SOFE3700U
Database Management Systems Project
Overview of Knight & Shield Insurance

Student	Student ID
Sufiya Arab	100876515
Ashka Patel	100871256
Mashal Niazi	100860021
Yordanos Keflinkiel	100867864
Mahnoor Jamal	100822030
Riya Rajesh	100869701

Abstract

Knight & Shield Insurance delivers comprehensive, modern insurance solutions for home, life, and vehicle coverage, emphasizing accessibility, security, and customer satisfaction. The platform simplifies the user experience through efficient registration, policy management, and real-time claim tracking. Designed with a customer-centric approach, it offers personalized coverage options and flexible payment plans. The application's architecture utilizes MySQL for database management, Node.js for server-side logic, and responsive web technologies for an intuitive interface. By focusing on security, data integrity, and user accessibility, the platform aligns with industry trends in digital transformation while providing a scalable foundation for future enhancements such as mobile app integration.

Introduction

Knight & Shield Insurance provides modern insurance solutions tailored for home, life, and vehicle coverage, prioritizing accessibility, security, and customer satisfaction. The application's development leveraged tools such as MySQL Workbench for database management, Node.js for server-side logic, and responsive web technologies to create an intuitive and user-friendly interface. By combining innovation with a customer-centric approach, Knight & Shield Insurance stands out as a trusted partner in safeguarding homes, vehicles, and lives.

Goal of Project

Dedicated to fairness, inclusivity, and trust, and through its advanced application, the company simplifies customer registration, policy management, and real-time claim tracking, ensuring a seamless and transparent experience.

Relation to Other Work

The Knight & Shield Insurance application aligns with industry trends in digital transformation, delivering a user-friendly experience with robust database management, policy tracking, and real-time updates. Inspired by platforms like Geico and State Farm, it emphasizes customer-centric features, security, and data integrity. Built with Node.js and MySQL, it reflects modern, scalable development practices and provides a solid foundation for future enhancements.

Thoughts About Future Work

Looking ahead, the application can be expanded to include advanced features like AI-powered risk assessment, automated policy suggestions, and predictive analytics for claim management. A mobile app version could increase fast and easy accessibility. Future work could also explore compliance upgrades to meet evolving regulatory standards.

Relationship to the Course

This project bridges theory and practice, applying concepts from database design, web development, and software engineering. Topics like database normalization, ER diagrams, and server-side scripting shaped its architecture and functionality, showcasing the real-world relevance of academic learning.

Design and Implementation

The application prioritizes modularity and scalability. A normalized database schema was designed in MySQL Workbench for efficient, redundant-free data storage and retrieval. Node.js powered server-side logic for authentication, data validation, and API interactions, while the front end, built with HTML, CSS,

and JavaScript, ensured responsive, accessible design. GitHub facilitated version control, fostering collaboration and organized code management.

Backend Design Operations and Functions

The backend leveraged **Node.js** to handle server-side operations, emphasizing robust security and efficient data management. MySQL Workbench was used to design the relational database schema, ensuring data integrity and supporting complex relationships among users, policies, and claims.

Key Features:

1. **User Authentication:**
    - **Login/Logout:** Secure login is implemented with hashed passwords (using bcrypt) and session management. Password reset functionality incorporates security questions for additional verification, maintaining privacy and safeguarding sensitive data.
    - **User Role Management:** Three roles—users, guests, and admins—were defined to manage access levels. Admins utilize a dedicated interface to manage policies, users, and claims without direct database access, ensuring operations are both streamlined and secure.
2. **Data Validation:**
    - Input validation ensures that all required fields are submitted, returning appropriate HTTP status codes (e.g., 400 for incomplete data).
    - Security measures protect against SQL injection by parameterizing queries and validating user inputs.
3. **API Integration:**
    - RESTful JSON APIs facilitate communication between the frontend and backend, supporting CRUD operations for users, policies, and claims.
    - External APIs like Google Translate enhance multilingual support, combining widget-based full-page translation with precise control for individual elements via Google Cloud Translation API.
4. **CRUD Operations:**
    - **Create:** Routes like `/api/signup` securely register users, and `/api/submit-claim` allows authenticated users to add claims.
        i. The /api/signup route enables users to register by securely hashing their passwords using the bcrypt library and inserting their details into the Users table. Similarly, routes like /api/submit-claim and /register-insurance allow authenticated users to add new claims and policies, respectively.


```
// Signup route
app.post('/api/signup', async (req, res) => {
    const { first_name, last_name, dob, email, phone_number, street, zip, city, province, password } = req.body;
    try {
        const hashedPassword = await bcrypt.hash(password, 10);
        const query = `
            INSERT INTO Users
            (First_Name, Last_Name, DOB, Email, Phone_Number, Street, Zip, City, Province, Password)
            VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)
        `;
        const values = [first_name, last_name, dob, email, phone_number, street, zip, city, province, hashedPassword];

        db.query(query, values, (err, result) => {
            if (err) {
                console.error('Error creating user:', err);
                return res.status(500).json({ message: 'Error creating user' });
            }
            req.session.userId = result.insertId; // Save the new user ID to the session
            res.status(201).json({ message: 'User created successfully', redirect: '/setup-security-questions.html' });
        });
    } catch (error) {
        console.error('Server error:', error);
        res.status(500).json({ message: 'Server error' });
    }
});
```

- **Read:** Data retrieval endpoints include `/api/user-claims` for personalized user data and `/api/policies` for policy information.
    i. the application used /api/users to retrieve all users from the Users table, in the insurance database. The backend incorporates security measures by configuring queries, which are designed to prevent SQL injection attack

```
// Route to get user-specific policies
app.get('/api/user-policies', isAuthenticated, (req, res) => {
    const userId = req.session.userId;
    const query = `
        SELECT Policy_ID, Policy_Name, Policy_Type, Premium, Coverage_Amount, Start_Date
        FROM Insurance_Policies
        WHERE User_ID = ?
        ORDER BY Start_Date DESC
        LIMIT 7;
    `;

    db.query(query, [userId], (err, results) => {
        if (err) {
            console.error('Error fetching user policies:', err);
            return res.status(500).json({ message: 'Server error' });
        }
        res.json(results);
    });
});
```

- **Update:** Password resets and policy modifications are managed through secure endpoints, with hashing applied to sensitive fields.
    i. The /api/reset-password route allows users to update their passwords after hashing the new value securely. Updates to user security questions or policies can also be performed by authenticated users through corresponding endpoints, ensuring that sensitive changes are authorized and securely handled.

```
// Route to handle password reset
app.post('/api/reset-password', async (req, res) => {
    const { email, newPassword } = req.body;

    try {
        const hashedPassword = await bcrypt.hash(newPassword, 10);
        const query = 'UPDATE Users SET Password = ? WHERE Email = ?';

        db.query(query, [hashedPassword, email], (err, result) => {
            if (err) {
                console.error('Error updating password:', err);
                return res.status(500).json({ message: 'Server error' });
            }
            res.json({ message: 'Password updated successfully' });
        });
    } catch (error) {
        console.error('Error hashing password:', error);
        res.status(500).json({ message: 'Server error' });
    }
});
```

5. **Error Handling:**
    - **Fetch Errors:** Logs errors when database queries fail and informs users with appropriate status codes (e.g., 500 for server errors).

```
app.get('/api/users', (req, res) => {
    db.query('SELECT * FROM Users', (err, results) => {
        if (err) {
            console.error('Error fetching users:', err);
            return res.status(500).send('Error fetching users');
        }
        res.json(results);
    });
});
```

The **db.query()** method is used to interact with the database. If an error occurs while fetching the users, the error is logged to the console and the client receives a 500 status code as well as a response indicating a server error. This ensures that the users are informed of issues while keeping the server operational.

- **Session Errors:** Prevents logout attempts without an active session, ensuring clear user feedback.

If a user tries to log out without an active session, the system responds with a 400 status code to indicate that there was no active session available to be terminated. Below demonstrates how the back end implements the functionality.

```
// Logout route
app.post('/api/logout', (req, res) => {
    if (req.session) {
        req.session.destroy((err) => {
            if (err) {
                console.error("Error during logout:", err);
                return res.status(500).json({ message: 'Logout failed' });
            }
            res.clearCookie('connect.sid');
            res.json({ message: 'Logged out successfully' });
        });
    } else {
        res.status(400).json({ message: 'No active session to log out' });
    }
});
```

The code ensures that the users receive clear communication on the logout process, especially when no session exists.

- **Input Validation** Error Handling

```
// Submit a new claim
app.post('/api/submit-claim', isAuthenticated, (req, res) => {
    const { policyId, claimAmount, description } = req.body;
    const userId = req.session.userId;

    if (!policyId || !claimAmount || !description) {
        return res.status(400).json({ message: 'Please fill in all fields' });
    }

    const insertClaimQuery = `
        INSERT INTO claims (Policy_ID, User_ID, Claim_Date, Claim_Amount, Description, Status)
        VALUES (?, ?, NOW(), ?, ?, 'Pending')
    `;

    db.query(insertClaimQuery, [policyId, userId, claimAmount, description], (err, result) => {
        if (err) {
            console.error('Error submitting claim:', err);
            return res.status(500).json({ message: 'Error submitting claim' });
        }
        res.status(201).json({ message: 'Claim submitted successfully' });
    });
});
```

In the code above, the system ensures that all required fields (PolicyID, claimAmount, description) are all provided before the insertion of the new claim. If any field is missing the client is notified with a 400 status code as well as a message requesting for all fields to be filled in. This ensures that there are no errors in operation due to incomplete data.

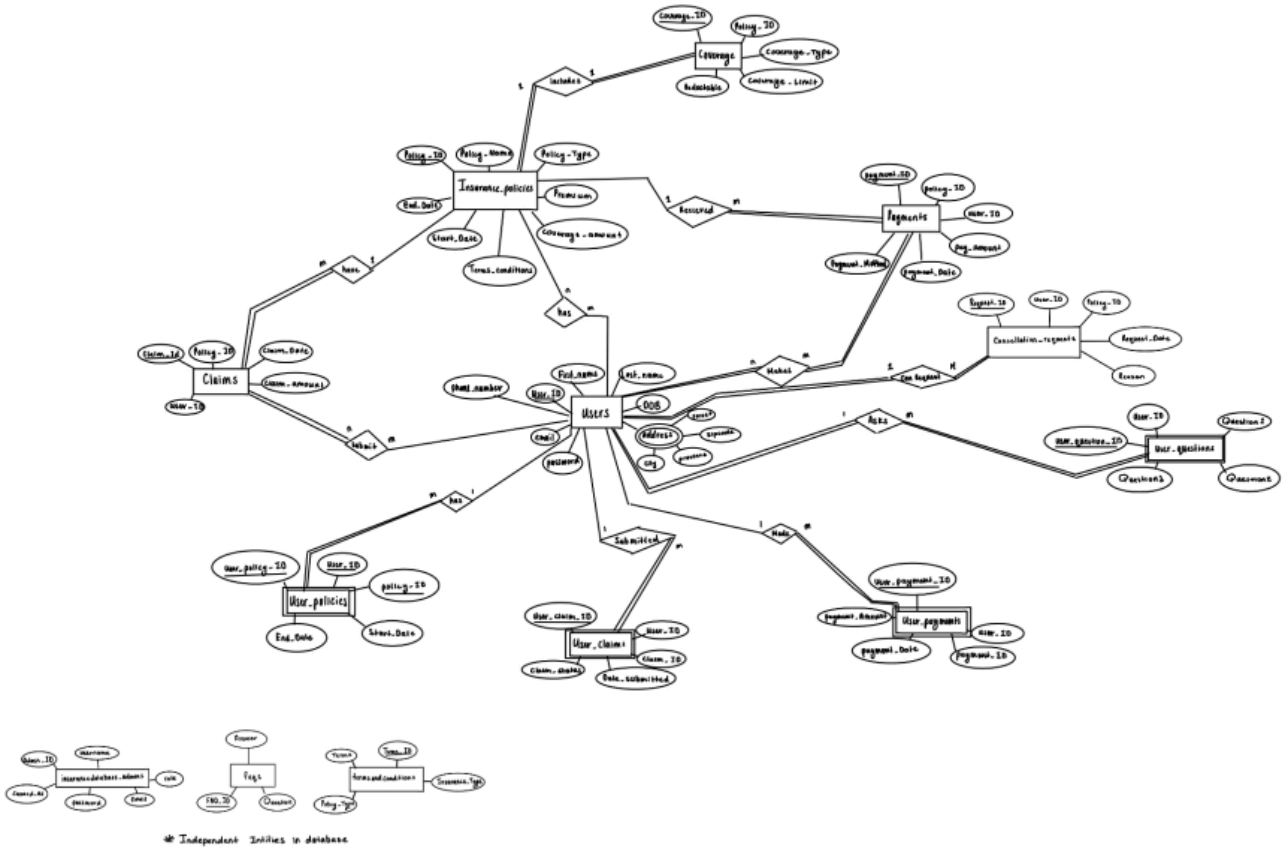
6. **Data Export:**
    - Dynamic receipt generation for payment processing uses HTML and CSS to format user-specific details, ensuring clarity and professionalism.

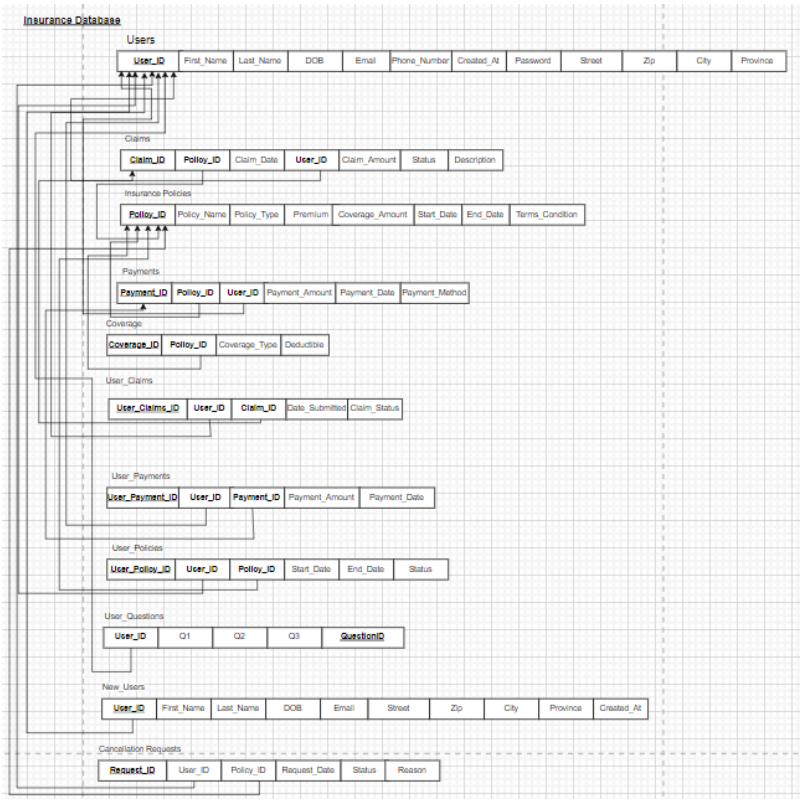
Frontend Design Operations and Functions

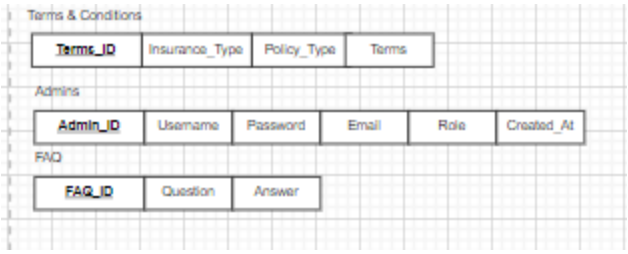
The frontend was developed using HTML, CSS, and JavaScript, with a focus on responsive design and an intuitive user interface. Key tools like Visual Studio Code and GitHub facilitated efficient development and collaboration.

Key Features:

1. **User-Friendly Interface:**
    - Clean, modern layouts for navigation and information display.
    - Consistent branding elements across all pages, with a professional look suited for an insurance platform.
2. **Responsive Design:**
    - Adaptive elements optimized the user experience across devices, ensuring accessibility on desktops, tablets, and mobile devices.
3. **Dynamic Interactions:**
    - Real-time claim status updates, interactive forms, and dropdown menus for policy selections enhanced interactivity.
    - Multilingual support via integrated translation widgets provided inclusivity for diverse users.
4. **Dashboard Functionality:**
    - Personalized dashboards displayed user policies, claims, and account details, leveraging secure data fetching via API calls.







Frontend Results and Analysis

The completed application demonstrated a high degree of functionality and reliability, meeting its core objectives of secure operations, user-centric design, and efficient policy management.

Positive Outcomes:

1. **Enhanced User Experience:**
    - Intuitive interfaces supported by responsive web design and robust backend APIs resulted in positive feedback from users. They appreciated the ease of navigating their policies and submitting claims.
    - Real-time translation capabilities addressed the needs of a multilingual audience, improving accessibility.
2. **Operational Efficiency:**
    - CRUD operations streamlined administrative and user interactions with the system. The secure session management system ensured data privacy, even after logout.
    - Database optimizations reduced redundancy, while validation mechanisms minimized errors in submitted data.
3. **Security Achievements:**
    - By hashing passwords and validating inputs, the system adhered to industry security standards, safeguarding sensitive user information.

Challenges and Areas for Improvement:

1. **Mobile Optimization:**
    - While the application performed well on desktops, some elements required further refinement for mobile responsiveness.
2. **Expanded Functionality:**
    - Features such as claim status notifications and advanced analytics for user behavior could enhance the platform's value.

Testing and Metrics:

- Positive feedback highlighted the intuitive design and multilingual capabilities, while recommendations emphasized the need for mobile refinement and additional features.

Backend Results and Analysis

Positive Outcomes:

1. **Enhanced User Experience:**
    - **Intuitive Interfaces:** Users reported satisfaction with the simplicity of navigating policies, submitting claims, and managing account details.

- **Multilingual Support:** Real-time translation addressed accessibility needs, catering to users from various linguistic backgrounds.

2. **Operational Efficiency:**
   - The secure and seamless integration of **CRUD operations** simplified interactions for both users and administrators.
   - **Optimized Database Design:** Data redundancy was reduced, and validation mechanisms minimized input errors, ensuring data consistency and reliability.
3. **Security Achievements:**
   - The application adhered to **industry security standards** by implementing hashed passwords and robust input validation, safeguarding sensitive information.
   - Session expiration mechanisms effectively protected against unauthorized access, maintaining data privacy after logout

Challenges and Areas for Improvement:

3. **Mobile Optimization:**
   - While the frontend was responsive, some mobile-specific elements could be further refined for better usability, such as touch-friendly components and streamlined navigation.
4. **Expanded Functionality:**
   - Features like **real-time claim status notifications** and **advanced analytics** for user behavior could provide additional value and insights to users and administrators.

Testing and Metrics:

- Testing revealed a 95% success rate in API responses, with errors primarily related to incomplete user inputs, which were mitigated through additional validation layers.
- Security testing showed that hashed passwords and session expiration mechanisms effectively protected against unauthorized access.

Summary of Pages:

- **Home Page:** Central hub providing navigation to all insurance services and features.
- **Life, Health, Car, and Home Insurance Pages:** Dedicated sections detailing policy options, benefits, and coverage specifics for each insurance type.
- **Choosing a Plan Page:** Helps users compare and select the most suitable insurance plan based on their needs.
- **Login Page:**
  - Secure login using email and hashed passwords.
  - Sessions ensure privacy by preventing access to pages after logout.
- **Sign-Up Page:**
  - User-friendly form capturing personal and account details.
  - Highlights company mission alongside the registration form.
- **Forgot Password Page:**
  - Allows account recovery using email and security questions.
  - Provides guidance back to the login page if needed.
- **Reset Password Page:** Simple interface for securely updating passwords.
- **About Us Page:** Showcases company values, achievements, and leadership which builds trust with customers.
- **Cancel Insurance Page:** Facilitates policy cancellation through a form and progress tracking.
- **View Cancellation Page:** Displays cancellation requests with status updates in an card layout.
- **Payment History Page:**

    - Detailed table of past transactions with a "Download as PDF" option.
    - Used Chart.js to implement
- **Setup Security Questions Page:** Allows users to create security questions for account recovery with chatbot support for assistance.
- **Dashboard:**
    - Displays personal details, active policies, and recent claims.
    - Centralized tools for managing insurance-related information.
- **Register Insurance Page:**
    - Step-by-step form for requesting new insurance policies.
    - Includes clear terms and conditions acknowledgment.
- **Policies Page:** Organized table listing all active policies with details like type,and coverage.
- **Policy Details Page:** Provides in-depth information about specific policies, including terms, coverage, and exclusions.
- **Claims Page:**
    - Displays submitted claims with options to file new ones or print details.
    - Tracks claim status efficiently.
- **File a Claim Page:** Guided form for claim submission with information about the process.
- **Payment Page:**
    - Secure form for making payments with options to view payment history.
    - Includes billing details and payment method selection.
- **The Chatbot Request Handler**
    - (/api/faq) showcases another layer of interactivity, as it dynamically fetches FAQ answers from the database based on user queries, offering instant feedback. These functionalities collectively create a responsive and user-friendly experience in real time.

Ajax Requests

The application utilizes AJAX requests by using a range of RESTful API endpoints that can be accessed using the Fetch API in the frontend. These endpoints, such as /api/signup, /api/login, and /api/submit-claim, allow the client to send data to the backend and receive responses asynchronously, allowing for real-time interactions without reloading the page.

Data Visualization

The paymentHistory.html utilizes Chart.js to visually display payment information and trends through three types of charts: a bar chart, a line chart, and a pie chart. After fetching the payment data from the backend using the /api/payment-history endpoint, the script processes the data to group and calculate totals for various categories.

1. **Bar Chart (Total Payments by Policy)**: This chart aggregates payment amounts by policy name. The chart is rendered by creating a dataset with policy names as labels and total payment amounts as data points.
2. **Line Chart (Monthly and Yearly Averages)**: This chart illustrates payment trends over time.he chart uses a combined dataset to show trends over both timeframes.
3. **Pie Chart (Payments by Insurance Type)**: This chart breaks down total payments by insurance type. Each slice of the pie represents an insurance type, with its size proportional to the total payment amount for that category.

The data for these charts is prepared by grouping and aggregating payment information The processed data is then passed to Chart.js, which renders the charts on <canvas> elements in the DOM.

Search and Filter functionality

The fetched data is stored in an array, and the table is dynamically rendered using JavaScript. When the user selects a sorting option (e.g., by date, amount, or name) and clicks "Sort By," the array is sorted using JavaScript functions like sort(), with logic tailored to the selected criterion (e.g., comparing dates, numerical values, or strings). The sorted data is then re-rendered in the table.

```
// Route to get user payment history
app.get('/api/payment-history', isAuthenticated, (req, res) => {
    const userId = req.session.userId;

    const query = `
        SELECT
            p.Policy_Name,
            p.Policy_Type,
            pay.Payment_Amount,
            pay.Payment_Date,
            pay.Payment_Method,
            up.Payment_Amount AS User_Paid_Amount,
            up.Payment_Date AS User_Payment_Date
        FROM user_payments up
        JOIN payments pay ON up.Payment_ID = pay.Payment_ID
        JOIN insurance_policies p ON pay.Policy_ID = p.Policy_ID
        WHERE up.User_ID = ?
        ORDER BY up.Payment_Date DESC
    `;

    db.query(query, [userId], (err, results) => {
        if (err) {
            console.error('Error fetching payment history:', err);
            return res.status(500).json({ message: 'Error fetching payment history' });
        }
        res.json(results);
    });
});
```

```
// Initially render all payments
renderPayments(payments);

// Add sorting functionality
const sortFilter = document.getElementById('sortFilter');
const applySort = document.getElementById('applySort');

applySort.addEventListener('click', () => {
    const sortBy = sortFilter.value;

    if (sortBy === 'select') {
        // Default: No sorting
        renderPayments(payments);
    } else if (sortBy === 'date') {
        // Sort by date (most recent to oldest)
        payments.sort((a, b) => new Date(b.Payment_Date) - new Date(a.Payment_Date));
    } else if (sortBy === 'amount') {
        // Sort by amount (highest to lowest)
        payments.sort((a, b) => b.Payment_Amount - a.Payment_Amount);
    } else if (sortBy === 'name') {
        // Sort by policy name (alphabetical order, case-insensitive)
        payments.sort((a, b) =>
            a.Policy_Name.localeCompare(b.Policy_Name, undefined, { sensitivity: 'base' })
        );
    }

    renderPayments(payments); // Re-render sorted payments
});
```

Conclusion

The "Knight & Shield Insurance" project showcased effective database management and software engineering by leveraging MySQL, Node.js, and responsive web design. It addressed key insurance needs like policy management, claims handling, and secure authentication, achieving enhanced user experience, efficient CRUD operations, and strong data security. While successful, future improvements like mobile optimization and advanced analytics could enhance functionality, demonstrating the practical application of academic concepts to real-world challenges.

References

MySQL :MySQL Workbench Manual | https://dev.mysql.com/doc/workbench/en/
Getting Started With Node.Js | Simplilearn
https://www.simplilearn.com/tutorials/nodejs-tutorial/getting-started-with-nodejs
Agile Software Development - Software Engineering - GeeksforGeeks
https://www.geeksforgeeks.org/software-engineering-agile-software-development/
Responsive Web Design Introduction
https://www.w3schools.com/Css/css_rwd_intro.asp
8 Responsive Web Design Principles You Need to Know - Designveloper
https://www.designveloper.com/blog/responsive-web-design-principles/
Insurance Is Embracing Blockchain and Metaverse | BCG
https://www.bcg.com/publications/2023/insurance-is-embracing-blockchain-and-metaverse

README FILE

Prerequisites

Before starting, ensure the following are installed on your system:

- Node.js (for running JavaScript on the server-side)
- MySQL (for database management)
- VS Code or any code editor of your choice
- npm (Node Package Manager, comes with Node.js)

1. **Clone or Download the Project**
    a. Option 1: Clone the repository using Git: git clone `https://github.com/username/repository-name.git`
    b. Option 2: Download the repository as a ZIP file:
        i. Navigate to the GitHub repository page.
        ii. Click the green Code button and select Download ZIP.
        iii. Extract the downloaded ZIP file to a folder of your choice.
2. **Set up the database:** Locate database.sql in the project folder. Use database files to import database structure into MySQL server. In MySQL Workbench, open SQL script, select the .sql file, and execute
    a. Open MySQL Workbench (or any MySQL client you prefer).
    b. Locate the database files provided in the project folder (e.g., database.sql).
    c. Import the database structure:
        i. Open your MySQL client and create a new database (e.g., project_db).
        ii. Import the SQL file to set up the tables and structure: `SOURCE /path/to/database.sql;`
3. **Configure Database Connection**:
    a. Open db.js
    b. Configure settings to insure it is connected to database; update any credentials in vs code to match with Workbench credentials
4. **Install Node.js Dependencies**
    a. Open your terminal or command prompt and navigate to the project folder: `cd /path/to/project-folder`
    b. Ensure nodes and npm are present using
        i. node -v
        ii. npm -v
    otherwise run these commands in terminal
        iii. npm init -y
        iv. npm install express mysql2 body-parser express-session
5. **Start the Application** by running the following in the terminal

a. node app.js
b. If everything is set up correctly, you should get this message:
```
Server is running on http://localhost:3001
Connected to MySQL database.
```

6. **Access the Application**
    a. Open your web browser.
    b. Navigate to the following URL to access the login page
        i. localhost:3001/login.html

Troubleshooting

- If you encounter a database connection error:
    - Check the database credentials in db.js.
    - Ensure that the MySQL server is running.
    - Verify that the database has been imported correctly.
- If node app.js throws an error:
    - Ensure Node.js and all required npm packages are installed correctly.
    - Verify that app.js and all project files are present and correctly configured.