Design and Implementation of Security-Conscious, Location-Sharing in a Geosocial Network

Thesis submitted by

Manas Pratim Biswas (002011001025)
Anumoy Nandy (002011001121)
Kunal Pramanick (302111001005)

under the guidance of

Dr. Munmun Bhattacharya

in partial fulfilment of the requirements
for the award of the degree of

Bachelor of Engineering in Information Technology

Department Of Information Technology
Faculty of Engineering & Technology
Jadavpur University

2020 - 2024

This page has been intentionally left blank

BONDAFIDE CERTIFICATE

This is to certify that this project titled ***"Design and Implementation of Security-Conscious, Location-Sharing in a Geosocial Network"***, submitted to the **Department of Information Technology, Jadavpur University, Salt Lake Campus, Kolkata**, for the award of the degree of **Bachelor of Engineering**, is a bonafide record of work done by **Manas Pratim Biswas (Regn. No.: 153760 of 2020-2021)**, **Anumoy Nandy (Regn. No.: 153823 of 2020-2021)** and **Kunal Pramanick (Regn. No.: 159991 of 2021-2022)**, under my supervision.

Countersigned By:

Dr. Munmun Bhattacharya
Assistant Professor
Information Technology

Prof. Bibhas Chandra Dhara
HOD, Information Technology
Jadavpur University

Acknowledgements

We would like to express our sincere gratitude to Dr. Munmun Bhattacharya for allowing us to pursue our final-year project under her supervision. In addition to the meticulous research guidance, her encouraging support for our ideas as well as the constructive criticisms during our experimental and initial developmental stages of the project would not have been possible.

We would also like to thank the entire Department of Information Technology including all the professors, lab assistants and non-teaching staff for being extremely cooperative with us throughout the developmental stages of the project.

Above all, we thank our parents for their support and encouragement in our academic pursuits.

Department of Information Technology
Jadavpur University
Salt Lake Campus, Kolkata

Manas Pratim Biswas Anumoy Nandy Kunal Pramanick

JADAVPUR UNIVERSITY

Dept. of Information Technology

Vision:

To provide young undergraduate and postgraduate students a responsive research environment and quality education in Information Technology to contribute in education, industry and society at large.

Mission:

M1: To nurture and strengthen the professional potential of undergraduate and postgraduate students to the highest level.

M2: To provide international standard infrastructure for quality teaching, research and development in Information Technology.

M3: To undertake research challenges to explore new vistas of Information and Communication Technology for sustainable development in a value-based society.

M4: To encourage teamwork for undertaking real life and global challenges.

Program Educational Objectives (PEOs):

Graduates should be able to:

PEO1: Demonstrate recognizable expertise to solve problems in the analysis, design, implementation and evaluation of smart, distributed, and secured software systems.

PEO2: Engage in the engineering profession globally, by contributing to the ethical, competent, and creative practice of theoretical and practical aspects of intelligent data engineering.

PEO3: Exhibit sustained learning capability and ability to adapt to a constantly changing field of Information Technology through professional development, and self-learning.

PEO4: Show leadership qualities and initiative to ethically advance professional and organizational goals through collaboration with others of diverse interdisciplinary backgrounds.

Mission - PEO matrix:

Ms/ PEOs	M1	M2	M3	M4
PEO1	3	2	2	1
PEO2	2	3	2	1
PEO3	2	2	3	1
PEO4	1	2	2	3

(3 – Strong, 2 – Moderate and 1 – Weak)

Program Specific Outcomes (PSOs):

At the end of the program a student will be able to:

PSO1: Apply the principles of theoretical and practical aspects of ever evolving Programming & Software Technology in solving real life problems efficiently.

PSO2: Develop secure software systems considering constantly changing paradigms of communication and computation of web enabled distributed Systems.

PSO3: Design ethical solutions of global challenges by applying intelligent data science & management techniques on suitable modern computational platforms through interdisciplinary collaboration.

This page has been intentionally left blank

Abstract

The recent advancements in mobile and internet technology, coupled with cheap internet data, have witnessed an exponential increase in the usage of internet and people connecting via social media networks. However, the traditional social networking sites such as *Facebook*, *Instagram* or *Twitter* have had witnessed multiple allegations of capturing and selling the user data for their corporate and business purposes. Most importantly, a breach in the user's location data can expose sensitive information regarding that user's frequent places of visit, acquaintances they meet with, food habits and political preferences.

Our project explores the applicability and implementation of a Privacy Network as a scalable Geosocial Networking website. A user-centric design for a secure location sharing is implemented into our project. The final product, *TraceBook*, is a web application that allows the users to register into our website with their details, add or remove friends and most importantly, decide and set their visibility and other privacy parameters very precisely to ensure a secure and robust social media usage. Even more, the users can query their friends based on parameters including but not limited to *age*, *gender*, *college* and *distance*.

The user auth and data is handled by a MongoDB backend, and the user location attributes are handled by a Postgres backend. PostGIS along with PostgreSQL is utilized to create custom SQL queries to fetch the user location details on-demand. Real-time location broadcasting and multicasting is achieved by upgrading the *HTTP* protocol to *WebSocket* protocol and utilizing raw *WebSockets* natively available in the client's browser. A separate WebSocket backend server handles all the WebSocket connections from the client side. Services such as *Vercel* and *Render* are utilized to host the frontend and backend servers. A complete documentation of all the backend API end-points, in compliance with the standardized OpenAPI specifications, is available as a Swagger documentation.

However, scaling WebSockets is challenging. Therefore, a *distributed Redis Pub-Sub* based architecture along with a *HAProxy load balancer* is proposed. Moreover, a *change-data-capture* mechanism using *Apache Kafka* is proposed to keep the MongoDB and PostgreSQL databases consistent and in synchronization with each other.

Keywords: typescript, websockets, mern, postgresql, postgis, redis, haproxy, kafka, social-network-analysis

Contents

List of Tables

List of Figures

Abbreviations

OSN	Online Social Network
POI	Position Of Interest
MERN	MongoDB, Express, React, Node
API	Application Programming Interface
IO	Input/Output
CLI	Command Line Interface
HTTP	Hyper Text Transfer Protocol
`req-res`	Request-Response
JSON	JavaScript Object Notation
URL	Uniform Resource Locator
CRUD	Create, Read, Update, Delete
MVVM	Model-View-View-Model
DOM	Document Object Model
UI	User Interface
`ws`	WebSocket Server
`ts-frontend`	Web/Client User Interface
`ts-backend`	MongoDB Server
`loc`	Postgres Server
`tsc`	TypeScript Compiler
`Pub-Sub`	Publisher-Subscriber
ORM	Object Relational Mapping
ERD	Entity Relationship Diagram
DNS	Domain Name System
SPOF	Single Point Of Failure

This page has been intentionally left blank

Chapter 1

Introduction

This chapter gives a basic introduction about the background of a Privacy Network, its necessity and inception. The later subsections give the consequent flow of the project report.

1.1 Geosocial Networking

Geosocial networking[1] is a type of social networking that integrates geographic services and capabilities such as geolocation to enable additional social dynamics. This technology allows users to connect and interact based on their physical locations, enhancing the social networking experience by facilitating real-time, location-based interactions. Popular applications of geosocial networking include location-based services like check-ins, geotagging, and finding nearby friends or events. These functionalities are supported by technologies such as GPS, Wi-Fi positioning, and cellular triangulation. By leveraging these technologies, geosocial networking can provide users with personalized content and recommendations based on their current location, thereby enriching the user experience. However, it also raises significant privacy concerns, as the collection and sharing of location data can lead to potential risks such as unwanted tracking and profiling. Consequently, there is a growing emphasis on developing secure geosocial networking applications that prioritize user privacy and data protection, incorporating advanced encryption and user consent mechanisms to safeguard sensitive location information.

The past decade has seen unprecedented advancements in mobile and internet technologies, resulting in an exponential increase in internet usage and social media connectivity. Affordable internet data has made the online world more accessible than ever before, fundamentally transforming how people interact and communicate. Social media platforms such as Facebook, Instagram, and Twitter have become integral to daily life, providing users with the means to connect, share, and engage with others on a global scale. These platforms, however, have also come under intense scrutiny for their handling of user data, with numerous allegations of data misuse for corporate gain.

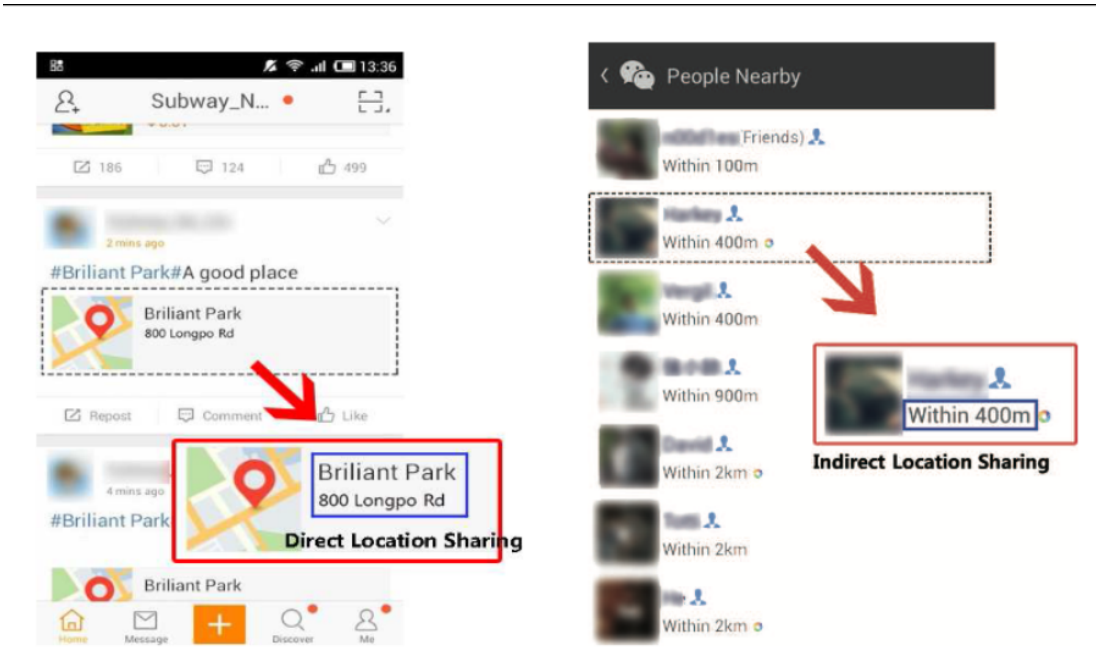
1.1.1 Security Vulnerabilities in Geosocial Networking

The convenience and connectivity offered by social media come at a significant cost to user privacy. Traditional social networking sites have been accused of capturing and monetizing user data, leveraging personal information for targeted advertising and other business

purposes. This practice has raised serious concerns about data security and user consent, highlighting a critical need for more transparent and ethical data management practices.

Among the various types of data collected, location data is particularly sensitive. A breach in location data can reveal detailed insights into a user's daily routines, including frequent places of visit, social circles, dietary preferences, and even political affiliations. For instance, if a social media platform tracks a user's location, it can infer whether the user visits specific types of restaurants, attends political rallies, or frequents particular neighborhoods. Such information, if exposed, can lead to a myriad of privacy invasions and security risks, including unwanted surveillance, targeted harassment, and identity theft.

Direct and indirect location sharing pose significant risks. Attackers can infer user demographics, such as age, gender, and education, from shared location profiles. Many social networks offer geolocation tags and check-in services, which attackers exploit by crawling web pages to extract location information and Points of Interest (POIs). For instance, direct location sharing on Online Social Networks (OSNs) exposed 16% of users' real POIs, while indirect sharing, like geolocation tags, exposed up to 33%.



(a) Direct location sharing on Weibo (b) Indirect location sharing on Wechat

Figure 1.1: Direct & Indirect Location Sharing

Intentional location spoofing is another major concern. Privileged insiders can use apps like *FakeGPS*[2] to provide fake locations. We address this issue by ensuring that location updates and friends' location queries are securely encrypted end-to-end. Sharing location with a large number of friends may also pose security hazards; hence, we allow users to set distance thresholds for better control over location sharing.

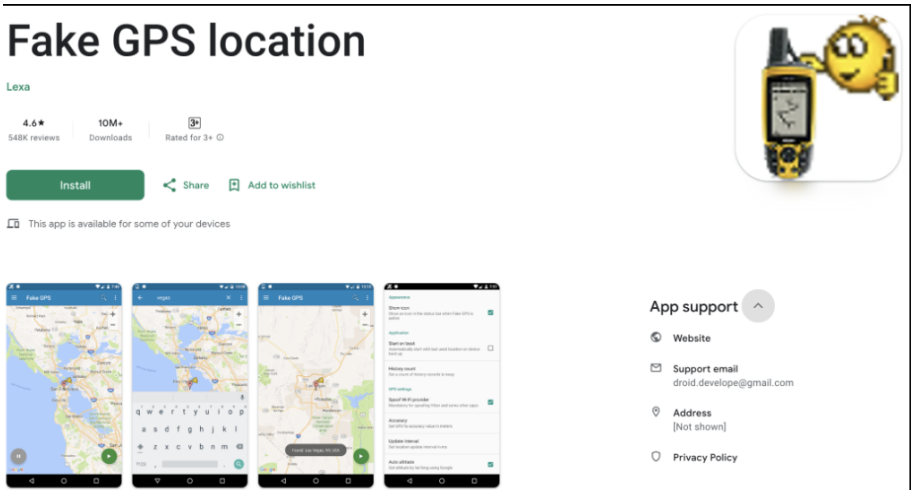


Figure 1.2: Location Spoofing

A three-week real-world experiment[3] with 30 participants revealed alarming results regarding location sharing on OSNs. The study found that direct location sharing exposed 16% of users' real Points of Interest (POIs), while indirect sharing exposed even more, reaching 33%. This underscores the critical need for robust privacy measures in geosocial networking applications.

The exploitation of user data by social media giants has prompted a growing public outcry and a demand for greater privacy protections. High-profile incidents, such as the Cambridge Analytica scandal[4], have underscored the potential for misuse of personal data and the far-reaching consequences of such breaches. These events have sparked a broader conversation about the ethical responsibilities of technology companies and the need for robust regulatory frameworks to safeguard user privacy.

1.1.2 Addressing Privacy Tradeoffs

In response to these concerns, there has been a surge in the development of privacy-centric social networking alternatives. These platforms prioritize user data protection and transparency, offering users greater control over their personal information.

In our current approach, we utilize two security parameters that the user can fine tune to filter his/her location. The user can set their own visibility to either *on/off* and can also *limit their visibility* till a certain range of distance. Thereby, The user can control the location shared to the rest of the users in the social network. We utilize a variable called ***isVisible*** which the user can update to control his/her visibility. *isVisible true* implies that the user wishes to share his/her locations, and *isVisible false* denotes that the user does not wish to share his/her location. The idea is that a particular pair of user can have four possibilities for the privacy parameters, as shown below:

Table 1.1: Privacy Filtration

Connection	*isVisible*	*VisibilityLevel*
Friend	True	High (Marker)
Friend	False	Medium (Blue Patch)
Non-Friend	True	Low (Red Patch)
Non-Friend	False	None

- A **Marker** shows the exact location of the user's friend, since the user's friend as allowed his/her location to be shared.
- A **Blue Patch** or a Medium visibility means that the exact location will not be shared to the user's friend, rather a Blue Patch would be shown which denotes the user to be present **within 10 *kms* radius.**
- A **Red Patch** or a Low visibility means that the exact location of the stranger will not be shared to the user, rather a Red Patch would be shown which denotes the user to be present **within 20 *kms* radius.**

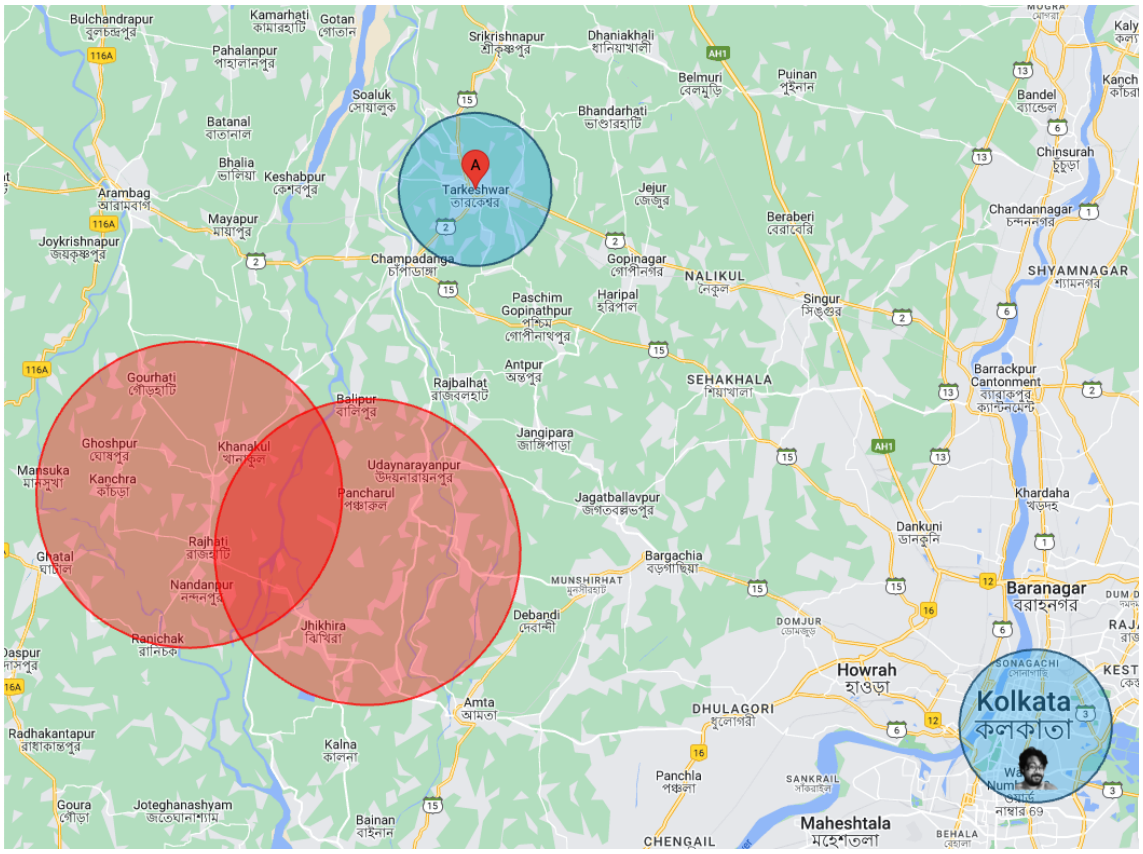


Figure 1.3: Privacy Filtration

1.2 Report Outline: The Privacy Network Approach

The report is organized as follows:

Chapter 2 presents the existing literature on privacy preserving schemes and techniques of location sharing for online mobile online social networks. Further, we summarize the features and technicalities of a similar web application based social networking site called *Loopt* that aimed to solve the similar issue.

Chapter 3 explores the architecture and the implementation details of a Privacy Network - *TraceBook*. We briefly discuss the history and relevance of the MERN stack followed by a thorough scrutiny of the proposed architecture and the consequent coding implementation, CI/CD pipelines and deployment of the web application.

Chapter 4 and Chapter 5 provides a summary of the implementation outputs through a conclusion and discusses various potential future features and proposes advanced architectural to attain a distributed and scalable system.

Chapter 2

Literature Review

This chapter starts by briefly covering the relevant works in designing privacy preserving and efficient location sharing schemes for mobile online social networks. We discuss the case studies and vulnerabilities found through previous works. This is followed by the introduction of a product - Loopt, now discontinued, that tried to solve a very similar problem as that of ours.

2.1 Related Works

In the domain of mobile Online Social Networks (mOSNs), privacy and security have garnered significant research interest. Various privacy-preserving schemes have been proposed, each with distinct advantages and limitations. Early research primarily focused on information privacy, user anonymity, and location privacy. Techniques for maintaining location anonymity often involve encrypting the current location before transmission. K-anonymity methods obfuscate user locations, while dummy locations are used to mask real ones. Location encryption and pseudonym methods, such as mix zones and m-unobservability, are other prominent approaches.

K-anonymity for location privacy involves obfuscating a user's actual location, as demonstrated by researchers like Gruteser and Grunwald[5]. The use of dummy locations, where fake locations are sent along with the real one, has been explored by Kido et al.[6]. Location encryption methods, such as those by Khoshgozaran et al.[7], offer another effective means of protecting location privacy. Pseudonym-based methods, mix zones, and m-unobservability, as explored by Beresford and Stajano[8], have also been developed.

Rahman et al.[9] proposed privacy context obfuscation based on various location parameters to achieve location obscurity. The concept of location sharing with privacy protection in online social networks was initially addressed by SmokeScreen, which facilitated location sharing between friends and strangers. This approach was later enhanced by Wei et al.[10] with MobiShare, which separated social and location information into Social Network Services (SNS) and Location-Based Services (LBS), respectively. However, MobiShare faced the issue of LBS potentially revealing social network topologies during queries.

Li et al.[11] advanced this concept with MobiShare+, introducing dummy queries and private set intersection to prevent LBS from accessing users' social information. BMobiShare

further improved transmission efficiency by using a Bloom Filter instead of the private set intersection method, though it incurred high computation costs.

To counter insider attacks, Li et al.[12] proposed a multiserver location-sharing system in 2015, enhancing security at the cost of increased resource demands and inefficiency. These systems, relying on third-party location servers, are susceptible to collusion between LBS and SNS, leading to potential social information exposure and high transmission and storage costs.

Recently, Xiao et al.[13] introduced CenLocShare, which combines SNS and LBS into a single server, reducing communication and storage costs while enhancing user privacy protection. This amalgamation addresses the inefficiencies and security concerns of previous multiserver approaches, offering a more streamlined and secure solution for privacy-preserving location sharing in mOSNs.

Existing solutions however, typically follow two main approaches: using separate servers for location-based and social network information, which incurs high storage and communication overhead, or integrating both into a single server, which may lead to server bottlenecks and vulnerabilities to security attacks. Bhattacharya et al.[14]. proposed a novel privacy-preserving, secure, and efficient location-sharing scheme for mOSNs that addresses these issues. The proposed scheme ensures efficient and flexible location updates, sharing, and queries among social friends and strangers. Security validation is performed through a random oracle-based formal security proof, Burrows-Abadi-Needham (BAN) logic-based authentication proof, and informal security analysis. Additionally, the scheme's security is verified using ProVerif 1.93. Experimental implementation and evaluation demonstrate the scheme's efficiency and practicality.

2.2 Loopt

Loopt, Inc. was an American company headquartered in Mountain View, California. It provided a service for smartphone users to selectively share their location with others. They created an interoperable social-mapping service that allowed individuals to use their location to discover the real world around them – enabling them to find and enjoy the people, places, and events that mean the most right here just by using their mobile phones[15].

Founded in 2005, Loopt received initial funding from Y Combinator[16]. It secured Series A and B financing led by Sequoia Capital and New Enterprise Associates. Thereafter, Loopt partnered with Boost Mobile, Sprint, and Verizon to expand its reach. The iPhone app, Loopt Mix[17], enabled users to find and meet new people nearby. In 2012, Loopt was acquired by Green Dot Corporation for $43.4 million.

Figure 2.1: Loopt Inc

Features:

- **Location Sharing**: Loopt allowed users to share their real-time location with friends and family.
- **Privacy Controls**: Users could customize who could see their location, ensuring privacy.
- **Cross-Platform Support**: The service worked across major mobile operating systems.
- **Integration with Social Networks**: Users could link Loopt with Facebook and Twitter.
- **Proximity-Based Messaging**: Users could send messages and photos based on their location1

Figure 2.2: Loopt: The Geo Social Network

It is noteworthy to mention that Loopt was among the pioneers of location-based social mapping services. Its innovative approach influenced subsequent location-sharing apps and services.

Chapter 3

System Architecture

In this chapter, we go through a brief overview and history of the MERN stack and then have a detailed discussion about the architecture of TraceBook - the frontend, backend, deployment and scaling.

3.1 MERN Stack with TypeScript

MERN stands for *MongoDB*, *Express.js*, *React.js* and *Node.js*. With *MongoDB* as the Database, *Express.js* acts a web server framework (integrated with Node.js), React.js is the web client library, and Node.js is the server-side JavaScript runtime. It helps developers to develop Web apps based solely on full-stack JavaScript. Due to the same JavaScript platform in both the frontend and the backend, uniformity in the codebase is maintained. Naturally, the stack is supported by a vast number of open-source packages and a dedicated community for programmers to increase scalability and maintain software products.

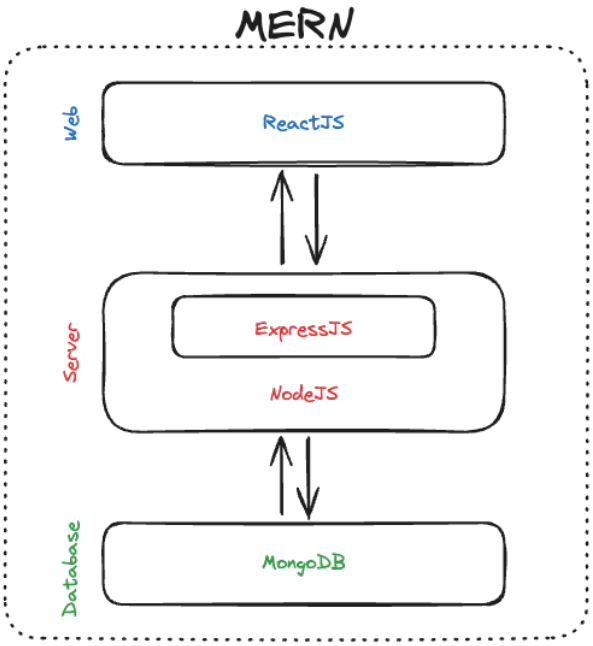


Figure 3.1: MERN Stack Architecture

3.1.1 Node

We are all familiar with the Google Chrome web browser. A typical web browser like Google Chrome runs HTML, CSS and JavaScript to render a website. The JavaScript code's execution actually happens by the virtue of an engine located inside the Chrome browser called ***V8***. Written in C++, V8 has proven to be an extremely powerful engine that can interpret and execute JavaScript very quickly.

Recognizing the robust capabilities of the V8 engine, the founders envisioned decoupling V8 from Chrome to develop a platform capable of executing JavaScript code on the server side. This vision materialized in 2009 when **Ryan Dahl** created ***Node.js***[18].

Node.js serves a role analogous to that of the *Java Virtual Machine* (JVM), functioning as a platform and runtime environment for applications. While the JVM executes *bytecode*, Node.js directly interprets and executes JavaScript code. In the JVM ecosystem, developers typically write source code in high-level languages such as Java, Scala, Groovy, or Kotlin, which is then compiled into bytecode. Conversely, Node.js natively understands and executes JavaScript, enabling developers to write application code directly in JavaScript. Additionally, languages that compile into JavaScript, such as TypeScript, can be utilized within the Node.js environment.

Non-Blocking programming in Node.js - The Node.js platform is versatile, perhaps not because of the V8 engine or the ability to support the JavaScript language, but in the Non-Blocking style of programming. Operations related to Non-Blocking in Node.js are mostly related to IO, such as reading and writing data to disk or calling Web APIs, for example. Furthermore, the use of Non-Blocking operations makes applications written on a Node.js platform capable of using computational resources (CPU) most efficiently.

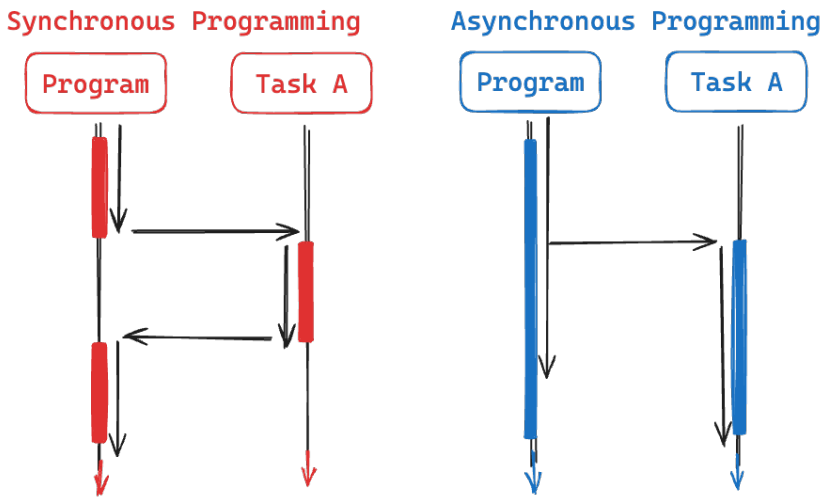


Figure 3.2: Synchronous vs Asynchronous Programming

In the creation of the Non-Blocking mechanism, Node.js applications operate according to the Event Loop design pattern. The illustration below explains this design in more detail:

- **Event Queue**: The Event Queue functions as a repository for storing events, which are specific processes within the program. Each event includes metadata that identifies the event type and its associated data. As a queue structure, the Event Queue operates on a First In, First Out (FIFO) principle. This ensures that events are processed in the order they were enqueued, maintaining the sequence of operations.
- **Main Thread**: The Main Thread is the primary thread responsible for the execution and termination of the program. It handles the computational processing of events retrieved from the Event Queue. This thread is the sole thread within the Node.js application that developers directly control, underscoring the single-threaded nature of Node.js applications. Consequently, developers are relieved from managing concurrency issues that are prevalent in multithreaded environments such as Java.
- **Node API**: The Node API is responsible for handling input/output (IO) operations through a multithreaded mechanism. Each completed IO operation generates a result that is encapsulated as an event and subsequently placed into the Event Queue for processing.

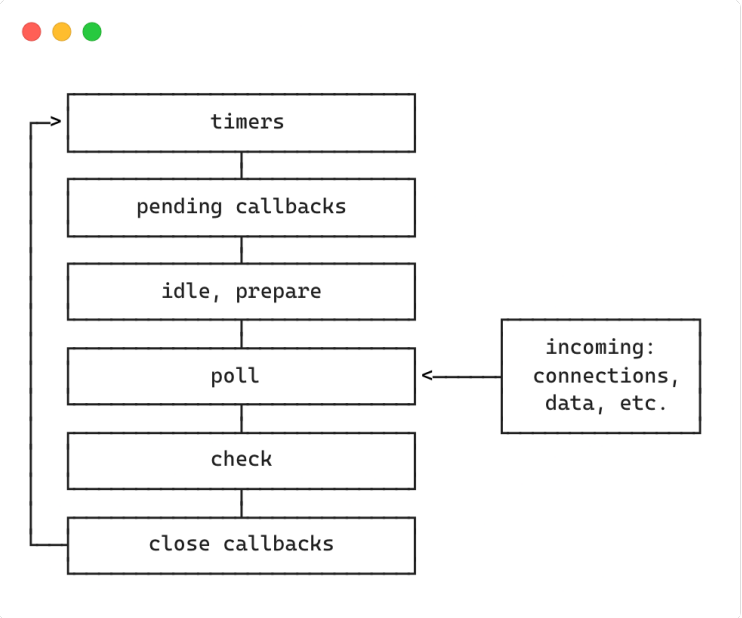


Figure 3.3: JavaScript Event Loop
The Event Loop[19] is a crucial concept for understanding asynchronous programming in JavaScript

With the above three components, the behavior of a Node.js application can be described as follows:

- The Main Thread executes the computational instructions as defined in the source code. When encountering IO operations, the Main Thread initiates a non-blocking call to the Node API and proceeds with executing subsequent commands without waiting for the IO operation to complete.
- Upon receiving a request from the Main Thread, the Node API handles the IO operations using multiple threads. Once an IO operation is completed, the Node API packages the result as an event and enqueues it into the Event Queue.
- The Main Thread processes events from the Event Queue sequentially. In the Main Thread, the code designed to handle these events is typically implemented as callbacks.

This process is iterative, creating a continuous cycle of event handling within the application. Consequently, the programming paradigm is oriented towards event-driven architecture rather than traditional sequential execution.

Node Package Manager - *npm, Yarn, and pnpm* are some of the dependency managers commonly utilized to support the development ecosystem of Node.js. With the vast majority of libraries available on `npm`, integrating these libraries is straightforward via the `npm` install command, facilitating efficient package management.

`npm` operates primarily in two significant roles:

- **Software Registry**: `npm` functions as an extensive software registry used widely for publishing open-source Node.js projects. It serves as an online platform where users can publish and share various tools and libraries written in JavaScript. As of April 2020, the `npm` registry hosts over 1.3 million code packages, making it a crucial resource for developers.
- **Command-Line Interface (CLI)**: `npm` also provides a robust command-line interface that facilitates interaction with online platforms such as servers and web browsers. This CLI utility assists in package installation, uninstallation, version management, and server-driven management. Additionally, `npm` manages the dependencies required to run *Node.js* projects efficiently.

Architectures of `npm`, `Yarn`, and `pnpm`:

- **npm**: The architecture of `npm` is designed around a centralized registry where all packages are stored and accessed. It follows a simple, straightforward approach where packages are installed in a *node_ modules* directory within the project structure. `npm` supports both global and local package installations and includes a p*ackage-lock* file to manage dependency versions precisely.
- **Yarn**: `Yarn` offers an alternative architecture with a focus on speed, reliability, and security. It introduces a *lockfile* (`yarn.lock`) to ensure consistent dependency installations across different environments. `Yarn` uses a deterministic algorithm to install

packages, reducing installation times and preventing mismatched versions. It also supports offline caching, allowing packages to be installed without an active internet connection after the first download.

- **pnpm** pnpm's architecture is designed to optimize disk space and improve performance. Unlike `npm` and `Yarn`, `pnpm` creates a single, centralized store for all packages on the system, and hard links are used to reference packages in individual projects. This approach avoids duplication of dependencies and speeds up installations. `pnpm` also ensures that the same package version is not installed multiple times, saving storage space and reducing redundancy.

3.1.2 Express

Express.js is a widely-adopted, open-source web application framework for Node.js, utilized extensively in commercial applications. Its robust community support and comprehensive documentation make it a reliable choice for developers, enabling the creation of projects ranging from small-scale applications to large enterprise solutions.

Express.js enhances the capabilities of Node.js by providing a wealth of support packages and additional features, which streamline the development process without compromising performance. Many renowned applications built on the Node.js platform leverage *Express.js* as a core component, highlighting its efficiency and effectiveness.

History and Development

Express.js was initially created by **TJ Holowaychuk** and first released on May 22, 2010, with version 0.12. In June 2014, *StrongLoop* took over the project management rights[20]. Subsequently, IBM acquired *StrongLoop* in September 2015[21], and by January 2016, *Express.js* management transitioned to the Node.js Foundation, ensuring its continued development and integration with the Node.js ecosystem.

Routers and Middleware

In web development, routers and middleware play crucial roles in managing HTTP requests and enhancing application functionality.

- **Routers**: A router is responsible for defining and handling the routes (endpoints) in a web application. It maps incoming HTTP requests to specific handlers based on the URL and HTTP method (`GET, POST, PUT, DELETE`, etc.). Routers help organize application logic and enable modular development by separating route definitions and their associated handlers.

- **Middleware**: Middleware functions are a series of functions that execute sequentially during the request-response cycle. They have access to the request object (`req`), the response object (`res`), and the next middleware function in the application's request-response cycle. Middleware can perform various tasks, such as request logging, authentication, parsing request bodies, handling errors, and more. By using middleware,

developers can extend the functionality of their applications in a modular and reusable manner.

Express.js as a Router

Express.js acts as a powerful router, enabling developers to define routes using a simple and intuitive syntax. Here is how *Express.js* handles routing:

- **Route Definition**: Developers can define routes using methods corresponding to HTTP verbs (`app.get()`, `app.post()`, `app.put()`, `app.delete()`, etc.). Each route can have a handler function that processes incoming requests and sends appropriate responses.
- **Route Parameters**: *Express.js* supports dynamic route parameters, allowing developers to capture values from the URL and use them within route handlers.
- **Modular Routing**: *Express.js* enables modular routing by allowing the creation of multiple router instances. Each instance can define a subset of routes, which can be mounted to specific path prefixes, improving code organization and maintainability.

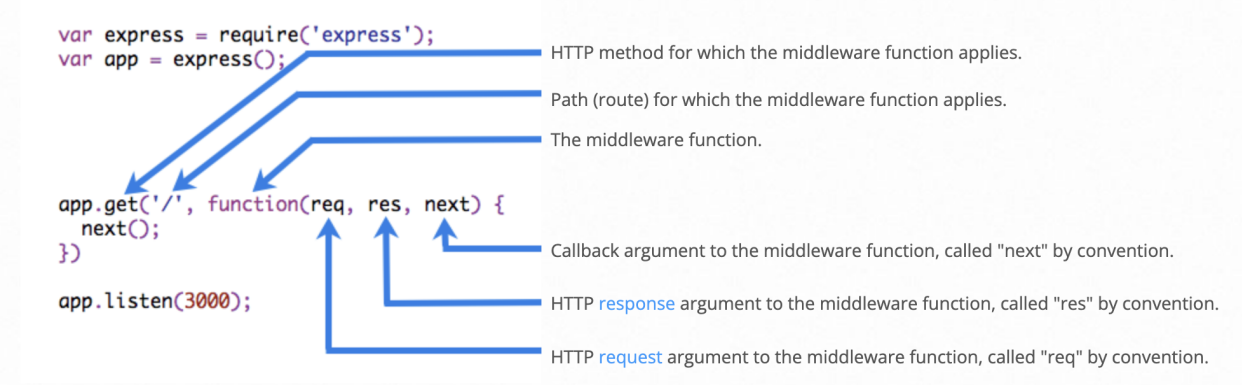


Figure 3.4: Anatomy of an Express Middleware
Middlewares[22] are used for handling the different routing of the webpage and it works between the request and response cycle

Express.js and Middleware

Express.js excels in middleware support, offering a flexible and powerful way to handle the request-response cycle:

- **Built-in Middleware**: *Express.js* includes built-in middleware for common tasks such as serving static files (express.static), parsing JSON and URL-encoded request bodies (`express.json`, `express.urlencoded`), and more.
- **Third-party Middleware**: Developers can leverage a vast ecosystem of third-party middleware to add functionality like session management, authentication, input validation, logging, and error handling.

- **Custom Middleware**: *Express.js* allows developers to create custom middleware functions tailored to their specific needs. These functions can perform a variety of tasks, from modifying the request and response objects to terminating the *request-response* cycle or passing control to the next middleware function.

The combination of robust routing capabilities and extensive middleware support makes *Express.js* an extremely suitable tool for building scalable and maintainable web applications on the Node.js platform.

3.1.3 MongoDB

MongoDB, developed by MongoDB Inc., was initially created in October 2007 as part of a *Platform as a Service* (PaaS) product akin to Windows Azure and Google App Engine. It was subsequently released as an open-source project in 2009.

MongoDB is a document-oriented NoSQL database, distinguishing itself from traditional relational databases by utilizing a flexible, JSON-like document structure called BSON (Binary JSON). This schema-less design allows for a dynamic and adaptable data model, making *MongoDB* particularly well-suited for applications requiring the storage of complex and evolving data.

Advantages of *MongoDB*

- **Flexible Schema**: MongoDB's document-based structure, with data stored in JSON-like formats, allows each collection to have documents of varying sizes and structures. This flexibility is advantageous for applications that require frequent and diverse data modifications without the rigidity of predefined schemas.
- **High Performance for CRUD Operations**: Since *MongoDB* is not bound by relational constraints, such as foreign keys or join operations, CRUD (Create, Read, Update, Delete) operations are executed more efficiently. This results in lower latency and higher throughput, making it suitable for applications with high transaction rates.
- **Scalability**: *MongoDB* supports horizontal scaling through sharding, where data is distributed across multiple nodes in a cluster. This architecture allows for seamless addition of new nodes to the cluster, facilitating system expansion and accommodating large-scale data growth.
- **Indexing for High Performance**: *MongoDB* automatically indexes the unique identifier field ($\_id$) in each document, ensuring high performance for query operations. Additional indexes can be created on other fields to further optimize query performance.
- **Efficient Data Access**: *MongoDB* caches frequently accessed data in RAM, minimizing disk I/O and improving read and write speeds. This in-memory caching capability is critical for applications demanding fast data retrieval and real-time processing.

Suitability for Dynamic User Data and Authentication Backend

MongoDB's flexible schema and powerful querying capabilities make it an excellent choice for storing dynamic user data and attributes. User profiles in modern applications often include a wide array of data points that can change over time, such as preferences, activity logs, and social connections. MongoDB's document-oriented model allows for easy modification and extension of user data structures without the need for costly schema migrations.

Additionally, *MongoDB* is highly suitable for use in authentication backend servers due to the following reasons:

- **Scalable User Management**: As the number of users grows, MongoDB's sharding capability ensures that the authentication system can scale horizontally to handle increased load, providing consistent performance.
- **Secure Data Storage**: *MongoDB* offers robust security features, including encryption at rest and in transit, role-based access control, and auditing capabilities, ensuring that sensitive user data is protected.
- **Efficient Session Management**: Authentication systems often require efficient session management and storage of session data. MongoDB's high read and write speeds, coupled with its ability to cache data in RAM, enable rapid access to session information, enhancing user experience.
- **Real-time Data Processing**: MongoDB's real-time data processing capabilities are critical for authentication systems that need to validate user credentials, track login attempts, and enforce security policies promptly.

By leveraging *MongoDB*, developers can build robust, scalable, and secure authentication systems that efficiently handle dynamic user data and meet the performance demands of modern applications.

3.1.4 React

React.js is a highly efficient JavaScript library developed by engineers at Facebook, widely adopted by major companies such as Yahoo, Airbnb, Facebook, and Instagram for their product development. It excels in creating large, scalable applications, making it a preferred choice to smaller projects.

Features of React.js

- **Component-Based Architecture**: React promotes the development of reusable components, enabling developers to break down complex problems into manageable, testable units. This modular approach simplifies the management and extension of the system. Unlike Angular, which requires an optimal structure and coding style, React offers flexibility in structuring components, making it adaptable to various project requirements.

- **Stateless Components**: React encourages maintaining components as stateless as possible, enhancing predictability and simplifying state management. Stateless components function similarly to static HTML pages, receiving inputs and rendering outputs based on those inputs. This characteristic explains their reusability and ease of testing.

Strengths of React.js

React is designed with performance optimization in mind, particularly in rendering views. Traditional MVVM (Model-View-View-Model) frameworks can struggle with efficiently displaying large datasets. React addresses this issue by re-rendering only the components that have changed, rather than the entire view.

One of React's core strengths lies in its use of the **Virtual DOM (Document Object Model)**. When a view update is required, React creates a virtual representation of the DOM. It then compares this virtual DOM with the actual DOM, identifies the differences, and applies only the necessary changes. This process, known as reconciliation, minimizes direct DOM manipulations, significantly enhancing performance.

For instance, in a list of 20 products, if the second product is updated, React will only re-render that specific product, leaving the other 19 products unchanged.

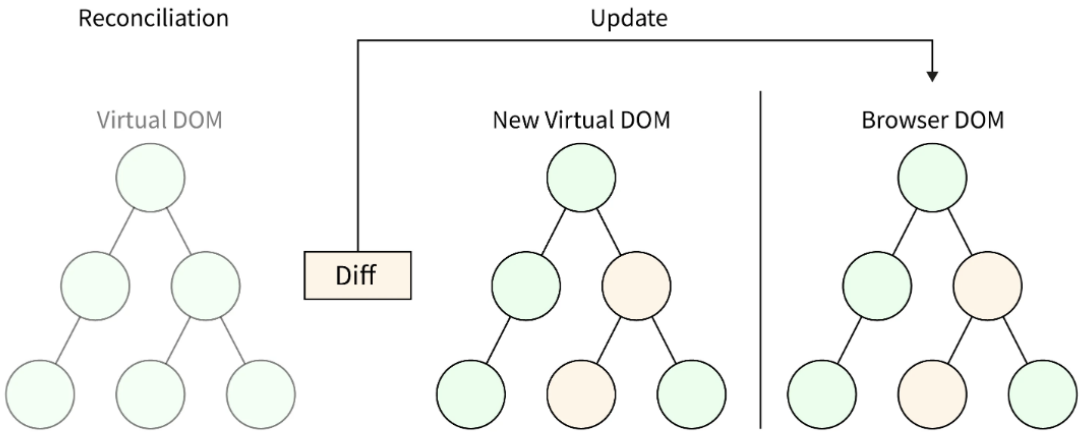


Figure 3.5: React Virtual DOM, Diffing and Reconcilliation

Suitability for Dynamic UIs with Real-Time Data

React's architecture and capabilities make it exceptionally well-suited for building dynamic user interfaces, particularly in scenarios where location data changes frequently due to real-time updates from a WebSocket backend.

Moreover, *React* can be easily integrated with numerous UI component driven libraries such as *Material UI*, *shadcn*, *ChakraUI*, *Shoelace* resulting in a faster development process.

React - A Suitable Choice for TraceBook

- **Efficient Data Handling**: React's ability to manage and render data efficiently is crucial for applications that require frequent updates. The Virtual DOM ensures that only the necessary parts of the UI are re-rendered, providing a smooth and responsive user experience.
- **Seamless Integration with WebSockets**: *React* can seamlessly integrate with WebSocket-based backends to handle real-time data updates. Components can subscribe to WebSocket events, ensuring that any changes in location data are immediately reflected in the UI without the need for full-page reloads.
- **Declarative UI Updates**: React's declarative approach to UI development allows developers to specify how the UI should look based on the current state. When location data changes, *React* automatically updates the affected components, ensuring that the UI remains consistent and up-to-date.
- **Improved User Experience**: The efficient update mechanism provided by the Virtual DOM enhances the user experience by reducing the time and resources needed to reflect data changes in the UI. This is particularly important for applications that display real-time location data, where quick updates are essential.

Hence, React's component-based architecture, performance optimization through the Virtual DOM, and seamless integration with real-time data sources like WebSockets make it an ideal choice for developing dynamic UIs that handle frequently changing location data.

3.1.5 TypeScript

TypeScript is a statically typed superset of JavaScript that compiles to plain JavaScript. It offers several advantages for both frontend and backend development.

Transpilation and Runtime Execution

In TypeScript development, the TypeScript compiler (`tsc`) is used to *transpile* TypeScript code into plain JavaScript code. The *transpilation* process involves converting TypeScript source files (.ts) into JavaScript files (.js) that can be executed by JavaScript engines.

Once the TypeScript code is *transpiled* into JavaScript, it can be run in the same way as any other JavaScript code. In frontend development, the *transpiled* JavaScript files are included in HTML documents and executed by web browsers' JavaScript engines. Similarly, in backend development, the *transpiled* JavaScript files are executed by Node.js, a runtime environment for executing JavaScript code outside a web browser.

Under the hood, the TypeScript compiler parses TypeScript code, type-checks it based on type annotations and inferred types, and generates equivalent JavaScript code. This

JavaScript code retains the same functionality as the original TypeScript code but lacks type annotations and other TypeScript-specific features.

The *transpilation* process ensures that TypeScript code can be executed in any JavaScript environment, enabling developers to write TypeScript code for both frontend and backend development. Additionally, TypeScript's static typing system provides compile-time type checking to catch errors early in the development process, while the resulting JavaScript code runs efficiently in modern JavaScript engines.

TypeScript - A Suitable Choice for TraceBook

Frontend Development

- **Enhanced Code Quality**: TypeScript's static typing system allows developers to catch type-related errors during development, leading to higher code quality and fewer runtime errors in frontend applications.
- **Improved Developer Productivity**: With features like type inference and intelligent code completion, TypeScript enables developers to write cleaner and more maintainable code. This leads to increased productivity and faster development cycles.
- **Better Tooling Support**: TypeScript has excellent tooling support, including integrated development environments (IDEs) like Visual Studio Code and powerful debugging tools. These tools enhance the development experience and streamline the frontend development process.
- **Stronger Codebase**: By enforcing type safety, TypeScript helps build a stronger and more robust codebase for frontend applications. This reduces the likelihood of bugs and makes it easier to refactor and maintain the code over time.

Backend Development

- **Unified Development Experience**: Using TypeScript in both frontend and backend development creates a unified development experience across the entire stack. Developers can leverage their knowledge of TypeScript to work seamlessly on both frontend and backend codebases.
- **Shared Code and Interfaces**: TypeScript allows for the creation of shared code and interfaces between frontend and backend components. This promotes code reuse and ensures consistency across the application, leading to more maintainable and scalable software solutions.
- **Enhanced Error Detection**: Similar to frontend development, TypeScript's static typing system helps detect errors in backend code during development. This reduces the risk of runtime errors and enhances the overall reliability of backend services.
- **Improved Collaboration**: With TypeScript, teams can collaborate more effectively on backend development projects. The use of static types and clear interfaces makes it easier for developers to understand and contribute to the codebase, leading to better teamwork and faster project execution.

Leveraging TypeScript in both frontend and backend development offers numerous benefits, including improved code quality, developer productivity, and collaboration, as well as stronger codebases and enhanced error detection.

3.2 System Design

The implementation design of a Privacy Network is discussed in this section. The implemented design can extend itself to a truly scalable and distributed web application. The final product, *TraceBook*, primarily has **four servers:**

- Frontend Server:
  - `ts-frontend`: The *React* Frontend
- Backend Server:
  - `ts-backend`: The *MongoDB* Backend
  - `loc`: The *Postgres* Backend
  - `ws`: The *WebSocket* Backend

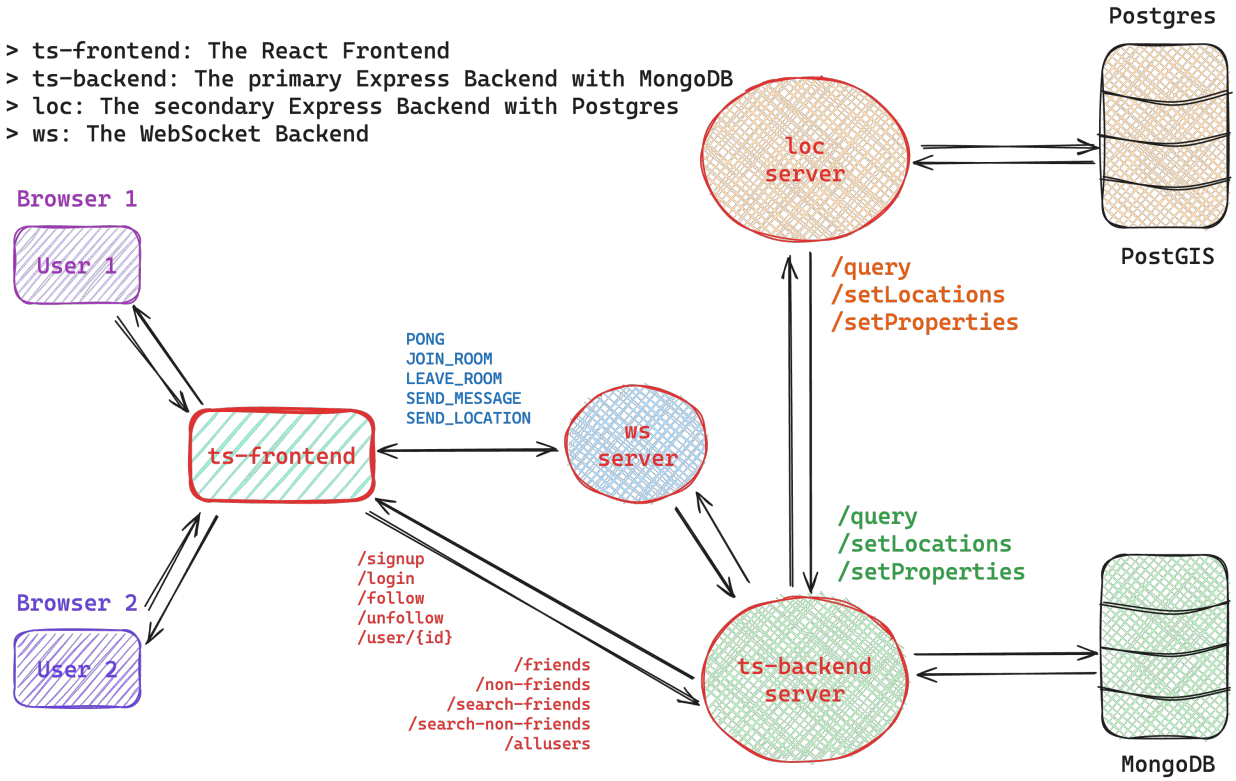


Figure 3.6: Architecture of *TraceBook*

The `ts-backend` server is tightly integrated with a *MongoDB* NoSQL database, which is responsible for storing all user authentication details and friendship connections. Conversely, the `loc` server is integrated with a PostgreSQL database, dedicated to storing user location data. Additionally, the `ws` server employs an in-memory data store to manage information about the *rooms* and the *users* within those rooms.

The `ts-frontend` server interacts with both the `ws` server and the `ts-backend` server. This interaction follows the traditional client-server architecture model, where communication occurs through request-response pairs. However, the `ws` server leverages WebSockets to facilitate real-time, bidirectional full-duplex communication. The WebSocket protocol operates over TCP and initiates a protocol upgrade via an HTTP request, transitioning from the `http` connection protocol to the `ws` protocol. Subsequently, the `ts-backend` server communicates with the `loc` server.

It is important to note that the `loc` server is never directly accessed by the `ts-frontend` server. The `interBackendAccess` middleware enforces this restriction, preventing any unauthorized access to the `loc` backend server. We implement a write-through operation from the `ts-backend` server to the `loc` server to maintain synchronization and consistency between the PostgreSQL and *MongoDB* databases.

As illustrated in the accompanying Figure3.6, the `ts-backend` and `loc` servers share several common API endpoints, such as `query`, `setLocations`, and `setProperties`. When performing write-through operations, a request is initially made to the `ts-backend` server from the `ts-frontend` server at a specific route. This request is subsequently forwarded to the `loc` server, where the PostgreSQL database is updated first. Only upon the successful completion of the CRUD operation in PostgreSQL do we proceed to perform the corresponding CRUD operation in the *MongoDB* database within the `ts-backend` server. Finally, the desired response is returned to the `ts-frontend` client.

This architecture ensures data integrity and consistency across the two databases, leveraging the strengths of both NoSQL and relational database management systems. The use of WebSockets enhances real-time communication capabilities, essential for dynamic geosocial networking applications.

- `ts-backend`: Manages user authentication and friendship data using MongoDB.
- `loc`: Handles user location data with PostgreSQL and PostGIS.
- `ws`: Manages real-time communication using WebSockets.
- `ts-frontend`: Acts as the client interface, communicating with `ts-backend` and `ws`.
- `interBackendAccess`: Middleware to enforce secure backend communication.
- **Write-through operation**: Ensures data consistency by updating PostgreSQL first, followed by *MongoDB*.

3.3 Privacy Frontend

3.3.1 A Component Based *React* with Google Maps API UI

The Google Maps API is a powerful tool provided by Google that allows developers to integrate the rich functionality of Google Maps into their web and mobile applications. It offers a variety of services such as displaying maps, adding markers, generating routes, and handling geolocation data, making it an essential component for location-based applications.

Features of Google Maps API:

- **Comprehensive Mapping Solutions**: Google Maps API provides a robust set of features for displaying maps, which can be tailored to suit various application needs. From basic map displays to complex route calculations and traffic visualizations, it covers all aspects of mapping requirements.
- **Accurate and Up-to-Date Data**: Google Maps API leverages Google's extensive and frequently updated geospatial database, ensuring that users have access to the most accurate and current location information. This reliability is crucial for applications that depend on precise location data, such as navigation apps and delivery services.
- **Customizable Map Styling**: The API allows developers to customize the appearance of maps to match the branding and design of their application. This includes changing the color scheme, hiding or showing specific map elements, and adding custom markers, providing a unique and cohesive user experience.
- **Seamless Integration with Other Google Services**: Google Maps API integrates smoothly with other Google services, such as Google Places and Google Earth. This allows developers to enrich their applications with additional functionalities like place searches, detailed place information, and 3D visualization.
- **Global Reach and Scalability**: With its global coverage, Google Maps API ensures that applications can cater to users from different regions without any additional configuration. It also supports high scalability, handling large numbers of users and extensive data loads efficiently, making it suitable for both small-scale and enterprise-level applications.
- **Real-Time Data and Traffic Information**: The API provides real-time data on traffic conditions, transit information, and estimated travel times, which is invaluable for applications focusing on navigation and logistics. This helps users make informed decisions based on current traffic and transit scenarios.

Marker Component:

The `<Marker>` component from the `react-google-maps/api` library is used to place a marker on the Google Map at a specified location.

Key Component:

The key prop is set to the length of the *qLocations* array. This ensures that the marker has a unique key, which is important for React's reconciliation process.

onClick event:

The `onClick` prop specifies a function to be executed when the marker is clicked. This function sets the state variable `clickedIndex` to the length of the *qLocations* array, indicating that this particular marker was clicked.

Position:

The position prop defines the geographical coordinates where the marker will be placed. It uses the `currentUserPosition` object, which contains the latitude and longitude of the current user's location.

Icon:

The icon prop customizes the appearance of the marker.

- **url**: Sets the URL for the marker's icon. If `curruser?.Photo` exists, it applies the insertTransformationParams function to the photo URL, otherwise inserts `DEFAULT_PROFILE_URL`.
- **scaledSize**: Specifies the size of the icon to be 40x40 pixels.
- **origin**: Sets the origin point of the icon at (0, 0).
- **anchor**: Sets the anchor point of the icon at (20, 40), ensuring the icon is positioned correctly on the map.
- **animation** : The `animation` prop adds a bounce animation to the marker, making it more noticeable and engaging for users.

Figure 3.7 effectively places a marker on the Google Map at the current user's location, represented with a custom icon. The marker includes an `onClick` event to handle interactions, which sets a state variable to indicate that the marker was clicked. The bounce animation adds a dynamic visual cue, enhancing user interaction and experience.

Circle Component

The `<Circle>` component from the `react-google-maps/api` library is used to place a marker on the Google Map at a specified location.

```jsx
<div className="map-container">
  <GoogleMap
    mapContainerStyle={containerStyle}
    center={currentMapCenter}
    zoom={9}
  >
    <Marker
      key={qLocations.length}
      onClick={() => {
        setClickedIndex(qLocations.length)
      }}
      position={{
        lat: currentUserPosition.lat,
        lng: currentUserPosition.lng,
      }}
      icon={{
        url: curruser?.Photo
          ? insertTransformationParams(curruser?.Photo)
          : DEFAULT_PROFILE_URL,
        scaledSize: new google.maps.Size(40, 40),
        origin: new google.maps.Point(0, 0),
        anchor: new google.maps.Point(20, 40),
      }}
      animation={google.maps.Animation.BOUNCE}
    />
  </GoogleMap>
</div>
```

Figure 3.7: Marker Component

Key Component:

Ensures each circle has a unique identifier based on the index.

Event Handlers:

onLoad, onCenterChanged, onUnmount, and *onClick* are provided, with *onClick* setting the *clickedIndex* to the current index.

center:

Specifies the center of the circle as a (*lat,lng*) value. The patches are shown around this point

options:

Determines the circle's appearance based on whether the current user is friends with the location owner (*circleOptionForFriends* or *circleOptionForNonFriends*)

This `<Circle>` component is utilized to render the Red and Blue Patches to show the Low Visibility Level and Medium Visibility Level. This is done using the *circleOptionForFriends* (Figure 3.8) method and *circleOptionForNonFriends* (Figure 3.9) method conditionally as per the `mask` property of the *qLocations* context (Figure 3.10)

3.3.2 *Vite* the Build Tool

Vite is a modern build tool that is specifically designed for building web applications. The relevance of using *Vite*in our project is:

- **Development Server**: *Vite*comes with a built-in development server that offers blazing-fast hot module replacement (HMR). This means that changes made to your code are reflected instantly in the browser without needing a full page reload, leading to a highly efficient development experience.
- **Lightning-fast Builds**: *Vite*leverages native ES Module (ESM) support in modern browsers to provide incredibly fast build times. By leveraging browser support for ESM, *Vite*skips the bundling step during development, resulting in near-instantaneous builds even for large applications.
- **Optimized for Speed**: *Vite*is optimized for speed throughout the development process. Its development server and build process are designed to provide the quickest feedback loop possible, enabling developers to iterate rapidly and efficiently.

```
export const circleOptionForFriends = {
  strokeColor: '#0c4a6e',
  strokeOpacity: 0.8,
  strokeWeight: 2,
  fillColor: '#0ea5e9',
  fillOpacity: 0.35,
  clickable: true,
  draggable: false,
  editable: false,
  visible: true,
  radius: 10000,
  zIndex: 1,
}
```

Figure 3.8: Circle for Friends Attributes

```
export const circleOptionForNonFriends = {
  strokeColor: "#FF0000",
  strokeOpacity: 0.8,
  strokeWeight: 2,
  fillColor: "#FF0000",
  fillOpacity: 0.35,
  clickable: true,
  draggable: false,
  editable: false,
  visible: true,
  radius: 20000,
  zIndex: 1,
};
```

Figure 3.9: Circle for Non-Friends Attributes

```
{qLocations.map((loc, index) => {
    if (
      loc.lat &&
      loc.lng &&
      loc.hasOwnProperty("mask") &&
      loc.mask === true &&
      isIDExistInMyLocation(loc.id)
    ) {
      return (
        <>
          <Circle
            key={index}
            onLoad={onLoad}
            onCenterChanged={() => {}}
            onUnmount={onUnmount}
            onClick={() => {
              setClickedIndex(index);
            }}
            center={{
              lat: loc.lat,
              lng: loc.lng,
            }}
            options={
              ifCurrentUserMyFriend(loc.id)
                ? circleOptionForFriends
                : circleOptionForNonFriends
            }
          />
        </>
      );
    }
  })}
```

Figure 3.10: Conditional Rendering of Friends & Non-Friends

- **Flexible Configuration**: While *Vite*offers sensible defaults out of the box, it also provides a flexible configuration system that allows developers to tailor the build process to their specific needs. This includes options for customizing plugins, loaders, and other build-related settings.
- **Plugin Ecosystem**: *Vite*has a growing ecosystem of plugins that extend its functionality. These plugins cover a wide range of use cases, from optimizing asset loading to integrating with various frameworks and libraries, further enhancing Vite's versatility.
- **Framework Agnostic**: While *Vite*is commonly associated with Vue.js due to its official support for Vue, it is not limited to any specific framework. *Vite*can be used with any JavaScript framework or library, including *React*, *Angular*, and *Svelte*, making it a versatile choice for web development projects.
- **Modern Build Pipeline**: *Vite*embraces modern web technologies and standards, such as ES modules, which aligns with the direction of the web platform. By leveraging native browser capabilities, *Vite*offers a forward-looking approach to building web applications.

3.4 Privacy Backend and Databases

The `ts-backend` server is seamlessly integrated with a *MongoDB* NoSQL database, which handles all user authentication information and friendship connections. In contrast, the `loc` server is paired with a PostgreSQL database, specifically designed to store user location data. Additionally, the `ws` server uses an in-memory data store to manage the data related to *rooms* and *users* joined to those rooms.

The `ts-frontend` server communicates with both the `ws` server and the `ts-backend` server. This communication follows the traditional client-server model, where interactions occur through request-response cycles. However, the `ws` server utilizes WebSockets for real-time, bidirectional full-duplex communication. The WebSocket protocol runs over TCP and initiates a protocol upgrade via an HTTP request, transitioning from the `http` protocol to the `ws` protocol. Subsequently, the `ts-backend` server communicates with the `loc` server.

It is crucial to note that the `ts-frontend` server never directly accesses the `loc` server. The `interBackendAccess` middleware enforces this security measure, blocking any unauthorized access to the `loc` backend server. We implement a write-through operation from the `ts-backend` server to the `loc` server to maintain synchronization and consistency between the PostgreSQL and *MongoDB* databases (Figure 3.11).

The choice of *MongoDB* for user and friendship data offers several advantages. *MongoDB's* schema-less design allows for flexibility in storing diverse user data and rapid iterations in development. Its horizontal scalability ensures that the system can handle a growing number of users efficiently.

On the other hand, PostgreSQL, coupled with the PostGIS extension, is highly effective for managing spatial data, making it ideal for location-based services. PostGIS adds support for geographic objects to the PostgreSQL database, enabling advanced location queries and spatial operations. This integration allows for precise and efficient management of user location data, enhancing the application's ability to handle complex geospatial queries.

As mentioned earlier, we perform a write-through operation (Figure 3.11) from the `ts-backend` to the `loc` backend. This happens via an *interBackendAccess* middleware, which is described in Figure 3.12

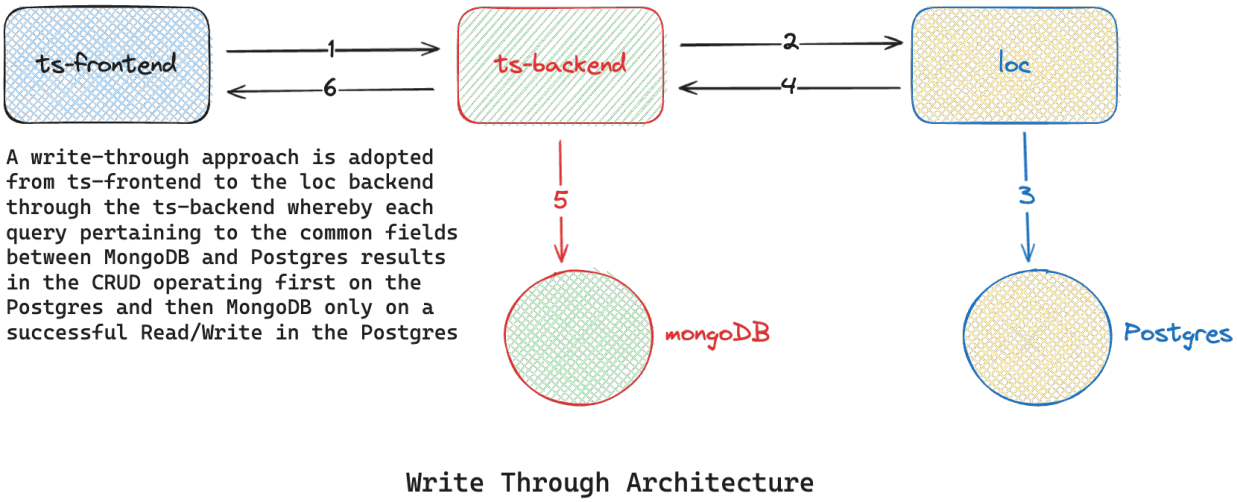


Figure 3.11: Maintaining Consistency Across the SQL-NoSQL Databases

3.4.1 Database Schema, ORMs and ERDs

We have employed Object-Relational Mapping (ORM) tools to facilitate interaction with our databases:

- ***Mongoose* for MongoDB:** Mongoose is an ORM for MongoDB, providing a straightforward way to model application data and interact with *MongoDB* databases. It enables developers to define schemas, enforce data validation, and perform CRUD operations using a JavaScript-based syntax.
- ***Prisma* for PostgreSQL:** *Prisma* is a modern database toolkit for TypeScript and Node.js, offering type-safe database access and powerful query capabilities. It simplifies database interactions by generating TypeScript-based query builders and providing auto-completion for database schema.

Our database schema comprises collections/tables designed to accommodate the application's data structure. We utilize *MongoDB Atlas* to host the cloud *MongoDB* instance and *Supabase* for hosting the cloud *PostgreSQL* instance, ensuring reliable and scalable database management.

```ts
import { Request, Response, NextFunction } from "express";
import HttpStatusCode from "../types/HttpStatusCode";
import bcrypt from "bcryptjs";

export const interBackendAccess = async (
  req: Request,
  res: Response,
  next: NextFunction,
) => {
  const { authorization } = req.headers;

  try {
    if (!authorization) {
      return res
        .status(HttpStatusCode.UNAUTHORIZED)
        .send({ errors: "You are unauthorized to access this resource" });
    }

    if (!process.env.BACKEND_INTERCOMMUNICATION_SECRET) {
      console.error(
        "BACKEND_INTERCOMMUNICATION_SECRET is undefined. Check the .env",
      );
      return res
        .status(HttpStatusCode.INTERNAL_SERVER_ERROR)
        .send({ errors: "Internal Server Error" });
    }

    const token = authorization.replace("Bearer ", "");

    const isMatch = bcrypt.compareSync(
      process.env.BACKEND_INTERCOMMUNICATION_SECRET,
      token,
    );

    if (!isMatch) {
      return res
        .status(HttpStatusCode.UNAUTHORIZED)
        .send({ errors: "You are unauthorized to access this resource" });
    }
    next();
  } catch (error) {
    console.error(error)
    return res
      .status(HttpStatusCode.UNAUTHORIZED)
      .send({ errors: "You must be logged in" });
  }
};
```

Figure 3.12: *interBackendAccess* Middleware
This middleware cuts off any unauthorized access to the `loc` server

Entity-Relationship Diagrams (ERDs)

To visualize the structure of our databases, we have created Entity-Relationship Diagrams. These diagrams illustrate the relationships between different entities and the attributes associated with each entity.

```
export interface UserDocument extends Document {
  name: string;
  username: string;
  email: string;
  password: string;
  Photo?: string;
  age?: number;
  gender?: string;
  college?: string;
  friends: Types.ObjectId[];
  visibility: boolean;
  postgresId: string;
}

const userSchema: Schema<UserDocument> = new Schema({
  name: { type: String, required: true },
  username: { type: String, required: true },
  email: { type: String, required: true },
  password: { type: String, required: true },
  Photo: { type: String },
  age: { type: Number },
  gender: { type: String }
  college: { type: String },
  friends: [{ type: Schema.Types.ObjectId, ref: "user" }],
  visibility: { type: Boolean },
  postgresId: { type: String, required: true }
});

const User = mongoose.model<UserDocument>("user",
userSchema);
export default User;
```

Figure 3.13: *MongoDB* Schema using *Mongoose*

In our application, we use *Prisma* to manage the PostgreSQL database hosted on *Supabase*. The *Prisma* Schema (Figure 3.14) defines all the necessary data models and their relationships, facilitating seamless interaction between our application and the database.

This setup ensures that our location data is stored, queried, and managed efficiently.

```
generator client {
  provider = "prisma-client-js"
}

datasource db {
  provider  = "postgresql"
  url       = env("DATABASE_URL")
  directUrl = env("DIRECT_URL")
}

model Location {
  id        String   @id @default(dbgenerated("gen_random_uuid()"))
@db.Uuid
  latitude  Float
  longitude Float
  createdAt DateTime @default(now()) @db.Timestamp(6)
  updatedAt DateTime @db.Timestamp(6)
  users     User[]
}

model User {
  id         String    @id @default(dbgenerated("gen_random_uuid()"))
@db.Uuid
  name       String
  email      String    @unique
  age        Float
  gender     String
  college    String
  isVisible  Boolean
  locationId String?   @db.Uuid
  location   Location? @relation(fields: [locationId], references:
[id], onDelete: Cascade)
}
```

Figure 3.14: *Prisma* Schema

The ERD for the *MongoDB* schema (Figure 3.13) illustrates the collections and their relationships, while the ERD for the PostgreSQL schema (Figure 3.15) depicts the tables and their associations. These diagrams serve as valuable references for database design and maintenance, aiding in understanding the data model and ensuring consistency in database operations.

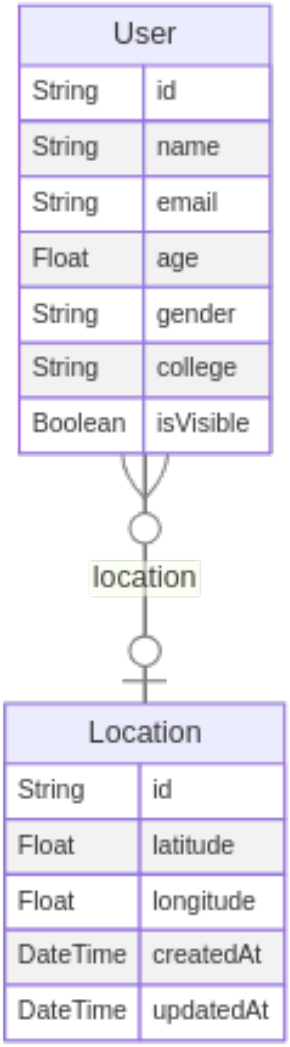


Figure 3.15: Entity-Relationship Diagram for PostgreSQL Schema using *Prisma*

3.4.2 PostGIS for Geolocation SQL Queries

PostGIS is an extension for the PostgreSQL relational database that enables geographic objects to be stored and queried. It adds support for geographic objects, allowing location queries to be run in SQL. PostGIS is crucial for any application that requires geospatial data processing, providing powerful tools to handle geographic data types and perform spatial operations.

PostGIS Capabilities and Use

PostGIS allows for the storage and manipulation of spatial data types such as points, lines, and polygons. It provides a range of spatial functions, operators, and indexing capabilities that make it highly efficient for geospatial queries. These capabilities are essential for applications dealing with mapping, location-based services, and spatial data analysis.

In our application, PostGIS is used to manage and query user location data stored in our PostgreSQL database hosted on *Supabase*. This enables us to perform complex geospatial queries to filter and identify users based on various parameters such as age, gender, college, and proximity to a specific geographic point.

Custom SQL Queries for Filtering Users

We have written custom SQL queries (Figure 3.16) utilizing PostGIS functions to filter and query users based on specific criteria. Below is an example of a function named `nearby_privacy_entities_visibility` that retrieves users based on age, gender, college, and distance from a given point.

Explanation of the SQL Query

The function `nearby_privacy_entities_visibility` performs the following tasks:

- **Input Parameters:**
  - `lat, long`: Latitude and longitude of the reference point.
  - `d`: Maximum distance from the reference point.
  - `max_age`: Maximum age of users.
  - `college_param`: College filter parameter.
  - `gender_param`: Gender filter parameter.
  - `isVisible`: Visibility status of the users.
  - `entity_type`: Type of entity to filter (e.g., 'user').
- **Returned Table Columns:**

  - User details such as `id`, `name`, `email`, `age`, `gender`, `college`, `isVisible`, and their geolocation `lat`, `long`.
  - `dist_meters`: Distance from the reference point.

- **SQL Logic:**
  - Joins the `"Location"` table with the `"User"` table using the `locationId`.
  - Filters users based on the distance from the reference point using the `ST_DISTANCE` function.
  - Applies additional filters such as `college_param`, `gender_param`, and `max_age`.
  - Ensures that only users with the specified visibility status are included.
  - Orders the results based on proximity to the reference point using the `<->` operator.

By leveraging PostGIS, we can efficiently manage and query geospatial data, providing a robust solution for location-based user filtering in our application.

```sql
CREATE OR REPLACE FUNCTION nearby_privacy_entities_visibility(
    IN lat FLOAT,
    IN long FLOAT,
    IN d FLOAT,
    IN max_age FLOAT,
    IN college_param TEXT,
    IN gender_param TEXT,
    IN isVisible BOOLEAN,
    IN entity_type TEXT
)
RETURNS TABLE (
    id TEXT,
    name TEXT,
    email TEXT,
    age FLOAT,
    gender TEXT,
    college TEXT,
    isVisible BOOLEAN,
    lat FLOAT,
    long FLOAT,
    dist_meters FLOAT
)
LANGUAGE SQL
AS $$
    SELECT
        entity.id,
        entity.name,
        entity.email,
        entity.age,
        entity.gender,
        entity.college,
        entity."isVisible",
        latitude AS lat,
        longitude AS long,
        ST_DISTANCE(
            ST_GeomFromText('POINT(' || long || ' ' || lat || ')', 4326)::geography,
            ST_SetSRID(ST_MakePoint(longitude, latitude), 4326)::geography
        ) AS dist_meters
    FROM
        "Location"
    JOIN
        "User" entity ON "Location".id = entity."locationId"
    WHERE
        ST_DISTANCE(
            ST_GeomFromText('POINT(' || long || ' ' || lat || ')', 4326)::geography,
            ST_SetSRID(ST_MakePoint(longitude, latitude), 4326)::geography
        ) <= d AND entity_type = 'user'
        AND (college_param IS NULL OR entity.college = college_param
        AND (gender_param IS NULL OR entity.gender = gender_param)
        AND (max_age IS NULL OR entity.age <= max_age)
        AND entity."isVisible" = isVisible
    ORDER BY
        ST_SetSRID(ST_MakePoint(longitude, latitude), 4326)::geography <-> ST_GeomFromText('POINT(' || long || ' ' || lat || ')', 4326)::geography
$$;
```

Figure 3.16: Custom SQL Query Using PostGIS for Geolocation

3.4.3 WebSocket Server for Real-Time Location Sharing

The WebSocket server is designed to manage real-time location sharing. Utilizing the `ws` library, the server handles complex interactions by upgrading HTTP connections to WebSocket (`ws`) protocol, maintaining connection health through a `PING/PONG` mechanism, and processing various incoming messages. Additionally, an in-memory user manager tracks user and room data, ensuring efficient handling of location-sharing tasks.

WebSocket Protocol Upgrade

The WebSocket server initiates by listening for HTTP requests on a specified port. Upon receiving an upgrade request, the server switches the protocol from HTTP to `ws`, enabling real-time, bidirectional communication.

```
const app = express();
const PORT: string | number = process.env.PORT || 8080;

const httpServer = app.listen(PORT, () => {
  console.log(`HTTP Server is running on PORT ${PORT}`);
});

const wss = new WebSocketServer({ noServer: true });

httpServer.on("upgrade", (req, socket, head) => {
  socket.on("error", onSocketPreError);

  console.log(
    `HTTP Server requesting to upgraded to ws to run on PORT ${PORT}`,
  );

  wss.handleUpgrade(req, socket, head, (ws) => {
    socket.removeListener("error", onSocketPreError);
    wss.emit("connection", ws, req);
  });
});

  });
});
```

Figure 3.17: WebSocket Protocol Upgrade

The upgrade process involves:

- Listening for HTTP requests on the specified port.
- Handling upgrade requests to switch from HTTP to WebSocket protocol.
- Emitting a connection event once the upgrade is successful, allowing communication over the WebSocket.

```
wss.on("connection", async function connection(ws: WebSocket, req) {
  const token: string = url.parse(req.url, true).query.token || "";

  const checkIfAuthorized = await tokenIsValid(token);
  if (!token || !checkIfAuthorized) {
    console.log("Unauthorized user received in ws backend");
    ws.send(
      JSON.stringify({
        type: "UNAUTHORIZED",
        payload: {
          message:
            "You are not authorized to connect to the ws WebSocketServer",
        },
      }),
    );
    ws.terminate();
    return;
  }

  ws.isAlive = true;

  ws.on("error", onSocketPostError);

  ws.on("message", function message(data: any) {
    try {
      const jsonData = JSON.parse(data.toString());
      messageHandler(ws, jsonData);
    } catch (error) {
      console.error(error);
    }
  });

  ws.on("close", () => {
    console.log("Connection closed from ws server");
  });
});
```

Figure 3.18: WebSocket Protocol Upgrade

Connection Health Management

To maintain active connections, the server implements a PING/PONG architecture. This mechanism periodically sends PING messages to clients and expects PONG responses, ensuring that connections remain alive.

Key steps in this architecture include:

- Sending PING messages at regular intervals.
- Marking clients as alive upon receiving a PONG response.
- Terminating connections that do not respond within a predefined interval.

Message Handling

The WebSocket server includes a comprehensive message handler to process various types of incoming messages. This handler categorizes messages and performs appropriate actions based on their types.

- `JOIN_ROOM`: Adds a user to a specified room and broadcasts the event to other users in the room.
- `LEAVE_ROOM`: Removes a user from a room and informs remaining users.
- `SEND_MESSAGE`: Relays chat messages to all users in the room.
- `SEND_LOCATION`: Updates and broadcasts the user's location to the room.

User Management

An in-memory user manager is employed to keep track of users and their room affiliations. This manager maintains a list of rooms, each containing a list of users and their associated WebSocket connections.

```
interface User {
  name: string;
  id: string;
  email: string;
  age: number;
  gender: string;
  college: string;
  lat: number;
  lng: number;
  dist_meters?: number | undefined;
  Photo?: string | undefined;
  mask?: boolean | undefined;
  conn: WebSocket;
}

interface Room {
  users: User[];
}
```

Figure 3.19: In-Memory User Manager Interface

Functions of the User Manager include:

- `addUser`: Adds a user to a room, storing their connection and relevant details.

- `removeUser`: Removes a user from a room, ensuring their data is cleaned up.
- `getUser`: Retrieves user information based on room and user IDs.
- `broadcastToRoom`: Sends messages to all users in a specified room, excluding the sender.

Authentication

The server employs token-based authentication to ensure that only authorized users can establish WebSocket connections. The token is validated using a secret key and verified against an external authentication service.

The authentication process includes:

- Extracting the token from the connection request URL.
- Validating the token using a predefined secret key.
- Verifying the user's identity with the `ts-backend` and ***MongoDB*** before establishing the connection.

The WebSocket server efficiently handles real-time location sharing through a robust architecture that includes protocol upgrades, connection health management, message handling, and user management. Future enhancements may include persistent storage for user locations and additional message types to support more interactive features.

3.4.4 OpenAPI Specs and Swagger Documentation

We have meticulously documented the OpenAPI specifications for the `ts-backend` server, providing a comprehensive guide to its functionality and endpoints. The documentation covers every aspect of the server's API, detailing the request and response payloads, supported HTTP methods, path parameters, query parameters, request headers, and response status codes.

Each endpoint in the OpenAPI documentation is thoroughly described, outlining its purpose, expected input data, and the format of the response. The documentation includes examples of both request and response payloads, ensuring clarity and ease of understanding for developers and API consumers.

Furthermore, the OpenAPI specs document any authentication mechanisms or security requirements implemented for the `ts-backend` server, providing clear guidelines on how to authenticate requests and handle sensitive data securely.

In addition to the endpoint descriptions, the OpenAPI documentation includes information about data models or schemas used in the API, specifying the structure and data types of input and output objects.

The OpenAPI documentation serves as a valuable resource for developers integrating with the `ts-backend` server, offering detailed insights into its capabilities and usage. It promotes consistency and standardization in API development and facilitates seamless collaboration between frontend and backend teams.

3.5 Cloud Hosting and Deployment

The deployment of the web application involves hosting the frontend on *Vercel* and the backend servers (`ts-backend`, `loc`, and `ws`) on *Render*. The deployment process is automated using GitHub Actions, which triggers deployments on code changes to the main branch. The domain `sanam.live` was previsouly purchased, and a subdomain `privacynetwork.sanam.live` is configured to link the frontend to this web address.

3.5.1 Frontend Hosting on *Vercel*

The frontend of the application, developed using TypeScript, is deployed on *Vercel*. *Vercel* provides an efficient and straightforward integration with GitHub, allowing for automatic deployments whenever there are code changes pushed to the main branch. The deployment steps include:

1. Connecting the GitHub repository to *Vercel*.
2. Configuring the build settings in *Vercel*, specifying the build command and the output directory.
3. Setting up the environment variables in the *Vercel* dashboard under the project settings.

3.5.2 Backend Hosting on *Render*

The three backend servers (`ts-backend`, `loc`, and `ws`) are hosted on *Render*. *Render* offers a free instance suitable for hosting the backend servers. The deployment process for each server involves:

1. Connecting the GitHub repository to *Render*.
2. Setting up the build and start commands for each server in *Render's* dashboard.
3. Configuring the environment variables for each backend server in *Render's* dashboard.

3.5.3 Domain Configuration

The domain `sanam.live` has been purchased and configured. A subdomain `privacynetwork.sanam.live` is set up using `DNS` `CNAME` records to link the frontend to this web address. The steps to configure the subdomain are:

1. Log in to the domain registrar's dashboard and navigate to the `DNS` settings.
2. Add a `CNAME` record with the subdomain (`privacynetwork`) pointing to the *Vercel* domain.
3. Verify the `DNS` settings in *Vercel* to ensure the subdomain is correctly linked to the frontend.

3.5.4 Environment Variables Setup

Environment variables are essential for configuring the application across different environments. Each backend server and the frontend have their respective `.env` files. Below are the example `.env` setups for the four servers:

Frontend (`ts-frontend`)

```
VITE_BACKEND_URI=<BACKEND_API_URI: backend server address>
VITE_WS_URI=<WEBSOCKET_URI: web socket address>
VITE_GOOGLE_API_KEY=<GOOGLE_MAPS_API_KEY: API KEY for Google map>
```

Authentication and Main Backend Server (`ts-backend`)

```
GOOGLE_CLIENT=<GOOGLE_CLIENT_ID: for logging in with Google>
GOOGLE_API_KEY=<GOOGLE_API_KEY: for logging in with Google>
MONGO_URI=<MONGO_URI: for MongoDB database connection>
SECRET_KEY=<CLIENT_SECRET: for generating the JWT token>
PORT=<PORT: at which the backend server runs>
LOCATION_BACKEND_URI=<LOCATION_BACKEND_URI>
BACKEND_INTERCOMMUNICATION_SECRET=<BACKEND_INTERCOMMUNICATION_SECRET>
```

Location Server (`loc`)

```
ADMIN_PASSWORD=<SUPABASE_ADMIN_PASSWORD>
SUPABASE_URI=<SUPABASE_URI: for Supabase database connection>
SUPABASE_KEY=<SUPABASE_KEY: for Supabase database connection>
```

```
DATABASE_URL=<DATABASE_URL: for connecting to Supabase Postgres instance>
DATABASE_POOL_URL=<DATABASE_POOL_URL>
DIRECT_URL=<DIRECT_URL>
BACKEND_INTERCOMMUNICATION_SECRET=<BACKEND_INTERCOMMUNICATION_SECRET>
```

WebSocket Server (`ws`)

```
SECRET_KEY=<CLIENT_SECRET>
TS_BACKEND_URI=<AUTHENTICATION_SERVER_BACKEND_URI>
```

By correctly setting up the environment variables and configuring the deployment pipelines, the application ensures seamless integration and continuous deployment, enabling efficient and scalable development and deployment workflows.

```ts
export class UserManager {
  private rooms: Map<string, Room>;
  constructor() {
    this.rooms = new Map<string, Room>();
  }

  addUser(
    name: string,
    userId: string,
    roomId: string,
    socket: WebSocket,
    email: string,
    age: number,
    gender: string,
    college: string,
    lat: number,
    lng: number,
    dist_meters: number | undefined,
    Photo: string | undefined,
    mask: boolean | undefined,
  ) {
    if (!this.rooms.get(roomId)) {
      this.rooms.set(roomId, {
        users: [],
      });
    }

    const existingUser = this.getUser(roomId, userId);
    if (existingUser) {
      console.log(`User with id ${userId} already exists in the room!`);
      return;
    }

    this.rooms.get(roomId)?.users.push({
      id: userId,
      name,
      conn: socket,
      email,
      age,
      gender,
      college,
      lat,
      lng,
      dist_meters,
      Photo,
      mask,
    });
    console.log(`User with id ${userId} added!`);
    socket.on("close", () => {
      console.log("Socket is getting closed!");
      this.removeUser(roomId, userId);
    });
  }
}
```

Figure 3.20: Adding a User to the In-Memory User Manager

```typescript
import { WebSocket } from "ws";
import { OUTGOING_MESSAGE } from "./messages/outgoingMessages";

export class UserManager {
  private rooms: Map<string, Room>;
  constructor() {
    this.rooms = new Map<string, Room>();
  }

  removeUser(roomId: string, userId: string) {
    const room = this.rooms.get(roomId);
    if (room) {
      room.users = room.users.filter(({ id }) => id !== userId);
      console.log(`User with id ${userId} removed!`);
    } else {
      console.error(`Error occurred while removing user with id ${userId}`);
    }
  }

  getUser(roomId: string, userId: string): User | null {
    const user = this.rooms.get(roomId)?.users.find(({ id }) => id === userId);
    return user ?? null;
  }

  // broadcasts message to all the users present in the room having roomId
  broadcastToRoom(roomId: string, userId: string, message: OUTGOING_MESSAGE) {
    const room = this.rooms.get(roomId);
    if (!room) {
      console.error("Room not found");
      return;
    }

    room.users.forEach(({ conn, id }) => {
      if (id === userId) {
      } else {
        console.log(
          "Outgoing broadcasted message to user:",
          id + " " + JSON.stringify(message),
        );
        conn.send(JSON.stringify(message));
      }
    });
  }
}
```

Figure 3.21: Removing a User from the In-Memory & Broadcasting in User Manager

```ts
require("dotenv").config();
import axios from "axios";
import jwt from "jsonwebtoken";
import HttpStatusCode from "../types/HttpStatusCodes";

interface TokenPayload {
  _id: string;
}

export default async function tokenIsValid(token: string) {
  const JWT_SECRET = process.env.SECRET_KEY;
  const TS_BACKEND_URI = process.env.TS_BACKEND_URI;
  try {
    if (!JWT_SECRET) {
      console.error("SECRET_KEY is undefined. Check the .env");
      return false;
    }

    if (!TS_BACKEND_URI) {
      console.error("TS_BACKEND_URI is undefined. Check the .env");
      return false;
    }

    const info = jwt.verify(token, JWT_SECRET) as TokenPayload;

    const response = await
axios.get(`${TS_BACKEND_URI}/api/user/${info._id}`, {
      headers: {
        Authorization: `Bearer ${token}`,
        "Content-Type": "application/json",
      },
    });

    if (response.status !== HttpStatusCode.OK) {
      return false;
    }
  } catch {
    return false;
  }
  return true;
}
```

Figure 3.22: WebSocket Authentication Process

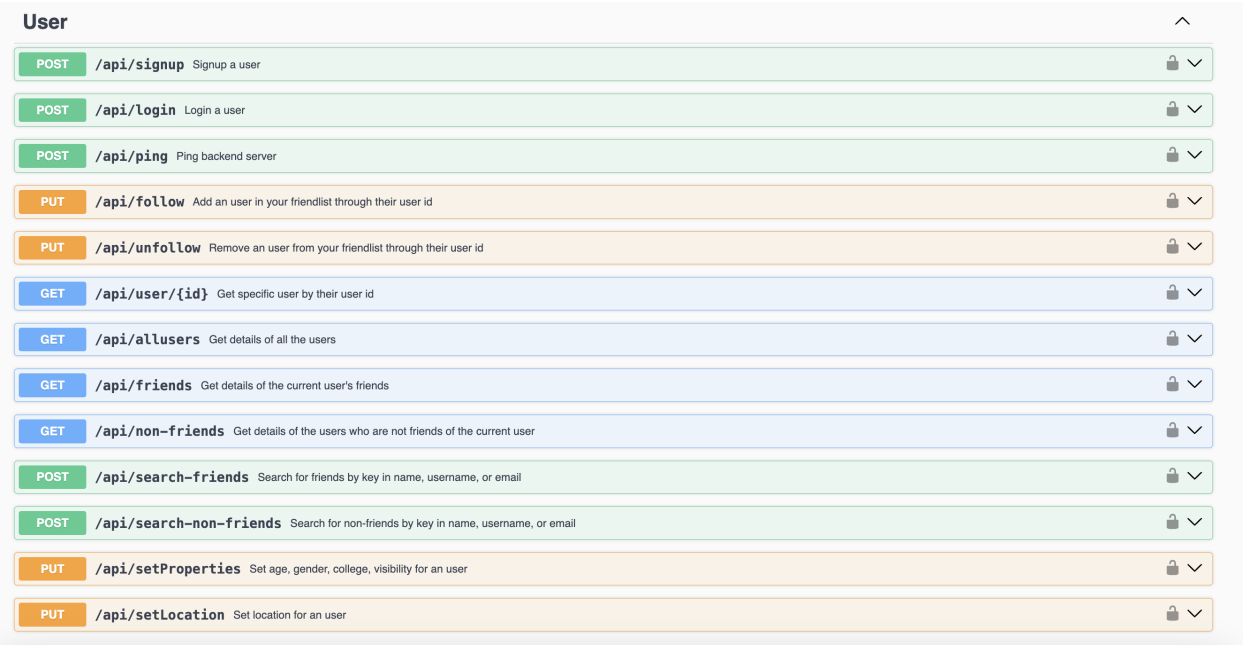


Figure 3.23: Swagger Documentation: API Endpoints

Figure 3.24: Swagger Documentation: Data Schemas

Chapter 4

Conclusion

This thesis focuses on the implementation of a security-conscious Geo Social Networking web application, named *TraceBook*. To address privacy concerns, *TraceBook* prioritizes user control over location data. Users can define their visibility settings with granular control, ensuring a secure and customizable social media experience.

The front-end technologies used to develop the web application are varied and robust. React, a component-based UI library, is employed to create a smooth and interactive user experience. For styling, the application utilizes *Material UI*, *TailwindCSS*, and raw CSS to ensure a cohesive and visually appealing design. Additionally, *Vite* is used as the build tool, providing a lightning-fast development server and contributing to an overall smoother development experience.

The backend of the web application employs a distributed architecture to ensure scalability and efficiency. A MongoDB server is used to store user authentication and profile data, providing a reliable and efficient database solution for these critical components. For handling user location attributes, a Postgres server is implemented, utilizing PostGIS to enable precise location-based queries. Real-time communication is facilitated by a WebSocket server, which allows for immediate location updates and interactions through a bidirectional full-duplex communication channel. Furthermore, comprehensive documentation for all backend API endpoints is provided using OpenAPI Specs. This documentation adheres strictly to standardized OpenAPI specifications and is accessible in the well-known Swagger format, ensuring clarity and consistency for developers.

Git is used for version controlling, and GitHub hosts the git repository of our application codebase. Platforms like *Vercel* and services like *Render* are utilized to deploy the web application's frontend and backends respectively, while GitHub Actions CI/CD pipelines ensuring smooth deployments.

By prioritizing user privacy and utilizing scalable architecture, *TraceBook* lays the groundwork for a secure and dependable Geosocial Networking platform.

TraceBook can be accessed at ***https://privacynetwork.sanam.live***

Chapter 5

Limitations and Future Work

In this chapter, we discuss a few of the bottlenecks, limitations and technical difficulties that exist in our current approach of building *TraceBook*. In particular, challenges in scaling the system, making it consistent, available and distributed are touched upon. We suggest some of the possible remediation for the same. In the later part of the chapter, we discuss the future scope of the application and propose some more challenging yet relevant features that can be incorporated into *TraceBook*.

5.1 Limitations

TraceBook uses a Websocket architecture for the real-time location sharing among the users. However, WebSockets are difficult to scale.

5.1.1 Scalability

While a single server can theoretically handle 65,536 TCP connections[23] due to the maximum number of available ports, this doesn't directly translate to the same limit for WebSocket connections. A single connection via the `ws:// protocol` is stateful and reserves resources in our *TraceBook* WebSocket server throughout the time a client remains connected with our backend. The connection is blocking in nature. Roughly 20,000 concurrent connections are possible for a 8GB RAM machine, and vertical scaling is capped by hardware limitations.

Resource Consumption: Each open WebSocket connection consumes server resources, including memory, CPU, and network bandwidth. The amount of resources used per connection depends on factors like message frequency, message size, and processing complexity. A server with limited resources may reach its capacity well before reaching the port limit, resulting in performance degradation or connection drops.

Thread Management: Traditional server models often rely on threads to manage individual connections. However, creating and managing a large number of threads can be inefficient and lead to overhead. This is because context switching between threads consumes resources and can become a bottleneck for high connection counts.

5.2 Future Work

We present a multi-faceted approach that tackles scalability issues from various angles, aiming to significantly increase the number of concurrent WebSocket connections a single server can handle. By implementing these propositions in the future, we aim to achieve a significant improvement in scalability, enabling our application to handle a much larger volume of concurrent WebSocket connections efficiently and hence accommodate a larger number of simultaneous users.

5.2.1 Scaling WebSocket Servers with Redis Pub-Sub and HAProxy

As the demand for real-time communication, location sharing and low-latency interactions between clients and the *TraceBook* servers continues to escalate, the traditional paradigms of web communication face challenges in scaling effectively. WebSocket, with its stateful nature, presents a promising solution, yet its scalability remains a concern [24]. This section explores a solution for scaling WebSocket servers to handle high volumes of concurrent connections and real-time communication demands.

Traditional HTTP servers excel in horizontal scaling due to their stateless nature. However, WebSockets, being stateful protocols, present challenges when scaling. A single WebSocket server maintains client state, and simply adding more servers doesn't inherently replicate this state across them. Additionally, a single server cannot efficiently broadcast messages to a large number of connected clients. Also note that, horizontal scaling alone is insufficient for replicating states across multiple websocket servers, while vertical scaling encounters hardware limitations.

The proposed architecture leverages *Redis Pub-Sub* for message broadcasting and keeping the connections stateful, while HAProxy is being used for load balancing client connections across multiple WebSocket server instances.

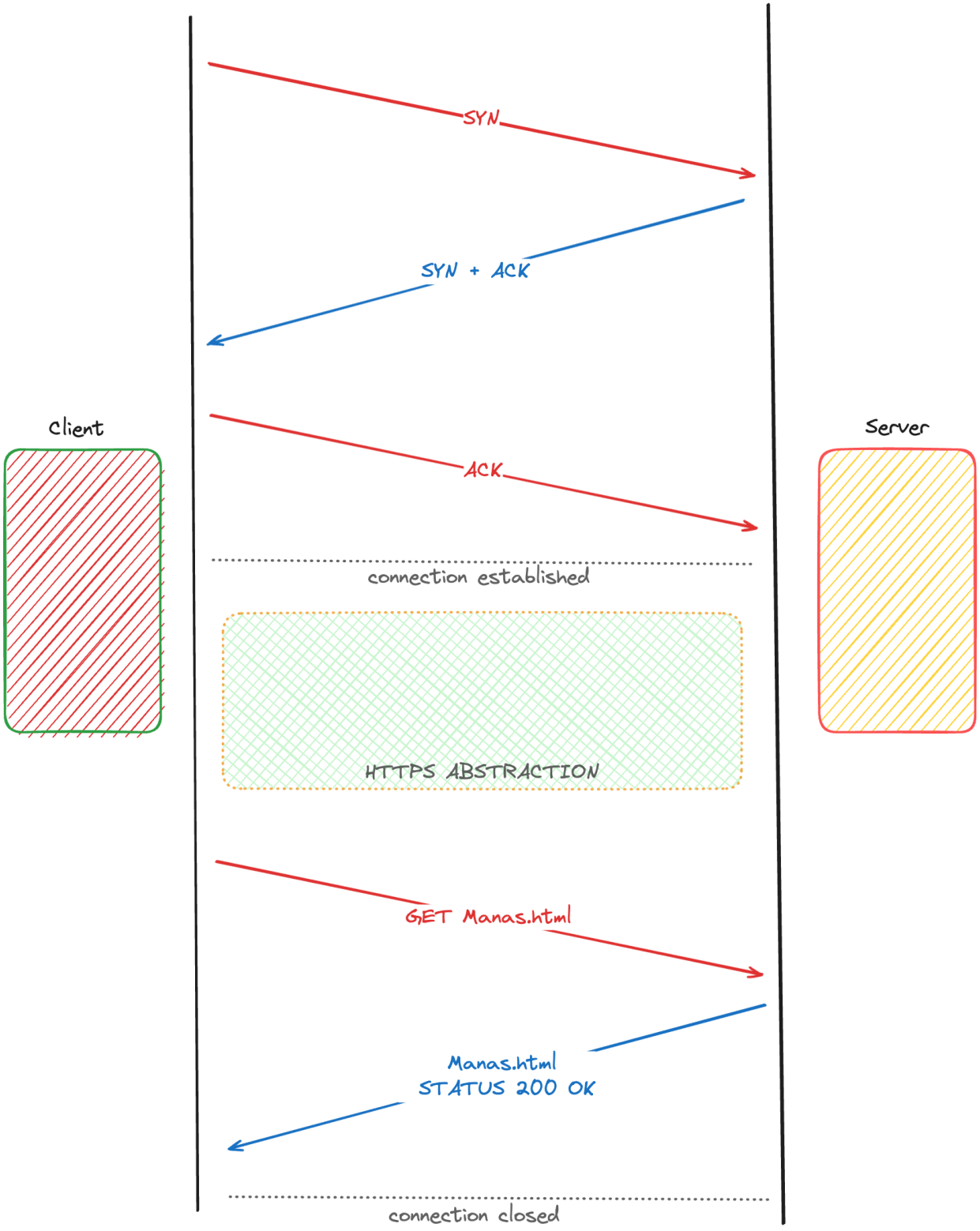


Figure 5.1: HTTP: A Stateless Protocol

Millions of clients can be served independently across multiple HTTP servers that are load balanced by reverse proxy servers, making it horizontally scalable.

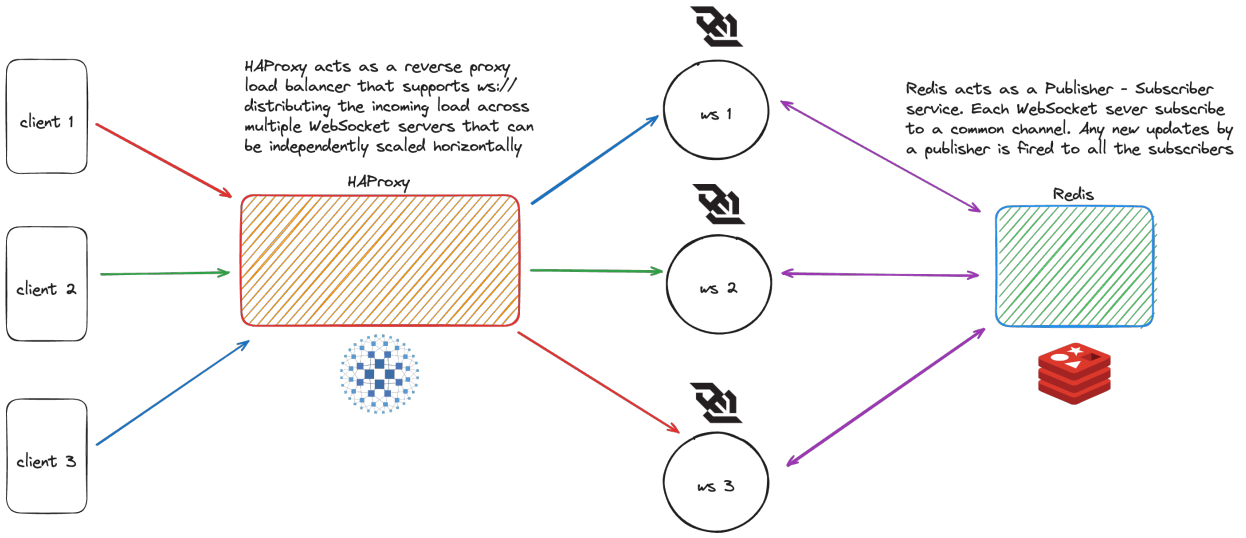


Figure 5.2: A Scaled WebSocket Architecture

- **Redis Pub-Sub**: Acts as a message broker, enabling real-time communication between WebSocket servers. Clients connect to individual WebSocket servers. When a server needs to broadcast a message, it publishes it to a dedicated *Redis* channel. All subscribed WebSocket servers receive the message and can then forward it to their connected clients. This approach decouples message broadcasting from individual WebSocket servers, improving scalability.
- **HAProxy**: Functions as a load balancer, distributing incoming client connections across the pool of WebSocket servers[25]. This ensures even distribution of client load and prevents overloading any single server. Additionally, HAProxy can implement health checks to identify and remove failing WebSocket servers from the pool, maintaining overall system availability.

This architecture offers several advantages for scaling WebSocket applications:

- **Horizontal Scalability**: By adding more WebSocket servers to the pool behind HAProxy, the system can handle increasing client connections.
- **Real-time Communication**: *Redis Pub-Sub* facilitates efficient broadcasting of messages to all connected clients with low latency.
- **High Availability**: HAProxy ensures continuous service even if individual WebSocket servers fail.
- **Improved Performance**: Distributing client load across multiple servers enhances overall application responsiveness.

While this architecture provides a robust foundation for scaling WebSockets, some limitations require consideration

- **Redis Pub-Sub Single Point of Failure (SPOF)**: Although *Redis* offers high availability configurations, it can still be a SPOF. Exploring alternative *pub-sub* solutions with inherent high availability, like *Apache Kafka*, could enhance fault tolerance.
- **State Management Complexity**: Maintaining consistency across replicated state stores can be challenging. Implementing distributed locking mechanisms or optimistic locking could ensure data integrity during concurrent state updates.
- **Distributed Cache Invalidation**: The effectiveness of a distributed cache layer hinges on proper invalidation strategies. Implementing cache invalidation mechanisms like cache expiration or message bus invalidation could optimize cache utilization.

Future work directions can further improve the architecture:

- **Integration with Container Orchestration Platforms**: Investigating seamless integration with container orchestration platforms like Kubernetes could enable dynamic scaling of WebSocket servers based on real-time traffic demands.
- **Load Balancing Algorithm Optimization**: Exploring more advanced load balancing algorithms within HAProxy, such as weighted round-robin or least connections, could potentially improve client distribution across WebSocket servers.
- **WebSocket Protocol Extensions**: Researching the utilization of WebSocket protocol extensions for message fragmentation and compression could further optimize real-time communication efficiency.

5.2.2 Change Data Capture for Synchronization of SQL-NoSQL Databases

This section discusses the limitations of the current approach for synchronizing data between the PostgreSQL database and the MongoDB database, and proposes a future work direction using *Apache Kafka* for Change Data Capture (CDC)[26] to achieve real-time data consistency across these heterogeneous databases.

Current Approach and Limitations:

The current solution synchronizes common user data fields (`college, age, gender, isVisible`) from MongoDB to PostgreSQL upon updates in MongoDB. This approach offers several benefits:

- **Reduced Redundancy**: User data is not directly updated in PostgreSQL, minimizing redundancy.

- **Focus on MongoDB Updates**: Since CRUD operations primarily occur in MongoDB, this simplifies the synchronization logic.
- **ACID Compliance**: Updates are performed sequentially, ensuring ACID properties within each database.

However, a critical limitation exists:

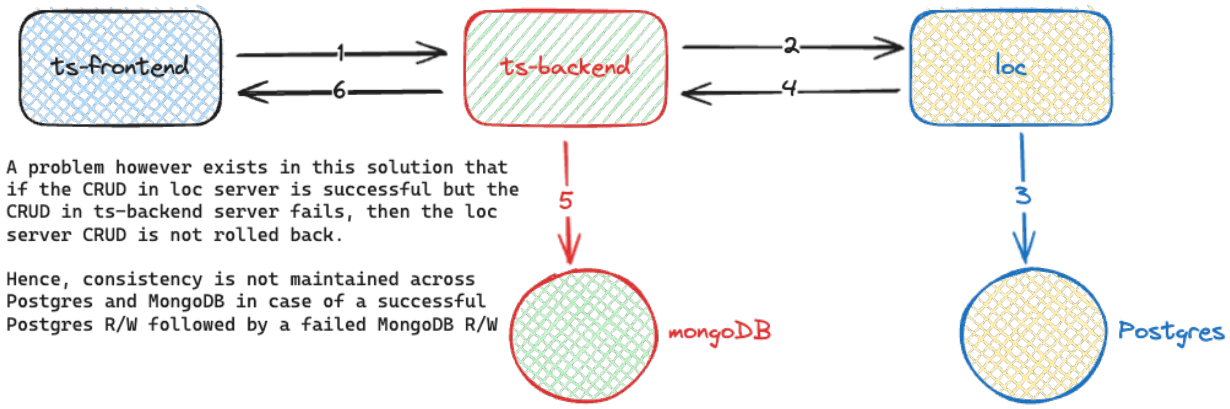


Figure 5.3: Database *Write-Through* Approach

- **Eventual Consistency Issue**: If a successful update in PostgreSQL is followed by a failed update in MongoDB, data consistency is compromised. This scenario creates an orphaned update in PostgreSQL, as the corresponding MongoDB update fails to reflect the change.

To address the eventual consistency issue and achieve real-time data synchronization across PostgreSQL and MongoDB, we propose implementing *Kafka CDC*. Here's how it would work:

- **Change Capture on Both Databases**: *Kafka* connectors would be deployed to capture changes (*inserts*, *updates*, *deletes*) from both PostgreSQL and MongoDB in real-time.
- **Publishing Changes to *Kafka* Topics**: Captured changes would be published as messages to dedicated *Kafka* topics for each database (e.g., `"postgres_changes"` and `"mongodb_changes"`).
- **CDC Consumer Service**: A dedicated consumer service would subscribe to both *Kafka* topics. Upon receiving messages, it would perform the following actions:

  - **For PostgreSQL Changes**: If the message originates from the `"postgres_changes"` topic, the consumer service would attempt to update the corresponding data in MongoDB.
  - **For MongoDB Changes**: Similar to above, if the message originates from the `"mongodb_changes"` topic, the consumer service would update the corresponding data in PostgreSQL.

- **Error Handling and Transaction Management**: The consumer service would implement robust error handling mechanisms. In case of update failures in either database, the service would attempt retries or trigger rollback mechanisms to maintain data consistency across both databases.

The immediate benefits upon introducing the above architecture is as follows:

- **Real-Time Synchronization**: Updates are reflected in both databases with minimal latency, ensuring data consistency.
- **Bi-directional Synchronization**: The solution allows for updates originating from either database to be reflected in the other, providing flexibility.
- **Improved Scalability**: Kafka's distributed architecture facilitates horizontal scaling to handle large data volumes and high update rates.

Table 5.1: Comparison of *Kafka* and *Redis*

Parameter	**Kafka**	**Redis**
Message Size	Supports message size up to 1GB	Supports smaller message size
Message Delivery	Subscribers pull messages from queue	Servers pushes messages to subscribers
Message Retention	Retains message after retrieval	Does not retain message
Latency	Low latency. Slightly slower than Redis due to data replication	Ultra low latency when distributing smaller-sized messages
Parallelism	Supports parallelism, multiple consumers	Does not support parallelism
Throughput	High throughput - Asynchronous read/write	Lower throughput - Server waits for reply

5.2.3 Integrating a Redis Distributed Caching Layer

While the proposed architecture utilizing *Kafka CDC* effectively synchronizes data across relational and non-relational databases, further optimization can be achieved by introducing a *Redis* distributed caching layer[27]. This section explores the potential benefits and considerations for implementing a *Redis* cache.

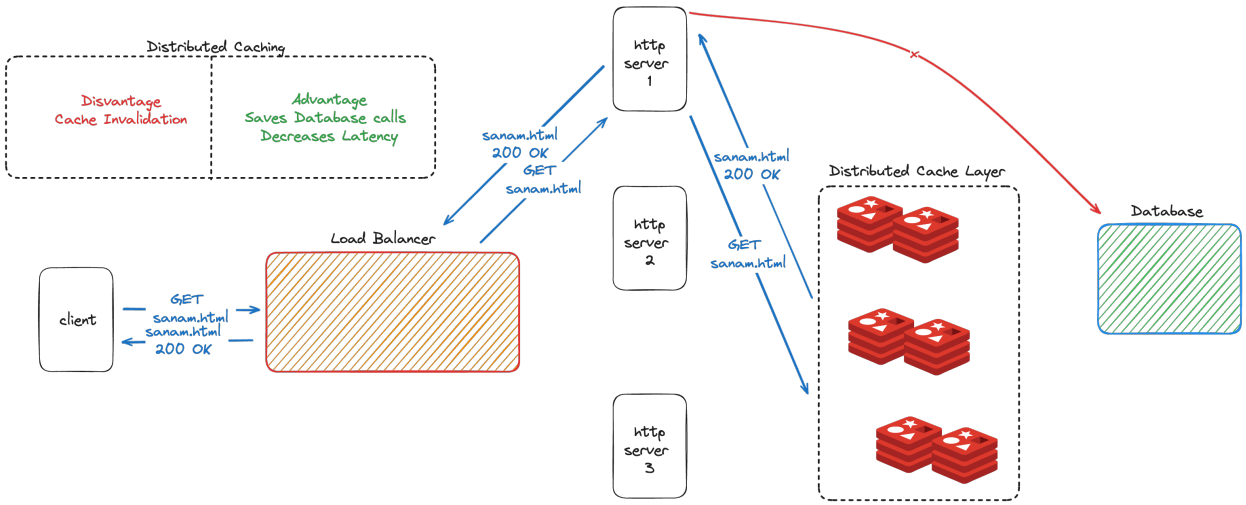


Figure 5.4: Distributed Redis Caching Layer

Potential Benefits

- **Reduced Network Traffic**: Frequently accessed data can be cached in memory by *Redis*, minimizing the number of database calls required. This can significantly improve application performance, especially for read-heavy workloads.
- **Faster Response Times**: By serving data from the in-memory cache, response times for data retrieval operations can be significantly reduced compared to traditional database access.
- **Improved Scalability**: Caching can effectively offload database load, allowing the system to handle increased traffic without overwhelming the databases.

We explored several promising avenues for future work that can enhance the robustness, scalability, and performance of the proposed architecture. By pursuing these future work directions, the *TraceBook* architecture can be continuously refined to deliver a highly scalable, secure, and performant solution for protecting user data and enabling privacy-preserving location sharing.

Bibliography

[1] Wikipedia, "Geolocation networking," 2008-Present.

[2] Lexa and G. P. Store, "Geolocation networking."

[3] H. Li, H. Zhu, S. Du, X. Liang, and X. Shen, "A privacy leakage of location sharing in mobile social networks: Attacks and defense," in *IEEE Transactions on Dependable and Secure Computing*, 2018.

[4] BBC, "Meta settles cambridge analytica scandal case for $725m," 2022.

[5] M. Gruteser and D. Grunwald, "Anonymous usage of location-based services through spatial and temporal cloaking," in *Proc. 1st Int. Conf. Mobile Syst., Appl. Services (MobiSys)*, 2003.

[6] H. Kido, Y. Yanagisawa, and T. Satoh, "Protection of location privacy using dummies for location-based services," in *Proc. 21st Int. Conf. Data Eng. Workshops (ICDEW)*, 2005.

[7] A. Khoshgozaran and C. Shahabi, "Blind evaluation of nearest neighbor queries using space transformation to preserve location privacy," in *Springer*, 2007.

[8] A. R. Beresford and F. Stajano, "Location privacy in pervasive computing," *IEEE Pervas. Comput*, 2003.

[9] M. Rahman, M. Mambo, A. Inomata, and E. Okamoto, "An anonymous on-demand position-based routing in mobile ad hoc networks," in *Proc. Int. Symp. Appl. Internet (SAINT)*, 2006.

[10] W. Wei, F. Xu, and Q. Li, "Mobishare: Flexible privacy-preserving location sharing in mobile online social networks," in *Proc. IEEE INFOCOM*, 2012.

[11] X. Chen, Z. Liu, and C. Jia, "Mobishare+: Security improved system for location sharing in mobile online social networks," *J. Internet Serv. Inf. Secur.*, 2014.

[12] J. Li, H. Yan, Z. Liu, X. Chen, X. Huang, and D. S. Wong, "Location-sharing systems with enhanced privacy in mobile online social networks," *IEEE Syst. J.*, 2017.

[13] X. Xiao, C. Chen, A. K. Sangaiah, G. Hu, R. Ye, and Y. Jiang, "Cenlocshare: A centralized privacy-preserving location sharing system for mobile online social networks," *Future Gener. Comput. Syst.*, 2018.

[14] M. Bhattacharya, S. Roy, K. Mistry, H. P. H. Shum, and S. Chattopadhyay, "A privacy-preserving efficient location-sharing scheme for mobile online social network applications," 2020.

[15] Y. Combinator, "Loopt," 2005.

[16] B. B. Nerd, "Y combinator's first batch (yc so5)," 2024.

[17] LoopTMix, "Loopt: The geo social network."

[18] WikiPedia, "Node.js."

[19] Nodejs, "What is the event loop?."

[20] T. F. Stack, "The unbelievable history of the express javascript framework," 2016.

[21] P. Newswire, "Ibm acquires strongloop to extend enterprise reach using ibm cloud," 2015.

[22] Expressjs, "Writing middleware for use in express apps."

[23] uNetworking AB, "Millions of active websockets with node.js," 2019.

[24] Alby, "The challenge of scaling websockets," 2023.

[25] HAProxy, "Haproxy configuration for websocket."

[26] Confluent, "What is change data capture?."

[27] Redis, "Distributed redis caching."