SRI KRISHNA COLLEGE OF TECHNOLOGY
(An Autonomous Institution)
Approved by AICTE | Affiliated to Anna University Chennai|
Accredited by NBA - AICTE| Accredited by NAAC with 'A' Grade
KOVAIPUDUR, COIMBATORE 641042

LIFE INSURANCE PORTAL

A PROJECT REPORT

Submitted by

NITHIN R	**-**	**727822TUAM032**
SHYLENDRA PRABU R	**-**	**727822TUAM053**
KRITHIK SS	**-**	**727822TUAM023**
LOKESH KRISNA C	**-**	**727822TUAM027**

In partial fulfilment for the award of the degree of

BACHELOR OF ENGINEERING

IN

COMPUTER SCIENCE AND ENGINEERING

(ARTIFICIAL INTELLIGENCE AND MACHINE LEARNING)

JULY 2024

BONAFIDE CERTIFICATE

Certified that this project report **LIFE INSURANCE PORTAL** is the Bonafide work of **NITHIN R 727822TUAM032, SHYLENDRA PRABU R 727822TUAM053, KRITHIK SS 727822TUAM023, LOKESH KRISNA C 727822TUAM027** who carried out the project work under my supervision.

SIGNATURE

Mrs. SOUNDARYA. S

SUPERVISOR

Assistant Professor,

Department of Computer Science and Engineering (Artificial Intelligence and Machine Learning),

Sri Krishna College of

Technology, Coimbatore-641042

SIGNATURE

Dr. SUMA SIRA JACOB

HEAD OF THE DEPARTMENT

Associate Professor,

Department of Computer Science and Engineering,

Sri Krishna College of

Technology, Coimbatore-641042

Certified that the candidate was examined by me in the Project Work Viva Voce examination held on_____at Sri Krishna College of Technology, Coimbatore-641042.

INTERNAL EXAMINER **EXTERNAL EXAMINER**

ACKNOWLEDGEMENT

First and fore most we thank the **Almighty** for being our light and for showering his gracious blessings throughout the course of this project.

We express our gratitude to our beloved Principal, **Dr. M.G. Sumithra,**

for providing all facilities.

We are grateful to our beloved Head, Computing Sciences **Dr. T. Senthilnathan,** for her tireless and relentless support.

With the grateful heart, our sincere thanks to our Head of the Department **Dr. Suma Sira Jacob**, Department of Computer Science and Engineering for the motivation and all support to complete the project work.

We thank **Mrs. Soundarya S**, Assistant Professor, Department of Computer Science and Engineering (Artificial Intelligence and Machine Learning), for his motivation and support.

We are thankful to all the **Teaching and Non-Teaching Staff** of Department of Computer Science and Engineering and to all those who have directly and indirectly extended their help to us in completing this project work successfully.

We extend our sincere thanks to our **family members** and our beloved

friends, who had been strongly supporting us in all our Endeavour

ABSTRACT

The Life Insurance Portal designed for insurance leverages modern technologies such as React, Spring Boot, and MySQL DB to deliver an all-encompassing platform that transforms insurance management. Featuring a secure authentication system and role-based access controls, the portal allows clients to seamlessly browse various insurance plans, fill out personal details forms, and manage their policies online. Administrators benefit from a comprehensive dashboard that supports the creation, modification, and deletion of insurance plans, along with robust reporting tools to monitor client activities and analyze policy trends. The client dashboard provides a user-friendly interface for managing personal information, viewing policy details, and tracking application statuses. The insurance plan management module empowers administrators with detailed control over plan attributes such as type (Term Life, Whole Life, Universal Life, and Variable Life), coverage details, premium calculations, and eligibility criteria. A calendar view helps clients keep track of premium due dates and policy milestones, ensuring timely payments and policy updates. Utilizing React 's interactive frontend, Spring Boot's powerful RESTful APIs, and MySQL DB's scalable database structure, the system enhances operational efficiency by automating workflows and improving client communication. The portal's scalability ensures it can accommodate a growing client base and an expanding range of insurance products. Advanced reporting features provide actionable insights for strategic decision-making and resource optimization. In conclusion, the Life Insurance Portal offers a cutting-edge solution tailored to meet the needs of both clients and administrators, ensuring a seamless, efficient, and user-centric experience in managing life insurance policies.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF ABBREVIATIONS

Abbreviation	Acronym
HTML	HYPERTEXT MARKUP LANGUAGE
CSS	CASCADING STYLESHEET
JS	JAVASCRIPT
SDLC	SOFTWARE DEVELOPMENT LIFE CYCLE
HTTP	HYPER TEXT TRANSFER PROTOCOL
API	APPLICATION PROGRAMMIN INTERFACE
SQL	STRUCTURED QUERY LANGUAGE
XML	EXTENSIBLE MARKUP LANGUAGE

CHAPTER 1

INTRODUCTION

This project aims to provide a comprehensive and user-friendly solution for individuals seeking life insurance through an online portal. In this chapter, we will explore the problem statement, provide an overview, and outline the main objectives of the life insurance portal.

1.1PROBLEM STATEMENT

How can we develop an online life insurance portal that allows users to easily navigate through various insurance options, securely register and authenticate their identities, and efficiently manage their insurance profiles for a seamless and satisfactory experience?

1.2OVERVIEW

In the domain of life insurance, individuals often encounter challenges such as complex insurance options, confusing registration processes, and security concerns. To address these issues, we propose the creation of an online life insurance portal. This system will provide clear information on different insurance plans, simplify the registration process, and implement robust security protocols, ensuring a user-friendly and secure experience for all users.

1.3OBJECTIVE

The primary objective of this project is to develop an online life insurance portal that offersusers a user-friendly and secure platform to explore, select, and manage their insurance plans conveniently. By eliminating the complexities of traditional insurance methods and providing streamlined processes, the system aims to enhance user satisfaction and improve the overall efficiency of managing life insurance for all users.

CHAPTER 2

SYSTEM SPECIFICATION

In this chapter, we are going to see the software that we have used to build the website. This chapter gives you a small description about the software used in the project.

2.1 VS CODE

Visual Studio Code is a source code editor developed by Microsoft for Windows, Linux, and macOS. It includes support for debugging, embedded Git control, syntax highlighting, intelligent code completion, snippets, and code refactoring. It is also customizable, so users can change the editor's theme, keyboard shortcuts, and preferences.

VS Code is an excellent code editor for React projects. It is lightweight, customizable, and has a wide range of features that make it ideal for React development. It has built-in support for JavaScript, JSX, and TypeScript, and enables developers to quickly move between files and view detailed type definitions. It also has a built-in terminal for running tasks. Additionally, VS Code has an extensive library of extensions that allow developers to quickly add features like code snippets, debugging tools, and linting support to their projects.

2.2 REACT JS

React is a powerful JavaScript library designed for building user interfaces with a component-based architecture. This modular structure allows for the creation of reusable UI components, significantly enhancing development efficiency and maintainability. React 's use of the Virtual DOM enables it to perform efficient updates, minimizing costly interactions with the real DOM and ensuring smooth and high-performance user experiences. Its declarative approach makes it easy for developers to design interactive and dynamic user interfaces. Additionally, React boasts a robust ecosystem of libraries and tools, making it suitable for a wide range of applications, from simple websites to complex web applications.

2.3 SPRING BOOT

Spring is a comprehensive Java framework that facilitates enterprise-level application development by providing a robust set of features to simplify coding and enhance productivity. Its modular design, known as the Spring Framework, supports various functionalities such as dependency injection, aspect-oriented programming, and a powerful MVC framework for web applications. Spring Boot, a project within Spring, streamlines the setup process with pre-configured starter templates and an embedded server, reducing the complexity of configuration and deployment. The Spring ecosystem is well-regarded for its extensive documentation, community support, and seamless integration with other technologies, making it a popular choice for building scalable and maintainable Java applications.

CHAPTER 3

PROPOSED SYSTEM

This chapter gives a small description about the proposed idea behind the development of our website

3.1 PROPOSED SYSTEM

This system offers a multitude of benefits from various perspectives. The online exam registration platform empowers users to register conveniently, explore a wide array of exams, and complete the registration process with ease. Payment options include online transactions or offline methods, catering to the preferences and convenience of candidates.

Once a registration is completed, it is directed to the administration center or designated personnel responsible for processing registrations. The administrative team ensures timely processing of registrations, with specialized staff members handling data entry and verification within a specified timeframe. Upon completion, the confirmed registrations are updated in the system, ready for the examination.

This system significantly reduces the administrative workload and streamlines operations in the exam registration process. Especially during peak periods or high registration volumes, where manual processes may lead to delays or errors, the online registration platform mitigates such challenges. Candidates can directly register online, bypassing potential bottlenecks such as limited office hours or manual processing delays.

Moreover, the system enhances operational efficiency, enabling administrators to manage registrations promptly and effectively. By leveraging technology to automate data entry and verification, the online registration platform ensures swift and accurate processing, resulting in improved candidate satisfaction and efficiency in exam management.

3.2ADVANTAGES

Convenience: Users can easily explore a comprehensive list of available life insurance plans, select their desired options, and complete the application process from anywherewith internet access. This eliminates the need for in-person visits to insurance offices, saving time and effort for busy individuals and families.

Accessibility: The life insurance portal provides users with round-the-clock access to detailed information about insurance plans, including coverage options, premiums, benefits, and requirements. This accessibility ensures that users can make informed decisions at their convenience.

Transparency: Each insurance plan is presented with comprehensive and easy-to-understand information, including eligibility criteria, coverage details, premium amounts, and policy terms. This transparency helps users compare and select the bestplan for their needs.

Flexibility: Users have the flexibility to choose their preferred payment method for premiums, whether it's online transactions, automatic bank withdrawals, or traditionaloffline payments. This accommodates diverse payment preferences and ensures a smooth financial process.

CHAPTER 4

METHODOLOGIES

This chapter gives a small description about how our system works.

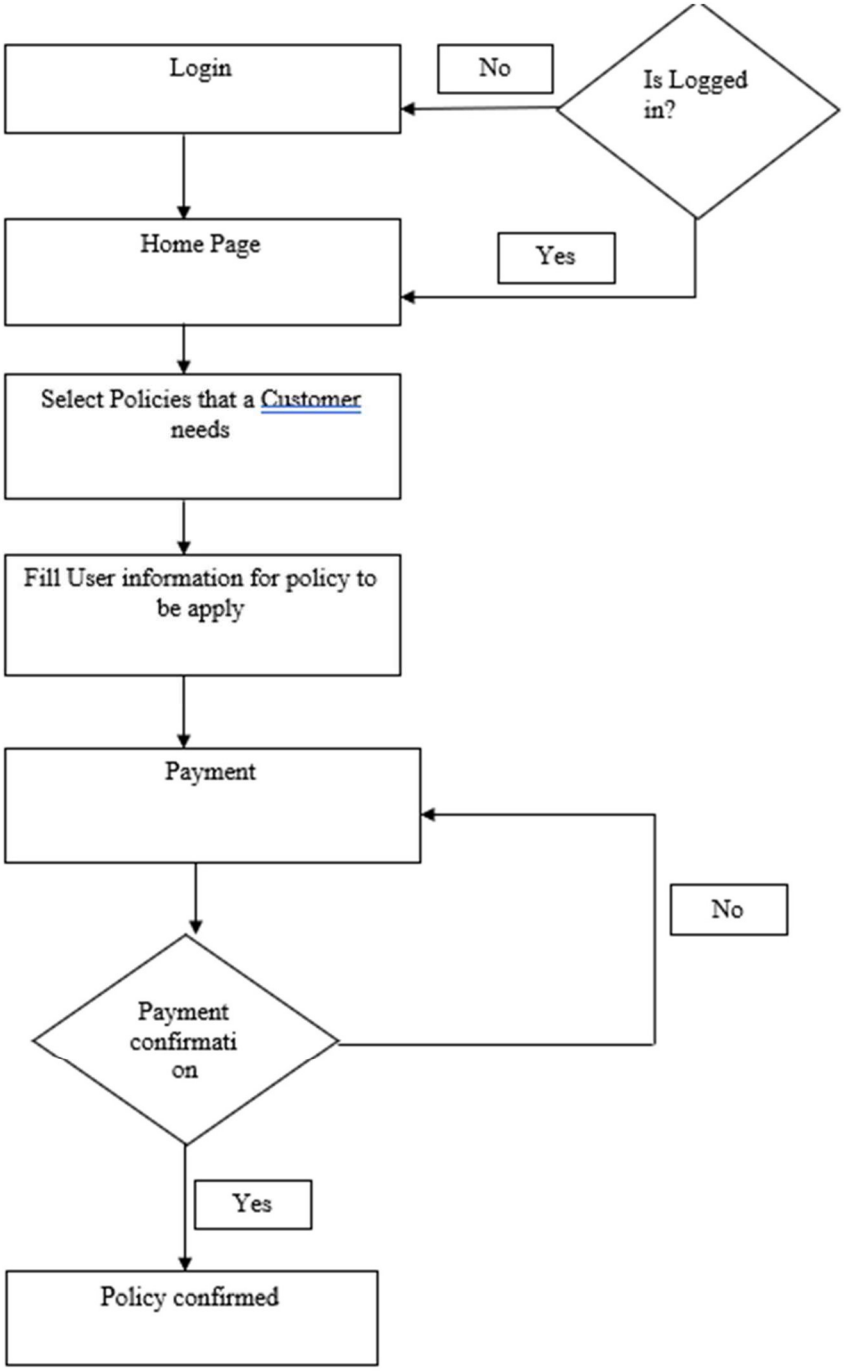


Fig 4.1 Process flow diagram

User Registration:

Users can register on the platform by providing essential details such as name, email address, and password.

Additional information like address and contact number may be included for communication and personalized service purposes.

Plan Exploration:

Upon registration/login, users can explore available life insurance plans by browsing through categories or using the search functionality.

Each insurance plan listing includes details such as plan name, description, coverage options, premium amounts, and eligibility criteria.

Application Process:

Users can select the desired life insurance plan(s) and proceed to the application process.

During the application, users may need to provide additional information like personalhealth details, beneficiary information, or identification documents.

Payment Processing:

The platform securely processes premium payments using various payment methods, such as credit/debit cards, net banking, or digital wallets.

Users receive a confirmation of successful payment upon completion.

Policy Issuance:

After the application is confirmed, users receive a digital policy document with details such as coverage amount, policy terms, and premium payment schedule.

The policy document serves as proof of coverage and includes instructions for future premium payments.

Policy Management:

Users can access their policy details at any time through their account on the platform. Users can update their personal information, payment methods, or beneficiaries as needed.

Claim Process:

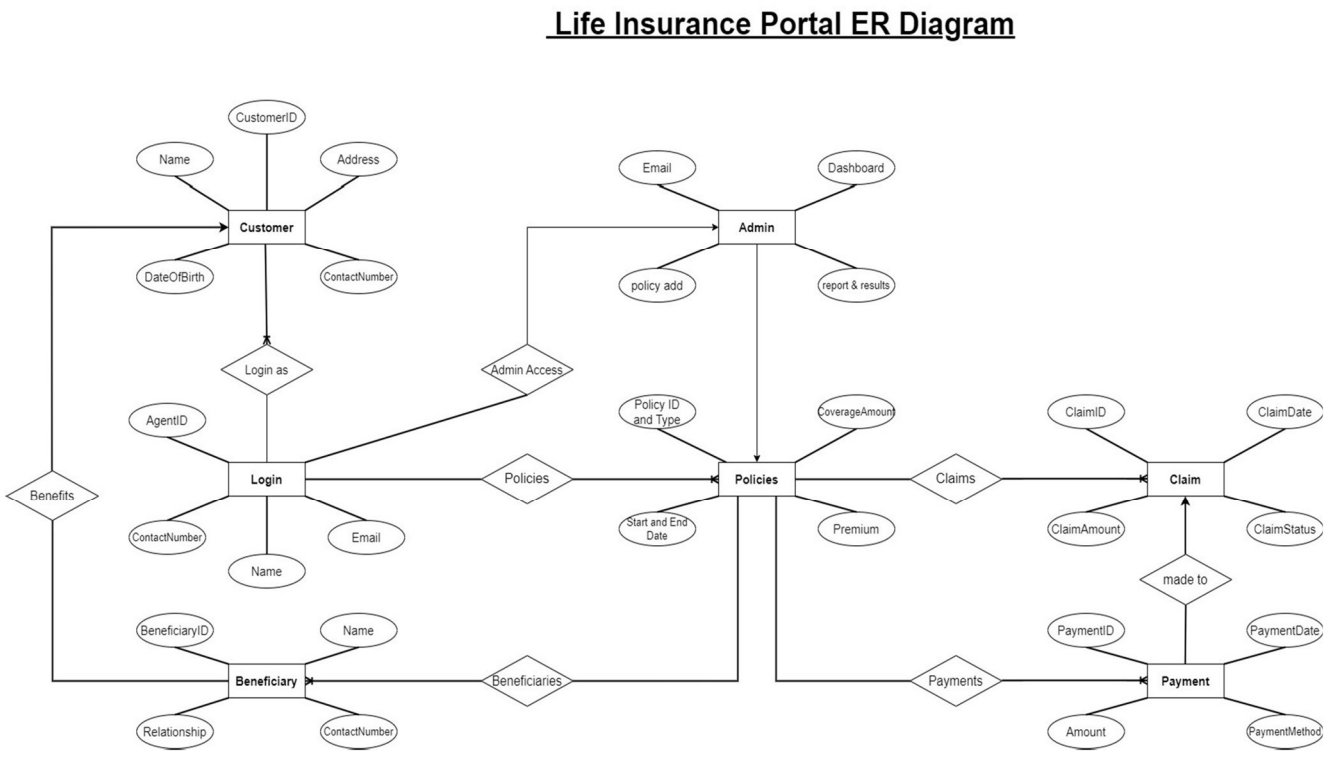
In the event of a claim, users can initiate the process through the platform by providingnecessary claim details and documentation.

Insurance administrators review and verify the claim information.

The platform provides real-time updates on the status of the claim, ensuring transparency and prompt communication.

ER Diagram:

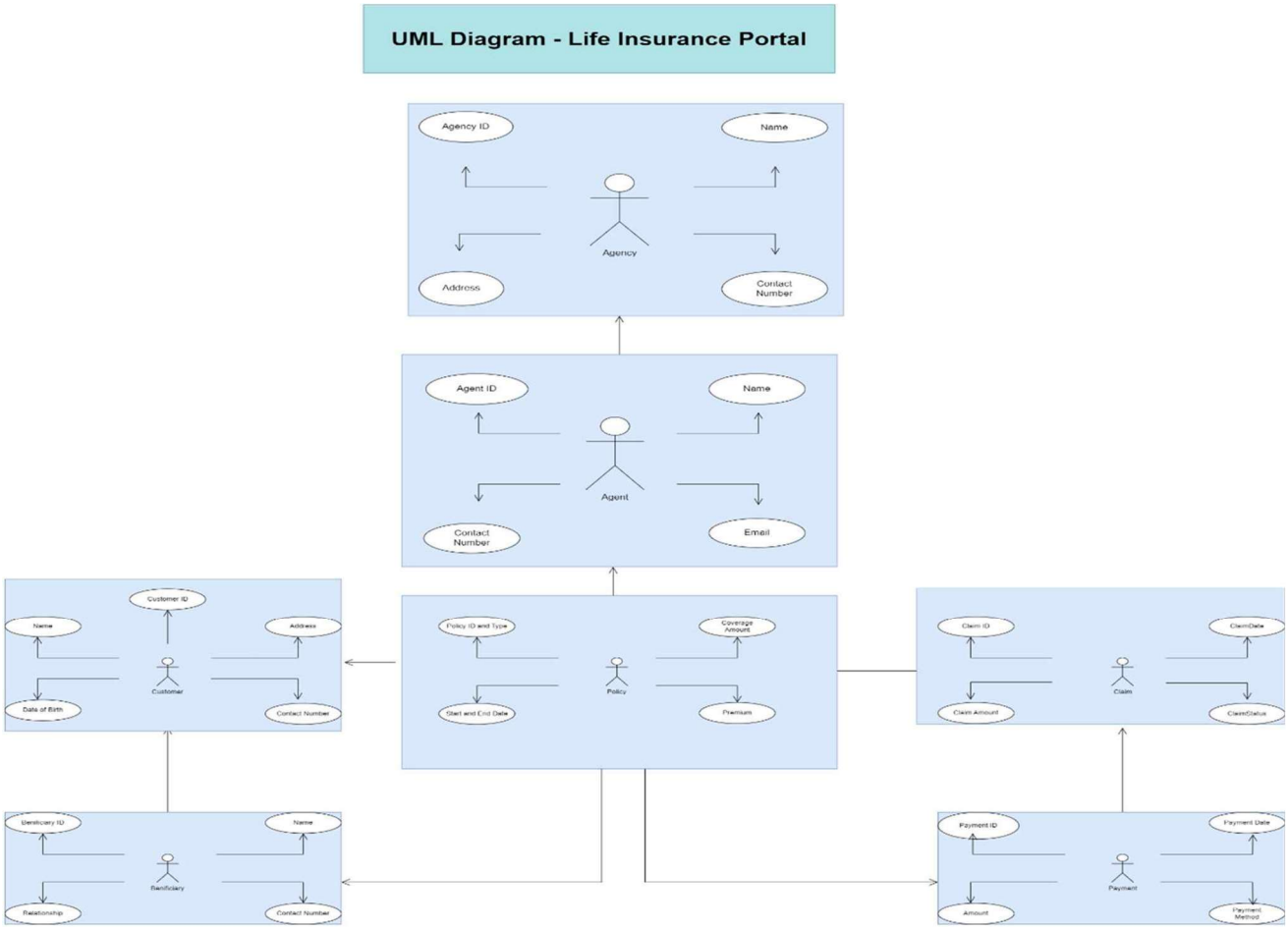
The ER diagram for the Life Insurance Portal illustrates the key entities and their relationships within the system. The primary entities include Customer (with details such as Customer ID, Name, Address, Date Of Birth, and Contact Number), Admin (handling tasks like policy addition and report generation), Policies (linked to both customers and administrators, containing Policy ID, Type, Coverage Amount, Premium, and dates), Beneficiary (related to customers, including Beneficiary ID, Name, Relationship, and Contact Number), Payment (tracking payments with Payment ID, Date, Amount, and Method), and Claim (connected to policies, encompassing Claim ID, Date, Amount, and Status). This diagram captures the management of customer information, policy details, beneficiary records, and the claims and payments processes, all coordinated by administrators.



4.2 ER Diagram

UML Diagram:

The UML diagram for the Life Insurance Portal depicts the relationships and interactions among the primary entities: Agency, Agent, Customer, Policy, Beneficiary, Payment, and Claim. An **Agency** is identified by an Agency ID, Name, Address, and Contact Number. **Agents**, linked to agencies, have Agent ID, Name, Contact Number, and Email. **Customers** are characterized by Customer ID, Name, Date of Birth, Address, and ContactNumber. Customers can have **Beneficiaries** with Beneficiary ID, Name, Relationship, and Contact Number. **Policies** are described by Policy ID, Type, Coverage Amount, Premium, and Start and End Dates. Policies are connected to **Claims**, which include Claim ID, Claim Date, Claim Amount, and Claim Status. **Payments** linkedto claims are defined by Payment ID, Payment Date, Amount, and Payment Method. This diagram highlights the structured flow and connections among these entities, ensuring a cohesive life insurance management system.



4.3 Class Diagram

CHAPTER 5
IMPLEMENTATION AND RESULT

This chapter gives a description about the output that we produced by developingthe website of our idea.

1. LOGIN

When User enters our Website he will be asked about his login details like email id andpassword. The login details will be verified with the details given while the user creates an account.

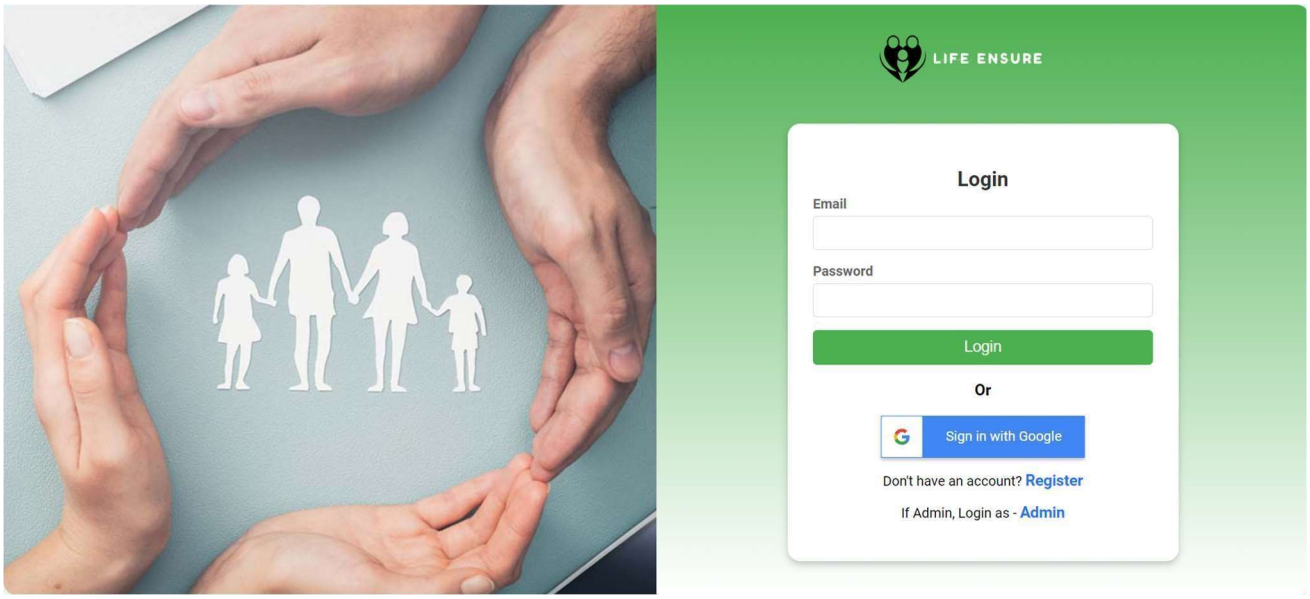


Fig 5.1 LOGIN PAGE

2. USER REGISTER

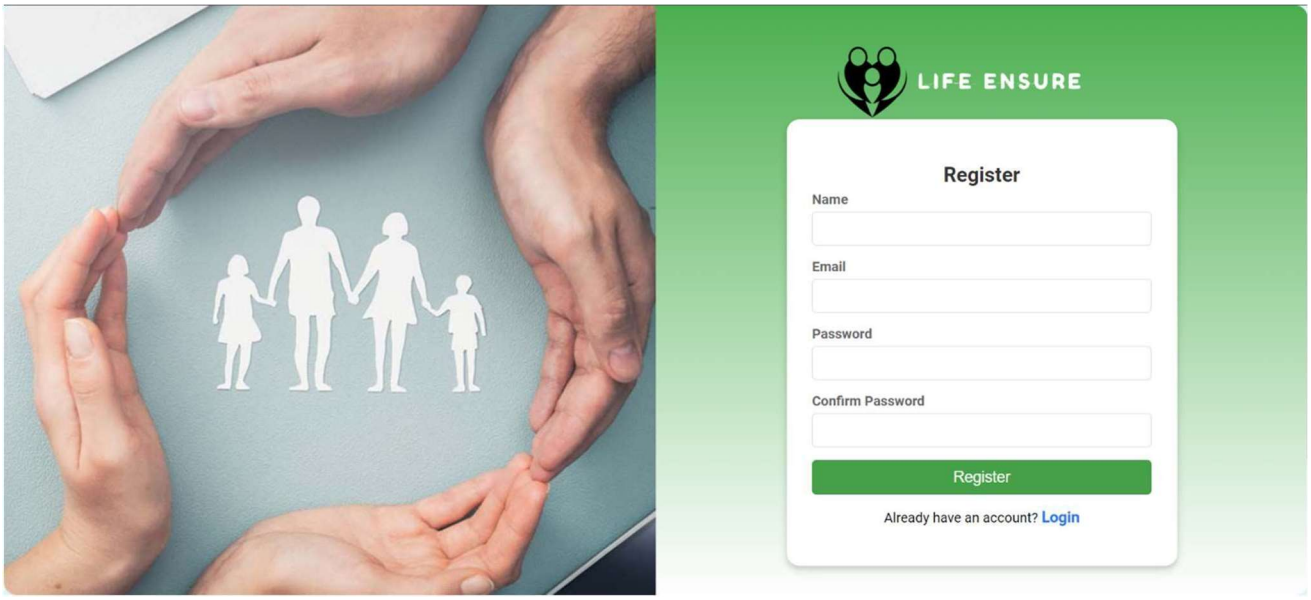
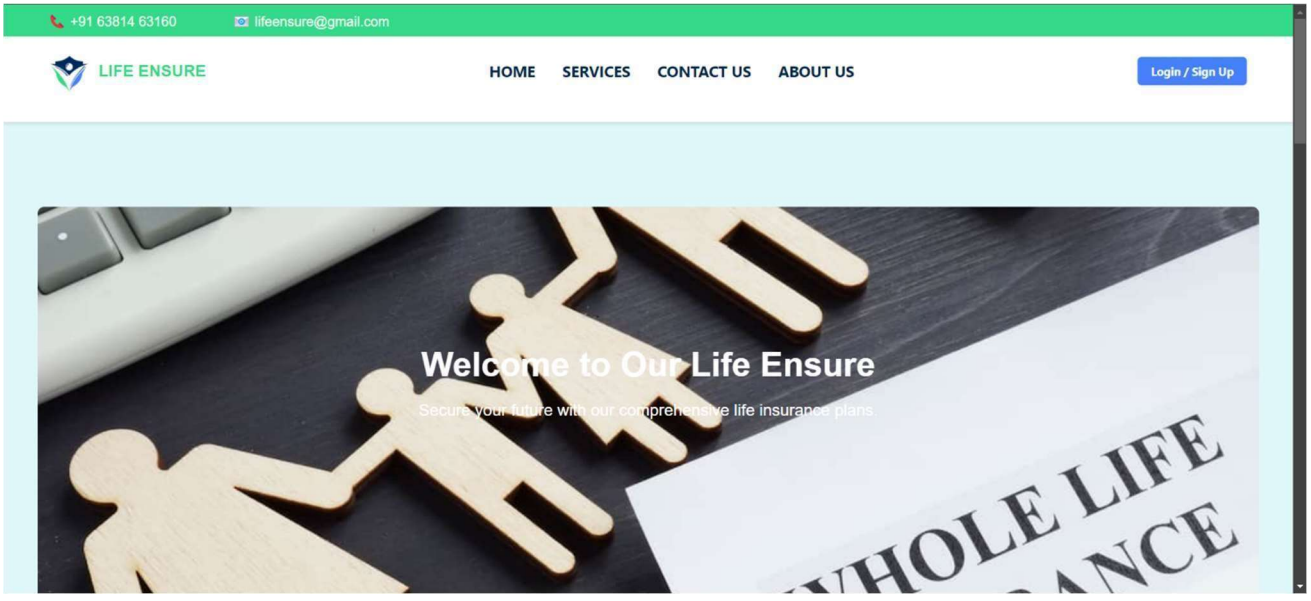


Fig 5.2 USER REGISTER

3. HOME PAGE



5.3 HOME PAGE

4. SERVICES

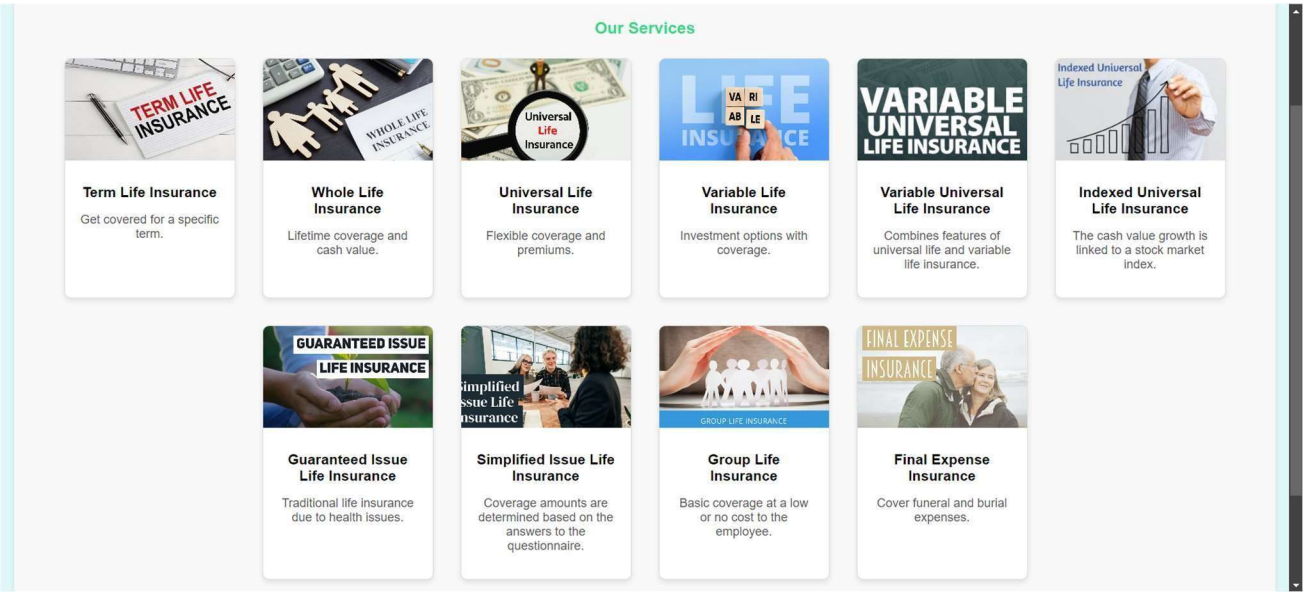


Fig 5.4 SERVICES

5. ABOUT US

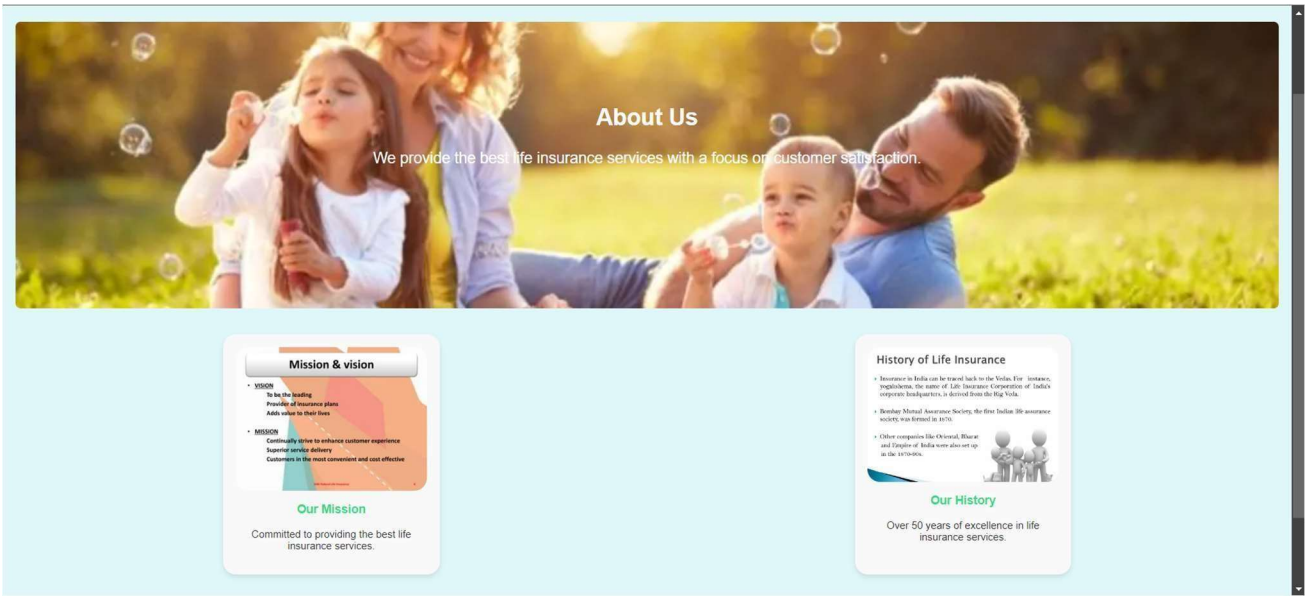


Fig 5.5 ABOUT US

6. CONTACT US

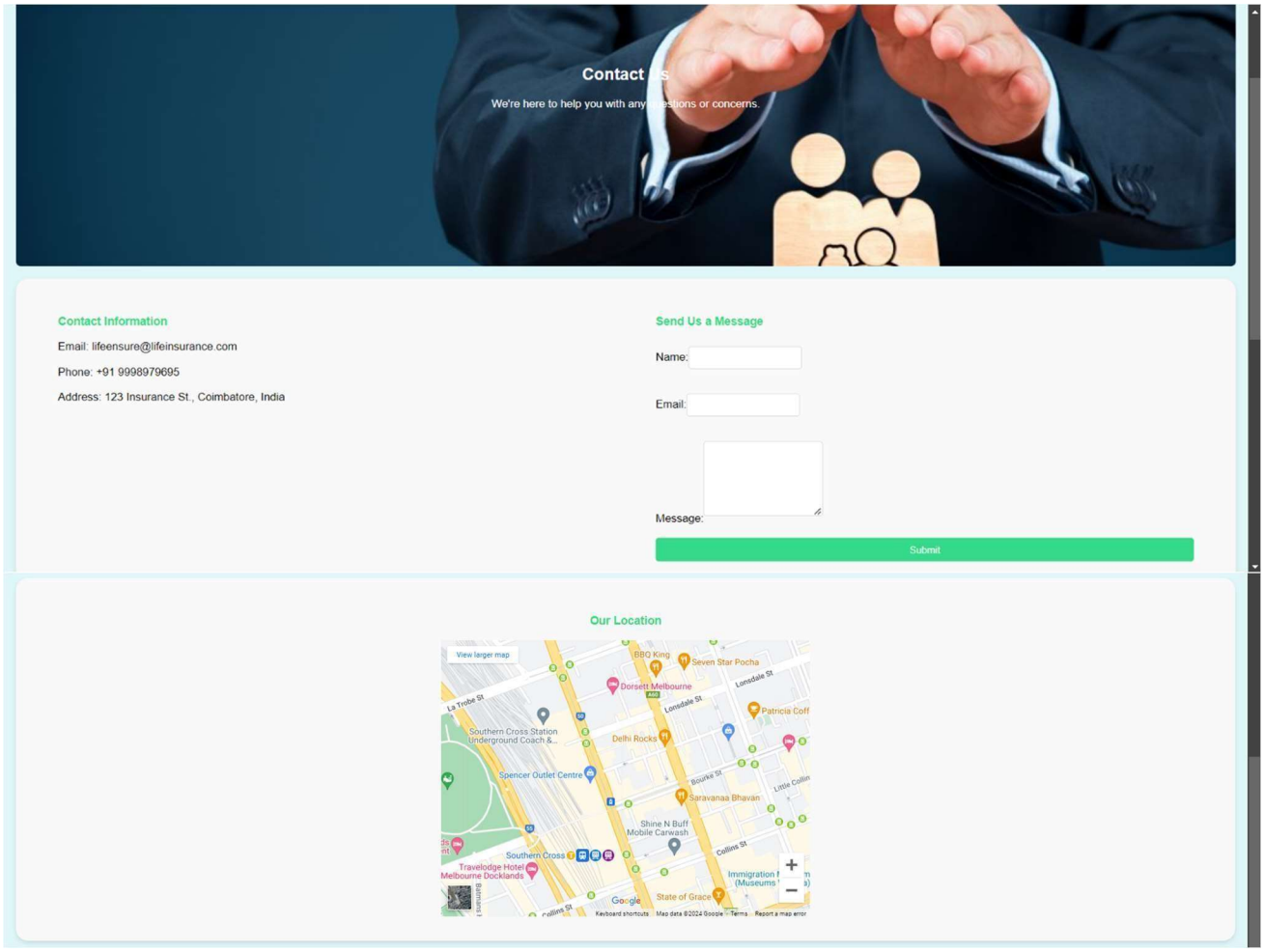


Fig 5.6 CONTACT US

7. INSURANCE FORM

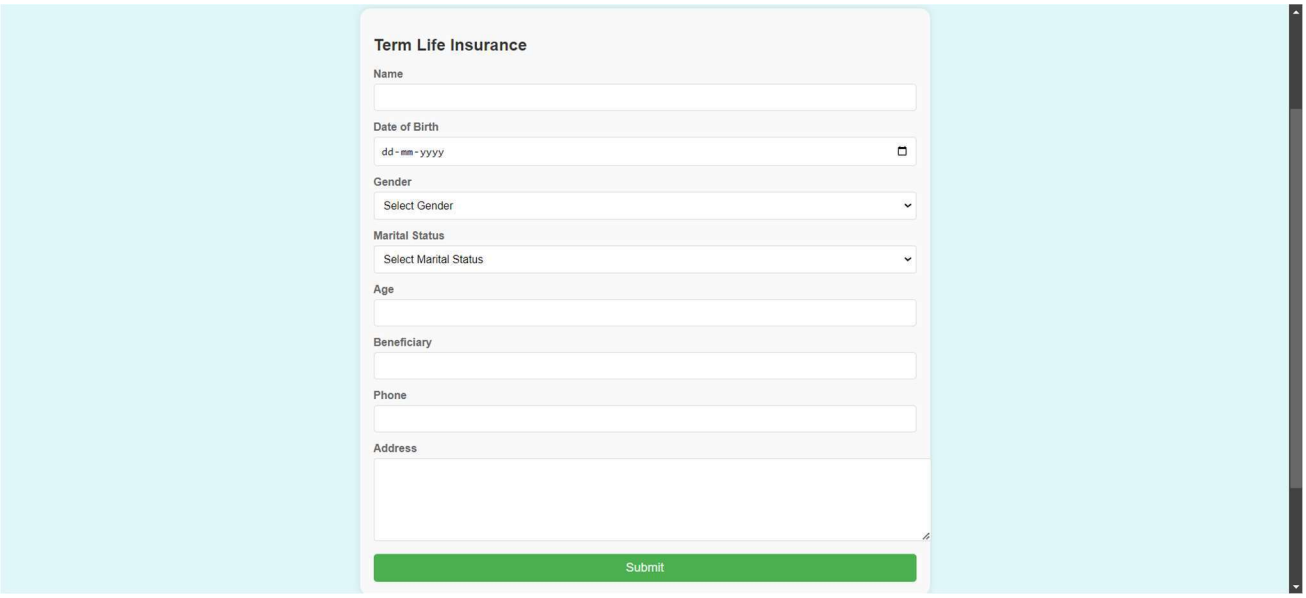


Fig 5.7 INSURANCE FORM

8. PAYMENT PAGE

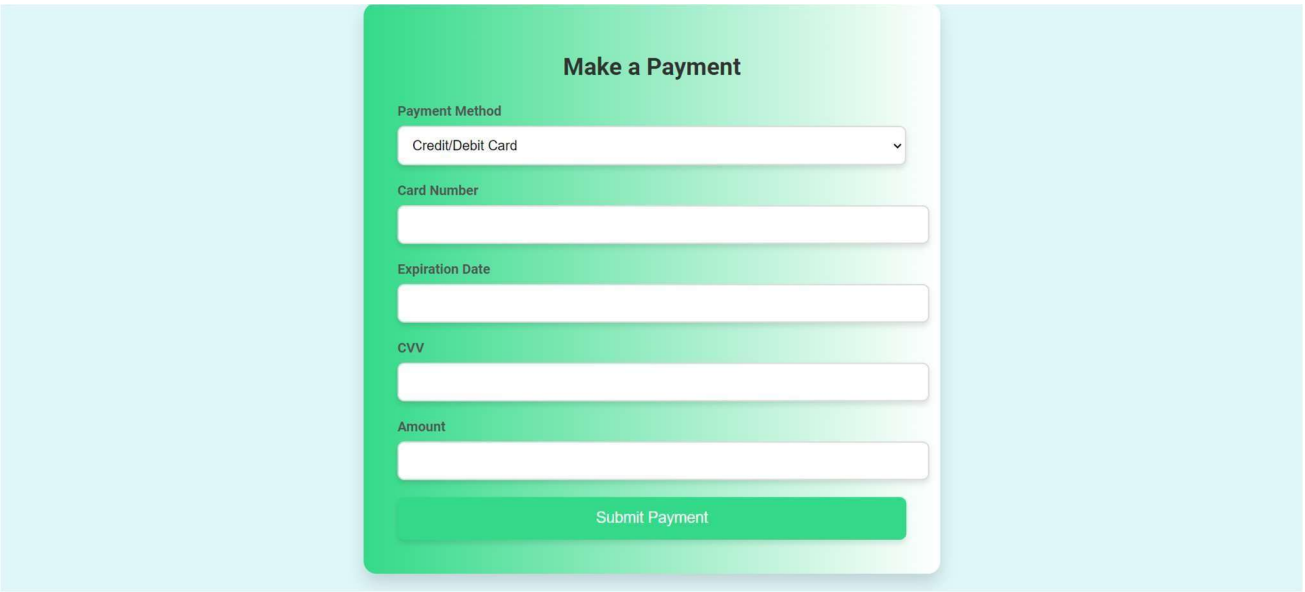


Fig 5.8 PAYMENT PAGE

9. ADMIN DASHBOARD

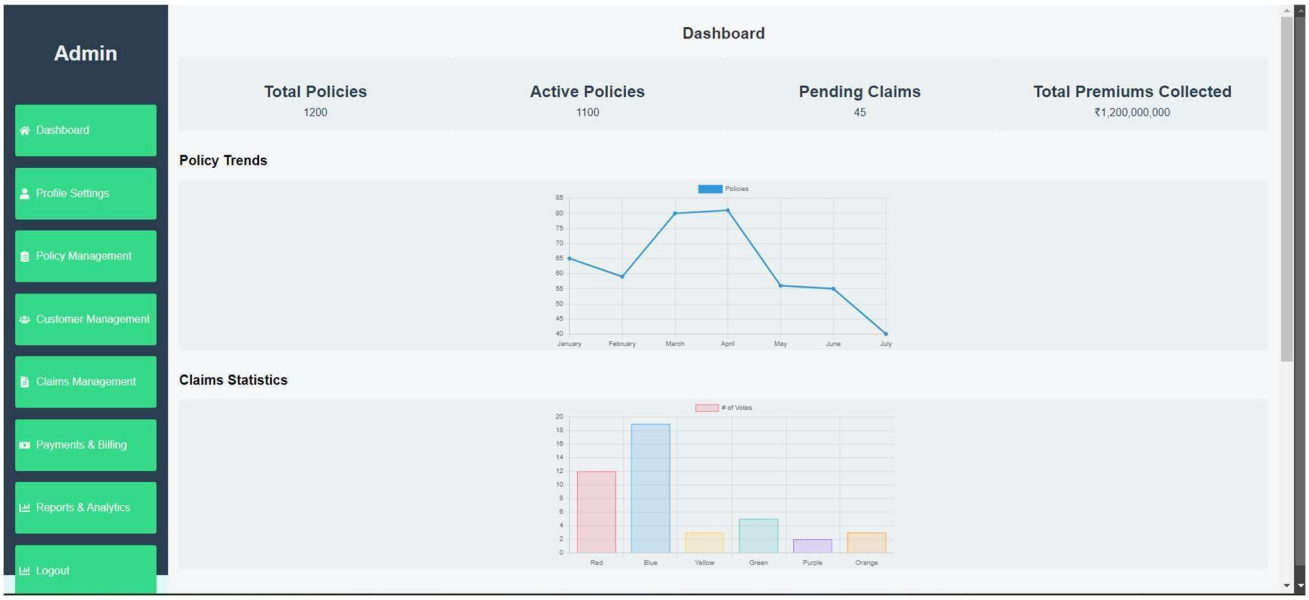


Fig 5.9 ADMIN DASHBOARD

10. CODING

Login:

```
const Login = () => {
  const [email, setEmail] = useState('');
  const [password, setPassword] = useState('');
  const [errors, setErrors] = useState({});
  const navigate = useNavigate();

  const validateForm = () => {
    const errors = {};
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;

    if (!email) {
      errors.email = 'Email is required';
    } else if (!emailRegex.test(email)) {
```

```
      errors.email = 'Invalid email format';
    }

    if (!password) {
      errors.password = 'Password is required';
    } else if (password.length < 6) {
      errors.password = 'Password must be at least 6 characters long';
    }

    setErrors(errors);
    return Object.keys(errors).length === 0;
  };

  const handleSubmit = (e) => {
    e.preventDefault();

    if (validateForm()) {
      navigate('/home');
    }
  };

  return (
    <div className="auth-container">
      <div className="auth-left">
        <div className="image-overlay"></div>

<img src={limage} alt="Login" />
      </div>
      <div className="auth-right">
        <img src={logo} alt="Logo" className="login-logo" />
        <div className="auth-box">
          <h2>Login</h2>
```

```
<form onSubmit={handleSubmit}>
  <div className="input-group">
    <label htmlFor="email">Email</label>
    <input
      type="email"
      id="email"
      name="email"
      value={email}
      onChange={(e) => setEmail(e.target.value)}
      required
    />
    {errors.email && <span className="error">{errors.email}</span>}
  </div>
  <div className="input-group">
    <label htmlFor="password">Password</label>
    <input
      type="password"
      id="password"
      name="password"
      value={password}
      onChange={(e) => setPassword(e.target.value)}
      required
    />
    {errors.password && <span className="error">{errors.password}</span>}
  </div>
  <button type="submit" className="btn">Login</button>
<div className="google-btn">
  <GoogleButton onClick={handleGoogleSignIn}  />
  </div>
```

```
            <div className="switch-link">
              <p>Don't have an account? <Link to="/register">Register</Link></p>
            </div>
            <div className="switch-link">
              <p>If Admin, Login as - <Link to="/adminlogin">Admin</Link></p>
            </div></form>
          </div></div>
        </div>);};

export default Login;
```

Home page:

```
const Home = () => {
  return (
    <main>
      <Header />
      <section id="home"  className="hero-section">
        <Slideshow />
        <div className="hero-content">
          <h2>Welcome to Our Life Ensure</h2>
          <p>Secure your future with our comprehensive life insurance plans.</p>
        </div>
      </section>
      <section id="services"  className="services-section">
        <h2>Our Services</h2>
        <div className="cards-container">
          <Link to="/termlife" className="card">
            <img src={termimg} alt="Term Life Insurance" className="card-image"/>
            <div  className="card-content">
              <h3>Term Life Insurance</h3>
```

```
            <p>Get covered for a specific term.</p>
          </div>
        </Link>
        <Link to="/wholelife" className="card">
          <img src={wholeimg} alt="Whole Life Insurance" className="card-
image"/>
          <div className="card-content">
            <h3>Universal Life Insurance</h3>
            <p>Flexible coverage and premiums.</p>
          </div>
        </Link>
        <Link to="/variablelife" className="card">
          <img src={variableimg} alt="Variable Life Insurance" className="card-
image"/>
          <div className="card-content">

            <h3>Variable Life Insurance</h3>
            <p>Investment options with coverage.</p>
          </div>
        </Link>

        <Link  to="/variableuniversalLife"  className="card">
          <img src={varuniimg} alt="Variable Universal Life Insurance"
className="card-image"/>
          <div className="card-content">
            <h3>Variable Universal Life Insurance</h3>
            <p>Combines features of universal life and variable life insurance.</p>
          </div>
        </Link>
```

```
        <Link to="/guaranteedissuelife" className="card">
          <img src={garentimg} alt="Guaranteed Issue Life Insurance"
className="card-image"/>
          <div className="card-content">
            <h3>Guaranteed Issue Life Insurance</h3>
            <p>Traditional life insurance due to health issues.</p>
          </div>
        </Link>
        <Link to="/simplifiedissuelife" className="card">
          <img src={simpleimg} alt="Simplified Issue Life Insurance"
className="card-image"/>
          <div className="card-content">

            <h3>Simplified Issue Life Insurance</h3>
            <p>Coverage amounts are determined based on the answers to the
questionnaire.</p>
          </div>
        </Link>

        <Link to="/finalexpense" className="card">
          <img src={finalimg} alt="Final Expense Insurance" className="card-
image"/>
          <div className="card-content">
            <h3>Final Expense Insurance</h3>
            <p>Cover funeral and burial expenses.</p>
          </div>
        </Link>
      </div>
    </section>
```

```
        <section  id="faq"  className="faq-section">
          <h2>Frequently Asked Questions</h2>
          <div className="faq">
            <h3>What is term life insurance?</h3>
            <p>Term life insurance provides coverage for a specific period, usually 10, 20,
      or 30 years.</p>
          </div>
          <div className="faq">
            <h3>How do I choose the right life insurance plan?</h3>

            <p>Consider your financial goals, budget, and the needs of your
      dependents.</p>
          </div>
        </section>
        <section  id="stats" className="stats-section">
          <h2>Our Achievements</h2>
          <div className="stat">
            <h3>10,000+</h3>

            <p>Policies Sold</p>
          </div>
        <section id="contact"  className="contact-section">
          <h2>Contact  Us</h2>
          <p>Email: lifeensure@lifeinsurance.com</p>
          <p>Phone: +123 456 7111</p>
        </section>
        <Footer />

      </main>);};
export default Home;
```

Services:

```
const Services = () => {
  return (
    <main>
      <Header/>
      <section id="services" className="services-section">
        <h2>Our Services</h2>
        <div className="cards-container">
          <Link to="/termlife" className="card">
            <img src={termimg} alt="Term Life Insurance" className="card-image"/>
            <div className="card-content">
              <h3>Term Life Insurance</h3>
              <p>Get covered for a specific term.</p>
            </div>
          </Link>
          <Link to="/wholelife" className="card">
            <img src={wholeimg} alt="Whole Life Insurance" className="card-image"/>
            <div className="card-content">
              <h3>Whole Life Insurance</h3>

              <p>Lifetime coverage and cash value.</p>
            </div>
          </Link>
          <Link to="/universallife" className="card">
            <img src={universalimg} alt="Universal Life Insurance" className="card-
image"/>
            <div className="card-content">
              <h3>Universal Life Insurance</h3>
            <p>Flexible coverage and premiums.</p></div>
```

```
        </Link>
        <Link to="/variablelife" className="card">
          <img src={variableimg} alt="Variable Life Insurance" className="card-
image"/>
          <div className="card-content">
            <h3>Variable Life Insurance</h3>
            <p>Investment options with coverage.</p>
          </div>
        </Link>
        <Link to="/variableuniversalLife" className="card">
          <img src={varuniimg} alt="Variable Universal Life Insurance"
className="card-image"/>
          <div className="card-content">
            <h3>Variable Universal Life Insurance</h3>
            <p>Combines features of universal life and variable life insurance.</p>
          </div>
        </Link>
        <Link to="/indexeduniversallife" className="card">
          <img src={indeximg} alt="Indexed Universal Life Insurance" className="card-
image"/>
          <div className="card-content">
            <h3>Indexed Universal Life Insurance</h3>
            <p>The cash value growth is linked to a stock market index.</p>
          </div></Link>
        <Link to="/simplifiedissuelife" className="card">
          <img src={simpleimg} alt="Simplified Issue Life Insurance" className="card-
image"/>
          <div className="card-content">
            <h3>Simplified Issue Life Insurance</h3>
            <p>Coverage amounts are determined based on the answers to the
questionnaire.</p>
          </div></Link>
```

```
<Link to="/grouplife" className="card">
                <img src={groupimg} alt="Group Life Insurance" className="card-image"/>
                <div className="card-content">
                  <h3>Group Life Insurance</h3>
                  <p>Basic coverage at a low or no cost to the employee.</p>
                </div>
               </Link>
               <Link to="/finalexpense" className="card">
                <img src={finalimg} alt="Final Expense Insurance" className="card-image"/>
                <div className="card-content">
                  <h3>Final Expense Insurance</h3>
                  <p>Cover funeral and burial expenses.</p>

                </div></Link>
              </div></section>
             <Footer/></main>);
        };

        export default Services;
```

Dashboard:

```
            const Dashboard = () => {
  const pieData = {
    labels: ['Total Policies', 'Active Policies', 'Pending Claims', 'Total Premiums
        Collected'],
    datasets: [
      {
        data: [1200, 1100, 45, 1200000],
        backgroundColor: ['#3498db', '#2ecc71', '#e74c3c', '#f39c12'],
        hoverBackgroundColor: ['#2980b9', '#27ae60', '#c0392b', '#e67e22']
      }
    ]
  };
```

```
const lineData = {
  labels: ['January', 'February', 'March', 'April', 'May', 'June', 'July'],
  datasets: [
    {
      label: 'Policies',
      data: [65, 59, 80, 81, 56, 55, 40],
      fill: false,

      backgroundColor: '#3498db',
      borderColor: '#3498db'
    }
  ]
};

const barData = {
  labels: ['Red', 'Blue', 'Yellow', 'Green', 'Purple', 'Orange'],
  datasets: [
    {
      label: '# of Votes',
      data: [12, 19, 3, 5, 2, 3],
      backgroundColor: [
        'rgba(255, 99, 132, 0.2)',
        'rgba(54, 162, 235, 0.2)',
        'rgba(255, 206, 86, 0.2)',
        'rgba(75, 192, 192, 0.2)',
        'rgba(153, 102, 255, 0.2)',
        'rgba(255, 159, 64, 0.2)'
      ],
```

```
return (
  <div  className="dashboard-container">
    <Sidebar />
    <div  className="dashboard-content">
      <h1>Dashboard</h1>
      <div className="metrics">
        <div className="metric">
          <h2>Total Policies</h2>
          <div>1200</div>
        </div>
        <div className="metric">
          <h2>Pending Claims</h2>
          <div>45</div>
        </div>
        <div className="metric">
          <h2>Total Premiums Collected</h2>
          <div>₹1,200,000,000</div>
        </div>
      </div>
      <div className="charts">
        <h2>Policy Trends</h2>
        <div className="chart">
          <Line data={lineData} />
        </div>
        <h2>Claims Statistics</h2>
        <div className="chart">
          <Bar data={barData} />
```

```
        </div>
        <h2>Overview</h2>
        <div  className="chart  pie-chart-container">
          <Pie data={pieData} />
        </div>
       </div>
       <h2>Recent Policies</h2>

     </div>
    </div>
  );
};

export default Dashboard;
```

Payment:

```
    const Payment = () => {
    const [formData, setFormData] = useState({
       cardNumber: '',
       expirationDate: '',
       cvv: '',
       paypalEmail: '',
       bankAccount: '',
       amount: ''

    });
    const [paymentMethod, setPaymentMethod] = useState('card');
    const [showModal, setShowModal] = useState(false);
    const navigate = useNavigate();
    const handleChange = (e) => { const { name, value } = e.target;
```

```
    setFormData({
        ...formData,
        [name]: value
    });
};

const handlePaymentMethodChange = (e) => {
    setPaymentMethod(e.target.value);
};

const handleSubmit = (e) => {
    e.preventDefault();
    console.log('Form submitted:', formData);
    setShowModal(true);
};

const closeModal = () => {
    setShowModal(false);
};

useEffect(() => {
    if (showModal) {
        const timer = setTimeout(() => {
            navigate('/home');

        }, 2000); // Redirect after 2 seconds

        return () => clearTimeout(timer); // Clean up the timer on component
unmount
    }
}, [showModal, navigate]);
```

```
    return (
        <div className="payment-container">
            <h1>Make a Payment</h1>
            <form onSubmit={handleSubmit}  className="payment-form">
                <div className="form-group">
                    <label htmlFor="paymentMethod">Payment Method</label>
                    <select id="paymentMethod" value={paymentMethod}
onChange={handlePaymentMethodChange} required>
                        <option value="card">Credit/Debit Card</option>
                        <option value="paypal">UPI Payment</option>
                        <option value="bank">Net Banking</option>
                    </select>
                </div>

                {paymentMethod === 'card' && (
                    <>
                        <div className="form-group">
                            <label htmlFor="cardNumber">Card Number</label>
                            <input type="text" id="cardNumber" name="cardNumber"
value={formData.cardNumber} onChange={handleChange} required />
                        </div>

                        <div className="form-group">
                            <label htmlFor="cvv">CVV</label>
                            <input type="text" id="cvv" name="cvv" value={formData.cvv}
onChange={handleChange} required />
                        </div>
                    </>
                )}
```

```
            {paymentMethod === 'paypal' && (
              <div className="form-group">
                <label htmlFor="paypalEmail">UPI Id</label>
                <input type="email" id="paypalEmail" name="paypalEmail"
value={formData.paypalEmail} onChange={handleChange} required />
              </div>
            )}
            {paymentMethod === 'bank' && (
              <div className="form-group">
                <label htmlFor="bankAccount">Bank Account Number</label>
                <input type="text" id="bankAccount" name="bankAccount"
value={formData.bankAccount} onChange={handleChange} required />
              </div>
            )}
            <div className="form-group">
              <label htmlFor="amount">Amount</label>
              <input type="number" id="amount" name="amount"
value={formData.amount} onChange={handleChange} required />
            </div>
            <button type="submit" className="submit-button">Submit
Payment</button>
          </form>
          {showModal && (
            <div className="modal">
              <div className="modal-content">
                <span className="close" onClick={closeModal}>&times;</span>
                <p>Your Payment is Successful</p></div></div>)}</div>);};
export default Payment
```

CHAPTER 6
BACKEND SYSTEM SPECIFICATION

In this chapter, the content discusses the software employed for constructing the website. This chapter provides a brief description of the software utilized in the project.

6.1 MySQL

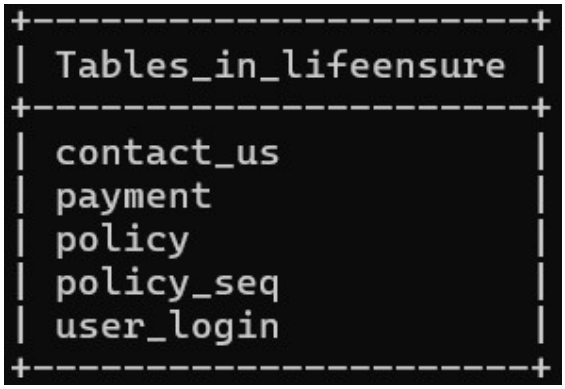


Fig 6.1 MySQL

Local storage is a type of web storage for storing data on the client side of a web browser. It allows websites to store data on a user's computer, which can then be accessed by the website again when the user returns. Local storage is a more secure alternative to cookies because it allows websites to store data without having to send it back and forth with each request. It is similar to a database table in that it stores data in columns and rows, except that local storage stores the data in the browser rather than in a database.

Local storage is often used to store user information such as preferences and settings, or to store data that is not meant to be shared with other websites.

It is also used to cache data to improve the performance of a website. Local storage is supported by all modern web browsers, including chrome, Firefox, Safari, and Edge. It is accessible through the browser's JavaScript API. Local storage is a powerful tool for websites to store data on the client side. It is secure, efficient, and can be used to store data that does not need to be shared with other websites.

Local Storage is a great way to improve the performance of a website by caching data. Local storage in web browsers allows website data to be stored locally on the user's computer. It is a way of persistently storing data on the client side, which is not sent to the server with each request. This allows users to store data such as preferences, login information, and form data without needing to send it to a server.

It is typically stored in a browser's cookie file, but it can also be stored in other locations such as HTML5 Local Storage and Indexed. The data stored in local storage is persistent and can be accessed by the website even if the user closes the browser or navigates to another page. It is a great way for websites to store user-specific data, as it is secure, reliable, and fast. It is also a great way for developers to store data that does not need to be sent to the server with each request.

One of the key benefits of using local storage is its reliability. Unlike server-side storage, which can be affected by network outages or other server issues, local storage is stored locally on the user's machine, and so is not affected by these issues. Another advantage of local storage is its speed. Because the data is stored locally, it is accessed quickly, as there is no need to send requests to a server.

6.2 REST API

A REST API (Representational State Transfer Application Programming Interface) is a popular architectural style for designing networked applications. It is based on a set of principles and constraints that allow for scalability, simplicity, and interoperability between systems.

Client-Server: Separated entities communicate over HTTP or a similar protocol, with distinct responsibilities and the ability to evolve independently.

Stateless: Each request from the client to the server must contain all the necessary information to understand and process the request. The server does not maintain any client state between requests.

Uniform Interface: The API exposes a uniform interface, typically using HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources. Resources are identified by URLs (Uniform Resource Locators).

Cacheable: Responses can be cached by the client or intermediaries to improve performance and reduce the load on the server.

Layered System: Intermediary servers can be placed between the client and server to provide additional functionality, such as load balancing, caching, or security.

6.3 SPRINGBOOT

Spring Boot is an open-source Java framework that simplifies the development of standalone, production-ready applications. It offers several advantages for building robust and scalable applications.

Simplified Configuration: Spring Boot eliminates the need for complex XML configuration files by leveraging sensible default configurations and annotations.

Embedded Server: Spring Boot includes an embedded server (e.g., Apache Tomcat, Jetty) that allows developers to create self-contained applications. This eliminates the need for external server installation and configuration, making it easier to package and deploy the application.

Dependency Management: Spring Boot incorporates the concept of starter dependencies, which are curated sets of libraries that provide commonly used functionalities. It simplifies dependency management and ensures that all required dependencies are included automatically, reducing configuration issues and potential conflicts.

Auto-Configuration: Spring Boot's auto-configuration feature analyzes the class path and automatically configures the application based on the detected dependencies. It saves developers from writing boilerplate configuration code, resulting in faster development and reduced code clutter.

Actuator: Spring Boot Actuator provides out-of-the-box monitoring and management endpoints for the application. It offers metrics, health checks, logging, and other management features, making it easier to monitor and manage the application in production environments.

DevOps Friendliness: Spring Boot's emphasis on simplicity and ease of use makes it DevOps friendly. It supports various deployment options, including traditional servers, cloud platforms, and containerization technologies like Docker. It also provides features for externalized configuration, making it easier to manage different environments.

6.4 SYSTEM ARCHITECTURE

The application adopts a contemporary and scalable three-tier architecture. It comprises the frontend layer, backend layer, and the database layer. Each of these layers fulfills a pivotal role in the application's comprehensive functionality, facilitating seamless communication and efficient data management.

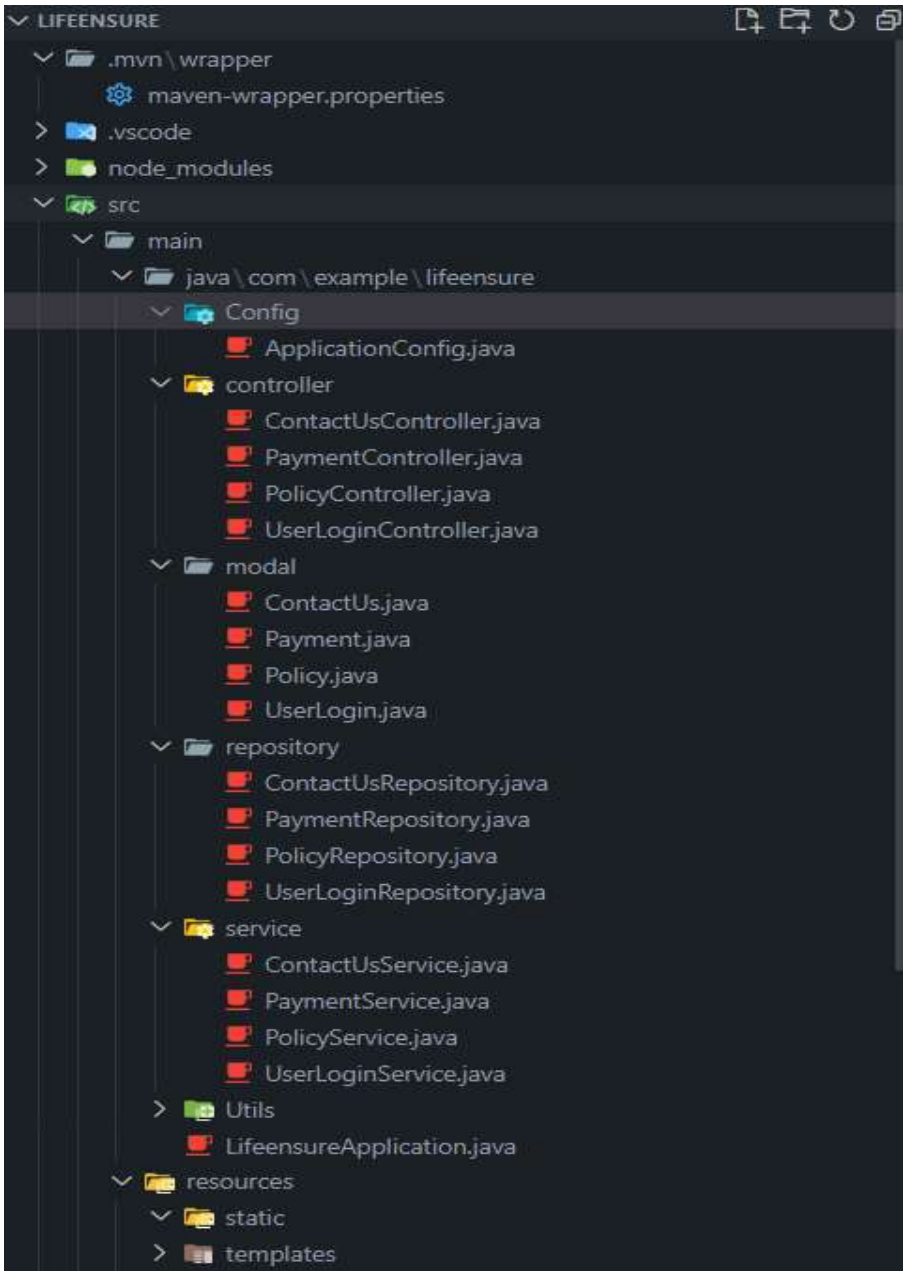


Fig 6.2 Backend System Architecture

The backend layer of the Life Insurance Portal is built upon the Spring Boot framework, a Java-based solution renowned for its capacity to simplify the development of resilient and scalable web applications. Spring Boot brings to the table a comprehensive array of features and libraries, offering streamlined solutions for managing HTTP requests, data persistence, implementing robust security measures, and seamlessly integrating with external systems. Within the application, the backend takes on the primary responsibility of crafting RESTful APIs. These APIs are meticulously designed to empower CRUD (Create, Read, Update, Delete) operations, catering to the dynamic world of employee tax details management. Furthermore, the backend is equipped to handle user management and authentication, ensuring a secure and personalized user experience for both administrators and employees. In pursuit of enhanced security and modularity, the backend is strategically structured according to the principles of Spring Boot architecture.

Spring Boot: Spring Boot is a Java framework that simplifies the process of building enterprise-grade applications. It provides a robust set of features and conventions for developing backend systems, including dependency management, configuration, and automatic setup. Spring Boot follows the principle of convention over configuration, reducing the amount of boilerplate code required.

REST API: The backend of the Life Insurance Portal exposes a RESTful API that allows the frontend to communicate with the server. REST (Representational State Transfer) is an architectural style for designing networked applications. It uses standard HTTP methods (GET, POST, PUT, DELETE) to perform CRUD (Create, Read, Update, Delete) operations on resources. The API endpoints define the URLs and request/response formats for interacting with the system.

Controller: In the application, controllers have a crucial role in managing incoming HTTP requests. Controllers map these requests to the appropriate methods within the system. API endpoints are defined by controllers, and they orchestrate the processing logic of incoming requests. Controllers act as the gateway between the frontend and backend, receiving user inputs, validating and processing data, interacting with services, and returning the relevant.

Services: Services within the application encapsulate the essential business logic. Responsible for orchestrating complex operations and facilitating interactions between different system components, these operations encompass data retrieval, validation, transformation, and storage. In this context, services manage policies, handle user authentication, and other application- specific functionalities, ensuring a seamless user experience.

Repositories: In the application, repositories serve as an abstraction layer for interacting with the database. They define the methods required for executing CRUD (Create, Read, Update, Delete) operations and querying the database using SQL or Object-Relational Mapping (ORM) frameworks like Hibernate. These repositories are instrumental in storing and retrieving policies and user-related data from the database, ensuring the persistence and accessibility of vital information.

Security: Security measures are a top priority within the application. Authentication and authorization protocols are diligently implemented to safeguard user data and system integrity. A robust security framework, such as Spring Security, is employed to manage user authentication and access control. It offers features such as user registration, login, password hashing, and role-based permissions, contributing to a secure and reliable application

This structured and modular approach ensures that the Life Insurance System is efficient, secure, and scalable, providing a comprehensive solution

6.5 ACTIVITY DIAGRAM

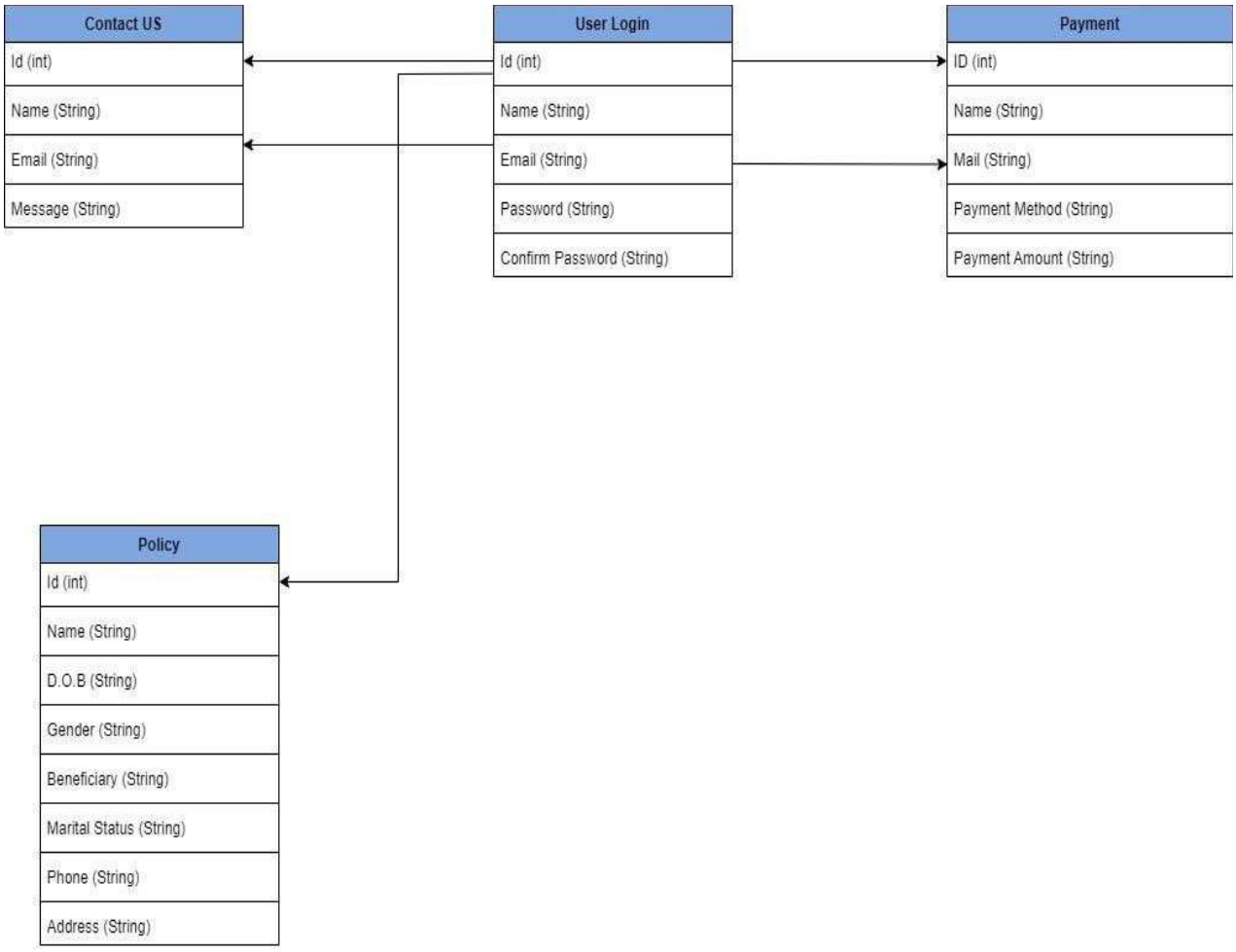


Fig 6.3 ACTIVITY DIAGRAM

Coding:

UserLoginController.java:

```
@RestController
@RequestMapping("/api/users")
public class UserLoginController
{ @Autowired
   private UserLoginService
   userLoginService @GetMapping
   public List<UserLogin> getAllUsers() {
      return
      userLoginService.getAllUsers();}

  @GetMapping("/{id}")

   public ResponseEntity<UserLogin> getUserById(@PathVariable int id) {
      Optional<UserLogin> userLogin = userLoginService.getUserById(id);
return userLogin.map(ResponseEntity::ok).orElseGet(()-> ResponseEntity.notFound().build());}

   @PostMapping

   public UserLogin createUser(@RequestBody UserLogin userLogin)
      { return userLoginService.createUser(userLogin);}

   @PutMapping("/{id}")

      public ResponseEntity<UserLogin> updateUser(@PathVariable int id, @RequestBody
UserLogin userDetails) {

      try {

         UserLogin updatedUser = userLoginService.updateUser(id,
         userDetails); return ResponseEntity.ok(updatedUser);

         } catch (RuntimeException e) {

         return ResponseEntity.notFound().build();}}
```

```
@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteUser(@PathVariable int id) {
    userLoginService.deleteUser(id);
    return ResponseEntity.noContent().build();}}
```

PolicyController.java:

```
@RestController
@RequestMapping("/api/policies")
public class PolicyController {
    @Autowired

    private PolicyService policyService;


    @Autowired

    private UserLoginService userLoginService;


    @GetMapping

    public List<Policy>
        getAllPolicies() { return
        policyService.getAllPolicies();}


    @GetMapping("/{id}")

    public ResponseEntity<Policy> getPolicyById(@PathVariable int id)
        { Optional<Policy> policy = policyService.getPolicyById(id);

    if (policy.isPresent()) {

            return ResponseEntity.ok(policy.get());

        } else {

            return ResponseEntity.notFound().build();}}


    @PostMapping
```

```
    public ResponseEntity<Policy> createPolicy(@RequestBody Policy
        policy) { if (policy.getUserLogin() != null &&
        policy.getUserLogin().getId() != 0) {
            Optional<UserLogin> userLogin =
userLoginService.getUserById(policy.getUserLogin().getId());
            if (userLogin.isPresent()) {
                policy.setUserLogin(userLogin.get
                ());
                Policy createdPolicy =
                policyService.createPolicy(policy); return
                ResponseEntity.ok(createdPolicy);
            } else {
                return ResponseEntity.badRequest().body(null);
            }} else {
            return ResponseEntity.badRequest().body(null);}}
    @PutMapping("/{id}")
    public ResponseEntity<Policy> updatePolicy(@PathVariable int id, @RequestBody Policy
policyDetails) {
        Policy updatedPolicy = policyService.updatePolicy(id,
        policyDetails); if (updatedPolicy != null) {
            return ResponseEntity.ok(updatedPolicy);
        } else {
            return ResponseEntity.notFound().build();}
    }@DeleteMapping("/{id}")
    public ResponseEntity<Void> deletePolicy(@PathVariable int id) {
        policyService.deletePolicy(id);
        return ResponseEntity.noContent().build();}}
```

PaymentController.java:

```
@RestController
@RequestMapping("/api/payments")
public class PaymentController {
   @Autowired

   private PaymentService paymentService;
   @PostMapping
   public ResponseEntity<Payment> addPayment(@RequestBody Payment payment)
      { Payment savedPayment = paymentService.addPayment(payment);
      return ResponseEntity.ok(savedPayment);}

   @GetMapping

   public ResponseEntity<List<Payment>> getAllPayments() {
      List<Payment> payments =
      paymentService.getAllPayments(); return
      ResponseEntity.ok(payments);}

   @GetMapping("/{id}")

   public ResponseEntity<Payment> getPaymentById(@PathVariable int id)
      { return paymentService.getPaymentById(id)
            .map(ResponseEntity::ok).orElse(ResponseEntity.notFound().build());}

  @DeleteMapping("/{id}")

   public ResponseEntity<Void> deletePayment(@PathVariable int id) {
      paymentService.deletePayment(id);
      return ResponseEntity.noContent().build();}}
```

ContactUsController.java:

```
@RestController
@RequestMapping("/api/contactus")
public class ContactUsController {
   @Autowired
```

```
private ContactUsService
contactUsService; @PostMapping
public ResponseEntity<ContactUs> addContactUs(@RequestBody ContactUs contactUs) {
   ContactUs savedContactUs = contactUsService.addContactUs(contactUs);
   return ResponseEntity.ok(savedContactUs);}

@GetMapping

public ResponseEntity<List<ContactUs>> getAllContactUs() {
   List<ContactUs> contactUsList = contactUsService.getAllContactUs();
   return ResponseEntity.ok(contactUsList);}

@GetMapping("/{id}")

public ResponseEntity<ContactUs> getContactUsById(@PathVariable int id)
   { return contactUsService.getContactUsById(id)
         .map(ResponseEntity::ok)

         .orElse(ResponseEntity.notFound().build());}


@DeleteMapping("/{id}")

public ResponseEntity<Void> deleteContactUs(@PathVariable int id) {
   contactUsService.deleteContactUs(id);
   return ResponseEntity.noContent().build();

}}
```

6.6 SECURITY AND AUTHENTICATION

Security and authentication lie at the heart of the application's robust infrastructure, ensuring the protection of sensitive data and upholding the integrity of the system.

User Authentication: User authentication is a fundamental pillar, allowing users, including

administrators and employees, to register securely by leveraging email and strong, hashed passwords. A robust login system verifies user credentials and controls access to the application. This ensures that only authorized users can access and manage sensitive tax information.

Data Encryption: Data encryption, both in transit and at rest, is a core component of the security framework. Secure communication channels protect data during interactions between the frontend and backend, while encryption of sensitive data in the database safeguards information in the event of a breach. This dual-layer encryption approach ensures that user data, including personal and financial details, remains confidential and protected.

Session Management: Session management maintains secure user sessions, preventing unauthorized access or data exposure. By implementing secure session tokens, the application ensures that users' sessions are both authenticated and securely managed throughout their interaction with the platform.

Security Frameworks: Utilizing security libraries and frameworks, such as Spring Security, enhances the efficiency of authentication and access control. Spring Security provides comprehensive features for securing applications, including user registration, login, password hashing, and role-based permissions, which are essential for protecting sensitive user data and controlling access within the system.

Protection Against Security Threats: The application includes protection against common security threats such as cross-site scripting (XSS) and SQL injection. Input validation and sanitization mechanisms are in place to ensure that user inputs are correctly processed and stored, preventing malicious attacks that could compromise the system.

User Awareness and Education: User awareness and education contribute to the overall security posture, ensuring that users are informed and capable of recognizing potential threats. The application provides users with guidelines on creating strong passwords, recognizing phishing attempts, and securely managing their accounts.

Regular Security Audits: Regular security audits and vulnerability assessments are conducted to proactively identify and address potential weaknesses. These audits help maintain the application's resilience against emerging security risks and ensure that any vulnerabilities are promptly mitigated.

Coding:

Jwt Authentication Filter Class:

```
Package com.example.emptax.config;
import java.io.IOException;
import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.web.authentication.WebAuthenticationDetailsSource;

import org.springframework.stereotype.Component;

import org.springframework.web.filter.OncePerRequestFilter;
import com.example.emptax.service.JwtService;
import com.example.emptax.service.UserRegisterDetailsService;
import jakarta.servlet.FilterChain;
import  jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.security.authentication.UsernamePasswordAuthentication
Token;

@Component

public class AuthenticationFilter extends OncePerRequestFilter
   { @Autowired
   private JwtService jwtService;
   @Autowired
   private UserRegisterDetailsService
   userRegisterDetailsService; @Override
   protected void doFilterInternal(HttpServletRequest request,

                                   HttpServletResponse response,
                                   FilterChain filterChain)
         throws ServletException, IOException {

      String authHeader = request.getHeader("Authorization");
```

```
        String token = null;
        String username = null;

        if (authHeader != null && authHeader.startsWith("Bearer ")) {
            token = authHeader.substring(7);
            username = jwtService.extractUsername(token);

        }

        if(username
!=null&&SecurityContextHolder.getContext().getAuthentication()
== null) {

            UserDetails                                    userDetails =
userRegisterDetailsService.loadUserByUsername(usernam
            e); if (jwtService.validateToken(token,
            userDetails)) {
                UsernamePasswordAuthenticationTokenauthToken = new
UsernamePasswordAuthenticationToken(userDetails,
                null, userDetails.getAuthorities());
                authToken.setDetails(new
WebAuthenticationDetailsSource().buildDetails(request));

            SecurityContextHolder.getContext().setAuthentication(authToken);}}

        filterChain.doFilter(request, response);}
```

Jwt Service Class:

```
package com.example.emptax.service;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import java.security.Key;
```

```
import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;
import io.jsonwebtoken.io.Decoders;
import io.jsonwebtoken.security.Keys;

@Component

public class JwtService {
    @Value("${application.jwt.secret-key}")
    private String secretKey;
    @Value("${application.jwt.token-expiration:1800000}")

    private long tokenExpiration;

    public String extractUsername(String token) {
        return extractClaim(token,
        Claims::getSubject);}

    public Date extractExpiration(String token) {

        return extractClaim(token, Claims::getExpiration);}

    public <T> T extractClaim(String token, Function<Claims, T> claimsResolver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);}

    private Claims extractAllClaims(String token)
        { return Jwts.parserBuilder()
                .setSigningKey(getSignKey())          .build()

                .parseClaimsJws(token)

                .getBody();

    private Boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());}
```

```
    public Boolean validateToken(String token, UserDetails userDetails)
      { final String username = extractUsername(token);
      return                (username.equals(userDetails.getUsername())&&
!isTokenExpired(token));}

    public String generateToken(String username) {
      Map<String, Object> claims = new HashMap<>();
      return createToken(claims, username);}
      private String createToken(Map<String,
      Object> claims, String username) { return
      Jwts.builder()
            .setClaims(claims)

            .setSubject(username)

            .setIssuedAt(new Date(System.currentTimeMillis()))

            .setExpiration(new Date(System.currentTimeMillis() + tokenExpiration))

            .signWith(getSignKey(), SignatureAlgorithm.HS256)

            .compact();

    private Key getSignKey() {

      byte[] keyBytes = Decoders.BASE64.decode(secretKey);
      return Keys.hmacShaKeyFor(keyBytes);}}
```

CHAPTER 7
INTEGRATION

7.1 INTRODUCTION TO AXIOS:

Axios is a widely used JavaScript library that simplifies the process of making HTTP requests, particularly in web development, where it is a preferred choice for interacting with APIs in both browser-based and server-side environments. The library is known for its promise-based architecture, which provides a straightforward way to handle asynchronous operations. By leveraging promises, developers can write cleaner and more readable code, utilizing. then() and .catch() for handling responses and errors, respectively, or opting for async/await syntax to achieve an even more intuitive flow of asynchronous code. One of the key advantages of Axios over native browser methods, like fetch, is its ability to handle automatic JSON data transformation. This means that when sending a request, Axios automatically converts JavaScript objects into JSON format, and similarly, it parses the JSON responses into JavaScript objects, saving developers from manual parsing or stringifying, which reduces potential errors and streamlines the process of working with APIs.

Moreover, Axios supports a full range of HTTP methods, including GET, POST, PUT, DELETE, PATCH, and HEAD, making it versatile for any RESTful API interaction. This flexibility allows developers to perform various CRUD (Create, Read, Update, Delete) operations with minimal code. Another powerful feature of Axios is the ability to set up request and response interceptors. Interceptors are essentially functions that are run before a request is sent or after a response is received, allowing for pre-processing of request data, adding custom headers, handling global errors, or even retrying failed requests. This feature is particularly useful in large applications where consistent request or response handling is necessary across different components or modules.

Axios also provides robust error handling capabilities, distinguishing between different types of errors such as network errors, timeouts, and HTTP status codes. This granularity enables developers to implement more precise error-handling logic, such as retrying requests, redirecting to error pages, or providing user-friendly error messages. Additionally, Axios allows for the cancellation of requests using the Cancel Token feature, which is particularly beneficial in scenarios where a request is no longer needed, such as when a user navigates away from a page before a request completes. This prevents unnecessary load on the server and enhances the user experience by ensuring that only relevant requests are processed.

Another significant aspect of Axios is its ability to be customized with configurations. Developers can set global defaults for base URLs, headers, and timeouts, which apply to all requests, or they can override these settings on a per-request basis. This level of customization simplifies the management of API interactions, especially in applications that interact with multiple APIs or require specific settings for different environments. Furthermore, Axios's compatibility with older browsers makes it a more reliable choice than the native fetch API in certain cases, as it uses XML Http Request under the hood in the browser environment, ensuring broader support. In summary, Axios is a highly versatile and developer-friendly library that simplifies the process of making HTTP requests, offering a rich set of features that enhance the efficiency, reliability, and readability of web applications.

7.2 IMPLEMENTATION OF AXIOS:

CODING:

LOGIN:

```
import axios from 'axios';
const Login = () => {
  const [email, setEmail] = useState('');

  const [password, setPassword] =
  useState(''); const [errors, setErrors] =
  useState({});  const navigate =
  useNavigate();
  const validateForm = ()
    => { const errors = {};
    const emailRegex = /^[^\s@]+@[^\s@]+\.[^\s@]+$/;
    if (!email) {
      errors.email = 'Email is required';

    } else if (!emailRegex.test(email)) {
      errors.email = 'Invalid email
      format';}

    if (!password) {

      errors.password = 'Password is required';

    } else if (password.length < 6) {

      errors.password = 'Password must be at least 6 characters long';}

    setErrors(errors);

    return Object.keys(errors).length === 0;};

  const handleSubmit = async (e) => {
    e.preventDefault();
  if(validateForm()
```

```
    ) { try {

const response = await axios.post('http://localhost:8080/api/v1/auth/authenticate',
{ email,
        password,});

       if (response.status === 200) {
        const { token } = response.data;

        localStorage.setItem('jwtToken', token);
        localStorage.setItem('isLoggedIn', 'true');
        page navigate('/home');}
} catch (error) {  console.error('Login
 failed:', error);
  setErrors({ server: 'Login failed. Please check your credentials and try again.'});
  }}}
```

POLICIES:

```
const fetchUserId = async () => {
    try {
      const token = localStorage.getItem('jwtToken');
      const response = await axios.get('http://localhost:8080/api/users', {
        headers: {
          'Authorization': Bearer ${token},},});
      setUserId(response.data.id);
    } catch (error) {
      console.error('Failed to fetch user ID:', error);}};
  fetchUserId();}, []);
  if (Object.keys(validationErrors).length === 0) { try {
      const token = localStorage.getItem('jwtToken');
      const response = await axios.post('http://localhost:8080/api/policies', {
        policyName: 'Term Life Insurance', // Add the policy name
        ...formData, // Spread the form data into the request body
        user_id: userId, // Replace with dynamic user ID}, {
        headers: {
          'Content-Type': 'application/json',
          'Authorization': Bearer ${token},},});
      setSuccessMessage('Form submitted successfully!');
      setTimeout(() => navigate('/payment'), 2000);  } catch (error) {
      console.error('There was an error submitting the form!', error}} else {
    setErrors(validationErrors);}};
```

PAYMENT:

```
const handleSubmit = async (e) => {
    e.preventDefault(); try {
        const token = localStorage.getItem('jwtToken');
        const response = await axios.post('http://localhost:8080/api/payments', {
            name: formData.name,
            email: formData.email,
            paymentmethod,
            paymentAmount: formData.amount,}, { headers: {
                'Content-Type': 'application/json',
                'Authorization': Bearer ${token},},});
        console.log('Payment response:', response.data);
        setShowModal(true);
    } catch (error) {
        console.error('There was an error making the payment!', error);}};
const closeModal = () => {
    setShowModal(false);};
useEffect(() => {
    if (showModal) {
        const timer = setTimeout(() => {
            navigate('/home');
        }, 2000);
        return () => clearTimeout(timer }
}, [showModal, navigate]);
```

CHAPTER 8
CONCLUSION

8.1 CONCLUSION:

In summary, the life insurance portal developed using React.js for the front end, Spring Boot for the back end, and Axios for API communication represents a modern, scalable solution tailored to meet the demands of today's digital insurance market.

The use of React.js as the front-end framework ensures a highly interactive and responsive user interface. React 's component-based architecture has allowed for the creation of reusable and modular UI elements, making the development process more efficient and enabling quicker iterations for future enhancements. The portal’s interface is designed to be intuitive, offering users a seamless experience when exploring insurance options, submitting applications, and managing their policies.

On the back end, Spring Boot has proven to be a powerful choice, offering a robust environment for building and deploying RESTful services. Its extensive ecosystem and integration capabilities have facilitated the implementation of key features such as authentication, data processing, and transaction management. Spring Boot's inherent scalability ensures that the portal can handle increasing loads, making it future-proof for growing user bases and additional functionality.

The integration with Axios has been pivotal in bridging the front end with the back end, enabling efficient and secure data exchange. Axios' promise-based HTTP client has provided smooth and consistent handling of asynchronous requests, resulting in a responsive application that can efficiently manage user interactions and backend processing without delays or disruptions.

Given the sensitivity of personal and financial information in the insurance industry, the portal has been designed with a strong emphasis on security. Both the front- end and back-end components adhere to best practices for data protection, including the use of HTTPS, secure token-based authentication, and thorough validation of user inputs. Compliance with industry regulations has been a priority, ensuring that the portal meets all necessary legal and security standards.

In conclusion, the life insurance portal is not just a digital interface for insurance products but a comprehensive platform that brings together cutting-edge technology, user- centric design, and industry best practices. It stands ready to serve customers effectively, driving business growth and providing a scalable solution that will adapt to future challenges and opportunities.

8.2 FUTURE SCOPE:

8.2.1 Enhanced Customer Experience

Personalized Recommendations: Implement AI-driven recommendation systems to suggest the best insurance plans based on individual customer profiles and their specific needs.

Chatbots and Virtual Assistants: Develop intelligent chatbots to provide instant support, answer queries, and assist customers through the application process.

8.2.2 Advanced Analytics and Reporting

Predictive Analytics: Utilize big data analytics to predict customer needs, optimize pricing models, and assess risk factors more accurately.

Real-Time Reporting: Enhance the reporting system to provide real-time insights and dashboards for both customers and administrators, improving decision-making processes.

8.2.3 Mobile Application Development

User-Friendly Mobile App: Develop a comprehensive mobile application to allow customers to manage their policies, make payments, and file claims on the go.

Push Notifications: Integrate push notifications to remind customers of premium due dates, policy renewals, and other critical updates

CHAPTER 9
REFERENCES

1) React.js Official Documentation:

Provides in-depth information on how to build user interfaces using React. Link:

https://reactjs.org/docs/getting-started.html

2) Spring Boot Official Documentation:

Comprehensive guide on building and deploying applications using Spring Boot. Link:

https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/

3) Axios GitHub Repository:

Offers insights into how Axios works, including its features and usage examples. Link:

https://github.com/axios/axios

4) Life Insurance Portal Development Best Practices:

Discusses best practices and considerations when developing a life insurance portal.

Link: https://www.javatpoint.com/spring-boot-rest-example (for REST API with Spring Boot)

5) Building Secure Web Applications with React and Spring Boot:

A guide on integrating React.js with Spring Boot while maintaining security standards.

Link: https://dzone.com/articles/spring-boot-react-full-stack-development-part-1

6) Case Study: Life Insurance Application Development:

A case study on building a life insurance application, which might offer insights into the challenges and solutions.

Link: https://www.simform.com/full-stack-web-development/

These references should provide a strong foundation for your report, helping to validate the technologies used and the methodologies applied in the development of the life insurance portal.