StarPoint CRM

Independent Study Thesis

Presented in Partial Fulfillment of the Requirements for
the Degree Computer Science in the
Mathematics and Computer Science Department at The
College of Wooster

by
Erick Delgado
The College of Wooster
2024

Advised by:

Thomas Montelione (Mathematics and

Computer Science Department)

THE COLLEGE OF
WOOSTER

© 2024 by Erick Delgado

Abstract

This study introduces the design and analysis of a web-based Customer Relationship Management (CRM) system. It focuses on comparing different database structures and using normal forms to make storing and accessing data more efficient. The main goal of this study is to explore how databases for CRM systems can be designed to be secure, efficient, and reliable, especially when handling sensitive information. By exploring various database designs and their adherence to normal forms, the research seeks to find the best ways to organize customer data to improve security and system performance. It describes a CRM system that protects sensitive information and allows administrators to easily update member data, showing how database design affects both system operation and user experience. Using MySQL, PHP, and web hosting services, this study lays the groundwork for understanding the role of database architecture in developing CRM systems. It involves a thorough review of database design literature and a practical examination of these designs in a CRM setting. This research is a foundation for future work in CRM systems, emphasizing the importance of database design in effectively managing customer relationships. By starting with database architectures and normal forms, the study adds to the discussion on how to enhance CRM databases for better security and efficiency, providing valuable insights for both developers and researchers.

This work is dedicated to my family.

Acknowledgments

I would like to acknowledge Dr. Thomas Montelione for guiding me through a tumultuous time in my academic career. Without his knowledge, wisdom, and much needed discipline this thesis would not have been achievable.

Contents

List of Figures

List of Listings

CHAPTER 1

Introduction

Businesses today have become incredibly reliant upon technology and utilize it to further advance their goals. Thus, appropriate access to data establishes the basis for operational effectiveness, leading to more informed decisions. As businesses expand, they often face a critical decision in selecting the appropriate systems to handle their data repositories.

In the recent past, many enterprises relied on File Management Systems (FMS) for data storage and retrieval. While FMS offers simplicity, it often falls short in areas of security, data redundancy, field consistency, and potentially leading to operational inefficiencies. These inefficiencies can be seen in delayed data retrieval times, large increases in error rates, and even data losses, placing undue strain on employees. Often, the business must spend additional time correcting these errors.

Enter Database Management Systems (DBMS): designed with advanced functionalities, they directly address the limitations inherent in their FMS counterparts. In elementary terms, a DBMS is "a set of prewritten programs that are used to store, update, and retrieve a database." [27] Beyond this basic function, a DBMS ensures data is both systematically organized and readily accessible. This becomes especially critical in areas like a Customer Relationship Management (CRM), where timely and accurate data retrieval can make the difference between a missed opportunity and a satisfied customer. By incorporating a DBMS, businesses can capitalize on its ability to provide seamless data integration and minimize data duplication issues.

As CRM systems become increasingly central to business strategies, the integration of a well-rounded DBMS becomes imperative.

The main goal of this research topic is the design and implementation of a secure web-based Customer Relationship Management (CRM) system with an emphasis on functionality and data security. McCain found that companies use CRM tools because they enhance communication tracking with customers and allow data-driven decision-making. [24] Ninety one percent of larger companies use CRM tools, which has led to a reported increase in sales revenue by 45%. The company's return on investment is impressive at $8.71 for every dollar spent and 47% of firms see improved customer retention. Additionally, CRM adoption can boost conversion rates by 300%. [24] With this in mind, the project goal is to create a secure web-based application that will integrate an efficient database into a customer relationship management application. The software aims to create a solution where businesses can meet their customer's needs in one place including sales management, lead distribution, reporting, forecasting, customer service, and rewards.

CHAPTER 2

Background Information

There are multiple definitions for what a customer relationship manager (CRM) is but put most simply "a CRM tool technically is just a database empowered with some specific functions by an overlaying software" [27]. This technology aids organizations in managing contacts, identifying potential opportunities, and managing marketing campaigns. By leveraging CRM systems, companies can gain valuable insights into customer behavior, thereby leading to larger profit margins. Based on data from Statista [1], which references a Gartner study, the revenue for CRM companies has grown from nearly $14 billion in 2010 to approximately $69 billion in 2020. This marks a rise of around $55 billion or an impressive 393% growth. While this doesn't directly represent adoption rates, the revenue trends provide insights into the growing use of CRMs in businesses (Figure 2.1).

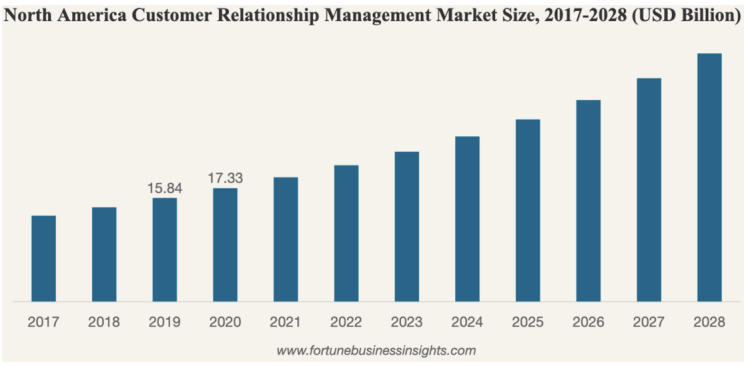

Figure 2.1: North America CRM Market Growth from 2017 projected to 2028 [24]

The meteoric growth in CRM revenue directly displays the immense value

companies derive from these tools. The transformation of a database into a function-rich platform facilitates deeper customer insights and more personalized interactions. As businesses increasingly recognize the importance of customer-centric approaches in today's competitive landscape, it's evident that CRMs have transitioned from being optional tools to essential components in successful business operations. Opting for a Customer Relationship Manager (CRM) over an inventory management system in the application developed in this thesis is pivotal due to the emphasis on fostering interpersonal relationships with customers rather than managing physical material goods. A CRM puts customers first in its design principles, enabling an organization to analyze and predict customer behaviors, tailor personalized experiences, and build lasting relationships. Inventory management systems predominantly focus on the logistical aspects of products, optimizing stock levels, and prioritizing order fulfillment. Inventory management systems lack the capability to strategically manage customer interactions. Thus, the application underscores the belief that investing in customer relationships transcends the operational efficiencies gained by optimizing inventory.

It is critical to distinguish between customer relationship management (CRM) software applications and inventory management systems because they serve fundamentally different aspects of business operations. While inventory management systems are designed to optimize stock levels and streamline order processing, CRM software focuses on enhancing interactions with customers, building relationships, and improving customer satisfaction. This distinction is important because it highlights the need for a robust approach to business management, where optimizing internal operations through inventory management must be complemented by external customer engagement strategies facilitated by CRM to ensure sustainable business practices.

2.1 Databases and Database Management Systems

At the core of every CRM lies a robust database and an effective database management system (DBMS). A database, is a structured collection of interrelated data, serves as the backbone, ensuring that customer information is stored, retrieved, and manipulated efficiently to furnish the CRM's functionalities. The DBMS, on the other hand, acts as an interface between the database and the end-users, ensuring that data is organized and remains easily accessible. The ensuing sections will elaborate upon various database models, specifically exploring the distinctions and applications of relational versus non-relational databases.

2.1.1 What is a Database?

A database can be described as a structured, self-describing collection of related records. More specifically a relational database, which is predominantly used today and the focus of this study, is a database that is seen as a self-describing compilation of interconnected tables. The term "self-describing" is essential in understanding databases. An example of related tables can be seen in how the ADVISER and STUDENT tables might share a common column, such as Adviser-Name.

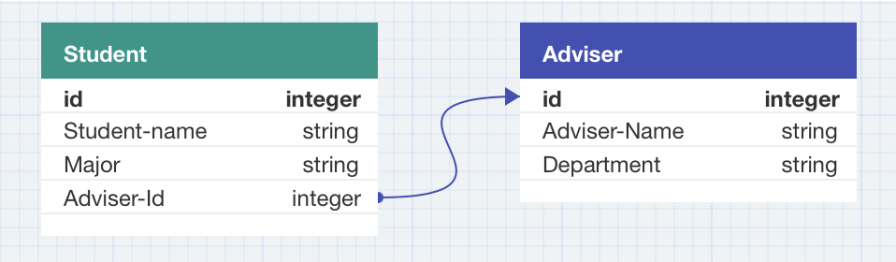

Figure 2.2: Database Relation Example

The concept of a database being "self-describing" put most simply carries its own information about its internal structure. This intrinsic characteristic ensures that anyone can determine the contents of a database by simply examining it internally.

This can be compared to how a library's catalog, found within the library, offers insights into its collection of books. In sum, a database contains user data, metadata, and performance-enhancing structures like indexes. For a collection to be termed a 'database', it needs to: represent real-world aspects, maintain logical coherence, and serve a specific objective for a designated group of users [22].

2.1.2 Relational Databases

Relational Databases have become the preferred medium of choice for holding data because of what they are able to accomplish. A Relational Database organizes its data into one or more tables. Each of these tables has columns and rows with an associated unique identifier often referred to as a key. These unique keys, which may be assigned as a primary or foreign, serve as a critical element to ensure data integrity. A foreign key acts to establish a vital linkage between two tables by referencing the primary key of the other, thereby creating relational connections amongst distinct tables. Both primary and foreign keys collectively contribute to the systematic management of data within database systems. This allows the data, stored in structures, to be related to other data, typically through these common identifiers. Thus, the term "relational" comes from how the data is accessed, organized, and related. Another key characteristic of relational databases is the use of a schema that pre-defines the structure of the data, including its tables, columns, data types, and relationships. A Structured Query Language or SQL is commonly used to manage the create, read, update, and delete operations (CRUD) within the database. Each piece of data in a relational database correlates to a specific row and column within a table, allowing for efficient CRUD operations. The ability to establish relationships between different tables through keys enhances data integrity and enables complex queries and data analysis [19]. Relational Databases are by

far the most popular types of databases and are very closely related to spreadsheet structure.

2.1.2.1 Relational Database Management Systems

A relational database management system (RDBMS) is a type of database management software that stores, organizes, and retrieves data, using a structured format of rows and columns to facilitate data relationships and ensure data integrity. RDBMS organizes data in a structured tabular format, consisting of rows and columns that not only ensure the data is easily visualized but also maintains unique entries and consistent data types within each column. In the realm of RDBMS, technical jargon often translates simple concepts into more complex terminology: a "table" is known as a "relation," a "row" or "record" is termed a "tuple," and the entire database formed by these relations is referred to as a "relational database." This system, grounded in the mathematical theory of relations, adopts precise terms like "cardinality" to denote the number of rows and "degree" for the number of columns, enabling exact predictions of database operations' outcomes. Despite the complexity of the terms, the core principle of an RDBMS remains: it's essentially a well-organized collection of data items, where relations (tables) of various degrees (number of columns) coexist, providing a structured, reliable, and efficient way of managing data [19].

As discussed in the introduction, databases set themselves apart from file-based systems with specific characteristics crucial to their operation and effectiveness. A key aspect is the non-significance of the sequence of rows and columns, reflecting mathematical sets' properties where element order is inconsequential. This aspect is a complete detour from file-based systems, which often rely heavily on record order. While there is a physical sequence to how rows and columns are stored on a disk, this arrangement lacks meaningful impact from an external viewpoint. Furthermore, relational databases maintain atomicity, where each data point in a

row is singular and non-repetitive, ensuring clarity and consistency within each row. No two rows can be complete duplicates, establishing a critical degree of variation within the collected data [19].

Among the various types of RDBMSs, SQLite, MySQL, SQL Server, and PostgreSQL represent three levels of relational database managment systems. SQLite is ideal for smaller projects or applications requiring an embedded database system, as it is a lightweight, file-based database that does not necessitate a separate server. However, its simplicity also means it may not be able to support higher levels of concurrency or complex transactions. MySQL, widely acknowledged for its performance, reliability, and ease of use, supports large-scale web applications, although it compromises full SQL standard compliance for speed. PostgreSQL, offers sophisticated features like table inheritance and function overloading, is also known for its strict standards compliance and scalability similar to MySQL, making it suitable for complex applications requiring advanced data integrity and unique data types. For the development of this project, a CRM web application utilizing phpMyAdmin, MySQL emerges as the preferred choice. The decision is influenced by MySQL's compatibility with phpMyAdmin, an intuitive tool that simplifies MySQL database management, and its widespread use in web applications, making it a suitable foundation for managing customer relationships effectively [25].

Integration of Relational Database Management Systems (RDBMS) into various applications can be determined by its tangible advantages. The Relational database model offers significant advantages in comparison to its counterparts that make them a predominant choice for data storage as well as the preferred choice for this research papers goals. One benefit is that relational databases are simple. RDB's utilize a structure that represents data in tables, a format that is familiar to most users. This tabular structure not only makes data entry, queries, and administration straightforward but also facilitates a clear visualization of data

relationships. Additionally, RDBs exhibit a high degree of data independence. When structural alterations are required, the changes to the existing database do not require substantial modifications to the application they interact with. This feature ensures a smoother transition during updates and system migrations, reducing downtime and resource expenditure. Furthermore, relational databases are robust in terms of security. RDBMS tightly control who can access and alter data to protect against unauthorized actions. Thanks to these key benefits, including strong security and user-friendly interfaces, a RDBMS is recognized as a dependable platform for smoothly handling data storage and management in various scenarios.

However, while RDBMSs offer many great advantages, they are not without their drawbacks, particularly when dealing with complex data sets. One primary concern is the system's likelihood of slowed performance during join operations. For instance operations that require millions of queries will begin to strain a RDBMS due to the intense processing power required. Moreover, RDBMSs are often characterized by their significant memory usage, a major consequence of maintaining its structures, indexes, and relationship mappings that help ensure data integrity and consistency. Another key disadvantage of RDBMSs is scalability; for example, as data volumes swell, RDBMSs typically need more powerful hardware. This can be referred to as vertical scalability while its counterparts use horizontal scalability. A potential disadvantage of RDBMSs is that they require a rigid framework, such as predefined schemas that can be difficult to change once in use. Their structured nature further constrains their effectiveness by making them subpar at managing unstructured or semi-structured data types. However, it's also key to balance these drawbacks by recognizing that relational database management systems have a strong foundation in settings where consistency, reliability, and accurate data integrity are high priority.

Utilizing MySQL for a CRM system is a smart choice because it offers a structured way to store all of the customers information, making it easy to organize and access.

As a relational database, MySQL allows the project to link different pieces of data together, like customer contacts and their enrollment history. As a result the web application will be able to better understand the customers' needs and improve the businesses services [27].

2.1.3 Non-Relational Databases

Non-relational databases, unlike our previously discussed relational databases, do not use tables to store data and they do not need a fixed structure (or "schema") to start collecting information. This means data can be inserted into the database without having to decide its structure in advance. As a result of no predefined schema, you can also repeatedly change the structure of the database and have relatively no cascading negative effects. Similarly, to relational databases, non-relational databases also use unique keys to find and retrieve data. NoSQL databases are widely recognized for their flexibility in handling different domains and applications for unrelated data. As a result of diverging away from standard relational database structure, there exists a variety of alternative models including document-based, key-value, column-oriented, and graph-based structures.

In document-based databases, data management deviates from the conventional tables and instead uses documents, often in JSON, BSON, or XML formats. This approach aligns more closely with the data objects used in applications, reducing the need for extensive translation and enabling faster indexing of queries on individual elements within the documents.

The Key-Value models simplify the database structure to its most basic form, where each item of data is stored as a key paired with its value. This structure which can be described as a two-column table that has the ability to scale and retrieve data rapidly. It also accommodates an array of data types which can range from simple primitives to more complex structures.

Column-oriented databases organize data by columns instead of rows. This orientation is particularly advantageous when analytics need to be performed on a select few columns. This in turn prevents the wasteful consumption of memory taken up by irrelevant data. These databases are also most notable for their scalability, efficient data compression, and speed in data retrieval, making them suitable for managing extensive datasets [16].

Graph-based databases prioritize the interconnections between data points, which are represented as nodes linked by edges or relationships. These databases are particularly adept at visualizing the relationships between different nodes, providing immediate query responses, and maintaining high performance relative to the density of relationships within them. They also simplify data updates, as the introduction of new nodes or edges does not demand substantial changes to the existing database schema. You may be familiar with this type as it is often used to store friendship relations on social media platforms [16]. For example, Facebook utilizes graphs to map the connections between users, such as friendships. Each user is represented as a node, and the relationships between them are the edges, enabling Facebook to quickly navigate and analyze the complex network of connections among billions of users.

2.1.3.1 Non-Relational Database Management Systems

Some of the tools you may consider using for a non-relational database include Apache CouchDB, MongoDB, and Apache Cassandra. Apache CouchDB is a document-oriented database which excels with features like high scalability, data replication, and conflict resolution mechanisms. This makes it the ideal choice for web applications and distributed systems. Its seamless integration with various web technologies and ability to handle concurrency issues are vital for high-availability systems [28].

On the other hand MongoDB is a versatile, distributed database designed for the cloud. As a document database, it stores data in JSON-like formats, offering a more dynamic and flexible approach compared to traditional relational databases. MongoDB stands out for its diverse search capabilities, encompassing geographical, text, and graph searching. It ensures top-notch security measures, including SSL, firewalls, and encryption. [28] Additionally, its compatibility with Business Intelligence tools such as the MySQL protocol allows users to create comprehensive data visualizations solutions.

Conversely, Apache Cassandra is an open-source database known for its exceptional fault-tolerance on cloud infrastructure or local hardware. Its primary strength can be found in its ability to handle node failures without system shutdown and automatic data replication across multiple nodes. Cassandra has a unique feature known as peer-to-peer architecture, which strays away from the traditional master-slave relation, which contributes both to its scalability and fault tolerance. Peer-to-peer architecture implies that each node in the network operates both as a client and a server. As a result, peer-to-peer evenly distributes data among all nodes. This decentralized approach eliminates single points of failure, enhancing the system's overall resilience and performance. This structure allows for uninterrupted addition of new machines to the network and offers a choice between synchronous and asynchronous replication for updates [13]. Despite the rise of NoSQL databases, traditional databases still hold a significant presence [28].

In summary while NoSQL databases offer notable advantages, they also present certain drawbacks. One significant limitation is the constrained functionality of their built-in query language. While these languages are derived from certain SQL functions they typically offer far less versatility. In other words the language can create restrictions which then lead to tasks that require complex queries. In addition, migrating from one NoSQL system to another can pose major challenges due to the

lack of standardization among them. This could potentially lead to increased work, particularly when businesses need to migrate their data infrastructure. The need to develop custom tools for database interaction further adds to the complexity, as most NoSQL databases do not provide comprehensive management tools "out of the box". Finally, the practicality of NoSQL solutions is often tied to the scale of operation; many companies do not handle the volume of data or possess the specific operational prerequisites for which NoSQL databases are optimized for. As a result it typically makes these systems unnecessarily complex when a traditional databases would have sufficed. Hence, while powerful in the right contexts, NoSQL databases are not universally applicable and have obvious trade-offs [8].

2.1.3.2 Object Oriented Database Management Systems

To first understand Object-Oriented Database Management systems (OODBMS), we first must understand the object-oriented paradigm. The object-oriented paradigm is commonly used in computer programming because of its ability to be reused for any given problem. We will use a real-world example to further explain the object-oriented paradigm. For instance, assume that you have helicopter parents (or an overly attached girlfriend, whichever is more appropriate) and you need a way to have alone time for a few hours. Like many overly attached parents or girlfriends, they will become enraged with your disappearance. In your defense, there is no way to get away without causing some disturbance in their lives and you desperately need this time alone. As a result, you created a detailed list of messages you would expect from your parents or girlfriend, as well as a customized response for each message that a computer program will automatically send back to them after a random interval of time. The first challenge you encounter is that you have no reasonable way to interact between the prompts, responses, messages being received, and the random interval to send the messages. You decide to utilize the

object-oriented paradigm as you will be able to utilize objects, classes, inheritance, encapsulation, and polymorphism. The first set of prompts contains all the most common messages you are expecting to receive; “how was your day”, “Did you take out the trash”, and potentially “Have you forgotten about me today”. As a result of the varying prompts, you also wish to store different sentiment values and customized responses based on the messages received. In this hypothetical, let’s conceptualize a class named “Message”, which stores the attribute values of text and sentiment value, and other functionality that enables sending messages, perhaps named send_reply().

A class serves as a template for creating objects, providing a predefined structure that encapsulates data and behaviors. It includes attributes (such as text and sentiment value within our example) to hold data and other methods to perform actions, enabling objects, which are instances of the class, to interact with the encapsulated data in a manner we control. Classes support fundamental OOP principles such as inheritance, where a new class can be derived from an existing one, facilitating a single interface to represent different types of objects. As a result, a class provides a robust framework to model real-world entities in an organized manner. Continuing from the initial scenario we have defined; each instance of the “Message” class will be crafted from the personal messages our unsuspecting third party will provide. These instances represent a singular object, carrying not just the raw text of the expected message but also encompassing behaviors, like crafting a response, and perhaps other attributes like a *response_time*() that will set random intervals to simulate real human interaction.

Additionally, inheritance could be used to further specialize messages based on their urgency or emotional gravity. Complimenting this added functionality subclasses like “UrgentMessage” or “CasualMessage” could be created to override

the initial *send_reply*() method called by our original class "Message". The encapsulation principle comes into play by securing our message attributes, ensuring that the text response, sentiment scores, or reply to mechanisms cannot be altered externally, thereby safeguarding the system's operation. Polymorphism is extremely valuable, especially in a scenario where various message types, each represented by distinct classes inherited from a general "Message" class, need to be managed in a uniform manner. This process permits a generalized interaction with many diverse objects allowing the program to call upon the class methods. This orchestration of OOP principles seamlessly contributes to crafting a system that not only automates interactions but also ensures some level of peace and autonomy away from the crazy parents or girlfriends.

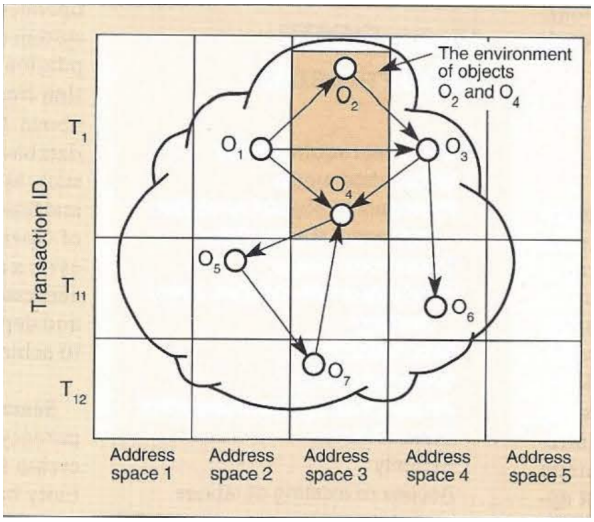

Figure 2.3: Interconnected web of data entities [31]

When leveraging the Object-Oriented Programming (OOP) principles in a Database Management System (DBMS)(Figure 2.3), classes like "Message" could be modeled to instantiate objects representing individual messages, each encapsulating data (such as text and sentiment) and behavior (such as automated replies). A specific instance of a message (object) could be stored, retrieved, or modified in the database, adhering to encapsulation by ensuring data is accessed or mutated using

defined methods. Inheritance allows the creation of specialized message subclasses, enabling the storage and management of diverse message types with additional or overridden behaviors while interacting with them through a generalized interface, demonstrating polymorphism. This ensures a coherent, structured, and unified approach to managing various message instances, thereby allowing the DBMS to organize, retrieve, and manipulate data efficiently and effectively while maintaining logical consistency and structural integrity. Object-oriented database Management Systems (OODBMS) offer several advantages such as providing a seamless, unified representation of data and behavior through objects, enhancing consistency, and eliminating mismatches between the application and data storage models. OODBMS support relationships(Figure 2.4), and are best used for managing complex data types and inter-object dependencies, offering a more natural data modeling approach for specific applications [2].

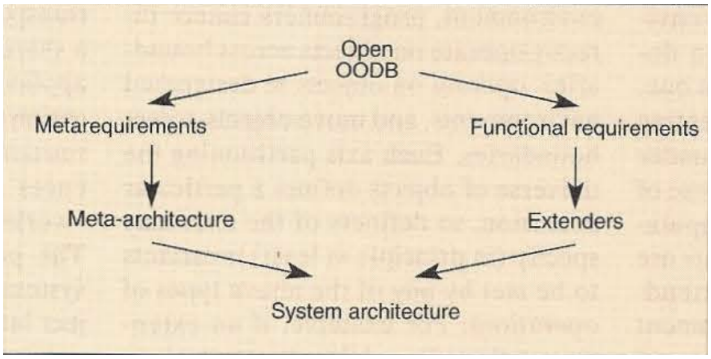

Figure 2.4: Intricate architecture of an Object-Oriented Database [31]

In summary, an Object-Oriented Database Management System (OODBMS) is distinguished by its foundation in object-oriented programming principles within database management. In other words, the DBMS facilitates data to be encapsulated and manipulated as objects, which are instances of classes comprising both data and operations. This model allows for complex data structures and intricate relationships to be managed, benefiting especially complex scenarios where data is intertwined in unique architectures. An additional OODBMS benefit is that it eliminates the

requirement for user-defined keys through the automatic generation of unique object identifiers. This is possible because it maintains a linkage to confirm referenced object integrity and provides a framework that enables a more direct representation of complex objects compared to relational systems. However, these benefits are far outweighed by OODBMS's numerous disadvantages. OODBMS often face security concerns due to insufficient authorization mechanisms and complications and a lack of support in developing object-oriented representation mechanisms. Perhaps the largest flaw of an OODBMS in terms of this research paper's goal is that there is no universal method to retrieve data from the database. As a result of OODBMS having far more cons than pros, the project has decided against the use of object-oriented database management systems. [8]

2.1.3.3 Hierarchical Database Management Systems (HDBMS)

Among the various types of data structures that businesses and organizations have to manage, hierarchical data stands out due to its unique characteristics and the challenges it presents. Hierarchical data is prevalent in scenarios such as organizational structures, content categorizations, document encryptions, and much more. Hierarchical data is characterized by its unique parent-child relational patterns, which establish a tree-like configuration among data points. Each node in this structure represents a data point that can have a single parent (excluding the root node) and multiple children, forming various levels in the hierarchy. Unlike flat or relational data structures, which treat data as independent tables of equivalently structured information, hierarchical models recognize the parent-child relationships within datasets. This data model is what ultimately led to the development of the hierarchical database model. This model is a system where data is organized into a tree-like structure, allowing a single parent to have links to one or many children, but each child having only one parent, thus creating a strict one-to-many relationship.

These models were foundational in applications requiring high-performance data retrieval evident in legacy systems like IBM's Information Management System (IMS). [11]

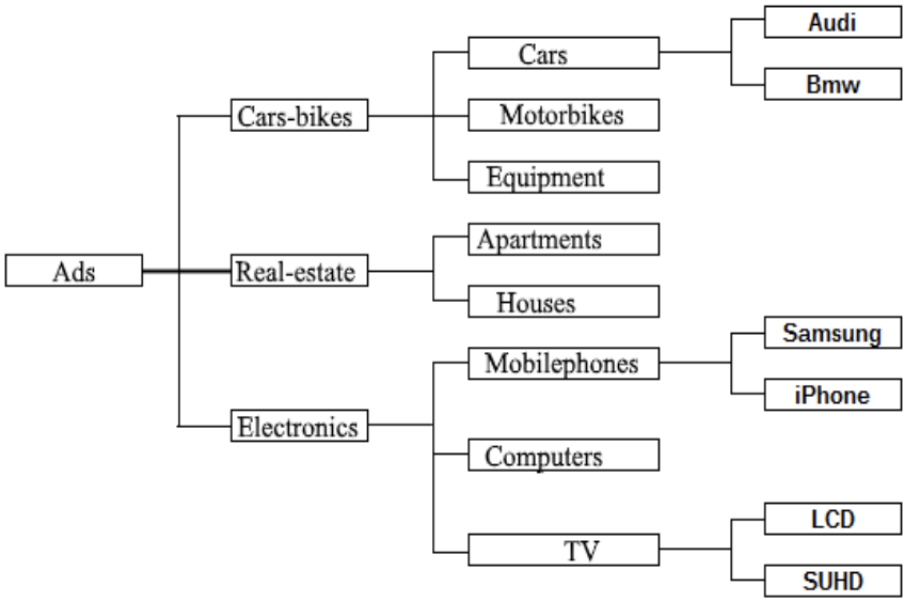

Figure 2.5: Structure of hierarchical data, showcasing parent-child relationships [11]

2.2 Database Design Theory

Database design theory is the logical design and not the physical design of a database. Database design theory is primarily related to normalization otherwise known as the five normal forms: 1st, 2nd, 3rd, 4th, and 5th normal forms. It is important to note that the primary goal of logical design is to create a structure that is independent of hardware, operating systems, DBMS, and specific applications, emphasizing the nature of the data over its usage. Database design theory encompasses a broad array of methodologies and principles aimed at creating efficient and robust database structures. Nevertheless, its distinct normalization stands as a pillar in this discipline, ensuring data integrity and helping eliminate redundancy. Within the scope of this paper, our focus will be limited to the discussion of the 1st to

3rd normal forms, even though there are additional aspects and complexities in database design theory. Subsequently, we will explore the normal forms in their hierarchical sequence, discussing them from the 1st through to the 3rd [7].

2.2.1 1st Normal Form

First Normal Form (1NF) is a fundamental concept in database normalization. It ensures that each attribute in a relation contains atomic, indivisible values. To further define 1NF consider you are given a relation r with attributes A_1 to A_n having types T_1 to T_n respectively, the relation is in 1NF if for every tuple t in r, the value of attribute A_i is of type T_i (where i ranges from 1 to n). In other words, this implies that each tuple in each relation has a single value, conforming to its specified type, for each of its attributes. Notably, 1NF does not add requirements on the types of attributes. As a result, relations containing relation-valued attributes (RVAs) are considered valid, but their design implications must be monitored [7].

Drawing a parallel, when we assert that a database is "normalized," it implies that the database conforms to 1NF. Simply put, being in the first normal form and being normalized are synonymous. This mutual exclusivity means that every relation that is in 1NF is also deemed normalized, and vice versa. This clarification is crucial, as it underscores the foundational importance of 1NF in the broader landscape of database normalization [7].

Before delving into the intricacies of violating 1NF, we must first define the term relational variables as relvars. Think of a relvars as a container holding the present value of a relation, similar to how a mathematical variable encapsulates a specific number. Relvars are the foundation of the relational model, and any discussion about data conforming to a particular normal form is essentially centered on these relvars. Modern databases often represent relvars using what is now colloquially termed as 'tables'. Although tables attempt to mirror relvars, they occasionally fall

short of embodying the the normal forms. Consequently, we must now consider normal form violations within our database design.

Consider a table within a database called 'Employee'. At first glance, it appears to adhere to 1NF. However, after further scrutiny, a deeper examination might reveal duplicate rows. Such redundancy violate 1NF, which asserts the uniqueness of every row within a relvar and the tables that are emulating them. Another breach of 1NF could be concealed columns or system-generated timestamps and unique IDs that are not user-initiated. Such deviations not only diverge from the standard of having consistent columns but also introduce potential inconsistencies [7].

2.2.2 2nd Normal Form

The journey of understanding database normalization continues with the Second Normal Form (2NF). However, before we continue, we must understand functional dependency (FD) as it is vital to understand before defining 2NF. An irreducible FD, represented as $X \rightarrow Y$, is termed irreducible if it stands true in a relational structure and no smaller subset of X can also determine Y. For instance, if we have a functional dependency where both the Supplier Number (SNO) and Part Number (PNO) together determine a Quantity (QTY), this relationship would be considered irreducible if neither SNO and not PNO alone could determine QTY. A relational structure is said to be in 2NF if, for every primary key and every non-key attribute, the functional dependency between them is irreducible. In other words, no part of the primary key can be removed without affecting the existing dependencies [7]. For example, the Second Normal form is breached when a non-key attribute is dependent on a portion of a composite key. It's important to remember that 2NF comes into play specifically when the primary key is composite, in other words; it comprises multiple fields.

Imagine an inventory database capturing details of various parts stored across

multiple warehouses. Consider the database below where both PART and WAREHOUSE act as a conjoined key.

PART	WAREHOUSE	QUANTITY	WAREHOUSE-ADDRESS

Figure 2.6: Database structure that violates second normal form [21]

In this presumed structure, our primary composite key consists of PART and WAREHOUSE fields. However, you may notice, that the WAREHOUSE-ADDRESS relates solely to the WAREHOUSE and does not relate to PART. This structure presents several challenges and heavy disadvantages. Firstly, every unique part stored in a warehouse leads to the repetition of that warehouse's address. As a result, any alteration in a warehouse's address would require updates across all entries of parts associated with that specific warehouse. This built-in redundancy welcomes a breeding ground for data inconsistencies; it's plausible for different records to mistakenly display diverse addresses for an identical warehouse. In addition, if a warehouse temporarily houses no parts, there's the risk of losing the capability to document its address. The most logical solution to these complications would be to reframe the database design by segmenting it into two separate records. As a result of this separation, this database will now adhere to 2NF [21].

PART	WAREHOUSE	QUANTITY

WAREHOUSE	WAREHOUSE-ADDRESS

Figure 2.7: Database structure that satisfies second normal form [21]

To reiterate, the process of restructuring records to align with normalization principles is coined "normalization." While the term can apply broadly, it can also be context-specific. For instance, data might be normalized concerning 2NF but still not meet the criteria of the Third Normal Form (3NF). Embracing the normalized

design undoubtedly elevates data integrity. It curbs redundancy and inconsistency. However, this might come at the expense of retrieval efficiency in certain scenarios. Let's say an application is designed to fetch the addresses of all warehouses housing a specific part. In our initial design, it would simply query one record type. But post-normalization, the application would need to traverse two distinct record types and then correlate the relevant data sets.

2.2.3 3rd Normal Form

Descending deeper into the normalization hierarchy, we encounter the Third Normal Form (3NF). By the preferred definition, a relational variable (Relvar) R is in 3NF if, for every nontrivial functional dependency X → Y in R, either X functions as a superkey or Y is a subkey. This criteria is distinct from 3NF and should not be seen as an extension of 2NF. A common misconception is to view normalization as a linear progression from 1NF to 2NF and so on. While it is true to state that any database in 3NF meets 2NF standards, it is not always true to state that a database in 2NF will adhere to 3NF requirements. Amongst academic circles and database enthusiasts, the debate surrounding the relevance of 3NF is currently ongoing. Some literature portrays 3NF as the true pillar of normalization, viewing it as an endpoint. While others regard 3NF as merely a stepping stone towards the more stringent Boyce-Codd Normal Form (BCNF). A relational variable (Relvar) R is in Boyce/Codd Third Normal Form (3NF) if and only if, for every nontrivial functional dependency X → Y that exists in R, X is a superkey. The Boyce/Codd Third Normal Form (3NF) and the preferred 3NF are both guidelines for organizing data, but they offer slightly different criteria. Boyce/Codd's 3NF has a straightforward requirement that for every relationship between X and Y in a data structure R, and X must always be superkey. This emphasizes that X has unique values and can serve as a distinct identifier to avoid any anomalies in the data table. In contrast, the preferred

3NF has much more flexibility and does not require a super key. It states that for every connection between X and Y in the same data set R, two options can be satisfied. This requirement entails that X either functions as a superkey (similar to the Boyce/Codd's rule), or Y acts as a subkey. By including the provision regarding Y, the preferred 3NF provides an additional avenue to achieve normalization, allowing for scenarios where Y's characteristics might help in reducing data redundancy or other inconsistencies [7]. For example, consider the following table structure where EMPLOYEE is the key while DEPARTMENT and LOCATION are non-key attributes.

Figure 2.8: Database structure that does not satisfy third normal form [21]

Using the preferred definition of 3NF, the database table with EMPLOYEE as the key and both DEPARTMENT and LOCATION as attributes present immediate issues. To be more specific, LOCATION is tied to the DEPARTMENT and is redundantly stated across records of employees belonging to that department. If there's a change in a department's location, it necessitates updates across all employee records, which opens the door to welcome data inconsistencies. For example, distinct records might show varied locations for an identical department due to this design. Furthermore, a department without employees in this structure might be unrepresented, leaving no way to store its location. To normalize this database structure, we can simply rearrange how the data interacts with each other. The EMPLOYEE attribute will remain as key; however, it will only be related to the DEPARTMENT. Now we can use DEPARTMENT as the primary key associated with the LOCATION in perhaps a differing table. The database structure now will adhere to the third normal form [21].

EMPLOYEE	DEPARTMENT

DEPARTMENT	LOCATION

Figure 2.9: Database structure that satisfies third normal form [21]

2.2.4 4th Normal Form

Progressing onward through the normal form hierarchy we reach the Fourth Normal Form (4NF), a progressively more advanced level of normalization that is not commonly discussed as the first three normal forms (1NF, 2NF, and 3NF). This is because the circumstances 4NF addresses are less frequently encountered in practical database design. It deals specifically with Multi-Valued Dependencies (MVDs), which are a generalization of Functional Dependencies (FDs) focused on attributes that are independent of each other but are dependent on a third attribute. A relation is in 4NF if it is in Boyce-Codd normal form (BCNF) and all non-trivial MVDs are also functional dependencies, in other words, they can be reduced to FDs from a superkey. This requirement ensures that the MVDs that do exist in a 4NF relation are fundamental to its structure and cannot be removed through further normalization. Despite its importance in theoretical database design, 4NF is often not discussed in depth for practical database implementations because the first three normal forms effectively handle most data redundancy issues encountered in relational databases. As a result of the relative rarity of MVDs in database practice, we will not elaborate on the 4th normal form any further [7].

2.2.5 5th Normal Form

In the realm of database normalization, the Fifth Normal Form (5NF), is defined by the presence of join dependencies (JDs) and is yet another complex level of database design. As articulated by C.J. Date, 5NF is achieved when every JD in a relvar (relational variable) is implied by candidate keys, facilitating nonloss

decompositions into three or more projections, unlike the decompositions addressed by Heath's Theorem which are limited to two. While this form may be essential for reducing redundancy to its most intrinsic level, it is beyond the necessities of practical database applications used within our CRM web application [7].

CHAPTER 3

SOFTWARE

The software section of this independent study implements a customer relationship manager system for an affordable healthcare insurance agency to manage their agents, leads, and current book of business. The web application is designed to allow company management to navigate lead generation, lead conversion, financial reporting, employee management, and customer service. This chapter focuses on presenting the underlying relational database model that is designed and implemented in the backend of the web application software.

3.1 DOCKER

Docker serves as a tool in emulating a working server environment that can be consistently replicated. By encapsulating the application with its dependencies into containers, developers can ensure that the CRM functions seamlessly across different computing environments and deployments. [9] When the application is ready for deployment, these containers can be easily transferred to any system that has Docker installed, effectively eliminating the "it works on my machine" syndrome. In the development of the Customer Relationship Manager (CRM) project, Docker was utilized to establish a consistent and scalable development environment. Official Docker images for PHP, MySQL, and Apache were orchestrated via Docker Compose to facilitate the integration of these services. 'Docker Compose' is employed to

define and run multi-container Docker applications with the use of a YAML file, conventionally named docker-compose.yml.

```
1  # Docker Version
2  version: '3'
3  services:
4    www:
5      build:
6          context: .
7          dockerfile: Dockerfile.php
8      image: php:apache
9      volumes:
10       - "./:/var/www/html"
11     ports:
12       - 8080:80
13       - 8443:443
14   db:
15     image: mysql:latest
16     command: --log-bin-trust-function-creators=1
17     environment:
18       - MYSQL_DATABASE=${MYSQL_DATABASE}
19       - MYSQL_USER=${MYSQL_USER}
20       - MYSQL_PASSWORD=${MYSQL_PASSWORD}
21       - MYSQL_ROOT_PASSWORD=${MYSQL_ROOT_PASSWORD}
22     volumes:
23       - "./db:/docker-entrypoint-initdb.d"
24   phpmyadmin:
25     image: phpmyadmin
26     ports:
27       - 8001:80
28     environment:
29       - PMA_HOST=db
```

Listing 3.1: Docker-compose.yml file

This file serves as the blueprint for the application's services, defining the configuration of the service environment in a clear and declarative manner. In the Listing 3.1, three services are specified: www for the web server (line 4), db for the database (line 14), and PHPMyAdmin for the database administration interface (line 24). To manipulate the container environment, Docker Compose commands such as 'docker-compose build' are used to build or rebuild services defined in the docker-compose.yml file. This command is particularly useful when changes are made to the Dockerfile or to the service's configuration. Once the building

process is complete, 'docker compose up' can be invoked to start up the entire application, creating and starting containers for each service. To stop and remove the containers, networks, or volumes defined by docker-compose.yml, the command 'docker-compose down' is used. The YAML file defines the build context and references a Dockerfile specifically tailored for the PHP environment (line 7). It also specifies volume mappings to ensure local development changes are synchronized with the container's web directory (line 9-10), and port mappings to expose the services on specified ports (line 11-13). For the database service, environment variables are used to initialize the MySQL database with custom parameters (line 17-21), and volumes are mounted to persist database data (line 22-23). Similarly, PHPMyAdmin is configured to connect to the MySQL service and is made accessible through a mapped port (line 26-27). These configurations orchestrated by Docker Compose streamline the process of setting up a local development environment that closely mirrors a production server, simplifying the transition from development to deployment.

The Dockerfile.php referenced in Listing 3.1 referenced in the docker-compose.yml is pivotal when executing the docker-compose build command as it dictates the custom build steps for the www service. This Dockerfile begins with the base image php:apache (line 8), which is a pre-configured image with PHP and Apache already installed, thus providing a solid foundation for running a PHP-based web application. Upon invoking the 'docker-compose build', the instructions in Dockerfile.php are carried out in a sequence. The RUN instruction is used to install additional PHP extensions like pdo, pdo_mysql, and mysqli, which are essential for the application to interact with the MySQL database and PHPMyAdmin. These extensions facilitate the use of the PDO (PHP Data Objects) and MySQLi (MySQL Improved) to establish a database connection, which is critical for any web application that interacts with a database.

The COPY command that follows after the RUN command transfers the custom.ini file from the local file system into the container's file system, specifically to the directory /usr/local/etc/php/conf.d/, where PHP looks for configuration files. This custom.ini contains settings that override the default PHP configurations, such as those that dictate file upload sizes, execution times, or memory limits, addressing the necessity to handle larger file sizes than the default settings permit. When the docker-compose build command is executed, Docker reads these instructions, builds the image for the www service accordingly, and prepares it to be used by the application. This process ensures that any developer or deployment environment running the application will have the exact PHP environment specified, with all the necessary extensions and configurations, thus avoiding inconsistencies and potential errors.

```
1 FROM php:apache
2
3 # Installs needed extensions to use pdo_connect and mysqli to connect to
    ↪ phpmyadmin
4 RUN docker-php-ext-install pdo pdo_mysql mysqli
5
6 # Copy custom php settings to deal with much larger file sizes
7 COPY custom.ini /usr/local/etc/php/conf.d/
```

Listing 3.2: DockerFile.php file

The custom.ini file found in Listing 3.3 is a configuration file for PHP that overrides certain settings in the PHP environment. The directives specified in this file are critical for the Customer Relationship Manager (CRM) application to function properly, especially when dealing with large amounts of data or long-running processes, which are common when importing or updating substantial datasets in a database. Here's a brief explanation of each directive in the custom.ini file:

- `upload_max_filesize = 32M`: This setting increases the maximum size of an

uploaded file. This is crucial for a CRM that may need to handle the uploading of large documents or data sets related to customer information.

- `post_max_size = 5M`: This sets the maximum size of post data allowed, which is especially important for forms in the CRM that could submit large amounts of data at once, exceeding the default limits.

- `max_execution_time = 300`: This increases the time limit for script execution. Importing data to MySQL can be time-consuming, and the default limit might be too short, leading to timeouts.

- `max_input_time = 300`: This extends the time PHP spends in parsing input data, which is essential for large, complex SQL queries or data uploads.

- `memory_limit = 128M`: This directive raises the memory that a script is allowed to allocate, ensuring that the PHP processes have enough memory to handle large operations without running out of memory, which could otherwise result in a script termination.

```
1 ; custom.ini
2 upload_max_filesize = 32M
3 post_max_size = 30M
4 max_execution_time = 300
5 max_input_time = 300
6 memory_limit = 256M
```

Listing 3.3: custom.ini file

These settings in the custom.ini file are necessary to accommodate the resource demands of CRM operations, such as batch processing large numbers of records, handling file uploads, and executing heavy SQL queries during data import/export routines. Without these custom settings, the default PHP environment might not support the operational requirements of the CRM, leading to potential failures in data processing and impacting the overall reliability of the application. Integrating

this file into the Docker container with the application code ensures that these settings are applied consistently in every environment where the container is deployed.

An '.env' file Listing 3.4, standing for "environment file," is a standard practice within modern application development, particularly for managing sensitive or environment-specific configurations. The .env file is instrumental in enhancing security as it provides a secure place to store sensitive information like MYSQL_PASSWORD, MYSQL_ROOT_PASSWORD, GMAIL_USER, and GMAIL-_PASSWORD, which are stored separately from the codebase. For obvious reasons, this practice is vital to maintain security, as including such details directly in the application can lead to significant security risks. Additionally, the environment variables such as MYSQL_HOST, MYSQL_USER, and MYSQL_DATABASE enable the application to be configured differently across various environments (like development, testing, and production) without needing any modification to the code. This separation aligns with best practices in software development, enhancing maintainability and scalability. When moving the application or sharing it with other developers, the .env file can also be easily adjusted to suit new setups. In the Dockerized environment of the CRM, these variables from the .env file are used to establish a general configuration. For example, 'Docker Compose' can interpolate the values, allowing for a dynamic configuration of the services. The MYSQL_* variables are crucial for setting up the MySQL service within the container, and the GMAIL_* variables are necessary for the email functionalities of the CRM. It is crucial to remember that .env files containing sensitive data should not be included in version control systems. Since this application is being version-controlled using GitHub we explicitly state to ignore the .env file within a .gitignore file. As a result, a template file (like .env.example) with placeholder values is shared through our version control, guiding other developers to set up their own .env file accordingly.

```
1 MYSQL_HOST = MYSQL_HOST
2 MYSQL_USER = MYSQL_USER
3 MYSQL_PASSWORD = MYSQL_PASSWORD
4 MYSQL_ROOT_PASSWORD = MYSQL_ROOT_PASSWORD
5 MYSQL_DATABASE = MYSQL_DATABASE
6
7 GMAIL_USER = GMAIL_USER
8 GMAIL_PASSWORD = GMAIL_PASSWORD
```

Listing 3.4: .env file

The backup MySQL file referenced in the Docker setup for the Customer Relationship Manager (CRM) project is a comprehensive SQL dump, primarily generated through phpMyAdmin, as indicated by the comments in the file. This backup file encapsulates the entire state of the database at the time of its generation, which includes not only the structure of the database tables but also the data contained within them.

In the Dockerized environment of the CRM project, when 'docker compose up' is executed, this backup file is referenced to initialize the MySQL container with the existing data and structure. This process ensures that the development, testing, or production environment of the CRM has a ready-to-use database setup that mirrors the state captured in the backup, providing consistency and reliability in the application's deployment and development lifecycle.

3.2 User Interface

User Interface design or UI for short encompasses the design and layout through which users interact with the application. It's a critical element that bridges the gap between the user and the system's functionality, making it both accessible and efficient. The design and evaluation of a user interface require not just understanding but also experience. Unlike following a traditional cooking recipe, UI design rules often outline goals instead of specific actions and are deliberately generalized

for broad applicability. However, this generalization leads to open interpretation and introduces challenges in specific design contexts. One of the complexities in UI design is the frequent occurrence of conflicting guidelines. A single design situation might be addressed by multiple applicable rules, often suggesting different approaches. As a result, UI designers must make thoughtful decisions on which rule should take precedence. Some design problems in UI often involve balancing conflicting goals, such as:

- Balancing a bright screen with long battery life
- Achieving a design that is both lightweight and sturdy
- Creating an interface that is easy to learn and understand.
- Ensuring the system is powerful yet simple
- Striking a balance between high resolution and fast loading

Addressing these conflicting goals usually involves numerous trade-offs, requiring a balanced approach between the competing design rules. In summary, user-interface design guidelines are primarily built around human psychology. Understanding these guidelines requires not just knowledge of the rules but an understanding of the psychological principles behind them. [18]

We have finally arrived to StarPoint's primary page where users will first navigate. As a user navigates to the index.php page, they will find themselves navigating through a interface designed to facilitate their health plan enrollment process. This website focuses on delivering a personalized and intuitive user experience that makes it easy for anyone to use.

At the top of the page, the navigation bar provides quick access to various sections within the index.php page. The navbar acts as a central hub for users, allowing them to traverse different sections of the portal without the need to load

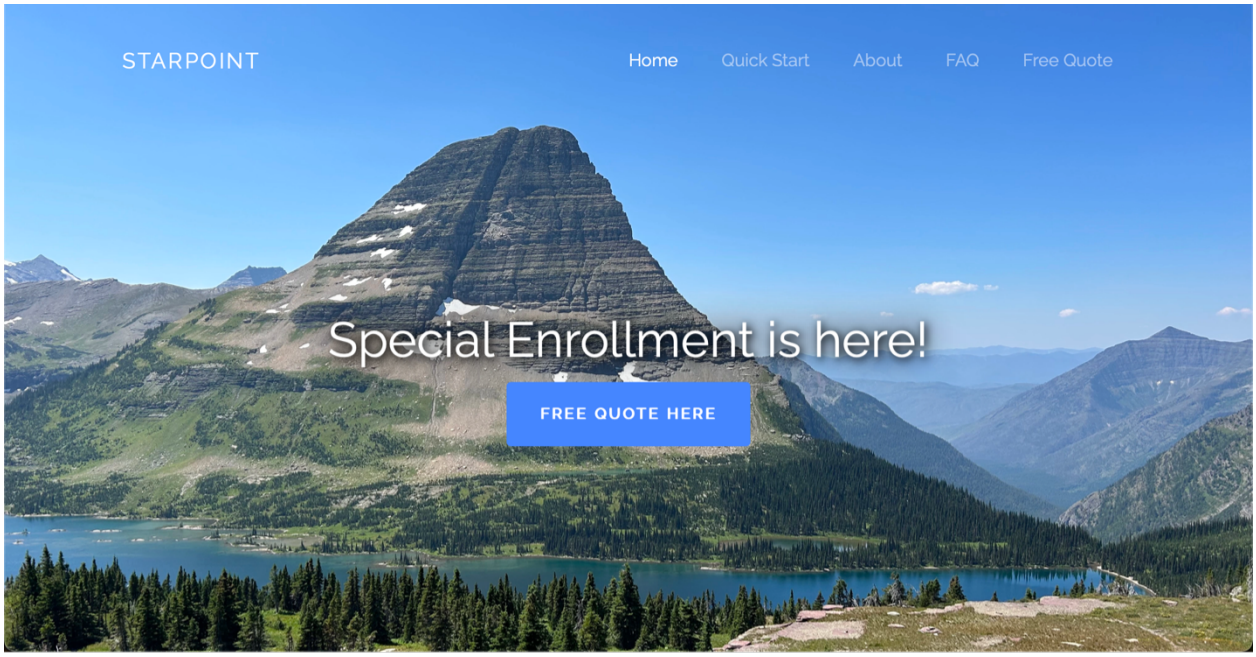

Figure 3.1: User Landing Page for StarPoint

new pages. The use of in-page navigation links in the navbar is a key feature. When a user selects an option such as "Home," "Quick Start," "About," "FAQ," or "Free Quote," they are directed to the respective section within the same page. This single-page layout eliminates the need for traditional page reloads, fostering a fluid and uninterrupted user experience. In other words, it acts as a comprehensive guide, where all necessary information is available at a glance and just a scroll away.

Similarly, the page is built to contain adaptive features built around the navigation bar, particularly when accessed from devices with smaller screens, such as smartphones or tablets (Figures 3.2 and 3.3). In these instances, the navigation bar collapses into a 'hamburger' menu icon – a universally recognized symbol for a menu in modern web design. This compact, icon-based menu ensures that the user interface remains uncluttered and accessible, even if accessed via smaller screen sizes. When a user taps or clicks on this hamburger icon, it expands to reveal the full range of navigation options, just like those seen in a standard view on larger screens. despite the condensed view.

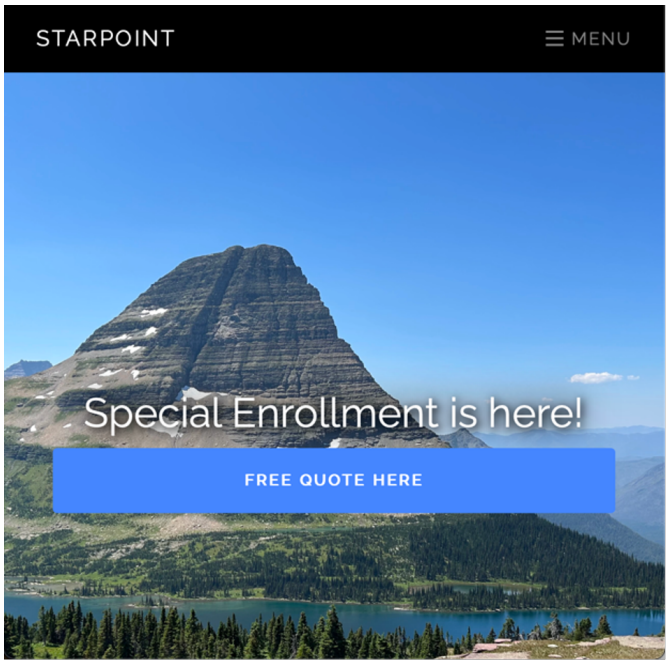

Figure 3.2: User Landing Page on a tablet

Figure 3.3: User Landing Page navigation on a mobile phone

As a user continues beyond the displayed hero image and further into the index.php page they encounter the 2nd section of the page, “Quick Start”.

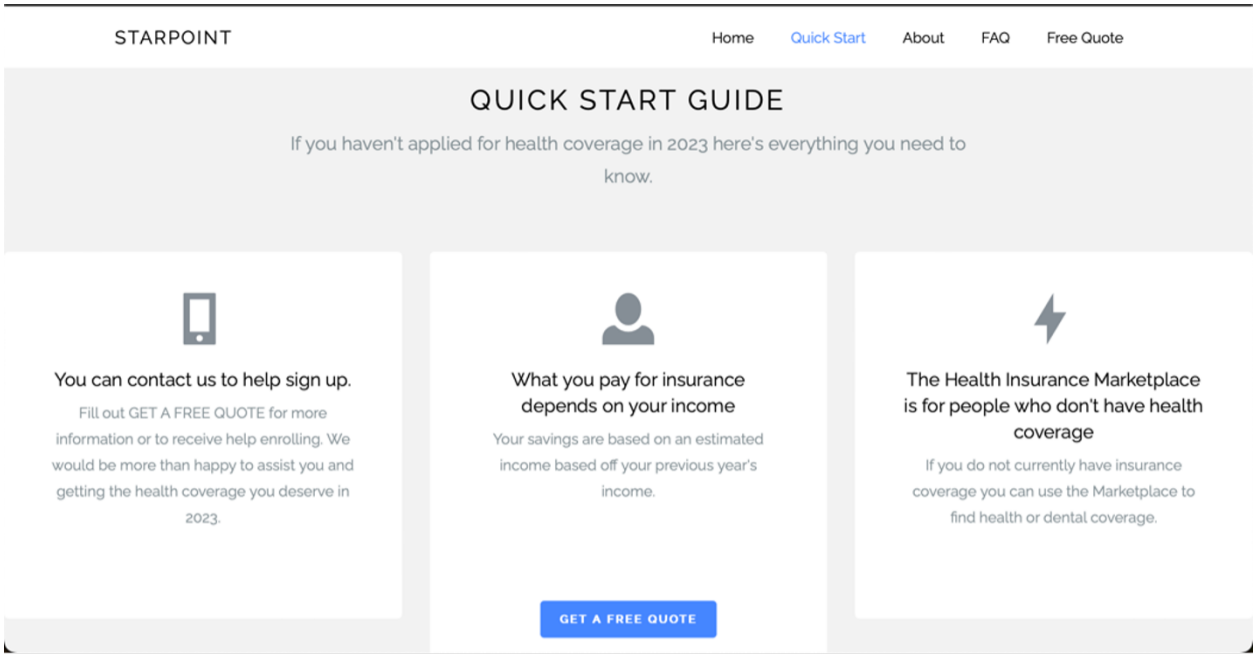

Figure 3.4: Quick start guide section

The use of HTML sections in the design enhances the user’s experience. By allowing a user to cascade through the pages sections they can acclimate to the pages design at their own pace. At the same time, if a user were to revisit the site, they can use the navigation bar to jump to the exact section they wish to encounter. This sectional approach not only organizes content in a user-friendly manner but also aids in maintaining a clean and navigable interface.

This design philosophy aligns well with modern web browsing habits. During a time where most users access websites through their mobile devices, a single-page layout with sections offers a seamless scrolling experience. This fluidity within the design allows for a scrolling behavior prevalent among mobile users. [18] You may use apps that have this feature already such as Instagram and TikTok. In addition, we are predefining the way a user must use the webpage. Restricting the actions a user can perform enables the page to illustrate the action we would like them

to take. In this instance, the design leads the user to navigate to the "Get A Free Quote" section. If they choose not to navigate to this section but rather continue to scroll and learn more about the page, they inevitably will eventually reach the same section.

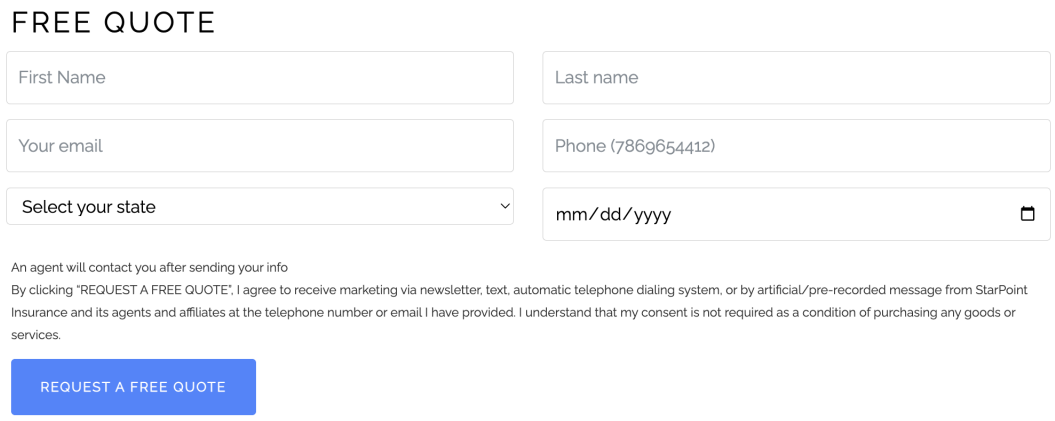

Figure 3.5: Contact form

The contact form (Figure 3.5), positioned as the concluding section of the index page, represents the climax of the user's path through the page. It's not just a component of the page; it is almost the entire purpose of the landing page. This placement is deliberate and designed to convert a visitor's growing interest into actionable engagement. After navigating through the information found within the varying sections the visitor meets the contact form, functioning as the index pages call to action. In actuality, the contact form's design and placement at the end of the page serve two purposes. It not only captures leads but also acts as a metric for gauging the effectiveness of the displayed content. A high rate of form submissions indicates that the preceding sections have successfully persuaded the user. Conversely, a lower submission rate can signal the need for content optimization. Once a user submits the form, they will be automatically contacted at the email they entered. As a potential customer, or in the case of a healthcare

agency as a potential member, their experience will continue via contact through email or other contact form from a representative of the agency.

3.3 PHP and JavaScript

For the remainder of this study, we will continue displaying each view page but only talk about the functionality of the backend rather than the front-end user interface. For reference, the image below is a diagram of which pages exist and how a user can navigate between them.

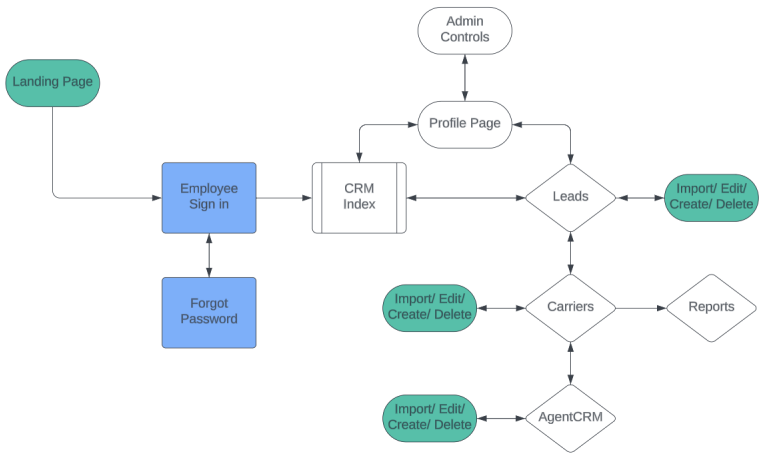

Figure 3.6: StarPoint flowchart

3.3.1 Landing Page Functionality

Beyond the user interface of the StarPoint landing page, there is only one section of the webapp where a user can submit a form. In other words, the "free quote" section of the page is the only part of the page that contains backend functionality. It exists as the main call to action; we hope that the page converts the page visitor into an actionable member. The contact form structure is as follows.

```
357 <form action="../php/new-lead.php" method="post">
358     <div class="form-row">
359         <div class="form-group form-group-inline">
```

```
360             <label for="first_name" class="sr-only">First Name</label>
361             <input type="text" class="form-control" id="first_name"
362                    name="first_name" placeholder="First Name"
363                    autocomplete="given-name" required maxlength="50"
364                    pattern="[A-Za-z\s]+">
365         </div>
366         <div class="form-group form-group-inline">
367             <label for="last_name" class="sr-only">Last Name</label>
368             <input type="text" class="form-control" id="last_name"
369                    name="last_name" placeholder="Last name"
370                    autocomplete="family-name" required maxlength="50"
371                    pattern="[A-Za-z\s]+">
372         </div>
373     </div>
374 ...
```

Listing 3.5: Form action of the StarPoint Landing Page

As you can see the form action is 'new-lead.php' which is a file that serves to process a user's submission to get a free quote by processing the information into the database. All the fields below our form are input elements that are sent as a post to the 'new-lead.php' file. Before the file is submitted there are javascript check conditions to make sure each field is filled out as required.

```
53  // Check if there are any errors
54  if (count($errors) == 0) {
55      $pdo = pdo_connect_mysql();
56      $stmt = $pdo->prepare('INSERT INTO Leads (first_name, last_name, email,
            ↪ phone, state, DOB) VALUES (?, ?, ?, ?, ?, ?)');
57
58      if ($stmt->execute([ $_POST['first_name'], $_POST['last_name'], $_POST['
            ↪ email'], $phone_number, $_POST['state'], $date_of_birth ])) {
59          $_SESSION['message'] = 'Thank you, StarPoint will contact you shortly
                ↪ ';
60
61          $mail = new PHPMailer(true);
62          $mail->isSMTP();
63          $mail->Host = 'smtp.gmail.com';
64          $mail->SMTPAuth = true;
65          $mail->Username = $config['GMAIL_USER'];
66          $mail->Password = $config['GMAIL_PASSWORD'];
67          $mail->SMTPSecure = 'ssl';
68          $mail->Port = 465;
69
70          $mail->setFrom('StarPoint.Insurance@gmail.com');
71          $mail->addAddress($_POST['email']);
72          $mail->isHTML(true);
73          $mail->Subject = 'Acknowledgement of Inquiry - StarPoint Insurance';
```

```
74          $mail->Body = 'Hello ' . $_POST['first_name'] . ',<br><br>Thank you
                ↪ for reaching out to StarPoint Insurance.
75          Your inquiry is important to us. We are currently reviewing your
                ↪ request and will ensure a response is provided
76          within 2-3 business days. Should you have any immediate concerns or
                ↪ additional queries, please feel free to contact
77          our support team at StarPoint.Insurance@gmail.com.<br><br>Best
                ↪ Regards,<br>The StarPoint Insurance Team';
78          $mail->send();
79
80      } else {
81          $_SESSION['message'] = 'There was an error with your information';
82      }
83  } else {
84      $_SESSION['errors'] = $errors;
85  }
86  header('Location: ../view/index.php#section-contact');
87  exit();
```

Listing 3.6: new_lead.php file

The 'new-lead.php' file in the provided PHP code segment (Listing 3.6) begins with initializing a session, a common practice in PHP to maintain user state and data across web pages. This is done through the session_start() function, which typically starts a new session or resumes an existing one. Following this, the script includes the 'functions.php' file, a critical inclusion that allows 'new-lead.php' to access essential functions, such as mysql_pdo_connect(), which is used later for establishing a connection between the database and the web page (line 55).

After this, the script integrates PHPMailer, a popular library for handling email-sending functions in PHP. This integration is achieved using the "use" statement for the PHPMailer and Exception classes. The script specifically includes the PHPMailer, Exception, and SMTP classes from the PHPMailer library, located in the 'PHPMailer' directory, relative to the current script path. These classes are essential for enabling the script to send emails, handle exceptions, and use SMTP for email transport. Additionally, the script prepares for environment-specific configurations by loading a '.env' file. It accomplishes this by constructing the path to the '.env' file relative to the current directory and then parsing this file using parse_ini_file(). This process

allows the script to access configuration settings, like our email password, stored in the '.env' file, which is a common practice for managing sensitive information in a secure and flexible manner. With these configurations and libraries in place, the execution then proceeds with its core functionalities of form data validation and processing.

As discussed previously, the core functionality of 'new-lead.php' is to perform extensive validation checks on the user's form data to insert into our Leads data table. The file first checks if all the required fields like first name, last name, email, phone number, state, and date of birth are filled in by examining the $_POST data. Any missing or improperly filled fields prompt the script to store an error message in an array named $errors.

Specifically, for the email field, the script employs PHP's filter_var function with the FILTER_VALIDATE_EMAIL filter to verify the email format. If the email does not meet the required standard, an error message is appended to the $errors array. In the case of first and last names, the script uses a regular expression to ensure that these fields contain only letters, spaces, hyphens, and apostrophes. Non-conforming names result in an error message being recorded.

Additionally, the script processes the phone number to remove any non-numeric characters and then verifies if the remaining string is exactly 10 digits long. Deviations from this expected format will lead to adding an error message in the $errors array (lines 81 and 84).

Another critical validation performed by the script is for the state field. The script contains a predefined array of valid US state abbreviations and checks if the submitted state is included in this array. If an invalid state is selected, an error message is added to the $errors array. The last validation on date of birth begins with formatting to ensure it is not set in the future, using PHP's DateTime class for

this purpose. Invalid dates of birth are flagged with an error message added to the $errors array.

Upon successful validation of the form data in the 'new-lead.php' file, the script begins to process the user into the database, handling the insertion of data into a database and sending an email confirmation. The script checks if there are no errors collected in the $errors array. If this array is empty, indicating that all validations have passed, the script moves forward with database operations. It establishes a connection to the MySQL database using the pdo_connect_mysql() function, defined in the included 'functions.php' file. This function returns a PDO (PHP Data Objects) instance, which is a secure way to interact with the database.

The script then prepares an SQL statement to insert the validated form data into the 'Leads' table of the database. The prepare method of the PDO instance is used to prepare the SQL statement, which is another secure way to handle database queries, helping to prevent SQL injection attacks. The data from the form (first name, last name, email, phone number, state, and date of birth) is bound to the prepared statement and executed. The phone number and date of birth have already been processed and formatted appropriately in the previous steps of the script.

If the execution of the database insertion is successful, the script sets a success message in the session, informing the user that StarPoint will contact them shortly. It then proceeds to send an email to the user, confirming their inquiry. For this purpose, the script creates an instance of the PHPMailer class and configures it with the necessary SMTP settings, including host, SMTP authentication, username, and password, which are retrieved from the configuration file. The email is set to be sent from 'StarPoint.Insurance@gmail.com', and the user's email address is added as the recipient. The subject and body of the email are set appropriately, with the body containing a personalized message and contact information for further inquiries.

In the case where the database insertion fails, a different message is set in the

Figure 3.7: Automated email response

session, indicating an error with the user's information. If the $errors array is not empty, meaning there were validation errors, the errors are stored in the session for later display to the user.

The final action the script takes is to redirect the user to the 'index.php' page and then exits to prevent further script execution. This redirection is a common practice to avoid form resubmission issues and to guide the user flow within the application. If we did not redirect the user, they would be stuck on the new-lead.php location and there is no UI for them to interact with at this location. The redirect allows them to return to the environment they are familiar with.

Beyond the contact section of the index page is the last section known as a footer. The footer in this case does not have additional information typically offered by traditional pages. Such as career openings with the agency, additional information, additional contact forms, and other details. In our case, the footer of the index page simply serves as a method for employees of the agency to log in. The lack of additional hyperlinks comes as a result of focusing on creating the CRM portion of the webpage rather than creating a page that is appealing to customers.

3.3.2 Employee Login

If an employee has not been registered previously by the agency for access to the CRM they will either not have access or must request the website's manager to grant them privileges. If a user who is not an employee with the company clicks the employee login page, they will be unable to log in since they also do not have credentials.

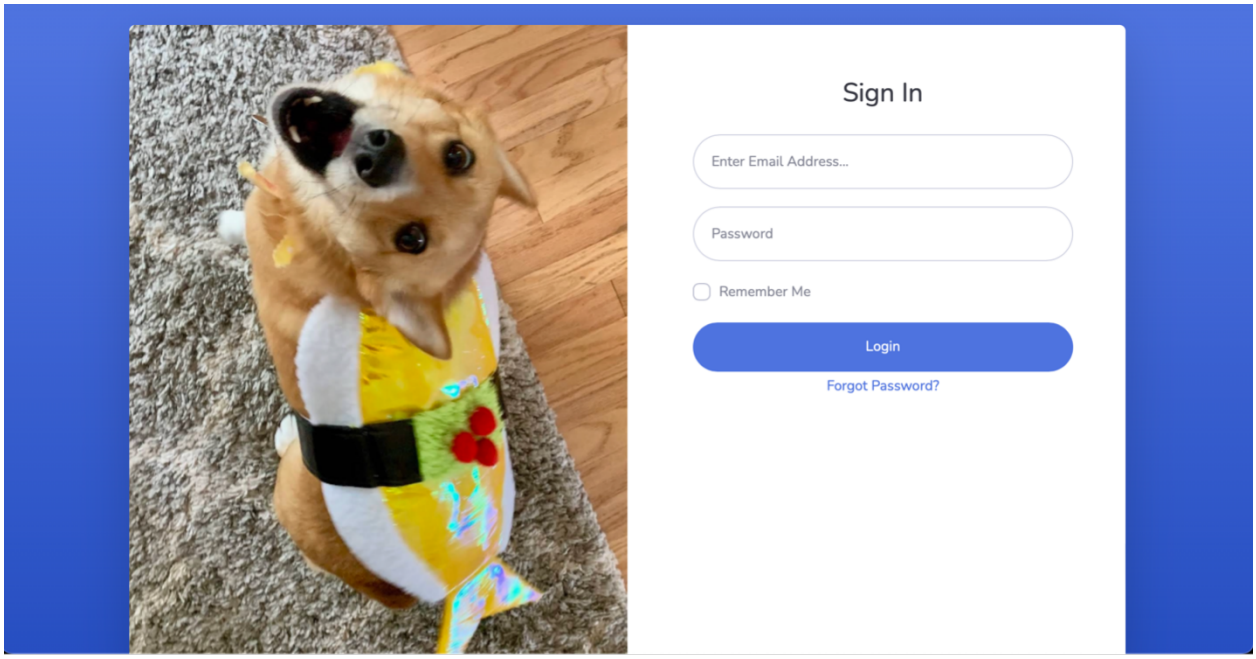

Figure 3.8: Login page for employees

The login page form includes the necessary fields to authenticate a user; the users email and password. When a user submits this form its method is a post request that is processed by the authenticate.php file. This method ensures that we do not create new records in the database since we do not allow anyone to register for access other than by the administrator.

When a user completes the form and the post data has been sent, the first thing the authenticate.php file does is to validate the information submitted by the user.

```
5  if (!isset($_POST['email'], $_POST['password'])) {
6      $_SESSION['error'] = 'Please fill both the email and password fields!';
```

```
7       header('Location: ../login.php');
8       exit();
9   }
```

Listing 3.7: Password validation

The first condition statement checks to see if a user submitted any information within the email and passwords fields. If a user has not placed any information within either field on the page, they will be redirected back to the login page. If a user passes this condition, we will move onto the next condition which will begin to authenticate a user.

```
12  $pdo = pdo_connect_mysql();
13  $stmt = $pdo->prepare('SELECT * FROM Employee WHERE email = ?');
14  $stmt->execute([ $_POST['email'] ]);
15  $account = $stmt->fetch(PDO::FETCH_ASSOC);
16
17  if ($account) {
18      if (password_verify($_POST['password'], $account['password'])) {
19          if ($account['activation_code'] != '1') {
20              $_SESSION['error'] = 'Your account has not been verified by an
                    ↪ admin yet. Please wait for verification.';
21              header('Location: ../view/login.php');
22              exit();
23          } else {
24              session_regenerate_id();
25              $_SESSION['loggedin'] = TRUE;
26              $_SESSION['name'] = $account['username'];
27              $_SESSION['id'] = $account['id'];
28              $_SESSION['role'] = $account['role'];
29
30              if (isset($_POST['rememberme'])) {
31                  $cookiehash = !empty($account['rememberme']) ? $account['
                        ↪ rememberme'] : password_hash($account['id'] . $account
                        ↪ ['email'] . 'yoursecretkey', PASSWORD_DEFAULT);
32                  $days = 30;
33                  setcookie('rememberme', $cookiehash, (int)(time()+60*60*24*
                        ↪ $days));
34                  $stmt = $pdo->prepare('UPDATE Employee SET rememberme = ?
                        ↪ WHERE id = ?');
35                  $stmt->execute([ $cookiehash, $account['id'] ]);
36              }
37
38              $date = date('Y-m-d\TH:i:s');
39              $stmt = $pdo->prepare('UPDATE Employee SET last_seen = ? WHERE id
                    ↪  = ?');
40              $stmt->execute([ $date, $account['id'] ]);
41
```

```
            $_SESSION['success'] = 'Login successful!';
            header('Location: ../view/crm/index.php');
            exit();
        }
    } else {
        $_SESSION['error'] = 'Incorrect email and/or password!';
        header('Location: ../view/login.php');
        exit();
    }
} else {
    $_SESSION['error'] = 'Incorrect email and/or password!';
    header('Location: ../view/login.php');
    exit();
}
```

Listing 3.8: Employee authentication and session management

In the authentication process of our web application, the script begins by establishing a connection to the MySQL database using a variable $pdo. This connection is crucial as it facilitates all interactions with the database. The pdo_connect_mysql() function, which is essential for this purpose, is included in a file named "functions.php". This file, referenced at the beginning of our "authenticate.php" script, is a repository of commonly used functions throughout the web application. It plays a vital role in maintaining code efficiency and reusability.

```
function pdo_connect_mysql() {
    // Read from .env file
    $envPath = __DIR__ . '/../../.env';
    if (!file_exists($envPath)) {
        $envPath = __DIR__ . '/../.env';
    }
    $dbconfig = parse_ini_file($envPath);

    $DATABASE_HOST = isset($dbconfig["MYSQL_HOST"]) ? $dbconfig["MYSQL_HOST
        ↪ "] : 'db';
    $DATABASE_USER = $dbconfig["MYSQL_USER"];
    $DATABASE_PASS = $dbconfig["MYSQL_PASSWORD"];
    $DATABASE_NAME = $dbconfig["MYSQL_DATABASE"];

    try {
        $pdo = new PDO('mysql:host=' . $DATABASE_HOST . ';dbname=' .
            ↪ $DATABASE_NAME . ';charset=utf8', $DATABASE_USER,
            ↪ $DATABASE_PASS);
        $pdo->setAttribute(PDO::ATTR_ERRMODE, PDO::ERRMODE_EXCEPTION);
        return $pdo;
    } catch (PDOException $exception) {
```

```
19          exit('Database connection error: ' . $exception->getMessage());
20      }
21  }
```

Listing 3.9: PDO Connection

The function begins by attempting to locate a configuration file, '.env', which is contains essential database connection parameters (lines 2-7)(Listing 3.9). (Please reference the docker section for further information regarding the '.env' configuration) The function first looks for this file two levels up from the current directory, as indicated by $envPath = __DIR__ . '/../../.env';, where __DIR__ is a special constant in PHP that provides the directory of the current script. If this initial search fails, indicated by the if (!file_exists($envPath)) check, it modifies the path to search one level up. Once located, the function uses parse_ini_file to read the '.env' file, extracting database configurations into the $dbconfig array. The webapp has this added functionality because once a user signs in they are routed to a folder a step further away from where the functions.php file lives. As a result, we must check to see if we must only step out once or twice to find our functions file.

Following this, the function sets up several variables for the database connection, utilizing the $dbconfig array (line 7). It determines the database host, username, password, and database name, employing a default value for the host if not specified in the configuration. Most notably, the function defines a global constant WEBSITE_NAME with a value of 'StarPoint', although this is not directly related to the database connection process.

The core of the function is the establishment of the PDO connection within a try-catch block, which is a standard error-handling practice in PHP. It attempts to create a new PDO object with the specified database host, name, charset, user, and password. The PDO error mode is set to exception (PDO::ERRMODE_EXCEPTION), which allows the developers to handle any errors during database interactions. If the PDO connection is successful, the function returns this object, enabling further

database operations. However, in the event of a connection failure, the catch block captures the PDOException, and the function terminates by displaying the error message. This response is crucial for troubleshooting database connection issues in PHP, which also notifies the user of any issues encountered during the connection.

Once the web app has established a connection to the MySQL database and stores its value in the $pdo variable it can continue the authentication process. The script first checks if the $account variable, which holds the user data fetched from the database, is not empty. This is a crucial step to ascertain whether the user with the submitted email exists in the database. If $account is false (meaning no user was found), the script sets an error message about incorrect credentials and redirects to the login page.

The next condition statement checks if the user's password matches the password found within the database. The authenticate.php file uses a password_verify() to check if the submitted password matches the hashed password as database passwords are not stored as plain text. This function is vital for security, as it correctly handles password hashing and comparison, ensuring the protection of user credentials. The authentication further checks if the user's account has been activated; check against the 'activation_code'. If the account is not activated, an error message is set, and the user is redirected back to the login page. This step is critical in ensuring that only verified users can log in. Upon successful password verification and account activation, the script regenerates the session ID for security reasons and sets various session variables such as loggedin, name, id, and role. This action effectively logs the user in and personalizes their session. The user is then redirected to a specific page within the application, indicating a successful login.

Their also exists an optional conditional block which handles the 'Remember Me' functionality. If the user chooses to be remembered, the script sets a cookie with a unique hash. This feature enhances user experience by allowing them to stay

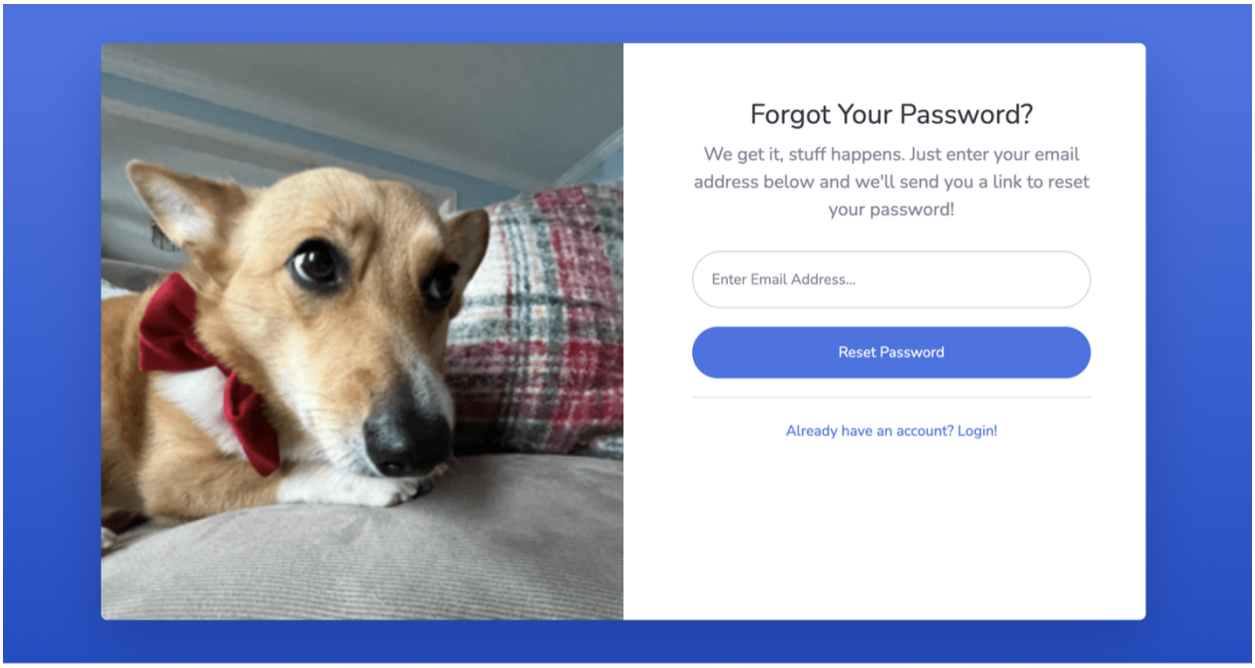

Figure 3.9: Forgot Password Page

logged in for a specified period (30 days in this case). Here, the script updates the 'last_seen' timestamp in the database for the logged-in user, which can be useful for tracking user activity and engagement within the application.

Upon successful authentication the user will be granted access to the customer relationship manager page and can freely view customers data. The user will also be redirected to the CRM index page which serves as a landing page for the dashboard. If they are unable to login successfully the user will stay on the login.php page and be prompted that their email or password was incorrect.

3.3.3 Employee Resetting Password/ Forgot Password

When a user attempts to log in to the CRM application and forgets their password they can navigate to the following page (forgot-password.php).

The "forgot-password.php" file in Listing 3.10 allows users to type in their email which they use to authenticate into the platform. When the script detects a submission of the password reset request, the file first ensures that the email field is

populated. It then queries the database for a user with the provided email address. If the user is found, the script generates a unique token using PHP's 'random_bytes' function. This token, along with the current timestamp, is updated in the user's record in the database. This step is for securing the password reset process, as it ensures that the reset link sent to the user is unique and completed in a timely manner (lines 37-40).

```
23 if ($_SERVER["REQUEST_METHOD"] == "POST") {
24     // Check if the email field is set and not empty
25     if (isset($_POST['InputEmail']) && !empty($_POST['InputEmail'])) {
26         $email = $_POST['InputEmail'];
27         $pdo->beginTransaction();
28         try {
29             // Fetch user data from the database using email
30             $stmt = $pdo->prepare('SELECT id, email FROM Employee WHERE email
                   ↪  = ?');
31             $stmt->execute([$email]);
32             $employee = $stmt->fetch(PDO::FETCH_ASSOC);
33             if ($employee) {
34                 // Generate a unique token
35                 $token = bin2hex(random_bytes(16));
36                 // Get the current timestamp
37                 $currentTimestamp = date("Y-m-d H:i:s");
38                 // Store the token and the current timestamp in the database
39                 $stmt = $pdo->prepare('UPDATE Employee SET
                       ↪ forgot_password_token = ?, last_seen = ? WHERE id = ?')
                       ↪ ;
40                 $stmt->execute([$token, $currentTimestamp, $employee['id']]);
41                 // Commit the transaction
42                 $pdo->commit();
43                 // Send email with new password
44                 ...
```

Listing 3.10: Forgot Password PHP

The sending of the password reset email is handled by PHPMailer with SMTP settings to send an email through a Gmail account. The email contains a link that includes the generated token, allowing the user to reset their password securely. This method of sending a personalized password reset link via email is a standard practice in web security, ensuring that only the intended recipient can access the link to change their password (lines 49+).

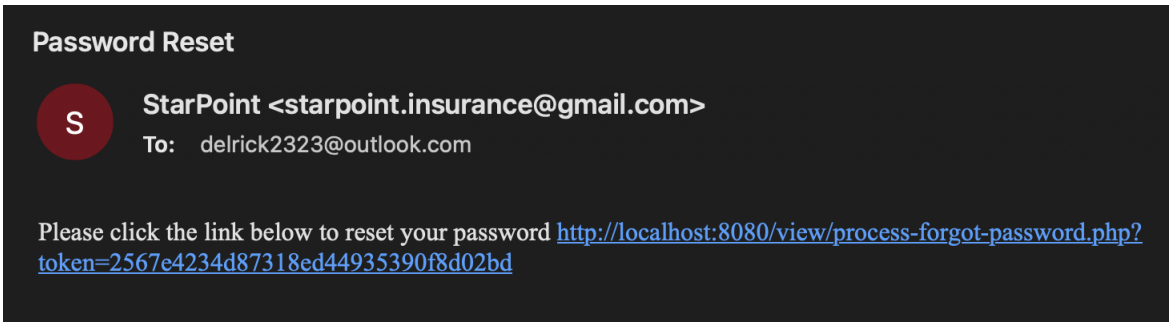

Figure 3.10: Forgot Password Emailed Link

Once the user receives the email, they will click the link to reset their password and be routed to the "process-forgot-password.php" (Listing 3.11). This file begins processing once the user clicks the link, which includes a unique token as a query parameter. The script's primary functions are to validate this token is valid in the database, ensure the request is completed within a certain period, and securely update the user's password.

```
17 ...
18 // Check if the token is present in the URL
19 if (!isset($_GET['token'])) {
20     die('No token provided');
21 }
22
23 $token = $_GET['token'];
24
25 // Connect to MySQL database
26 $pdo = pdo_connect_mysql();
27
28 // Fetch user data from the database using the token
29 $stmt = $pdo->prepare('SELECT id, forgot_password_token, last_seen FROM
       ↪ Employee WHERE forgot_password_token = ?');
30 $stmt->execute([$token]);
31 $employee = $stmt->fetch(PDO::FETCH_ASSOC);
32
33 if (!$employee) {
34     die('Invalid token');
35 }
36
37 // Check if the token is still valid (within the last 15 minutes)
38 $fifteenMinutesAgo = date("Y-m-d H:i:s", strtotime("-15 minutes"));
39 if ($employee['last_seen'] < $fifteenMinutesAgo) {
40     die('Token expired');
41 }
42
43 if ($_SERVER["REQUEST_METHOD"] == "POST") {
```

```
44      // Check if the new password and confirm password fields are set and not
            ↪  empty
45      if (isset($_POST['NewPassword']) && !empty($_POST['NewPassword']) &&
46          isset($_POST['ConfirmPassword']) && !empty($_POST['ConfirmPassword'])
                ↪ ) {
47
48          $newPassword = $_POST['NewPassword'];
49          $confirmPassword = $_POST['ConfirmPassword'];
50
51          if ($newPassword !== $confirmPassword) {
52              die('Passwords do not match');
53          }
54
55          // Hash the new password
56          $hashedPassword = password_hash($newPassword, PASSWORD_DEFAULT);
57
58          // Update the employee's password in the database
59          $stmt = $pdo->prepare('UPDATE Employee SET password = ?,
                ↪ forgot_password_token = NULL, last_seen = NULL WHERE id = ?');
60          $stmt->execute([$hashedPassword, $employee['id']]);
61
62          // Redirect to the login page with a success message
63          header('Location: login.php?message=Password+reset+successful');
64          exit;
65      } else {
66          die('Both password fields must be filled out');
67      }
68  }
```

Listing 3.11: Password Reset Validation and Update

Upon receiving the request, the script first checks for the presence of the token in the URL. If a token is not provided, the execution is terminated immediately with a message indicating the absence of a token. This preliminary check is crucial for preventing unauthorized access to the password reset functionality (lines 18-21).

If a token is present, the script proceeds to establish a connection to the database (line 25). It performs a lookup for an employee record matching the provided token (lines 18-31). This step is verifying the legitimacy of the password reset request. If no matching record is found, or if the token is older than 15 minutes, the script terminates the execution with an error message, thereby ensuring that only valid requests proceed.

The script includes a security measure to assess the token's validity based on its

age (lines 37-41). It calculates a timestamp for fifteen minutes before the current time and compares it with the ‘last_seen‘ timestamp stored in the database alongside the token. If the token is older than fifteen minutes, the script concludes that the token has expired and terminates the execution. This mechanism serves to minimize the window during which a token can be used, enhancing the overall security of the process.

When the request to reset the password is submitted through the form (detected via the POST method), the script checks whether the new password and its confirmation are provided and match each other. This step is for user error prevention and ensuring that the user’s intent to reset the password is clear and confirmed. If the passwords do not match or if any field is left empty, the script terminates and notifies the user accordingly (lines 51-53).

Upon successful validation of the new passwords, the script hashes the new password using PHP’s ‘password_hash‘ function, providing a secure way to store passwords in the database. Hashing transforms the password into a format that cannot be easily reversed, adding a layer of security by protecting the actual password even if the database is compromised. It then updates the employee’s record by setting the new hashed password and clearing the ‘forgot_password_token‘ and ‘last_seen‘ fields. This update effectively completes the password reset process and revokes the token to prevent reuse (lines 55-60).

For the final step, the script redirects the user to the login page, appending a message to the URL indicating that the password reset was successful. This redirection serves both to inform the user of the successful password update and to guide them towards logging in with their new password, thereby concluding the password reset process (lines 62-67).

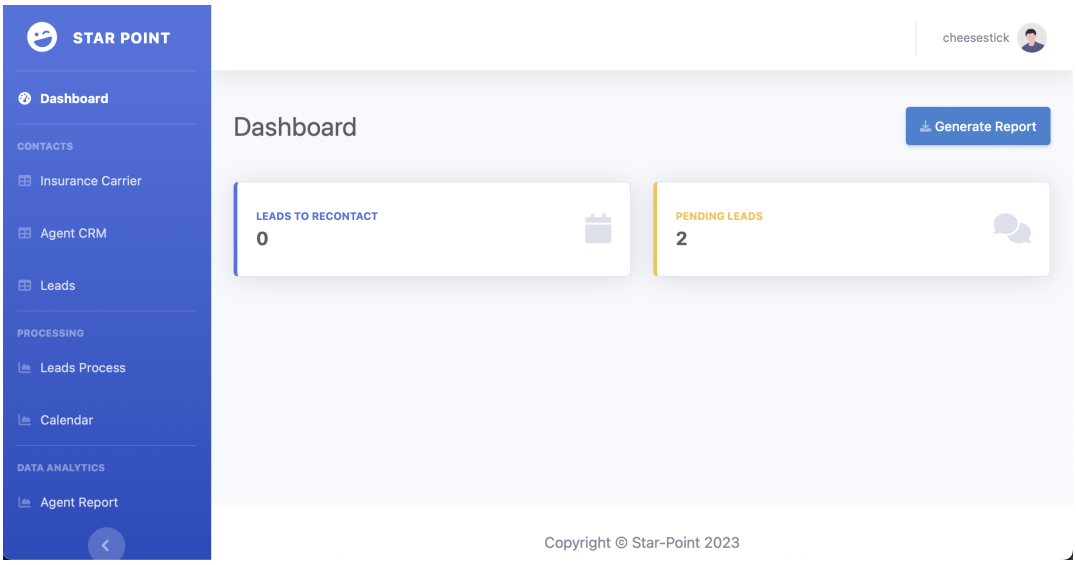

Figure 3.11: CRM Landing Page Dashboard

3.3.4 CRM Dashboard and Leads Table

Other than the CRM dashboard serving as the landing page following authentication, the page sets dynamic dashboard variables, a navigation bar, and a personal profile name. When a user is authenticated successfully, we set personalized session variables such as id, username, and role. The result of this action comes to fruition by having the ability to display their credentials on this page by stating 'htmlspecialchars((string)$session_data["name"]);.' Each page found within the CRM web app contains the profile and navigation bar. If each page were to contain static code per page, when a developer changed "dashboard" to "main" that would mean each page would need to change. This static allocation would also introduce inconsistency as some pages may be changed while others are forgotten.

```
1 <?php
2 include '../../php/functions.php';
3
4 session_start();
5 $pdo = pdo_connect_mysql();
6 check_loggedin($pdo, '../index.php');
7
8 // Combined SQL Query to count current events for today and pending leads
9 $query = "SELECT
```

```
10 (SELECT COUNT(*) FROM events WHERE datestart <= CURDATE() AND dateend >=
       ↪ CURDATE()) AS event_count,
11 (SELECT COUNT(*) FROM Leads WHERE serviced = 0) AS pending_count";
12
13 // Prepare and execute the query
14 $stmt = $pdo->prepare($query);
15 $stmt->execute();
16
17 // Fetching the result
18 $result = $stmt->fetch(PDO::FETCH_ASSOC);
19 $currentEventCount = $result['event_count'];
20 $pendingLeadsCount = $result['pending_count'];
21 ?>
22
23 <?=CRM_header('CRM Dashboard')?>
24        <!-- Content Wrapper -->
25        <div id="content-wrapper" class="d-flex flex-column">
26
27            <!-- Main Content -->
28            <div id="content">
29
30                <?= CRM_topbar($_SESSION) ?>
31
32                <!-- Begin Page Content -->
33                <div class="container-fluid">
```

Listing 3.12: CRM Landing Page php

As you can see the beginning of the dashboard page references the functions.php file that houses critical functions. The first function the server runs is the connection to our database followed by the check_loggedin($pdo, '../index.php'); function.

```
47 function check_loggedin($pdo, $redirect_file = 'index.php') {
48     // Check for remember me cookie variable and loggedin session variable
49     if (isset($_COOKIE['rememberme']) && !empty($_COOKIE['rememberme']) && !
           ↪ isset($_SESSION['loggedin'])) {
50         // If the remember me cookie matches one in the database then we can
               ↪ update the session variables.
51         $stmt = $pdo->prepare('SELECT * FROM Employee WHERE rememberme = ?');
52         $stmt->execute([ $_COOKIE['rememberme'] ]);
53         $account = $stmt->fetch(PDO::FETCH_ASSOC);
54         // If account exists...
55         if ($account) {
56             // Found a match, update the session variables and keep the user
                   ↪ logged-in
57             session_regenerate_id();
58             $_SESSION['loggedin'] = TRUE;
59             $_SESSION['name'] = $account['username'];
60             $_SESSION['id'] = $account['id'];
61             $_SESSION['role'] = $account['role'];
```

```
62            // Update last seen date
63            $date = date('Y-m-d\TH:i:s');
64            $stmt = $pdo->prepare('UPDATE Employee SET last_seen = ? WHERE id
                  ↪  = ?');
65            $stmt->execute([ $date, $account['id'] ]);
66        } else {
67            // If the user is not remembered redirect to the login page.
68            header('Location: ' . $redirect_file);
69            exit;
70        }
71    } else if (!isset($_SESSION['loggedin'])) {
72        // If the user is not logged in redirect to the login page.
73        header('Location: ' . $redirect_file);
74        exit;
75    }
76 }
```

Listing 3.13: Logged in function

The function takes two input parameters: the connection object to the database and a redirect page. The first condition statement checks for a remember me cookie variable as well as if a user has the boolean loggedin session variable. If a user is not loggedin then they should not have access to the page and will be redirected to the specified location.

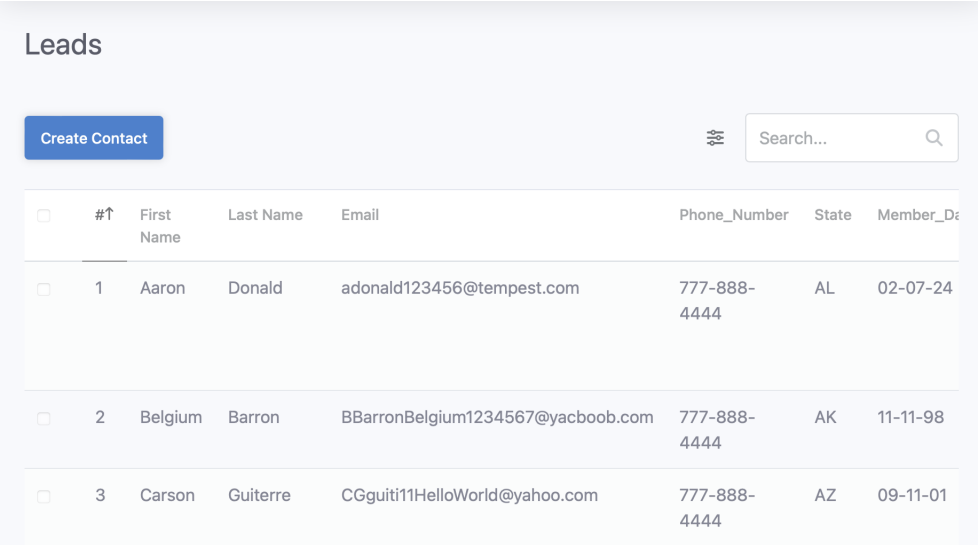

Figure 3.12: leads.php

As discussed previously, when a user submits the "get a free quote" form the information they provided is then inserted into the 'Leads' database table. If an

employee first logged in and wanted to view all leads, they would use the side navigation bar button to route themselves to the lead.php page. Please note that for each table within the CRM web app, all tables use a similar dynamic method to set the column headers and records accordingly. This is done by first using a config.php file that stores the database table names, primary key, and other miscellaneous items such as sortable and validation of regular expressions.

```
948 // Global Settings for Leads
949 $default_leads_column = 'id';
950 $Leads = 'Leads';
951 $Leads_Columns = [
952     'id' => [
953         'label' => '#',
954         'sortable' => true,
955         'type' => 'integer'
956     ],
957     'first_name' => [
958         'label' => 'First Name',
959         'sortable' => true,
960         'type' => 'string',
961         'input' => [
962             'placeholder' => 'Erick',
963             'type' => 'text',
964             'required' => false,
965             'validate_msg' => 'Last name must be between 1 and 50 characters
                    ↪ !',
966             'validate_regex' => '/^[a-zA-Z\'-]{1,50}$/',
967         ]
968     ],
969     ...
```

Listing 3.14: config.php page

This is a critical function because it allows a developer to predefine the database attributes and, in the future, allows the user to dynamically create, update, or delete certain attributes of a database table.

```
165 <div class="table">
166     <table>
167         <thead>
168             <tr>
169                 <td class="checkbox">
170                     <input type="checkbox" class="select-all">
171                 </td>
```

```
172                 <?php foreach ($AgentCRM_Columns as $column_key => $column):
                        ↪ ?>
173                 <td<?=$order_by==$column_key?' class="active"':''?>>
174                     <?php if ($column['sortable']): ?>
175                     <a href="agentcrm.php?page=1&records_per_page=<?=
                            ↪ $records_per_page?>&order_by=<?=$column_key?>&
                            ↪ order_sort=<?=$order_sort == 'ASC' ? 'DESC' : 'ASC
                            ↪ '?>&from_date=<?=$from_date?>&to_date=<?=$to_date
                            ↪ ?><?=isset($_GET['search']) ? '&search=' .
                            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">
176                     <?php endif; ?>
177                         <?=$column['label']?>
178                         <?php if ($order_by == $column_key): ?>
179                         <i class="fa-solid fa-arrow-<?=str_replace(array('ASC',
                                ↪  'DESC'), array('up', 'down'), $order_sort)?>-
                                ↪ long fa-sm"></i>
180                         <?php endif; ?>
181                     <?php if ($column['sortable']): ?>
182                     </a>
183                     <?php endif; ?>
184                 </td>
185                 <?php endforeach; ?>
186                 <td></td>
187             </tr>
188         </thead>
```

Listing 3.15: Dynamic setting of column headers within the leads table

The above code snippet is the beginning of the leads table with PHP embedded within it, and it primarily focuses on rendering the header column headers of the table. This section is crucial as it defines the headers and incorporates interactive features like sorting and selection. At the beginning of the header, there's a dedicated table cell (<td>) for a checkbox. This checkbox, with the class 'select-all', is used to select or deselect all rows in the table, a common feature in tables where multiple entries might need to be selected simultaneously. Following this, the code uses a PHP foreach loop to iterate over an array named $Leads_Columns. This array, which is found within our config.php file, contains the details of the columns to be displayed in the table. For each column in this array, a table cell (<td>) is created. The code checks if the current column is the one by which the table is sorted (as indicated by the $order_by variable matching the $column_key).

Within each table cell, if the column is marked as 'sortable' in the $Leads_Columns

array (indicated by $column['sortable'] on line 178), the column label is wrapped in an anchor (<a>) tag. This tag creates a hyperlink that, when clicked, will re-request the 'leads.php' page with updated query parameters. These parameters include the current page number, records per page, the column to order by, the sort direction (toggled between 'ASC' and 'DESC'), and any date filters or search terms applied. This functionality allows for dynamic sorting of the table data based on the column headers.

Additionally, if the current column is the one by which the data is sorted, an icon is displayed. This icon visually indicates the current sort direction (ascending or descending), enhancing the user's understanding of the data presentation. The PHP foreach loop ensures that this functionality is applied to each column defined in the $Leads_Columns array, making the table header dynamic and interactive. The final table cell (<td>) in the row is left empty, possibly to maintain a uniform structure or for additional features not shown in the snippet.

Next, the script determines the number of records to display on each page. This value is also obtained from a URL parameter ('records_per_page'). If not specified, it defaults to a predetermined value stored in $default_records_per_page (In this case the default is 10 records specified in config.php).

The 'records_per_page' parameter can also be set to 'all', indicating that all records should be displayed on a single page. The script then identifies which column the data should be sorted by. This is achieved by reading the 'order_by' URL parameter and ensuring it matches one of the keys in the $Leads_Columns array, which presumably contains valid column names for sorting. If the 'order_by' parameter is not set or is invalid, the script defaults to sorting by a default column, likely 'id' or another relevant field.

Similarly, the sort order (ascending or descending) is determined based on the 'order_sort' URL parameter. If the parameter is 'DESC', the sorting will be in

descending order; otherwise, it defaults to ascending order ('ASC'). The script also handles date filters for the data. It reads 'from_date' and 'to_date' parameters from the URL and stores them. These parameters are used to filter the data between specific dates.

Moreover, the script includes functionality to search within the data. If a 'search' parameter is present in the URL, the script constructs a SQL 'WHERE' clause that searches across all columns specified in the $Leads_Columns array. In addition to searching, the script constructs conditions for the SQL query based on the 'from_date' and 'to_date' parameters. These conditions are applied to a 'created' column in the database, which is a DATETIME column. The script checks for various combinations: both dates present, only 'from_date', or only 'to_date', and accordingly, it constructs the 'WHERE' clause for the SQL query.

```
188 ...
189 <tbody>
190     <?php if (empty($results)): ?>
191     <tr>
192         <td colspan="10" style="text-align:center;">There are no records.</td
                ↪ >
193     </tr>
194     <?php endif; ?>
195     <?php foreach ($results as $result): ?>
196     <tr>
197         <td class="checkbox"><input type="checkbox" value="<?=$result['
                ↪ contact_id']?>" name="record[]"></td>
198         <?php foreach ($AgentCRM_Columns as $column_key => $column): ?>
199         <?php if ($column['type'] == 'datetime'): ?>
200         <td class="<?=$column_key?>"><?=date('m-d-y H:i', strtotime($result[
                ↪ $column_key]))?></td>
201         <?php elseif ($column['type'] == 'date'): ?>
202         <td class="<?=$column_key?>"><?=date('m-d-y', strtotime((string)
                ↪ $result[$column_key]))?></td>
203         <?php elseif ($column['type'] == 'integer'): ?>
204         <td class="<?=$column_key?>"><?=number_format($result[$column_key])
                ↪ ?></td>
205         <?php elseif ($column['type'] == 'tel'): ?>
206         <td class="<?= $column_key ?>"><?= formatPhoneNumber($result[
                ↪ $column_key]) ?></td>
207         <?php elseif ($column['type'] == 'time'): ?>
208         <td class="<?= $column_key ?>"><?= date('H:i', strtotime($result[
                ↪ $column_key])) ?></td>
209         <?php else: ?>
```

```
210         <td class="<?=$column_key?>"><?=htmlspecialchars((string) $result[
                ↪ $column_key], ENT_QUOTES)?></td>
211         <?php endif; ?>
212         <?php endforeach; ?>
213         <td class="actions">
214             <a href="agentcrm-update.php?contact_id=<?=$result['contact_id
                    ↪ ']?>" class="edit"><i class="fa-solid fa-pen fa-xs"></i></
                    ↪ a>
215             <a href="agentcrm-delete.php?contact_id=<?=$result['contact_id
                    ↪ ']?>" class="trash"><i class="fa-solid fa-xmark fa-xs"></i
                    ↪ ></a>
216         </td>
217     </tr>
218     <?php endforeach; ?>
219 </tbody>
220 </table>
```

Listing 3.16: Dynamic setting of records within the leads table

Let's continue with dynamically setting the records of the leads table within the leads.php file. After dynamically setting the column headers this HTML table specifically focuses on the table body (<tbody>) which dynamically generates table rows based on the data available in the $results array, which again is found within the config.php file. At the outset, the script checks if the $results array is empty. If so, it displays a single table row with the message "There are no records." spanning across multiple columns. This is a common way to inform users when there is no data to display, ensuring the user interface remains informative and clean.

If the $results array is not empty, the script iterates over each item in this array using a PHP foreach loop. Each iteration represents a data record, and for every record, a table row (<tr>) is created. Within each row, the first cell contains a checkbox, allowing users to select individual records. This checkbox is associated with the record's unique identifier, which in this instance is the 'id' field.

Following this, another foreach loop iterates over the $Leads_Columns array. This array defines the structure and data types of the columns to be displayed in the table. For each column, the script generates a table cell (<td>) and populates it with

the corresponding data from the current $result. The way data is formatted and displayed is dependent on the type of the column as defined in $Leads_Columns:

- If the column type is 'datetime', the corresponding data is formatted as 'Y-m-d H:i'.
- If the column type is 'date', the data is formatted as 'Y-m-d'.
- If the column type is 'integer', the data is formatted with number_format(), which adds commas as needed between thousands.
- For all other types, the data is displayed as-is, but passed through the htmlspecialchars() function to prevent XSS attacks by escaping HTML characters.loading

In summary, each row includes a special 'actions' cell, containing links to 'leads-update.php' and 'leads-delete.php' with the corresponding record's ID as a parameter. These links are used for editing and deleting the respective record. The links are visually represented with icons which allows the user to have a better functioning interface.

Listing 3.17: Records Per Page Selection

```
224 <div class="records-per-page">
225     <select name="bulk_action" class="bulk-action">
226         <option value="" disabled selected>Bulk Actions</option>
227         <option value="delete">Delete</option>
228         <option value="edit">Edit</option>
229         <option value="export">Export</option>
230     </select>
231     <a href="agentcrm.php?page=1&records_per_page=5&order_by=<?=$order_by?>&
        ↪ order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date=<?=
        ↪ $to_date?><?=isset($_GET['search']) ? '&search=' . htmlentities(
        ↪ $_GET['search'], ENT_QUOTES) : ''?>">5</a>
232     <a href="agentcrm.php?page=1&records_per_page=10&order_by=<?=$order_by
        ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
        ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
        ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">10</a>
```

```
233     <a href="agentcrm.php?page=1&records_per_page=20&order_by=<?=$order_by
            ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
            ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">20</a>
234     <a href="agentcrm.php?page=1&records_per_page=50&order_by=<?=$order_by
            ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
            ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">50</a>
235     <a href="agentcrm.php?page=1&records_per_page=100&order_by=<?=$order_by
            ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
            ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">100</a>
236     <!-- <a href="agentcrm.php?page=1&records_per_page=all&order_by=<?=
            ↪ $order_by?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date
            ↪ ?>&to_date=<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">all</a> -->
237 </div>
```

Moving beyond the table heading and records the user has several "bulk" functionalities typically found within a table. At the bottom left the user has a select option between delete, edit, and export. If a user wanted to delete several records from the table, they would achieve this by using the bulk actions sections of the table. As well as pagination, a sequence of numbers assigned to differing pages of the data table. The HTML anchor (<a>) tags are designed to provide pagination and data display options for the web page. Each link allows users to specify how many records they wish to view per page, with options of 5, 10, 20, 50, 100, or all records at once.

```
1  <div class="records-per-page">
2      <select name="bulk_action" class="bulk-action">
3          <option value="" disabled selected>Bulk Actions</option>
4          <option value="delete">Delete</option>
5          <option value="edit">Edit</option>
6          <option value="export">Export</option>
7      </select>
8      <a href="agentcrm.php?page=1&records_per_page=5&order_by=<?=$order_by?>&
           ↪ order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date=<?=
           ↪ $to_date?><?=isset($_GET['search']) ? '&search=' . htmlentities(
           ↪ $_GET['search'], ENT_QUOTES) : ''?>">5</a>
9      <a href="agentcrm.php?page=1&records_per_page=10&order_by=<?=$order_by
           ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
           ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
           ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">10</a>
```

```
10      <a href="agentcrm.php?page=1&records_per_page=20&order_by=<?=$order_by
            ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
            ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">20</a>
11      <a href="agentcrm.php?page=1&records_per_page=50&order_by=<?=$order_by
            ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
            ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">50</a>
12      <a href="agentcrm.php?page=1&records_per_page=100&order_by=<?=$order_by
            ↪ ?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date?>&to_date
            ↪ =<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">100</a>
13      <!-- <a href="agentcrm.php?page=1&records_per_page=all&order_by=<?=
            ↪ $order_by?>&order_sort=<?=$order_sort?>&from_date=<?=$from_date
            ↪ ?>&to_date=<?=$to_date?><?=isset($_GET['search']) ? '&search=' .
            ↪ htmlentities($_GET['search'], ENT_QUOTES) : ''?>">all</a> -->
14  </div>
```

Listing 3.18: Records Per Page Selection

Following the dropdown, a series of anchor (<a>) tags are provided. These links allow users to specify how many records they wish to view per page on the 'leads.php' page, with options of 5, 10, 20, 50, 100, or all records. Each link includes URL parameters that maintain the current state of the page, such as the sorting order (order_by and order_sort), date filters (from_date and to_date), and any active search terms. The use of htmlentities in the search term parameter helps prevent XSS attacks by safely encoding any special characters (lines 8-13).

In the second section within another sub- named "pagination," the code handles the page navigation. This part of the code dynamically generates links for navigating through the pages of records. It considers the current page number ($page), total number of results ($num_results), and the number of records per page ($records_per_page).

- If the user is not on the first page and there are multiple pages of results, links to previous pages and the first page are provided, along with 'Prev' and 'Next' buttons for easier navigation.

- The code dynamically calculates and displays page numbers and includes '...'

(dots) as placeholders to indicate the presence of additional pages not directly linked.

- The current page number is highlighted with a 'selected' class, enhancing user orientation within the pagination system.

This pagination mechanism is designed to be responsive to the amount of data and the current page the user is on, providing a user-friendly way to navigate through large sets of data. It also retains any active filters and sorting options while navigating between pages, ensuring a consistent and intuitive user experience.

3.3.5 Profile Page

On each page of the CRM the currently loggedin user will have the option to view their profile at the top right of the page.

Figure 3.13: Profile Page Settings

Once a user clicks either the profile icon or their username the modal in figure 3.14 will appear.

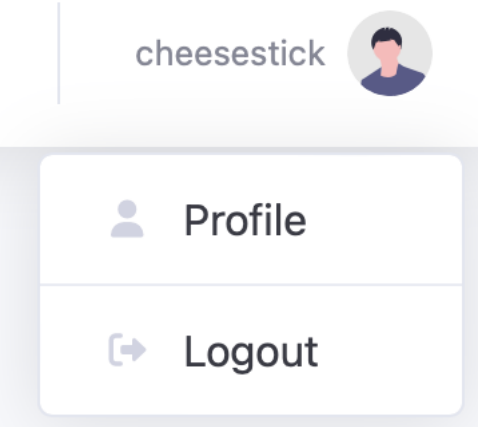

Figure 3.14: Profile Page Settings

If a user chooses to logout and selects the "logout" button found within these selections they will be shown the options in figure 3.15. Allowing the user to have the final option to either cancel their selection or continue to logout.

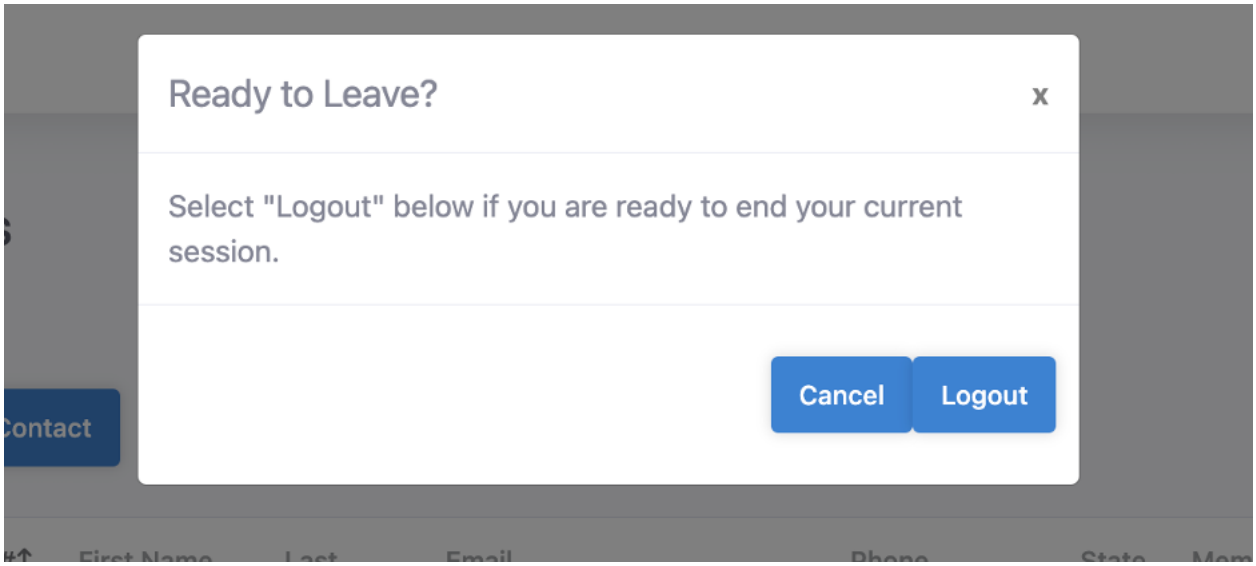

Figure 3.15: Signout warning modal

If a user clicks either the "X", Cancel, or anywhere outside of the boxed modal they will return back to the page they were previously on. When a user selects logout the button uses a file called "logout.php" to execute the following user command (line 394, Listing 3.19). As you may have noticed when a user select logout the logout modal or the center-adjusted box is summoned. This box is

predefined within the functions.php file as any change to the way a user signs out will be directly impacted here.

```
378 function logout_modal(){
379     echo <<<EOT
380     <!-- Logout Modal-->
381     <div class="modal fade" id="logoutModal" tabindex="-1" role="dialog"
            ↪ aria-labelledby="exampleModalLabel"
382         aria-hidden="true">
383         <div class="modal-dialog" role="document">
384             <div class="modal-content">
385                 <div class="modal-header">
386                     <h5 class="modal-title" id="exampleModalLabel">Ready to
                            ↪ Leave?</h5>
387                     <button class="close" type="button" data-dismiss="modal"
                            ↪ aria-label="Close">
388                         <span aria-hidden="true">x</span>
389                     </button>
390                 </div>
391                 <div class="modal-body">Select "Logout" below if you are ready
                        ↪  to end your current session.</div>
392                 <div class="modal-footer">
393                     <button class="btn btn-secondary" type="button" data-
                            ↪ dismiss="modal">Cancel</button>
394                     <a class="btn btn-primary" href="../../php/logout.php">
                            ↪ Logout</a>
395                 </div>
396 
397             </div>
398         </div>
399     </div>
400     EOT;
401 }
```

Listing 3.19: PHP function for logout modal

```
1  <?php
2  session_start();
3  // Destroy the session associated with the user
4  session_destroy();
5  // If the user is remembered, delete the cookie
6  if (isset($_COOKIE['rememberme'])) {
7      unset($_COOKIE['rememberme']);
8      setcookie('rememberme', '', time() - 3600);
9  }
10 // Redirect to the login page:
11 header('Location: ../view/index.php');
```

Listing 3.20: PHP function logout

The above logout.php file performs several key actions to ensure that the user's session is properly terminated and that any persistent login credentials are removed. The script begins with session_start(), which seems counterintuitive but is necessary to access and manipulate the session data. Next, the script calls session_destroy() on line 4 which effectively destroys all data associated with the current session. This step is critical for logging out, as it clears any session variables and ensures that no residual data from the user's session remains on the server. The script also includes a check for a 'rememberme' cookie. The 'rememberme' cookie is used to keep users logged in even after the browser is closed, thereby bypassing the need to log in again for subsequent visits. If this cookie is set (checked using isset($_COOKIE['rememberme'])), the script proceeds to unset it. This is done in two steps:

- 'unset($_COOKIE['rememberme'])' removes the cookie from the $_COOKIE array, but this alone does not delete the cookie from the user's browser.
- 'setcookie('rememberme', '', time() - 3600)' is then used to send a command back to the user's browser to delete the cookie. It sets the cookie's expiration date to one hour in the past, which instructs the browser to discard it.

Finally, the logout script redirects the user to the user index page using the header('Location: ../view/index.php') function. As discussed previously this is the same page employees will log in from and also serves as the user platform to understand more about the company.

3.3.6 Administrative Page

If a user wishes to change any information within their profile they will navigate to the profile section (Figure 3.16). In other words, a user will click once at the top

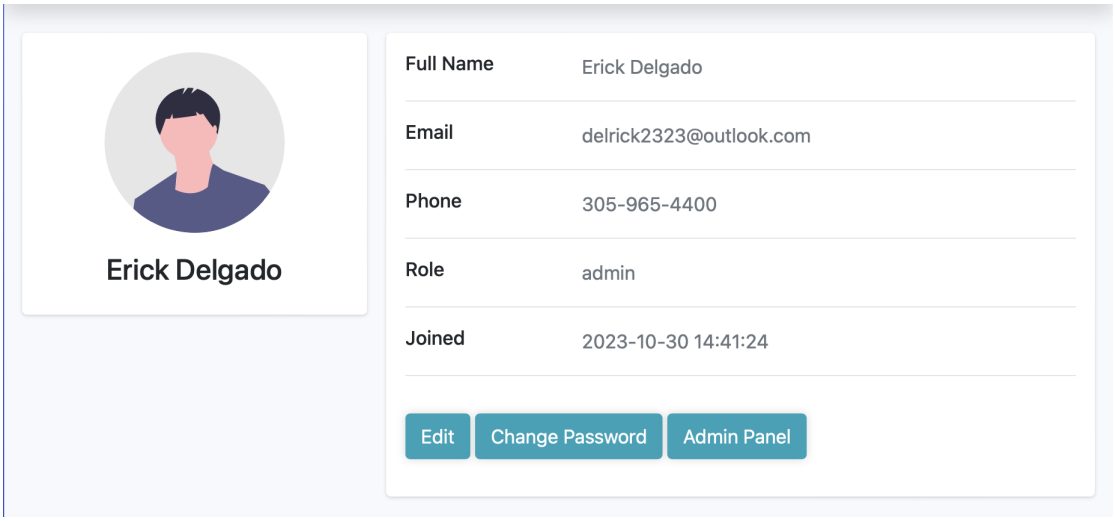

Figure 3.16: Profile Page

right of the page to open the profile selections and then choose profile to view the below page.

The profile page in figure 3.16 displays the user's full name, email, phone, their assigned role, and the DateTime they joined the company. If the employee's role is not admin the 'Admin Panel' button selection will not appear.

```
196 <div class="row">
197     <div class="col-sm-12">
198         <a class="btn btn-info" href="profile.php?action=edit">Edit</a>
199         <a class="btn btn-info" href="profile.php?action=change_password">
                ↪ Change Password</a>
200         <?php if ($contact['role'] == 'admin'): ?>
201             <a class="btn btn-info" target="_blank" href="admin.php">Admin
                    ↪ Panel</a>
202         <?php endif; ?>
203     </div>
204 </div>
```

Listing 3.21: Admin Panel Dynamic Button

This is achieved in line 200 by creating a conditional if statement to check if the role attribute of the $contact array is equivalent to 'admin'. When this condition is true, indicating that the user is an administrator, the snippet renders an HTML anchor (<a>) tag. This tag, styled as a button with Bootstrap classes 'btn btn-info', links to 'admin.php', the URL of the admin panel or dashboard of the

web application. The link is set to open in a new tab or window, as denoted by target="_blank". The conditional logic ensures that this navigation option to the admin panel is exclusively available to users identified as administrators.

Before we move on talking about administrative controls let's first describe the profile page as well as how a user can change their password. If a user were to want to change their password they would simply select the appropriate HTML anchor (<a>) tag to edit their profile (line 198). The 'href' attribute of the link points to 'profile.php', and it includes a query string '?action=edit'. This query string is a method to pass information to the 'profile.php' page, indicating that the user intends to edit their profile. As a result, when the link is clicked, the user is navigated to the 'profile.php' page, where they are presented with a form to edit their profile details.

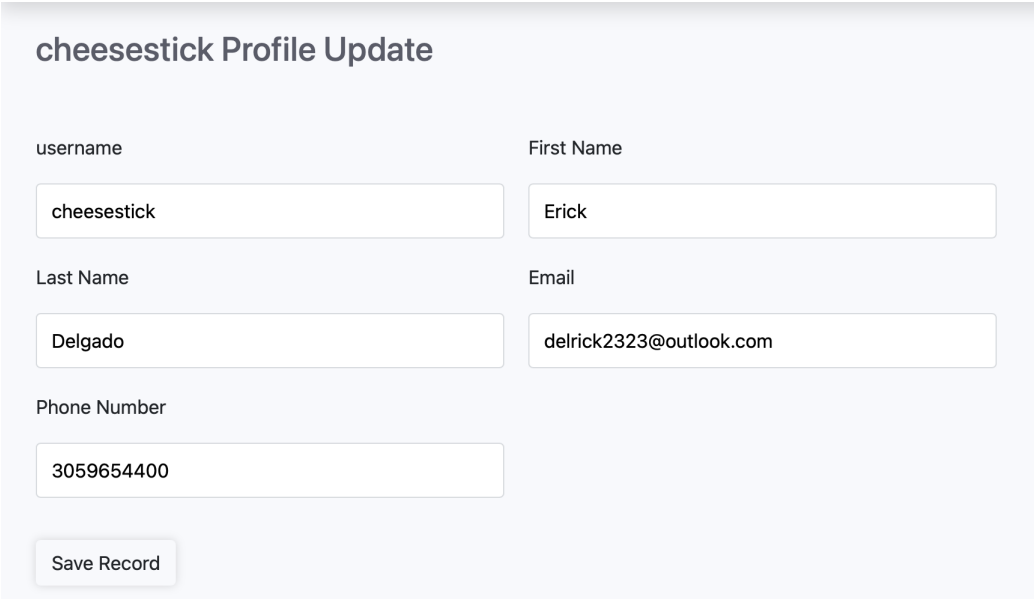

Figure 3.17: Edit profile

```
209 <?php elseif ($_GET['action'] == 'edit'): ?>
210 <div class="container-fluid">
211     <div class="d-sm-flex align-items-center justify-content-between mb-4">
212         <div class="wrap">
213             <h1 class="h3 mb-0 text-gray-800"><?=$contact['username']?>
                    ↪ Profile Update</h1>
214         </div>
215     </div>
```

```
216
217     <form action="?id=<?=$contact['id']?>" method="post" class="crud-form">
218         <div class="cols">
219             <?php foreach ($Employee_Profile_Columns as $column => $array):
                    ↪ ?>
220                 <?php if (isset($array['input'])): ?>
221                     <?php $input = $array['input']; ?>
222                     <div class="style-form-control">
223                         <label for="<?=$column?>"><?=$array['label']?></label>
```

Listing 3.22: Edit Profile

```
224 <?php if ($input['type'] == 'text' || $input['type'] == 'hidden' || $input['
        ↪ type'] == 'email' || $input['type'] == 'number' || $input['type'] ==
        ↪  'tel'): ?>
225     <input id="<?=$column?>" type="<?=$input['type']?>" name="<?=$column?>"
            ↪ placeholder="<?=$input['placeholder']?>" value="<?=
            ↪ htmlspecialchars((string)$contact[$column], ENT_QUOTES)?>" <?=
            ↪ $input['required'] ? 'required' : ''?> <?=isset($input['custom'])
            ↪  ? $input['custom'] : ''?>>
226 <?php elseif ($input['type'] == 'date'): ?>
227     <input id="<?=$column?>" type="<?=$input['type']?>" name="<?=$column?>"
            ↪ <?=$input['required'] ? ' required' : ''?> <?=isset($input['
            ↪ custom']) ? $input['custom'] : ''?> value="<?=date('m-d-Y')?>">
228 <?php elseif ($input['type'] == 'datetime-local'): ?>
229     <input id="<?=$column?>" type="<?=$input['type']?>" name="<?=$column?>"
            ↪ value="<?=date('m-d-Y\TH:i', strtotime($contact[$column]))?>" <?=
            ↪ $input['required'] ? 'required' : ''?> <?=isset($input['custom'])
            ↪  ? $input['custom'] : ''?>>
230 <?php elseif ($input['type'] == 'textarea'): ?>
231     <textarea id="<?=$column?>" name="<?=$column?>" placeholder="<?=$input['
            ↪ placeholder']?>" <?=$input['required'] ? 'required' : ''?> <?=
            ↪ isset($input['custom']) ? $input['custom'] : ''?>><?=
            ↪ htmlspecialchars($contact[$column], ENT_QUOTES)?></textarea>
232 <?php elseif ($input['type'] == 'select'): ?>
233     <select id="<?=$column?>" name="<?=$column?>" <?=$input['required'] ? '
            ↪ required' : ''?> <?=isset($input['custom']) ? $input['custom'] :
            ↪ ''?>>
234         <?php foreach ($input['options'] as $option): ?>
235             <option value="<?=$option?>" <?=$contact[$column] == $option ? '
                    ↪ selected' : ''?>><?=$option?></option>
236         <?php endforeach; ?>
237     </select>
238 <?php elseif ($input['type'] == 'checkbox'): ?>
239     <input id="<?=$column?>" type="hidden" name="<?=$column?>" value="0" <?=
            ↪ isset($input['custom']) ? $input['custom'] : ''?>>
240     <input type="<?=$input['type']?>" name="<?=$column?>" value="1" <?=
            ↪ $contact[$column] == 1 ? 'checked' : ''?> <?=isset($input['custom
            ↪ ']) ? $input['custom'] : ''?>>
241 <?php elseif ($input['type'] == 'radio'): ?>
242     <?php foreach ($input['options'] as $option): ?>
243         <div>
```

```
244             <input id="<?=$option?>" type="<?=$input['type']?>" name="<?=
                    ↪ $column?>" value="<?=$option?>" <?=$contact[$column] ==
                    ↪ $option ? 'checked' : ''?> <?=isset($input['custom']) ?
                    ↪ $input['custom'] : ''?>>
245             <label for="<?=$option?>"><?=$option?></label>
246         </div>
247     <?php endforeach; ?>
248 <?php endif; ?>
249 </div>
250 <?php endif; ?>
251 <?php endforeach; ?>
252 </div>
253 ...
```

Listing 3.23: edit profile continued

The profile.php?action=edit page begins with a PHP elseif condition checking if the 'action' GET parameter is set to 'edit'. If this condition is met, the following HTML and PHP code is executed to display the edit form. The form is wrapped inside a div with the class 'container-fluid' which is a predefined bootstrap styling. Within this container, there's a div that is designed to align items center with some margin at the bottom. This div contains another div with the class wrap, where the title of the form is displayed. The title dynamically includes the username of the contact being edited, indicating that this form is meant for updating the profile of a specific user.

The form element itself is set up with an action that points to the current page (indicated by the?id=<?=$contact['id']?> in the action attribute (line 217)), meaning the form data will be sent to the same page for processing upon submission. The method is 'post', ensuring data is sent via HTTP POST. Inside the form, class cols defins a column layout. Then, a PHP foreach loop iterates over the array $Employee_Profile_Columns define within the config.php file. For each iteration process, a column of the employee profile, and depending on the type of input required (text, email, number, telephone, date, datetime-local, textarea, select, checkbox, or radio), renders the appropriate HTML input element.

Each of these dynamically mapped fields is populated with existing data (using

htmlspecialchars to safely encode special characters) and includes various attributes like 'required' and 'custom'. For example, 'date' inputs default to the current date, and select inputs to iterate over their options to create a dropdown menu. Checkbox and radio inputs handle their checked state based on the current data. As seen previously the form also includes error and success message handling. Checking if there are any error or success messages (stored in $error_msg and $success_msg variables) and, if present, displays them to the user. This is an essential feature for providing feedback about the form submission process.

Once the form is submitted the below PHP code will then attempt to update the profile attributes accordingly.

```
27 // Check if POST data exists (user submitted the form)
28 if (isset($_POST['submit'])) {
29     // Iterate through the fields and extract the data from the form
30     $data = [];
31     foreach ($Employee_Profile_Columns as $column => $array) {
32         if (isset($_POST[$column])) {
33             $data[$column] = $_POST[$column];
34             // Validate
35             if ((isset($array['input']['required']) && !$array['input']['
                   ↪ required'] && empty($_POST[$column]))) {
36                 continue;
37             }
38             if (isset($array['input']['validate_regex']) && !preg_match(
                   ↪ $array['input']['validate_regex'], $_POST[$column])) {
39                 $error_msg = isset($array['input']['validate_msg']) ? $array['
                       ↪ input']['validate_msg'] : 'Please enter a valid ' .
                       ↪ $column . '.';
40             }
41         }
42     }
43     // If no validation errors, proceed to update the record(s) in the
           ↪ database
44     if (!$error_msg) {
45         // Update the record
46         $stmt = $pdo->prepare('UPDATE ' . $Employee . ' SET ' . implode(', ',
               ↪  array_map(function ($column) {
47             return $column . ' = :' . $column;
48         }, array_keys($data))) . ' WHERE id = :id');
49         // bind over the data to the placeholders in the prepared statement
50         foreach ($data as $column => $value) {
51             $stmt->bindValue(':' . $column, $value);
52         }
53         // Bind ID
```

```
54          $stmt->bindValue(':id', $userId);
55          // Execute the SQL statement
56          $stmt->execute();
57          // Get the updated contact from the contacts table
58          $stmt = $pdo->prepare('SELECT * FROM ' . $Employee . ' WHERE id = ?')
                ↪ ;
59          $stmt->execute([ $userId ]);
60          $contact = $stmt->fetch(PDO::FETCH_ASSOC);
61          // Output message
62          $success_msg = 'Updated Successfully!';
63      }
64  }
```

Listing 3.24: Admin Panel Page

When the user submits the form (checked by isset($_POST['submit'])), the script iterates over the columns defined in $Employee_Profile_Columns. For each column, it extracts the corresponding data from the POST request. The script also includes validation logic, such as checking for required fields and applying regex-based validation if specified. If any validation fails, an error message is set. If there are no validation errors, the script proceeds to update the user's data in the database. A SQL 'UPDATE' statement is dynamically constructed using the data from the form. The prepared statement is then executed using PDO, with each piece of data bound to the corresponding placeholder in the SQL statement. After executing the update, the script fetches the updated user data from the database and updates the $contact variable with this new data. This step ensures that the most current data is available for display or further processing. In the last step, if the update is successful, a success message is set, informing the user that the profile was updated successfully.

3.3.7 Administrative Page

Beyond the profile page, if a user is an admin, they can enter the admin page through the profile section detailed in the last section. When an admin wishes to change any information within their profile or other profiles they would do so here.

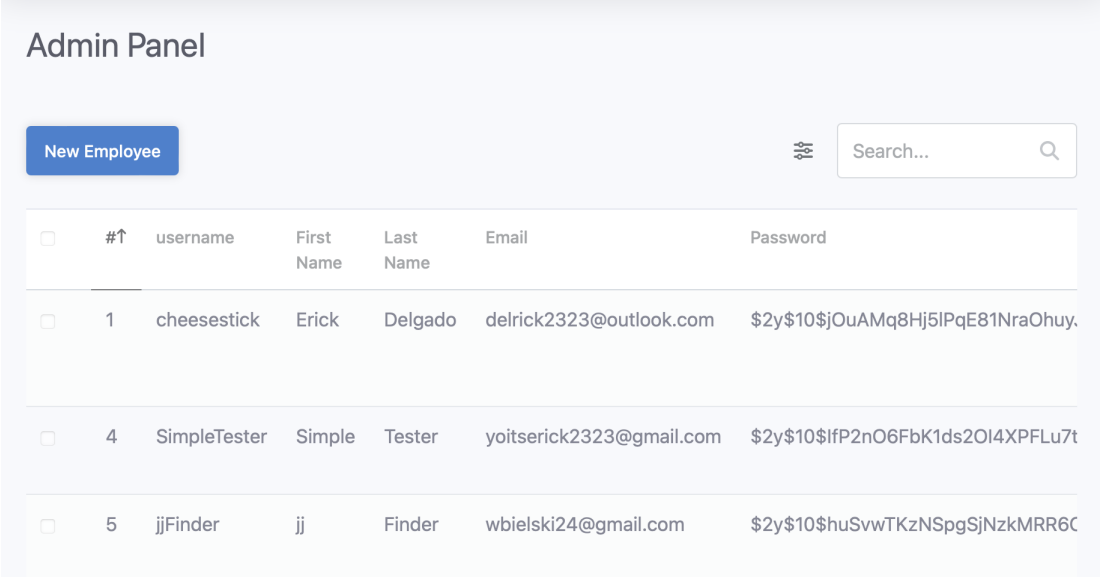

Figure 3.18: Admin Panel Page

The admin panel displays the contents of the "Employee" table. This is done using a very similar format to the Leads table the only noticeable difference is that we are of course using a separate table for displaying data.

3.3.7.1 Creating a New Employee

Building from the previous section, when the management of the CRM platform wishes to create additional employees, they can simply select the "New Employee" icon displayed at the top of the page.

However, a user who does not have admin privileges will not be able to see this page or any other admin-dedicated page. This is the result of introducing a new function called check_role() in line 23. The function is designed to check the user's role against a predefined set of roles to control access to admin controls.

```
8  // check if we are logged in
9  check_loggedin($pdo, '../index.php');
10 check_role(['admin'], 'index.php');
```

Listing 3.25: Call the check role function

It takes two parameters: '$roles', which can be a single role as a string or an

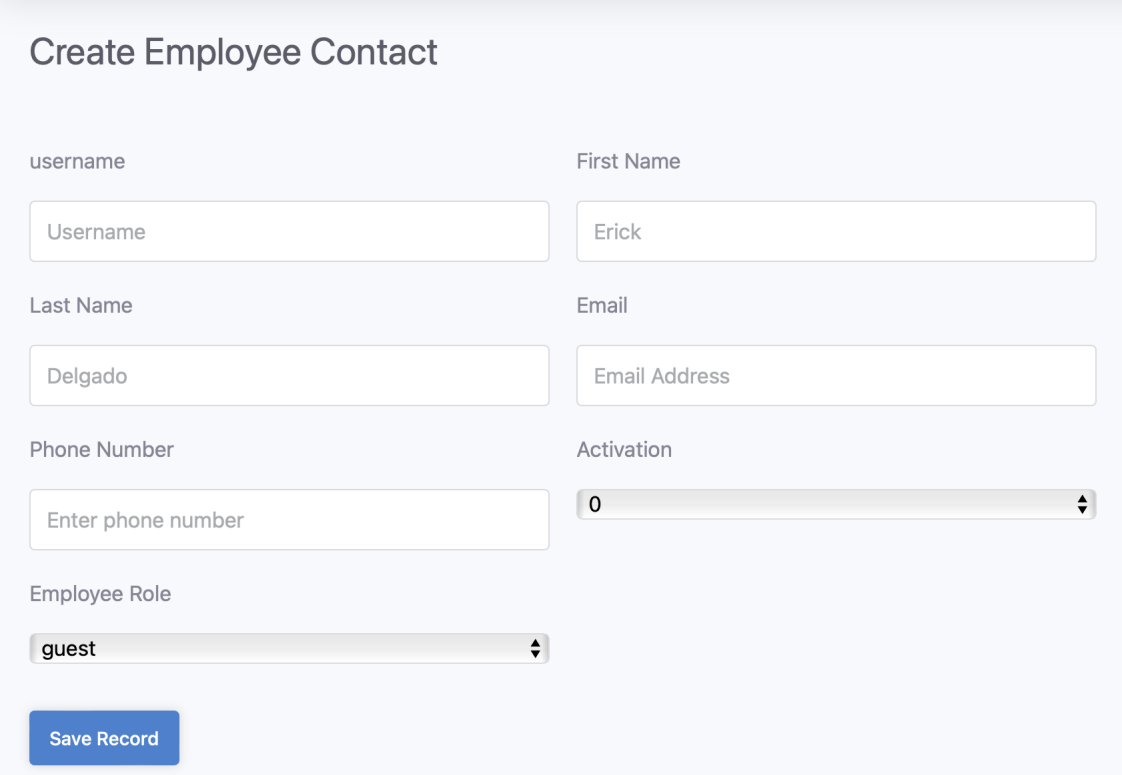

Figure 3.19: Create an employee

array of roles, and '$redirectFile', which specifies the location to redirect the user if their role does not match any of the roles specified in '$roles'. The '$redirectFile' parameter has a default value of ''index.php'', meaning that if no redirection file is specified when the function is called, it will default to redirecting the user to 'index.php'.

```
34 function check_role($roles, $redirectFile = 'index.php') {
35     $redirectFile = filter_var($redirectFile, FILTER_SANITIZE_URL);
36     $roles = is_array($roles) ? $roles : [$roles];
37
38     if (!in_array($_SESSION['role'], $roles)) {
39         header('Location: ' . $redirectFile);
40         exit;
41     }
42 }
```

Listing 3.26: Implementation of the check role function

The function begins by sanitizing the ‘$redirectFile‘ parameter using the ‘FILTER_SANITIZE_URL‘ filter, which removes all illegal characters from a URL, ensuring that the redirection path is safe and valid. Next, the function ensures that ‘$roles‘ is an array. If ‘$roles‘ is provided as a string (indicating a single role), it is converted into an array with that single role as its element. This standardization allows for consistent processing regardless of whether a single role or multiple roles are checked. The core functionality is to check if the current user’s role, stored in the session variable ‘$_SESSION[’role’]‘, is in the array of allowed ‘$roles‘. This is accomplished by using the ‘in_array‘ function, which searches the array for the specific value and returns ‘true‘ if the value is found. If the user’s role is not found in the ‘$roles‘ array, the function initiates a redirection to the specified ‘$redirectFile‘ using the ‘header(’Location: ’ . $redirectFile)‘ function, effectively blocking access to the current page or script for users who do not have the required role. The ‘exit;‘ command immediately following the redirection instruction ensures that the script execution is terminated after the redirection command, preventing any further script execution that might be intended only for authorized roles.

Once we have ensured that the current user can access this webpage, we display the form contents for a user to fill out and submit. Upon submission, it processes the form data, validating and sanitizing input against defined criteria. This includes ensuring required fields are filled and data conforms to specified formats, thereby maintaining data integrity and security.

```
29 // Check if POST data exists (user submitted the form)
30 if (isset($_POST['submit'])) {
31     // Iterate through the fields and extract the data from the form
32     $data = [];
33     foreach ($Employee_Admin_Columns as $column => $array) {
34         if (isset($_POST[$column])) {
35             $data[$column] = $_POST[$column];
36             // Validate
37             if ((isset($array['input']['required']) && !$array['input']['
                   ↪ required'] && empty($_POST[$column]))) {
38                 continue;
```

```
39              }
40              if (isset($array['input']['validate_regex']) && $array['input']['
                    ↪ validate_regex']) {
41                  if (!preg_match($array['input']['validate_regex'], $_POST[
                        ↪ $column])) {
42                      $error_msg = isset($array['input']['validate_msg']) ?
                            ↪ $array['input']['validate_msg'] : 'Please enter a
                            ↪ valid ' . $column . '.';
43                  }
44              }
45          }
46      }
47      // If no validation errors, proceed to insert the record(s) in the
            ↪ database
48      if (!$error_msg) {
49          // Generate a new random password
50          $newPassword = generateRandomPassword(); // You can specify the
                ↪ length as a parameter
51
52          // Hash the new password
53          $hashedPassword = password_hash($newPassword, PASSWORD_DEFAULT);
54
55          // Add the hashed password to the data array
56          $data['password'] = $hashedPassword;
57
58          // Regsitered field
59          $data['registered'] = date('Y-m-d H:i:s');
60
61          // Insert the records
62          $stmt = $pdo->prepare('INSERT INTO ' . $Employee . ' (' . implode(',
                ↪ ', array_keys($data)) . ') VALUES (' . implode(', ', array_map
                ↪ (function ($column) {
63              return ':' . $column;
64          }, array_keys($data))) . ')');
65
66          // Bind over the data to the placeholders in the prepared statement
67          foreach ($data as $column => $value) {
68              $stmt->bindValue(':' . $column, $value);
69          }
70
71          // Execute the SQL statement
72          $stmt->execute();
```

Listing 3.27: New employee php

Once the script recognizes that the user's data was correctly input it continues handling the employee creation. This includes generating and hashing a new password for the user, which is then prepared along with other user data for insertion into the database. This process not only secures the user's password

through hashing but also prepares the data for database entry, ensuring that the information stored is accurate and obfuscated.

```
74  // Send email with new password
75  // (Ensure you have PHPMailer set up correctly)
76  $mail = new PHPMailer(true);
77  try {
78      // server config
79      $mail->isSMTP();
80      $mail->Host = 'smtp.gmail.com';
81      $mail->SMTPAuth = true;
82      $mail->Username = $config['GMAIL_USER'];
83      $mail->Password = $config['GMAIL_PASSWORD'];
84      $mail->SMTPSecure = 'ssl';
85      $mail->Port = 465;
86
87      // Recipients
88      $mail->setFrom('StarPoint.Insurance@gmail.com', 'StarPoint');
89      $mail->addAddress($data['email'], ($data['email']));
90
91      // Content
92      $mail->isHTML(true); // Set email format to HTML
93      $mail->Subject = 'Starpoint Credentials';
94      $mail->Body = '
95          <html>
96          <head>
97          <title>Welcome to StarPoint Insurance</title>
98          </head>
99          <body>
100         <h1>Welcome to StarPoint Insurance!</h1>
101         <p>Hello ' . htmlspecialchars($data['first_name']) . ',</p>
102         <p>We are excited to have you on board. Your account has been created
                ↪  successfully. Here are your initial login details:</p>
103         <p><strong>Email:</strong> ' . htmlspecialchars($data['email']) . '<
                ↪ br>
104             <strong>Password:</strong> ' . htmlspecialchars($newPassword) .
                    ↪ '</p>
105         <p>For your security, please ensure to change this password as soon
                ↪ as you log in. Remember, your password is confidential and
                ↪ should not be shared with anyone.</p>
106         <p>If you have any questions or encounter any issues, please do not
                ↪ hesitate to contact our support team.</p>
107         <p>Best regards,<br>
108         The StarPoint Insurance Team</p>
109         </body>
110         </html>
111     ';
112
113     $mail->send();
114
115 } catch (Exception $e) {
```

```
116     $error_msg = "Message could not be sent. Mailer Error: {$mail->ErrorInfo
            ↪ }";
117 }
118
119 // Output message
120 $success_msg = 'Message has been sent and Employee Created!';
```

Listing 3.28: New employee php

Similar to the leads creating and forgot password sections we handle email communication through PHPMailer. The script configures the SMTP settings, crafts a predefined email with the new employee's login details, and sends it, leveraging PHPMailer's capabilities to manage email-sending. In case of email sending failure, the script is designed to catch exceptions and provide a user-friendly error message, ensuring that the administrator is aware of any issues in the communication process. It is important to note that industry standards have moved beyond this. The current industry standard is to send the user a link to the webpage that our system recognizes.

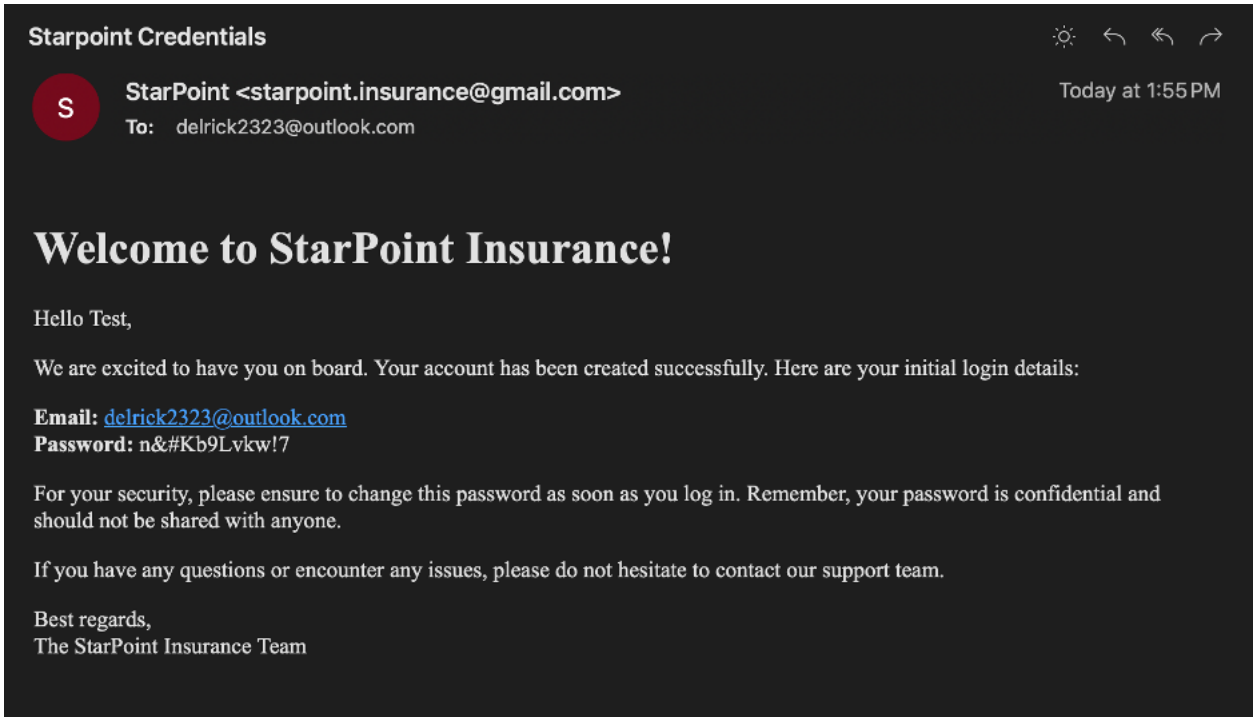

Figure 3.20: New Employee Email

3.3.7.2 Updating an Employee

In the provided Listing 3.27, starting from line 30, the script started by preparing a database query aimed at fetching all the details of an employee from the 'Employee' table, where the employee's ID matches a specific value retrieved from the query string parameter. We can use this matched ID to create a prepared statement, ensuring that the operation is securely executed to prevent SQL injection. Upon retrieving the employee data, the script checks for the existence of the fetched record. If the query returns no results, indicating that no employee exists with the specified ID, the script halts execution and outputs an error message. This safeguard ensures that the following operations are only performed on existing records, thereby maintaining data integrity. The next section of the code is triggered upon the detection of form submission. As stated in the previous section it validates any data submitted by the user before continuing.

Provided that the validation phase is completed without issues, the script advances to update the employee's record in the database. It constructs a dynamic SQL 'UPDATE' statement based on the validated data, where each piece of data is securely bound to a placeholder in the prepared statement. Following the execution of the update statement, the script re-fetches the updated details of the employee from the database, a step that confirms the successful update and allows for the immediate reflection of changes.

The execution sets a success message, signifying the completion of the update process. This feedback mechanism is for user interaction, providing clear communication regarding the outcome of their actions. By re-fetching the updated employee data, the script ensures that any subsequent interactions with the user interface present the most current information, thereby enhancing the application's responsiveness and accuracy.

3.3.8 Leads Processing and Calendar

The lead processing system serves as a bridge between the identification of potential members and their successful conversion into active members for the health care agency. It integrates the front-end interface used by agents and the back-end database that stores lead information, which allows users to view potential interest in enrollment and capture these opportunities. This system is designed for healthcare agents, allowing them to manage and engage with leads efficiently through a structured workflow.

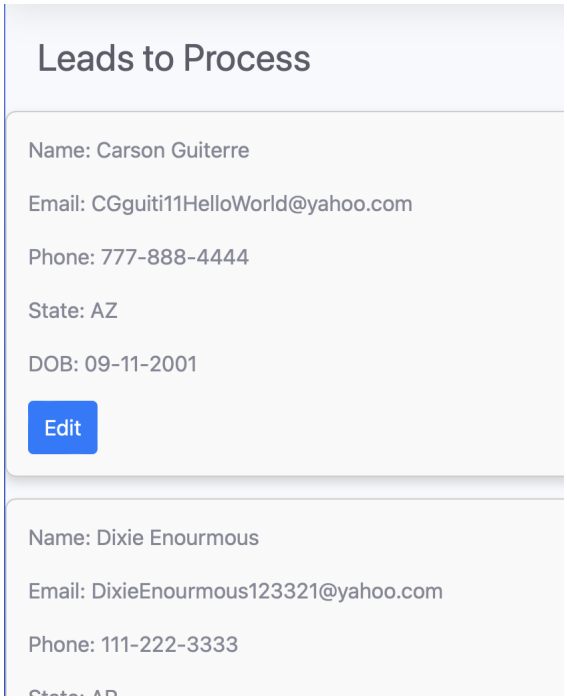

Figure 3.21: Lead process UI

Upon the identification and importation of leads into the platform, each entry is tagged with relevant information such as 'serviced'. This initial step is crucial for organizing leads in a manner that facilitates effective follow-up. The primary engagement with leads is initiated through the 'lead_process.php' page, where

agents can contact leads directly via phone or email. This direct line of communication allows for assessing the lead's interest and moving forward in the conversion process.

When a lead is successfully contacted and expresses interest in enrolling with the agency, the agent facilitates this enrollment through an external platform called Health Sherpa. In instances where a lead does not respond or wishes to be recontacted later, the agent is provided with the capability to set a recontact date. This flexibility is key to accommodating the preferences of leads and hopefully increases the likelihood of successful engagement. Following the scheduling of a follow-up, the system updates the lead's attributes to reflect their 'recontact_datetime' and 'processed' status, simultaneously generating an event in the 'event' table to integrate the follow-up into the agent's workflow.

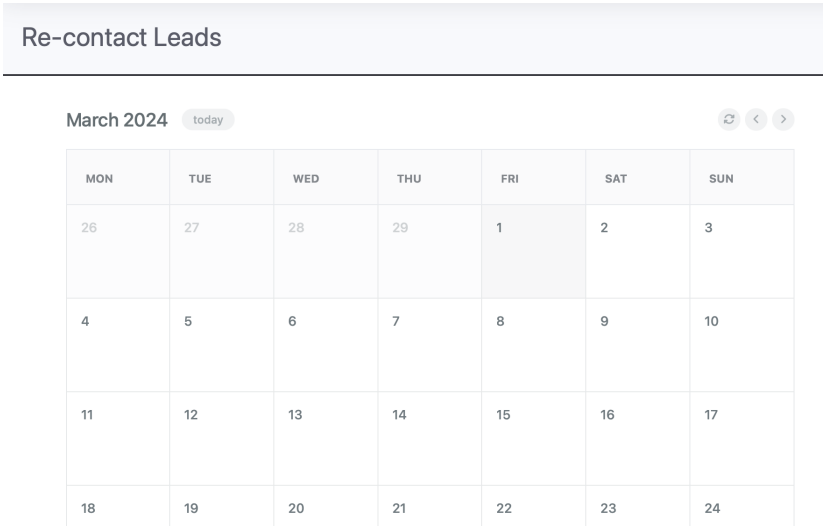

Figure 3.22: Calendar User Interface

The underlying database supports efficient data retrieval and updating using event triggers, ensuring that leads and calendar events are interconnected in a manner that enhances the lead management process. This involves careful consideration of the tables used, key attributes, and the relational aspects of the database design.

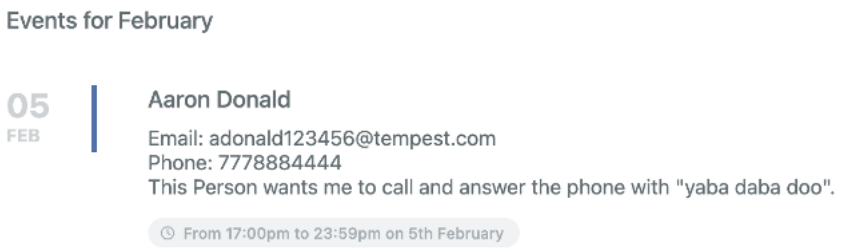

Figure 3.23: Calendar Quick Reminder User Interface

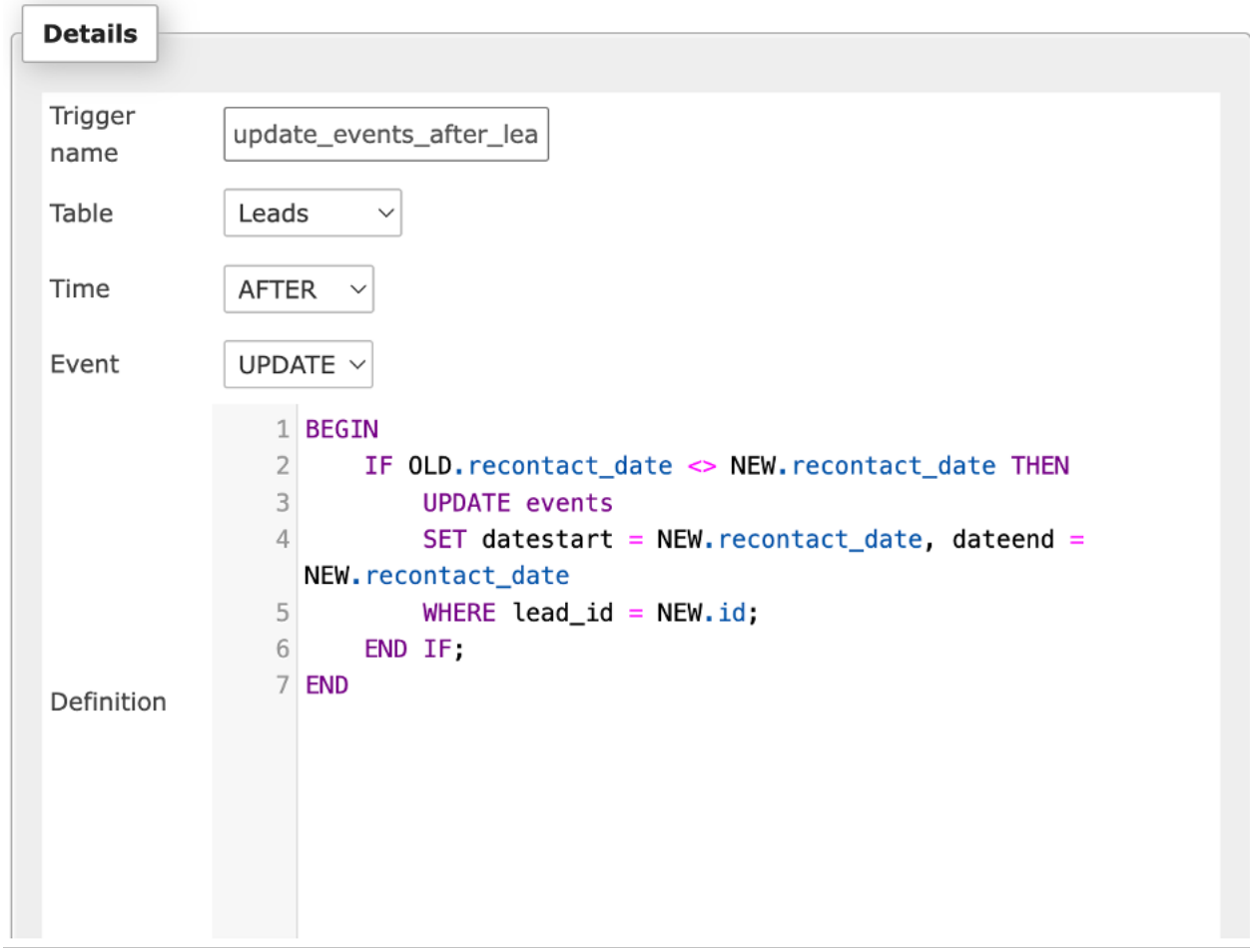

Figure 3.24: Database trigger

The trigger 'update_events_after_lead_update' is utilized to maintain the data integrity between the 'Leads' table and the 'events' table. This trigger is designed to automatically execute after an update operation on the 'Leads' table, specifically

when the ‘recontact_date‘ column is modified. Upon detecting a change in the ‘recontact_date‘ for any lead, the trigger updates the ‘events‘ table, setting both the ‘datestart‘ and ‘dateend‘ columns to match the new ‘recontact_date‘ for all events associated with the updated lead. This automated synchronization ensures that event planning remains aligned with the latest engagement strategies and contact schedules, thereby enhancing the operational efficiency of the CRM system. The trigger encapsulates complex business logic within the database layer, reducing the need for additional application logic to keep these key data elements in sync. By leveraging this trigger, the system ensures that updates to lead recontact dates are immediately reflected across related events, facilitating timely and coordinated actions based on the most current information. This not only streamlines workflow but also significantly contributes to improved customer relationship management by ensuring that all interactions are based on the most up-to-date scheduling information.

```
200 public function update_lead_on_event_change($eventId) {
201     // First, retrieve the lead_id associated with this event
202     $stmt = $this->pdo->prepare('SELECT lead_id, description, datestart FROM
            ↪  events WHERE id = ?');
203     $stmt->execute([$eventId]);
204     $event = $stmt->fetch();
205     if ($event && $event['lead_id']) {
206         // If a corresponding lead exists, update the Leads table
207         $stmtLeads = $this->pdo->prepare('UPDATE Leads SET notes = ?,
                ↪ recontact_date = ? WHERE id = ?');
208         $stmtLeads->execute([$event['description'], $event['datestart'],
                ↪ $event['lead_id']]);
209         return true;
210     }
211     // If no corresponding lead, or lead_id is not set, return false
212     return false;
```

Listing 3.29: Code Implemented Database trigger

The above PHP function, ‘update_lead_on_event_change‘, demonstrates an application-level approach to maintaining data consistency across related tables within a database, specifically targeting updates from the ‘events‘ table to the ‘Leads‘

table. This function operates by first querying the database for an event's associated lead based on the event ID. Upon retrieving the event details, including the lead ID, description, and start date, the function proceeds to update the corresponding lead's notes and recontact date if the lead exists. This update mechanism requires explicit invocation within the application code, contrasting with the automatic execution characteristic of database triggers.

In comparison to database triggers, which are embedded within the database management system and automatically respond to data modification events, this PHP function represents an alternative architectural choice. The function lives at the application level which means that any interaction with the 'events' table must use this logic to ensure data consistency. This approach contrasts with the centralized logic of database triggers, which ensures that any data modification through any path results in consistent data updates without requiring explicit calls to synchronization functions.

3.3.9 Reports

Reporting is a critical component in the management of a healthcare agency and for many other businesses. Reports often compile and analyze data concerning member demographics, enrollment figures, claims processing, and lead distribution, which are then used to propel the business further. Efficient reporting in such agencies leads to enhanced operational workflows, proactive risk management, and the development of competitive services. Within the CRM two main reports can be generated with the information provided "Expected commission reports" and "Agent Processed Leads". "Expected commission reports" serve as a financial forecast, which will cast potential revenue. By analyzing these reports, the business gains insight into expected revenue streams, enabling better cash flow management and financial planning. This predictive capability is crucial in maintaining a stable

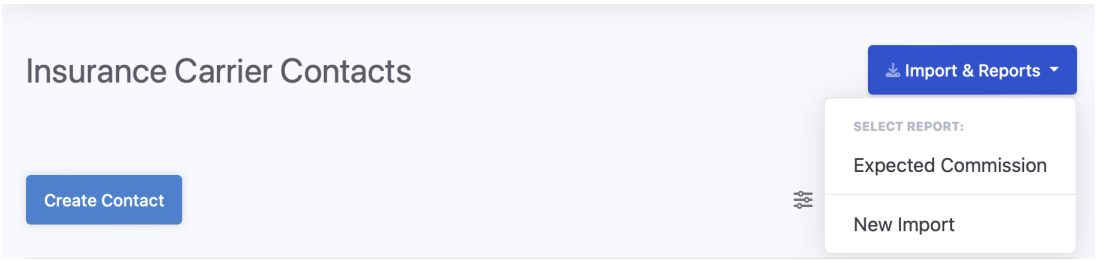

Figure 3.25: Health Insurance Carrier Header

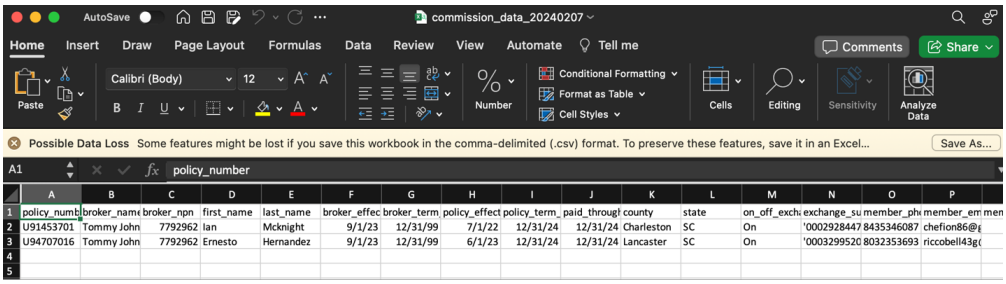

Figure 3.26: Commission Statement Report in excel

and reliable business operation, as it helps to prepare for periods of high and low revenue, ensuring that resources are allocated appropriately.

The reports are created upon submission of a date form, the script processes the user's input to construct a specific date, which is then used to query a database for relevant commission records. This operation relies on a SQL query that filters records based on the constructed date, ensuring that only applicable commission data is retrieved. The result is a comprehensive selection of records that meet the criteria of being active within the given time frame and having been paid through that date.

```
21 // Prepare SQL Query
22 $sql = "SELECT * FROM HealthCarrier WHERE
23        broker_effective_date <= :commissionDate AND
24        broker_term_date > :commissionDate AND
25        policy_effective_date <= :commissionDate AND
26        policy_term_date >= :commissionDate AND
27        paid_through_date >= :commissionDate";
```

Listing 3.30: Commission Statement SQL Query

The script exports this data into a CSV file format, a functionality that is initiated

only if the date input is valid and the query returns data. With headers that clearly define each column, the outputted CSV file provides a structured format for reviewing and analyzing the commission data. The next report “Agent processes leads” serves the company by being able to track internal agents. This report tracks the number of leads assigned to each agent when they have processed a lead. By leveraging this information, management can identify high-performing or low-performing agents and understand the methodologies that lead to conversions. In essence, the "Agent processes leads" report is not merely a tracking tool but a critical component of the company’s strategic operations. It plays a pivotal role in enhancing sales performance, optimizing marketing initiatives, and ultimately driving the company’s growth and profitability.

```
82  // Check if the 'report' query parameter is set and equals 'allLeads'
83  if(isset($_GET['report']) && $_GET['report'] == 'allLeads') {
84
85      // Prepare the SQL statement
86      $sql = "SELECT closure, team_name, COUNT(*) as total_leads, closure_date
            ↪ , closure_time FROM AgentCRM GROUP BY closure, team_name,
            ↪ closure_date, closure_time";
87      $stmt = $pdo->prepare($sql);
88
89      // Execute the statement and fetch the results
90      $stmt->execute();
91      $results = $stmt->fetchAll(PDO::FETCH_ASSOC);
92
93      // Set headers to force download
94      header('Content-Type: text/csv; charset=utf-8');
95      header('Content-Disposition: attachment; filename="AgentsCompletedLeads.
            ↪ csv"');
96
97      // Open the output stream
98      $output = fopen('php://output', 'w');
99
100     // Add column headers to the CSV
101     fputcsv($output, ['Closure', 'Team Name', 'Total Leads', 'Closure Date',
            ↪  'Closure Time']);
102
103     // Add rows to the CSV
104     foreach($results as $row) {
105         // Ensure that the 'team_name' key exists in the $row array
106         $teamName = array_key_exists('team_name', $row) ? $row['team_name'] :
                ↪  'No Team';
107
```

```
108         // Output the data to the CSV
109         fputcsv($output, [
110             $row['closure'],
111             $teamName,
112             $row['total_leads'],
113             $row['closure_date'],
114             $row['closure_time']
115         ]);
116     }
117
118     // Close the output stream
119     fclose($output);
120     exit();
121 }
```

Listing 3.31: Agent Leads Report PHP implementation

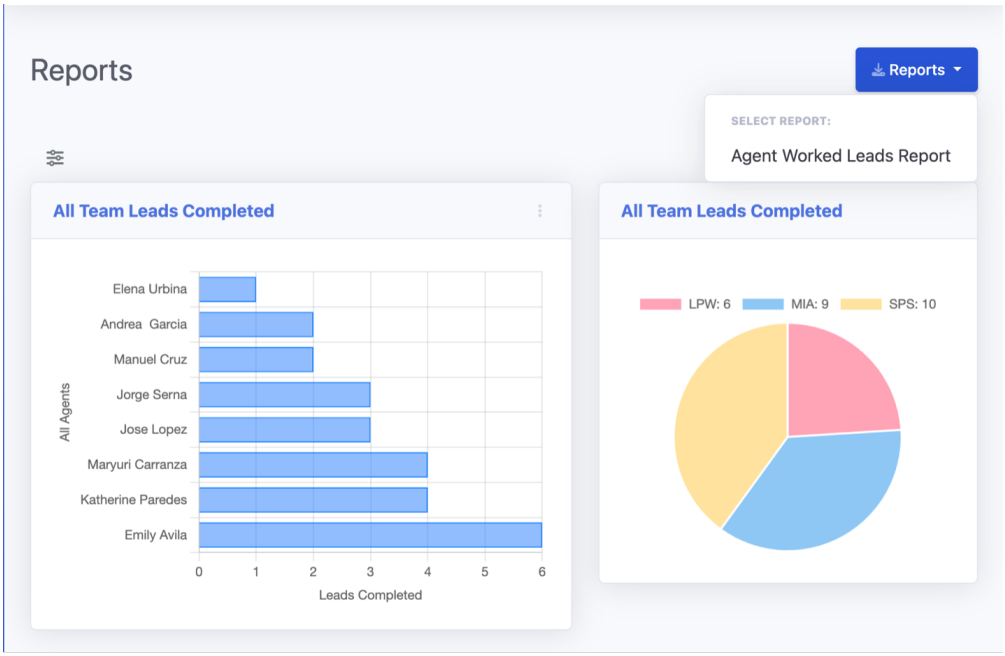

Figure 3.27: Agent Leads Report Graphs

The script checks if the query parameter 'report' is present in the GET request and whether it matches the value 'allLeads'. This conditional check determines whether the script should proceed with the report generation process. Once the condition is satisfied, the script constructs an SQL statement. This statement is designed to select and aggregate data from the 'AgentCRM' table, grouping the results by the closure status, team name, closure date, and closure time. The COUNT(*) function within

A1 Closure

	A	B	C	D	E
1	Closure	Team Name	Total Leads	Closure Date	Closure Time
2	Maryuri Carr	LPW	1	12/6/23	9:49:00
3	Andrea Garc	MIA	1	12/6/23	12:56:00
4	Emily Avila	MIA	1	12/6/23	14:04:00
5	Emily Avila	MIA	1	12/6/23	8:55:00
6	Andrea Garc	MIA	1	12/7/23	12:07:00

Figure 3.28: Agent Leads Report in Excel

the SQL query calculates the total number of leads for each group, providing a quantitative measure of the agents' performance in terms of lead processing. Finally, once all the data has been written to the CSV file, terminates the execution with an 'exit' command.

CHAPTER 4

Conclusion

This paper examined various database management systems and the theoretical aspects of the relational database model, including database architecture and normal forms. Designing an efficient customer relationship management system for an organization can be a challenging task due to multiple components that need to be put in place.

Developers must first understand the business they are working with before designing a CRM. The technical development of database management systems must align with the strategic objectives and operational requirements of the business they are designed to support. It is not enough to merely build a system that meets technical specifications; the system must also facilitate the organization's ability to manage relationships with its customers or members effectively and efficiently. The goal of this project was to create a secure, web-based Customer Relationship Management (CRM) system, through an examination of database design principles. This study has laid down a foundational framework for developing CRM systems that are secure. The research underscored the importance of choosing the right database architecture and adherence to normal forms in enhancing system performance and ensuring robust data security. By incorporating technologies such as MySQL and PHP, this project has highlighted the critical role of database design in the implementation of CRM systems. The findings from this preliminary analysis serve as a valuable resource for developers and researchers aiming to optimize

CRM systems for better performance, security, and user satisfaction. The software created is a web application that allows healthcare insurance agents to understand what leads to processes from the database. It also primarily serves as a way for management to store, edit, retrieve, and delete any data within the database. A user interface was built to provide employees or managers with an easy-to-understand structure to assist in their tasks. The system created provides primitive CRM functionalities in comparison to industry leaders such as Salesforce. However, it lays the groundwork for potential future improvements which will be discussed further in the next section.

4.1 Limitations and Improvements

During the process of designing and implementing the customer relationship manager software system, various limitations appeared, as well as methods for future improvement. The software system created a simple yet functional way of storing critical information such as new leads, members who are currently enrolled in a health insurance carrier, and managing data from an already existing CRM application called 'AgentCRM'. The major flaw of this system is that "customers" or potential members do not adhere to a "pipeline" or "workflow". In other words, a future improvement to this system would be to add a lifecycle to a lead from creation to conversion. The software currently has a way to manipulate leads, but it would be critical to give agents more control in viewing where a lead is in its "life cycle".

Another issue with the software is its scalability and adaptability for wider adoption. Should another healthcare insurance agency wish to utilize the CRM platform, it would necessitate a comprehensive overhaul of the database configuration. This

inflexibility stems from the platform's initial design as a single-tenant system, which inherently limits its ability to accommodate multiple users or organizations.

Transitioning towards a multi-tenant architecture could dramatically enhance the platform's utility, enabling it to host multiple businesses simultaneously and thus serve a broader purpose. By embracing such a model, the CRM platform could significantly increase its functionality, making it a more versatile and valuable tool for a large range of businesses not just healthcare insurance agencies. In addition, the current format of the CRM is focused on just one health insurance carrier, "Insurance Carrier". In the realm of insurance their is a plethora of different insurance carriers and it is impractical to have a CRM that is tailored to only one carrier. In the future it is imperative to have a multi-tenant architecture that allows a insurance agency to store multiple insurance carrier member data.

A significant flaw of this software is its user interface design, which negatively impacted both efficiency and the overall user experience. Currently, users must navigate through several different pages to carry out fundamental tasks like creating, updating, or deleting information from the provided tables. For instance, selecting the 'Create Contact' button on pages such as "Insurance Carrier", Agent CRM, or Leads redirects users to a separate page to input contact information. This inefficient process could be vastly improved by introducing a pop-up modal on the same page. This modal would offer the same functionalities as found on the create.php page but without necessitating disruptive page changes or loading. By adopting this approach, the software would greatly enhance usability and significantly elevate the user experience by simplifying data management tasks and making them quicker to accomplish. With the additions of the features described above, the resulting CRM that is implemented can fully replace many of the existing CRM solutions that many healthcare insurance agencies rely on. CRMs are critical for managing member interactions, streamlining operations, and enhancing the overall quality

of service. The implementation of such a comprehensive system demonstrates a significant step forward in leveraging technology to support the vital work of healthcare agencies, ensuring they have the tools necessary to deliver exceptional care efficiently and securely.

References

[1] Apps Run The World and Statista. *Customer relationship management (CRM) software market revenues worldwide from 2015 to 2026*. https://www.statista.com/statistics/605933/worldwide-customer-relationship-management-market-forecast/. Accessed: February 12, 2024. 2022 (page 3).

[2] M. Atkinson et al. *The Object-Oriented Database System Manifesto*. Accessed: 10/10/23. URL: https://grch.com.ar/docs/unlu.poo/oodbms-manifesto.pdf (page 16).

[3] K. L. Berg, T. Seymour, and R. Goel. "History Of Databases". In: *International Journal of Management & Information Systems (IJMIS)* 17.1 (2012), pp. 29–36. DOI: 10.19030/ijmis.v17i1.7587.

[4] Uma Bhat and S. Jadhav. "Moving Towards Non-Relational Databases". In: *International Journal of Computer Applications* 1.13 (2010), p. 40. URL: https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=2e81477202af88194dfa12fd4088d335905fabb6.

[5] J. Black, T. Ellis, and D. Makris. "A Hierarchical Database for Visual Surveillance Applications". In: ().

[6] D. C. Bose. *Inventory Management*. PHI Learning Pvt. Ltd., 2006.

[7] C.J. Date. *Database Design and Relational Theory: Normal Forms and All That Jazz*. Apress, 2019, p. 449. ISBN: 1484255399 (pages 19–20, 23–25).

[8] Z. Deineko et al. "Features of Database Types". In: (2021) (pages 13, 17).

[9] Docker, Inc. *Docker Compose Documentation*. https://docs.docker.com/compose/. Accessed: February 12, 2024. 2023 (page 27).

[10] K. Fraczek and M. Plechawska-Wojcik. *Comparative Analysis of Relational and Non-Relational Databases in the Context of Performance in Web Applications*. Accessed: 10/10/23. URL: https://www.researchgate.net/profile/Malgorzata-Plechawska-Wojcik/publication/314639479_Comparison_of_Relational_Document_and_Graph_Databases_in_the_Context_of_the_Web_Application_Development/links/59f5378c458515547c21d08b/Comparison-of-Relational-Document-and-Graph-Databases-in-the-Context-of-the-Web-Application-Development.pdf.

[11] C. Gyorodi et al. "Improve Query Performance On Hierarchical Data. Adjacency List Model Vs. Nested Set Model". In: *International Journal of Advanced Computer Science and Applications* 7.5 (2016). DOI: 10.14569/IJACSA.2016.070755. URL: http://thesai.org/Downloads/Volume7No5/Paper_55-Improve_Query_Performance_on_Hierarchical_Data.pdf (page 18).

[12] C. Gyorödi, R. Gyorödi, and R. Sotoc. "A Comparative Study of Relational and Non-Relational Database Models in a Web-Based Application". In: *International Journal of Engineering and Information Systems* 5.10 (Oct. 2021), pp. 73–80. ISSN: 2643-640X.

[13] harkiran78. *Top 10 Open-Source NoSQL Databases in 2020*. https://www.geeksforgeeks.org/top-10-open-source-nosql-databases-in-2020/. 2022. (Visited on 10/22/2023) (page 12).

[14] Jan L. Harrington. *Relational Database Design and Implementation*. 4th. Morgan Kaufmann, 2016.

[15] M. Hillyer. *An Introduction to Database Normalization*. Tech. rep. MySQL AB, 2003.

[16] hiteshreddy2181. *Types of NoSQL Databases*. https://www.geeksforgeeks.org/types-of-nosql-databases/. 2023. (Visited on 10/22/2023) (page 11).

[17] N. Jatana et al. "A survey and comparison of relational and non-relational database". In: *International Journal of Engineering Research & Technology* 1.6 (2012), pp. 1–5.

[18] Jeff Johnson. *Designing with the Mind in Mind*. 2nd. Morgan Kaufmann, 2013. URL: https://learning.oreilly.com/library/view/designing-with-the/9780124079144/xhtml/CHP007.html (pages 34, 37).

[19] Atul Kahate. Pearson Education, 2004 (pages 6–8).

[20] S. Kantamneni. *User Experience Design*. Wiley, 2022.

[21] W. Kent. "A Simple Guide to Five Normal Forms in Relational Database Theory". In: *Communications of the ACM* 26.2 (1983), pp. 120–125 (pages 21, 23–24).

[22] David Kroenke et al. *Database Concepts*. 9th. Pearson, 2019 (page 6).

[23] T. F. Lee, J. G. Wang, and Y. C. Chen. "Review of Hierarchical Database Access Control for E-medicine Systems". In: *Tzu Chi Medical Journal* 35.2 (2022), pp. 143–147. DOI: 10.4103/tcmj.tcmj_124_22.

[24] A. McCain. *23 KEY CRM STATISTICS [2023]: GROWTH, REVENUE + ADOPTION RATES*. Accessed: 09/10/23. Jan. 2023. URL: https://www.zippia.com/advice/crm-statistics/ (pages 2–3).

[25] Ostezer and Mark Drake. *SQLite vs MySQL vs PostgreSQL: A Comparison Of Relational Database Management Systems*. https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems. Updated on March 9, 2022. 2014. (Visited on 10/22/2023) (page 8).

[26] J. Paredaens et al. *The Structure of the Relational Database Model*. Springer Science & Business Media, 2012.

[27] A. Specchia. *Customer Relationship Management (CRM) for Medium and Small Enterprises*. Productivity Press, 2022 (pages 1, 3, 10).

[28] C. Strauch. *NoSQL Databases*. Computer Science and Media (CSM), Hochschule der Medien, Stuttgart. Lecturer: Prof. Walter Kriha. Accessed: 10/10/23. URL: https://www.researchgate.net/profile/Jesus-Sanchez-Cuadrado/publication/257491810_A_repository_for_scalable_model_management/links/568baf0508ae051f9afc5857/A-repository-for-scalable-model-management.pdf (pages 11–12).

[29] C. Strauch. *NoSQL Databases [Lecture notes]*. Hochschule der Medien, Stuttgart.

[30] Larry Ullman. *PHP and MySQL for Dynamic Web Sites: Visual QuickPro Guide*. 5th. Peachpit Press, 2017. URL: https://learning.oreilly.com/library/view/php-and-mysql/9780134301945/cover.xhtml.

[31] D. L. Wells, J. A. Blakeley, and C. W. Thompson. *Architecture of an Open Object-Oriented Database Management System*. Accessed: 10/10/23. 1992. URL: https://www.researchgate.net/profile/Craig-Thompson-17/publication/2954132_Architecture_of_an_Open_Object-Oriented_Database_Management_System/links/5740a1ce08ae9ace8416086d/Architecture-of-an-Open-Object-Oriented-Database-Management-System.pdf (pages 15–16).