

ΕΛΛΗΝΙΚΟ ΜΕΣΟΓΕΙΑΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ
ΤΜΗΜΑ ΜΗΧΑΝΙΚΩΝ ΠΛΗΡΟΦΟΡΙΚΗΣ

Πτυχιακή εργασία

NoSQL συστήματα σε σύγχρονες εφαρμογές εταιρικών κοινωνικών δικτύων: Μελέτη Περίπτωσης: Neo4j

Αλέξανδρος Μεντουδάκης (ΑΜ: ΤΠ5066)

Επιβλέπον καθηγητής: Δημοσθένης Ακουμιανάκης

Ηράκλειο, Ιούλιος 2024

Acknowledgements

First, I would like to thank my parents for their support, understanding and patience that have showcased all those years. In addition, I want to express my sincere appreciation to my supervisor, Dr. Demosthenes Akoumianakis, for everything he has done throughout this thesis, all those discussions we had and both the advices and support that he has offered me. I am grateful for this collaboration since his insightful feedback has not only played such a crucial part in both the formation and outcome of this research but also in the comprehension of many important aspects of computer science and life in general. I could not complete this paragraph, without mentioning a co-student and a good friend of mine, Manos Nitis, who I was lucky enough to meet during my years in university whose help and support was invaluable. Finally, I want to give the appropriate credit to myself, because even thought I was one step from giving up everything associated with my studies 4 years ago, I managed to get it together and fulfill the promise that I had given to myself at that time. To summarize, it was quite an interesting journey that has only make me wiser.

Table of Contents

Acknowledgements 2

Περίληψη 6

Abstract 7

1. Introduction 8

1.1. Thesis Objectives 9

1.2. Scope and Limitations of Thesis 9

1.3. Thesis organization 10

2. Theoretical Background 10

2.1. Social networks 10

2.2. Enterprise Social Networks (ESNs) 12

2.3. NoSQL systems, native graph stores and path querying 14

2.3.1. Definition and challenges 14

2.3.2. Native graph stores and the Property Graph data model 14

2.3.3. Paths and Path Queries 16

2.4. Neo4j and Cypher Language 17

2.4.1. Neo4j 17

2.4.2. Cypher 18

3. Building an ESN: Key technologies 21

3.1. Neo4j Aura account 21

3.2. Node.js installation 22

3.3. Flutter (front end, UI) 24

4. The ESN Graph Data Model and App design 26

4.1. Nodes and relationships 27

4.2. Front – Back End Modeling 29

5. The ESN App – Scenarios of use 33

5.1. Registration and authentication 33

5.2. Home Screen 35

5.3. Search Screen **36**

5.4. Network Management **36**

5.5. Connection Mechanism **37**

5.6. Messaging Mechanism **39**

5.7. Posting **41**

5.7.1. Uploading a post **41**

5.7.2. Interacting with a Post **43**

5.7.2.1. Liking a Post **43**

5.7.2.2. Commenting a Post **44**

5.8. Team Management **46**

5.8.1. My Teams Screen **46**

5.8.2. Creating a Team **47**

5.8.2.1. Single Team's Screen **49**

5.8.2.2. Inviting New Members **49**

5.8.3. Team Chat Rooms **52**

5.8.3.1. Creating a public team chat room **52**

5.8.3.2. Sending a team Message **55**

5.8.4. Members Screen **58**

5.8.5. Uploading a Team Post **59**

5.8.6. Invitations Screen **61**

5.8.7. Discover Teams Screen **62**

6. Illustrative path queries examples using Cypher ***64***

7. Conclusion ***69***

7.1. Assessment ***69***

7.2. Future research and extensions ***69***

8. References ***71***

Περίληψη

Η παρούσα διπλωματική εργασία παρουσιάζει την ανάπτυξη μιας εφαρμογής εταιρικού κοινωνικού δικτύου. Η υλοποίησή της έγινε χρησιμοποιώντας τις ακόλουθες υπηρεσίες, το Flutter για την δημιουργία του front end, από το οποίο οι χρήστες μπορούν να αλληλοεπιδράσουν με το περιβάλλον χρήστη της εφαρμογής, το Node.js ως server (διακομιστής), όπου προορίζονται να φτάσουν οι κλήσεις των αιτημάτων πρωτοκόλλου μεταφοράς υπερκειμένου (HTTP requests) και τη Neo4j (βάση δεδομένων σε μορφή γράφου) ως βάση δεδομένων του έργου. Ο στόχος της εφαρμογής είναι η παροχή μιας πλατφόρμας στους χρήστες μέσω της οποίας μπορούν να δημιουργήσουν συνδέσεις, να επικοινωνήσουν, να δημιουργήσουν ομάδες αλλά και να κοινοποιήσουν διάφορα είδη περιεχομένου σε ένα ασφαλές και αποτελεσματικό περιβάλλον. Αξιοποιώντας τις δυνατότητες του Flutter για δημιουργία εφαρμογών σε πολλαπλές πλατφόρμες, η διεπαφή του χρήστη με την εφαρμογή έχει σχεδιαστεί για να είναι διαισθητική και να ανταποκρίνεται σε διάφορες συσκευές. ToNode.js παρέχει ισχυρή λειτουργικότητα από την πλευρά του διακομιστή, χειρισμό ελέγχου ταυτότητας χρήστη (αυθεντικοποίηση χρήστη), επεξεργασία δεδομένων και αλληλεπίδραση με τη βάση δεδομένων Neo4j. Η δομή σε σχήμα γράφου της Neo4j επιτρέπει την αποτελεσματική αποθήκευση και ανάκτηση πολύπλοκων σχέσεων, ζωτικής σημασίας για τη μοντελοποίηση των συνδέσεων του κοινωνικού δικτύου. Μέσω αυτής της υλοποίησης, η διατριβή διερευνά την ενσωμάτωση αυτών των τεχνολογιών σε ένα έργο προκειμένου να δημιουργηθεί μια επεκτάσιμη, γρήγορη σε απόκριση και πλούσια σε χαρακτηριστικά εφαρμογή κοινωνικής δικτύωσης, προσφέροντας πληροφορίες για τις προκλήσεις και τις ευκαιρίες της δημιουργίας ψηφιακών κοινωνικών δικτύων για σύγχρονες πλατφόρμες. Σε όλη αυτή την εξερεύνηση, η διατριβή ρίχνει φως στις μετασχηματιστικές δυνατότητες της Neo4j ως καταλύτη καινοτομίας στον τομέα των εφαρμογών κοινωνικής δικτύωσης. Αξιοποιώντας τη δύναμη της τεχνολογίας της βάσης δεδομένων γραφημάτων, μπορούμε να χρησιμοποιήσουμε αναλυτικά στοιχεία για να εξάγουμε χρήσιμα/κρίσιμα δεδομένα από τις δημιουργημένες σχέσεις μεταξύ των χρηστών και από τις αλληλεπιδράσεις τους με την εφαρμογή (ψηφιακά ίχνη), έτσι ώστε όχι μόνο να μπορούμε να δημιουργήσουμε το ψηφιακό τους προφίλ αλλά και να προσαρμόσουμε ανάλογα το περιεχόμενο της διεπαφής τους με αυτήν, προσφέροντας όσο τον δυνατόν καλύτερη εμπειρία χρήστη.

Abstract

This thesis presents the development of an enterprise social network application. Its implementation has been using the following services, Flutter as the front end, from which users' can interact with the applications UI, Node.js as the back end (server), where the API calls are destined to reach and Neo4j (graph database) as the database of the project. The application aims to provide users with a platform for connecting, communicating and sharing content within a secure and efficient environment. Leveraging Flutter's cross-platform capabilities, the user interface is designed to be intuitive and responsive across various devices. The Node.js back end provides robust server-side functionality, handling user authentication, data processing and interaction with the Neo4j database. The Neo4j's graph structure enables efficient storage and retrieval of complex relationships, crucial for modelling social network connections. Through this implementation, the thesis explores the integration of these technologies into one project, whose goal is the creation of a scalable, quickly responsive and feature-rich social networking application, offering insights into the challenges and opportunities of building modern digital social network platforms. Throughout this exploration, the thesis sheds light on the transformative potential of Neo4j as a catalyst of innovation in the realm of social networking applications. By harnessing the power of graph database technology, we can use analytics to extract useful/crucial data from the users' created relationships and from their interactions with the application (digital traces), so that we can not only create their digital profile but also customize their interface's content, accordingly, thus providing better user experience.

1. Introduction

Social Networks in general serve as digital platforms where individuals can connect, interact and share information with each other. These platforms facilitate the creation, sharing and exchange of content such as text, images and videos. Users can typically create profiles, establish connections with others (friends, followers, etc.) and engage in various activities such as posting, commenting, liking and messaging [1]. Social networks vary in purpose and scope, ranging from general interest platforms to specialized networks for professional connections or just for sharing visual content. They play such a significant role in modern communication, fostering connections, facilitating collaboration and shaping online communities. By adopting those features, integrating them into a corporate environment and adding real-time collaboration and management tools, we transition from general purpose social networks to Enterprise Social Networks (ESNs), which are specialized platforms designed to enhance internal communication, knowledge sharing and collaboration within an organization. Later, this thesis will also conduct a more detailed reference about ESNs implementation's benefits and drawbacks for organizations. However, regardless to social network's type, their data collection plays a crucial role in improving the user's experience with the associated application, since its content can be customized and filtered to each user's preferences and needs, but also to make useful assumptions about that corresponding technology's adoption and success rate. Additionally, with the exponential growth of user data, it has become essential to leverage advanced management and analytics techniques to extract valuable insights from this wealth of information.

Taking into consideration this enormous amount of data, this thesis delves into the intersection of social networks and NoSQL databases, specifically native graph data stores, with the implementation of a social network database using Neo4j – a leading graph database technology. The primary objectives of this thesis are to explore the application of graph databases in the context of social networks. By utilizing Neo4j's capabilities, we aim to construct an efficient and robust social network to appropriate the benefits of graph databases in providing powerful insights not only into users' interactions with each other but also into their digital traces though the social network's dedicated mechanisms such as connecting users, expressing opinion, commenting, etc.). A secondary target is to evaluate the performance, effectiveness and quickness of path querying which is the prominent form of querying in native graph datastores.

1.1. Thesis Objectives

The objective of this thesis is two-fold. a two-way journey. First, we seek to familiarize ourselves with graph-based data models and assess their capabilities in relation to the classical relational model of data and the supporting relational technologies (RDBMS). In this effort, we are particularly keen on the label Property Graph data model which is the underlying data model supported by Neo4j. We set out to achieve this objective by designing and developing a social network application using cutting-edge and widely deployed technologies, Flutter and Node.js to couple the Neo4j graph datastore. By doing so, we aim to unwind the capabilities and advantages of Neo4j, a leading graph database management system, in an increasingly important domain such as digital social networking.

1.2. Scope and Limitations of Thesis

The scope of the thesis can be briefly summarized as follows:

- This thesis explores the application of Neo4j, a graph database management system, in a social network application to highlight Neo4j's capabilities to efficiently store and retrieve interconnected user data, thus enabling complex graph-based querying.
- The thesis showcases practical implementations of Cypher queries and graph utilities for social network data, providing valuable insights into relationships that can be created in a social network application.

In terms of thesis limitations, is it fair to admit the following:

- By focusing primarily on the application of Neo4j in social network data, our work may not extensively cover other types of graph databases or NoSQL databases.

Overall, the thesis set out to present a through exploration of Neo4j's capabilities in social network applications, specifically focusing on users' connections and interactions with the application, thereby demonstrating the practicality of graph databases.

1.3. Thesis organization

This thesis workflow begins with the detailed description of both Social Networks (SNs) and the theory of NoSQL Database Management Systems (DBMS), emphasizing in the categories of Enterprise Social Networks (ESNs) and Graphs respectively. Following that, the thesis will focus on the Neo4j database and its corresponding language, Cypher. After that, there will be a reference to the technologies that were used to implement the ESN mobile application's project, alongside their installation's steps. Furthermore, we will focus on the modelling part of the application for both the front and back end since the NoSQL DBMS are schema-less. Finally, we will demonstrate all the scenarios of use that exist within the application, accompanied with screenshots from both the User Interfaces and Neo4j database, and then conclude by providing some examples of path queries implemented with Cypher Language.

2. Theoretical Background

This chapter reviews the theoretical and technological strands upon which the present work is grounded. Specifically, we set off by elaborating briefly the concepts of social networks and the special case of enterprise social networks to anchor the wider context of the work. Then we review challenges and promises of NoSQL systems with an emphasis on native graph stores which are the primary technological focus of our work. The two themes are strongly interrelated as social networks serve as repositories of vast amount of data whose processing and analytical exploitation establish the rationale for moving away from traditional RDBMS to NoSQL systems.

2.1. Social networks

Electronical Social Networks or social networks for short, are digital platforms that allow users and organizations to connect, communicate, share information and form relationships [2]. Social networks play a crucial role in everyday life, as they enable people to connect with friends and family or others having the same or similar interests or living in the same area, etc. At core, social networks facilitate connection and communication, interaction with others, sharing of information and maintenance of relationships regardless of geographic location or time zones. Their ease of connectivity, "breeds" the creation of communities and the exchange of ideas. Additionally, social

network platforms serve as forums for information dissemination, enabling the rapid spread of news and trends. In recent years a wide variety of social networking applications have emerged playing a crucial role in a range of enterprise-wide activities such as advertising, business operations and marketing strategies [3]. For many researchers, the key factor driving the adoption of these social networking applications is that they provide companies with the ability to reach out to potential new customers [4].

Social networks are one of the most important uses of the internet today. Social scientists observe that social networking may potentially lead to the creation of virtual communities where people can interact, exchange opinion and share knowledge with others. Presence in these communities requires a profile in which users declare (typically) a code name, photos they wish to share, preferences or domains of interest and much more. In many cases user profiles may be populated by additional data maintained or traced by the application itself. On the grounds of such data users can connect with others by sending friend requests, following profiles or joining groups based on shared interests.

Communication is a fundamental aspect of social networks, facilitated via various mechanisms such as messaging, commenting and sharing. Their users can engage in real time conversations and exchange views and opinions in every-day matters. Social networks also serve as hubs for content discovery and consumption. Users can be informed from news feeds or from people and pages they follow about important issues, as well as trending topics tailored to their own interests. Additionally, social networks offer a wide range of tools to enhance user experience and engagement [5]. They (tools) might provide features like live streaming, story sharing, event creations gaming and marketplace functionalities, depending on each platform's content and purpose. Social networks can be categorized into various types based on their corresponding purpose, target audience or nature of interactions they implement.

- **General Social Networks:** This type of social network provides users with mechanisms that enable them to share updates and stay in touch with their friends, family members and others or even find new friends through similar interests.
- **Professional Networks:** These social networks are designed for business relationships. They can be used to make new professional contacts, enhance

business connections and explore job opportunities. They might also utilize mechanisms to connect co-workers.

- **News or Informational Networks:** These social networks provide users with the utility to post news stories, informational or how-to content. They also include communities of people who are interested in finding answers to everyday problems and are like web forums. By providing answers to various problems or teaching how to perform different tasks, their members foster a sense of helping.
- **Communication Networks:** This kind of social network focuses upon allowing users to directly communicate with each other in one-on-one or group chats.
- **Media Sharing Networks:** These platforms center around sharing and discovering multimedia content such as photos and videos.
- **Educational Networks:** They offer remote learning, enabling students and teachers to collaborate on school projects, conduct research and interact through blogs and forums.
- **Enterprise Social Networks:** These platforms provide a specialized type of general social networks, since they offer similar features, such as profiles, news feeds, groups, messaging and content sharing, but they are designed for usage within a corporate environment, thus enabling collaboration and communication between colleagues [4].

It is essential to comprehend that even though social networks can be separated into the above categories, depending on their purposes and implementation techniques, there are applications that adopt more than one utility, to build a multidimensional platform, where users can benefit from every single of provided capability. By doing so, they quip their users with the right tools to perform various needed tasks from one place, thus saving their valuable time and energy, searching for different kind of platforms to fulfill those needs.

2.2. Enterprise Social Networks (ESNs)

Enterprise social networks or ESNs is defined as the way organizations use social networks to connect employees, business processes and internal activities [4]. While most of the features, which ESNs provide, are pretty much like any other general social

network platform's, they significantly differ upon their scope and accessibility. ESNs integrate all these social networking features into the corporate environment, promoting a more cohesive and efficient workforce inside a specific organization. Since this type of networking only addresses people within that corresponding organization, it can also be described as a private social network. Nowadays, many companies utilize the use of ESNs in their efforts to make collaboration and communication between their employees simpler, more convenient and faster. At times company employees may need to work with an external team to launch a new product or service. ESN's implementation fulfills the crucial need for a controlled environment, in which those shared projects can be conducted, for security, monitoring, management issues. Furthermore, remote working for members of a single team has become more and more popular over recent years, so it is essential for them to possess the right tools, to maximize their engagement, collaboration but also ideas and knowledge sharing. When these tools are used together and in the appropriate context, it translates into increased employee engagement and higher job morale, resulting in more productive and satisfied employees.

Studies have shown, that upon receiving the maximum value out of ESN's integration within an organization, that platform needs to be the main source of communication and collaboration for that corresponding company, since continuous context switching can not only lead to loss of productivity but also make employees feel frustrated and overwhelmed [7, 8].

Although Enterprise Social Networks can provide many advantages inside an organization, they can also be the source of some unwanted results, if they are not carefully managed and monitored. Some of the most common drawbacks of an ESN are the following. First and foremost, they can be as distracting as public social media platforms, thus affecting employees' productivity. In addition, receiving messages and notifications at odd hours of the day can make employees frustrated and stressed, also resulting in information overload. Finally, using all these tools for collaboration, document management and project management, it is easy for conversations to reach an out-of-context points, so at the end of the day, employees will be digging through large conversations to find what they really need, wasting valuable time and energy.

Overall, while Enterprise Social Networks (ESNs) offer many potential benefits for organizations, they are also accompanied by several risks and challenges, that need to be carefully managed, to ensure successful implementation and adoption.

2.3. NoSQL systems, native graph stores and path querying

2.3.1. Definition and challenges

Before looking into the details, it is perhaps appropriate to briefly comment on what NoSQL databases bring to the state of the art in the area of data management. To be able to fully understand the reasons behind the implementations of such systems, one should first appreciate the problems to which they offer a solution [9]. First, NoSQL systems in contrast with the RDBMS, are designed to scale horizontally [10, 11], meaning that they scale by adding more servers and not by providing more resources to a single server, thus allowing them to handle high traffic loads efficiently. In addition, these systems (NoSQL) do not require a pre-fixed schema, making it easier for both the storage of any semi-structured or unstructured generated data but also the convenient adaptation and transformation of data models [12]. Furthermore, relational technologies cannot handle efficiently neither the massive volume of data (being generated nowadays by modern applications such as social networks, IoT devices and analytics), nor their complex types, (lack of uniform) structure and interconnections. Due to these complications the implementation of different systems is often seen as one-way road. Finally, the cost of infrastructure for traditional RDBMS, that may require specialized high-performance servers could be significantly reduced, since NoSQL systems can run on commodity hardware that can be replaced or upgraded effortlessly and in no time.

The combination of the above led to the rise of interest in alternative data models and systems such as of graph, document, column-based and more, all coined by the term NoSQL. As a concluding remark, it is crucial to mention that the misleading term "NoSQL" can be described as "Not Only SQL," which implies that if relational technology is suitable for a specific use case where none of the above issues could be occurred, then it should be used. In what follows, we will briefly focus on the graph-family of data stores, since it constitutes a rapidly evolving and much promising category for implementing applications such as the one described in this thesis.

2.3.2. Native graph stores and the Property Graph data model

As soon as we start talking about graphs, our mind immediately thinks of circles connected with lines. They (graphs) are a fundamental structure in databases and

computer science in general. They consist of vertices (nodes) that represent the entities and edges, which are used to signify the connections or interactions between them [13]. Depending on the type of those edges, graphs are divided into two main categories, directed and undirected, indicating whether those relationships have a direction, meaning we end up with a single specified starting and ending node, or not. Our focus will be on the directed ones, since any application which includes user interactions and connections makes them a necessity and because Neo4j (https://neo4j.com/), does not support undirected edges, anyway.

The Property graph model serves as one popular framework used by systems in the graph stores family to organize and retain information in the form of interconnected nodes and edges [13]. At core the Property Graph utilizes nodes to represent entities and edges to store the relationships those entities have with each other. A key feature that makes this model stand out is the fact that both nodes and edges possess properties, pairs of key-value attributes, whose functionality is to furnish those (nodes and edges) with additional details. Another eminent characteristic of the property graph model lies in its adaptability and richness of expression. Nodes and edges possess the capability to contain diverse properties, thereby enabling a comprehensive and intricate portrayal of data. This inherent flexibility renders property graphs highly suitable for modeling a broad spectrum of domains and applications, ranging from social networks and recommendation systems to infrastructure networks and knowledge graphs. An example of a property graph is illustrated below, as shown in Figure 1.

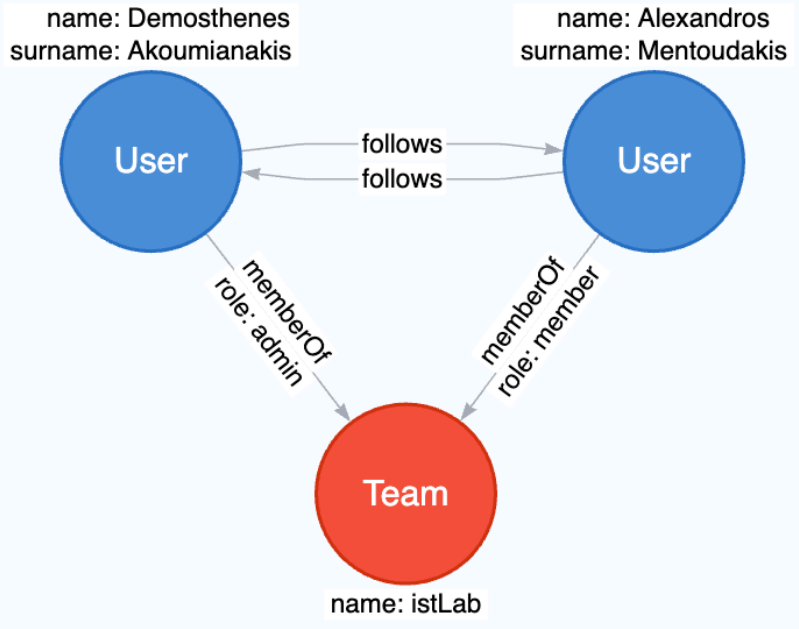


Figure 1: An example of a property graph database.

2.3.3. Paths and Path Queries

Graph databases implementing the property graph model typically furnish query languages and APIs for navigating and querying the graph data. A key concept in navigating graphs in general and property graphs in particular is that of a path which in turn anchors the notion of path queries.

The word "path" can be interpreted in various ways depending on the field that is being used. In general, a path is a sequence of points or procedures which we can follow so that starting from a single position we manage to reach a specified destination or result [14]. In the graph theory field, a "path" is referred to a sequence of interconnected nodes (vertices) and edges in a graph that starting from a single node we can end up to another one. Furthermore, since there are alternative structures of graph data, we end up with slightly different understanding and representations of "path" even in the same field, i.e. hierarchical and interconnected graph data structures, where in the first case there is a single root node that is the starting point, we have (n) nodes and (n-1) edges so we can reach each one of them and in the second case we have multiple sequences of nodes and edges which we can follow, in order to reach a single node starting from another one.

Regarding to the interconnected graph data structures, which is also the case for the SNs and therefore also for our Enterprise Social Network application we come up with different types of paths. First, there is the walk path type where the same vertex can be visited again and the same edge traversed multiple times. In addition, there is the simple path, where the repetition of a vertex or edge is not allowed and the trail where no edge is repeated but vertices might, which is also the principle Neo4j follows for every implemented query. There is also the cycle type, in which a path starts and ends with the same vertex but the repetition of any other vertex or edge is not allowed. Finally, we have the Eulerian and Hamiltonian path types, where in the first case the path visits every edge exactly once and in the second every vertex.

Following our previous reference to paths and their different interpretations according to the field and context of their usage, we reached a point where we had multiple routes thought which we were able to end up on the same destination, starting from the same initial node. So, some reasonable questions could come up to someone's

mind such as, “Which one of the provided paths should I follow and why”, “Are we wasting valuable time searching for every path without providing any extra value” and “How can we optimize this procedure”. This kind of concerns and complications led to the implementation of path queries mechanisms, which are used for optimized graph traversal and information retrieval.

They entail the precise specification of a group of criteria sequence of steps that traverse through a graph, defined by nodes and edges, with the primary objective of extracting targeted information or ascertaining relationships between nodes.

The key constituents of a path query encompass the following:

- **Starting Node**: This denotes the point of initiation for the query within the graph.
- **Ending Node**: This signifies the traversal's ultimate destination or termination point.
- **Intermediate Nodes**: These represent nodes that the query may sequentially visit during traversal.
- **Edges**: These interconnecting relationships or links facilitate the traversal between nodes.

However, to make the most out of path queries, there was a need to develop brand-new query languages and tools that could not only support all the theory upon which they were constructed but also make them simpler to use and much more optimized than the existing ones. In this thesis, we aim to demonstrate the ease of use and capabilities of Neo4j, a member of the graph-based family database management systems and its associated query language Cypher.

2.4. Neo4j and Cypher Language

2.4.1. Neo4j

Neo4j stands out as a leading open-source property graph database management system, backed by commercial support (https://neo4j.com/). It upholds the ACID (Atomicity, Consistency, Isolation, Durability) principles and adopts a property graph model for data storage. Functioning as a NoSQL graph database system, it leverages graph structures to execute semantic queries, incorporating nodes, edges, and properties. The graph framework comprises interconnected nodes and relationships, both equipped

with named values known as properties. Relationships forge connections between nodes, facilitating the retrieval of associated data, with these relationships being necessarily either incoming or outgoing, enabling exploration of node connections. Properties are depicted as key-value pairs, with the key being a string, and values encompassing primitive types or arrays of a single primitive type.

Neo4j plays a pivotal role in scenarios necessitating interactions among data relationships, as it stores and provides immediate access to relationships, yielding precise and direct responses. Some of the most common and prominent use cases of Neo4j include:

- Fraud Detection and Analytics [15]
- Recommendation Engine and Product Recommendation System [16]
- Network and Database Infrastructure Monitoring for IT Operations [17]
- Social Media and Social Network Graphs [18]
- Knowledge Graphs [19]
- Life Science [20]

Finally, it is worth mentioning that Neo4j demonstrates adaptability and scalability, accommodating the addition of new nodes and relationships to accommodate evolving system requirements. Furthermore, Neo4j boasts a comprehensive array of graph algorithms, spanning functionalities such as pathfinding and similarity analysis. These algorithms serve the purpose of effectively analyzing and extracting valuable insights from graph data, facilitating in-depth exploration and comprehension of complex interconnected relationships within the dataset.

2.4.2. Cypher

Neo4j integrates Cypher, a user-friendly path-querying language for handling intricate queries involving nodes and their associated relationships. A path signifies one or more nodes linked by relationships, typically acquired through query or traversal operations. Being a path query language tailored for graph databases, Cypher boasts numerous functionalities that streamline the manipulation of graph patterns [21]. By utilizing pattern matching, users can articulate graph patterns, specifying the relationships and connections of interest. Furthermore, Cypher incorporates labels, allowing for the classification of nodes and relationships based on distinct characteristics or attributes.

Beyond pattern matching and labeling, Cypher encompasses a diverse array of operations for manipulating and analyzing graph data. Users can employ filtering, sorting, and aggregations to refine their queries and extract desired insights. Additionally, Cypher supports graph algorithms, facilitating the execution of specialized computations on graph structures.

To optimize query processing efficiency, Cypher leverages indexing, caching, and various performance enhancements. These techniques enhance the scalability and effectiveness of queries, facilitating swift data retrieval and processing. In essence, Cypher provides a comprehensive suite of tools and optimizations conducive to scalable and efficient operations within graph databases.

Interacting with a graph database necessitates the utilization of the following Cypher commands for querying purposes:

- **MATCH**: Retrieves nodes, relationships, or patterns from the graph based on specific criteria.
- **WHERE**: Filters query results based on specific conditions.
- **RETURN**: Specifies what should be included in the query result.
- **ORDER BY**: Sorts the query result.
- **SKIP/LIMIT**: Paginates the query result.

When it comes to modifying the graph, the following commands are involved in the process:

- **CREATE**: Creates nodes, relationships, or properties within the graph.
- **MERGE**: Creates nodes uniquely.
- **CREATE UNIQUE**: Creates relationships uniquely.
- **DELETE**: Removes nodes and relationships.
- **SET**: Updates properties and labels.
- **REMOVE**: Removes properties and labels.
- **FOREACH**: Updates data within a collection, whether it comprises components of a path or the result of an aggregation.

In light of the above we should clarify that in Neo4j the path does not work as a specific Entity that can be stored but rather than representation of the structure that is constructed by nodes and relationships as a result of a query, and it can be obtained by

assigning a pattern to an identifier. Its internal structure is an array of nodes and relationships in the way the directions of the relationships are specified within the Cypher statement. It is guaranteed that every single relationship will be traversed one time at most per direction for each “MATCH” pattern that is included within the Query, following the trail path model type.

3. Building an ESN: Key technologies

In this section, the author is describing, step-by-step, every action that was taken to install all the technologies, whose presence was necessary, in order to complete the implementation of this thesis' goal, meaning the creation of a social network application using the Neo4j as its database. Starting from the account registration for the Neo4j database and its instance creation, following the back end's (Node.js) installation and their communication's initialization, going all the way to the Flutter's installation, the framework used to build the UI (User Interface) of this application. Finally, each of the step will also be accompanied with the corresponding screenshot images of the installation process.

3.1. Neo4j Aura account

First, we need to complete the registration's steps in the Neo4j auraDB site, so we can create a database instance and start it off, see Figure 2, with which (instance) our back end (server) will communicate. After we complete every step needed, we obtain the connection URI, database name, that we chose and the password we set, tools that are crucial for the interconnection of the two technologies (database and server). Finally, it is crucial to install one essential plugin: the APOC Library. This plugin offers a comprehensive set of procedures and functions that significantly expand the capabilities of the Cypher query language.

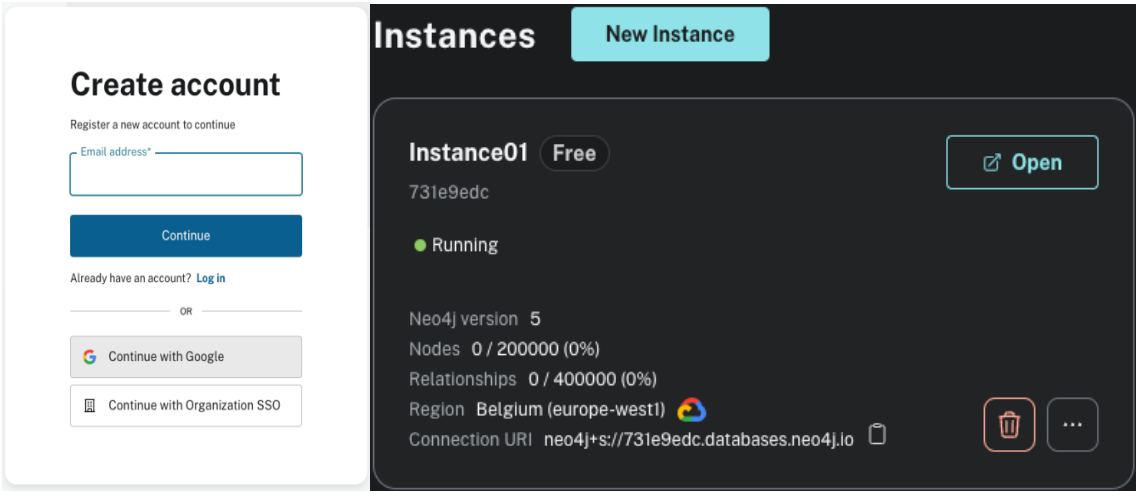


Figure 2:Neo4j registration and new Instance creation.

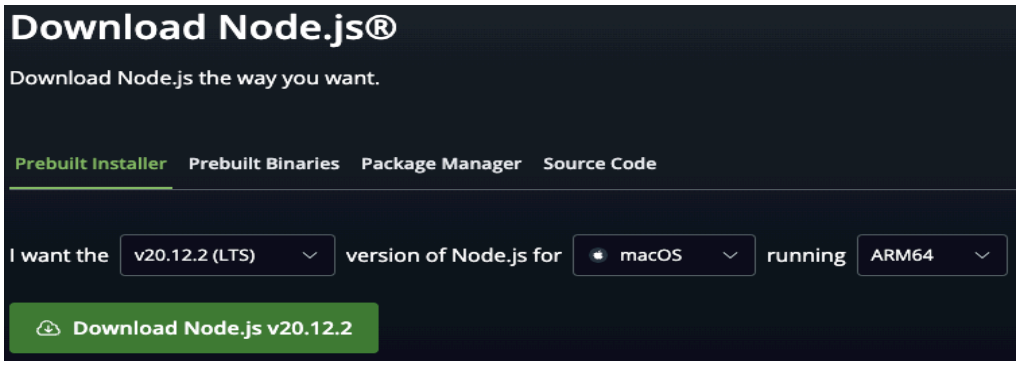
3.2. Node.js installation

Node.js is an open-source, cross-platform JavaScript runtime environment that allows developers to execute JavaScript code outside of a web browser. It is built on the V8 JavaScript engine, which is the same engine that powers the Google Chrome browser. Node.js enables developers to run JavaScript on the server-side, making it possible to create scalable and high-performance network applications. Some key characteristics of Node.js are as follows:

- **Event-Driven -** Node.js operates on an event-driven architecture, which means that it uses asynchronous, non-blocking I/O operations. This allows Node.js to handle a large amount of concurrent connections efficiently, making it well-suited for building real-time applications like chat applications, online gaming platforms, and streaming services.
- **Single-Threaded, Non-Blocking -** Node.js follows a single-threaded, event-loop model, where a single thread handles all incoming requests asynchronously. This architecture enables Node.js to handle many concurrent connections without the overhead of creating new threads for each request, making it lightweight and efficient.
- **Package Ecosystem (npm) -** Node.js has a vast ecosystem of open-source libraries and modules available through the Node Package Manager (npm). npm provides a convenient way for developers to share, reuse, and manage dependencies for their Node.js projects.
- **Cross-Platform -** Node.js is designed to be cross-platform, meaning it can run on various operating systems such as Windows, macOS, and Linux, making it versatile for developing applications across different environments.
- **Scalability -** Due to its asynchronous, non-blocking nature and event-driven architecture, Node.js is highly scalable and well-suited for building applications that require handling many concurrent connections, such as web servers and microservices architectures.

To spin off a server (localhost) and connect it with a database (Neo4j), the first step is to install the Node.js (JavaScript runtime environment) into your machine. Anyone can do so from the official Node.js website. As soon as the installation is completed, it is

crucial for two packages to be installed through the npm (Node package manager), express and neo4j-driver. There are also some other packages, that can contribute into making the server's development process faster, simpler and more efficient. These packages help upon authentication, images, formatting, sockets, encryption, etc. Finally, it is essential to point out that this server runs locally on someone's machine, so he can start building the back end, see **Error! Reference source not found.**. Finally, when the JavaScript code is in production ready form and after taking into consideration all the important security issues, then someone could upload it in a host provider. To start the server locally you must run the following command in the terminal after you have been directed to the project's folder, node "filename.js".




```
alexandrosmentoudakis@ALEXANDROSs-MacBook-Air ~ % node --version
v20.11.1
```

```
const appUtils = require("./app");

const server = appUtils.app.listen(8080, () => {
  console.log("Server is up and running, listening on 8080 port!\n");
});

const io = require("./socket").init(server);

io.on("connection", (socket) => {
  console.log("Client connected with id ", socket.id);

  socket.on("disconnect", (reason) => {
    console.log("Client with id ", socket.id, "has disconnected!");
  });
});
```

Figure 3: Download Node.js and start a server locally.

Using the following code, see **Error! Reference source not found.**, the author manages to connect the database with the back end. After that, the exported driver can be imported in any JavaScript file, which needs to run a single transaction with Neo4j database. In addition, it is worth mentioning, that every transaction action with that

database, which has a starting point and an ending point, follows the ACID principles, meaning that the database remains in a valid form after the transaction ends, even in the event of any unexpected errors that might occur during the process.

```
const neo4j = require("neo4j-driver");

const driver = neo4j.driver(
  "neo4j+s://63ec5690.databases.neo4j.io", // database connection URI
  neo4j.auth.basic("database name", "database password"),
  { disableLosslessIntegers: true } // added so we return the integer values normally
);

module.exports = driver;
```

Figure 4: Establish communication with database.

3.3. Flutter (front end, UI)

Flutter[1] is an open-source UI (user interface) developments kit (SDK) created by Google. It is used for building natively compiled applications for mobile, web and desktop from a single codebase, meaning that you only need to write the code and its (Flutter's) behind the scene mechanisms are responsible for providing the platform-specific output. By doing so, Flutter is enabling developers to write the code once and deploy it across multiple platforms, thus allowing for efficient and rapid development of cross-platform applications. Flutter key features and characteristics include the following:

- **Single Codebase -** Flutter allows developers to write their code once in the Dart programming language and deploy it across multiple platforms, including iOS, Android, web and desktop, reducing development time and effort.
- **Fast development -** Flutter provides a hot reload feature, which enables developers to instantly see changes made to the code reflected in the application's UI. This results in faster iteration circles and more efficient development workflows.
- **Rich UI Components -** Flutter offers a comprehensive set of customizable UI widgets and components, allowing developers to create beautiful and engaging user interfaces with ease. These widgets are designed to look and feel native on each platform that has a vast ecosystem of open-source libraries and modules.

[1] https://flutter.dev/?gad_source=1&gclid=CjwKCAjw-O6zBhASEiwAOHeGxe3CMAZ93JQ_u6XIjM51RNmbU-RY8EVOCXRoujAbz89MUa2YmuZJ4xoCu2gQAvD_BwE&gclsrc=aw.ds

- **High Performance -** Flutter apps are compiled directly to native machine code, rather than being interpreted by a bridge, resulting in high performance and smooth animations. This ensures a consistent and responsive user experience across different platforms.
- **Access to Native Features -** Flutter provides access to native device features and APIs through platform channels, allowing developers to integrate platform-specific functionality into their apps when needed.
- **Community and Ecosystem -** Flutter has a vibrant and growing community of developers, along with a rich ecosystem of packages and plugins available through the Flutter's pub.dev repository. These packages offer additional functionality and extend the capabilities of Flutter apps.
- **Support for Material Design and Cupertino -** Flutter includes support for both Material Design, Google's design language for Android apps, and Cupertino, Apple's design language for iOS apps, ensuring that Flutter apps can adhere to platform-specific design guidelines.

To install the Flutter framework, the user must visit the official Flutter website. Upon executing all the installation steps, Flutter and all the included tools are ready for use, see Figure 5. He then runs the following command in terminal, flutter create "application-name", and everything is completed to start building cross-platform applications for mobile, web or desktop.

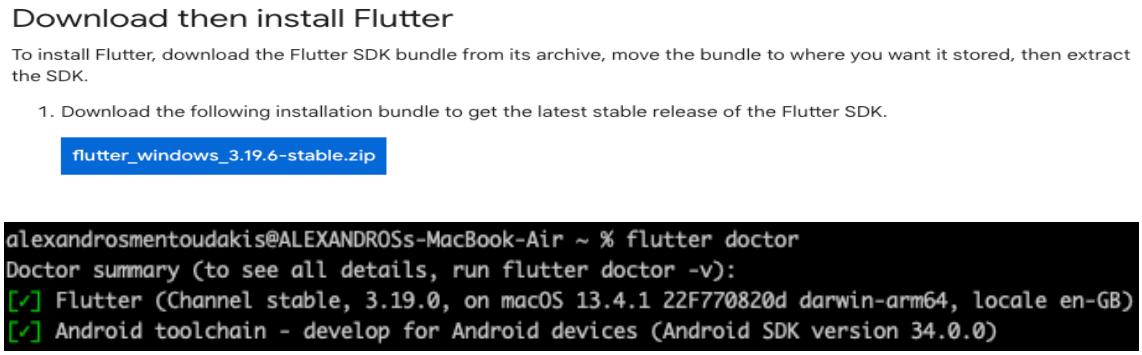


Figure 5: Flutter installation.

4. The ESN Graph Data Model and App design

This chapter presents the structure of the property graph data model for our social network application. This will serve as reference for subsequent elaborations in chapters to follow. The graph data model, as depicted below (Figure 6), displays all the nodes and relationships, that can be utilized inside this social network, without taking into consideration the properties of each element, since there will be analytical description of each one in the next section of the thesis.

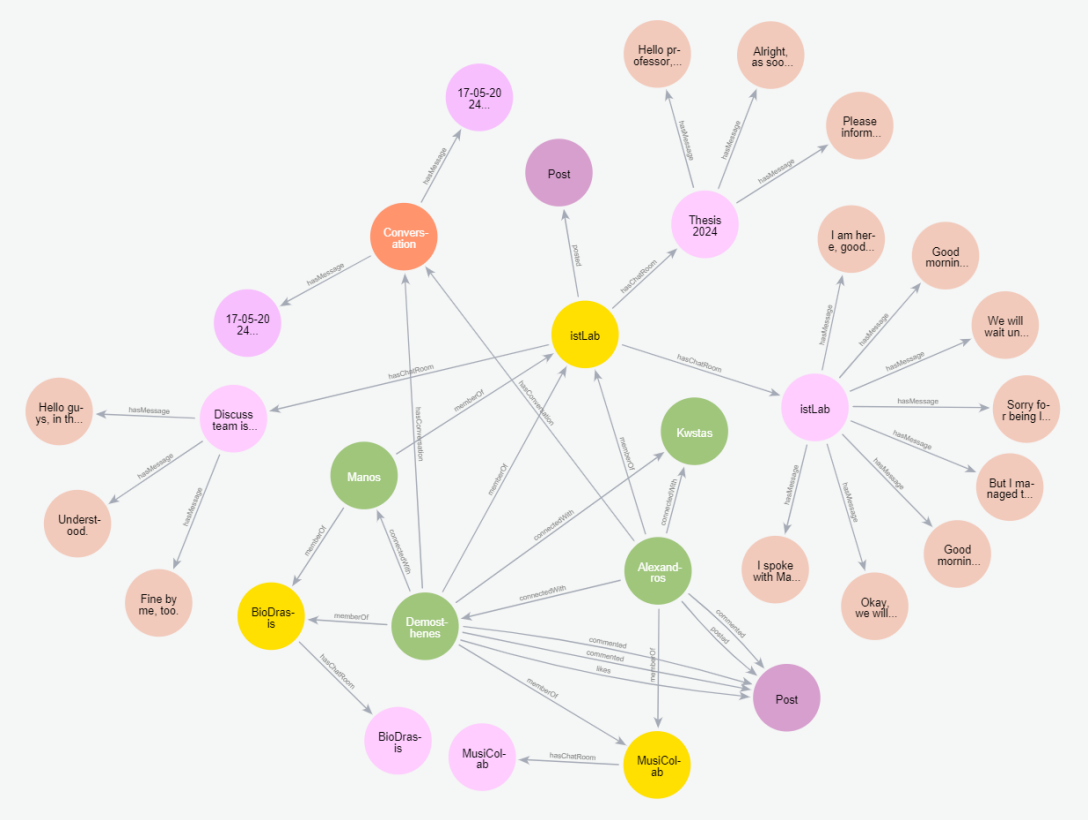


Figure 6: Structure of graph data model.

Leveraging the power of the Neo4j's graph database and the Arrows.app, a graph data modeling tool developed by the Neo4j Labs, the author designed a comprehensive and detailed model that captures every single node, representation of an entity, which can be created by this application's UI (Flutter's mobile application), with their corresponding relationships, that indicate associations between those nodes. Before, we delve into the details of each component's representation and usage (excluding their extra properties for now), it is crucial to point out that every occurred user's reference, refers to those who have been authenticated through the application's corresponding mechanism:

4.1. Nodes and relationships

For the purposes of our application nodes in the property graph are distinguished as follows.

- **User Node -** This node represents a person that has successfully completed the sign-up procedure of the front end application. It (node) holds all the information that were requested from that person. This is the main node of the database graph model.
- **Conversation Node -** This node represents a conversation's channel that has been initialized in the front end when one of two connected users sent the first message to the other one. This conversation and its content can only be seen or accessed from those two users.
- **Message Node -** This node represents a single message that has been sent through some conversation's channel from one user to another. It can be created and therefore displayed only via that corresponding channel.
- **Post Node -** This node represents the content that any user shares with others inside this application. Each post's visibility and comment ability can be customized by that user.
- **Team Node -** This node represents a team that has been successfully created by a user, in order to provide some digital space used to group specific users, with similar interests or goals, thus enabling them to foster collaboration, communication and community-building.
- **Team Chat Room Node -** This node represents a digital room of the social network's application that belongs to a specific team. Its usage is based on the creation of a channel through which all the associated members of that team can communicate with each other, either all together (public rooms) or not (private rooms).
- **Team Message Node -** This node represents a single message that is sent inside a specific team chat room through the communication mechanism of the application. This message is only visible in that chat room and from any member that has access to that.
- **Team Post Node -** This node represents the content that is it shared to a team from a single member and its visibility is limited to that team's members. They, alongside

its creator, are the only ones who can react, comment or send it through the team messaging mechanism.

As per relationships these are as follows:

- **REQUESTCONNECTION Relationship -** This relationship connects two user nodes and indicates that the user of the staring node has sent a connection request to the user of the ending node. Both those users are notified from the corresponding mechanism of the application.
- **CONNECTED Relationship -** This relationship connects two user nodes and indicates that they (users) have obtained the ability to exchange messages, team invitations and content between each other inside the application.
- **MEMBEROF Relationship -** This relationship connects one user node with a team one and indicates that this user is a member of that team. He (user) can interact with every single piece of information that is related with this team, posts, team chat rooms and members.
- **HASINVITED Relationship -** This relationship connects one team node with a user one, indicating that some other user, who is not only a member of this team but also connected with the invited user, has sent an invitation to him asking whether he wants to join that team or not.
- **POSTED Relationship -** This relationship can be used in two occasions. One the one hand, it connects a user node with a post one, to indicate that this user has uploaded some content into the application. On the other hand, it connects a team node with one team post node, to indicate that some content has been uploaded to this specific team "channel" by one of its members.
- **LIKES Relationship -** This relationship connects a user node with a post one, to indicate that this user has reacted to some content that either he or someone else has shared. Each user can react one time at most to one post.
- **COMMENTED Relationship -** This relationship connects a user node with a post one, to indicate that this user has create a comment that belongs to this specific content that either he or someone else has shared. Each user can create as many comments as he wants on some post.
- **HASCONVERSATION Relationship -** This relationship connects two user nodes with a conversation node and indicates that they have initialized a channel, through which they can exchange messages.

- **HASMESSAGE Relationship -** This relationship can be used in two occasions. One the one hand, it connects a message node with a conversation one, to indicate that this created message belongs to that specific conversation between two users. On the other hand, it connects a team message node with one team chat room node, to indicate that a message sent inside a team belongs to a specific chat room of that corresponding team.

After this detailed description of all potential nodes and edges within the Neo4j database, it becomes evident that the structure and connections are vital for efficient data management and retrieval. A deep understanding and careful management of these connections (edges) empowers the optimization of performance, maintain data integrity and facilitate seamless data access. Furthermore, mastering the node-edges interactions can lead to well-designed databases, capable of scaling and adapting to diverse data processing.

4.2. Front – Back End Modeling

In this section, we will elaborate on the properties of each entity and how they are represented in the front/back end. Modeling entities in both the front end and back end of an application is essential for ensuring consistency, interoperability, efficiency, scalability, data integrity, security, and maintainability throughout the development lifecycle. It forms the foundation upon which the entire application architecture is built, making it a critical aspect of the development process. These models differ from how they are stored in the database, since Neo4j's querying mechanism make it simpler and faster to retrieve data, making it unnecessary to store them as extra properties of the nodes. Every Enum and capitalized type is custom-made.

- **User Model**
  - userId (string)
  - firstName (string)
  - lastName (string)
  - imageUrl (string)
  - gender (Enum)
  - occupation (string)
  - birthday (date-time)
  - email (email)
  - password (string/hashed)
  - createdAt (date-time)
  - updatedAt (date-time)
  - posts (List<Post>)
  - connections (List<Connection>)
  - sentConnectionRequests (List<ConnectionRequest>)
  - recievedConnectionRequests (List<ConnectionRequest>)
  - teams (List<Team>)

- teamRequests (List<TeamRequest>)
- teamInvitations (List<TeamInvitation>)

- **Team Model**
  - teamId (string)
  - creatorId (string)
  - teamName (string)
  - teamDescription (string)
  - teamRules (string)
  - teamLabels (List< Enum >)
  - teamVisibility (Enum)
  - teamRequestAbility (Enum)
  - createdAt (date-time)
  - teamProfileImageUrl (string)
  - lastMemberJoinedAt (date-time)
  - membersTotalNum (integer)
  - members (List<Member>)
  - teamPosts (List<Post>)
  - userInvitations (List<TeamInvitation>)
  - userRequests (List<TeamRequest>)

- **Post Model**
  - postId (string)
  - creatorId (string)
  - content (string)
  - imageUrl (string)
  - postVisibility (Enum)
  - commentAbility (Enum)
  - createdAt (date-time)
  - updatedAt (date-time)
  - comments (List<Comment>)
  - postLikes (List<PostLike>)

- **Conversation Model**
  - conversationId (string)
  - otherChatUser (User)
  - messages (List<Message>)
  - lastMessage (Message)

- **Connection Model**
  - connectionId (string)
  - connectedUser (User)
  - createdAt (date-time)

- **ConnectionRequest Model**
  - connectionRequestId (string)
  - sender (User)
  - requestedUser (User)
  - createdAt (date-time)
  - connectionRequestStatus (Enum)

- **Comment Model**
  - commentId (string)
  - creator (User)
  - content (string)
  - createdAt (date-time)
  - updatedAt (date-time)

- **PostLike Model**
  - likeId (string)
  - creator (User)
  - createdAt (date-time)
- **Message Model**
  - messageId (string)
  - creatorId (string)
  - content (string)
  - createdAt (date-time)
- **TeamInvitation Model**
  - teamInvitationId (string)
  - team (Team)
  - creator (User)
- **TeamChatRoom Model**
  - teamId (string)
  - creatorId (string)
  - chatRoomType (Enum)
  - messages (List<Message>)
  - allowedMembersIdsList (List<string>)
- **Member Model**
  - user (User)
  - joinedAt (date-time)
  - role (Enum)
- **Feed Model**
  - post (Post)
  - creator (User)
  - commentsTotalNum (integer)
  - postLikesTotalNum (integer)
  - requestedUser (User)
  - createdAt (date-time)
  - teamInvitationStatus (Enum)
- **TeamRequest Model**
  - teamRequestId (string)
  - team (Team)
  - creator (User)
  - createdAt (date-time)
  - teamRequestStatus (Enum)

Every single one of the models, that was described above has the same structure both in the front and back end, thus significantly reducing the possibilities of any

unexpected error's occurrence during those two technologies communication and data exchange.

Additionally, another crucial aspect of these models is their construction during the user's interaction with the application. Whenever a user logs in and is being transferred to the home page of this application, there is no need or any extra value gained for him from having obtained every single one of his potential properties immediately. By carefully managing these step-by-step constructions, not only of the user model but also of any other model that is used inside this social network application, its (application's) response and UI update times are being significantly improved, resulting to much better user experience.

5. The ESN App – Scenarios of use

In this section, we describe how someone can interact with a social network application through the UI (User Interface) of his device. Whenever is necessary, it is also displayed how that interaction transforms the Neo4j graph database model. Every utility that this social network can provide will also be explained and presented with screenshot images from the corresponding UIs and the database that were used.

5.1. Registration and authentication

Users can create a brand-new account for the social network application and sign-in. When the user fulfills of all the necessary data and press the sign-up button, (see Figure 7, left), a request with all those data is sent to the server. There they will be validated and if everything is right, there will be two upcoming events. First, the server communicates with the Neo4j database through the neo4j-driver transaction mechanism and a new node with the User label is created, see Figure 8, having all those data as properties (key-value format). Second, he sends back a response to the client, to update the UI and inform the user that his sign-up is completed successfully and he is now ready to sign-in (see Figure 7, right).

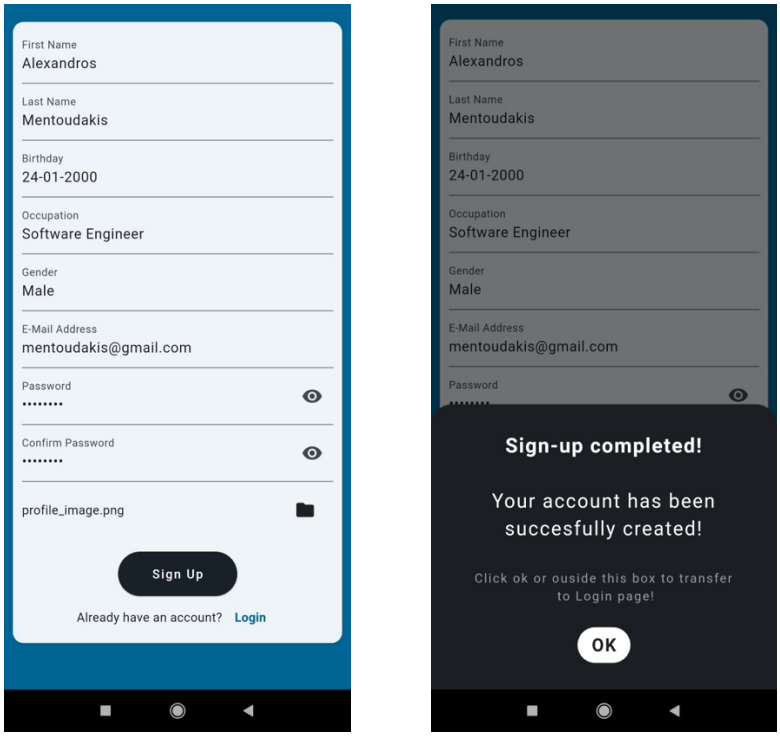


Figure 7: Right: user sign-up, Left: successful sign-up message.

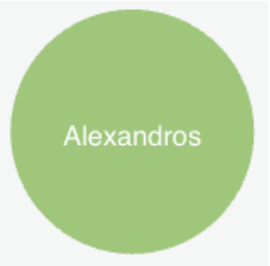


Figure 8: Created User node in Neo4j.

Additionally, we also created two other users, see Figure 9, so we can properly showcase the available interactions within the social network application. The current database snapshot is like this, see Figure 10.

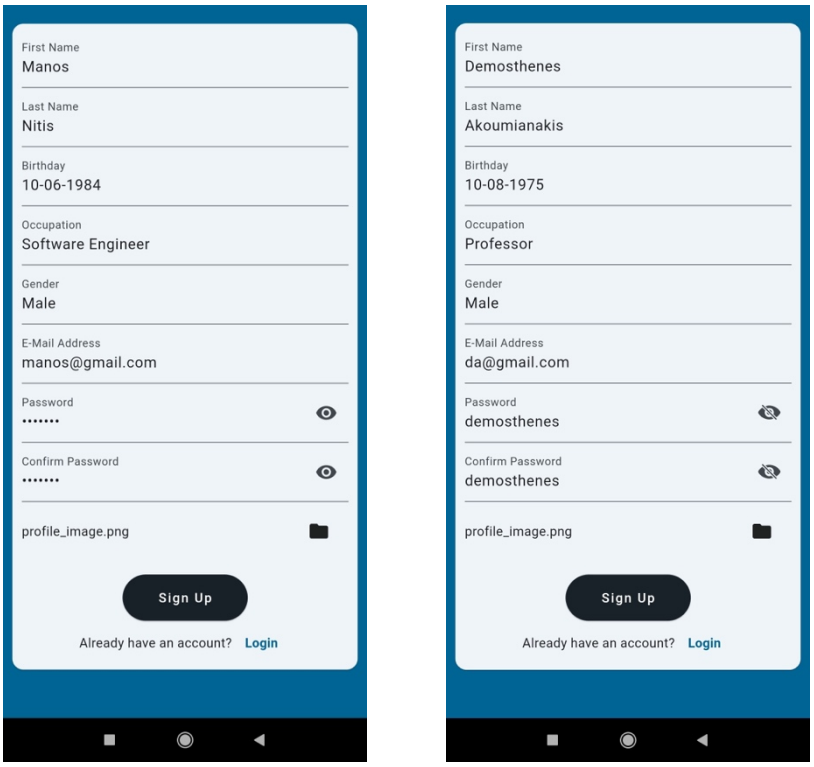


Figure 9: Creating two more user profiles.



Figure 10: Current database snapshot.

Once a user account is created, the authentication routine will allow a registered user to access the app. Upon completing the login form (see, Figure 11 left), an authentication request is sent to the server. There, the submitted data are being checked for their validity and a response is returned to the front end (mobile application). That response has either a status code of 200 if the user is successfully authenticated or anything else from 400 – 500, according to the error event's type. Assuming now that the credentials were valid, thus the authentication was successful the user will be transferred to the application's home screen (see, Figure 11 right).

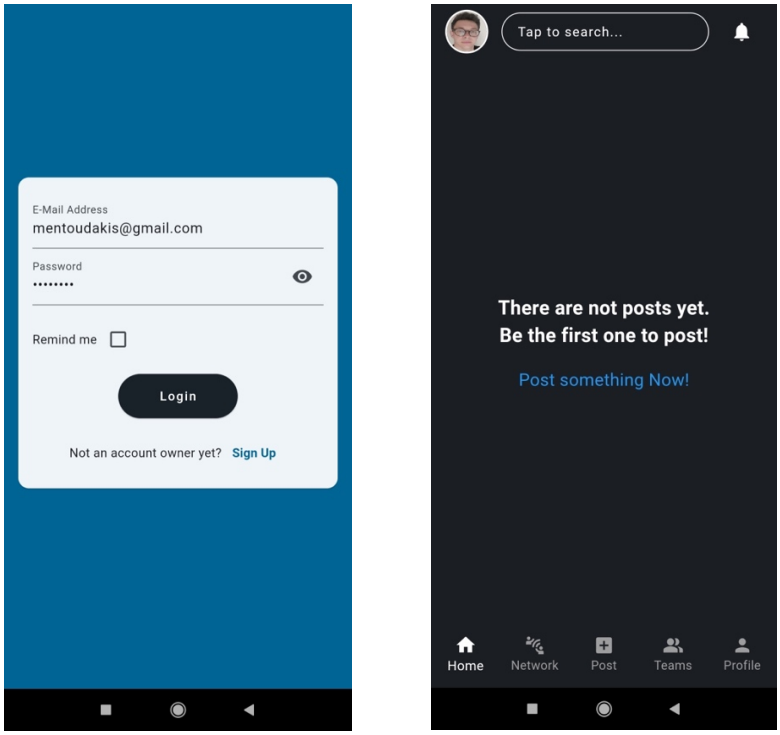


Figure 11: Right: user sign-in, Left: logged in user is transferred to home screen.

Alongside those response data, a token was also included, that was previously created and stored in the server, that is needed for any up-coming request that might occur by the application's UI.

5.2. Home Screen

Following user's transfer to home screen, from where he can have access to the available public content of this application, there are now several paths, which he can follow, see navigation bar at the bottom of Figure 11 right. We will describe all the available scenarios step-by-step down the road. Every separate screen can be accessed by the navigation bar that is located at the bottom of the home screen.

5.3. Search Screen

Upon pressing search bar, the logged in user is transferred to the search screen, where he can find any other existing ones. Whenever he writes a letter, every user whose first or last name matches the written word will be displayed bellow. The user can then visit their profile by clicking on the corresponding card, (see Figure 12).

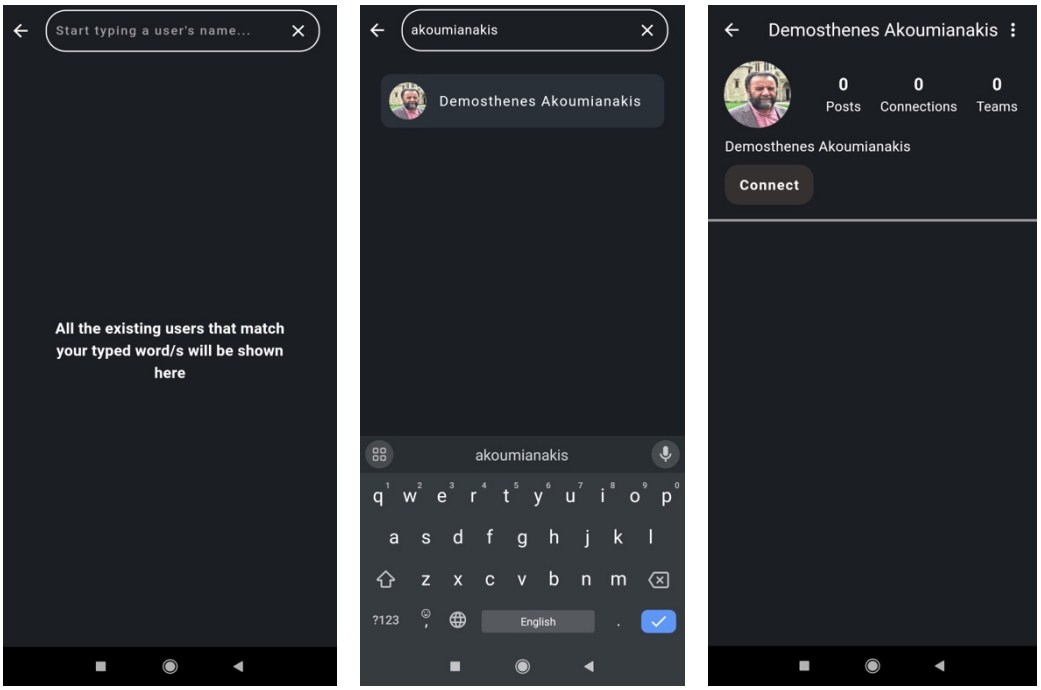


Figure 12: Search and visit user's profile.

5.4. Network Management

Upon pressing navigation bar's network button (second from left), the user is transferred to network screen, (see Figure 13), where he can both view and access all his network's data and also be recommended about new protentional connections and teams. By pressing one of the two arrows, he can either navigate to a screen, where he can see more details about his network, (see Figure 14, left), or to the invitations screen where he can view and interact with every in-coming or out-going invitation (user connection, team join), in which he is involved, (see Figure 14, center, right).

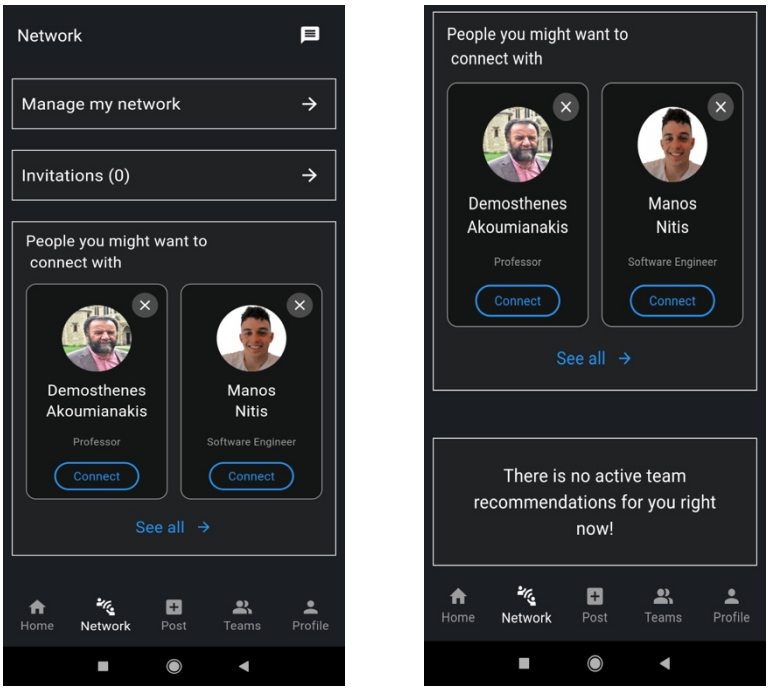


Figure 13: Network management screen.

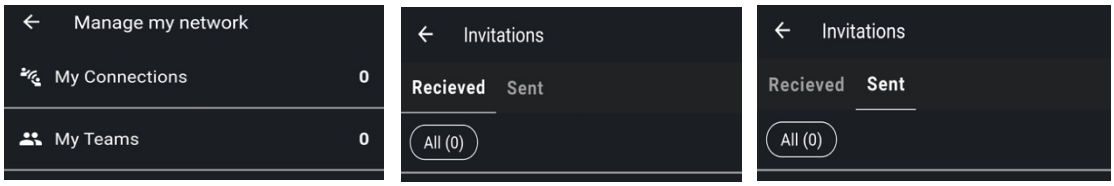


Figure 14: Network details, in-coming and out-going requests.

5.5. Connection Mechanism

From this screen the user has also the capability to connect with another user. By pressing the connect button, (see Figure 15, left), a connection request is sent to the back end, including the sender's and requested users' ids, then the server and database communicate to find the corresponding users and update the database snapshot, accordingly, see Figure 16. Additionally, the server communicates with the front end of both users through socket mechanism to make the essential updates in the UIs, (see Figure 15, center and right). The requested user can also navigate to the sender's profile by pressing his name in the card, (see Figure 17, left), so he can see more details about him, prior to his answer. From there, he can answer this connection request by pressing the "answer" button, (see Figure 17, center and right). Assuming now, that he accepts this request, the server communicates with the database and the front ends to update the snapshot and UIs, accordingly, see Figure 18. They are now treated as connected users, thus unlocking more ways of interacting with each other.

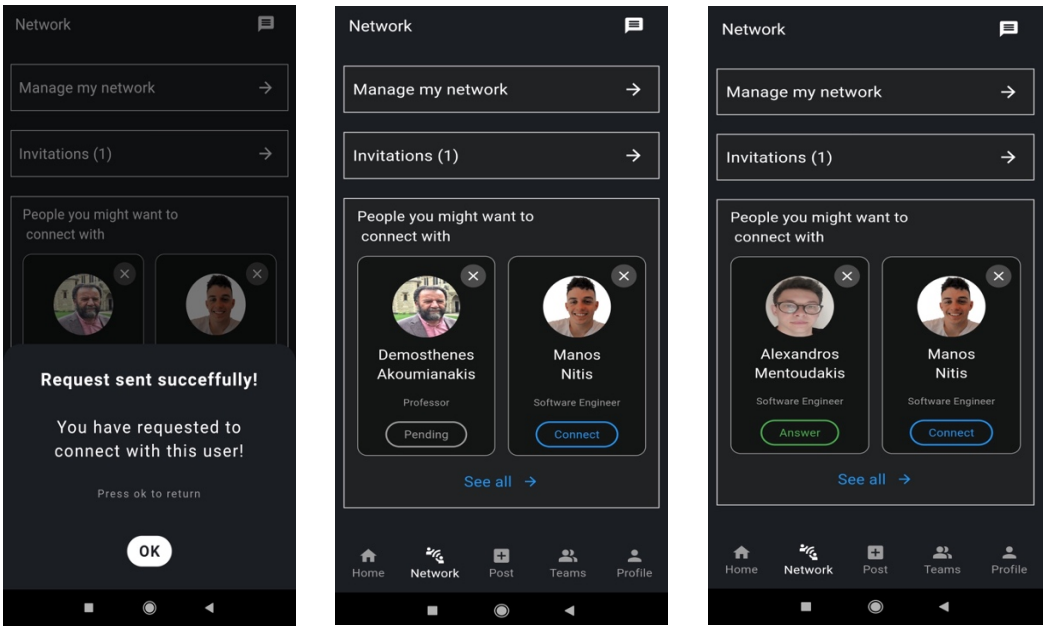


Figure 15:Send connection request, updated UIs.

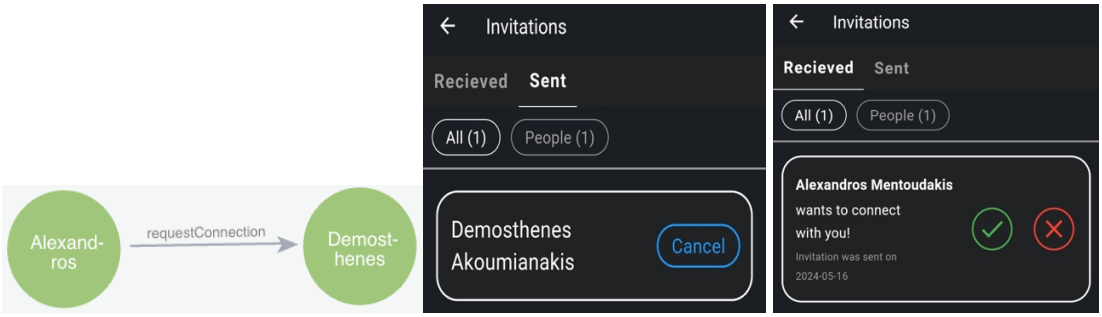


Figure 16: Created relationship between users, sender and requested users' UIs.

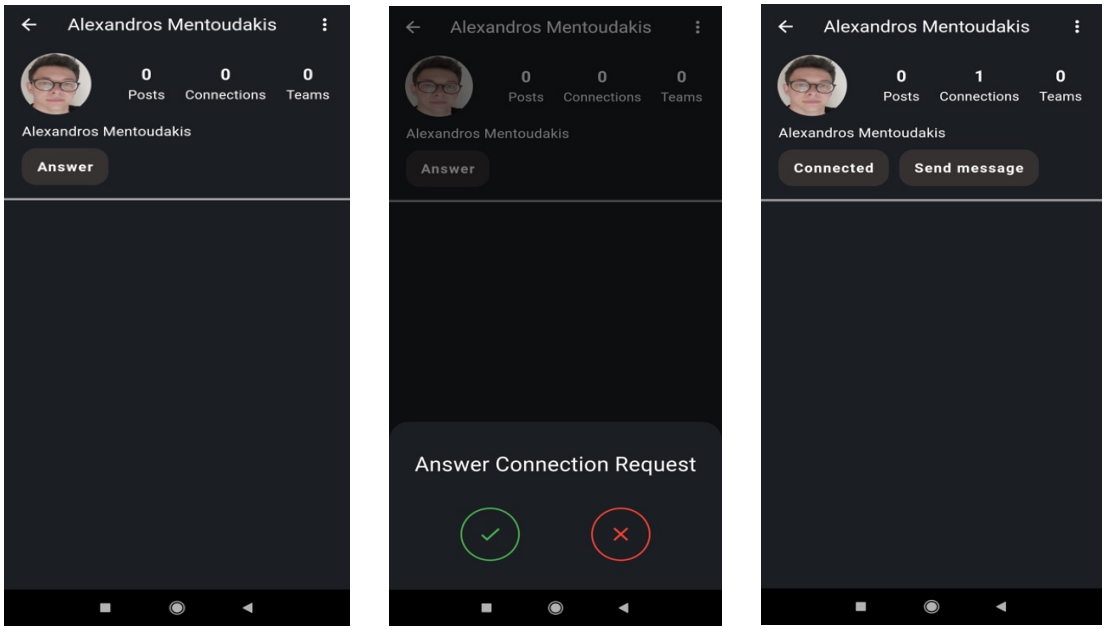


Figure 17:Visit user profile and accept request.

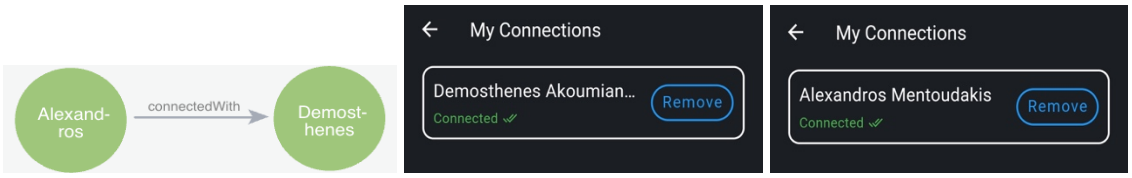


Figure 18: Created relationship between users, updated UIs.

5.6. Messaging Mechanism

Upon pressing the message button of the network screen, the user is navigated to the conversations' screen, (see Figure 19, left and center), where he can join an existing or start a new conversation, with every user that is currently connected with. Every existing conversation will be displayed below. By tapping one of the "bubbles", he will be transferred to a single conversation screen (see Figure 19, right), from where he can exchange messages with that corresponding user. If those users have already exchange messages before they will also be displayed. Each time a brand-new conversation between two users is created, then the database snapshot is updated by creating a new node that connects with both, see Figure 20. Every new message will be stored to that specific conversation node with its creator's id, content and timestamp.

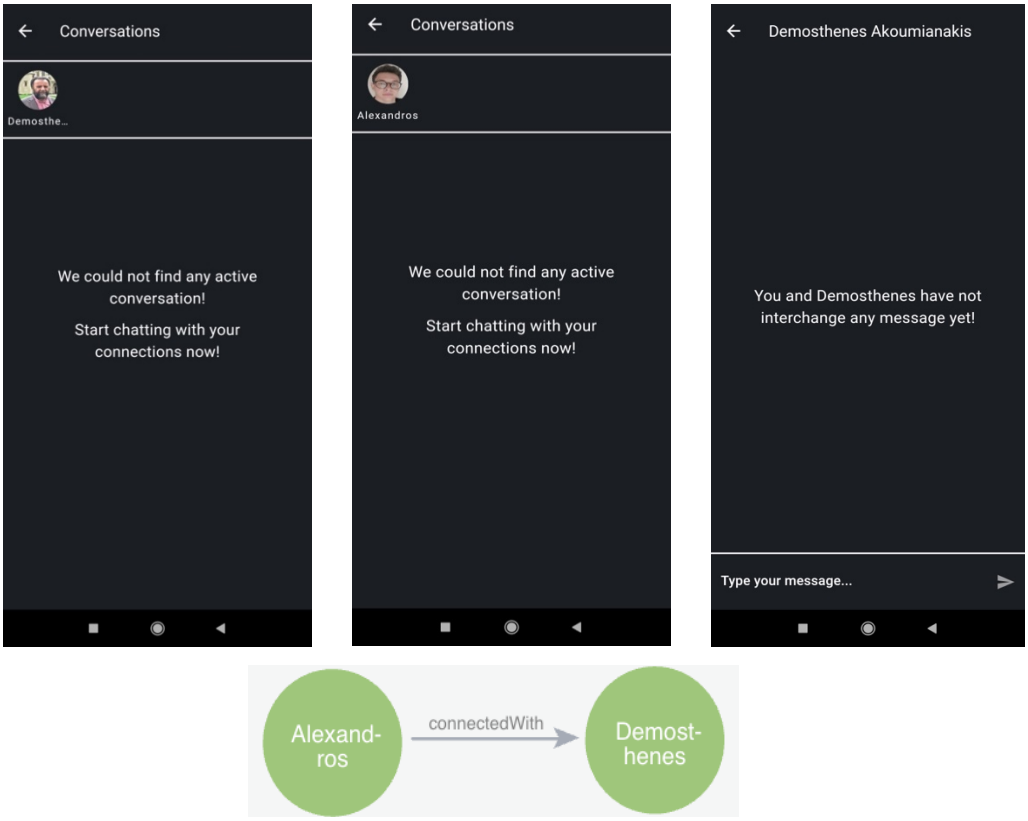


Figure 19:Left-Center: Conversation's screen, right: single conversation screen.

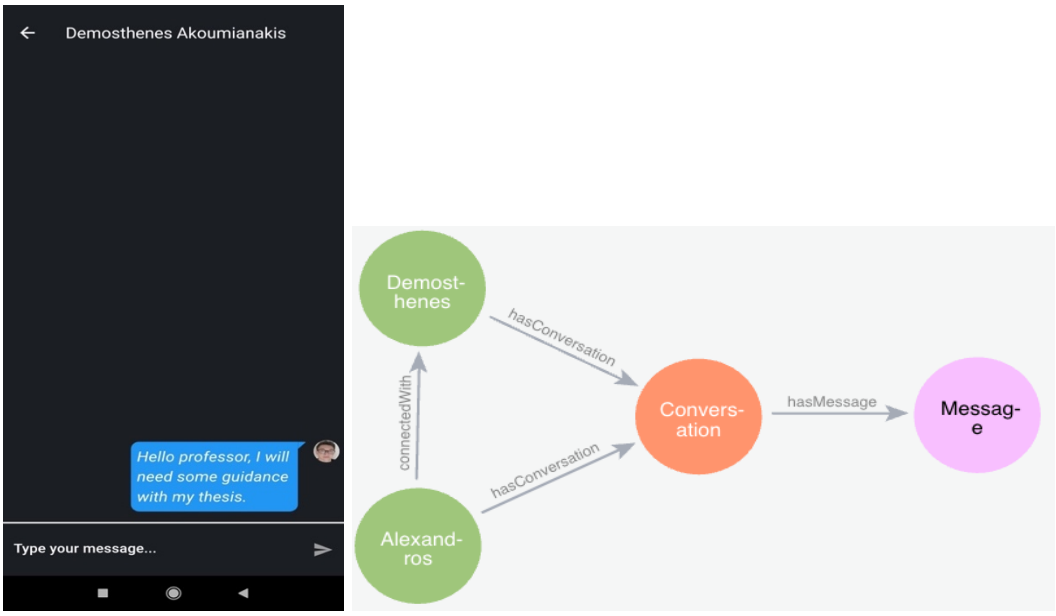


Figure 20: First message between two users.

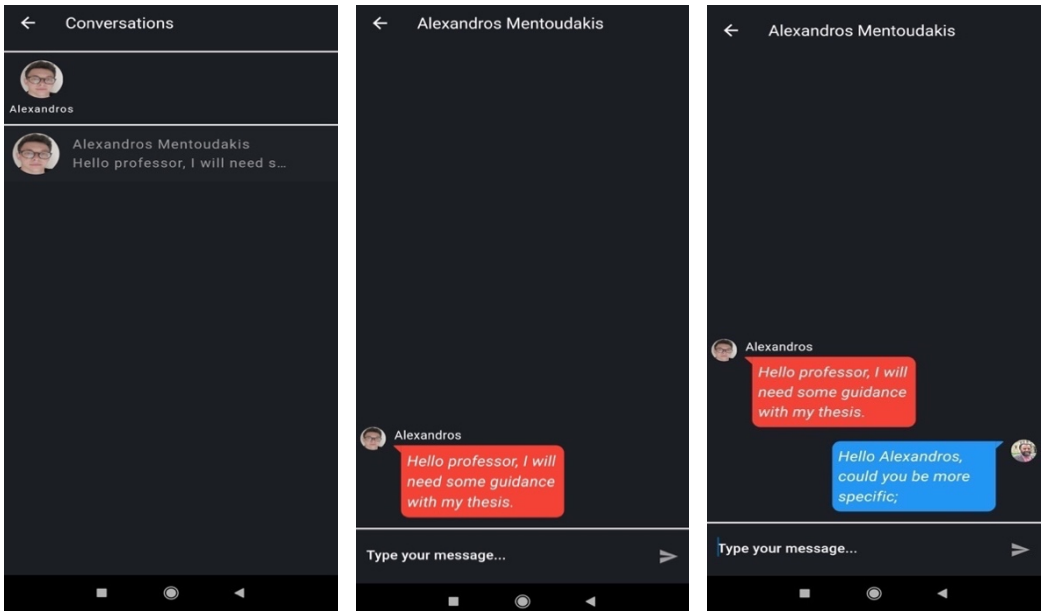


Figure 21: Response message.

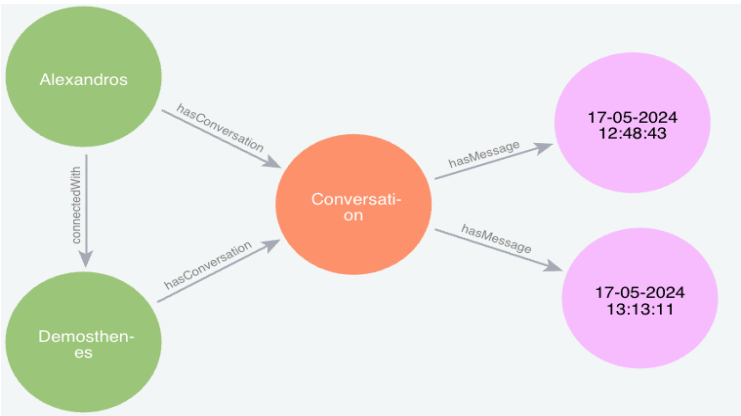


Figure 22: Exchange of messages between two users.

5.7. Posting

5.7.1. Uploading a post

Any authenticated user has the capability of sharing content with others. By pressing the post icon in the bottom navigation bar (third from left), a modal opens immediately, allowing him to interact with the corresponding content-upload mechanism, (see Figure 23, left). Inside this modal, he can select an image, either from gallery or take a new one, write some text, (see Figure 23, center and right), but also choose who can see or interact with his content, once it has been uploaded, (see Figure 24, left and center). When the user fulfills all the necessary details, (see Figure 24, right), she can now press the text-button, "Post", to upload his content in the application. After pressing it, a request is sent to the server, where the data will also be validated and if everything is right, he (server) will then communicate with the database, instructing it to update the current snapshot, accordingly, (see Figure 25, left). Following the database update mechanism, the server will also send back a response to the client to inform the user that everything went as expected and his content is successfully uploaded, (see Figure 25, right). Finally, using the socket mechanism this created content will be sent to all the appropriate users, see Figure 26.

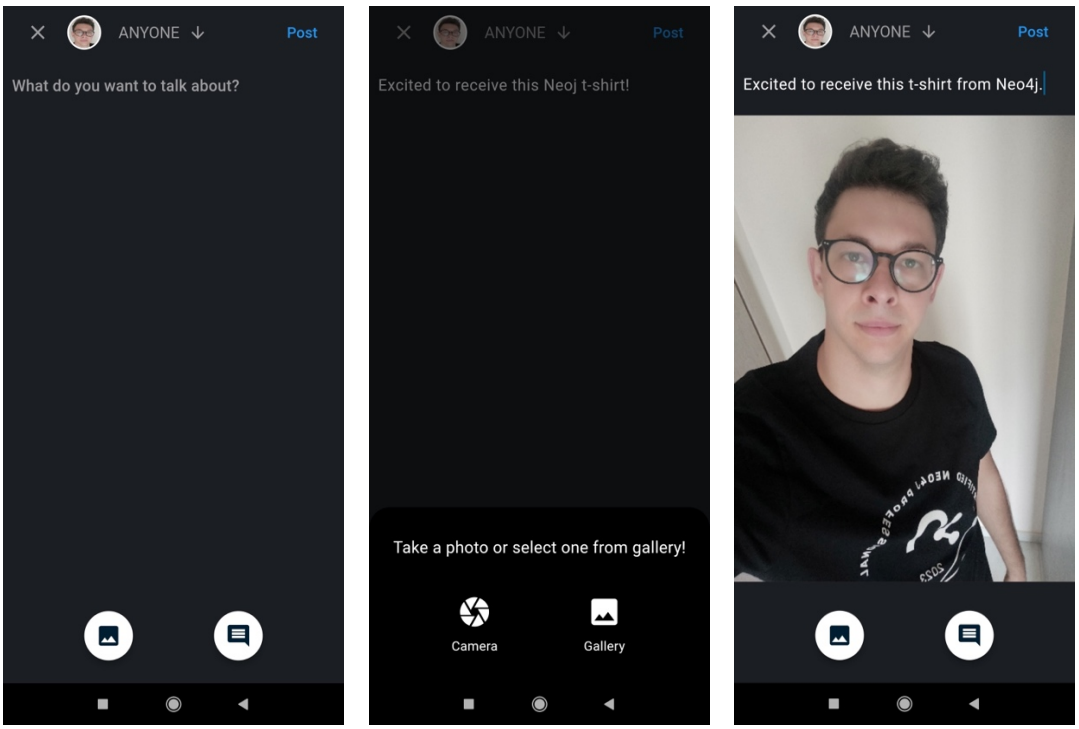


Figure 23: Open Modal, fulfill content's details.

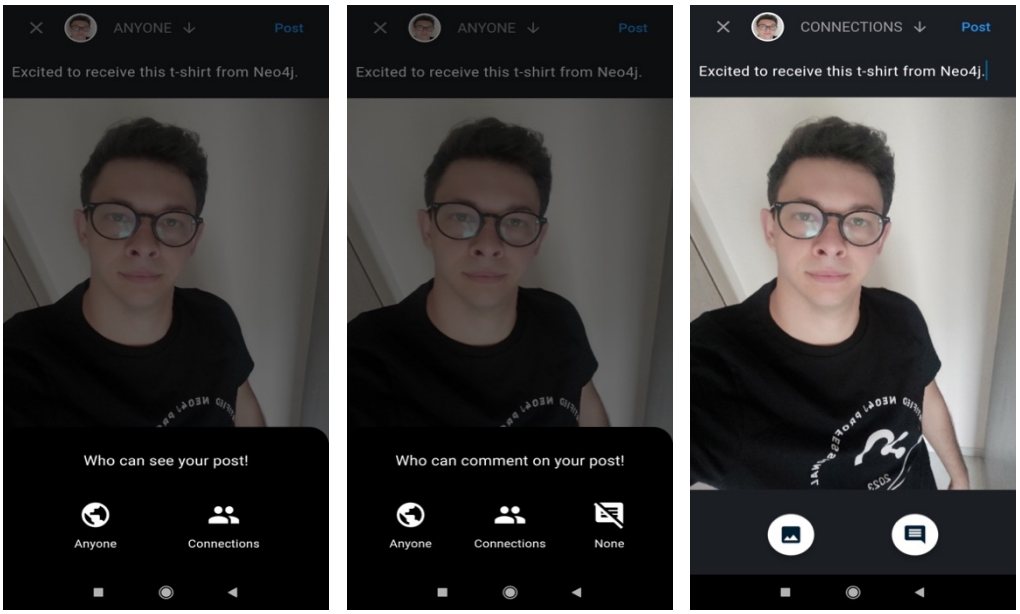


Figure 24: Choose who can interact with the content.

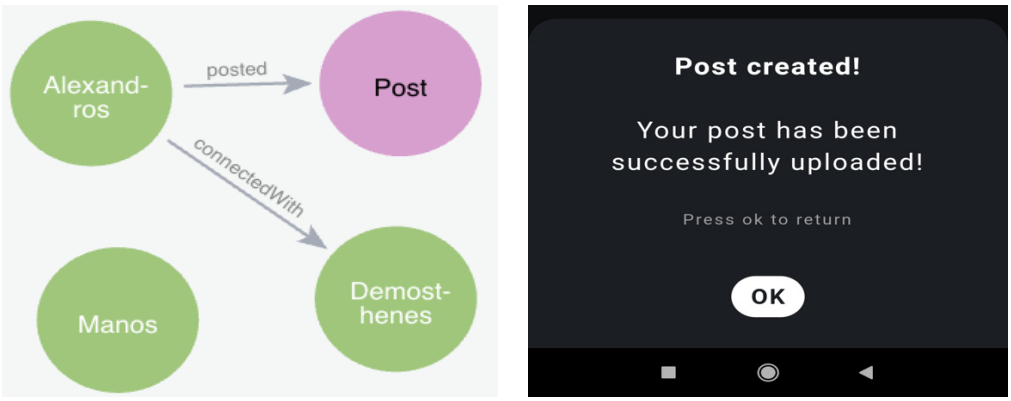


Figure 25: Updated snapshot of involved nodes and response message.

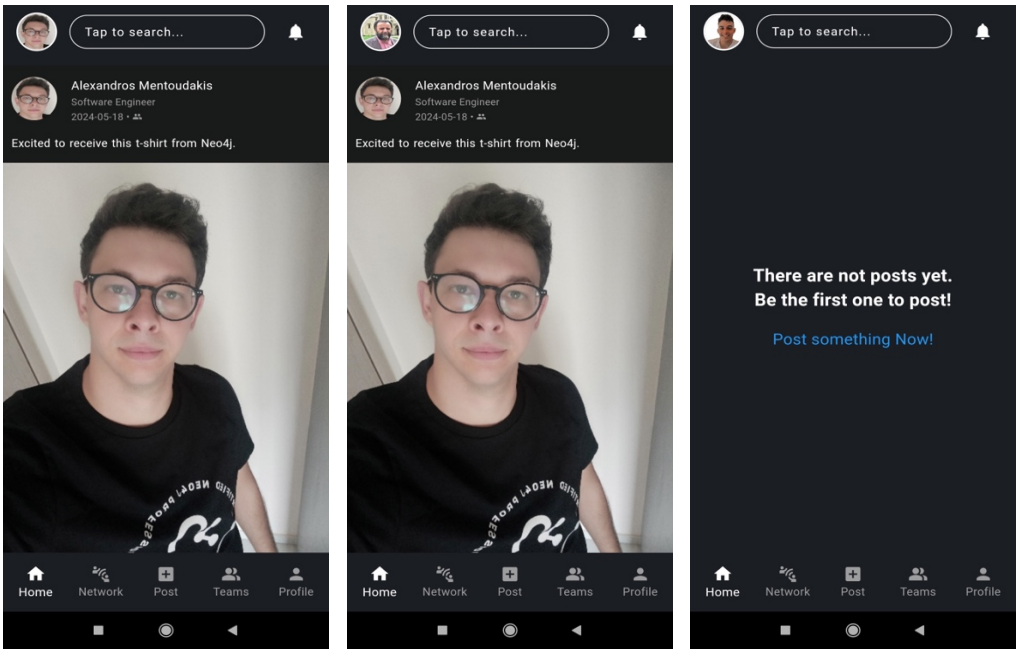


Figure 26: Uploaded content sent to appropriate clients.

5.7.2. Interacting with a Post

Following some post's upload to the application, (see Figure 27, right), users who have been granted access, can interact with it either from the home screen by pressing one of the available icon-buttons, located at the bottom, or from its post screen, where they can transfer to, by clicking on its image, (see Figure 27, left and center). After a user is navigated to some single post's screen, every missing associated with that information is being fetched from the server. Then, they can view all the existing information and utilize either the like mechanism, to show that they like the post and its content or the comment mechanism, to express their opinion about this post, ask related questions or provide some extra insights about its content to the rest of users who have access to it.

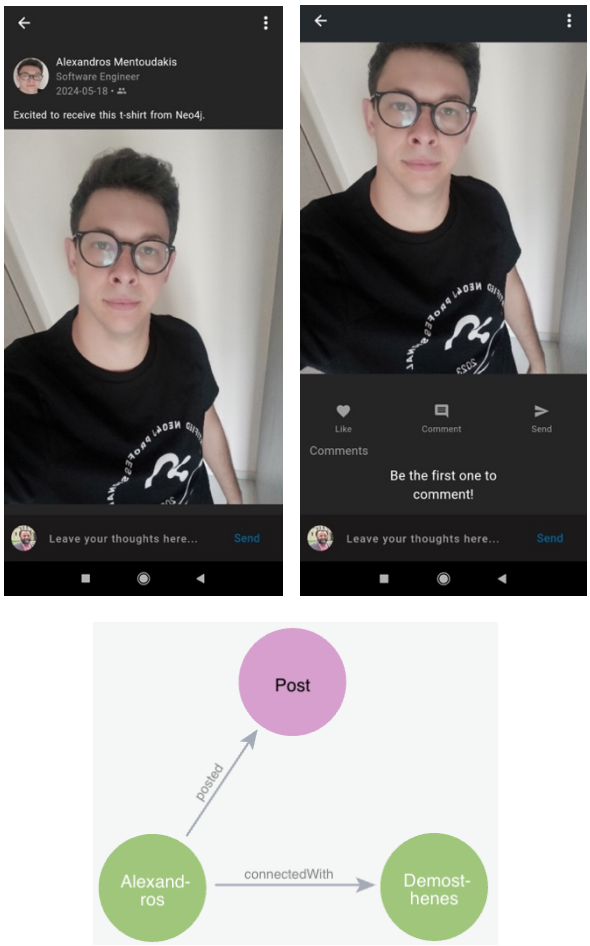
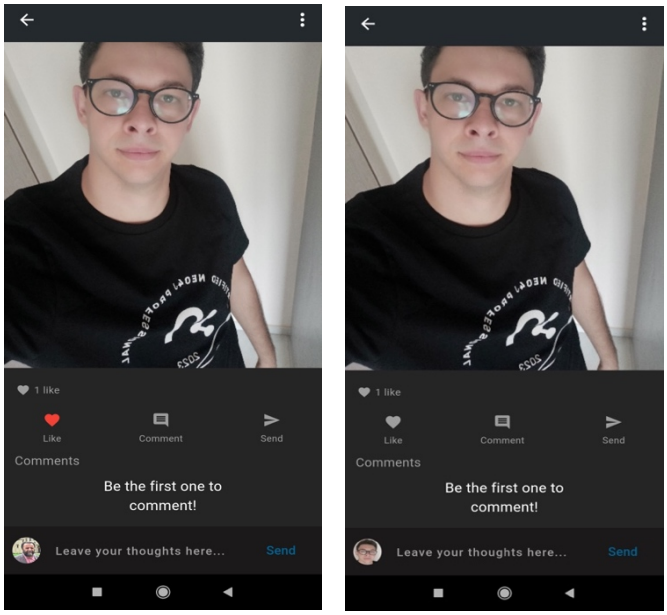


Figure 27: Post screen, current snapshot of the 2 users and the post.

5.7.2.1. Liking a Post

Whenever a user presses the like icon button, a request is being sent to the server, whose work is to validate if the user has the access to procced with this action. Afterwards, he instructs the database to update the current snapshot, accordingly, and

sends back the appropriate response to the client. The user can cancel the previous action by pressing the button again. Every client who is currently accessing this post will be informed about these actions and will update his UI as needed.



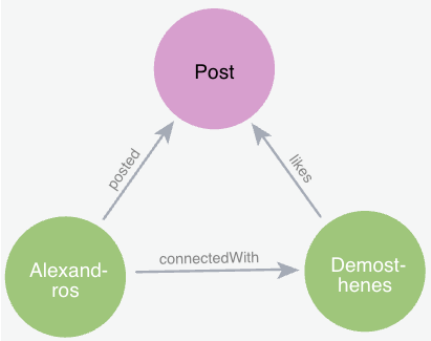


Figure 28: User likes another user's post, updated database snapshot.

5.7.2.2. Commenting a Post

By clicking either the comment button, second from left, or the container that is located at the bottom of the screen, the user can create a comment that will be associated with this specific post. Every authorized user can create as many comments as he wishes, see Figure 32, which will be stored within a relationship that relates that user with the post, by writing some text and pressing the blue "Send" button. Following that button's click, a request is sent to the server, informing him that someone wants to comment a post. First, he communicates with the database, to find the post with its creator and the user. Second, he validates the post's comment-ability status and, if needed, the user's connection type with the creator of the post. Assuming now that everything is valid, he instructs the database to update the current snapshot, accordingly, see Figure 29, and sends back a

response. Finally, every client who is currently accessing this post will be informed about these changes and will update his UI.

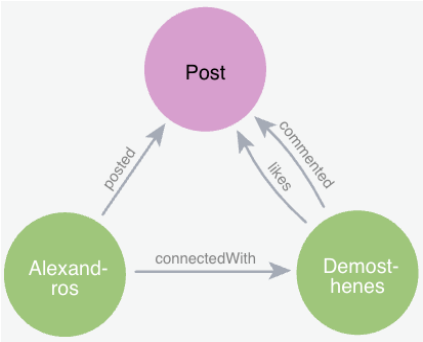


Figure 29: Updated database snapshot after first comment.

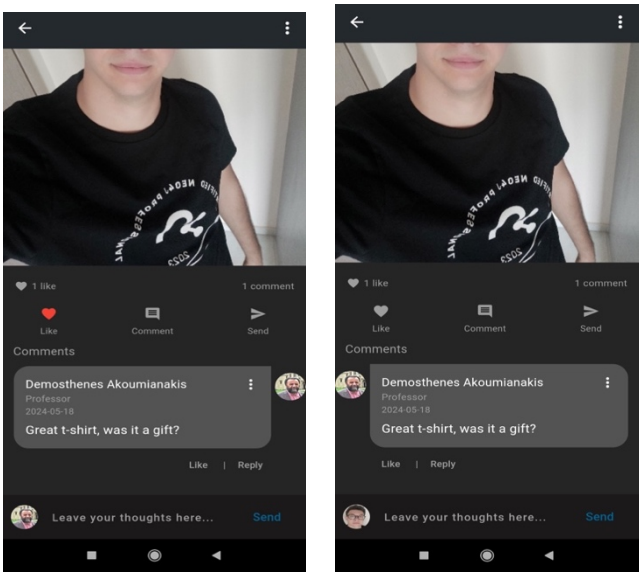


Figure 30: Updated UIs of associated clients after first comment.

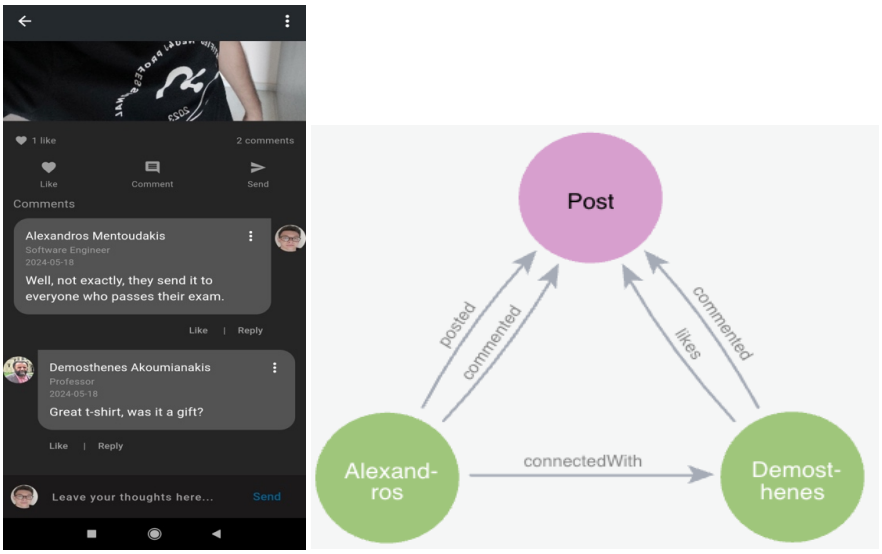


Figure 31: Updated UIs and snapshots after response comment.

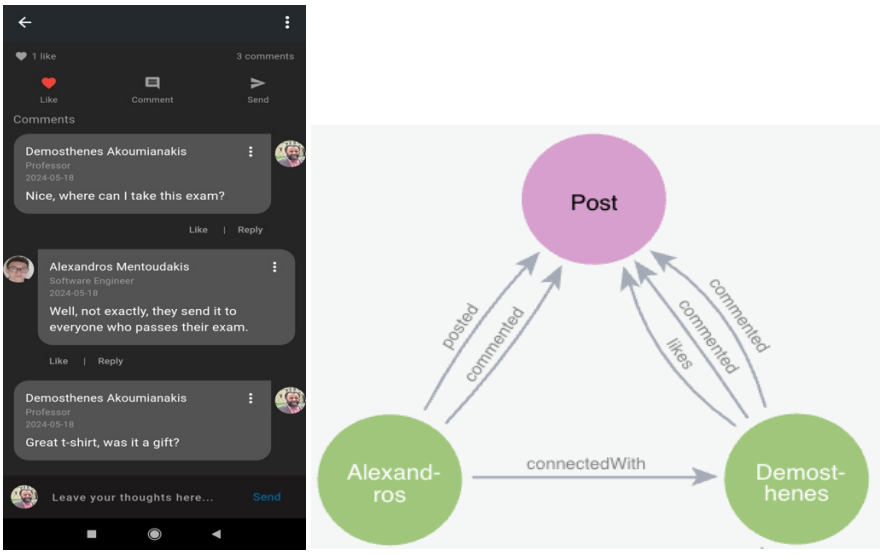


Figure 32: Updated UIs and snapshots after second comment from same user.

5.8. Team Management

Upon pressing the team's icon, second from right, the user is navigated to the team management's section. This section consists of 3 screens, "My Teams Screen", "Invitations Screen" and "Discover Screen", about which we will discuss with more details later. He can navigate from one to another, using the link-items below the app-bar and the first time he accesses this section he is automatically navigated to "My Teams" screen, (see Figure 33,left). In this section, he is provided with the utility of creating or joining teams, user-groups that can only be accessed by their corresponding members and creators. After joining a team, he can upload content that is only visible to anyone within that specific team or create team chat rooms to discuss any issue that might occur down the road. Those chat rooms can either be public or private. Furthermore, if the team's creator (admin) allows it, members of the team can invite their connections to join. The creator of a team is the only one authorized to delete that team, any of its members or posts.

5.8.1. My Teams Screen

In this screen, every team in which the logged in user is either admin or member, will be displayed as card, including some associated information, such as team name and user's team role, (see Figure 33, right). By pressing on the card, the user can navigate to that corresponding team's screen. He can also initialize the procedure of new team's creation by pressing one of the "Create" buttons or navigate to "Discover Screen" by pressing the "Discover" button.

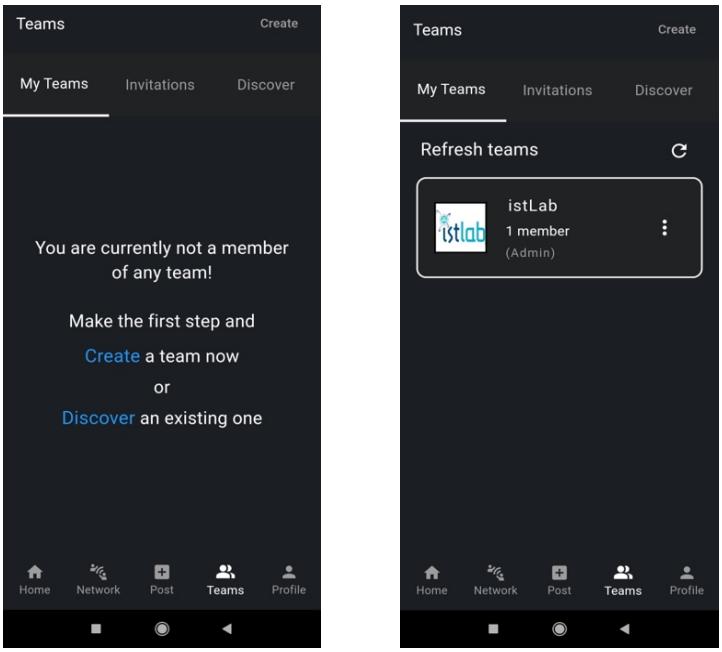


Figure 33: My Teams Screen empty and with a team.

5.8.2. Creating a Team

Upon pressing the text-button "Create", a modal opens and the procedure of creating a new team is initialized, (see Figure 34, left and right). Here, the user must choose an image and fulfill all the necessary information about the team he wants to create, such as name, description and rules. By pressing the "Add Label" button, he is transferred to the "Team Labels Screen", (see Figure 34, center), from where he can choose the appropriate "tags" for his soon-created team. He has also the option of making the team either public or private, meaning that users outside the team can view or not the uploaded content. Additionally, he can allow the members to invite their connections by pressing the checkbox. After he fills out all the needed information and presses the "Create" button, the client sends a request to the server, informing him that someone wants to create a team. He validates all the received data and then he proceeds to the communication with the database. After the current snapshot is successfully updated, thus creating the team and team chat nodes and connecting them to the corresponding creator's user node, (see Figure 35, right), the server sends a response to the client, "telling" him that everything went as supposed. Finally, the client updates the UI, accordingly, and displays a success message, (see Figure 36, left).

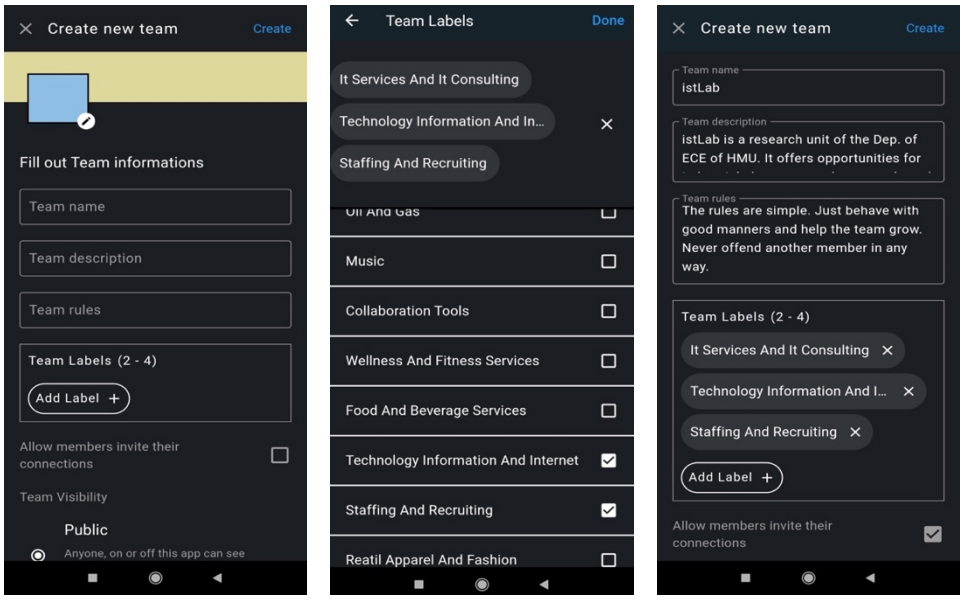


Figure 34: Create new Team modal and Team Labels Screen.

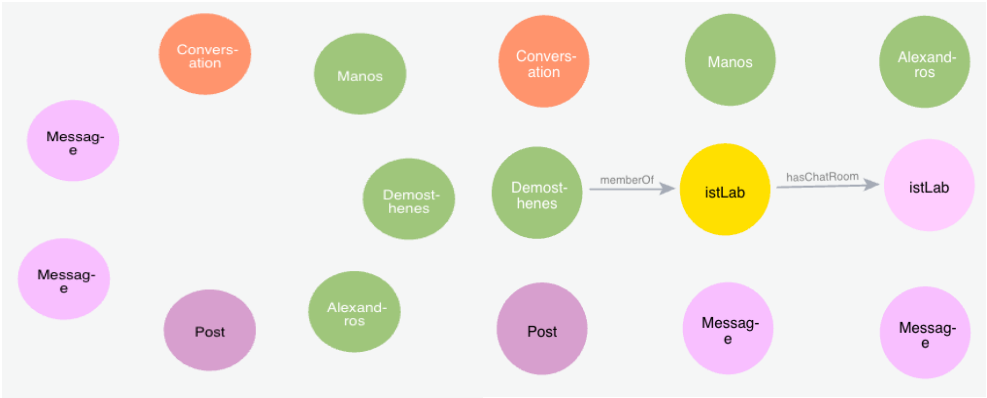


Figure 35: Database snapshot before and after team's creation.

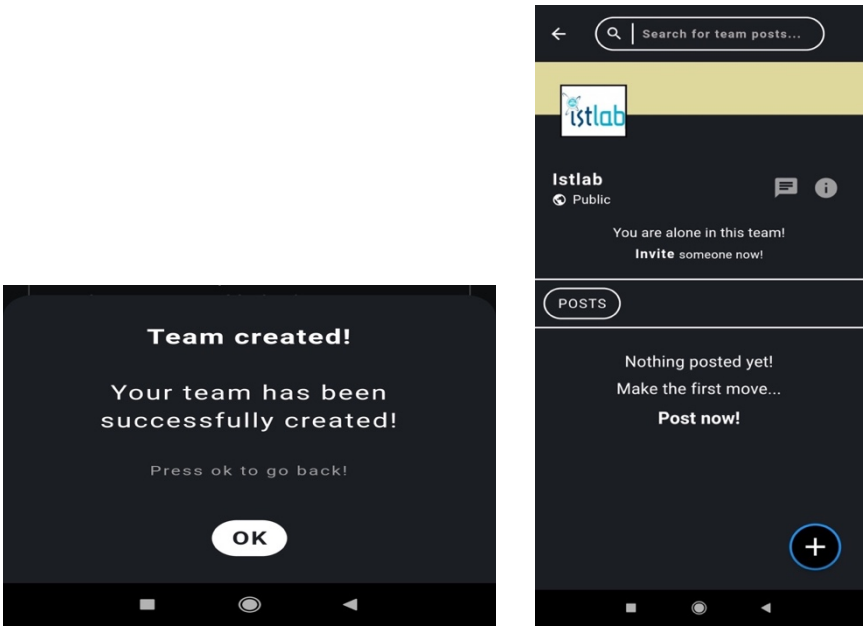


Figure 36: Success message and created team's main screen.

5.8.2.1. Single Team's Screen

Upon pressing the "OK" button of the modal, the user is transferred to his created team's screen, see Figure 36. Here he can view some of the team's information, such as name and type but also all the uploaded posts. He can now proceed with any of the following actions. First, he can press the "plus" button, circle one down-right on the screen, to open the modal with the team actions, ("add new post", "invite new member"), which can also be utilized from the corresponding text-buttons of the screen, "Invite" and "Post now!". Second, he can press the message icon-button, to navigate into the team chat rooms' screen, where all the created chat rooms alongside the default one, are located. Third, he can press the search bar at the top to navigate to the team posts' screen, where he can also view and interact with all the uploaded content. Finally, by pressing the information icon button, he can view the rest of the team's information, such as description, rules and team labels.

5.8.2.2. Inviting New Members

By pressing either the "Invite" text-button or the "plus" icon-button and then choosing the "add member" option, (see Figure 37, center), the user is transferred to a new screen from where he can invite new members to this team, (see Figure 37, right). He can only invite users who are connected to him. By pressing the "Invite" button, a request is sent to the server, including the ids of the team and two users. There, he validates that the creator of the request is authorized to complete this action and if this is the case he then continues with the database communication. The current snapshot is being updated, (see Figure 38, right), resulting to the creation of a new relationship between the corresponding team and invited user, which also contains the invitation's creator id and timestamp. Afterwards, the server sends back a response to the client informing him that everything went right. Finally, the client displays a success message and updates the UI, accordingly, see Figure 39. The requested user can view and answer this invitation either for the "Network", see Figure 40, or the "Invitations" screen, (we will soon describe this screen). If he accepts it, the current database snapshot will be updated, see Figure 41, and every member of the team will be notified, resulting to updating his UI, see Figure 42.

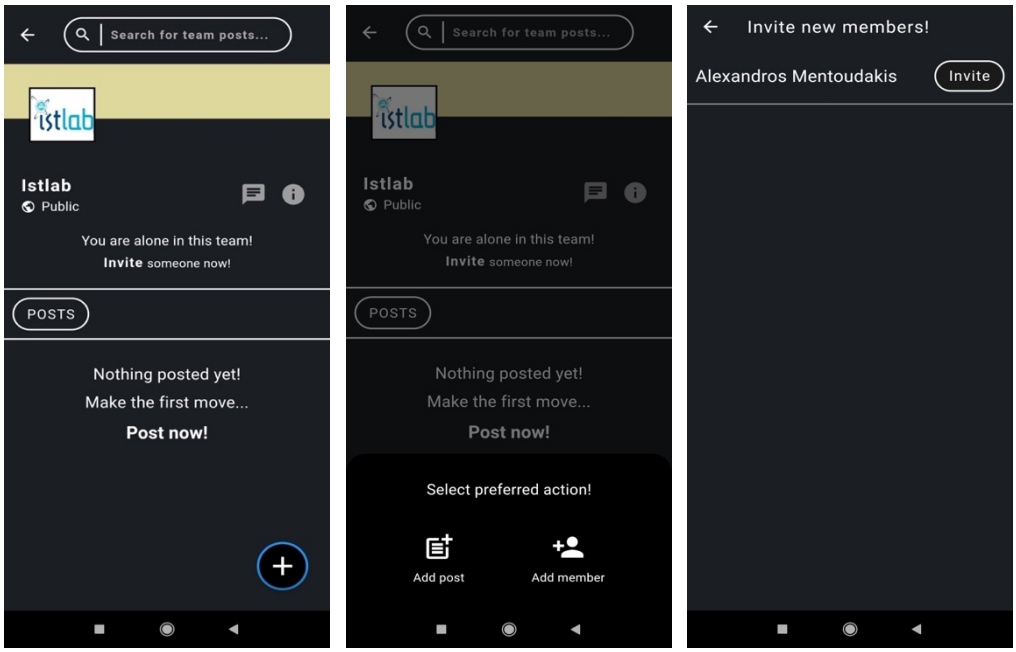


Figure 37: Navigate to "Invite new members" screen.

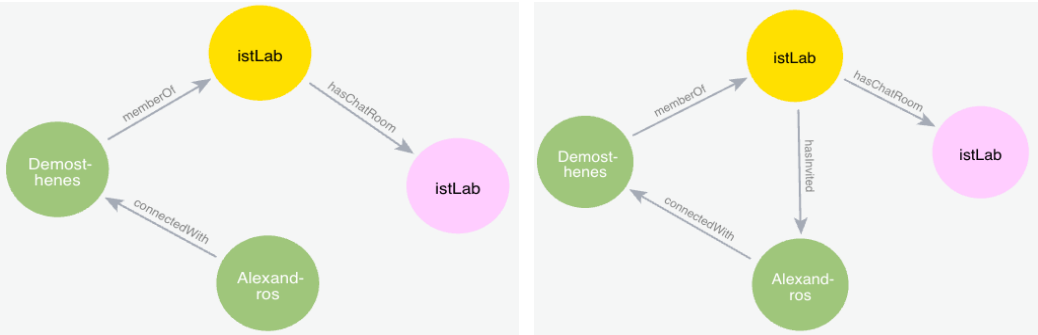


Figure 38: Left: Database snapshot before invitation, Right: Updated database snapshot.

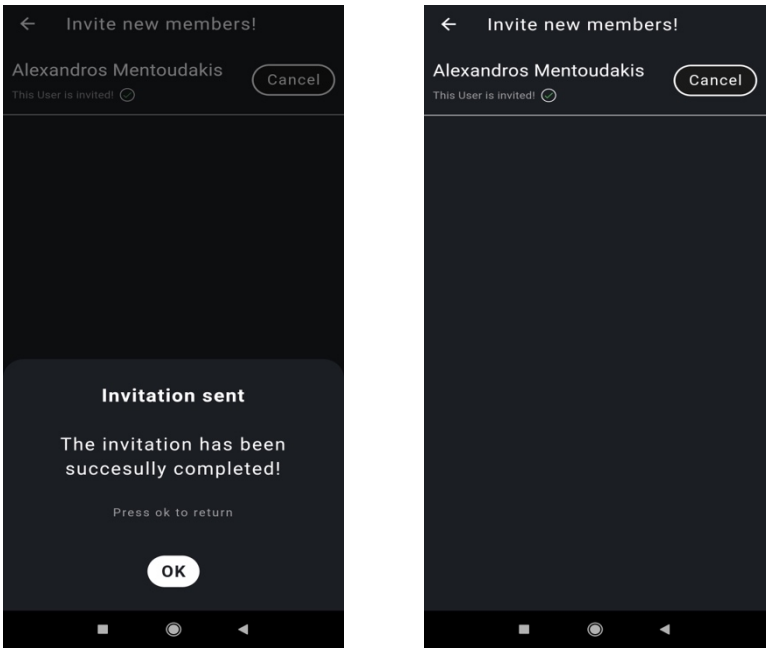


Figure 39: Left: Response message, Right: Updated UI after invitation.

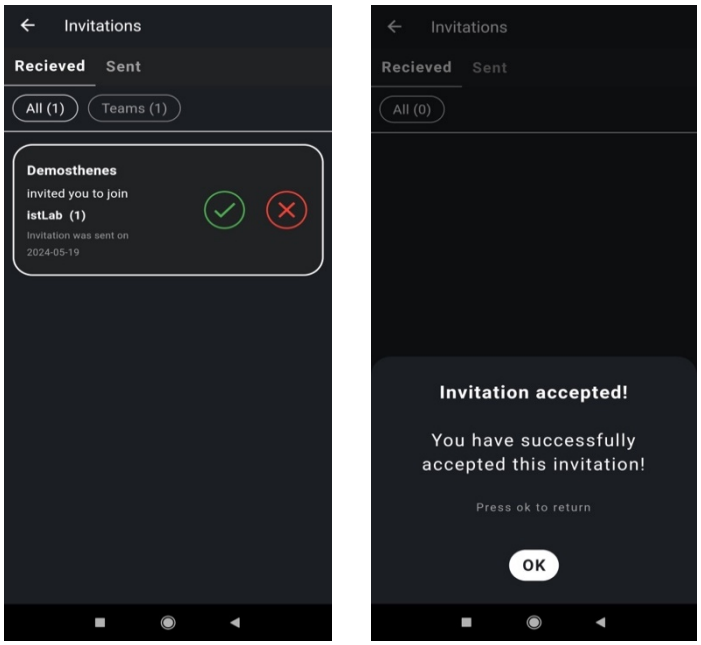


Figure 40: Requested user sees and accepts the invitation from network screen.

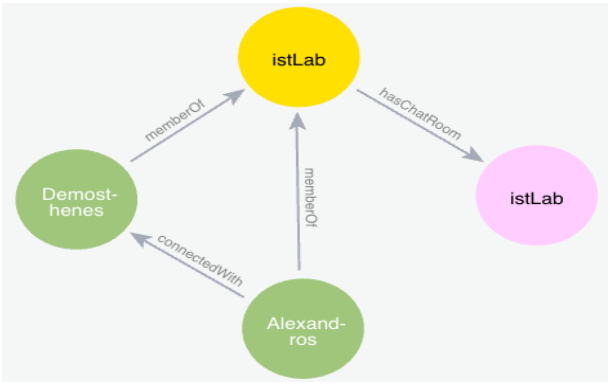


Figure 41: Updated database snapshot after invitation's acceptance.

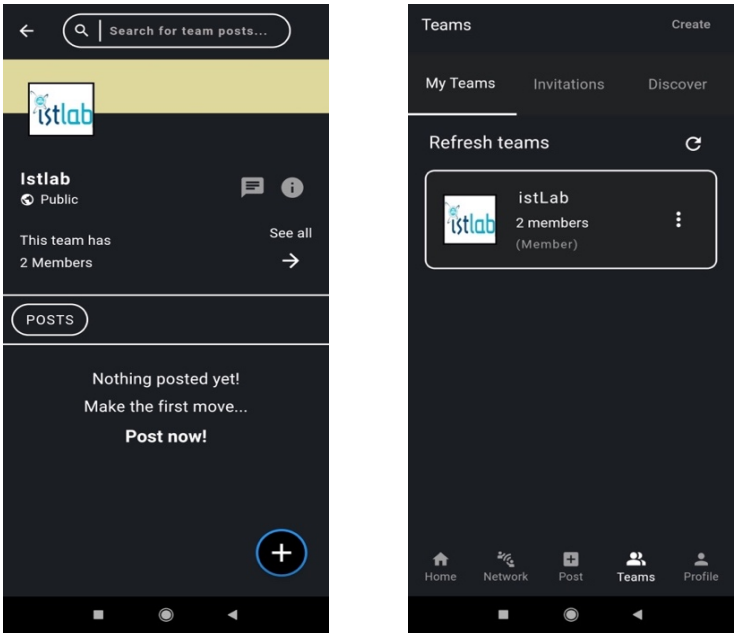


Figure 42: Updated UIs of sender and requested user.

5.8.3. Team Chat Rooms

In this screen, where the members can navigate to, pressing the message icon-button, we have developed a comprehensive interface that serves as a central hub for managing all existing team's chat rooms, see Figure 43. This interface distinctly categorizes those rooms into public and private segments, ensuring seamless navigation and enhanced user experience. Public chat rooms are accessible to all users, fostering open communication and community engagement. In contrast, private chat rooms are restricted to authorized members only, providing a secure environment for sensitive discussions and confidential collaborations. The team screen incorporates intuitive features for easy identification and access to both types of chat rooms.

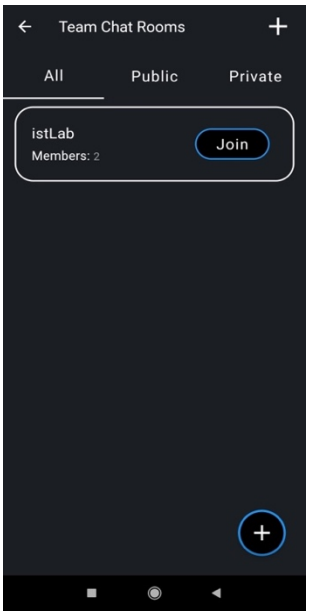


Figure 43: Team chat rooms screen.

5.8.3.1. Creating a public team chat room

Those chat rooms, public or private, can be created from anyone within the team by pressing either the "cross" icon-button in the right side of the app-bar or the icon-button located in the bottom-right of the screen. After pressing either one of two, immediately, a modal pops-up, (see Figure 45, left), where the members need to fulfill the needed information, such as name and type, but also the allowed members if it is a private one. They can add those by pressing the add button of the modal and thus transferring to the associated screen. Following both the completion of necessary data and "Create" button's press, a request is sent to the server, that includes the entered data alongside with the ids of the user and team. After he validates the data, he proceeds with the database

communication, so the current snapshot will be updated, see Figure 44. Finally, it sends back a response to the client, so the UI can also be updated, accordingly, (see Figure 45, right).

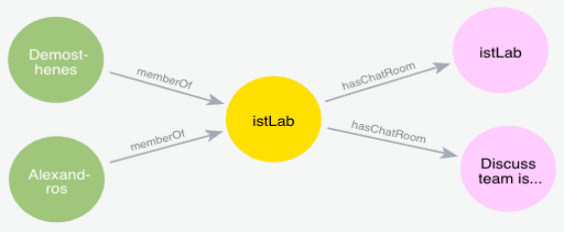


Figure 44: Updated database snapshot with new chat room.

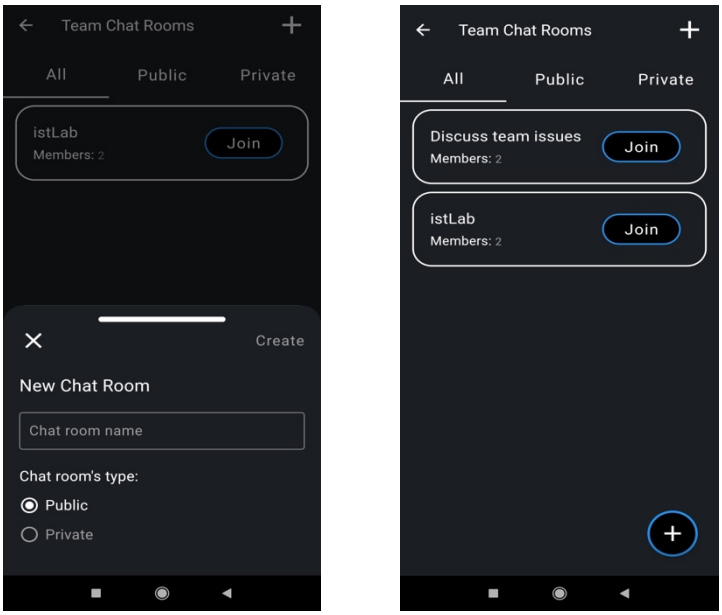


Figure 45: Creation of new team chat room.

To properly demonstrate the procedure of creating a new private room, a third member needs to be added, so we will first connect the third user with the team creator, after we will send an invitation and finally accept it to add him in the team. So, the current database snapshot will look like this, see Figure 46. After that we will create another team chat room, private this time. When the modal opens, we will select the "private" as type and press the "Add member" button, (see Figure 47, left). Immediately we are transferred to another screen, so we can declare who will have access to this chat room, (see Figure 47, center). After adding one of the two members, (see Figure 47, right), we press the "done" button to return. Finally, we press the "Create" one and the new private chat room is added successfully, following the same procedure as the public one, but this time with the addition of a list, that contains the user ids of the allowed members. This room will only be visible to the appropriate users, see Figure 50.

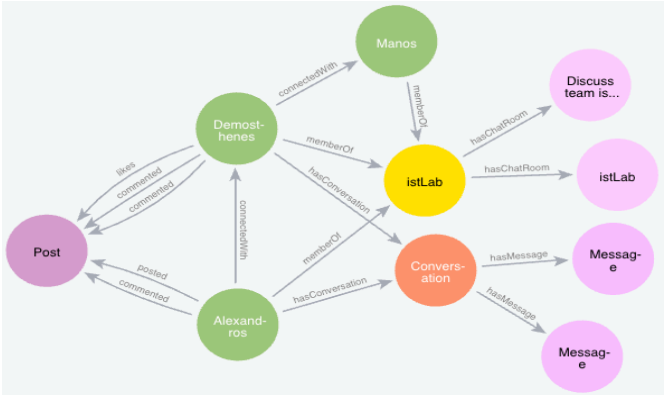


Figure 46: Complete database snapshot.

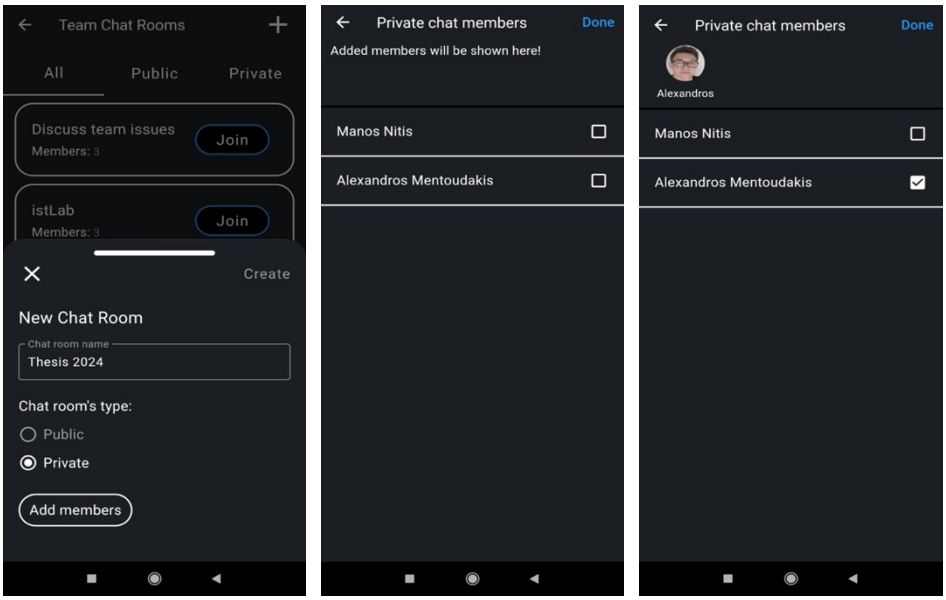


Figure 47: Private type selection and addition of one member.

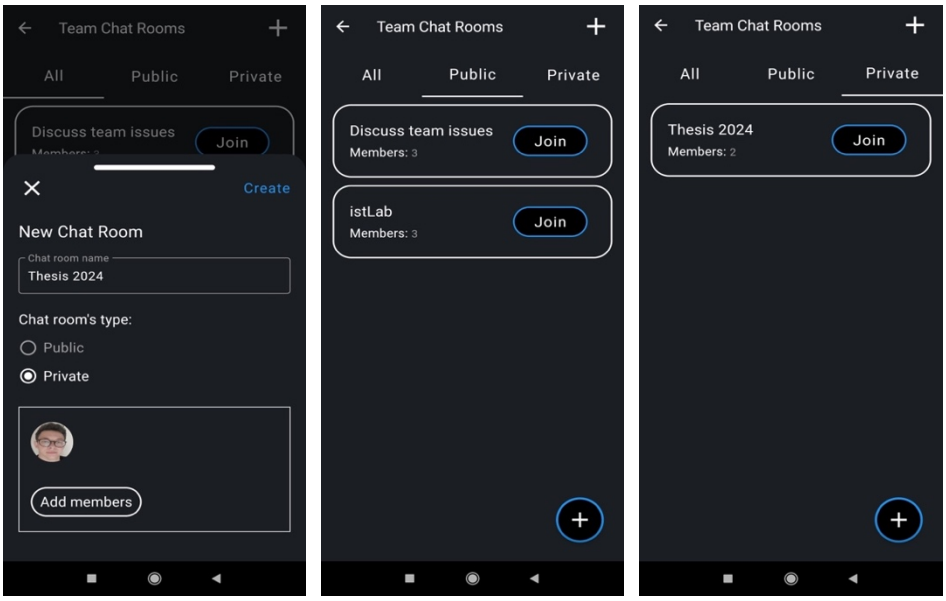


Figure 48: Private chat room creation, swap between different kinds.

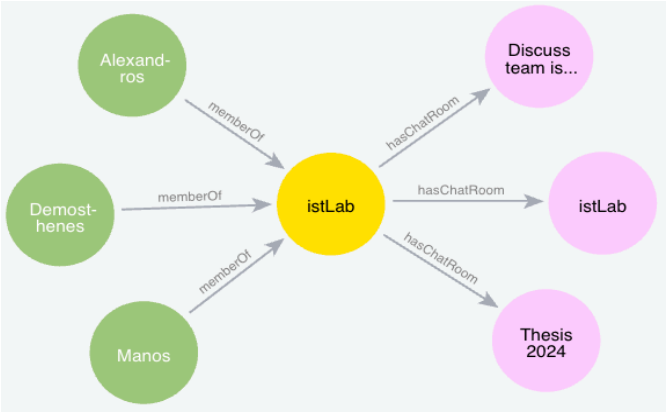


Figure 49: Updated database snapshot with team, member and chat rooms.

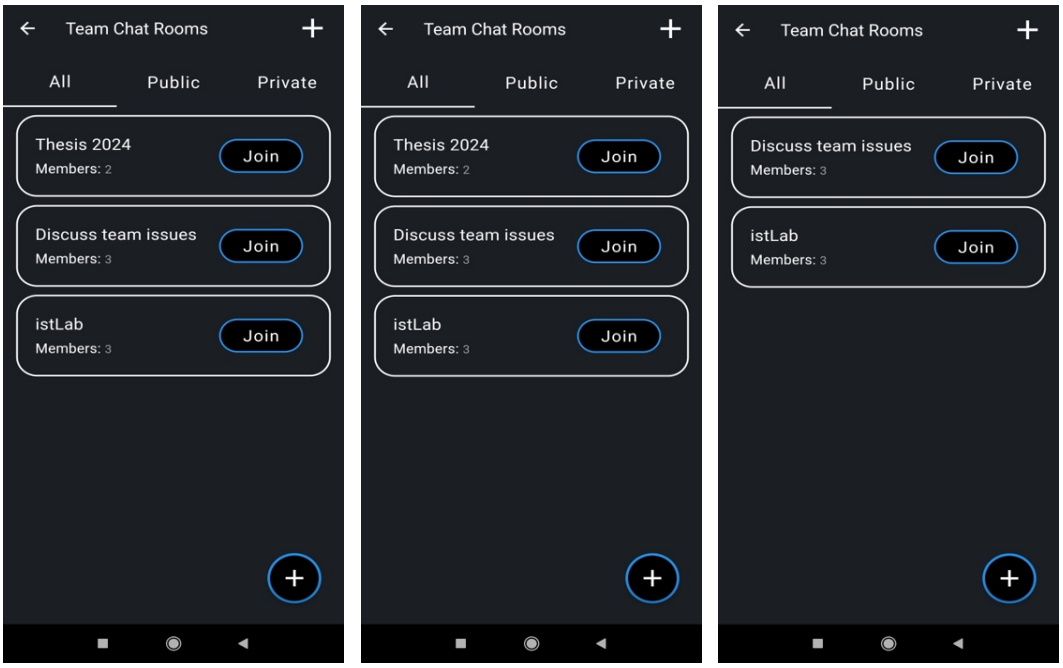


Figure 50: Updated UIs of the three team members.

5.8.3.2. Sending a team Message

Upon pressing the "Join" text-button, members can seamlessly navigate into the corresponding chat room, whether it is public or private, after being authorized. This mechanism is designed to facilitate real-time communication, enabling members to participate in ongoing conversations or initiate new discussions effortlessly. With user-friendly navigation and robust communication tools, this application ensures that team members can stay connected and engage effectively within their respective chat environments. Every message that was sent through a specific team chat room will be related in the database with that corresponding room, storing alongside the content of the message, also its creator id and timestamp. The procedure of sending a message is straight-forward, you just type its content and press the “send” icon-button. Every

message that was sent by a member, who has been removed from the team, will be deleted from both the database and the clients. Finally, those messages are only accessed from users within the team, team members, who also have been authorized to join in the related to them chat rooms.

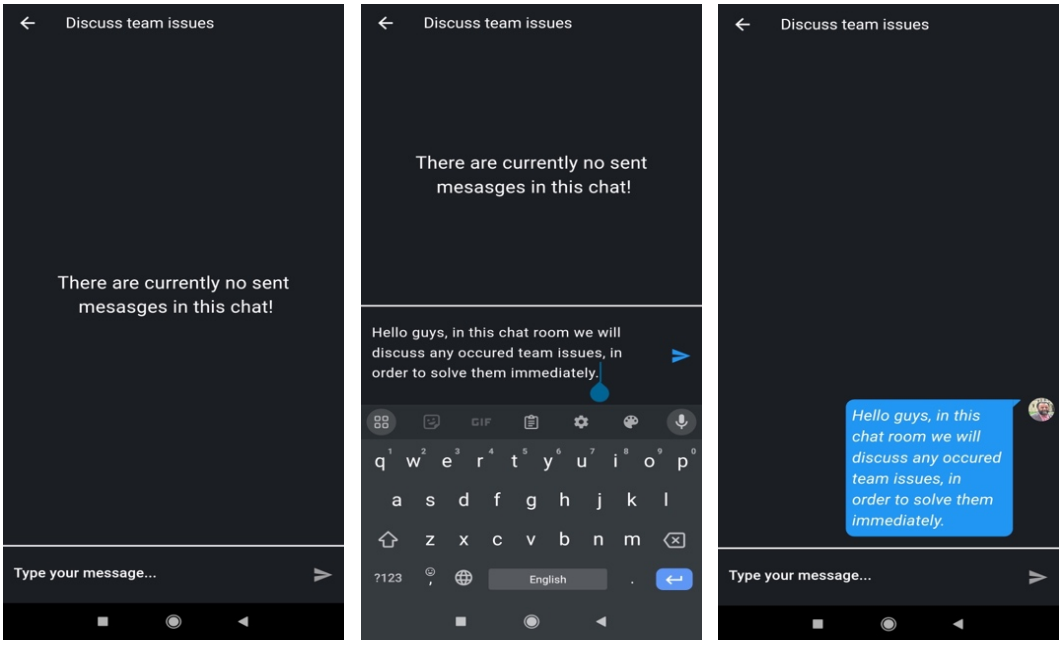


Figure 51: First team message.

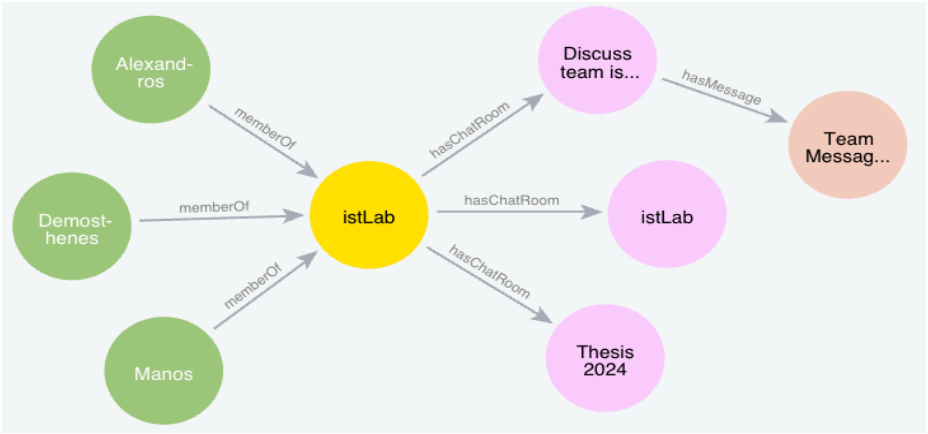


Figure 52: Updated database snapshot after first message.

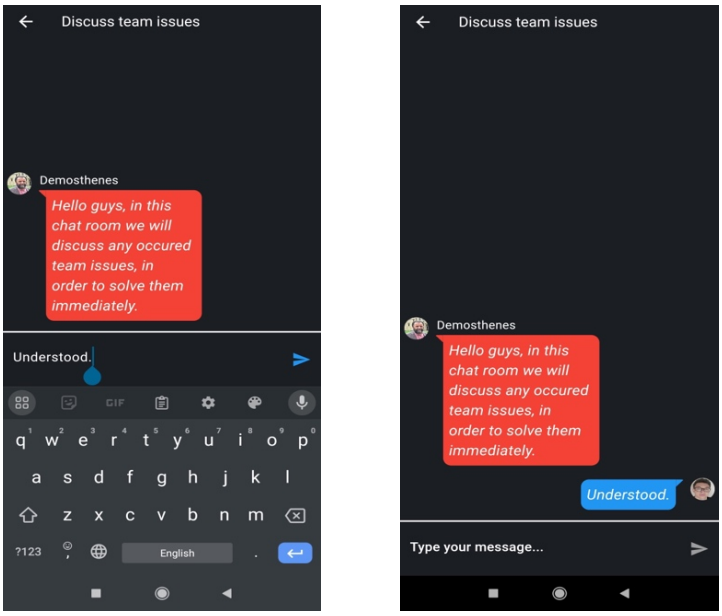


Figure 53: Response from another member.

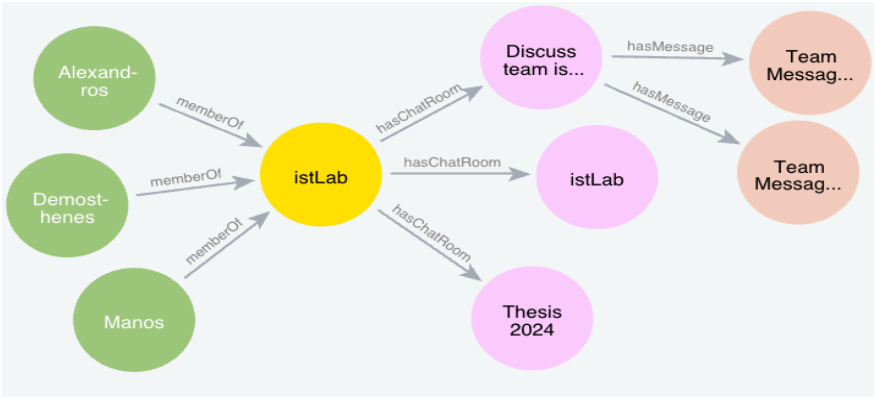


Figure 54: Updated database snapshot after response message.

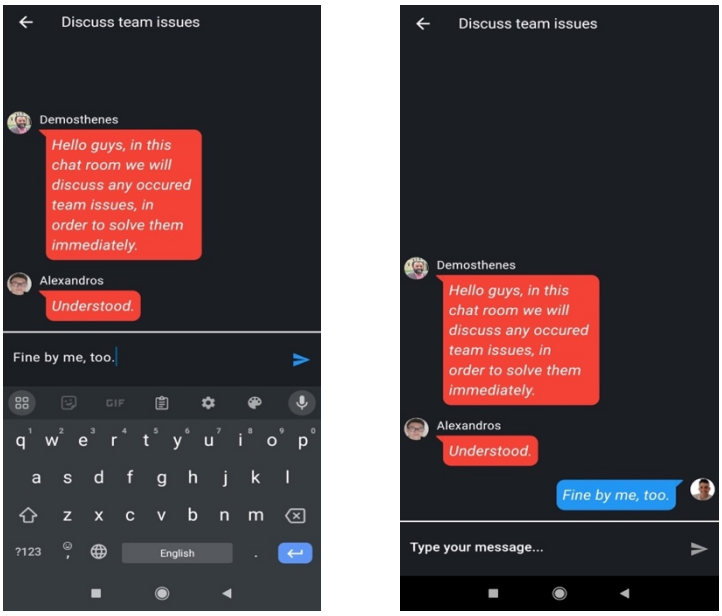


Figure 55: Third member joining the conversation and sending message.

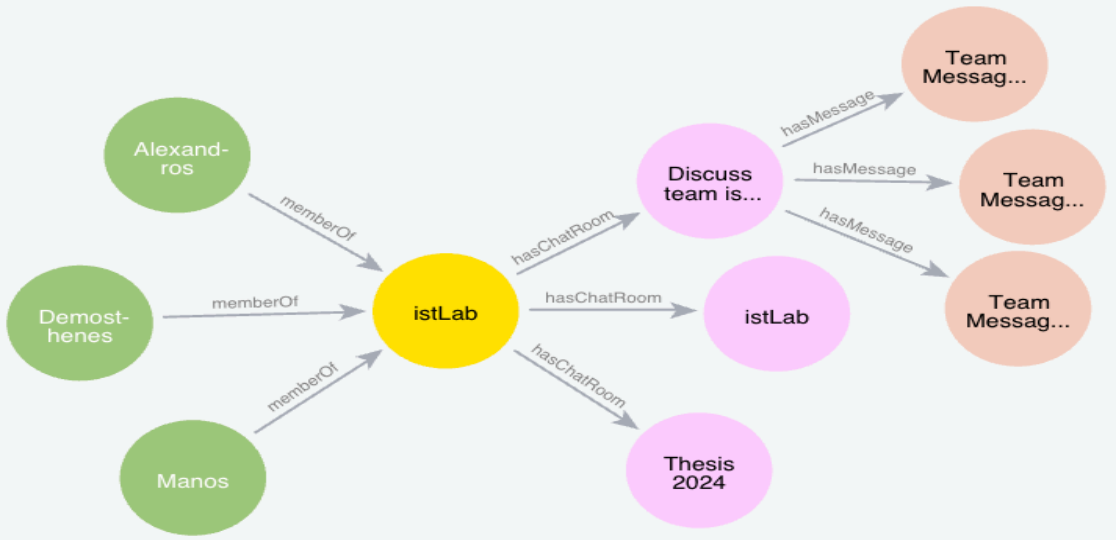


Figure 56: Updated database snapshot after third message.

5.8.4. Members Screen

The "Members Screen" can be accessed from every team that has more than 1 member. Upon tapping the forward arrow, located center-left in the team's main screen, the user is transferred there, (see, Figure 57, left). This screen showcases all team members, each represented by a circular user image, full-name, occupation, and the first letter, capitalized, of his role within this team, A for admin and M for members. By tapping on a member's image, users are seamlessly transferred to that individual's profile, providing quick access to detailed information and facilitating personal connections. Additionally, the screen features a robust search bar, allowing users to efficiently locate any team member by typing their first or last name, (see, Figure 57, center). This functionality streamlines the process of finding and connecting with specific individuals within the team, promoting efficient communication and collaboration. If the logged in user is the creator of the team (admin), a red delete icon-button is added in the end of the container, providing him with the capability of removing a member from this team. Upon pressing it, a modal is being displayed, to ensure that the admin wants to proceed with this action, (see, Figure 57, right). If he then continues, then this member is no longer part of the team and all his uploaded content alongside with every message regardless of the team chat room is also deleted.

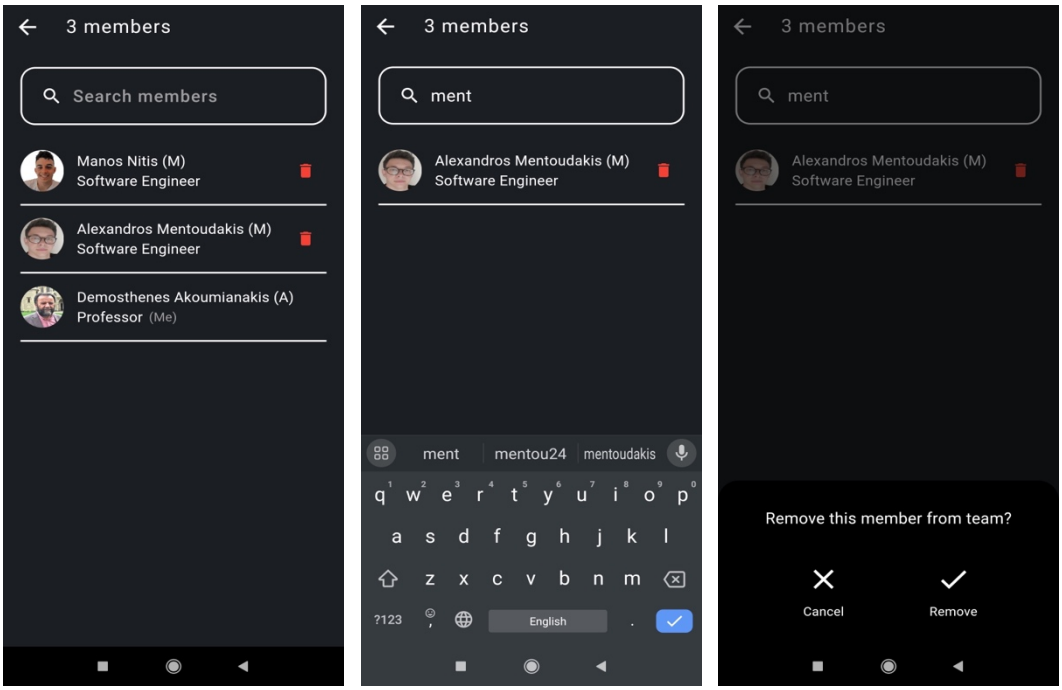


Figure 57: Search for specific member and try to delete it.

5.8.5. Uploading a Team Post

Upon pressing the "Add Post" option, a modal open, from where team members can upload content to their teams, (see Figure 58, left and center). This mechanism is designed to facilitate internal communication, idea sharing, and collaborative announcements within a secure environment. Users can create and upload text posts, accompanied with images, (see Figure 58, right), ensuring a rich and engaging communication experience. Once uploaded, thus resulting to database's snapshot and client's UI updates, see Figure 59, these posts are accessible to all team members, fostering an open and interactive platform for discussions, updates, and feedback. They can interact with them from any of the following screens, "Team Posts Screen", team's main screen or from their associated screens, where they can navigate to by clicking on a post, see Figure 60. Finally, the interaction mechanisms include liking and commenting capabilities.

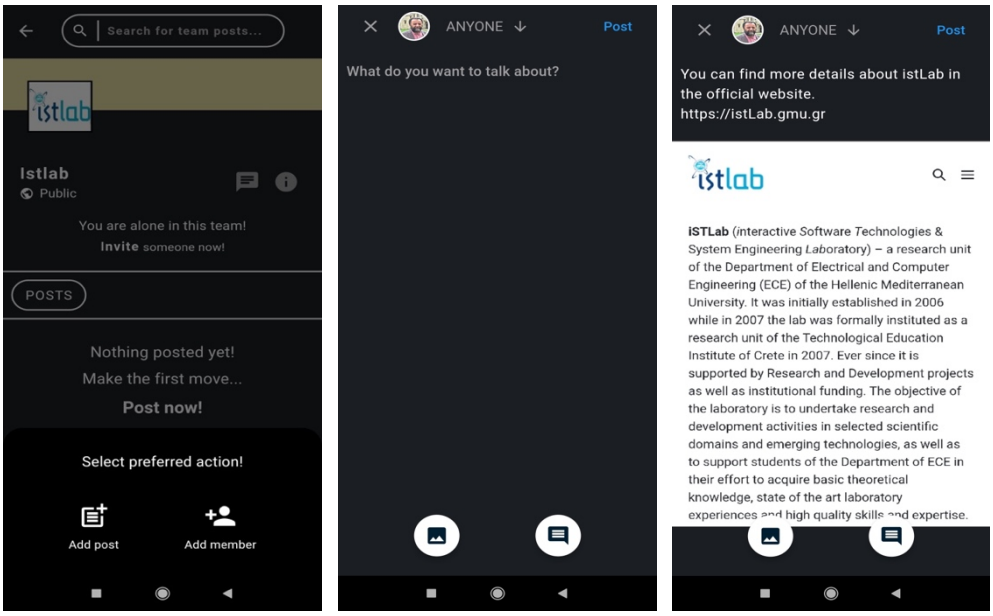


Figure 58: Initialize team post's creation procedure.

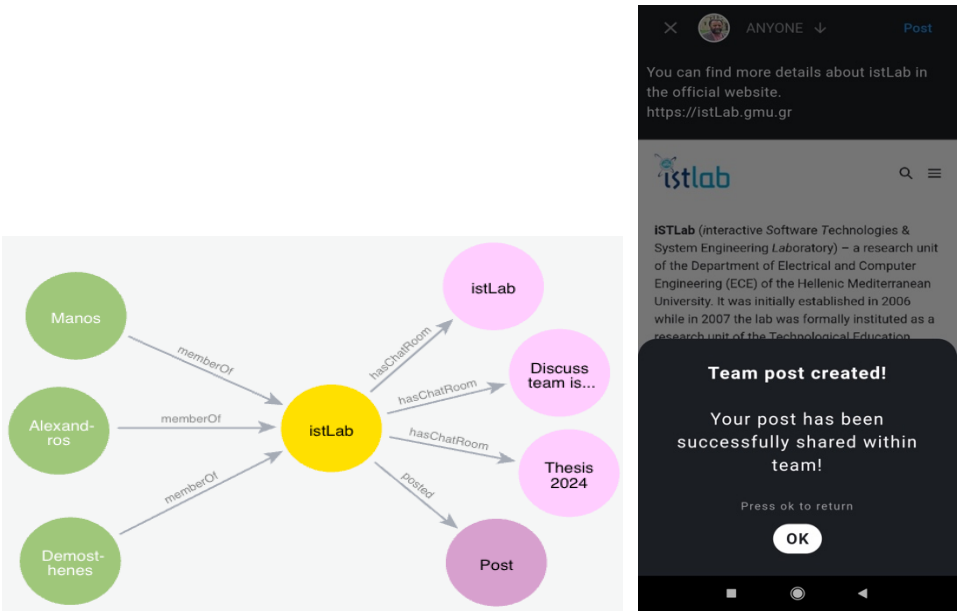


Figure 59: Updated database snapshot and success message.

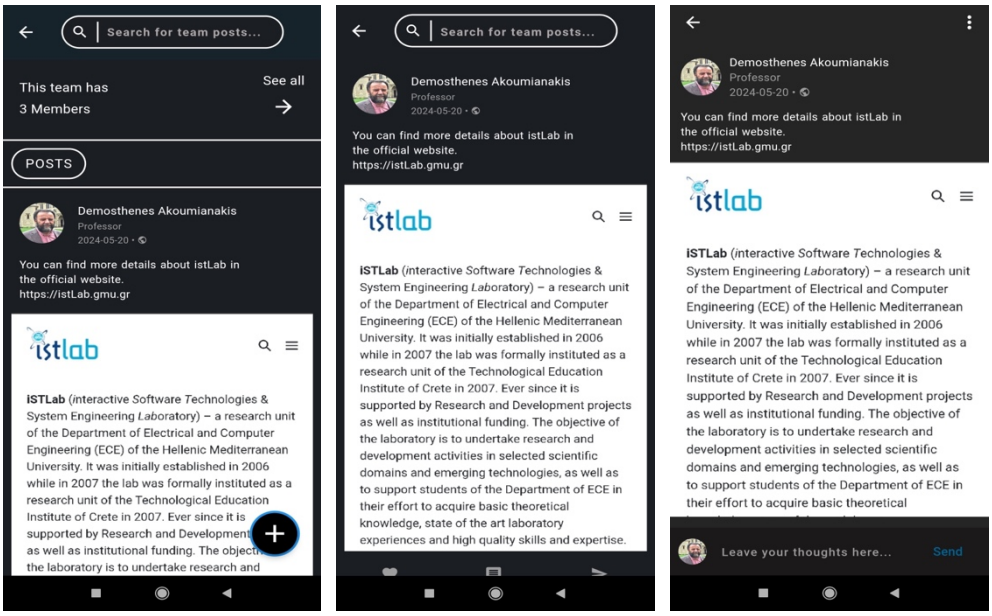


Figure 60: Different screens of interacting with a team post.

5.8.6. Invitations Screen

This is a dedicated screen, where users can view every invitation that is associated with joining a new team, thus providing a centralized location for managing membership requests. This screen displays a comprehensive list of pending invitations, in card-shaped form, see Figure 61, each with details such as, sender's and team's names, number of team's members and invitation date. Users can easily review and respond to these invitations with options to accept or decline directly from this interface. The design ensures that users stay informed about their potential team memberships and can make timely decisions regarding their participation. By streamlining the invitation management process, this screen enhances user experience and facilitates efficient onboarding into new teams. By clicking on either the sender's or team's name the logged in user is transferred to their respectively profile pages. There, he can obtain more useful insights about them before answering the associated invitation.

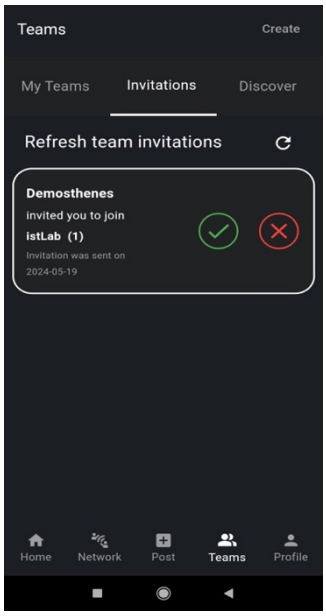


Figure 61: Invitations screen.

5.8.7. Discover Teams Screen

In this screen the user can discover and join new teams. Every team, that is either of type “public” or the logged in user is not already part of, will be displayed here, thus providing him the capability to either visit or become a member of them. The first time a logged in user is navigated to this screen a pop-up message is displayed, prompting him to take a quick guide tour that describes every single utility of this screen, see Figure 62. Following the end of this procedure, the user will be asked to set the filters for the type of teams, that he wishes to view or join in the future, see Figure 63. He can at any point also change those filters by pressing the filter-button that is located on the right of the screen. Finally, he can remove any team from the recommendation’s list, by pressing the “cross” button on the up-right of each card-shaped container. Every team, that has been removed will never be displayed to this user ever again, unless he presses the “Clear removed list” button, thus resulting to resetting the removed teams.

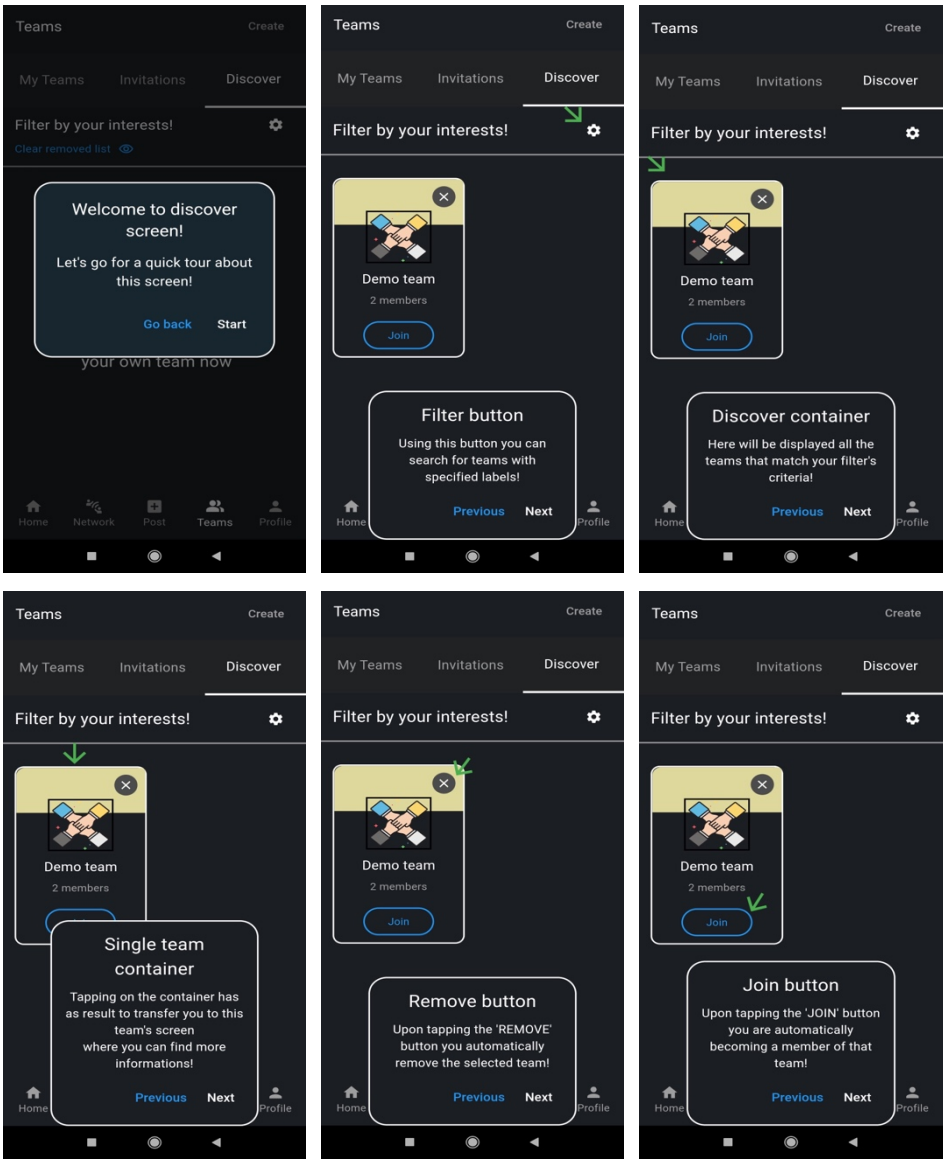


Figure 62: Screen's guide tour.

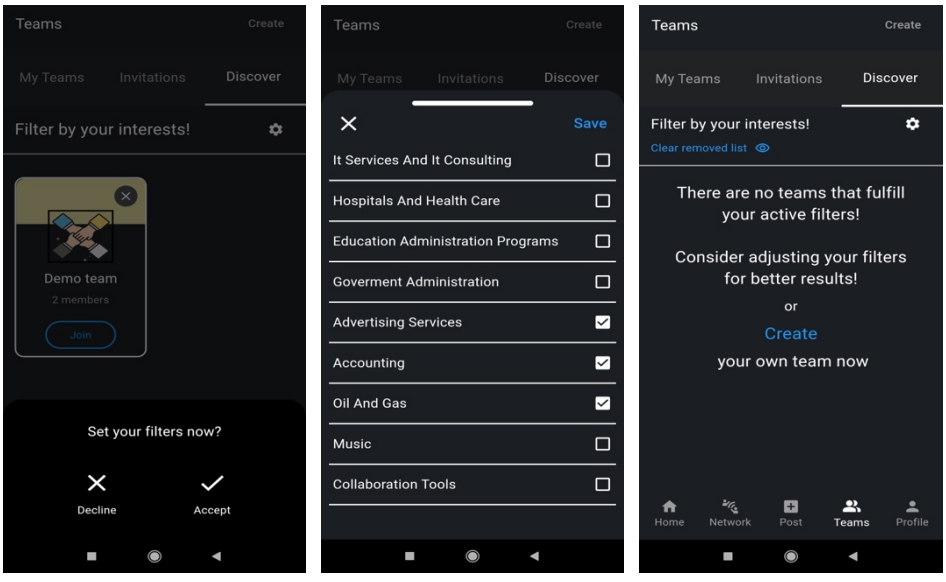


Figure 63: Setting filters for team recommendations.

6. Illustrative path queries examples using Cypher

In this section, we will delve into the practical layer of path queries. Having previously both referred and described their nature of existence, constituents and theoretical background, see sections 2.4 and 2.5.2, we will now dive deeper by providing some examples of path queries, using the data from our application. The main purpose of those implementations is to demonstrate Neo4j's robust power and versatility in real-world scenarios, leveraging the Cypher query language, which enables us to uncover complex relationships and patterns within connected data efficiently. By exploring various examples and scaling up their complexity, we will illustrate how these demanding queries, that require traversing through multiple nodes and relationships, can be implemented and answered effectively, thus showcasing Neo4j's capabilities to manage, transform and analyze interconnected data.

The current database snapshot of our application is as seen below, see Figure 64. This graph contains every node and relationship that was created during the step-by-step demonstration of our application's capabilities in section 6. We will take advantage of those data to showcase some path query examples in Neo4j, using the Cypher language. For each one of the examples, we will describe in detail both the code that was used but also the corresponding actions of the Neo4j database. After everything is explained and analyzed we will also display the appropriate results of the implemented query.

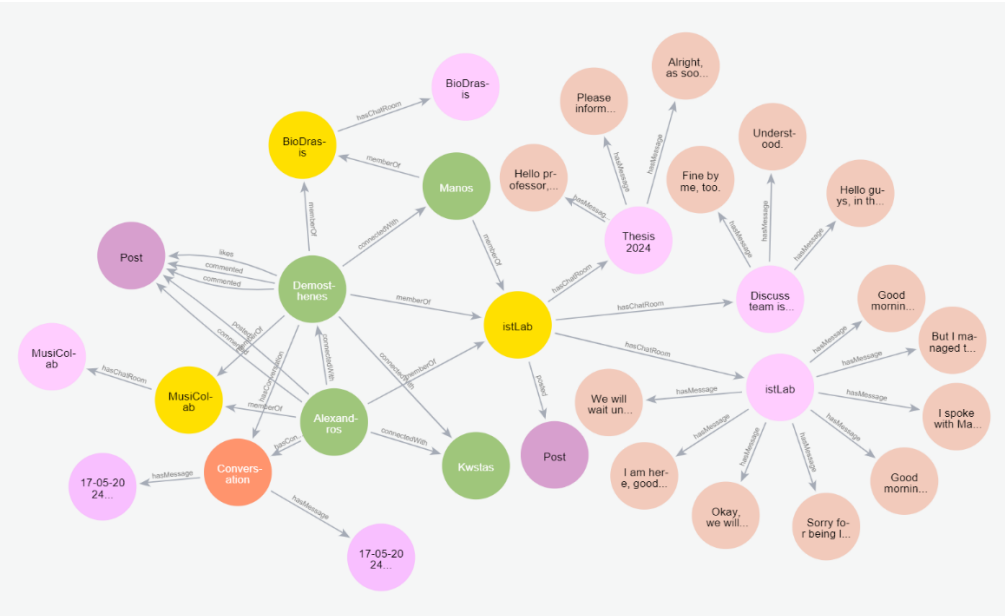


Figure 64: Current Database snapshot.

Query 1: Find and return the names of every existing team alongside a list containing all their corresponding member's names.

```
1 MATCH (foundTeam:Team)
2 OPTIONAL MATCH (foundTeam) <- [:memberOf] - (member:User)
3 WITH foundTeam.teamName AS teamName, COLLECT(member.firstName) AS teamMembers
4 RETURN teamName, teamMembers
```

Figure 65: Implemented Query 1.

Description: The query as depicted above, see Figure 65, starts by using the "MATCH" keyword. The provided pattern of line 1 matches all teams, nodes which contain the "Team" label. This produces three siblings for the variable *foundTeam*, namely T1, T2, T3 represented as three rows in a table with the *foundTeam* identifier, see Figure 66Figure 64, that correspond to the teams MusiColab, istLab and BioDrasis.

foundTeam
T1
T2
T3

Figure 66: First match clause bindings.

The "OPTIONAL MATCH" keyword in line 2 matches all the users who are members of each team. This yields a binding of the newly introduced variable member to which *foundTeam* was bound by the pattern of line 1, see Figure 67, left. The "WITH" keyword of line 3 is used both to project a subset of variables which are currently in the scope and their bindings after succeeding the first two "MATCH" clauses and to provide the identifiers with which we can access the data. In line 3, we have also used the "COLLECT" keyword, that is responsible for inserting all the members in a list, see Figure 67, right.

foundTeam	member
T1	U1
T1	U2
T2	U3
T2	U1
T2	U2
T3	U3
T3	U2

foundTeam	member
T1	[U1, U2]
T2	[U3, U1, U2]
T3	[U3, U2]

Figure 67: Variable bindings before and after using the "with" keyword.

Finally, the "RETURN" keyword of line 4, projects the data that were retrieved through their associated identifiers. So, the name of every team is projected, along with their corresponding members in a list, as depicted below, see Figure 68.

- **Query results**

```
1  "MusiColab"     ["Alexandros", "Demosthenes"]
2  "istLab"        ["Manos", "Alexandros", "Demosthenes"]
3  "BioDrasis"     ["Manos", "Demosthenes"]
```

Figure 68: Results of implemented query 1 in table form.

Query 2: Find and return the names of every user alongside a list containing the names of each potential new connection based on his already existing ones.

```
1 MATCH (initialUser:User) - [:connectedWith] - (connectedUser:User)
2 OPTIONAL MATCH (connectedUser) - [:connectedWith] - (potentialConnection:User)
3 WHERE initialUser <>  potentialConnection AND NOT EXISTS {
4   (initialUser) - [:connectedWith] - (potentialConnection)
5 }
6 WITH initialUser.firstName AS userName, COLLECT(potentialConnection.firstName) AS potentialConnections
7 RETURN userName, potentialConnections
```

Figure 69: Implemented query 2.

Description: The query as depicted above, see Figure 69, starts by using the "MATCH" keyword to match the provided pattern of line 1, which initially finds all the users, nodes with label *User*, (see Figure 70, left), and continues with binding each one of them with their associated connections. This produces the following table, with two columns, for the variables initialUser and connectedUser, where each of the users, Alexandros, Demosthenes, Kwstas and Manos, is represented by U1, U2, U3, U4, see Figure 70, right.

initialUser	connectedUser
U1	U2
U1	U3
U2	U1
U2	U3
U2	U4
U3	U1
U3	U2
U4	U2
U4	U2

initialUser	connectedUser	potentialConnection
U1	U2	U4
U1	U3	
U2	U1	
U2	U3	
U2	U4	
U3	U1	
U3	U2	U4
U4	U2	U3
U4	U2	U1

Figure 70: Bindings after the match clause in line 1.

Both keywords of line 2 and 3, "OPTIONAL MATCH" and "WHERE", by working simultaneously, they find the existing connections while filtering out those of column 2 that are either the same as column 1 or already exist as pair in the first two columns, see Figure 71. Furthermore, by using both the "WITH" keyword, in line 6, we project a subset of the variables currently in scope and their bindings to the succeeding part of the query and the "COLLECT" keyword, also in line 6, we group the retrieved data within a list, see Figure 72.

initialUser	potentialConnection
U1	U4
U2	
U3	U4
U4	U3
U4	U1

Figure 71: Bindings after the optional match clause.

initialUser	potentialConnection
U1	[U4]
U2	[ ]
U3	[U4]
U4	[U3, U1]

Figure 72: Transformed bindings using the collect keyword.

Finally, in line 7, the "RETURN" keyword, is being used to project the name of each user alongside the potential connections list. The table consisting of the rows containing the result of the expressions is as depicted below, see Figure 73.

- **Query results**

	userName	potentialConnections
1	"Alexandros"	["Manos"]
2	"Demosthenes"	[]
3	"Kwstas"	["Manos"]
4	"Manos"	["Kwstas", "Alexandros"]

Figure 73: Results of implemented query 2 in table form.

7. Conclusion

In this section, the author will conduct a comprehensive evaluation of this thesis, reflecting on the insights gained from his endeavor and suggesting potential avenues for potential expansion of the research.

7.1. Assessment

This thesis has provided the author with the opportunity to conduct research on the area of social networks, obtaining invaluable insights and benefits from their corresponding methodologies and components. By delving into the intricated and demanding implementation (design and development) of a social network application from scratch, the author has enriched his skill set, providing hands-on experience in mobile application development, back end systems and graph database management. Furthermore, the interdisciplinary nature of this project has fostered a holistic understanding of software engineer principles, from user interface design to data modeling and back end architecture. Moreover, grappling with the challenges inherent in build social network applications has honed the author's problem-solving abilities and project management skills. By navigating through the complexities of integrating these cutting-edge technologies (Flutter, Node.js, Neo4j), the author has not only expanded his technical proficiency but also cultivated a deeper appreciation for the nuances of building robust, scalable software solutions. Finally, following the author's choice of Neo4j's usage in this project, he had the potential to explore some of the most valuable capabilities that a graph database management system has to offer.

7.2. Future research and extensions

Whenever we discuss about social network applications, we always come up with brand-new ideas to implement inside it. Thus, understanding that there are unlimited possibilities, and the only limit is the sky, as they say. So, we will provide some useful ideas that could be added inside this application. First, we could create a task management system and collaboration tool, that will provide the teams with the utility to increase their members' productivity. They (team members) could be assigned specific tasks of a project and by creating/editing content (files, etc..) in real-time, increase their collaboration skills and team spirit. Second, we could implement a mechanism that will enable teams to schedule and conduct meetings either to share new information and

knowledge or to work together in real time on a single task. Third, we could also provide the utility of shared team's spaces, through which two or more distinct teams can create a digital space inside the application, to work together upon some shared project that they might agree on. Finally, we could add calls and videos inside every conversation room between two users.

8. References

1. W. Kim, O.-R. Jeong, and S.-W. Lee, “On social Web sites,” *Inf. Syst.*, vol. 35, no. 2, pp. 215–236, Apr. 2010.
2. C. P.-Y. Chin, K.-K. R. Choo, and N. Evans, “Enterprise Social Networks: A Successful Implementation within a Telecommunication Company,” p. 12.
3. Martina Drahošova, Peter Balco (2016) The Benefits and Risks of Enterprise Social Networks , Department of Information Systems, Comenius University, Faculty of Management, Bratislava, Slovakia
4. Mark Fidelman (2013) “The Rise of Enterprise Social Networks”, https://www.slideshare.net/jdeyaref/the-riseofenterprisesocial
5. J. Wyatt, S. Djamasbi, and M. Traietti, “User Experience of Enterprise Social Networks and Collaboration,” *Enterp. Soc. Netw.*, p. 5.
6. J. DiMicco, D. R. Millen, W. Geyer, C. Dugan, B. Brownholtz, and M. Muller, “Motivations for social networking at work,” in *Proceedings of the ACM 2008 conference on Computer supported cooperative work - CSCW ’08*, San Diego, CA, USA, 2008, p. 711.
7. G. Stei, S. Sprenger, and A. Rossmann, “Enterprise Social Networks: Status Quo of Current Research and Future Research Directions,” in *Business Information Systems*, vol. 255, W. Abramowicz, R. Alt, and B. Franczyk, Eds. Cham: Springer International Publishing, 2016, pp. 371–382.
8. P. Mathiesen and E. Fielt, “Enterprise Social Networks: A Business Model Perspective,” p. 14, 2013. S. Han, J. Min, and H. Lee, “Building relationships within corporate SNS accounts through social presence formation,” *Int. J. Inf. Manag.*, vol. 36, no. 6, pp. 945–962, Dec. 2016.
9. Lourenço, J.R., Cabral, B., Carreiro, P. et al. Choosing the right NoSQL database for the job: a quality attribute evaluation. Journal of Big Data 2, 18 (2015). https://doi.org/10.1186/s40537-015-0025-0
10. Schram, A., Anderson, K.M.: MySQL to NoSQL: data modeling challenges in supporting scalability. In: Proceedings of the 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH 2012), pp. 191–202. ACM, New York (2012)
11. Cattell, R.: Scalable SQL and NoSQL data stores. SIGMOD Rec 4, 12–27 (2011), doi:10.1145/1978915.1978919
12. Stonebraker, M.: SQL databases v. NoSQL databases. Commun. ACM 53(4), 10–11 (2010), doi:10.1145/1721654.1721659
13. Robinson, I., Webber, J., Eifrem, E.: Graph Databases – New Opportunities for Connected Data, O'Reilly Media, Inc., ISBN:978-1-4919-3089-2
14. Wu, Y.: Path Query. In: LIU, L., ÖZSU, M.T. (eds) Encyclopedia of Database Systems. Springer, Boston, MA., 2009, https://doi.org/10.1007/978-0-387-39940-9_140
15. Srivastava, S., Singh, A.K. Fraud detection in the distributed graph database. Cluster Comput 26, 515–537 (2023). https://doi.org/10.1007/s10586-022-03540-3
16. Sen, S., Mehta, A., Ganguli, R. et al. Recommendation of Influenced Products Using Association Rule Mining: Neo4j as a Case Study. SN COMPUT. SCI. 2, 74 (2021). https://doi.org/10.1007/s42979-021-00460-8
17. Diederichsen, L., Choo, KK.R., Le-Khac, NA. (2019). A Graph Database-Based Approach to Analyze Network Log Files. In: Liu, J., Huang, X. (eds) Network and System Security. NSS 2019. Lecture Notes in Computer Science(), vol 11928. Springer, Cham. https://doi.org/10.1007/978-3-030-36938-5_4

18. A. Celesti, A. Buzachis, A. Galletta, G. Fiumara, M. Fazio and M. Villari, "Analysis of a NoSQL Graph DBMS for a Hospital Social Network," 2018 IEEE Symposium on Computers and Communications (ISCC), Natal, Brazil, 2018, pp. 01298-01303, doi: 10.1109/ISCC.2018.8538469.
19. Y. Zou and Y. Liu, "The Implementation Knowledge Graph of Air Crash Data based on Neo4j," 2020 IEEE 4th Information Technology, Networking, Electronic and Automation Control Conference (ITNEC), Chongqing, China, 2020, pp. 1699-1702, doi: 10.1109/ITNEC48623.2020.9085182.
20. Kumar, N., Mukhtar, S. (2023). Building Protein–Protein Interaction Graph Database Using Neo4j. In: Mukhtar, S. (eds) Protein-Protein Interactions. Methods in Molecular Biology, vol 2690. Humana, New York, NY. https://doi.org/10.1007/978-1-0716-3327-4_36
21. Francis, N.; Green, A.; Guagliardo, P.; Libkin, L.; Lindaaker, T.; Marsault, V.; Plantikow, S.; Rydberg, M.; Selmer, P.; Taylor, A. Cypher: An Evolving Query Language for Property Graphs. In Proceedings of the SIGMOD, Houston, TX, USA, 10–15 June 2018; pp. 1433–1445.