Customer Support Web Application for Tunisie Telecom Clients

Project Report in Web Services
As a partial fulfillment of the IT325 Web Services course

BY
Asma Boubaker

Professor
Montassar Ben Messaoud

Academic Year
2024 - 2025

Abstract

This project focuses on developing a web application specifically designed for Tunisia Telecom clients to address challenges related to customer support and mobile usage management. The application allows users to share their experiences and problems, creating a community-driven support platform. Additionally, a novel feature was implemented, utilizing machine learning algorithms to predict users' future data usage and balance based on their historical usage data. This feature helps clients understand their future needs, offering personalized recommendations on how much to recharge and how to avoid running out of data, calls, or wasting money on unused subscriptions. By providing these predictive insights, the application aims to optimize the customer experience, making it easier for users to manage their mobile services proactively. This project introduces an innovative solution to customer support applications by integrating predictive analytics, which enhances the overall user experience for Tunisia Telecom clients.

Acknowledgments

I would like to express my sincere gratitude to all those who have supported me throughout the course of this project.

I am deeply grateful to my supervisor, Dr. Montassar Ben Messaoud, for his invaluable guidance in conducting the course and for providing a sample project that helped shape our understanding of the web service structure. This was essential in guiding the direction of the actual project.

I would like to extend a special thanks to Abdelhamid Jbara, a master's student at the University of Debrecen, Hungary, for his assistance with shaping the predictive modeling aspects of the project. His expertise in machine learning was instrumental in refining the approach to the predictions and making the feature more robust.

I would also like to thank my colleague, Kenza Bacha, for her collaborative efforts throughout the project. Her continuous support, exchange of ideas, and problem solving allowed us to complete the project within a tight timeframe.

Finally, I am especially grateful to my family for their unwavering support, encouragement, and patience throughout this journey. Their belief in me provided the motivation to complete this project.

I truly appreciate the contributions of everyone who played a role in the success of this project.

Contents

1 List of Figures

2 List of Tables

3 Introduction

This project aims to create a client support web application that outperforms existing solutions by providing a seamless user experience and enabling clients to make the best purchasing decisions through accurate predictions of their future usage. By integrating machine learning techniques, the application introduces a unique feature not commonly found in customer support services in Tunisia, enhancing both customer satisfaction and efficiency. This version focuses primarily on developing the idea and concept rather than delving deeply into the technical complexities of the web application.

The report is structured as follows: it begins with an overview and scope of the project, followed by a detailed discussion of the design and architecture. Next, it covers the implementation process, an analysis of the potential threats to validity, and concludes with the findings and recommendations.

This report will provide insights into the challenges faced by Tunisia Telecom customers and demonstrate how the developed application addresses these issues through innovative features and predictive analytics. Additionally, it presents a novel approach to customer support tailored specifically to the unique needs of clients in Tunisia.

To better understand the context and objectives of the project, the next section will provide an overview and scope, laying the foundation for the technical and implementation details that follow.

4 Context Scope and Overview

4.1 Context of the Project:

Tunisie Telecom[1] is a leading telecom operator in Tunisia, serves over 6 million subscribers across fixed and mobile services.

Despite its large customer base, Tunisie Telecom faces challenges such as unclear recharge guidance, unresolved customer issues, and incomplete agency location data on platforms like Google Maps.

With a 28.99% market share in Tunisia's mobile subscriptions as of August 2024 [2], improving customer support is vital to enhancing satisfaction and maintaining competitiveness in the local telecom sector.

Stakeholders:

Customers: Will benefit from better support and guidance.

Tunisie Telecom: Gains increased customer loyalty.

Developers: Create innovative solutions tailored to the local market.

By addressing these challenges, the project seeks to enhance the overall experience for Tunisie Telecom's users.

4.2 Scope of the Project:

This project aims to improve client support and usage predictions, focusing on concept development rather than technical complexities in its first version. With a focus on Tunisia, where customer support in the telecom industry needs improvement, it addresses key issues in Tunisia Telecom's services without delving into full technical details.

4.3 Originality and Opportunities:

This project introduces predictive analytics and personalized usage recommendations, a unique feature in Tunisia's telecom industry. Using historical data, the system forecasts future usage, helping users manage recharges and subscriptions. It also includes a tool to locate the nearest agency via IP address, addressing the challenges of limited agency presence and inconsistent location data. By improving customer experience, this app could set a new standard for telecom support in Tunisia, benefiting both companies and clients.

4.4 Research on Similar APIs:

Existing Solutions: Internationally, customer support platforms like Zendesk[3] and Freshdesk[4] provide ticketing systems and AI-driven support, but they do not focus heavily on predictive

[1] https://en.wikipedia.org/wiki/Tunisie_Telecom

[2] https://www.telecomtrustconsulting.com/market-analysis-in-tunisia-more-than-one-million-lost/?utm_source=chatgpt.com

[3] https://www.zendesk.com

[4] https://freshdesk.com

analytics or personalized usage recommendations. Some applications, like T-Mobile's[5] customer service app, offer basic usage tracking, yet still lack tailored recommendations based on the users' future needs.

Local Examples: In Tunisia, telecom support apps are relatively underdeveloped. The market lacks an app offering predictive insights, personalized recharge recommendations, and accurate agency location identification—features that are common in more advanced telecom markets.

Comparative Analysis: While international apps excel at providing customer support features, they often fail to integrate predictive analysis or geo-location features, especially tailored to specific regional needs. This project fills that gap by introducing local relevance with personalized recommendations and the ability to help users find nearby service locations, something that is currently missing in the Tunisian telecom sector.

4.5 Chapter Summary:

This chapter explored the project's scope, originality, and opportunities, along with a review of similar APIs and solutions. The next chapter will cover the design and architecture of the web application.

[5] `https://www.t-mobile.com`

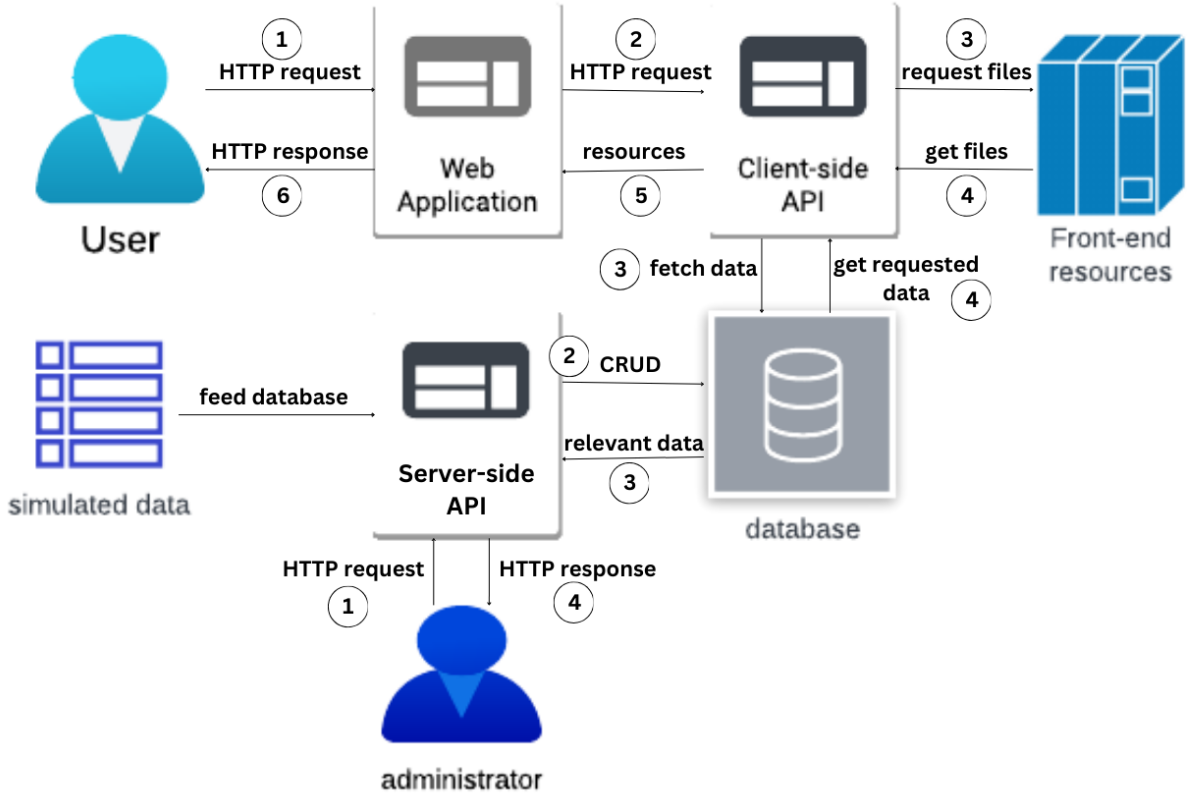

Figure 1: System Flow

5 Contribution and Design

5.1 Overview of the System Design:

5.1.1 Purpose and Components:

1. **Client-Side API:**

   Provides users with an interface to interact through routes like login, signup, home, questions, predictions, agency finder, and logout. It manages the presentation layer and converts user actions into meaningful data interactions. For more on client-side interface design, see [1].

2. **Server-Side API:**

   Serves as the system's backbone, providing endpoints for managing user data, balances, usage history, recharge history, subscription plans, and agency locations. It supports CRUD operations for questions and answers in the knowledge platform. RESTful API principles are discussed in [2].

3. **Database:**

Stores critical system data, including user credentials, balances, usage history, recharge records, questions, answers, and agency details. Simulated data is used in place of real Tunisie Telecom data. See [3] for data simulation methods.

4. **Simulated Data:**
   Simulated data mimics real-world data for testing and predictions, ensuring functionality for predictive analytics and account validation. See [4] for predictive modeling principles.

5.1.2 System Flow

The following describes the overall system flow according to the diagram, with step numbers indicated in parentheses.

1. **User Interactions**
   (a) **User Access**: The user accesses the web application (1) and interacts with various routes such as login, signup, or home through the client-side API (2).
   (b) **Server Requests via Client-Side Web Service**: User actions trigger HTTP requests to the client-side API (2), which processes these requests and fetches necessary frontend resources (3), such as balances or question submissions.
   (c) **Database Operations**: The client-side API interacts with the database (3) to retrieve or update relevant data (4), including user counts, questions, answers, balances, etc.
   (d) **Response Delivery**: Once the data is processed, it is sent back to the client-side API (5), which formats and delivers it to the user interface for display (6).
2. **Admin Interactions**
   (a) **Admin Access**: The admin directly accesses the server-side API (1) to perform CRUD operations on various system components.
   (b) **Database Operations**: The server-side API (2) interacts with the database (3) to retrieve or update data such as user balances, recharge history, agency details...
   (c) **Response Delivery**: After processing the requested operations, the server-side API sends the result back to the admin (3), indicating success or failure.
3. **Simulated Data Integration:** Simulated data is added to the database via CRUD operations by the admin, enabling predictive analytics and account validation features, as shown in the diagram.

5.2 Architectural Design

The system follows a **client-server model** with **RESTful APIs** to ensure effective and scalable communication between the client and the server. The architecture is divided into two primary components: the **client-side API** and the **server-side API**, each serving distinct roles within the system.

5.2.1 Client-Side API Functionalities

The **client-side API** manages user interactions with the web app, offering routes for **login** and **signup**. Users must have a valid phone number to sign up, with account creation denied if the number is absent. Simulated data is used due to the lack of real Tunisie Telecom data.

After logging in, users access the **home page**, where they view their balances and expiration dates. The **predict button** displays 24-hour predictions for usage, recharge needs, and projected balances, along with recommended subscription plans and statistical insights. Predictive insights are detailed in [5].

The **questions route** allows users to submit and view questions and answers. On the **find agency page**, users can locate nearby Tunisie Telecom agencies using their IP address.

Lastly, users can log out, ending their session and returning to the login page.

5.2.2 Server-Side API Functionalities

The **server-side API** is used by the administrator to manage the system through **RESTful endpoints** for **CRUD operations** on data such as user accounts, balances, recharge history, subscription plans, and agency locations.

It ensures system integrity by managing user data and balances, and supports the **knowledge-sharing platform** for administering questions and answers. Additionally, the server-side API handles the insertion of **simulated data** for testing and running **predictive analytics** like usage predictions and recharge forecasts.

5.2.3 Database

The database stores user credentials, balances, usage history, recharge history, questions, answers, and agency locations. Simulated data is used for testing and enabling predictive functionalities due to the lack of real data from Tunisie Telecom. It is optimized for handling both small and large data volumes, ensuring efficient CRUD operations by both APIs.

5.2.4 Architectural Considerations

The system is designed to be **scalable**, **maintainable**, and **usable**:

Scalability: The modular architecture supports independent scaling of client and server components, maintaining performance under higher demand.

Maintainability: Clear separation between client and server APIs simplifies updates and maintenance.

Usability: An intuitive user interface ensures easy navigation, while the admin interface enables efficient management through CRUD operations.

The modular design supports future updates and adaptability.

5.3 API Overview and Methodology:

5.3.1 API Endpoint Overview:

Client-Side Endpoints: These endpoints manage routes that interact directly with clients, providing features like user authentication, account management, and predictions. Acting

Endpoint	HTTP Method	Description
`/login`	GET, POST	Handles user login by verifying credentials and starting a user session.
`/signup`	GET, POST	Facilitates user registration and account creation.
`/logout`	GET	Logs the user out and redirects them to the login page.
`/`	GET	Displays the home page with the user's current account balance.
`/predict`	POST	Predicts future usage patterns and recommends suitable plans for the user.
`/questions`	GET, POST	Displays a list of questions or allows users to submit new questions.
`/question/<id>`	GET, POST	Displays and handles answers to a specific question based on its ID.
`/find`	GET, POST	Locates the closest agency to the user's given location coordinates.

Table 1: Client-Side API Endpoints

as a bridge between the front-end and back-end, they ensure seamless user interactions. By handling tasks like credential validation, session management, and data processing, these routes enable efficient front-end functionality. The methods for each endpoint are designed to provide accurate, timely responses. Detailed descriptions of the client-side endpoints and their HTTP methods are in **Table 1 for client-side API endpoints**.

Server-Side Endpoints: These endpoints handle system administration and database operations, ensuring efficiency and data consistency. They allow administrators to manage user accounts, data storage, and system monitoring. **Table 2 for server-side API endpoints** outlines the functionalities available to the admin for effective system management.

5.3.2 Front-End Handling :

The front-end interacts with the API through a simple interface, ensuring that the user can easily access the necessary functionalities while maintaining a clear and efficient design.
Initial User Access: Upon opening the application, users are initially presented with two pages: the **login** page to access their account and the **signup** page to create a new account. These pages interact with the corresponding API endpoints as described in **Table 1 for client-side API endpoints.** Users cannot access any additional features without logging in.
Post-Login Pages: After logging in, the user can access the following pages:
Home Page: Displays the user's current balance via the API.
Prediction Page: Shows the user's predictions, retrieved through the API endpoint.
Questions Page: Allows users to submit, search, and delete questions, using the relevant API methods.

Endpoint	HTTP Method	Description
/api/users	GET, POST, DELETE	Retrieve, add, or delete all users in the system.
/api/users/<int:user_id>	GET, DELETE	Retrieve or delete a specific user by their ID.
/api/usageHistory	GET, POST, DELETE	Retrieve, add, or delete all usage history records.
/api/usageHistory/<int:id>	GET, DELETE	Retrieve or delete a specific usage history record by ID.
/api/usageHistory/user/<int:id>	GET	Retrieve usage history for a specific user by their ID.
/api/balances	GET, POST, DELETE	Retrieve, add, or delete all balance records.
/api/balances/<int:id>	GET, DELETE	Retrieve or delete a specific balance record by ID.
/api/balances/user/<int:id>	GET, PATCH, DELETE	Retrieve, update, or delete balance records for a specific user by ID.
/api/recharges	GET, POST, DELETE	Retrieve, add, or delete all recharge records.
/api/recharges/<int:id>	GET, DELETE	Retrieve or delete a specific recharge record by ID.
/api/recharges/user/<int:id>	GET	Retrieve all recharge records for a specific user by their ID.
/api/monetaryPlans	GET, POST, DELETE	Retrieve, add, or delete all monetary recharge plans.
/api/monetaryPlans/<int:id>	GET, DELETE	Retrieve or delete a specific monetary recharge plan by ID.
/api/mobileDataPlans	GET, POST, DELETE	Retrieve, add, or delete all mobile data plans.
/api/mobileDataPlans/<int:id>	GET, DELETE	Retrieve or delete a specific mobile data plan by ID.
/api/agencyLocations	GET, POST, DELETE	Retrieve, add, or delete all agency location records.
/api/agencyLocations/<int:id>	GET, DELETE	Retrieve or delete a specific agency location by ID.
/api/questions	GET, DELETE	Retrieve or delete all questions submitted by users.
/api/questions/user/<int:id>	GET	Retrieve all questions submitted by a specific user.
/api/questions/<int:id>	DELETE	Delete a specific question by ID.
/api/answers	GET, DELETE	Retrieve or delete all answers submitted by users.
/api/answers/question/<int:id>	GET	Retrieve all answers for a specific question by ID.
/api/answers/user/<int:id>	GET	Retrieve all answers submitted by a specific user by ID.

Table 2: Server-Side API Endpoints

Answers Page: Accessible by clicking a question to view and add answers.

Find Agency Page: Displays agency locations on a map, marking the user's location and nearest agency when the "Find Nearest Agency" button is clicked.

Logout Button: Logs the user out and redirects them to the login page.

Each page ensures ease of use by making API calls, processing responses, and rendering the data for smooth interaction.

5.3.3 Reusability and Integration of APIs:

The system integrates external APIs for map and user location functionalities. A map API handles map features, while an IP-based location API retrieves the user's address. Using these pre-built services improves functionality, simplifies implementation, and reduces redundancy. For more on API integration, see [2] and [6].

5.3.4 Error Handling

Error handling is implemented at both the **client-side API** and **server-side API** based on their respective operations.
Client-Side API: Errors arise from user actions and are managed using **flash messages** to guide users. Common errors include:

- Logging in with invalid credentials.
- Signing up with a non-existent phone number or weak password.
- Submitting short questions or searching for nonexistent ones.

These are displayed as flash messages for corrective action. Internal errors are silently handled to maintain stability. For client-side error handling principles, see [1].
Server-Side API: Errors involve data validation and are handled through **HTTP error responses**. Common issues include:

- Incomplete or incorrect POST data.
- Invalid or nonexistent IDs during data retrieval, update, or deletion.

These errors return HTTP status codes (e.g., `400 Bad Request`, `404 Not Found`, or `500 Internal Server Error`) to indicate the issue. For API error handling strategies, refer to [7].

5.4 Database Design:

5.4.1 Schema Design:

The database schema consists of interconnected tables designed to manage user data, balances, usage history, and other key information. Important tables include:

- **User**: Stores details like phone number, username, password, and bonus plans, linked to `UsageHistory`, `Balance`, `Recharge`, `Question`, and `Answer`.
- **UsageHistory**: Tracks hourly user activity, including calls, SMS, and data usage.

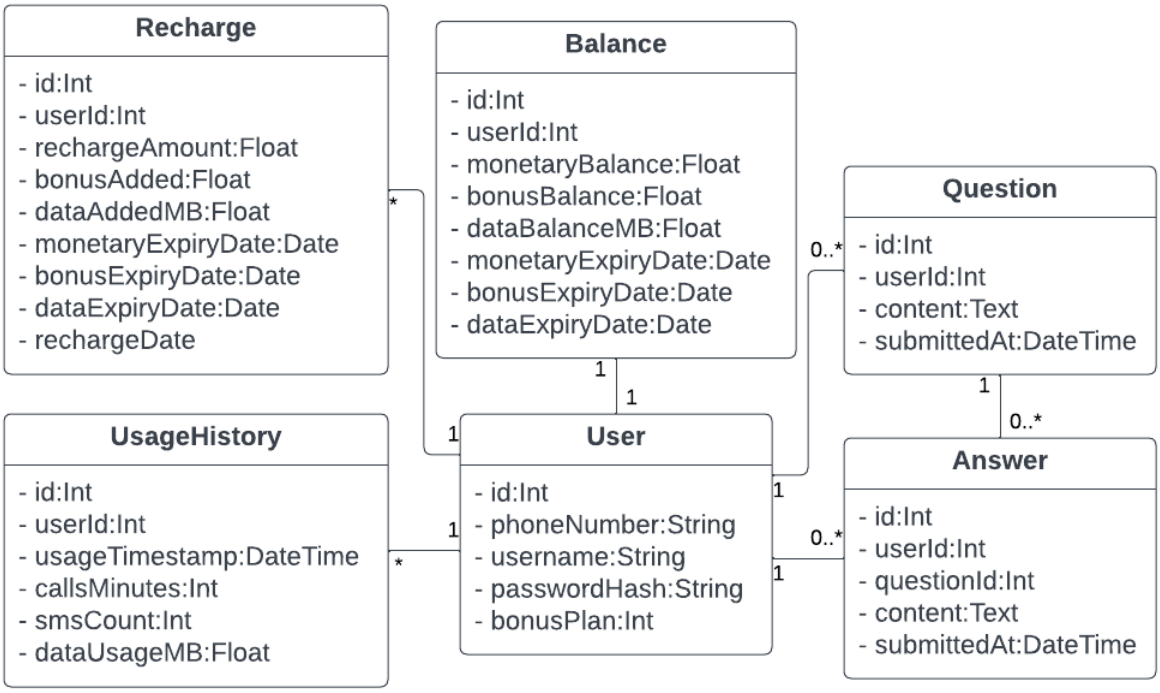

Figure 2: Class diagram for user interactions

- **Balance**: Manages monetary, bonus, and data balances, along with expiration dates.
- **Recharge**: Records recharge details, amounts, bonuses, and expiration dates.
- **MonetaryRechargePlan** and **MobileDataPlan**: Store predefined recharge plans for monetary and data services.
- **Question** and **Answer**: Support the knowledge-sharing platform with user-submitted questions and answers.
- **AgencyLocation**: Stores Tunisie Telecom agency information, including addresses and coordinates.

The schema supports scalability for future real-world data integration and modularity for feature additions. It ensures data consistency through foreign key relationships and cascade rules [8]. A class diagram in **Figure 2** illustrates the relationships within the database schema.

5.4.2 Data Collection:

The data used in the system is simulated due to the unavailability of real data from Tunisie Telecom. Key data collection challenges and approaches include:

- **User Data**: The system requires phone numbers from the company's database to create accounts. Since this data was unavailable, 10 users were manually created with randomly assigned phone numbers and bonus plans to simulate real users. New user accounts are added dynamically through the web application.

- **Usage History and Recharge Data**: These datasets are crucial for predictions but were unavailable. Simulated data was generated, ensuring logical constraints (e.g., usage decreases balances, recharges increase balances). This approach ensures data consistency for the prediction model while remaining artificial. Refer to [10] for techniques on predictive modeling.

- **Subscription Plans**: Tunisie Telecom's official website and APIs are inaccessible for automated retrieval. A set of monetary and data plans was manually created to replicate this functionality.

- **Agency Locations**: Geographic data for over 120 Tunisie Telecom agencies was manually collected from the company's website and cross-checked with Google Maps for accuracy [9].

- **Questions and Answers**: This data is collected dynamically within the system as users interact with the knowledge-sharing platform.

Despite limitations in obtaining real-world data, the system's schema and data collection approach ensure functionality and scalability, supporting future integration with live data sources.

5.5 Chapter Summary:

This chapter outlined the system's design and architecture, detailing the client-server model, API methodology, and database schema. Key components such as the client-side and server-side APIs, database, and simulated data were described alongside error-handling strategies and front-end interactions. The database design emphasized scalability and data collection challenges, highlighting solutions with simulated data.

With the design and architecture established, the next chapter delves into the technical implementation of the system.

6 Implementation

6.1 Setup and Application Construction

6.1.1 Folder Structure

The project is organized into a well-structured folder hierarchy to ensure clarity and maintainability. The structure is as follows:

- **configuration/**: Main folder for project configuration.
  - **dataSimulation/**: Contains files for simulated data in `.json` format and a script (`simulate.py`) for generating usage, recharge, and balance data.
  - **static/**: Includes the CSS file (`style.css`) for styling the web pages.
  - **templates/**: Contains HTML templates for the front-end, including:
    - `answers.html`
    - `findAgency.html`
    - `home.html`
    - `login.html`
    - `prediction.html`
    - `questions.html`
    - `signup.html`
    - `theme.html`
  - **__init__.py**: Initializes the project as a Python module.
  - **apiRoutes.py**: Contains server-side API routes.
  - **blueprint.py**: Defines the Flask Blueprint for modular route handling.
  - **dbInitialization.py**: Handles database setup and initialization.
  - **models.py**: Defines the database schema and models using SQLAlchemy.
- **.flaskenv**: Stores environment variables for the Flask application.
- **app.py**: The main entry point for running the application.

6.1.2 Libraries and Frameworks

The implementation uses several libraries essential for the application's functionality:
Flask[6], **Flask-Login**[7], **Flask-SQLAlchemy**[8], **SQLAlchemy**[9], **Flask-Smorest**[10], **Werkzeug**[11],

[6] `https://flask.palletsprojects.com/`
[7] `https://flask-login.readthedocs.io/`
[8] `https://flask-sqlalchemy.palletsprojects.com/`
[9] `https://www.sqlalchemy.org/`
[10] `https://flask-smorest.readthedocs.io/`
[11] `https://werkzeug.palletsprojects.com/`

Scikit-learn[12], **Pandas**[13], **NumPy**[14], **Geopy**[15].

These libraries enable the setup and smooth functioning of the project.

6.2 Database Creation and Data Simulation

6.2.1 Database Creation

The database schema is defined using SQLAlchemy models, which map the structure of the database and relationships between different entities.

User Model: The `User` model represents each user in the system. It includes fields such as `id` (primary key), `phoneNumber` (unique, non-nullable), `username` (nullable), `passwordHash` (nullable), and `bonusPlan` (non-nullable, integer). It establishes relationships with other models: one-to-one with `Balance` and one-to-many with `UsageHistory`, `Recharge`, `Question`, and `Answer`.

Balance Model: The `Balance` model stores each user's available balances, including `monetaryBalance`, `bonusBalance`, and `dataBalanceMB`, along with expiration dates for each balance type (`monetaryExpiryDate`, `bonusExpiryDate`, `dataExpiryDate`). It is linked to the `User` model via a one-to-one relationship.

UsageHistory Model: The `UsageHistory` table tracks each user's usage, including `usageTimestamp`, `callsMinutes`, `smsCount`, and `dataUsageMB`. It establishes a many-to-one relationship with the `User` model.

Recharge Model: The `Recharge` model tracks recharge with details like `rechargeAmount`, `bonusAdded`, `dataAddedMB`, and expiration dates for the recharge (`monetaryExpiryDate`, `bonusExpiryDate`, `dataExpiryDate`). Each recharge is linked to a specific user via a many-to-one relationship.

Question and Answer Models: The `Question` model stores questions submitted by users, with fields like `id` (primary key), `content` (the content of the question), and `submittedAt` (timestamp). Each `Question` can have multiple `Answers`, and each `Answer` corresponds to only one `Question`, forming a one-to-many relationship. The `Answer` model stores `content` (answer text), `submittedAt` (timestamp), and `questionId` (foreign key referencing the associated `Question`).

AgencyLocation Model: The `AgencyLocation` model stores details for each agency, including `name`, `address`, `phoneNumber`, `latitude`, and `longitude` (geographical coordinates).

6.2.2 Data Simulation

The data simulation process generates realistic data for users, recharge events, usage history, and balances using probabilistic distributions.

User Data Generation: Users are assigned random phone numbers with varying `bonusPlan` values, and initial balances (monetary, bonus, data) are set to 0.

[12] `https://scikit-learn.org/`

[13] `https://pandas.pydata.org/`

[14] `https://numpy.org/`

[15] `https://geopy.readthedocs.io/`

Recharge Simulation: Recharge events are generated based on Poisson distribution. Monetary recharges use random plans, updating `rechargeAmount`, `bonusAdded`, and `dataAddedMB`. Data recharges sample `dataAmountMB` from a normal distribution based on expected usage, with expiration dates added.
Usage Simulation: Usage history is simulated with random values for `callsMinutes`, `smsCount`, and `dataUsageMB`. The balance is decremented based on usage costs, ensuring sufficient balance before usage.
Data Consistency Validation: Consistency checks ensure recharges update balances, usage does not exceed available funds, and data is logically valid.

Simulated data is saved in JSON files for easy system integration and testing.

6.3 Front-End Implementation

The front-end is built with HTML, CSS, JavaScript, and **Flask's Jinja**[16]. The base theme, `theme.html`, includes the navigation bar, flash message handling, and Bootstrap for a responsive interface.

Each page (login, signup, home, prediction, questions, agency finder) extends `theme.html`, inheriting its structure and styling, with page-specific content and functionality. A dedicated CSS file customizes the style, enhancing usability.

6.4 Implementation of Client-Side API Endpoints

This section covers the key client-side API endpoints, their implementation, and their role in handling user interactions.

6.4.1 Authentication Endpoints

The `/login` endpoint authenticates users, redirecting to the home page on success or showing errors on failure. The `/signup` endpoint validates inputs, registers, and logs in users, redirecting them to the home page. The `/logout` endpoint logs users out and redirects to the login page.

6.4.2 User-Specific Operations

The `/` endpoint fetches and displays the user's balance. The `/predict` endpoint estimates usage and recharge using models like Linear Regression and Random Forest, offering plan recommendations. The `/questions` endpoint allows users to submit and search questions. The `/question/int:question_id` endpoint retrieves and saves answers.

6.4.3 Find Agency

The `/find` endpoint uses geolocation to find the nearest agency, calculating distances based on latitude and longitude.

[16] **Flask** is a lightweight Python web framework, and **Jinja** is its templating engine, enabling dynamic HTML generation.

6.4.4 Front-End Response Handling

Endpoints return JSON objects (e.g., agency locations) for dynamic front-end updates. Flash messages provide immediate feedback on actions like login or form submissions.

6.5 Implementation of Server-Side API Endpoints

This section describes the server-side API endpoints for managing users, balances, and usage history, using Flask and SQLAlchemy for efficient database interaction.

6.5.1 Users Endpoints

The Users endpoint (`/api/users`) supports GET, POST, and DELETE methods. GET retrieves all users, POST adds new users with unique data, and DELETE clears all users. Specific user data can be accessed or deleted via `/api/users/int:user_id`. Each user record includes `id`, `phoneNumber`, `username`, and `bonusPlan`.

6.5.2 Usage History Endpoints

The Usage History endpoint (`/api/usageHistory`) supports retrieving, adding, or deleting usage records, which include `userId`, `usageTimestamp`, `callsMinutes`, `smsCount`, and `dataUsageMB`. Usage data can be queried by user via `/api/usageHistory/user/int:user_id`.

6.5.3 Balances Endpoints

The Balances endpoint (`/api/balances`) retrieves, adds, or deletes balances linked to a user via `userId`. Balances include `monetaryBalance`, `bonusBalance`, and `expiryDates`. A user's balance can be managed via `/api/balances/user/int:user_id`.

6.5.4 Recharge Endpoints

The Recharges endpoint (`/api/recharges`) manages recharge records with methods to retrieve, add, or delete entries. Each record includes `userId` and recharge details like monetary and bonus. Specific user recharges are accessible via `/api/recharges/user/int:user_id`.

6.5.5 Recharge Plans Endpoints

Recharge plans, including Monetary Plans (`/api/monetaryPlans`) and Mobile Data Plans (`/api/mobileDataPlans`), are managed with GET, POST, and DELETE methods. Each plan, identified by `plan_id`, defines pricing, recharge amounts, and expiry dates.

6.5.6 Agency Locations Endpoints

The Agency Locations endpoint (`/api/agencyLocations`) manages location data for agency offices. Locations are retrieved, added, or deleted, with details such as name, address, and coordinates. Individual locations are accessed via `/api/agencyLocations/int:location_id`.

6.5.7 Question and Answer Endpoints

The Questions endpoint (`/api/questions`) manages user questions (retrieve, add, delete). Specific questions are accessed via `question_id`. Answers are handled through the Answers endpoint (`/api/answers`), with methods to retrieve by question or user.

6.6 External API Implementation

The "Find Closest Agency" feature integrates the Geolocation API to retrieve the user's latitude and longitude, which is then sent to the server for processing. The server uses this data to calculate the nearest agency from a list of agencies stored in the database. The Leaflet.js library is employed for rendering the map and placing markers for each agency. The user's location is marked with a distinct icon, and the closest agency is highlighted.

This approach relies on the browser's navigator.geolocation API for real-time location data, while Leaflet.js is used to visually display the agencies and user's position on an interactive map.

6.7 Debugging and Testing

The development process involved comprehensive debugging and testing to ensure system functionality and reliability. **Flask's built-in debugging mode**[17] was a key tool, providing real-time error feedback that helped trace and resolve issues efficiently. Its error logs were instrumental in pinpointing problems and verifying corrections.

In addition to Flask, **Insomnia REST client**[18] was used to test the API endpoints. Insomnia offered a user-friendly interface to send HTTP requests and inspect responses, ensuring endpoints handled both valid and invalid inputs appropriately. The **browser** was also crucial for testing client-side API interactions, allowing direct observation of rendered outputs and verifying front-end and back-end communication.

Unit testing was conducted to confirm API functionality under various scenarios, including edge cases. This ensured database queries integrated seamlessly and returned or modified the expected data. Bugs were identified through error logs, manual testing, and iterative refinement, with Flask's debugging mode and Insomnia providing detailed insights for fixes. This streamlined process guaranteed reliability and adherence to system requirements.

6.8 Chapter summary

This chapter outlined the project's implementation, starting with the setup of the Flask app and SQLAlchemy for database management. Data was simulated for users, usage, recharge plans, and agency locations. The front-end was developed using HTML, CSS, and JavaScript, interacting with client-side API endpoints. Server-side API endpoints were implemented for CRUD operations with proper validation. Debugging and testing were done using Flask's debugging mode, Insomnia, and manual testing. The next section will discuss threats to validity in this implementation.

[17] `https://flask.palletsprojects.com/en/latest/quickstart/#debug-mode`
[18] `https://insomnia.rest`

7 Threats to Validity

7.1 Internal Validity Threats

Implementation Errors: JavaScript and Jinja integration issues caused syntax conflicts, leading to false positives. SHA-256 hashing, though secure, is not the most robust for production environments.
Predictive Model Limitations: A linear regression model was used for prediction, assuming linear data, which may not reflect reality. Random Forest would likely improve accuracy but was not implemented due to time constraints.
Data Integrity Issues: The system relies on simulated data, which may not represent real-world user behavior. Additionally, incomplete subscription plan data due to scraping limitations affects accuracy.

7.2 External Validity Threats

Limited Data Variety: Simulated data lacks diversity, limiting the generalization of the system's functionality.
Dependency Failures: The external IP-based location API may cause minor discrepancies, and SQLite limits scalability for large-scale production systems.
Non-Realistic Environments: Testing in a controlled environment may not reflect real-world performance with higher user traffic or unstable networks.

7.3 Construct Validity Threats

Measurement Errors: While manual and automated tests focused on core functionality, edge cases were not fully tested, potentially missing performance and accuracy issues under real-world conditions.
Unaccounted Edge Cases: Edge cases, such as unusual user interactions or network issues, were not thoroughly tested, potentially leading to unpredicted behavior.

7.4 Conclusion and Mitigation Strategies

These threats, including data, model, and dependency limitations, may impact system accuracy and scalability. Mitigation strategies include expanding test data, switching to PostgreSQL for scalability, and improving the predictive model with algorithms like Random Forest. Enhancing security, resolving integration issues, and improving error handling will also strengthen the system in production.

8 Conclusion

This project set out to create a client support web application aimed at providing a seamless user experience, helping users make informed purchasing decisions through accurate predictions of their future usage. By incorporating machine learning techniques, this application introduces a unique feature in Tunisia, enhancing customer satisfaction and service efficiency.

The development process began with a thorough exploration of the project's scope and design, followed by a detailed implementation phase that involved building both client-side and server-side APIs, simulating user data, and integrating essential features such as error handling and validation. Despite challenges with simulated data and testing constraints, the project successfully demonstrated its potential to address issues faced by Tunisia Telecom customers.

The current version of the web application is only the first step in its development. It represents a proof of concept and serves as a foundation for numerous possible enhancements and future expansions. From integrating more advanced predictive models to improving scalability and data accuracy, the project holds great potential for further growth and refinement. The application is open to a wide range of solutions that could improve its functionality, robustness, and real-world applicability.

In conclusion, this project presents an innovative approach to customer support tailored to the needs of clients in Tunisia. While this version has its limitations, it lays the groundwork for future versions that could introduce more sophisticated features and broader user support, addressing the evolving challenges in customer service and predictive analytics.

9 References

1. Norman, D. A. (1988). *The Design of Everyday Things*. Basic Books.

2. Fielding, R. T. (2000). Architectural Styles and the Design of Network-based Software Architectures. *Doctoral Dissertation, University of California, Irvine*.

3. Luo, X., Tang, Y. (2016). Simulation Data Methods and Applications. *Journal of Simulation, 32*(4), 123-134.

4. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.

5. Tang, J., Qu, M. (2014). Personalized Predictive Analytics. *Proceedings of the ACM Conference on Web Search and Data Mining*, 45-54.

6. Richardson, L., Ruby, S. (2013). *RESTful Web APIs*. O'Reilly Media.

7. Hogan, P. (2008). *Practical API Design: Confessions of a Java Framework Architect*. Springer.

8. Connolly, T., Begg, C. (2015). *Database Systems: A Practical Approach to Design, Implementation, and Management*. Pearson Education.

9. Luo, X., Tang, Y. (2016). Simulation-based analysis for predictive analytics. *Journal of Simulation*, 10(3), 157-173.

10. Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Springer.