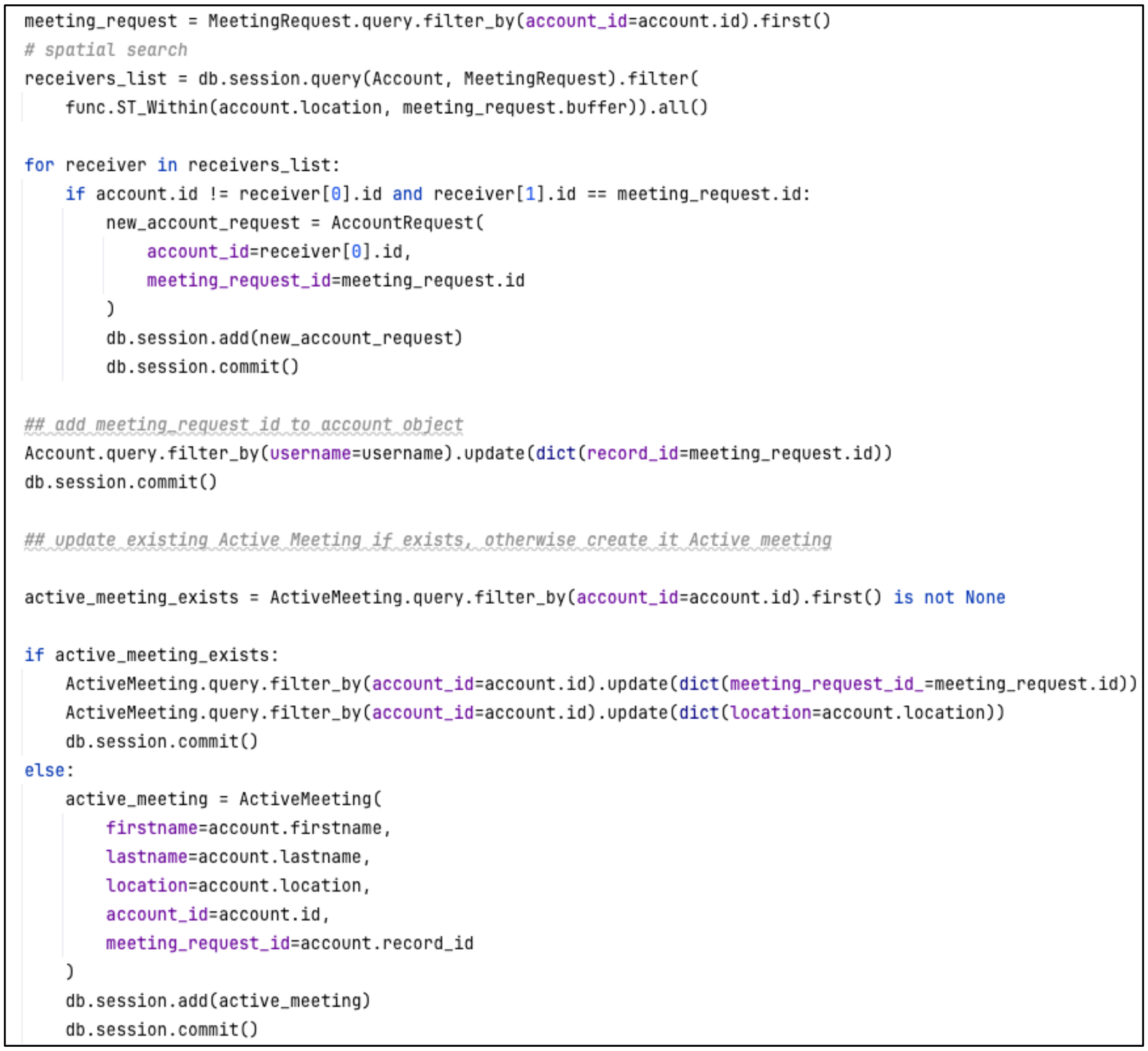
Figure 24. Code buffer search area

The code describes the series of steps to create and store the buffer search area. First, the program acquires the requestor’s account. Then it creates a variable called to_shapely. The to_shapely variable utilizes the shapely python library to convert the requestor’s location into a readable shapely point. Whenever a user clicks the location button on the user interface, their location is stored in the database as a WKTElement, a data format consisting of a string of encrypted characters. Therefore, when a user queries GLRSC-System-1, it returns WKTElement format which is converted into a more useful format for geospatial processing. Shapely converts the WKTElement into a Shapely geometry point that holds readable latitude and longitude coordinates. The program uses shapely once more to convert the shapely point into a shapely buffer. The parameters for the buffer function are distance, quad_segs, cap_style, and srid. The distance has a value of .02 in degrees. The quad_segs divides the buffer area into eight sections then restitches them together. The cap_style is the shape of the buffer which is set to square. The srid is set to 4326 which corresponds to the WGS 84 spatial reference system which uses degrees not meters. The program then converts the shapely buffer back into WKTElement so that it can be read and processed by PostgreSQL. The program sets the buffer as the Meeting Request object. Then the script adds the Meeting Request object to the database.

For reference, the parameters used to create the buffer are not optimal nor were they the parameters that were planned during pre-project planning. During testing, the results were observed in PgAdmin, a software for administering PostgreSQL databases. The original parameters were attempted but found to be suboptimal. For instance, the original plan was to use a cap_style of circle, but when using a srid of 4326, PgAdmin renders the buffer as an elongated oval. Different srids were tested but PgAdmin did not successfully render a map for any srid except for 4326. Lastly, because 4326 uses degrees not meters as a unit of measurement, there was no option but to use degrees as a unit of measurement. A degree measurement of .02 provides a buffer with a radius of roughly 2000 meters, but inaccurate measurements are expected. More research and work should be conducted to optimize rendering and to determine a methodology where meters can be used as a distance unit, and a circle can be used as the buffer shape in place of a square.

4.3.3 Spatial Search to Potential Participants' List

Step three begins in the same function, send_request, as step two. After the buffer has been successfully stored, the system creates a list of potential partners and store them in a table named Account_Request. All accounts whose locations lie within the buffer area have their information stored in Account_Request. Furthermore, the system stores the requestor information in a separate data table named Active_Meeting. Both Account_Request and Active_Meeting tables are defined in models.py and are created upon starting the application. The Active_Meeting table stores account information from those accounts who are confirmed active participants in the meeting. Figure 25 displays the script used to achieve the actions described.



```
meeting_request = MeetingRequest.query.filter_by(account_id=account.id).first()
# spatial search
receivers_list = db.session.query(Account, MeetingRequest).filter(
    func.ST_Within(account.location, meeting_request.buffer)).all()

for receiver in receivers_list:
    if account.id != receiver[0].id and receiver[1].id == meeting_request.id:
        new_account_request = AccountRequest(
            account_id=receiver[0].id,
            meeting_request_id=meeting_request.id
        )
        db.session.add(new_account_request)
        db.session.commit()

## add meeting_request id to account object
Account.query.filter_by(username=username).update(dict(record_id=meeting_request.id))
db.session.commit()

## update existing Active Meeting if exists, otherwise create it Active meeting

active_meeting_exists = ActiveMeeting.query.filter_by(account_id=account.id).first() is not None

if active_meeting_exists:
    ActiveMeeting.query.filter_by(account_id=account.id).update(dict(meeting_request_id_=meeting_request.id))
    ActiveMeeting.query.filter_by(account_id=account.id).update(dict(location=account.location))
    db.session.commit()
else:
    active_meeting = ActiveMeeting(
        firstname=account.firstname,
        lastname=account.lastname,
        location=account.location,
        account_id=account.id,
        meeting_request_id=account.record_id
    )
    db.session.add(active_meeting)
    db.session.commit()
```

Figure 25. Python code for creating a list of potential participants

The script first acquires the meeting request that is tied to the requestor's identification number. A variable named receivers_list runs a database query using a special PostGIS function, ST_Within. The ST_Within function takes two parameters, the first is a spatial object of any type, and the second is also a spatial object. The function checks if the first argument is within the second object. If the first object is within the second, the function returns true. Otherwise, it returns false. The database query uses the ST_Within as a filter to return only those accounts that lie within the spatial object defined in the second argument. The result is a list of accounts that lie within the buffer area.

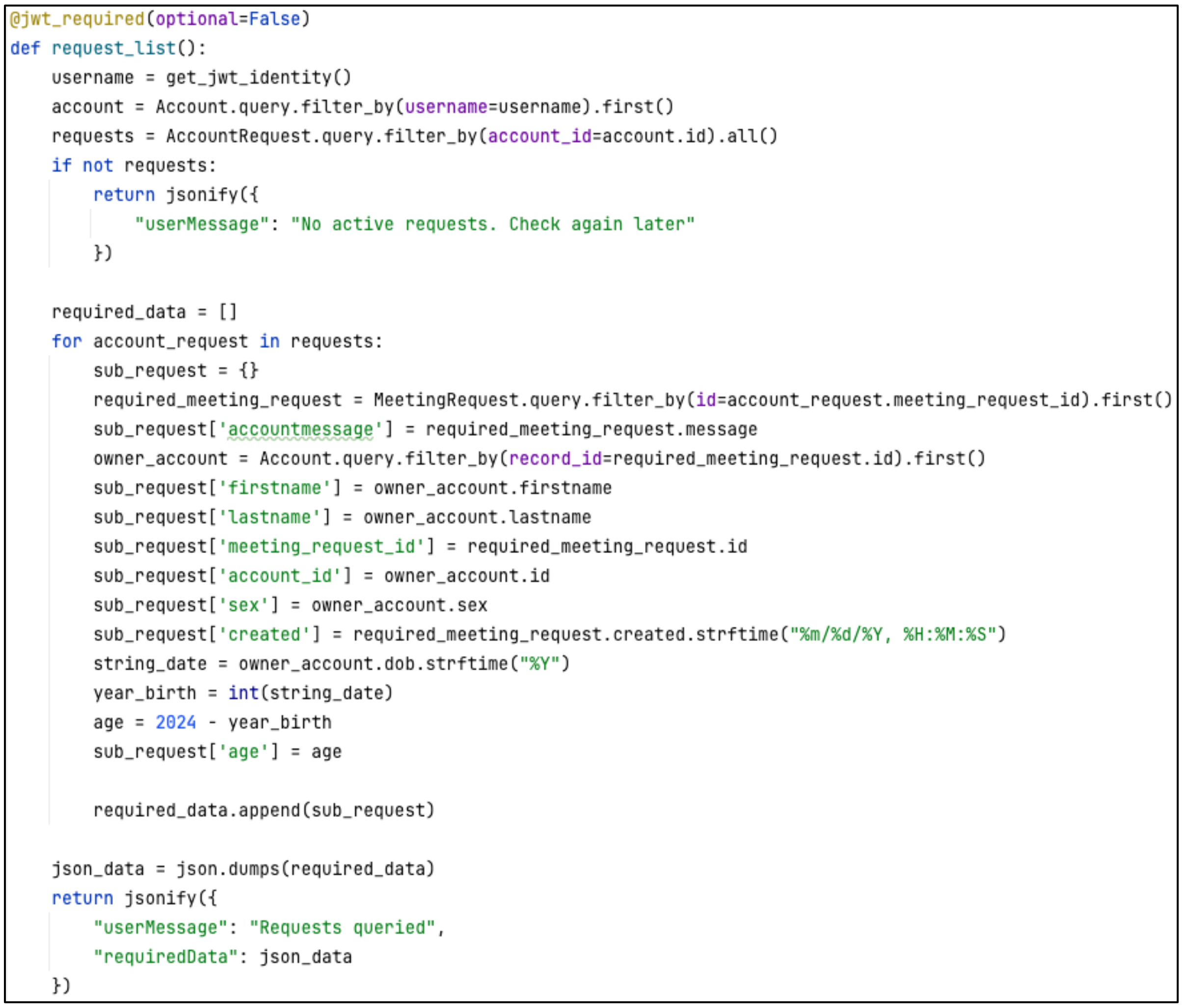
The script then defines a for loop which loops through each element in the list and locates the identification numbers for each account in the list. Using the account identification numbers, the script creates new Account_Request objects for each identification number and adds the objects to the corresponding data table, Account_Request. This action ensures that the list of potential participants is set in record and can be accessed in further processes. Further down in Figure 25, another object called active meeting is added to the database. The active meeting data table is a record of participants which are confirmed as active participants. Each account has only one active meeting object associated with it. The logic behind this decision is that one user can only attend one meeting at a time. The active meeting object is updated when a user sends or accepts a request to register the new meeting information. The log behind this decision is that if a user is sending a request, they are expected be one of the active participants.

Once the send request function is complete it returns a status of success to the requestor. A useful feature that is unique to React programming is that the HMTL on a website or web app is constantly being rerendered after reading any component file. Therefore, App.js is constantly being re-read and reprocessed as HTML on the interface. This is beneficial because after the user sends a request, the program can conditionally remove the meeting request box immediately. Afterall, the user can only attend one meeting at a time. Therefore, after a user sends a request, the request form is hidden. In its place a timer of fifteen minutes is rendered in “column3-box1”. The React application hides one component and shows another through conditional rendering, redux variables, and React re-rendering. When the React application re-renders it notices that the redux global variable requestSent has been modified to true. The conditional rendering brackets for the MeetingRequest.js box ensures that when requestSent is set to true, the system renders a component called MR_TimerDelete.js instead of the MeetingRequest.js component. The MR_TimerDelete.js file runs procedures according to instructions by Adhikary (2022).

4.3.4 Refresh Requests

While the requestor waits for fifteen minutes so that other users may respond to their request, the responder(s) is given fifteen minutes to respond to incoming requests. Apart from the timer that the system displays for the requestor, there are also timers that the system displays for the responder. The front-end system displays each individual request as a record in a request table on the user interface. Each record holds key details. For instance, within a request, a timer function is one of the columns in a request record. The initial value for the timer is fifteen minutes but countdowns toward zero per second as one second passes from the time that the request was registered in the PostgreSQL database. The timers in the meeting request records on the user interface use MR_TimerDelete.js as a template. Basically, they are the same code based off Adhikary's tutorial (2022). However, before a responder can accept, decline a request, or see the timers, they must first acquire the requests.

GLRSC-System-1's interface provides a component called MeetingListHeader.js placed in "column2-box3". The MeetingListHeader.js component contains a refresh_requests button that runs an axios call to a route named '/requests' in the Flask back-end. Figure 26 displays the function that runs in the Flask back-end.



```python
@jwt_required(optional=False)
def request_list():
    username = get_jwt_identity()
    account = Account.query.filter_by(username=username).first()
    requests = AccountRequest.query.filter_by(account_id=account.id).all()
    if not requests:
        return jsonify({
            "userMessage": "No active requests. Check again later"
        })

    required_data = []
    for account_request in requests:
        sub_request = {}
        required_meeting_request = MeetingRequest.query.filter_by(id=account_request.meeting_request_id).first()
        sub_request['accountmessage'] = required_meeting_request.message
        owner_account = Account.query.filter_by(record_id=required_meeting_request.id).first()
        sub_request['firstname'] = owner_account.firstname
        sub_request['lastname'] = owner_account.lastname
        sub_request['meeting_request_id'] = required_meeting_request.id
        sub_request['account_id'] = owner_account.id
        sub_request['sex'] = owner_account.sex
        sub_request['created'] = required_meeting_request.created.strftime("%m/%d/%Y, %H:%M:%S")
        string_date = owner_account.dob.strftime("%Y")
        year_birth = int(string_date)
        age = 2024 - year_birth
        sub_request['age'] = age

        required_data.append(sub_request)

    json_data = json.dumps(required_data)
    return jsonify({
        "userMessage": "Requests queried",
        "requiredData": json_data
    })
```

Figure 26. Python code to retrieve list of meeting requests

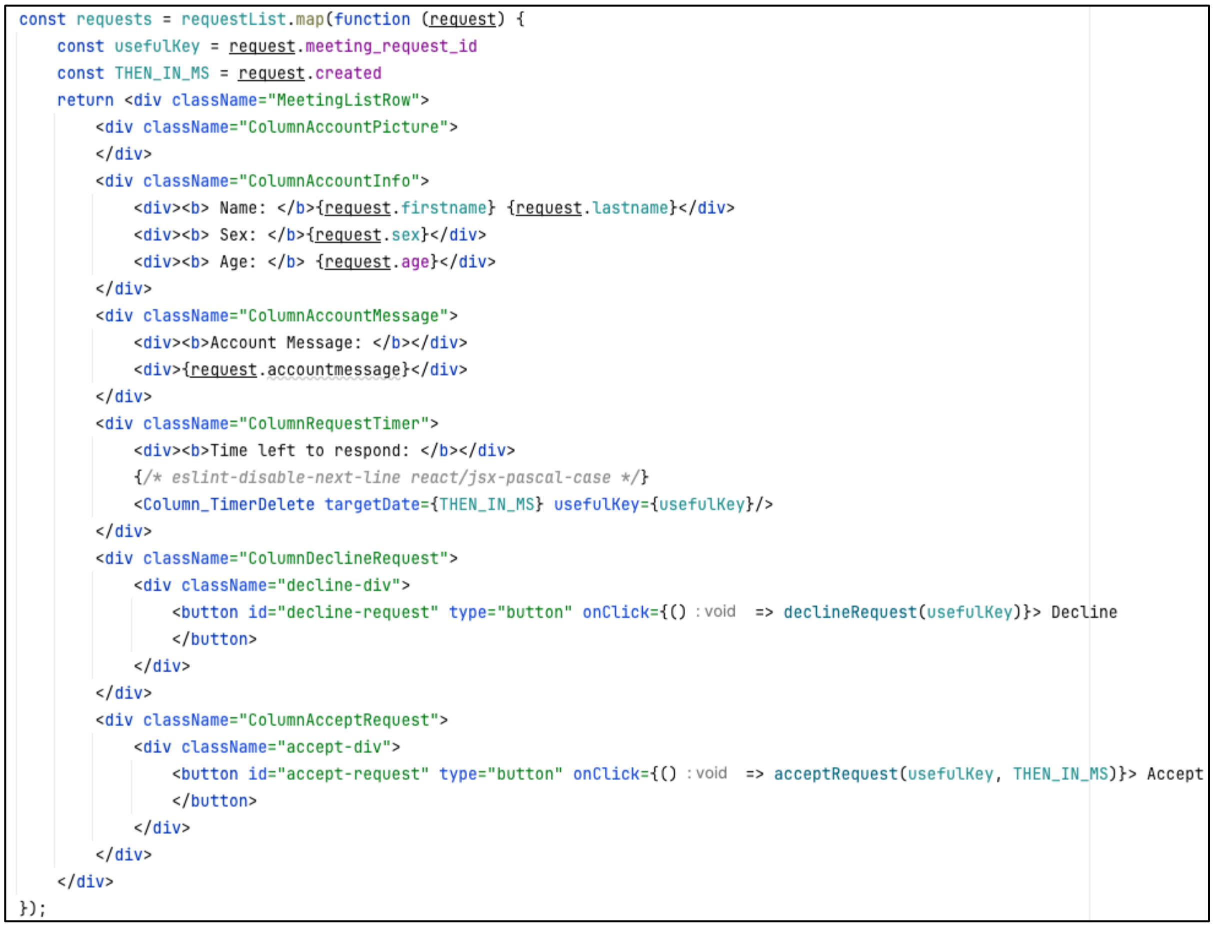
At the start of the script above, the Python code runs a query to acquire all the Account_Request rows where the responder's identification number is present. The requests object in the Python script holds the connection between responders and meeting requests. If there are no requests present, there has been no requestor in a 2km radius that has sent a request, meaning there are no active requests in the responder's geographic area. Otherwise, if there are active requests in the area, the script runs a for loop on the requests object. The for loop builds custom request objects for every record in the requests object. Important information exists in the Meeting Request and Account table. Therefore, GLRSC-System-1 navigates through all tables, extracting important information and building a custom interface request that is both visually appealing and functional within the React interface.

Figure 26 demonstrates how the system creates custom interface requests. For each record in the requests object, the system queries and stores the corresponding meeting request and account. Using the information from meeting request and account tables, the system builds a custom request with the following properties: first name, last name, meeting request id, account id, user's sex, custom user message, created timestamp, and the user's age. The system stores each custom request object within a list called required data. The system converts the required data into json format and returns the required data to the front-end system for presentation.

4.3.5 Request React Component with Decline and Accept Options

The React system receives the required data as a response within the axios function called refresh_requests withing the MeetingListHeader.js component. The function checks if there is valid data in the response. If there is valid data, the system stores the custom request objects in a redux global variable called requestList. The MeetingListHeader.js component provides an HTML layout that has one parent div with a className of Meeting List and two children divs with className of MeetingListHeader and MeetingListSub. The system renders requestList in MeetingListSub by using a variable named "requests" that maps through the "requestList" and customizes its content into usable HTML.

Figure 27 displays how the system defines the requests variable and shows the unique properties that enable accept and delete functionality.



```
const requests = requestList.map(function (request) {
    const usefulKey = request.meeting_request_id
    const THEN_IN_MS = request.created
    return <div className="MeetingListRow">
        <div className="ColumnAccountPicture">
        </div>
        <div className="ColumnAccountInfo">
            <div><b> Name: </b>{request.firstname} {request.lastname}</div>
            <div><b> Sex: </b>{request.sex}</div>
            <div><b> Age: </b> {request.age}</div>
        </div>
        <div className="ColumnAccountMessage">
            <div><b>Account Message: </b></div>
            <div>{request.accountmessage}</div>
        </div>
        <div className="ColumnRequestTimer">
            <div><b>Time left to respond: </b></div>
            {/* eslint-disable-next-line react/jsx-pascal-case */}
            <Column_TimerDelete targetDate={THEN_IN_MS} usefulKey={usefulKey}/>
        </div>
        <div className="ColumnDeclineRequest">
            <div className="decline-div">
                <button id="decline-request" type="button" onClick={() :void  => declineRequest(usefulKey)}> Decline
                </button>
            </div>
        </div>
        <div className="ColumnAcceptRequest">
            <div className="accept-div">
                <button id="accept-request" type="button" onClick={() :void  => acceptRequest(usefulKey, THEN_IN_MS)}> Accept
                </button>
            </div>
        </div>
    </div>
});
```

Figure 27. JavaScript variable that displays requests

The “requests” variable is one with embedded HTML. The system creates the variable by looping through the redux variable “requestList” which holds the custom request objects. For each custom request in the requestList, the system declares two important variables. One of these is the usefulKey which holds the unique meeting request id. The other variable is called THEN_IN_MS which holds the created timestamp from the meeting request. Both variables are necessary for JavaScript processes within the MeetingListRow div. The child div with a className of ColumnAccountInfo displays the requestor’s name, sex, and age. The child div with a className of ColumnAccountMessage displays the custom request message. The child div with a className of ColumnRequestTimer displays a component that countdowns the time left to respond to the request. The timer sets the countdown in minutes and seconds according to the difference from when the request was created to when the responder retrieves the request. The timer always sets to fifteen minutes or less.

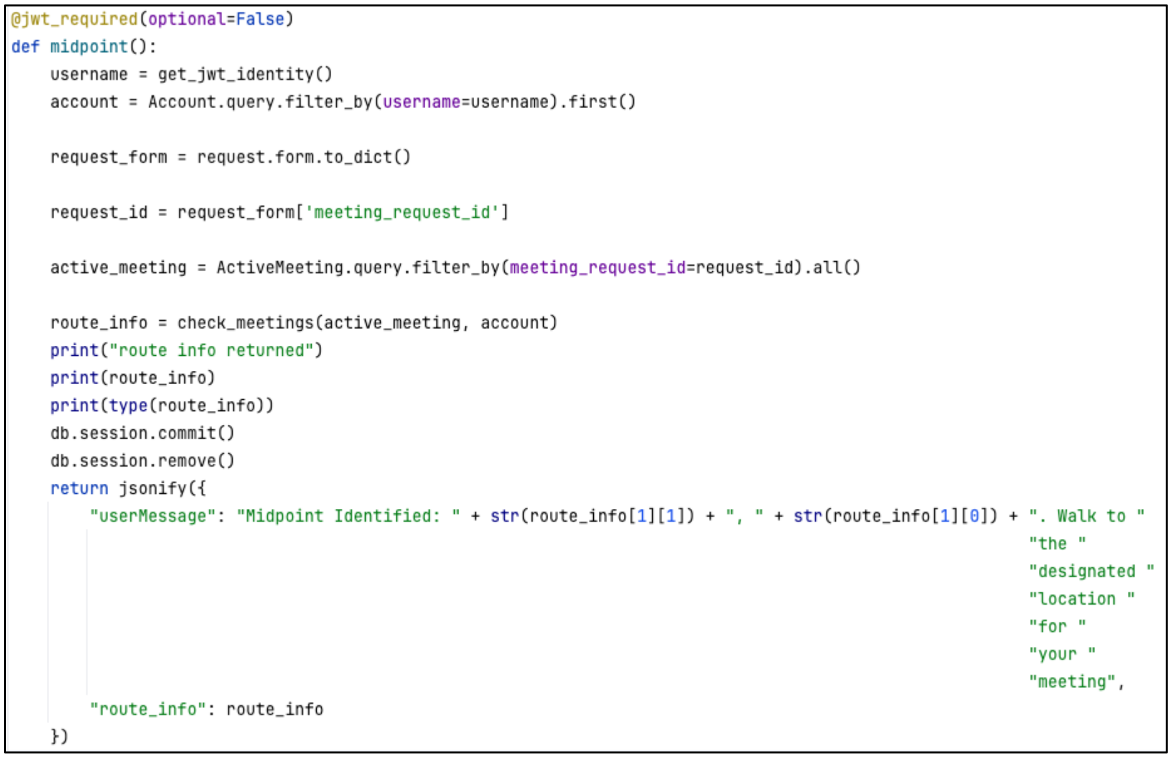
The child div with a className of ColumnDeclineRequest provides a button to decline a request. It provides an onClick function called declineRequest which takes the usefulKey as an argument. The child div with a className of ColumnAcceptRequest provides a button to accept a request. It provides an onClick function called acceptRequest which takes the usefulKey and THEN_IN_MS variables as arguments.

The declineRequest button triggers an axios call to the route named '/declineRequest' in the Flask back-end. The script in the Flask back-end takes the usefulKey to filter for the Account_Request row that the user belongs to and deletes the record. In other words, the script removes the user from the list of potential participants.

The acceptRequest button triggers two events. First, it sends an axios call to the route named '/acceptRequest', which transfers' the responders account information to the ActiveMeeting table. In other words, the acceptRequest button confirms the user as an active participant. The second event is an update on the UserConsole.js component. A redux variable named ucMessages updates to include an instruction to wait an allotted time for more participants to join the meeting. A timer is added to the ucMessages, using MR_TimerDelete.js as a template, which then appears in the UserConsole.js component. The new timer begins where the old timer left off. In total, all users responding to a particular request have fifteen minutes to respond from the time the request was stored in the PostgreSQL database.

4.3.6 Process and Return Midpoint Information

The requestor's timer and the responder's timer both serve a critical function in the last step of the meeting service. Both timers are synchronized in a countdown of fifteen minutes from the creation of the meeting request. Once the fifteen minutes are up, both timers, on both ends of the system, trigger an axios call to routes that return midpoint information. The responder's axios calls a route named '/getMidpoint' on the Flask back-end while the requestor's axios calls a route named '/getOwnerMidpoint'. These functions are responsible for calculating the midpoint between the responder or requestor and all other accounts in the Active_Meeting table. The'/getMidpoint' and '/getOwnerMidpoint' functions are divided into three-parts. This section first reviews the responder's '/getMidpoint' route then the requestor's '/getOwnerMidpoint' route. Figure 28 shows the first part of '/getMidpoint'.

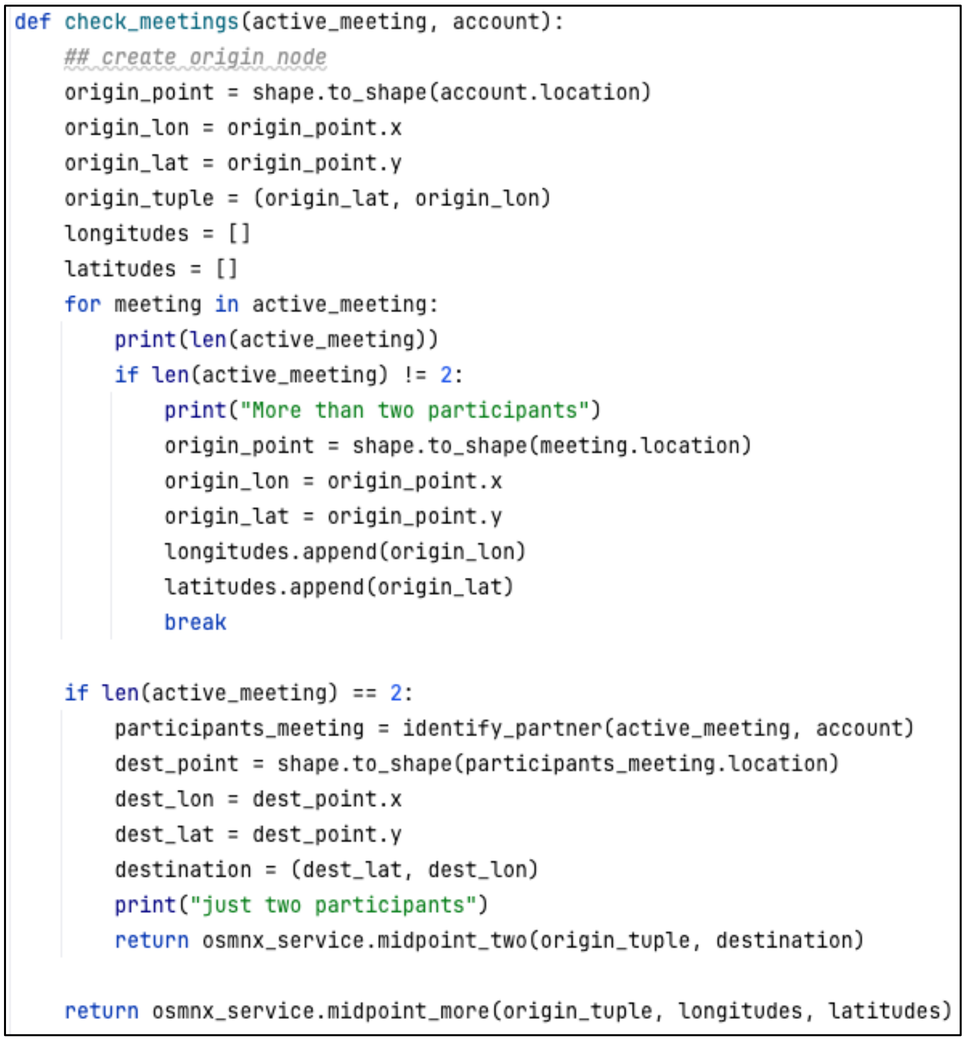


```python
@jwt_required(optional=False)
def midpoint():
    username = get_jwt_identity()
    account = Account.query.filter_by(username=username).first()

    request_form = request.form.to_dict()

    request_id = request_form['meeting_request_id']

    active_meeting = ActiveMeeting.query.filter_by(meeting_request_id=request_id).all()

    route_info = check_meetings(active_meeting, account)
    print("route info returned")
    print(route_info)
    print(type(route_info))
    db.session.commit()
    db.session.remove()
    return jsonify({
        "userMessage": "Midpoint Identified: " + str(route_info[1][1]) + ", " + str(route_info[1][0]) + ". Walk to "
                                                                                                         "the "
                                                                                                         "designated "
                                                                                                         "location "
                                                                                                         "for "
                                                                                                         "your "
                                                                                                         "meeting",
        "route_info": route_info
    })
```

Figure 28. Get midpoint starting function

The midpoint calculation begins by first querying for the account object in the data table. A script declares a request id which is set to the usefulKey described in 4.3.5. The usefulKey is the meeting request id. To obtain the information from Active_Meeting, the list of active participants, the application must search for an attribute that all records shares. In this case, the meeting request id is stored in every row and is, therefore, the shared attribute. Next, a variable named active_meeting stores the return list of the database query. A variable called route is designated to store the responder's coordinates and the midpoint coordinates.

The route variable is set to a function called check_meetings and is passed an argument of the active_meeting and the responder's account. The function check_meetings processes the active participants and the responder's account and returns routing information which includes the midpoint latitude and longitude coordinates. Figure 29 shows the second part of the process which is the functionality within check_meetings.

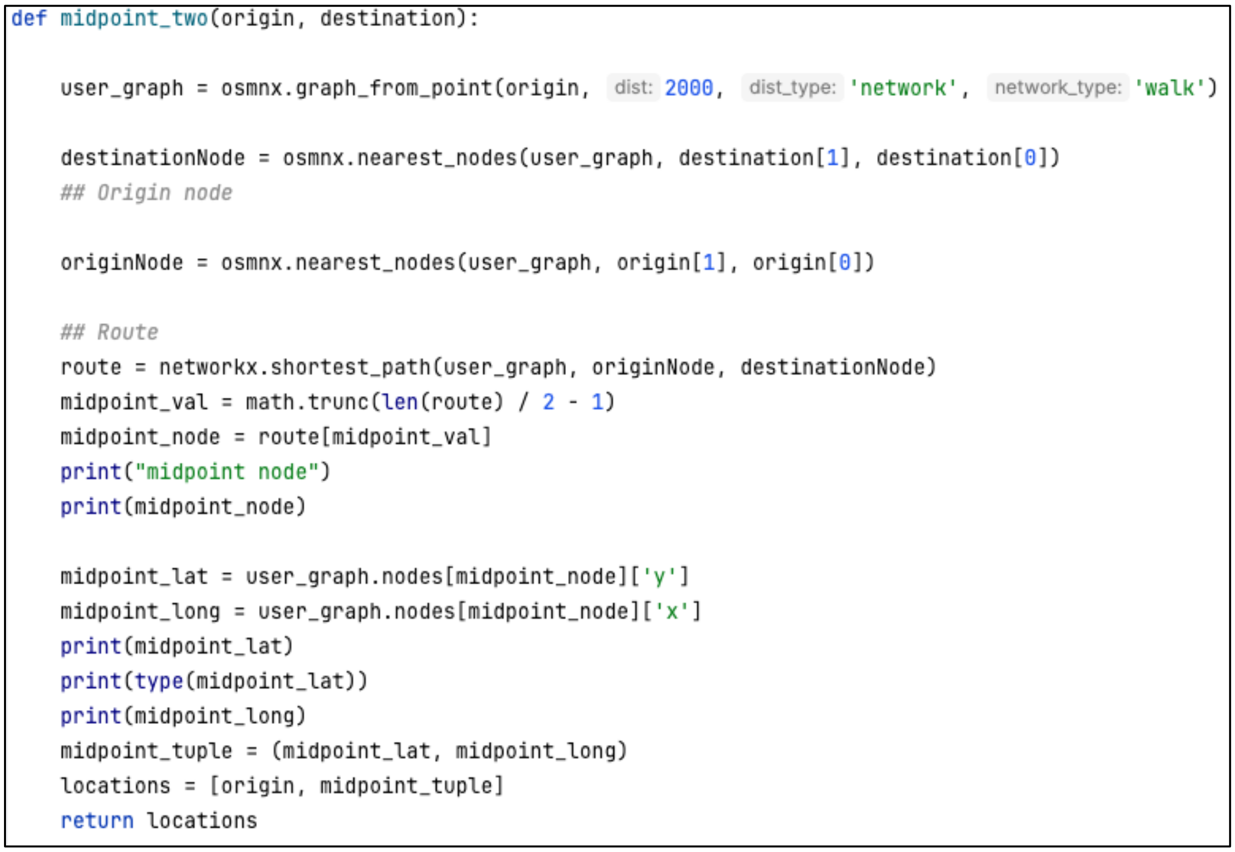


```python
def check_meetings(active_meeting, account):
    ## create origin node
    origin_point = shape.to_shape(account.location)
    origin_lon = origin_point.x
    origin_lat = origin_point.y
    origin_tuple = (origin_lat, origin_lon)
    longitudes = []
    latitudes = []
    for meeting in active_meeting:
        print(len(active_meeting))
        if len(active_meeting) != 2:
            print("More than two participants")
            origin_point = shape.to_shape(meeting.location)
            origin_lon = origin_point.x
            origin_lat = origin_point.y
            longitudes.append(origin_lon)
            latitudes.append(origin_lat)
            break

    if len(active_meeting) == 2:
        participants_meeting = identify_partner(active_meeting, account)
        dest_point = shape.to_shape(participants_meeting.location)
        dest_lon = dest_point.x
        dest_lat = dest_point.y
        destination = (dest_lat, dest_lon)
        print("just two participants")
        return osmnx_service.midpoint_two(origin_tuple, destination)

    return osmnx_service.midpoint_more(origin_tuple, longitudes, latitudes)
```

Figure 29. Get midpoint secondary function

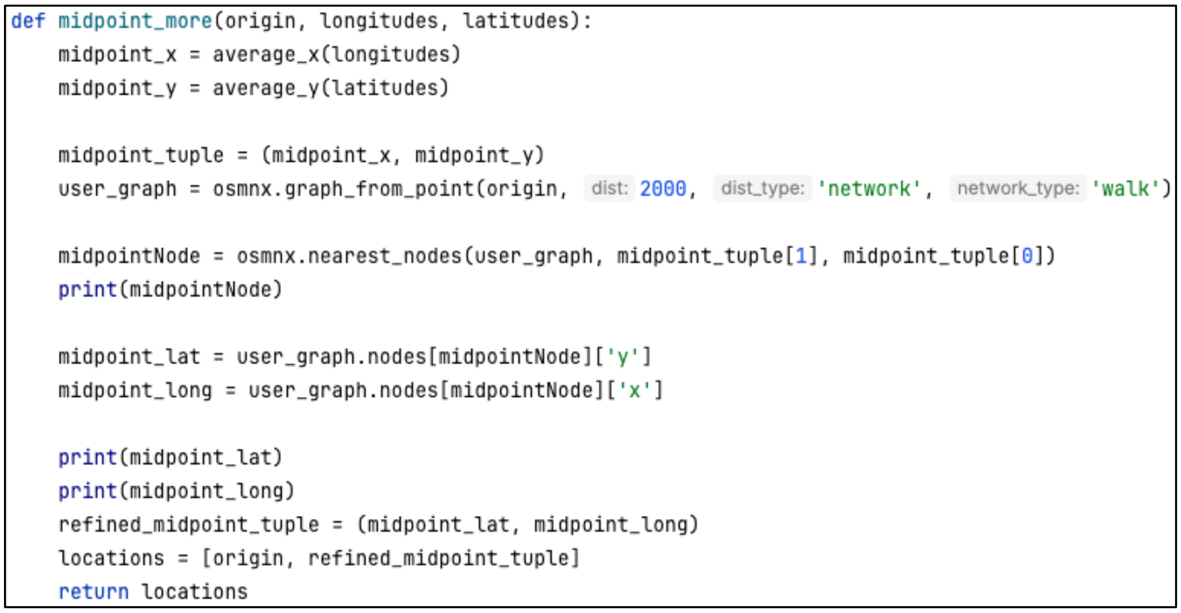
The function begins by defining the origin tuple. The origin tuple consists of the responder's latitude and longitude coordinates. The function then checks whether there are two or more active participants in the active_meeting list object. If there are more than two participants, the tuple latitude and longitude are appended to separate lists containing only longitude coordinates or only latitude coordinates. Then the script calls a function called midpoint_more which calculates a midpoint using a geographic mean for more than two participants.

However, if there are only two participants the script calls on a separate function. First, the script identifies the singular partner that joins the user in the meeting. The script then defines a destination variable that consists of the partner's latitude and longitude coordinates. The check_meetings function ends with a return call to a function called midpoint_two which calls external python packages to compute an optimal midpoint. Figure 30 displays the last part of the get midpoint process.


```
def midpoint_two(origin, destination):

    user_graph = osmnx.graph_from_point(origin,  dist: 2000,  dist_type: 'network',  network_type: 'walk')

    destinationNode = osmnx.nearest_nodes(user_graph, destination[1], destination[0])
    ## Origin node

    originNode = osmnx.nearest_nodes(user_graph, origin[1], origin[0])

    ## Route
    route = networkx.shortest_path(user_graph, originNode, destinationNode)
    midpoint_val = math.trunc(len(route) / 2 - 1)
    midpoint_node = route[midpoint_val]
    print("midpoint node")
    print(midpoint_node)

    midpoint_lat = user_graph.nodes[midpoint_node]['y']
    midpoint_long = user_graph.nodes[midpoint_node]['x']
    print(midpoint_lat)
    print(type(midpoint_lat))
    print(midpoint_long)
    midpoint_tuple = (midpoint_lat, midpoint_long)
    locations = [origin, midpoint_tuple]
    return locations
```
Figure 30. Tertiary midpoint function for two participants

In this function, the system makes calls to the OSMNX python package (Boeing 2017). The first call to OSMNX creates a graph centered on the responder's location with a radius buffer of 2km. The graph is defined as a street network that includes all edges where walking is permitted. Using the graph output and the locations of the responder and requestor, the script determines the nearest nodes from each user on the street network and stores them in separate variables. With the two nodes, the script can then call on another graph-based python package called NetworkX, and its function called shortest_path (Hagber et al. 2008). The shortest path function finds the shortest path between the responder's location and the requestor's location. When the path has been determined, the program evaluates it and extracts the middle node on the path. A path output is a series of edges and nodes in a singular non-circular line. Therefore, the nodes can be counted, and a median can be found. The middle node on the path is the midpoint for the meeting. However, it must also be converted to a format that is usable for the users. Therefore, the script converts the middle node into latitude and longitude coordinates. The script returns the responder's location and the middle node's latitude and longitude to the check_meetings function which then returns the coordinates to the get_midpoint function and then back to the React front-end interface.

Alternatively, the check_meetings function identifies more than two participants and calls on the midpoint_more function. Figure 31 displays the tertiary midpoint function for more than two participants.



```
def midpoint_more(origin, longitudes, latitudes):
    midpoint_x = average_x(longitudes)
    midpoint_y = average_y(latitudes)

    midpoint_tuple = (midpoint_x, midpoint_y)
    user_graph = osmnx.graph_from_point(origin, dist: 2000, dist_type: 'network', network_type: 'walk')

    midpointNode = osmnx.nearest_nodes(user_graph, midpoint_tuple[1], midpoint_tuple[0])
    print(midpointNode)

    midpoint_lat = user_graph.nodes[midpointNode]['y']
    midpoint_long = user_graph.nodes[midpointNode]['x']

    print(midpoint_lat)
    print(midpoint_long)
    refined_midpoint_tuple = (midpoint_lat, midpoint_long)
    locations = [origin, refined_midpoint_tuple]
    return locations
```

Figure 31. Tertiary midpoint function for more than two participants

The midpoint_more function takes the responder's latitude and longitude coordinates as well as two lists: one list of the participants' latitude coordinates and another list of the participants' longitude coordinates. The script calls on two functions to determine the average longitude and the average latitude. The average longitude is stored in the midpoint_x variable. The average latitude is stored in the midpoint_y variable. The program defines a midpoint tuple with the average longitude and the average latitude.

The average longitude and latitude can be used as the midpoint coordinates, but they need further processing because the midpoint coordinates could be positioned in a geographic space where public meetings cannot occur. For instance, the midpoint coordinates could be in a lake, on someone's property, or at an inaccessible location. To ensure that the midpoint coordinates can be used for a public meeting, the script calls on OSMNX to calculate a street network with the same arguments as midpoint_two. The program then uses the nearest_node function to find the nearest street intersection, a node on the street network, from the midpoint coordinates. The script then extracts the longitude and latitude coordinates from the nearest node. The new coordinates are assigned to a new variable called refined_midpoint_tuple which is returned to the React user interface.

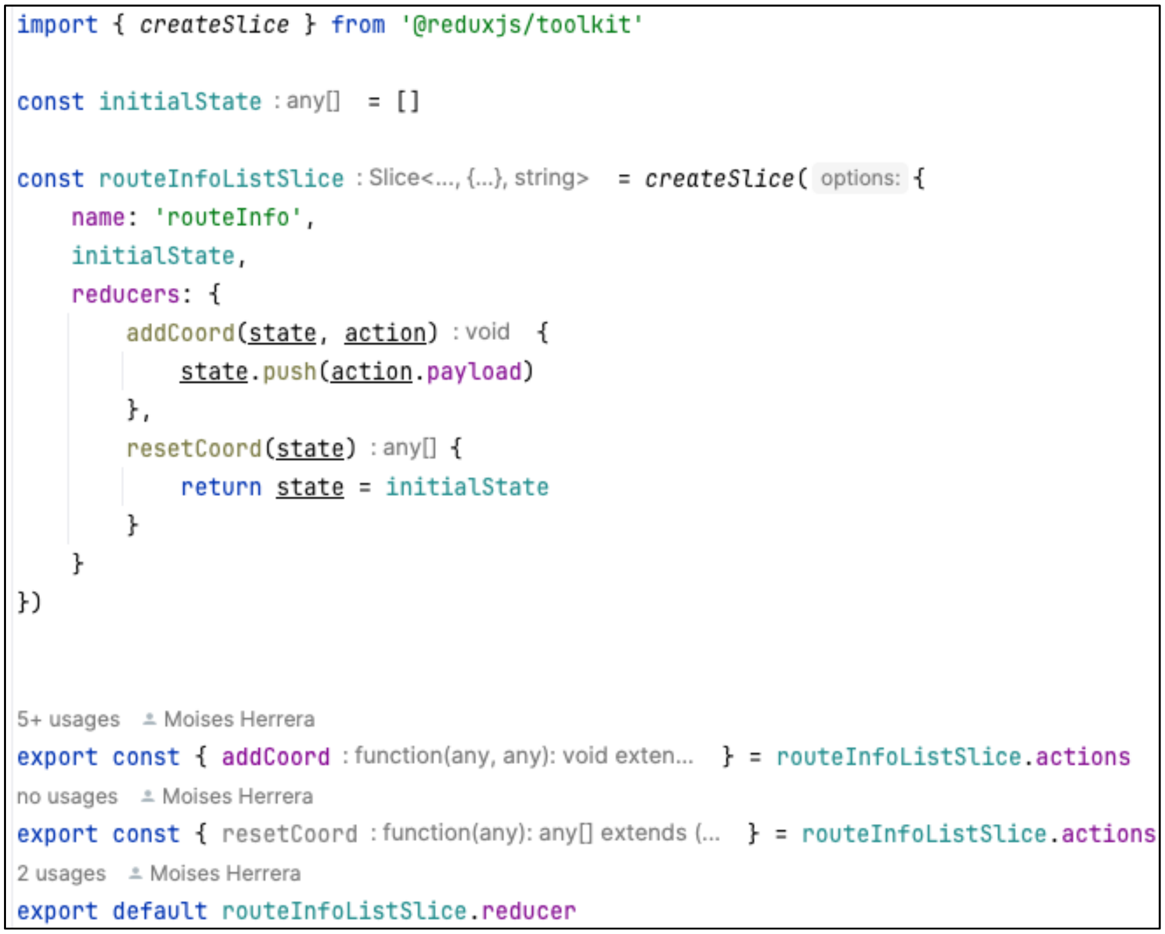
The requestor's '/getOwnerMidpoint' route works in a similar way in that is also calls on the secondary and tertiary functions described in Figure 30 and Figure 31. However, its primary function has additional steps that are only applicable to the requestor's account. If the list of active participants only contains the requestor's information, after 15 minutes have passed, the system sends a message to the React interface that no users have registered for the meeting. It also deletes the requestor's record in the active_meeting table. However, if there are active participants, the React system displays the message, "Walk to the designated location for your meeting. Arrive within 15 minutes", on both the requestor's and responder's UserConsole.js component. The React system also prints the midpoint coordinates in the UserConsole.js component.

4.4 Phase 4: Route Visualization

The objective for Phase 4 is to render the returned information from Phase 3 as a visual route within "column2-box2" of the user interface. The React system assigns "column2-box2" as the HTML element to display Midpoint_Router.js component. The steps to complete the phase are as follows: use a redux variable to prepare the route information, apply conditional rendering for Midpoint_Router.js, create a Mapbox profile and access token, program the Midpoint_Router.js component.

4.4.1 Redux to Prepare the Route Information

Once the route information arrives to the front-end, it must be stored in a secure redux variable to be used in a different React component. For instance, the route information arrives through one of two components, the UC_TimerDelete.js or the MR_TimerDelete.js. Both components contain timer functions which trigger a call to retrieve the route information once fifteen minutes pass. However, once the back-end system returns route information, the data is needed in a separate component called Midpoint_Router.js distance from UC_TimerDelete.js or MR_TimerDelete.js. To pass the information to Midpoint_Router.js, I used redux for storing the global route information and created a file named routeInfoListSlice.js within the redux folder. I also used the redux Store.js which was created in 4.1.7. Store.js stores all the redux variables in a system while routeInfoListSlice.js specifies the variable specific to the routing information and defines the functions that can be applied to the variable. Figure 32 shows routeInfoListSlice.js.



```
import { createSlice } from '@reduxjs/toolkit'

const initialState : any[]  = []

const routeInfoListSlice : Slice<..., {...}, string>  = createSlice( options: {
    name: 'routeInfo',
    initialState,
    reducers: {
        addCoord(state, action) : void  {
            state.push(action.payload)
        },
        resetCoord(state) : any[] {
            return state = initialState
        }
    }
})



5+ usages  Moises Herrera
export const { addCoord : function(any, any): void exten...  } = routeInfoListSlice.actions
no usages  Moises Herrera
export const { resetCoord : function(any): any[] extends (...  } = routeInfoListSlice.actions
2 usages  Moises Herrera
export default routeInfoListSlice.reducer
```

Figure 32. routeInfoListSlice.js

The image above depicts the Python code for routeInfoListSlice.js. It defines an initial state of the variable which in this case is an array of numbers. The array is the data structure where the system stores the coordinates that are needed to display the route. The file also contains a name property which helps the system recognize the variable throughout the code base. After that, the file defines functions that are applied to the variable. There are add and reset functions which enables the system to add coordinates to the array, or reset the array to its original default state, as needed. Lastly, the file exports the routeInfo property and the objects using the last three statements in the file. Store.js is a separate file that contains all the redux variables created throughout the development process. Figure 33 shows store.js contents at the end of the development process.

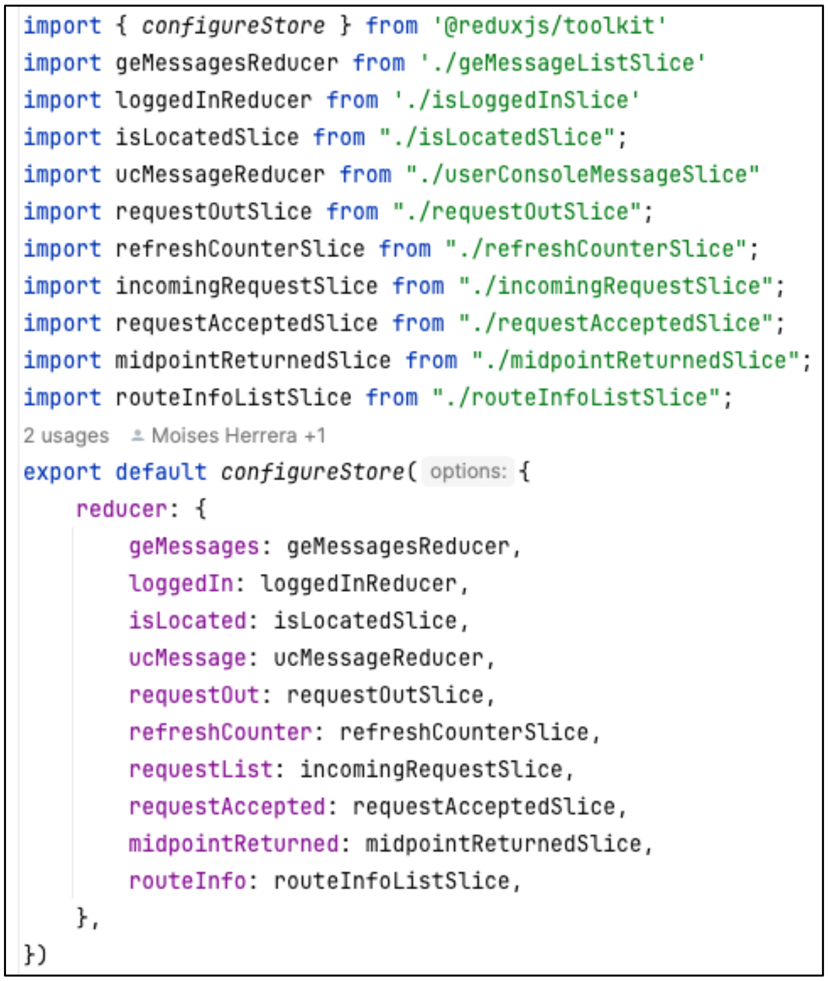

```
import { configureStore } from '@reduxjs/toolkit'
import geMessagesReducer from './geMessageListSlice'
import loggedInReducer from './isLoggedInSlice'
import isLocatedSlice from "./isLocatedSlice";
import ucMessageReducer from "./userConsoleMessageSlice"
import requestOutSlice from "./requestOutSlice";
import refreshCounterSlice from "./refreshCounterSlice";
import incomingRequestSlice from "./incomingRequestSlice";
import requestAcceptedSlice from "./requestAcceptedSlice";
import midpointReturnedSlice from "./midpointReturnedSlice";
import routeInfoListSlice from "./routeInfoListSlice";
2 usages   Moises Herrera +1
export default configureStore( options: {
    reducer: {
        geMessages: geMessagesReducer,
        loggedIn: loggedInReducer,
        isLocated: isLocatedSlice,
        ucMessage: ucMessageReducer,
        requestOut: requestOutSlice,
        refreshCounter: refreshCounterSlice,
        requestList: incomingRequestSlice,
        requestAccepted: requestAcceptedSlice,
        midpointReturned: midpointReturnedSlice,
        routeInfo: routeInfoListSlice,
    },
})
```

Figure 33. Redux store

The Python code above defines a redux store where all the redux variables, used in the system, are stored, and made available to the entire codebase. Here, you can see all the variables that have made the system functional: geMessages stores the messages displayed in the Geoprocessing Engine, loggedIn enables conditional rendering, isLocated enables conditional rendering, ucMessages stores the messages displayed on the User Console, requestOut enables conditional rendering, refreshCounter enables conditional rendering, requestList stores the incoming requests on the Meeting List component, requestAccepted enables conditional rendering, midpointReturned enables conditional rendering, and finally routeInfo shares the origin and destination to Midpoint_Router.js for route visualization.

4.4.2 Conditional Rendering for the Midpoint_Router.js Component

The system conditionally renders the Midpoint_Router.js component. The Midpoint_Router.js is paired with two redux variables that must be set to true before the component can appear on screen. The two redux variables are loggedIn and midpointReturned. In short, the user must be logged in and must have gone through the process of accepting a request or sending a request that was accepted by another user. The midpointReturned variable becomes true when the Flask back-end returns route information to either UC_TimerDelete.js or MR_TimerDelete.js and has been stored in the routeInfo redux variable.

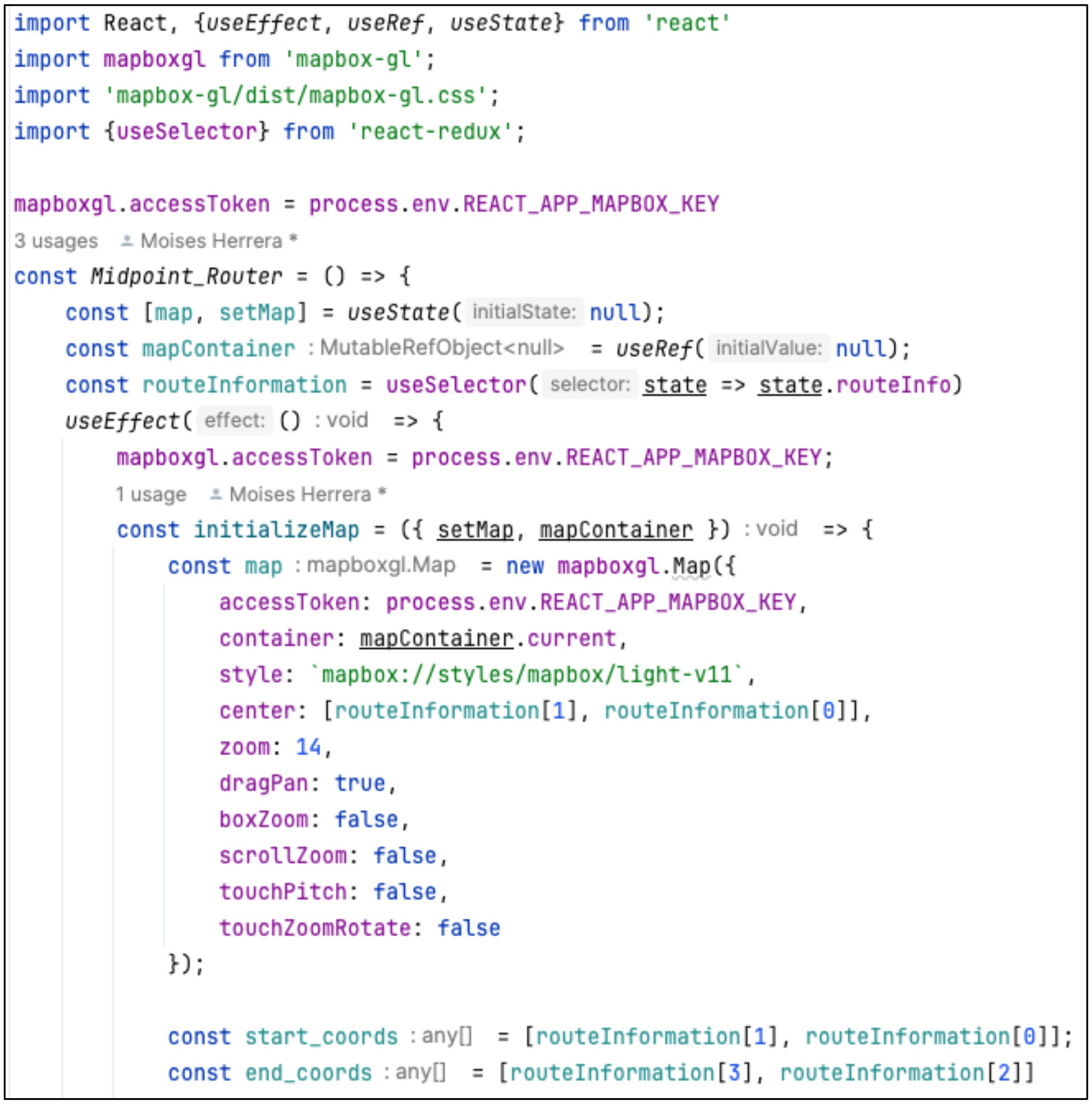
4.4.3 Mapbox Profile and an Access Token

I chose Mapbox software to render the route. I programmed the Midpoint_Router.js component to use Mapbox's API functions. Before Mapbox was used, I first created a Mapbox account and created an access token using the button create access token on the log in landing page. The access token allowed me to use Mapbox resource through HTTP requests. Once I made the access token, I created a variable named REACT_APP_MAPBOX_KEY in the .env file of the React repository's root level. The access key from the Mapbox page is to be copied and pasted as the value for REACT_APP_MAPBOX_KEY. Any variable in the .env file is secure and private while still being accessible to the codebase. The Midpoint_Router.js calls on the .env file to access the access token. Using a variable such as REACT_APP_MAPBOX_KEY from the .env file is like using a local storage global variable or a redux variable.

In Midpoint_Router.js, I programmed the JavaScript script to use Mapbox functions that take as arguments the origin and destination coordinates in routeInfo. The Mapbox function runs as soon as the system stores the route information in the redux variable. Once the Mapbox function sends the information to Mapbox, Mapbox responds with a route from the origin point to the destination point. I used Mapbox functions in the Midpoint_Router.js component to determine the route, the basemap, the zoom level, the control features, etc.

4.4.4 Midpoint_Router.js Component

I programmed all the code that produces the route visualization in the Midpoint_Router.js component. It contains several important operations such as importing Mapbox, initializing the map, acquiring the route, adding layers to the map, and displaying the map in an HTML div. Figure 34 displays the initial operations as read from top to bottom.

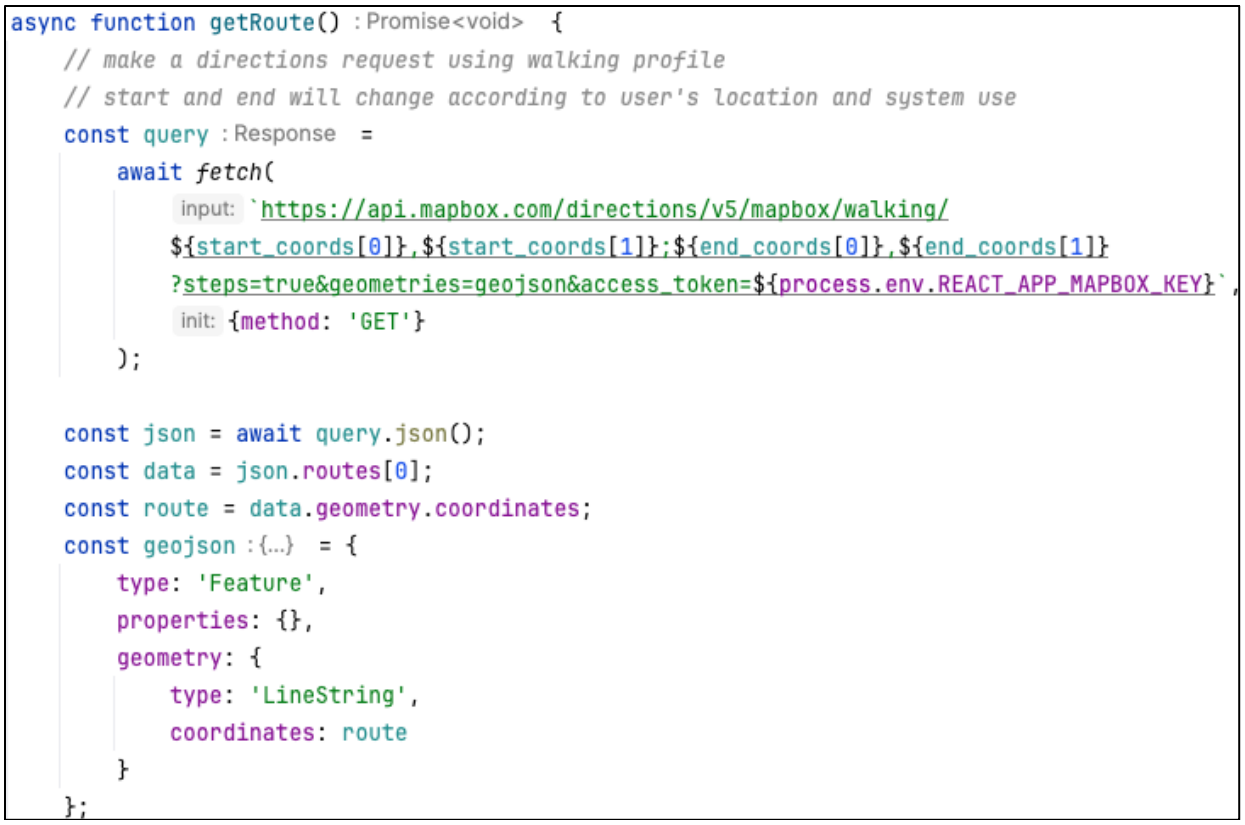


```
import React, {useEffect, useRef, useState} from 'react'
import mapboxgl from 'mapbox-gl';
import 'mapbox-gl/dist/mapbox-gl.css';
import {useSelector} from 'react-redux';

mapboxgl.accessToken = process.env.REACT_APP_MAPBOX_KEY
3 usages  Moises Herrera *
const Midpoint_Router = () => {
    const [map, setMap] = useState( initialState: null);
    const mapContainer : MutableRefObject<null>  = useRef( initialValue: null);
    const routeInformation = useSelector( selector: state => state.routeInfo)
    useEffect( effect: () : void  => {
        mapboxgl.accessToken = process.env.REACT_APP_MAPBOX_KEY;
        1 usage  Moises Herrera *
        const initializeMap = ({ setMap, mapContainer }) : void  => {
            const map : mapboxgl.Map  = new mapboxgl.Map({
                accessToken: process.env.REACT_APP_MAPBOX_KEY,
                container: mapContainer.current,
                style: `mapbox://styles/mapbox/light-v11`,
                center: [routeInformation[1], routeInformation[0]],
                zoom: 14,
                dragPan: true,
                boxZoom: false,
                scrollZoom: false,
                touchPitch: false,
                touchZoomRotate: false
            });

            const start_coords : any[]  = [routeInformation[1], routeInformation[0]];
            const end_coords : any[]  = [routeInformation[3], routeInformation[2]]
```

Figure 34. Initialize Map on Midpoint_Router.js

The script begins with import statements that allow the system to use Mapbox and Redux. The script sets the Mapbox access token to the .env variable, REACT_APP_MAPBOX_KEY, discussed in the previous section. Once the access token has been set the script declares its main function called Midpoint Router. The main function uses a React feature called useState which allows a file to create a variable and a function to determine the value of the variable. The file uses useState to set the initial value of “map” to null. The map variable displays the route. The script declares a variable named routeInformation and sets it to the redux variable routeInfo, which has the user’s coordinates and the midpoint coordinates.

The file uses a React feature called useEffect which tells the system to execute the following commands after rendering the component. The initializeMap function is set inside useEffect, which tells the system to render the map once the browser reads and displays the file. The initializeMap function defines a Mapbox map and sets the behavior of the map as well as its container which is defined as an HTML div in Midpoint_Router.js. Finally, in the last lines of code in Figure 34, the script extracts the user’s coordinates and the midpoint coordinates from the routeInformation variable and stores them in two variables, start_coords and end_coords. Figure 35 shows how the system sends an API request to Mapbox using the user’s coordinates and the midpoint coordinates.

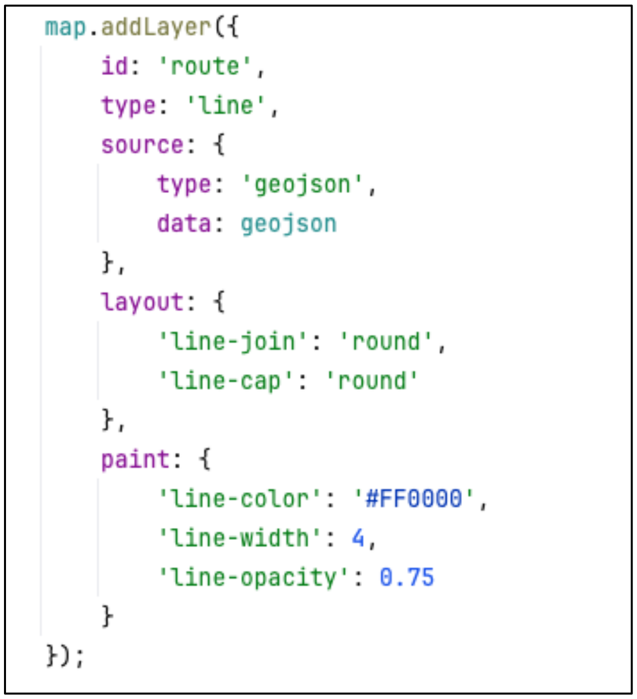


```
async function getRoute() : Promise<void>  {
    // make a directions request using walking profile
    // start and end will change according to user's location and system use
    const query : Response  =
        await fetch(
            input: `https://api.mapbox.com/directions/v5/mapbox/walking/
            ${start_coords[0]},${start_coords[1]};${end_coords[0]},${end_coords[1]}
            ?steps=true&geometries=geojson&access_token=${process.env.REACT_APP_MAPBOX_KEY}`,
            init: {method: 'GET'}
        );

    const json = await query.json();
    const data = json.routes[0];
    const route = data.geometry.coordinates;
    const geojson : {...}  = {
        type: 'Feature',
        properties: {},
        geometry: {
            type: 'LineString',
            coordinates: route
        }
    };
```

Figure 35. Mapbox API route request

In Figure 35, the script defines a function, called getRoute, which is used to get the route. The function getRoute executes later in the script. The function getRoute makes an asynchronous call to Mapbox's API. The API call takes origin coordinates, destination coordinates, as well as a profile string. The script uses start_coords for the origin coordinates and end_coords for the destination coordinates. For the profile string, the script uses the string "walking" which tells Mapbox that the route should be generated for pedestrian use. The system gives preference to routes optimized for walking subjects because it fits the scope of the thesis project and is a great fit for the system objective. The final parameter in the API request is the access token that was initially created in Mapbox and then stored as a .env variable. The response for the API request is a large data object that holds the necessary route information including path edges and nodes.

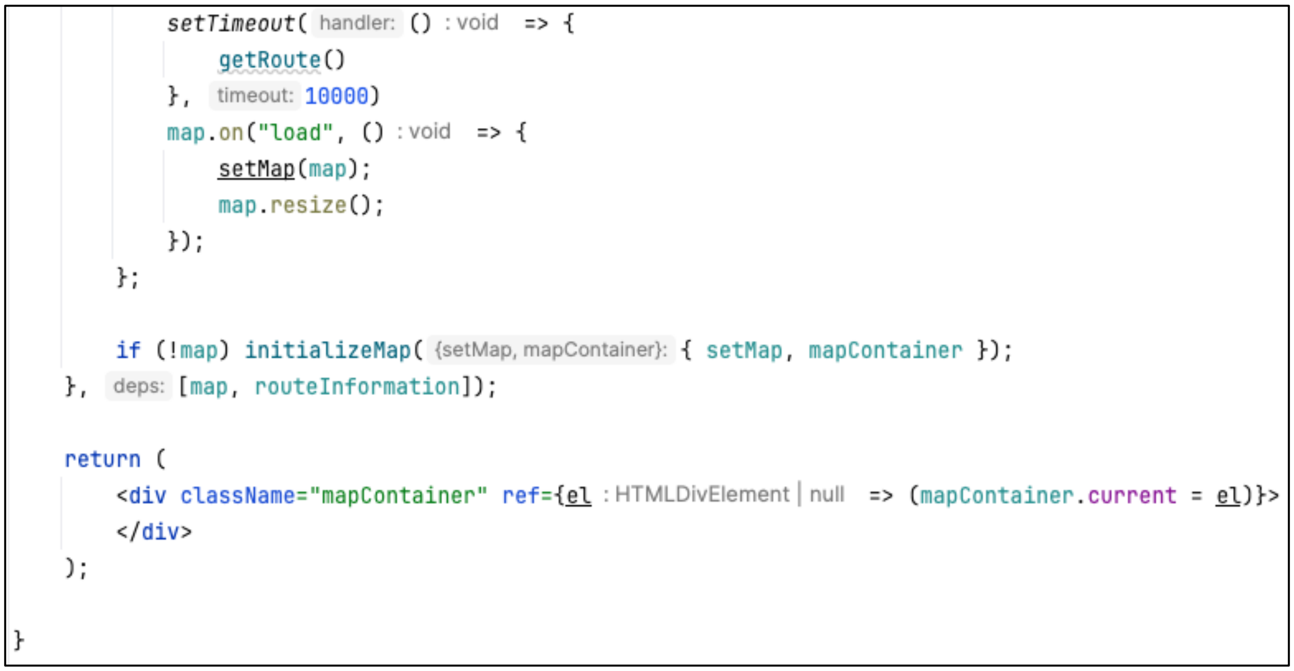
The next lines of code clean the data by first converting it into a legible json data format and then extracting only the needed information. The script defines the geojson variable which holds the necessary information to visualize the route. Figure 36 shows the next lines of code which are responsible for adding the route to the map.



```
map.addLayer({
    id: 'route',
    type: 'line',
    source: {
        type: 'geojson',
        data: geojson
    },
    layout: {
        'line-join': 'round',
        'line-cap': 'round'
    },
    paint: {
        'line-color': '#FF0000',
        'line-width': 4,
        'line-opacity': 0.75
    }
});
```

Figure 36. Add Route to Map

In Figure 36, the script adds the geojson route layer to the map ensuring that it renders a route to the midpoint. The layout object provides the route with some styles such as a line that has a round cap. Furthermore, the paint object describes the color, width, and opacity of the line. Two similar code blocks succeed Figure 36 which are responsible for adding origin and destination points using the coordinates from the end of Figure 34. At the end of the two code blocks the getRoute function closes. Figure 37 shows the final Python code that completes the Midpoint_Router.js component and its functionality in the system.



```
            setTimeout( handler: () :void  => {
                getRoute()
            }, timeout: 10000)
            map.on("load", () :void  => {
                setMap(map);
                map.resize();
            });
        };

        if (!map) initializeMap( {setMap, mapContainer}: { setMap, mapContainer });
    }, deps: [map, routeInformation]);

    return (
        <div className="mapContainer" ref={el :HTMLDivElement | null  => (mapContainer.current = el)}>
        </div>
    );

}
```

Figure 37. Load map and container

Figure 37 shows the code that runs after map initialization. The code executes the getRoute function from Figure 35 and Figure 36. The system uses the setTimeout function to allow the getRoute function ten seconds before running. This is done so that the map loads first and then the system sets the route. If the system were to conduct both at the same time the system would crash because it would try to add layers to a map that is not yet defined.

Finally, the div with className “mapContainer” stores the map. In Figure 34, the div is set to be the map’s container. The code within the return statement ensures that the browser loads the map within a div on the HTML-driven user interface.

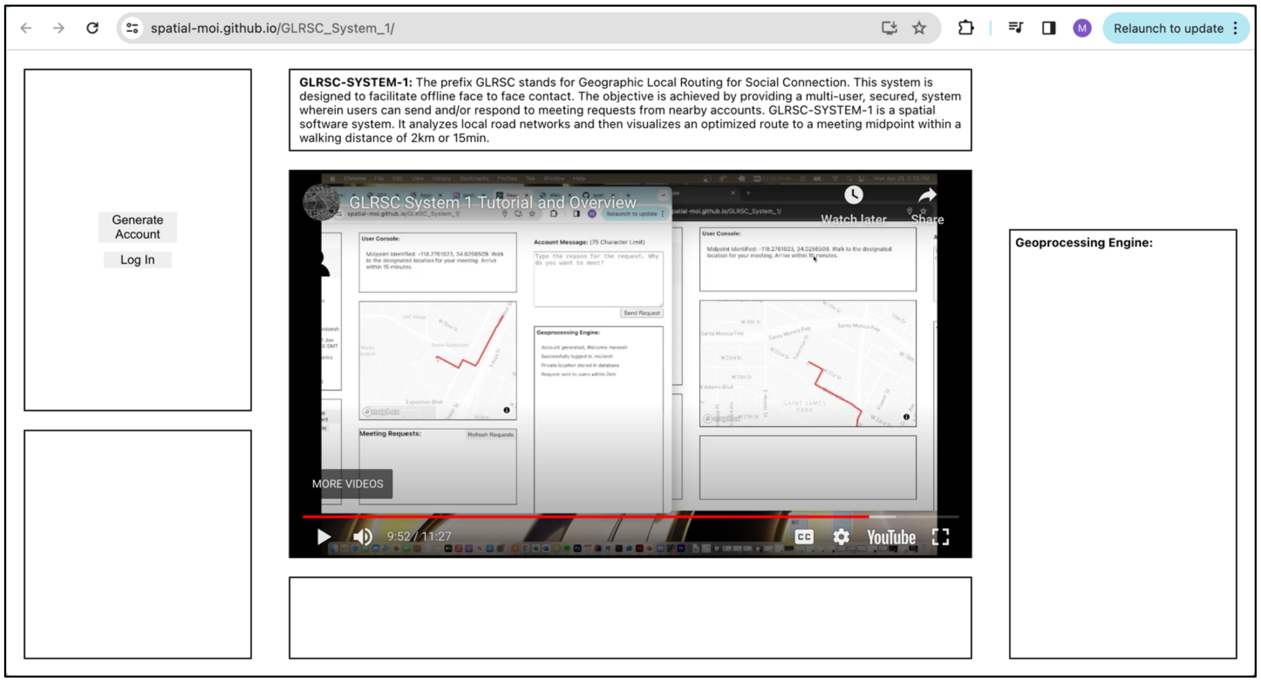
Chapter 5 Results

The results for this thesis demonstrate that GLRSC is a feasible application that can facilitate an immediate social encounter for the user. The implementation of GLRSC in GLRSC-System-1 was successful. Its success provides readers with the opportunity to visualize GLRSC in action. Furthermore, GLRSC-System-1 is fully functional and meets its objective of routing nearby peers to a midpoint along the road network for a thematic meeting. Because GLRSC-System-1 has met its objective, the methodology, described in Chapter 4 of this thesis, is a reliable guide for building a routing service and multi-user environment that implements GLRSC. GLRSC-System-1 is hosted on https://spatial-moi.github.io/GLRSC_System_1/. This chapter discusses the results divided into four sections: multi-user environment, deployment, meeting service, and routing service. All the results displayed in this chapter were gathered from the live production system proving that the system is fully functional and has met all its objectives.

5.1 Phase 1 Results

The Phase 1 objective was to create a multi-user environment where users can execute the basic CRUD functions on their accounts. The results of the Phase 1 methodology proved to be successful in creating a multi-user environment where a user can generate an account, log in, change their password, delete their account, and log out. The methodology also successfully set up a front-end system that creates a personalized space and that communicates algorithmic, and data driven processes within the system. Furthermore, the methodology proved successfully implements the system capability to store the user's location.

Figure 38 depicts the system's landing page.


spatial-moi.github.io/GLRSC_System_1/

GLRSC-SYSTEM-1: The prefix GLRSC stands for Geographic Local Routing for Social Connection. This system is designed to facilitate offline face to face contact. The objective is achieved by providing a multi-user, secured, system wherein users can send and/or respond to meeting requests from nearby accounts. GLRSC-SYSTEM-1 is a spatial software system. It analyzes local road networks and then visualizes an optimized route to a meeting midpoint within a walking distance of 2km or 15min.

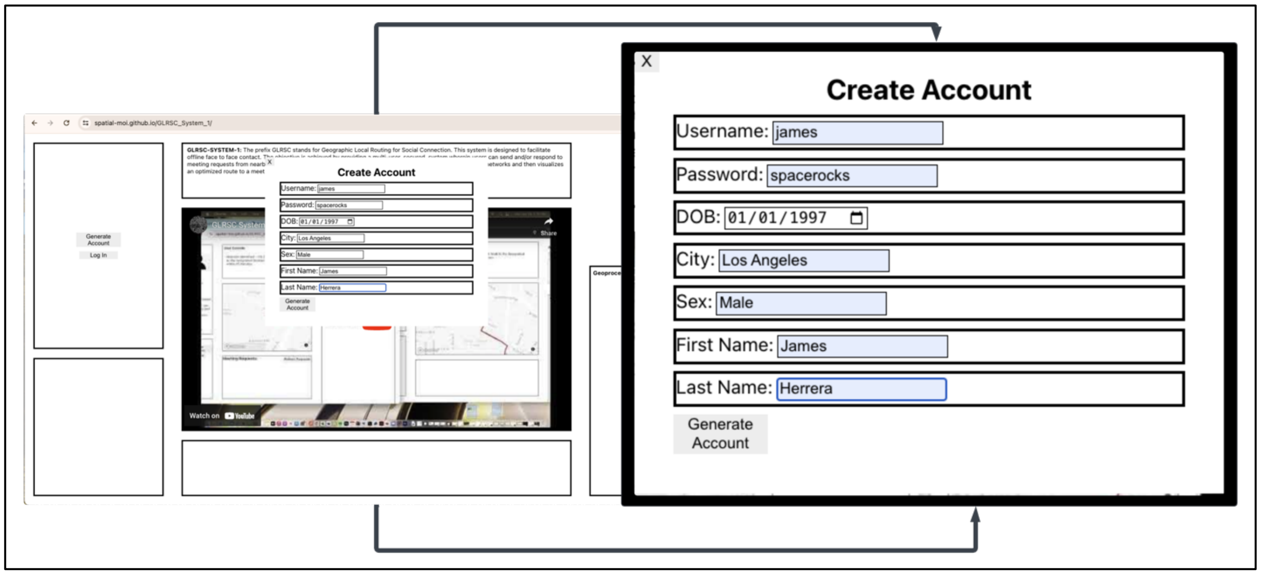
Figure 38. GLRSC-System-1 landing page

The system's landing page shows two buttons, an introductory paragraph, and tutorial. The generate account and log in buttons are essential to have on the landing page and serve as gates for the system's services. The introductory paragraph describes the system's objective, GLRSC, and the technology that enables its capabilities. The introductory paragraph is informative for a system user as well as a researcher who wants to understand the system from a scientific perspective. The video component provides a full tutorial of the system and has a duration of roughly eleven minutes; it also has educational value for visitors and researchers.

While Figure 38 shows a very minimal landing page, it also displays the organizational rules that were established in the methodology. The methodology's organizational rules enable the engineer with the foundations required to build the system. The rules also dictate the organizational structure of the code base. From Figure 38, the user sees that the system is divided into three columns. Within each column there are a few boxes. The first column has two boxes.

The second column has three boxes. The last column has two boxes. The boxes serve as containers for distinct system functionality.

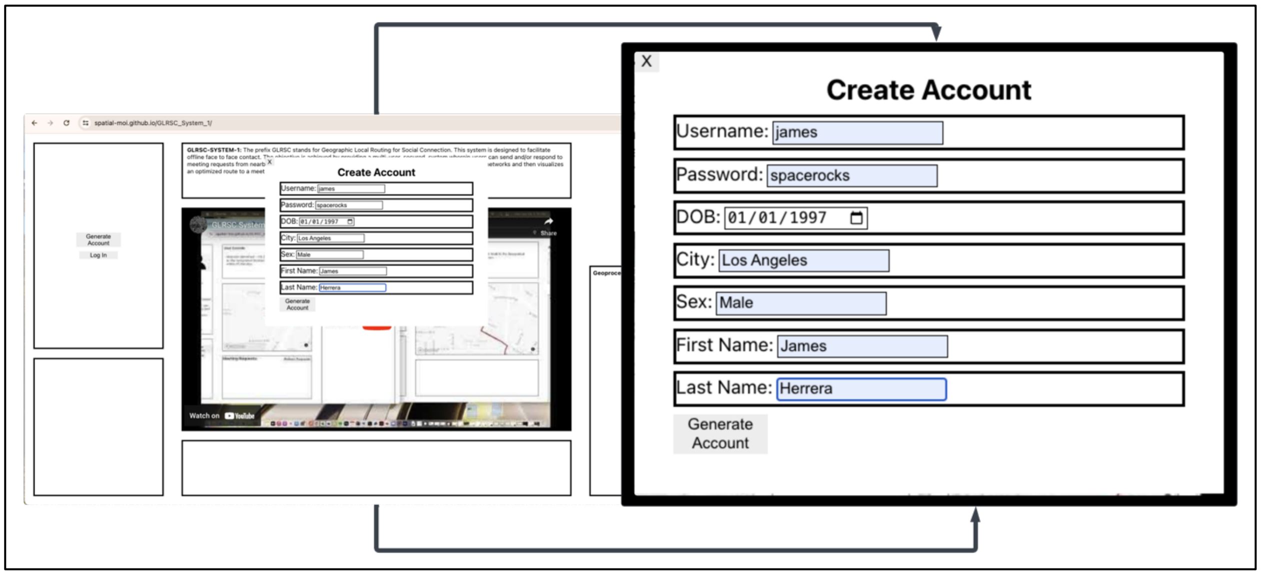
Figure 39 shows the results for the system's create account functionality.


Create Account

Username:	james
Password:	spacerocks
DOB:	01/01/1997
City:	Los Angeles
Sex:	Male
First Name:	James
Last Name:	Herrera

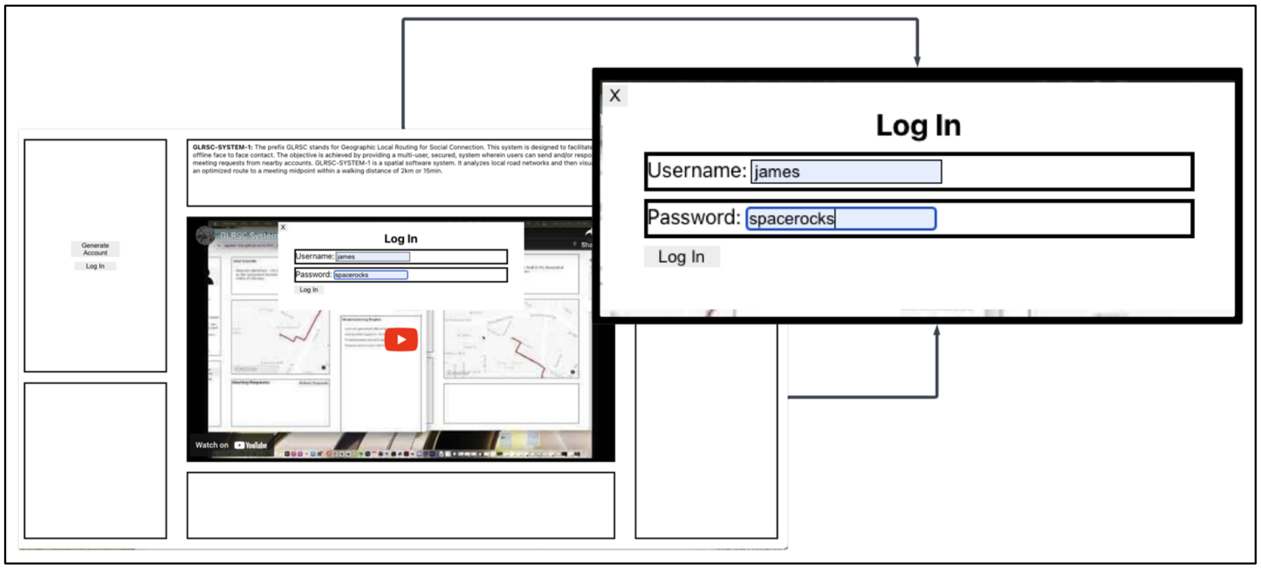
Figure 39. System's generate account function

The figure above shows how the user is expected to create or generate an account. The user must type in their username, password, date of birth, city, sex, first name, and last name. In



Username: james
Password: spacerocks
DOB: 01/01/1997
City: Los Angeles
Sex: Male
First Name: James
Last Name: Herrera

Figure 39, a user has completed the form. Once the user hits the ‘Generate Account’ button, the system successfully processes the request and returns a success message to the user. The next figure, Figure 40 demonstrates the system’s log in functionality.


GLRSC-SYSTEM-1: The prefix GLRSC stands for Geographic Local Routing for Social Connection. This system is designed to facilitat
offline face to face contact. The objective is achieved by providing a multi-user, secured, system wherein users can send and/or respo
meeting requests from nearby accounts. GLRSC-SYSTEM-1 is a spatial software system. It analyzes local road networks and then visu
an optimized route to a meeting midpoint within a walking distance of 2km or 15min.

Log In

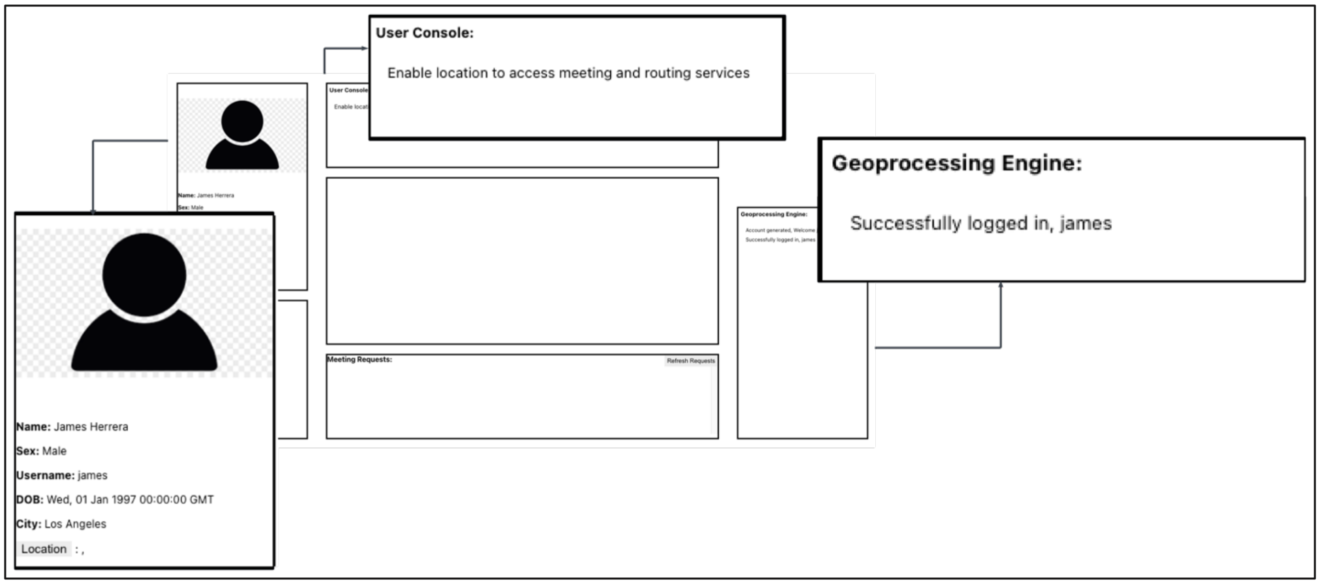
Username: james

Password: spacerocks

Figure 40. System's log in functionality

Figure 40 depicts the system's log in functionality. The user must enter their username and password in the form. In the example provided, a user has typed their username and password. Once the user hits the 'Log In' button, the user will enter the system's second page and receive a message that the user has successfully logged in.

Figure 41 shows the systems second page and zooms in on some key features.


User Console:

Enable location to access meeting and routing services

Geoprocessing Engine:

Successfully logged in, james

Name: James Herrera

Sex: Male

Username: james

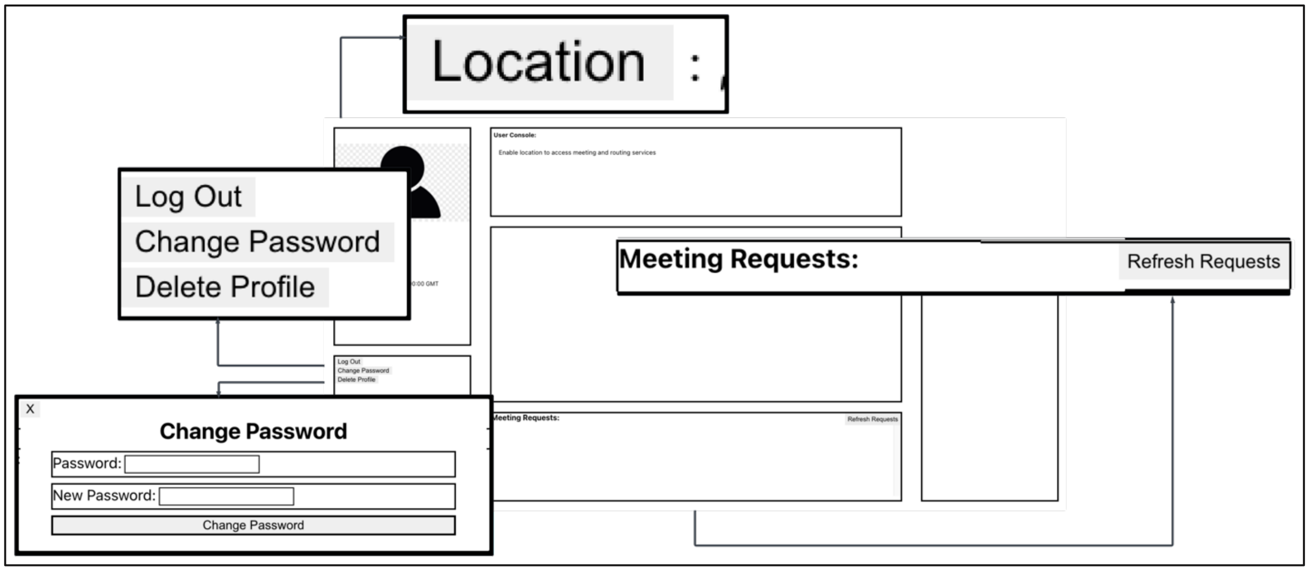
DOB: Wed, 01 Jan 1997 00:00:00 GMT

City: Los Angeles

Location : ,

Figure 41. Personalized account, user instructions, algorithmic transparency

Figure 41 helps demonstrate how the system provides a personalized private account by returning the user's information when a user logs in using the correct username and password. The personalized private account helps produce a multi-user environment by giving ever user their own private account which they control. The second zoom in on Figure 41 provides user instructions. One assertion that I had to make during this project is that the user does not know how the system works. Therefore, to provide for a more friendly user experience, I provided the system with an interface that gives user instructions. In Figure 41, these instructions state, "Enable location to access meeting and routing services". The statement attempts to convince the user to enable the system's geolocation capabilities by clicking the 'Location' button shown under the 'City' attribute within the user's account details and profile section. Lastly, when the user logs in, the system will provide them will messages in the Geoprocessing Engine that describe system processes. The Geoprocessing Engine's messages are meant to promote algorithmic transparency. In Figure 41, the message shown states that a 'Read' action has occurred, one of the four CRUD actions; The user has logged in. The system also provides messages for the other CRUD account including when a user creates an account, when the user changes their password, when a user deletes their account.


User Console:

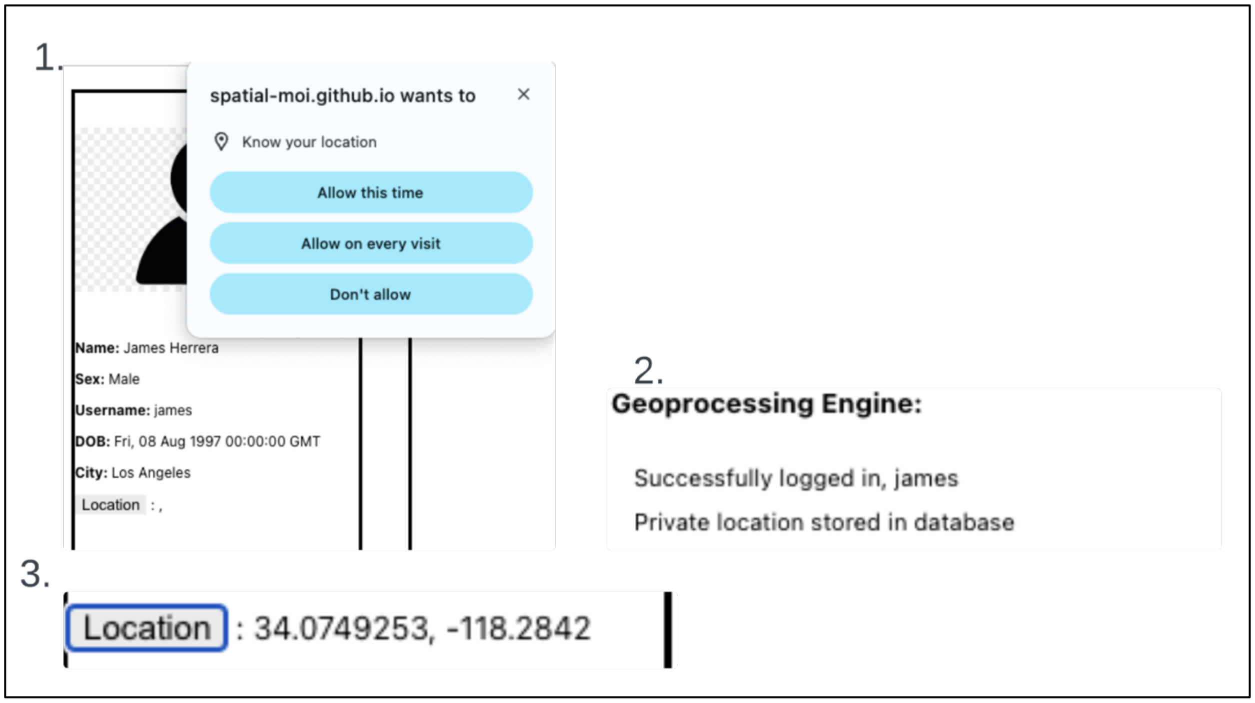
Enable location to access meeting and routing services

\Figure 42. Location, remaining CRUD, change password, meeting requests header

\Figure 42 has a zoomed in view on the panel containing the remaining CRUD buttons. The log out button leads the user back to the home page and provides a message in the Geoprocessing

Engine. The change password button makes a change password modal appear on the screen. The change password modal is seen in

\Figure 42 right below the CRUD buttons. In the modal, the user can type in a new password. When the user clicks “Change Password” their password will change, and the system will keep the user logged in. When the user logs in, the system provides the meeting request header with the “Refresh Requests” button in case the user wants to check for meeting requests upon signing in. However, the system’s entire functionality relies on having a longitude latitude coordinate for each registered user. The location button will raise a prompt to enable location services. The user must respond to the prompt to have access to any service.



Name: James Herrera
Sex: Male
Username: james
DOB: Fri, 08 Aug 1997 00:00:00 GMT
City: Los Angeles
Location : ,

Geoprocessing Engine:

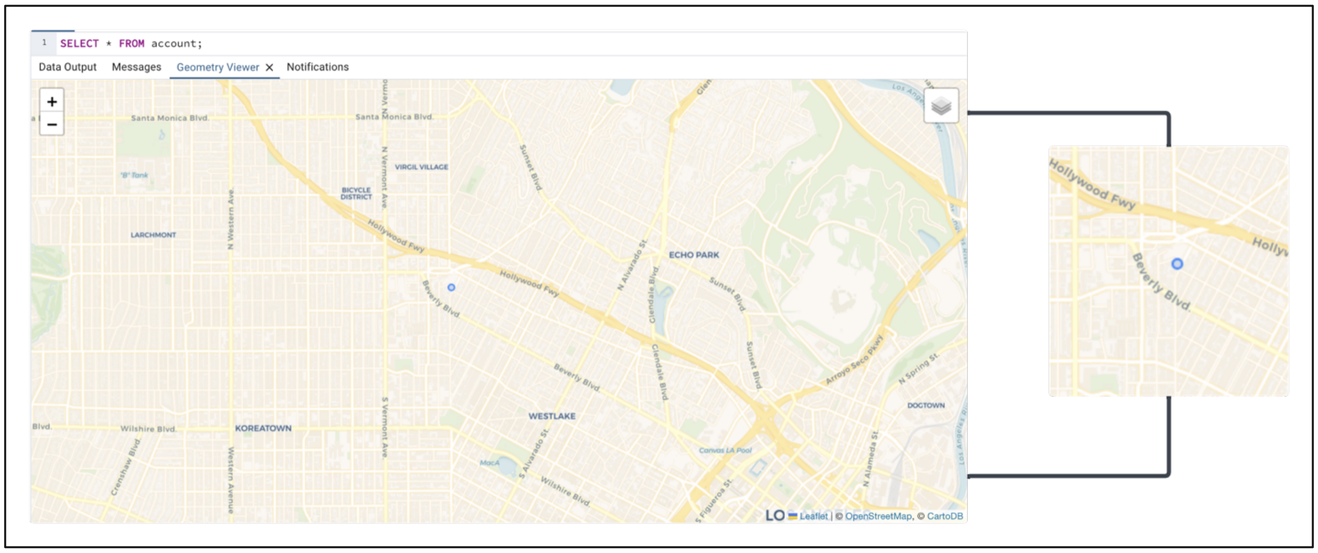
Successfully logged in, james
Private location stored in database

Location : 34.0749253, -118.2842

Figure 43. Phase 1 results for storing location

Figure 43 depicts some elements of the system interface at log in after the user clicks the location button. The location button is programmed to trigger a location prompt which appears at the top left corner of Figure 43. To proceed with the system's services, the user must either select, "Allow this time" or "Allow on every visit". When the user selects one of the two affirmative options, the user will receive a message in the Geoprocessing Engine on the lower right that their location has been stored in the database. At the same time, the system processes and returns the user's longitude and latitude coordinates and displays them to the right of the 'Location" button. In conclusion, Figure 43 shows that the system is successful in capturing the user's location and returning it to the user.

However, for Phase 1 to be successful, it is not sufficient for the system to only store the location, but it must also make sure the location is accurate. For this reason, PgAdmin software is used to view the location data on a map interface and to verify the results. Figure 44 shows how PgAdmin is used to verify the results.



```
SELECT * FROM account;
```

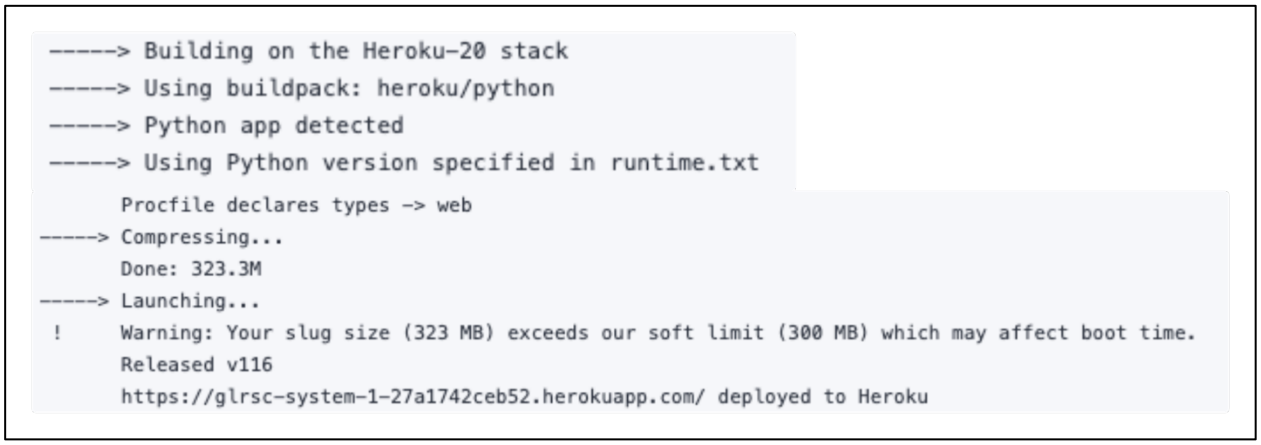
Figure 44. Verify location accuracy with Geometry Viewer in PgAdmin

When a user generates an account and stores their location, as described in the previous figure, the system stores the information in a PostgreSQL database within a table named account.

Using software like PgAdmin, I was able to view the results, the accounts registered, and their locations. I queried the account table and selected the location column. Each column that holds a spatial data type will also have an additional tab to open the Geometry Viewer. Using the Geometry Viewer, I was able to verify that all the tests of latitude and longitude coordinates were in an accurate position within the map. Figure 44 shows an accurate positioning of an intake longitude-latitude coordinate which I confirmed as my own location.

5.2 Phase 2 Results

The Phase 2 objective was to deploy the system in a production live environment using Heroku and Github. Phase 2 also provides a methodology for deployment including the steps required to deploy the Flask application and how to diagnose the system during production. This section presents a few figures to help prove that the system's deployment was a success using Heroku and Github. Figure 45 shows the Heroku GUI deployment logs that are produced during deployment.

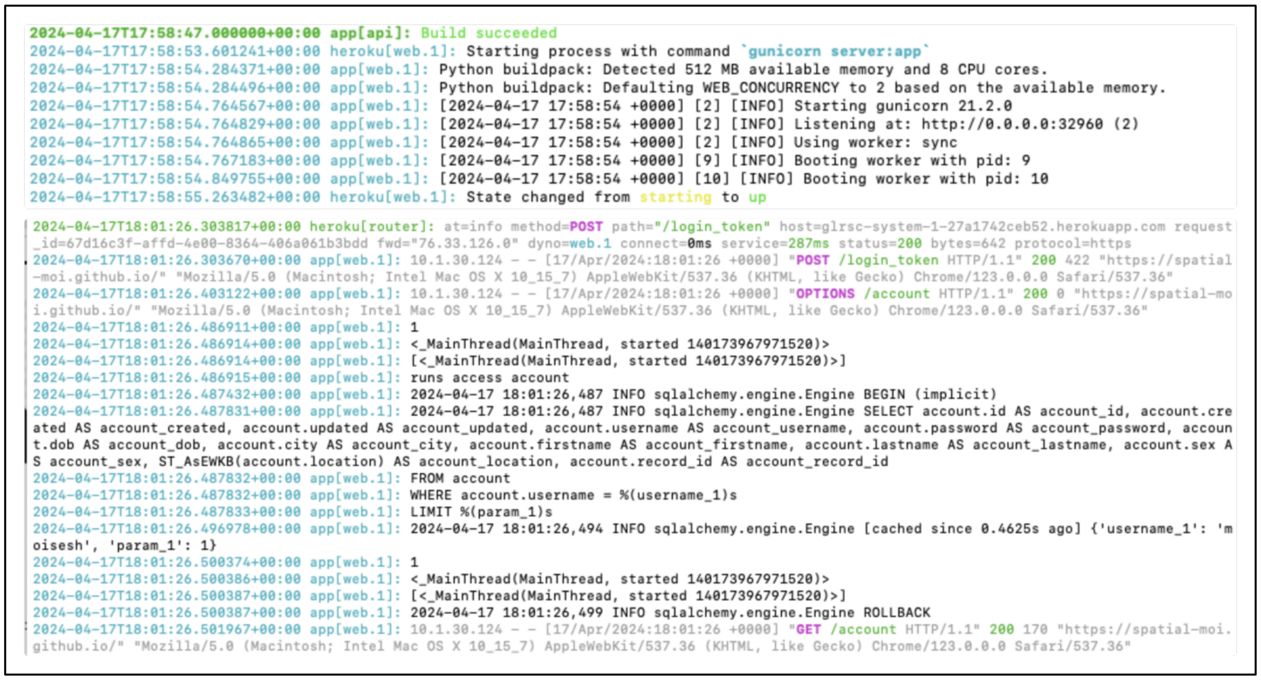


```
 -----> Building on the Heroku-20 stack
 -----> Using buildpack: heroku/python
 -----> Python app detected
 -----> Using Python version specified in runtime.txt
       Procfile declares types -> web
-----> Compressing...
       Done: 323.3M
-----> Launching...
 !     Warning: Your slug size (323 MB) exceeds our soft limit (300 MB) which may affect boot time.
       Released v116
       https://glrsc-system-1-27a1742ceb52.herokuapp.com/ deployed to Heroku
```

Figure 45. Heroku manual deploy success

On the Heroku GUI, the manual deploy option streamlines application deployment. The manual deploy option provides information on several operations and tests that execute during application deployment. Figure 45 shows some of the affirmative logs that indicate a successful deployment; these logs are printed during the final deployment for GLRSC-System-1. In Section 4.2.1, the methodology specifies several instructions. For instance, the methodology specified to change Heroku stacks to version 20. It also specifies to create a runtime.txt file and write the Python version in the file. Finally, it specifies to create a Procfile. In Figure 45, the Heroku GUI manual deploy console verifies that the prior instructions were done successfully. The GUI, seen in Figure 45, claims that system deployment is using Heroku stack 20, the runtime.txt file, and the Procfile. The console also prints the slug size which is useful because it prints the system's memory limit. Finally, on the last line of Figure 45, the console mentions that the system back-end is available at the specified URL domain, which indicates a successful deployment operation.

To verify that deployment is successful, I also visited the domain and discovered that an indicator of successful deployment with Heroku was if the webpage rendered a "Not found" message. Furthermore, I verified successful deployment by monitoring the system and incoming HTTP calls through the Heroku CLI. I tested the application functions such as create an account or delete an account. If my actions on the front-end were reflected on the Heroku CLI, I was assured that the system was working. Figure 46 demonstrates how the developer can use Heroku CLI to verify system deployment and monitor HTTP calls. Figure 46 is made possible by following the methodology described in Section 4.2.1.



```
2024-04-17T17:58:47.000000+00:00 app[api]: Build succeeded
2024-04-17T17:58:53.601241+00:00 heroku[web.1]: Starting process with command `gunicorn server:app`
2024-04-17T17:58:54.284371+00:00 app[web.1]: Python buildpack: Detected 512 MB available memory and 8 CPU cores.
2024-04-17T17:58:54.284496+00:00 app[web.1]: Python buildpack: Defaulting WEB_CONCURRENCY to 2 based on the available memory.
2024-04-17T17:58:54.764567+00:00 app[web.1]: [2024-04-17 17:58:54 +0000] [2] [INFO] Starting gunicorn 21.2.0
2024-04-17T17:58:54.764829+00:00 app[web.1]: [2024-04-17 17:58:54 +0000] [2] [INFO] Listening at: http://0.0.0.0:32960 (2)
2024-04-17T17:58:54.764865+00:00 app[web.1]: [2024-04-17 17:58:54 +0000] [2] [INFO] Using worker: sync
2024-04-17T17:58:54.767183+00:00 app[web.1]: [2024-04-17 17:58:54 +0000] [9] [INFO] Booting worker with pid: 9
2024-04-17T17:58:54.849755+00:00 app[web.1]: [2024-04-17 17:58:54 +0000] [10] [INFO] Booting worker with pid: 10
2024-04-17T17:58:55.263482+00:00 heroku[web.1]: State changed from starting to up
```

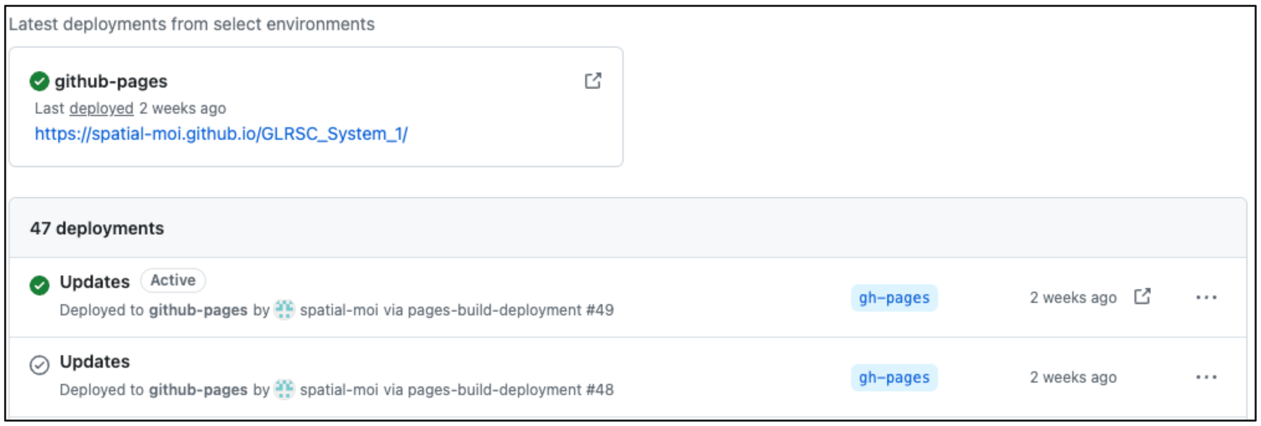
```
2024-04-17T18:01:26.303817+00:00 heroku[router]: at=info method=POST path="/login_token" host=glrsc-system-1-27a1742ceb52.herokuapp.com request
_id=67d16c3f-affd-4e00-8364-406a061b3bdd fwd="76.33.126.0" dyno=web.1 connect=0ms service=287ms status=200 bytes=642 protocol=https
2024-04-17T18:01:26.303670+00:00 app[web.1]: 10.1.30.124 - - [17/Apr/2024:18:01:26 +0000] "POST /login_token HTTP/1.1" 200 422 "https://spatial
-moi.github.io/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36"
2024-04-17T18:01:26.403122+00:00 app[web.1]: 10.1.30.124 - - [17/Apr/2024:18:01:26 +0000] "OPTIONS /account HTTP/1.1" 200 0 "https://spatial-mo
i.github.io/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36"
2024-04-17T18:01:26.486911+00:00 app[web.1]: 1
2024-04-17T18:01:26.486914+00:00 app[web.1]: <_MainThread(MainThread, started 140173967971520)>
2024-04-17T18:01:26.486914+00:00 app[web.1]: [<_MainThread(MainThread, started 140173967971520)>]
2024-04-17T18:01:26.486915+00:00 app[web.1]: runs access account
2024-04-17T18:01:26.487432+00:00 app[web.1]: 2024-04-17 18:01:26,487 INFO sqlalchemy.engine.Engine BEGIN (implicit)
2024-04-17T18:01:26.487831+00:00 app[web.1]: 2024-04-17 18:01:26,487 INFO sqlalchemy.engine.Engine SELECT account.id AS account_id, account.cre
ated AS account_created, account.updated AS account_updated, account.username AS account_username, account.password AS account_password, accoun
t.dob AS account_dob, account.city AS account_city, account.firstname AS account_firstname, account.lastname AS account_lastname, account.sex A
S account_sex, ST_AsEWKB(account.location) AS account_location, account.record_id AS account_record_id
2024-04-17T18:01:26.487832+00:00 app[web.1]: FROM account
2024-04-17T18:01:26.487832+00:00 app[web.1]: WHERE account.username = %(username_1)s
2024-04-17T18:01:26.487833+00:00 app[web.1]: LIMIT %(param_1)s
2024-04-17T18:01:26.496978+00:00 app[web.1]: 2024-04-17 18:01:26,494 INFO sqlalchemy.engine.Engine [cached since 0.4625s ago] {'username_1': 'm
oisesh', 'param_1': 1}
2024-04-17T18:01:26.500374+00:00 app[web.1]: 1
2024-04-17T18:01:26.500386+00:00 app[web.1]: <_MainThread(MainThread, started 140173967971520)>
2024-04-17T18:01:26.500387+00:00 app[web.1]: [<_MainThread(MainThread, started 140173967971520)>]
2024-04-17T18:01:26.500387+00:00 app[web.1]: 2024-04-17 18:01:26,499 INFO sqlalchemy.engine.Engine ROLLBACK
2024-04-17T18:01:26.501967+00:00 app[web.1]: 10.1.30.124 - - [17/Apr/2024:18:01:26 +0000] "GET /account HTTP/1.1" 200 170 "https://spatial-moi.
github.io/" "Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/123.0.0.0 Safari/537.36"
```

Figure 46. Heroku CLI logs indicate successful system deployment

The figure above displays many lines of codes that describe technical background processes and operations necessary for system deployment. Much of these messages can be difficult to understand. However, some of the messages are easy and practical to interpret. For instance, the first line in Figure 46 prints a message that the build has succeeded which indicates a successful deployment. Several lines below, the logs claim that the state of the application is now up, meaning that the application is now live and ready to take HTTP calls.

The Heroku CLI logs print when HTTP calls reach GLRSC-System-1. For instance, Figure 46 lists routes "/login_token" and "/account". These routes correspond to operations described in Section 4.1.6, and indicate that a live user is accessing and using the system in real-time. The Heroku CLI logs further prove that system deployment was a success and that users are accessing a functional real-time system.

It is also important to verify that the front-end is deployed successfully. After completing the methodology from Section 4.2.2, I visited the domain to verify if deployment is successful. However, it is also important to verify that the correct front-end version is deployed. Figure 47 depicts a method to verify that front-end deployment uses the correct front-end version.


Latest deployments from select environments

github-pages
Last deployed 2 weeks ago
https://spatial-moi.github.io/GLRSC_System_1/

47 deployments

Updates Active
Deployed to github-pages by spatial-moi via pages-build-deployment #49 gh-pages 2 weeks ago

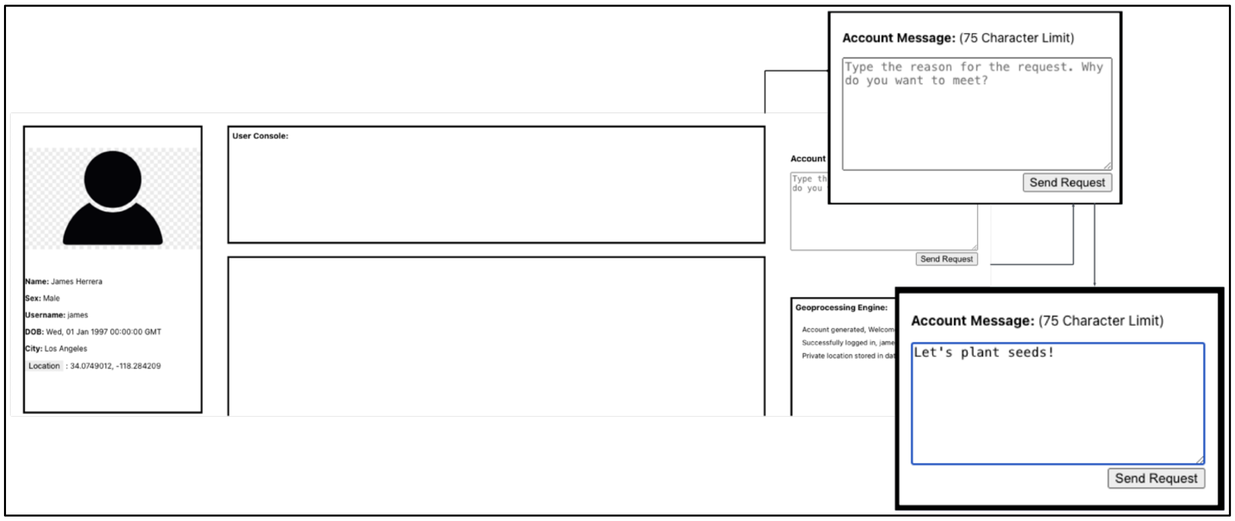
Updates
Deployed to github-pages by spatial-moi via pages-build-deployment #48 gh-pages 2 weeks ago

Figure 47. Github front-end deployment results

The figure above displays the Github Pages functionality within the Github GUI. The Github Pages tab allowed me to verify which deployments are being used for the live production environment. Figure 47 shows updates, which are deployment versions, that were made two weeks ago. On the middle-left portion of the figure, a label reads "47 deployments" which indicates there has been 47 different deployment versions that have been used since the beginning of the project's history. On the upper-left, the GUI lists the most recent and live deployment and gives a URL to the front-end site. Finally, I clicked on the URL, and it led to the system's landing page which verified a successful deployment effort.

5.3 Phase 3 Results

The Phase 3 objective is to provide the user with the ability to send a request to peers within a 2km radius and to respond to requests sent from within a 2km radius. Phase 3 is also responsible for programming the system to calculate the midpoint coordinates once it receives a meeting request and an affirmative response. The system should return the midpoint coordinates to the user. The methodology described in 4.3 proved to be successful in meeting the objective mentioned in this paragraph. Figure 48 displays the system's ability to prepare and send user-generated meeting requests.



Name: James Herrera
Sex: Male
Username: james
DOB: Wed, 01 Jan 1997 00:00:00 GMT
City: Los Angeles
Location : 34.0749012, -118.284209

Account generated, Welcom
Successfully logged in, jame
Private location stored in da

Account Message: (75 Character Limit)

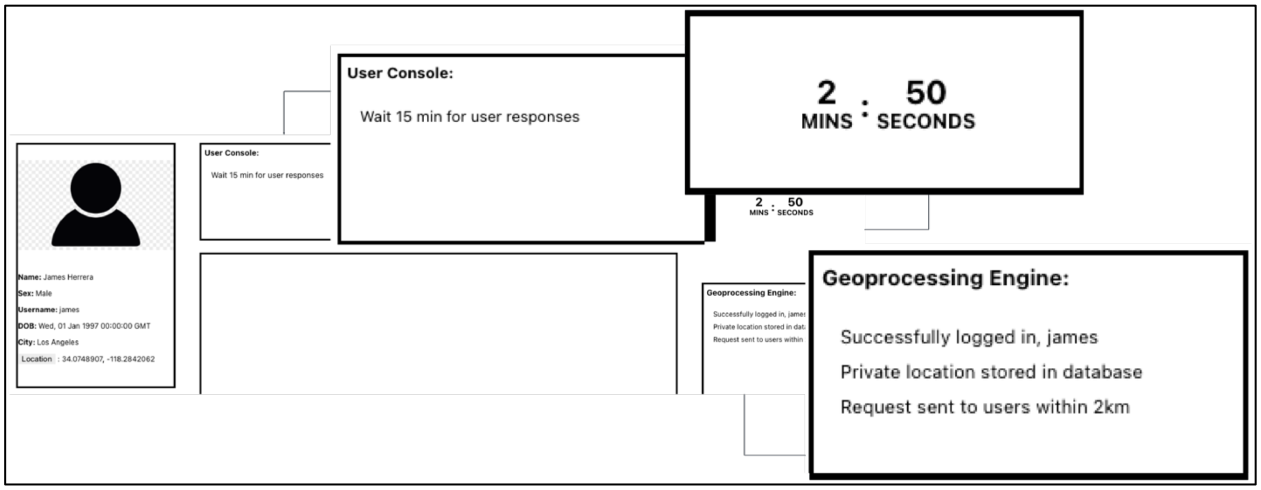
```
Type the reason for the request. Why
do you want to meet?
```

Account Message: (75 Character Limit)

```
Let's plant seeds!
```

Figure 48. User sends meeting request with message

As shown in the figure, in the deployed GLRSC-System-1 the user can prepare a custom generated messaged with their meeting request. The meeting request feature is only shown in GLRSC-System-1 after the user has clicked on the 'Location' button. Therefore, Figure 48 is a continuation of Figure 47. In Figure 48, a user has also typed a custom message in the 'Account Message' text box. The message reads, "Let's plant seeds". The custom message highlights the broad applicability of the system and its ability to create specific use cases as seen in Figure 7. Users can customize and set the theme of the meeting.



Name: James Herrera
Sex: Male
Username: james
DOB: Wed, 01 Jan 1997 00:00:00 GMT
City: Los Angeles
Location : 34.0748907, -118.2842062

User Console:

Wait 15 min for user responses

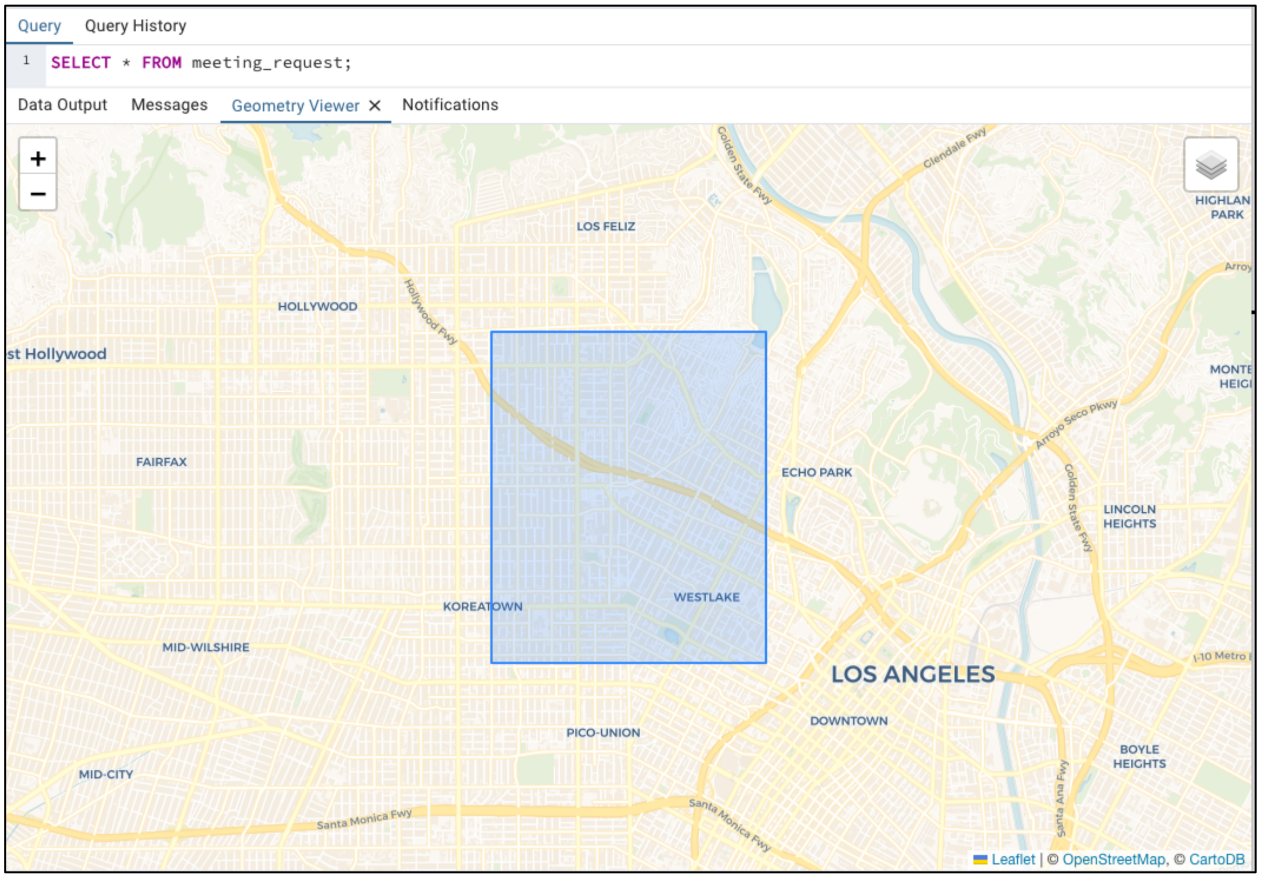
Geoprocessing Engine:

Successfully logged in, james
Private location stored in database
Request sent to users within 2km

Figure 49. User sends request results

Once the user clicks on the button "Send Request", the system will display three new elements. The first can be found in the User Console. The new message instructs the user to wait 15 minutes for users to respond to the request. In place of the meeting request box, the system renders a countdown of 15 minutes that helps the reader visualize the process taking place. Lastly, the Geoprocessing Engine, in accordance to the objective of providing algorithmic transparency, prints a message that informs the reader that their request has been sent to users within a 2km radius area. In the back-end, the 2km radius area is used to group accounts into a list of potential participants.

PgAdmin is used verify that the system creates a spatial buffer that is appropriate for this application. To meet project requirements and ensure that the system creates and uses a spatial buffer that is useful to GLRSC-System-1, some design modifications were made to the spatial buffer. What was originally planned to be a circular search area of 2km radius is instead a square area as seen in Figure 50.



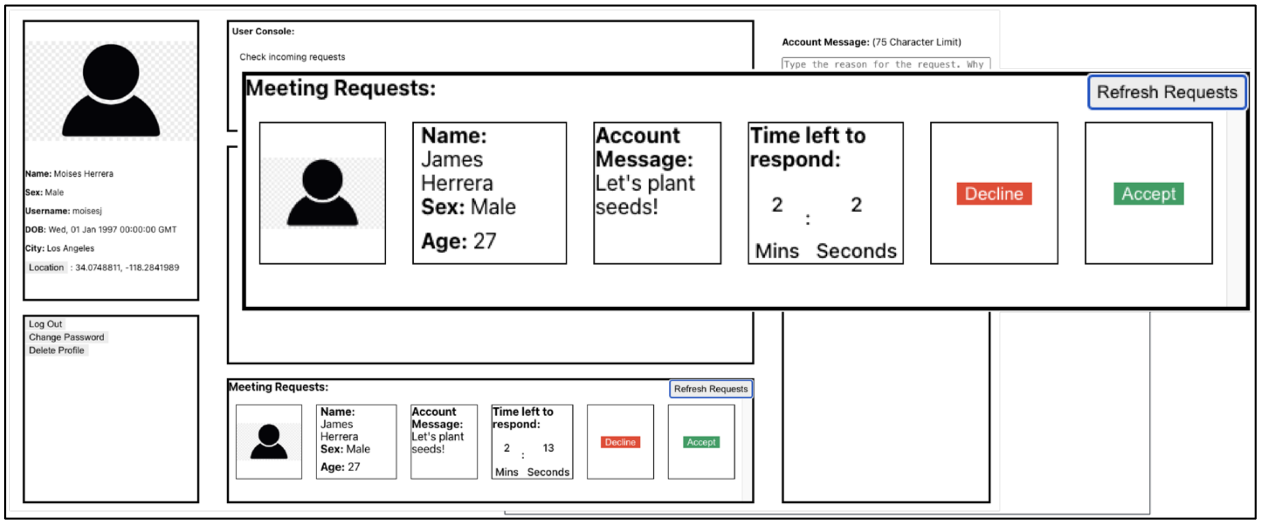
```
SELECT * FROM meeting_request;
```

Figure 50. PgAdmin Geometry Viewer renders rectilinear buffer

Nevertheless, Figure 50 shows a spatial buffer that is appropriate for GLRSC-System-1 use. Changes had to be made because attempts to produce a circular area were unsuccessful and no clear solution was available. The specific errors that were faced will be discussed in the Chapter 6. However, a rectilinear buffer does not indicate an unsuccessful search operation. In fact, the system uses the rectilinear buffer area and successfully searches for potential participants. The area's radius is roughly 2km, but some measurement errors should be expected.

From Figure 48 to Figure 50, this chapter section has taken the perspective of a user who is making a request for a meeting. However, GLRSC-System-1 is a multi-user environment that requires at least two user roles. Figure 51 shows the perspective of a user who is responding to a nearby meeting request right after the user clicks on the "Refresh Requests' button within the

Meeting List header. The prerequisites for a meeting request to appear on a user’s page are that a requestor has already sent the request and that the responder has enabled location storage on their account before the requestor sent the request.



Name: Moises Herrera
Sex: Male
Username: moisesj
DOB: Wed, 01 Jan 1997 00:00:00 GMT
City: Los Angeles
Location : 34.0748811, -118.2841989

User Console:
Check incoming requests

Account Message: (75 Character Limit)
Type the reason for the request. Why

Meeting Requests:

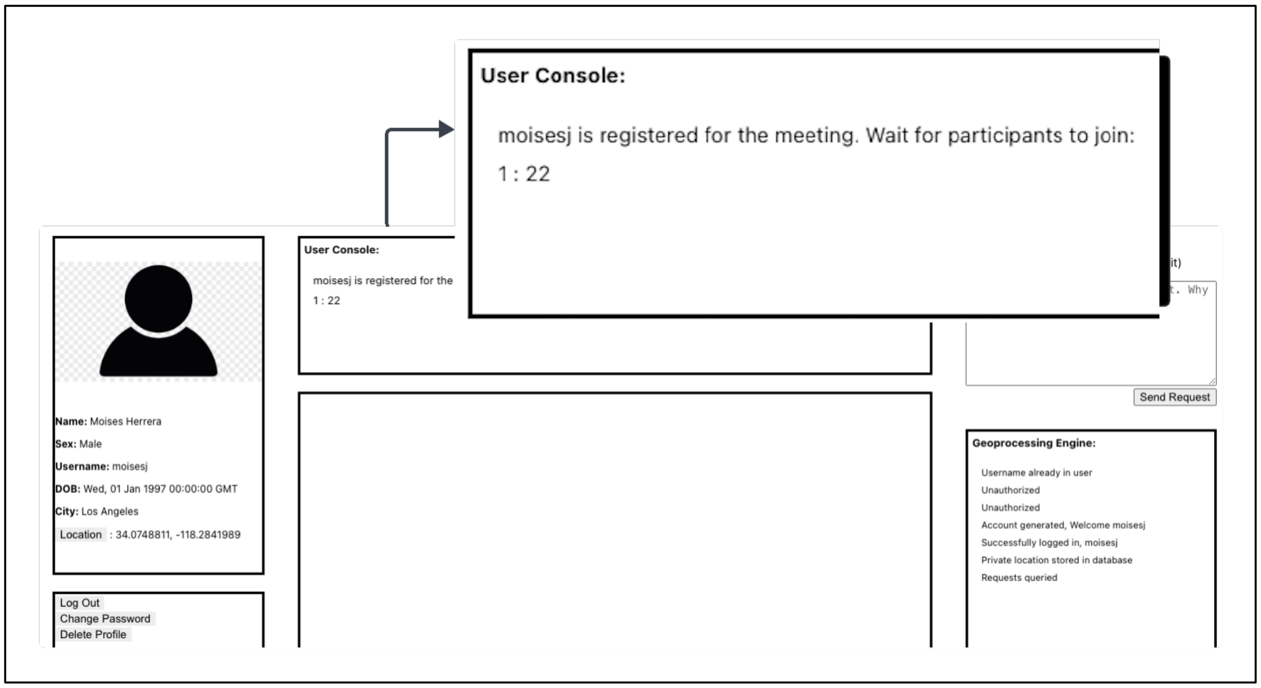
Name: James Herrera **Sex:** Male **Age:** 27
Account Message: Let's plant seeds!
Time left to respond: 2 : 2 Mins Seconds

Meeting Requests:

Name: James Herrera **Sex:** Male **Age:** 27
Account Message: Let's plant seeds!
Time left to respond: 2 : 13 Mins Seconds

Figure 51. Requestor receives request

Figure 51 shows a user who has clicked on the “Refresh Requests” button and has received a request. The request appears under the Meeting Requests header. Any and all requests appear under this header and more than one request can appear at any moment. However, a user can only accept one request at a time. A user’s system who has more than one request, and accepts one request, will remove all other requests from the list. Figure 51 shows a zoomed in view of one request. The request is divided into columns containing important meeting details. The first column shows a profile picture which is important for verifying identity. The second column shows key account details such as name, age, and sex. Again, these details are important for users who are responding to the request. The account message is in column three and it is used to establish the theme for the meeting. The fourth column holds a timer that counts down from 15 minutes which begins when a user hits the ‘Send Request’ button. The timer is important because it provides a user experience that is consistent, regular, ordered and predictable. Because of the timer, the users will receive notification on the midpoint within 15 minutes and the meeting can happen immediately within a 30-minute period. Finally, the last two columns hold two separate buttons: delete and accept. The decline button removes the user from the list of potential participants. The accept button will lead to Figure 52 on the responder's interface.


User Console:

moisesj is registered for the meeting. Wait for participants to join:

1 : 22

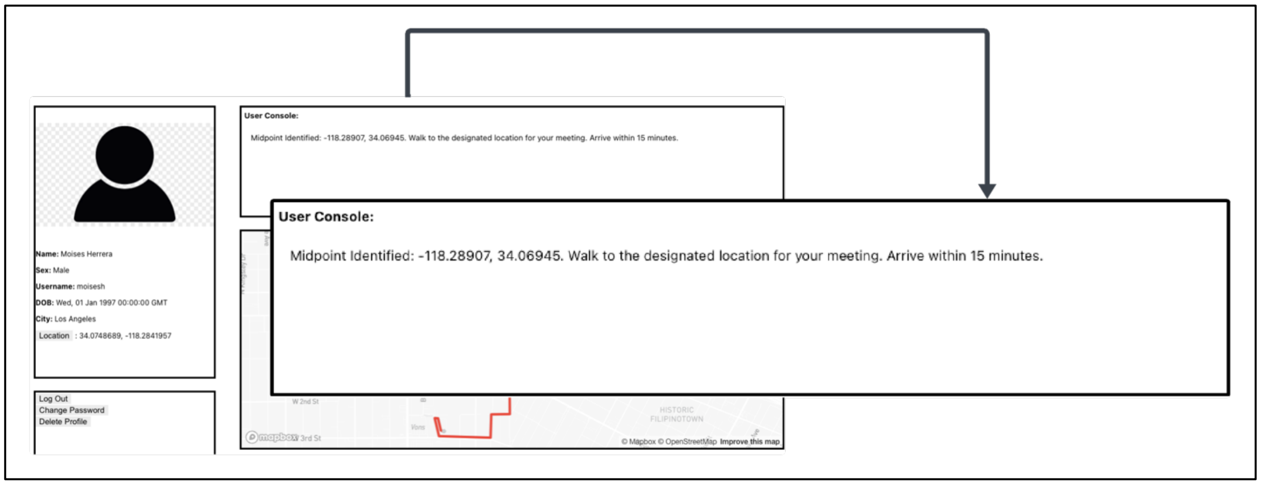
Name: Moises Herrera
Sex: Male
Username: moisesj
DOB: Wed, 01 Jan 1997 00:00:00 GMT
City: Los Angeles
Location : 34.0748811, -118.2841989

Geoprocessing Engine:

Username already in user
Unauthorized
Unauthorized
Account generated, Welcome moisesj
Successfully logged in, moisesj
Private location stored in database
Requests queried

Figure 52. Responder waits for meeting instructions

The figure above shows a user who has accepted a request and is waiting for the countdown to end. The countdown that appears on the User Console is the same countdown seen in Figure 49 and Figure 51. On both sides of the system, responders and requestor, the countdown remains the same. A synchronous timer across the multi-user environment allows the system to deliver important midpoint and meeting information at the same time. The timer also makes the system fair for users by giving each user the same amount of time to respond to the event. The timer is set to 15 minutes which is a short amount of time that facilitates fast turnarounds and meeting arrangements. After the countdown is over both user roles receive a similar message with important details as seen in Figure 53.



Name: Moises Herrera
Sex: Male
Username: moisesh
DOB: Wed, 01 Jan 1997 00:00:00 GMT
City: Los Angeles
Location : 34.0748689, -118.2841957

User Console:

Midpoint Identified: -118.28907, 34.06945. Walk to the designated location for your meeting. Arrive within 15 minutes.

User Console:

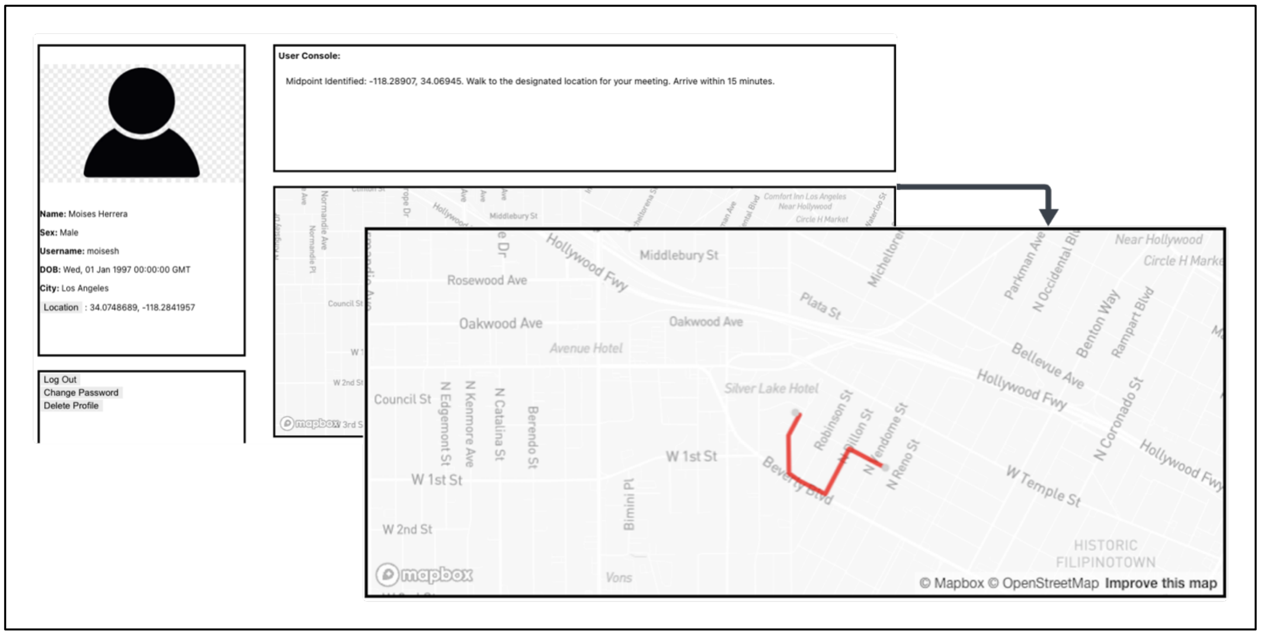
Midpoint Identified: -118.28907, 34.06945. Walk to the designated location for your meeting. Arrive within 15 minutes.

Figure 53. System returns midpoint and meeting instructions

Figure 53 reads, “Midpoint Identified: -118.284016, 34.0746283. Walk to the designated location for your meeting. Arrive within 15 minutes”. This message arrives to all active participants at the same time. Furthermore, the midpoint latitude-longitude coordinates are uniform for all active participants. In the back-end, the system uses all active participants’ coordinates to determine the midpoint. Figure 53 demonstrates that the system successfully computes the midpoint among nearby users in a shared meeting. It also sets the stage for the real in person meeting because it gives the user instructions to walk to the midpoint. Lastly, because the system returns the midpoint and instructions to all active meeting participants at the same time, one can be sure that, if the user chooses to walk to the midpoint, the user will arrive within the same period as all other active participants, given that all participants also follow the systems instructions.

5.4 Phase 4 Results

The Phase 4 objective is to display a route from the user's location to the midpoint introduced at the end of Phase 3. The methodology described in 4.4, when implemented, successfully renders the route from the user's location to the midpoint. Figure 54 shows the interface from a user's perspective once the system returns the midpoint.



Name: Moises Herrera
Sex: Male
Username: moisesh
DOB: Wed, 01 Jan 1997 00:00:00 GMT
City: Los Angeles
Location : 34.0748689, -118.2841957

Log Out
Change Password
Delete Profile

User Console:

Midpoint Identified: -118.28907, 34.06945. Walk to the designated location for your meeting. Arrive within 15 minutes.

Figure 54. Phase 4 results

Figure 54 demonstrates that GLRSC-System-1 is a complete system that has met its objective to facilitate an immediate meet up at a midpoint along the road network with nearby peers. The Midpoint_Router.js component is seen in "column2-box2" with a route from the user's location to the meeting midpoint. The results prove that the algorithmic, persistence, and visualization processes in the front and back-end successfully conduct their respective roles.

Furthermore, GLRSC-System-1 proved to be successful in a deployed live environment. The system was tested with live users who synchronously accessed, used, and queried the system. Figure 54 shows Phase 4 results from the responder's interface. However, Figure 55 shows Phase 4 results, as seen on GLRSC-System-1, for both the requestor and responder.

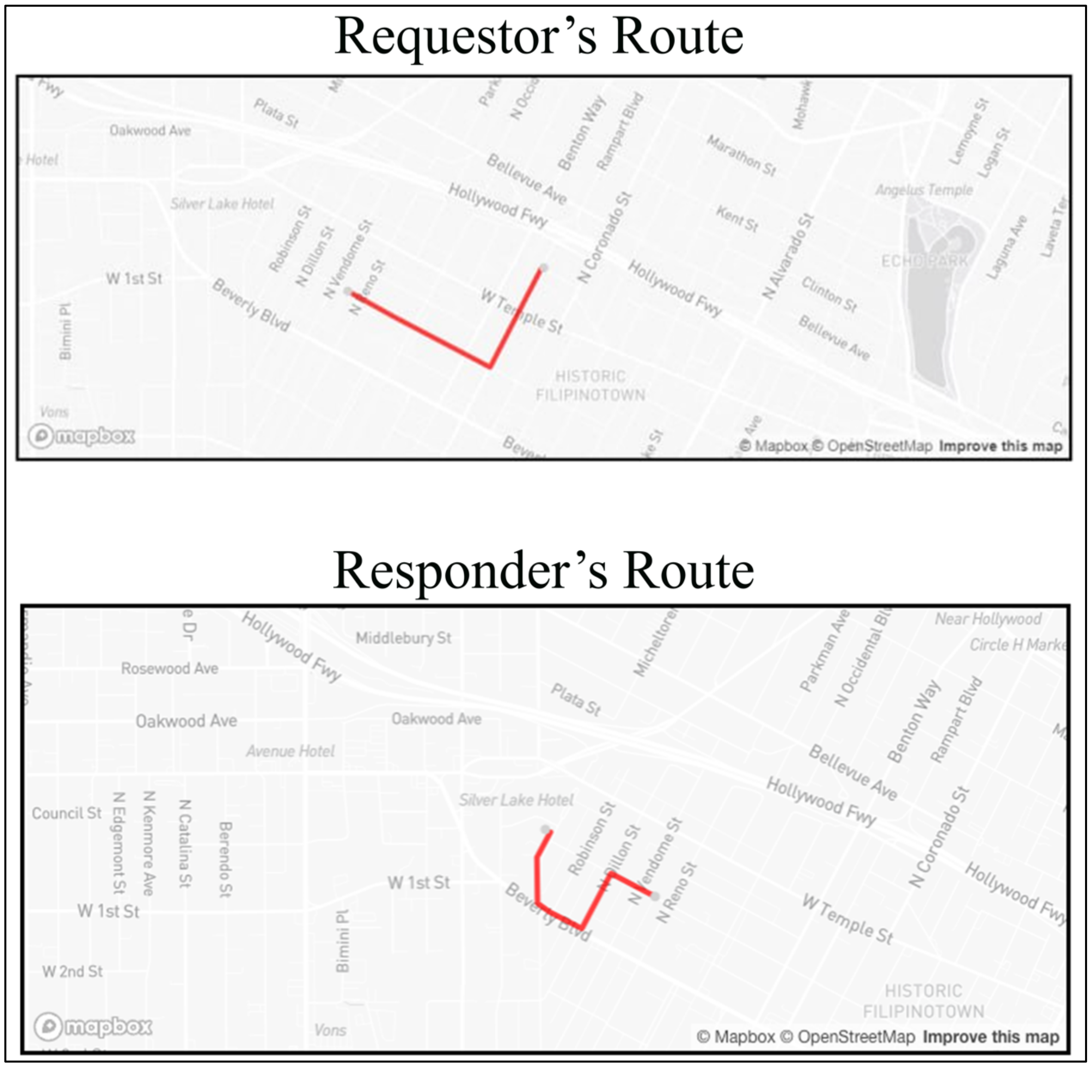


Figure 55. Phase 4 results for two live users

Figure 55 displays two routes that meet each other at a midpoint. Both routes are displayed within the GLRSC-System-1 interface on the users' respective accounts. The requestor's route begins at their location. From their map, the requestor's location is south of Hollywood Fwy and above Temple St. The route ends between N. Vendome St and N Reno St. The responder's route begins at their location. From their map, the responder's location is right at Silver Lake Hotel. The route ends between N. Vendome St and N Reno St. Both users' routes destination points land in the same area. Their timers are also synchronous meaning that they will receive the route on the interface at the same time. The User Console informs both users to arrive within fifteen minutes; the instructions provide a period whereby they are both expected to arrive, and it will help ensure that they coincide at the meeting midpoint.

Finally, to conclude, Phase 4 demonstrates GLRSC-System-1 success in returning midpoint information, routes, and instructions for a meeting with more than two participants. Figure 56 displays Phase 4 results, as seen on the GLRSC-System-1 interface, for three separate accounts who tested the system within a live environment.

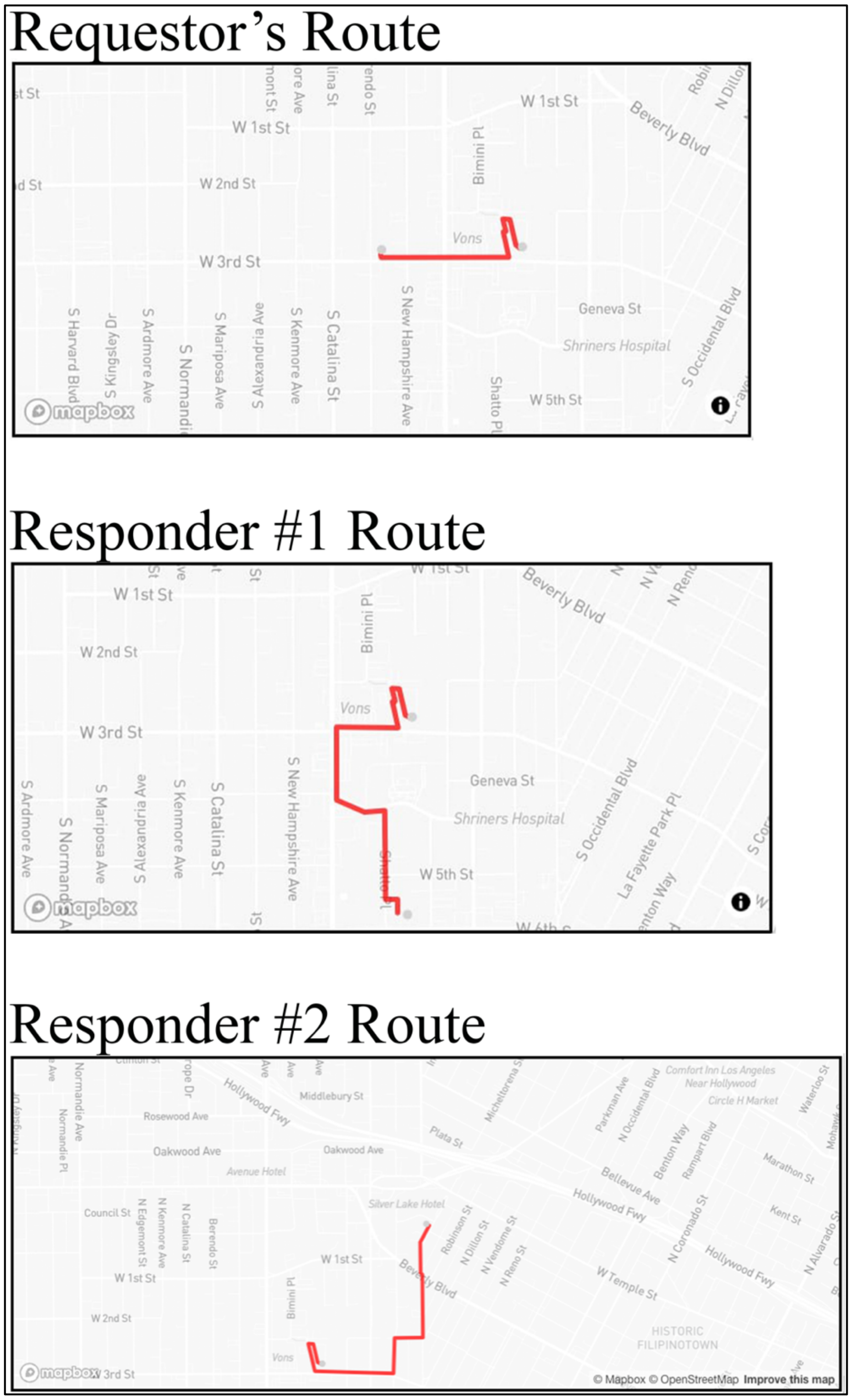


Figure 56. Phase 4 results for more than two users

Figure 56 displays three routes that meet one another at a midpoint. All three routes are displayed within the GLRSC-System-1 interface on the users’ respective accounts within “column2-box2”. The requestor’s route begins at their location, between S. New Hampshire Ave and S. Catalina St, and on 3rd St. Their route ends at a point just east of Vons supermarket. The first responder’s route begins at their location, on Shatto Pl, just south of W 5th St. Their route ends at the same point just east of Vons supermarket. The second responder’s route begins at their location, Silver Lake Hotel, and ends at the same point just east of Vons supermarket. These results are from the accounts of all three users who used the deployed system at the same time but in different locations. All three users were within a 2km radius zone from the requestor’s geolocation. As discussed with Figure 55, the system delivers the route information and further instructions synchronously ensuring that the users/pedestrians coincide at the meeting midpoint at the same time.

Chapter 6 Conclusions

Throughout the course of this thesis, GLRSC-System-1 implementation of GLRSC has been both successful and filled with moments where system improvement was recognized and documented. This thesis has been successful in introducing GLRSC. Through GLRSC implementation in GLRSC-System-1, this thesis demonstrates how routing technologies and multi-user environments can be integrates to achieve GLRSC. The application has real potential to increase the number of instances of social encounters for its users, and thereby alleviate the loneliness epidemic. Not only that, but GLRSC-System-1 outlines a method that allows the user to dictate the moment and topic of that encounter. The system also shows success in algorithmic transparency and user account security.

The results chapter proves that Chapter 4 Methodology describes methodologies that are effective for implementing a base-level GLRSC system, which consists of the four phases: multi-user environment, system deployment, a meeting service, and a routing service. Therefore, this thesis has successfully identified the components necessary for implementing a base-level GLRSC system. As such, this thesis provides the methodologies and frameworks necessary for future developers to create their own implementations of GLRSC and are encouraged to do so.

Lastly, this thesis present GLRSC-System-1 as a functional live and fully closed system that has the capacity to be used and applied in present time. The existence of GLRSC-System-1 is hopefully not only helpful for individuals who experience loneliness, but it also serves as a tangible resource from which discussions may emerge on GLRSC.

Nonetheless, as the first iteration or version, GLRSC-System-1 holds many areas of improvement, which will be considered in future work. For the current iteration, ten areas of improvement are identified: graph processing packages, deployment environments, design and styling, UI separation of concerns, safety, debugging services, socket technology, object relational mapping, geospatial operations, meeting place algorithm variations.

6.1 Graph Processing Packages

GLRSC-System-1 uses NetworkX and OSMNX python graph processing packages to process and identify a midpoint along the road network. However, from a user's perspective, OSMNX may be taking too long to return the midpoint. Currently, within the user experience, once a user has accepted a request or has sent a request, when the timer of fifteen minutes terminates, the system takes roughly 7 seconds to process a graph from OSMNX and identify a midpoint. The system takes another 2 or 3 seconds to render the route on the graph, for an estimate total of ten seconds. During these ten seconds, the user may lose interest in the system or believe that the system has failed. To engage users as much as possible, graph processing must be sped up. One mechanism that can potentially increase processing speed is to try alternative graph processing packages such as Medina, a python package with similar functionalities as OSMNX. However, the main obstacle to overcome is to identify the computer location where the processing takes place. Once the location is identified, actions can be taken to optimize the speed related to the computer or software component. It would be smart, even, to determine if there is a way to reduce Mapbox's search space so that it has less nodes and edges to check for a shortest path; this could possibly reduce the size of the reference dataset. But further research needs to be done to determine how to reduce Mapbox's reference dataset, if possible. However, further diagnoses should be made on the root source of slow processing speed during midpoint computation.

6.2 Deployment Environments

The system's current use of Heroku as a deployment platform is inadequate for production level services because Heroku does not provide a window to view spatial data recorded in the database. As a system administrator, it is necessary to check and test the functionality of the system including viewing the system records and verifying data. For instance, the system's ability to generate and store a buffer search area should be verified and tested in a production environment. However, Heroku does not have a service for viewing spatial data for deployed systems. Therefore, a system administrator cannot check to see if the buffer zone has a radius of 2km. There is only a high level of confidence that the buffer zone does have a radius of 2km because the system was tested to verify the correct size of the buffer zone in the development environment. Nevertheless, the buffer size cannot be fully verified in production. Therefore, Heroku is not adequate for supporting GLRSC-System-1 in a production level setting.

6.3 Design and Styling

GLRSC-System-1 was not developed with design nor styling in mind which resulted in a simple and plain interface. CSS could be applied to add some aesthetic dimensions. Images would also help make the background more appealing. The landing page can improve to be more dynamic, appealing, and engaging. Additionally, the route visualization was designed to be practical but not aesthetically pleasing or unique. Much more can be done to improve the route such as improving the color palette, choosing a more engaging basemap, styling the routes and end points, and incorporating route movement with animation.

6.4 UI Separation of Concerns

The system currently struggles to meet both requestor and responder expectations for conditional rendering because both roles share a single page interface. Conditional rendering in the system is difficult to comprehend, manage, and perfect because the single page interface uses the same redux variables for both user roles. As a result, some interface components still appear when they should be hidden. For instance, when a requestor sends a request, the meeting list header should disappear altogether, but it does not. Likewise, when a responder accepts a request, the message request form should disappear altogether, but it does not. The system cannot meet both conditional rendering standards using the same single page interface and shared redux variables. Therefore, there must be a separation of concerns on the user interface. In other words, the single page system must extend to at least two pages assigned to the each of the two user roles; there should be a single page for the requestor only, and another single page for the responder only.

6.5 Safety

Safety is a serious concern for public use of GLRSC-System-1. While GLRSC-System-1 is not released to the public, further research should prioritize features and systems that increase and ensure user safety. For instance, the system could develop a meeting log feature that will record the meeting details and participants. If an accident ever occurs, the system knows when, where and with whom it occurred. Additionally, GLRSC-System-1 should integrate identity verification systems to ensure that users are who they claim to be. A safety standard should be defined, documented, implemented, tested, and improved.

6.6 Debugging Services

Future iterations of GLRSC-System-1 will require working, improving, expanding, and reorganizing the codebase. During these tasks, more challenges and errors are likely to arise. GLRSC-System-1 requires a more robust system to identify system errors. During the development of GLRSC-System-1, the terminal and console log functions were invaluable assets that helped diagnose issues. However, stronger records and logging systems can vitally improve diagnosis speed. Logging systems should be programmed into the codebase to maximize diagnosis potential and feasibility. A stronger debugging system will increase confidence of the system and allow developers to work with software development more easily.

6.7 Socket Technology

Socket technology is the suite of packages and systems that allow chatboxes and instant messengers to function the way that they do. Sockets provide a path through which one computer can interact with the other without having to check a database. GLRSC-System-1 enables communication by requiring its users to check the database regularly; this approach makes for a poor user experience where communication is not automatic. GLRSC-System-1 communication could vastly improve with socket technology. For instance, once GLRSC-System-1 integrates socket technology there will be no need to have a refresh_requests button because all requests will arrive automatically, which will make the user experience more exceptional, engaging and captivating. More research should be conducted on how socket technology can improve GLRSC-System-1 objectives.

6.8 Object Relational Mapping

Object relational mapping, ORM, is a powerful database technology that helps back-end codebases create database designs through a script. The script itself could be written in any back-end programming language like Python or Java. Nevertheless, ORM makes system development easier. GLRSC-System-1 does use ORM. However, it was not done as efficiently as it could have been. GLRSC-System-1 uses Flask SQLAlchemy as an ORM library but fails to capture the value of the library. When the system starts, Flask SQLAlchemy reads the system's code and creates the database design, but table updates are not linked automatically. Instead, the system does all table updates manually. For instance, when the user accepts a request, the system manually adds the user to a separate table that records all active participants, but the action should be automatic. More research should be conducted to implement ORM correctly and take advantage of its features to the extent that they benefit GLRSC-System-1.

6.9 Geospatial Operations

A major area of improvement is in the system's geospatial operations. The project failed to meet the expectations laid out during project planning. For instance, the objective during Section 4.3.2 was to create a perfectly circular buffer area but a rectilinear buffer area was used instead as shown in Figure 50. The objective was not met because the system required a srid of 4326 corresponding with WGS 84 spatial reference system. PgAdmin software failed to display spatial data with other srid except 4326. Therefore, GLRSC-System-1 required srid 4326 which makes measurements using degrees not meters. When the system defined a circular buffer of radius 2km, the buffer area was distorted into an elongated oval. Because of this error, the system defines a square as the preferred buffer shape. Additionally, the system measures the square using degree not meters. Therefore, the original GLRSC-System-1 objective of searching for potential participants in a circular search space with radius of 2km was not met. Further research requires a comprehensive diagnosis of the issue and a solution.

6.10 Meeting Place Algorithm Variations

The final area of improvement that should be considered in future research and development is the meeting place algorithm. GLRSC-System-1 uses Dijkstra's shortest path algorithm to find a shortest path along the street network between two or more users. Then finds the median node in the path. The median node is assigned as the meeting place. However, there are other methods to identify a meeting place, within an area of 2km radius, that could potentially be safer or more optimal. Street networks are unpredictable, and the resulting midpoint could be under a highway or in an unsafe environment. Further research and development can consider meeting place algorithm variations. For instance, the system could use the nearest park or commercial shopping center as an optimal meeting place. The system can also allow users to set their own meeting place within a 2km radius. All variations are more predictable and maybe even more desirable but require distinct methodologies. User testing and research is required to define the most optimal meeting place algorithm.

References

Adhikary, T. 2022. “How to create a countdown timer using React Hooks.” GreenRoots. https://blog.greenroots.info/how-to-create-a-countdown-timer-using-react-hooks

ArcGIS Network Analyst. n.d. “ArcGIS Network Analyst Extension.” ESRI. Accessed May 1, 2024. https://www.esri.com/en-us/arcgis/products/arcgis-network-analyst/overview

Angeles, M. n.d. “Wireframing User Flow with Wireflows.” Balsamiq. Accessed May 1, 2024. https://balsamiq.com/learn/articles/wireflows/

AWS. n.d. “What is an IDE (Integrated Development Environment)?” AWS. Accessed May 1, 2024. https://aws.amazon.com/what-is/ide/

Bishop, B. 2008. *The Big Sort: Why the Clustering of Like-minded America Is Tearing Us Apart.* New York: Houghton Mifflin.

Biondi, H.G. 2021. “The Role of the Planning Phase in the Project Life Cycle.” Appvizer. https://www.appvizer.co.uk/magazine/operations/project-management/project-life-cycle-planning-phase

Boeing, G. 2017. “OSMnx: New Methods for Acquiring, Constructing, Analyzing, and Visualizing Complex Street Networks.” *Computers, Environment and Urban Systems* 65: 126-139. https://doi.org/10.1016/j.compenvurbsys.2017.05.004.

Brichter, L. 2017. “Our minds can be hijacked: The tech insiders who fear a smartphone dystopia.” The Guardian. https://www.theguardian.com/technology/2017/oct/05/smartphone-addiction-silicon-valley-dystopia

Cigna Corporation. 2021. “The Loneliness Epidemic Persists: A Post-Pandemic Look at the State of Loneliness among U.S Adults.” The Cigna Group. https://newsroom.thecignagroup.com/loneliness-epidemic-persists-post-pandemic-look

Cooper, L. 1968. “An extension of the generalized Weber problem.” *Journal of Regional Science*, 8, no.2: 181-197.

Cortright, J. 2015. “Less in Common.”, City Observatory. https://cityobservatory.org/wp-content/uploads/2015/06/CityObservatory_Less_In_Common.pdf

De Meulenaere, J., Baccarne, B., Courtois, C., and K. Ponnet. 2021. “The development and psychometric testing of the expressive and instrumental Online Neighborhood Network Uses Scale (ONNUS).” *Cyberpsychology: Journal of Psychosocial Research on Cyberspace* 15, no. 3: https://doi.org/10.5817/CP2021-3-8

Dijkstra, E.W. 1959. “A Note on Two Problems in Connexion with Graphs.” *Numerische Mathematik* 1: 269 – 271. https://doi.org/10.1007/BF01386390.

Duke, E., and C. Montag. 2017. “Smartphone addiction, daily interruptions and self-reported productivity.” *Addictive Behaviors Reports*. 6:90-95. https://doi.org/10.1016/j.abrep.2017.07.002

Dunkelman, M. J. 2014. *The Vanishing Neighbor: The Transformation of American Community.* New York: W.W. Norton & Company, Inc.

Esri. n.d. “What is GIS?” Esri. Accessed on May 1, 2024. https://www.esri.com/en-us/what-is-gis/overview

Garret, J.J. 2002. *The Elements of User Experience: User-Centered Design for the Web.* Thousand Oaks, CA: New Riders Publishing.

Singh, S. 2009. “GeoAlchemy: Using SQLAlchemy with Spatial Databases.” GeoAlchemy. https://geoalchemy.readthedocs.io/en/latest/

Github. n.d. “Let’s build from here. The world’s leading AI-powered developer platform.” Github. Accessed on May 1, 2024. https://github.com/

Guest, A.M., and S.K. Wierzbicki. 1999. “Social Ties at the Neighborhood Level: Two Decades of GSS Evidence.” *Urban Affairs Review* 35, no.1: 92-111. https://doi.org/10.1177/10780879922184301

Gunduz, U. 2017. “The Effect of Social Media on Identity Construction.” *Mediterranean Journal of Social Sciences* 8, no.5. https://doi.org/10.1515/mjss-2017-0026.

Fan, D.K., and P. Shi. 2010. “Improvement of Dijkstra’s Algorithm and Its Application in Route Planning.” *Seventh International Conference on Fuzzy Systems and Knowledge Discovery* 4: 1901-1904. https://doi.org/10.1109/FSKD.2010.5569452.

Faron, D. 2002. “Alfred Weber, Theory of Location of Industries 1909 CSISS Classics.” UC Santa Barbara: Center for Spatially Integrated Social Science. Retrieved from https://escholarship.org/uc/item/1k3927t6.

Flask. n.d. “User’s Guide.” Flask. Accessed on May 1, 2024. https://flask.palletsprojects.com/en/3.0.x/#

—. “Application Factories.” Flask. Accessed on May 1, 2024. https://flask.palletsprojects.com/en/2.3.x/patterns/appfactories/

—. “Configuration Handling.” Flask. Accessed on May 1, 2024. https://flask.palletsprojects.com/en/3.0.x/config/

—. “Flask-JWT-Extended’s Documentation.” Flask. Accessed on May 1, 2024. https://flask-jwt-extended.readthedocs.io/en/stable/

—. “Installation.” Flask. Accessed on May 1, 2024. https://flask.palletsprojects.com/en/3.0.x/installation/

—. “Modular Applications with Blueprints.” Flask. Accessed on May 1, 2024. https://flask.palletsprojects.com/en/3.0.x/blueprints/

Fortunati, L. 2005. “Is Body-to-Body Communication Still the Prototype?” *The Information Society* 21: 53 – 61. https://doi.org/10.1080/01972240590895919

Hagber, A., Schult, D., and P. Swart. 2008. “Exploring Network Structure, Dynamics, and Function using NetworkX.” USDOE, Los Alamos National Laboratory. https://www.osti.gov/servlets/purl/960616

Hakimi, S.L. 1965. “Optimum distribution of switching centers in a communication network and some related graph theoretic problems.” *Operations Research* 13, no. 3: 462-475. https://doi.org/10.1287/opre.13.3.462.

Heroku. n.d. “What is Heroku?” Heroku. Accessed on May 1, 2024. https://www.heroku.com/what?utm_source=google&utm_medium=paid_search&utm_campaign=amer_heraw&utm_content=general-branded-search-rsa&utm_term=heroku&gad_source=1&gclid=CjwKCAjwouexBhAuEiwAtW_Zx-KR8McKD-gpu3QP87UR-Lnj1r4PB7DNf3Hlsg6MbGI499SvjuRuZxoCNXYQAvD_BwE

—. “The Heroku CLI.” Heroku. Accessed on May 1, 2024. https://devcenter.heroku.com/articles/heroku-cli

Holt-Lunstad, J., Robles T.F., and D.A. Sbarra. 2017. “Advancing social connection as a public health priority in the United States.” *The American Psychologist* 72, no.6: 517-530. https://doi.org/10.1037/amp0000103

Hruby, K. 2021. "Map Routing." Bachelor Thesis, Charles University. http://hdl.handle.net/20.500.11956/148394

Hunt, M.G., Marc, R., Lipson, C., and J. Young. 2018 “No More FOMO: Limiting Social Media Decreases Loneliness and Depression.” *Journal of Social and Clinical Psychology* 37, no. 10:751. https://doi.org/10.1521/jscp.2018.37.10.751.

Iniguez, A. 2022. “Graph theory and its uses with 5 examples of real-life problems.” Xomnia. https://www.xomnia.com/post/graph-theory-and-its-uses-with-examples-of-real-life-problems/

Jupyter. n.d. “Jupyter – Free software, open standards, and web services for interactive across all programming languages.” Jupyter. Accessed on May 1, 2024. https://jupyter.org/

IntelliJ IDEA. n.d. “IntelliJ IDEA – The leading Java and Kotlin IDE.” JetBrains. Accessed on May 1, 2024. https://www.jetbrains.com/idea/

—. “Configure a Python SDK” JetBrains. Accessed on May 1, 2024. https://www.jetbrains.com/help/idea/configuring-python-sdk.html

Kannan, V., and P. Veazie. 2023. “US Trends in social isolation, social engagement, and companionship – nationally and by age, sex, race/ethnicity, family income and work hours, 2003-2020.” *SSM – Population Health* 21. https://doi.org/10.1016/j.ssmph.2022.101331

Kaspersen, L.B., and N. Gabriel 2008. “The importance of survival units for Norbert Elias's figurational perspective” *The Sociological Review* 56, no. 3: 370–387. https://doi.org/10.1111/j.1467-954X.2008.00795.x

Kim, M., and M. Cho. 2019. "Examining the role of sense of community: Linking local government public relationships and community-building." *Public Relations Review* 45, no. 2: 297 – 306. https://doi.org/10.1016/j.pubrev.2019.02.002.

Lambright, K. 2019. “Digital Redlining: The Nextdoor App and the Neighborhood of Make-Believe.” *Cultural Critique* 103: 84-90. https://doi.org/10.5749/culturalcritique.103.2019.0084

Larsen, M.C. 2022. “Social Media insecurities in everyday life among young adults – an anonymous Jodel disclosure” *Nordisk tidsskrift for pedagogikk og kritikk, Special issue: Digitalisering av utdannings- og oppvekstspraksiser* 8: 298-313. http://dx.doi.org/10.23865/ntpk.v8.4071

Lozinski, L. 2016. “The Uber Engineering Tech Stack, Part I: The Foundation” Uber. https://www.uber.com/blog/tech-stack-part-one-foundation/

Lucid. n.d. “UML Use Case Diagram Tutorial.” Lucidchart. Accessed on May 1, 2024. https://www.lucidchart.com/pages/uml-use-case-diagram

—. “Where seeing becomes doing?” Lucidchart. Accessed on May 1, 2024. https://www.lucidchart.com/pages/

Makridis, C.A., and Wu, C. 2021. "How social capital helps communities weather the COVID-19 pandemic." *PLOS ONE* 16, no. 9. https://doi.org/10.1371/journal.pone.0258021.

Mapbox. n.d. “API Reference.” Mapbox GL JS. Accessed on May 1, 2024. https://docs.mapbox.com/mapbox-gl-js/api/

Masi, C.M., Chen, H.Y., Hawkley, L.C., and J.T. Cacioppo. 2011. “A meta-analysis of interventions to reduce loneliness.” *Personality and social psychology review: an official journal of the Society for Personality and Social Psychology, Inc* 15, no. 3: 219–266. https://doi.org/10.1177/1088868310377394

Maurya, S.P., Ohri, A., and S. Mishra. 2015. “Open-Source GIS: A Review.” Department of Civil Engineering, IIT. https://www.researchgate.net/publication/282858368_Open_Source_GIS_A_Review

Mdn Web Docs. n.d. “Flexbox.” Mdn Web Docs. Accessed on May 1, 2024. https://developer.mozilla.org/en-US/docs/Learn/CSS/CSS_layout/Flexbox

—. “HTML elements reference” Mdn Web Docs. Accessed on May 1, 2024. https://developer.mozilla.org/en-US/docs/Web/HTML/Element

Meruliya, P. 2022. “Benefits of modular programming and how to avoid spaghetti mess.” DhiWise. https://medium.com/dhiwise/benefits-of-modular-programming-and-how-to-avoid-spaghetti-mess-with-dhiwise-4f37212fa074

National Conference on Citizenship. 2011. "Civic Health and Unemployment: Can Engagement Strengthen the Economy?" National Conference on Citizenship. https://ncoc.org/wp-content/uploads/2011/09/2012IssueBrief_CivicHealth_UnemploymentII.pdf

Nextdoor. 2023. “We believe in the possibilities nearby.” Nextdoor. https://about.nextdoor.com/

—. “Research in the Neighborhood.” Nextdoor. https://about.nextdoor.com/research/

OpenStreetMap. 2024. “Welcome to OpenStreetMap.” OpenStreetMap. https://www.openstreetmap.org/#map=4/38.01/-95.84

OSMnx. 2024. “OSMnx 1.9.3 Documentation.” OSMnx. https://osmnx.readthedocs.io/en/stable/#citation

Pattan, I. 2023. “CI/CD Pipeline for Web Application Deployment using Heroku and Github.” Cloudthat. https://www.cloudthat.com/resources/blog/ci-cd-pipeline-for-web-application-deployment-using-heroku-and-github

Pettit, A. D. 2020. “Developing a Web-Based Application for Finding Meeting Points.” Bachelor Thesis, The College of Wooster. https://openworks.wooster.edu/independentstudy/9080

PgAdmin. n.d. “PgAdmin – PostgreSQL Tools.” PgAdmin. Accessed on May 1, 2024. https://www.pgadmin.org/

PostGIS. n.d. “Introduction to PostGIS.” PostGIS. Accessed on May 1, 2024. https://postgis.net/workshops/postgis-intro/installation.html

—. “Spatial Indexing.” PostGIS. Accessed on May 1, 2024. https://postgis.net/workshops/postgis-intro/indexing.html

Postgresql. n.d. “PostgreSQL: The World’s Most Advanced Open-Source Relational Database.” PostgreSQL. Accessed on May 1, 2024. https://www.postgresql.org/

Postman. n.d. “Build APIs together.” Postman. Accessed on May 1, 2024. https://www.postman.com

Prakash, N. 2018. “Origins and Development of Graph Theory prior to 20th Century.” Medium. https://medium.com/@nikhil07prakash/origins-and-development-of-graph-theory-prior-to-20th-century-47543867c909

Putnam, R. 2000. *Bowling Alone: The Collapse and Revival of American Community*. New York: Touchstone.

Python. n. d. “Welcome to Python” Python. Accessed on May 1, 2024. https://www.python.org/

Quos, Y. 2023. “How to build a CRUD API using Python Flask and SQLAlchemy ORM with PostgreSQL” Medium. https://medium.com/@yahiaqous/how-to-build-a-crud-api-using-python-flask-and-sqlalchemy-orm-with-postgresql-7869517f8930

Rao, S., and L. Zhang. 2020. “The Algorithms that make Instacart Roll: How Machine Learning and other Tech tools Guide your Groceries from Store to Doorstep”. *IEEE Spectrum* 58, no. 3: 36-42. https://doi.org/10.1109/MSPEC.2021.9370062.

React. n.d. “The library for web and native user interfaces.” React. Accessed on May 1, 2024. https://react.dev/

Redux. n.d. “Getting Started with Redux.” Redux. Accessed on May 1, 2024. https://redux.js.org/introduction/getting-started

Rong, Q.G., Zhang, X.L., and S. Gu. 2017. "A Systematic Review of Logging Practice in Software Engineering." *24th Asia-Pacific Software Engineering*: 534-539. https://doi.org/10.1109/APSEC.2017.61.

Sachdev, N. 2020. “The Third Generation of Social Networking is Micro and Right in your neighborhood” The Tech Panda. https://www.thetechpanda.com/the-third-generation-of-social-networking-is-micro-and-right-in-your-neighbourhood/30628/

Schumaker, E. 2011. " 'I care about it': Sen. Chris Murphy's battle against loneliness" Politico. https://www.politico.com/news/2023/11/05/sen-chris-murphy-wants-to-help-you-make-friends-00125372

Singh, S. 2023. “The Algorithms Behind the Working of Google Maps” Medium. https://medium.com/@sachin.singh.professional/the-algorithms-behind-the-working-of-google-maps-73c379bcc9b9#:~:text=Google%20Maps%20essentially%20uses%20two,defined%20by%20edges%20and%20vertices.

Sharma, N. 2023. “70 Geospatial Python Libraries” Medium. https://medium.com/@ns_geoai/70-geospatial-python-libraries-54604d815a7b

Shaw, J.G., Farid, M., Noel-Miller, C., Joseph, N., Houser, A., Asch, S. M., Bhattacharya, J., L. Flowers. 2017. “Medicare Spends More on Socially Isolated Older Americans.” AARP Public Policy Institute. https://capitolhillvillage.org/wp-content/uploads/2018/11/medicare-spends-more-on-socially-isolated-older-adults.pdf

Solon, O. 2017. “Ex-Facebook President Sean Parker: Site Made to Exploit Human ‘vulnerability’” The Guardian.

https://www.theguardian.com/technology/2017/nov/09/facebook-sean-parker-vulnerability-brain-psychology

Tufte, E. 1983. *The Visual Display of Quantitative Information*. Cheshire, CT: Graphics Press.

Uhls, Y.T., Ellison, N.B., and K. Subrahmanyam. 2017. "Benefits and costs of social media in adolescence." *Pediatrics* 140, no. 2: 67-70. https://doi.org/10.1542/peds.2016-1758E.

U.S. Department of Commerce. 2023. "Community Resilience." National Institute of Standards and Technology. https://www.nist.gov/community-resilience

U.S. Department of Health and Human Services. 2023. *Our Epidemic of Loneliness and Isolation: The U.S Surgeon General's Advisory on the Healing Effects of Social Connection and Community*. V. Murthy. Office of the Surgeon General. https://www.hhs.gov/sites/default/files/surgeon-general-social-connection-advisory.pdf

Valtorta, N.K., Kanaan, M., Gilbody, S., Ronzi, S., and B. Hanratty. 2016. "Loneliness and social isolation as risk factors for coronary heart disease and stroke: systematic review and meta-analysis of longitudinal observational studies" *Heart* 102, no. 13:1009-1016. https://doi.org/ 10.1136/heartjnl-2015-308790.

Vaidhyanathan, S. 2019. *Antisocial Media: How Facebook Disconnects Us and Undermines Democracy*. Oxford: Oxford University Press.

Vogel, P. 2021. "Designing Openness-Infusing Socio-Technical Artifacts", PH. D Diss, University of Hamburg. https://ediss.sub.uni-hamburg.de/bitstream/ediss/9019/1/Dissertation_Vogel_Pascal.pdf

Vollman, M. 2018. "Hyperlocal Neighborhood Networks: Building Social Capital and Empowering Local Urban Communities" UrbanNet. https://www.urbanet.info/hyperlocal-neighbourhood-networks/

Wang, S., Feng, X., Murray, A., and Y. Zeng. 2018. "A context-based geoprocessing framework for optimizing meetup location of multiple moving objects along road networks", *International Journal of Geographical Information Science* 32, no. 7: 1368-1390. https://doi.org/10.1080/13658816.2018.1431838.

Weissbourd, R., Batanova, M., Lovison, V., and E. Torres. 2021. "Loneliness in America" Making Caring Common Project, Harvard Graduate School of Education. https://static1.squarespace.com/static/5b7c56e255b02c683659fe43/t/6021776bdd04957c4557c212/1612805995893/Loneliness+in+America+2021_02_08_FINAL.pdf

Xu, Z., and H.A. Jacobsen. 2010. "Processing proximity relations in road networks." *Proceedings of the 20120 ACM SIGMOD international conference on management of data:* 243-254. https://doi.org/10.1145/1807167.1807196.

Yildirim, G. 2023. "Routing Algorithms as an Application of Graph Theory" M.S. Thesis, Middle East Technical University. https://hdl.handle.net/11511/102142

Yongmei, R., Linghong, H., and M. Yongqing. 2015. "Logistics Distribution Route Optimization Method for Peach Products Transport." *2015 Seventh International Conference on Measuring Technology and Mechatronics Automation:* 609 – 612. https://doi.org/10.1109/ICMTMA.2015.153

Yoon, S., Ko, D., Koh, S., Nam, H., and S. An. 2011. "PR-RAM: The Page Rank Routing Algorithm Method in Ad-hoc wireless networks." *IEEE Consumer Communications and Networking Conference:* 96-100. https://doi.org/10.1109/CCNC.2011.5766654

Zhang, N., and Z. Huang. 2011. "Evaluation and Optimization of Bus Route Network in Wuhan China" *The Seventh Advanced Forum on Transportation of China:* 140-148. https://doi.org/10.1049/cp.2011.1392.

Appendix

Application name	Self-description	Link
Konnect	Make local friends	https://apps.apple.com/us/app/konnect-make-local-friends/id1182587961
YikYak	Anonymously connect in college	https://yikyak.com/
Bump Grinnell	Connect with your Grinnell communities	https://bump-bump-bumping.en.aptoide.com/app
Meetup	Find local groups	https://www.meetup.com/
Nextdoor	An app for neighborhoods	https://nextdoor.com/
PartyWith	Find people nearby who want to party	https://www.instagram.com/partywithapp/?hl=en
oOlala	A local hangout app	https://www.instagram.com/oolalaapp/
MeetnGreet	Make new friends nearby	https://www.facebook.com/sdmeetngreet
Spontaneous	Hangout with family and friends nearby	https://apps.apple.com/lb/app/sponty-spontaneous-gatherings/id1558525532
Localmind	Know what's happening anywhere	https://www.facebook.com/localmind/
Circle	The local app	https://play.google.com/store/apps/details?id=com.circleme&hl=en_US
Patch	Everything local	https://patch.com/
Mapbuzz	Meet people nearby	https://www.mapbuzz.com/
Nebenan	Social networking with neighbors	https://nebenan.de/
Hoplr	Neighborhood social networking	https://hoplr.com

OneRoof	Social platform for apartment residents	https://www.oneroofapp.com/
NearGroup	Private neighborhood networking app	https://neargroup.me/
PublicNext	Connects the neighborhood	https://publicnext.com/
SOSAFE	A citizen network	https://play.google.com/store/apps/details ?id=cl.sosafe.panicbuttonandroid.app&hl=en&gl=US
Therr.app	Local social networking	https://www.therr.app/
Playsee	Location focused social media	https://playsee.co/
Shoelace	Google's hyper local social networking application	https://techcrunch.com/2020/04/29/google-is-shutting-down-shoelace-the-social-app-youve-probably-never-heard-of/
SimplyLocal	Neighborhood public noticeboard	https://www.simplylocal.app/
Jodel	Your hyper local community	https://jodel.com/en/
Hobbin	Meet friends near me	https://apps.apple.com/us/app/hobbin-meet-friends-near-me/id1614610374
Localysis	Connect you to people in your desired location	https://hi-in.facebook.com/localisys/videos/localysis-is-a-universal-mobile-handy-app-that-connects-you-to-people-services-e/686129955107837/
WithLocals	Find local things to do near you	https://apps.apple.com/us/app/withlocals-tours-travel-app/id655695313
OneRoof	Meeting your neighbors made easy	https://www.oneroofapp.com/
Neighbors by Ring	Join the neighborhood	https://apps.apple.com/us/app/neighbors-by-ring/id1218902777