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

**Life Insurance portal**

**Software Design Pattern**

**A PROJECT REPORT**

***Submitted by***

**RITHAN KUMAR R (727822TUCS152)**

MOHAN KUMAR N (727822TUCS117)

PUGAZH UDHAYA AM (727822TUCS144)

MITHUN KUMAR K (727822TUCS112)

***in partial fulfilment for the award of the degree***

Of

#### BACHELOR OF TECHNOLOGY

**IN**

**Computer Science & Engineering**

**JULY - 2024**

**Life insurance portal**

**INTRODUCTION:**

Introducing our Life Insurance Portal, a modern solution designed to simplify the process of securing life insurance coverage. This portal offers a seamless and user-friendly experience, allowing individuals to explore various insurance options, compare quotes, and manage their policies online with ease. Built using React, the portal is optimized for performance and accessibility, ensuring a responsive and efficient interface across all devices. With a focus on transparency and user empowerment, our Life Insurance Portal provides clear information and tools to help users make informed decisions about their insurance needs. Whether you're looking to protect your family's future or secure your own peace of mind, our portal is here to guide you every step of the way.

**OBJECTIVE:**

The primary objective of life insurance portal is to maintain optimal inventory levels to ensure uninterrupted production and sales operations while minimizing the costs associated with holding insurance. This involves striking a balance between having sufficient stock to meet customer demand and avoiding excessive inventory that ties up capital and incurs storage costs. Effective life insurance management aims to enhance operational efficiency, reduce lead times, improve order fulfillment rates, and ultimately increase customer satisfaction. By aligning inventory levels with demand forecasts and production schedules, businesses can minimize stockouts, overstock situations, and wastage, thereby optimizing overall supply chain performance and profitability.

**USER FLOW CHART:**

Is Logged in?

Login

No

Home Page

Yes

Dashboard of the company

Life insurance schemes

User access

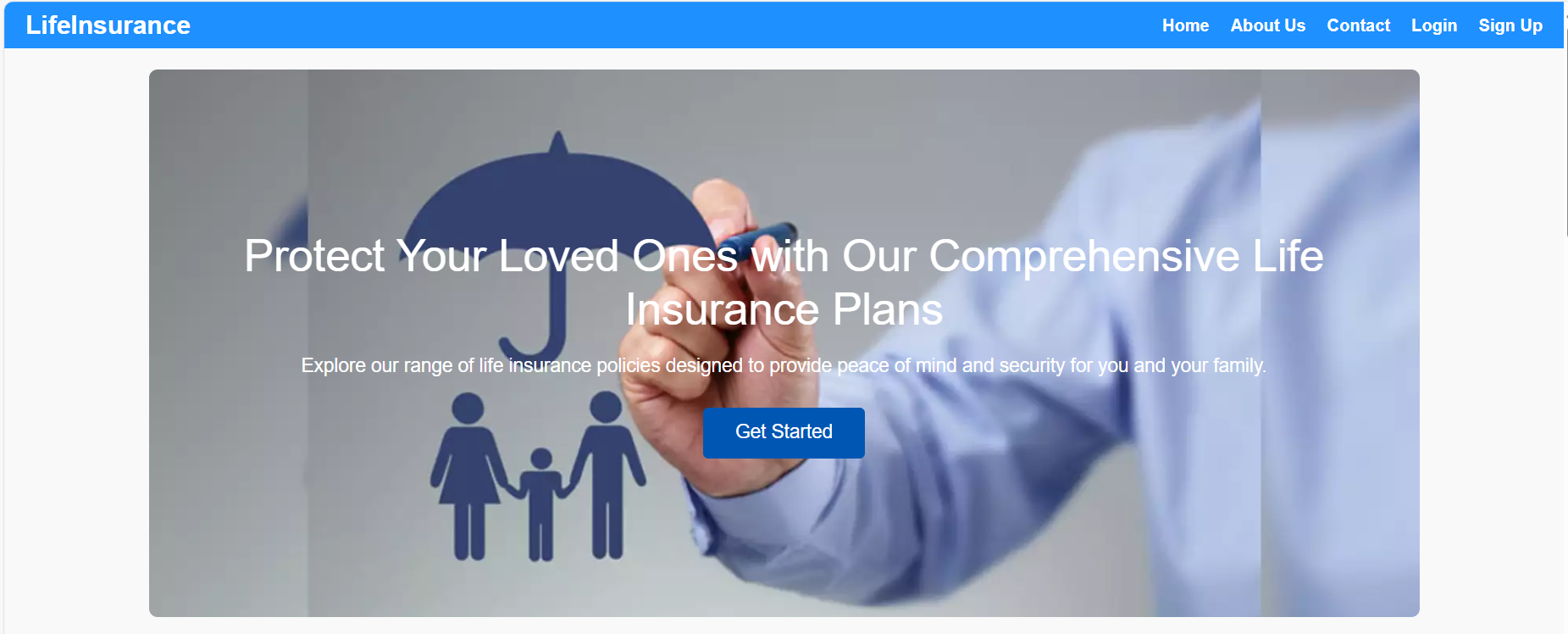
Report analysis

No

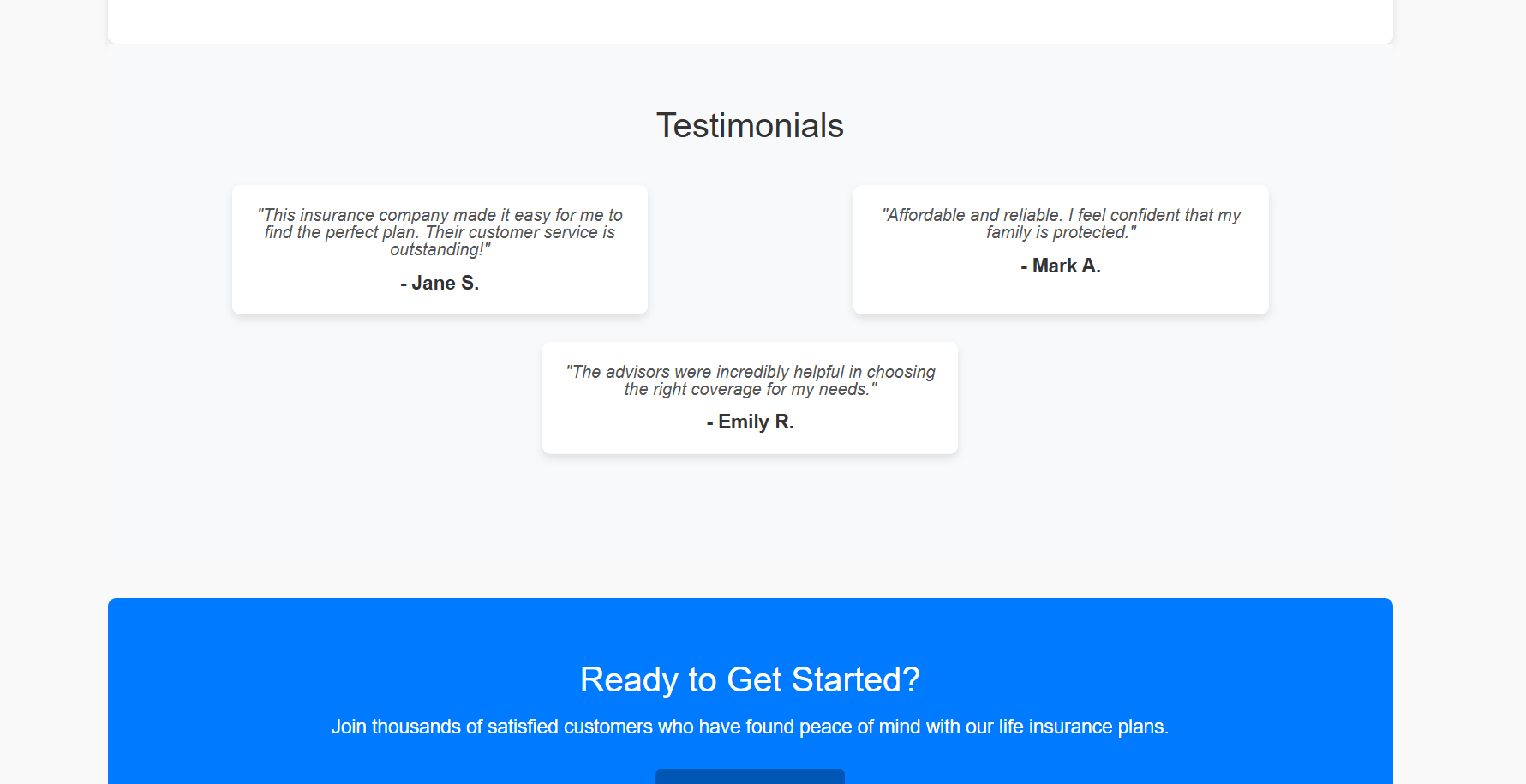
Yes

Logout

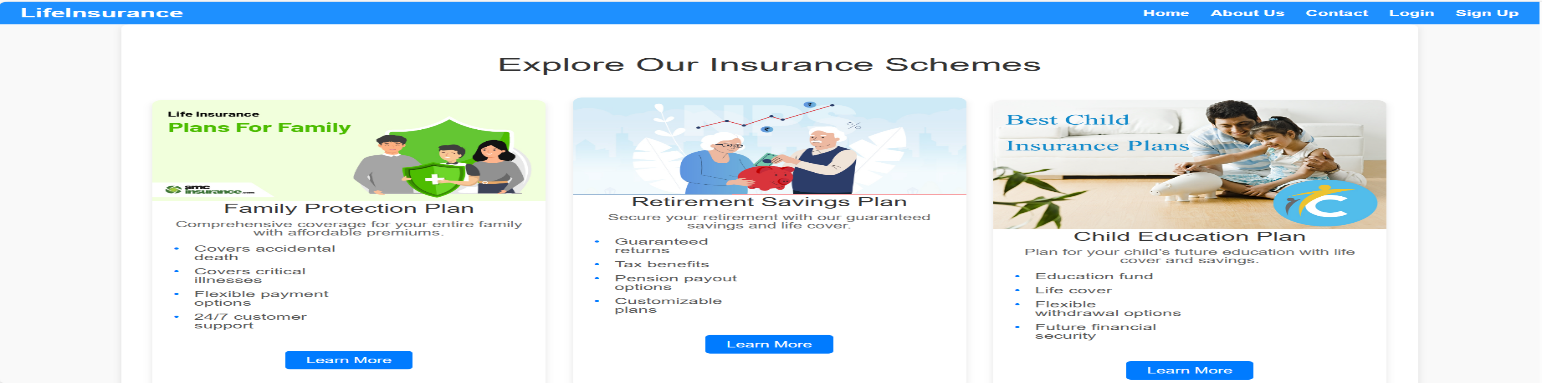
**Home page:**

****

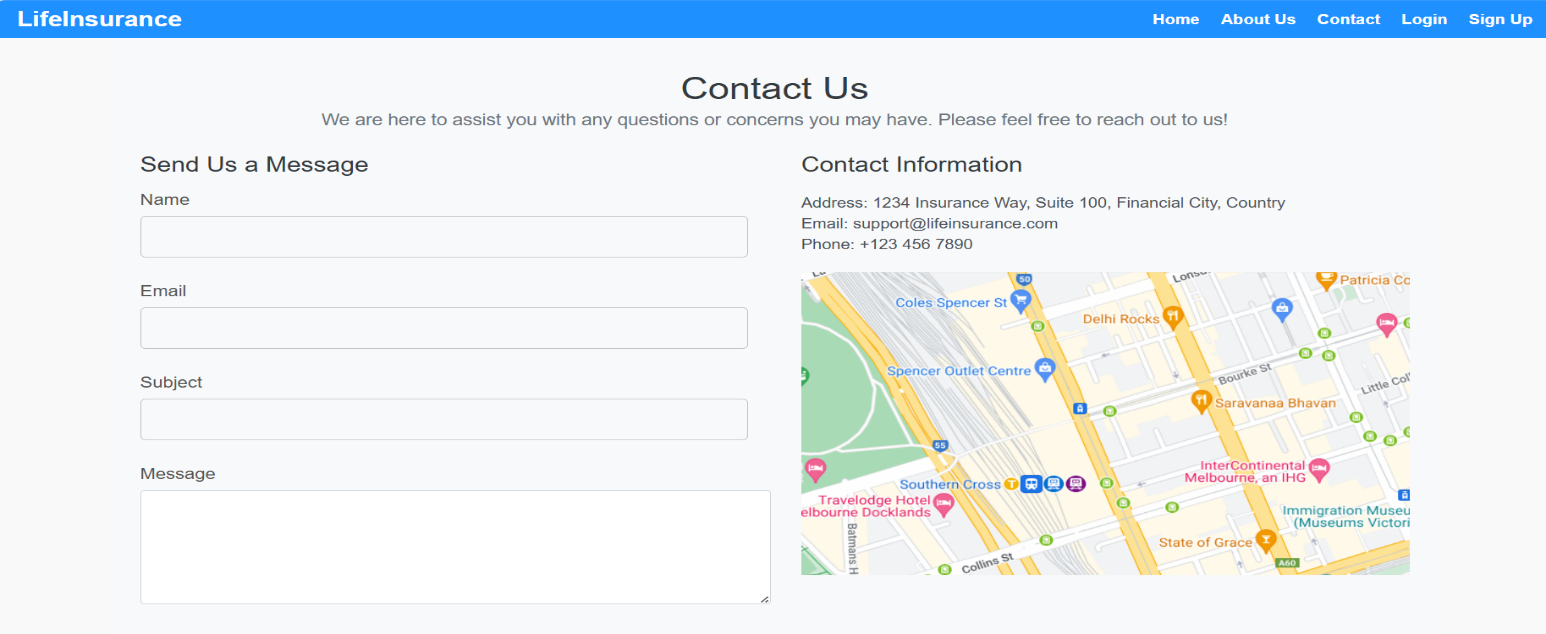
**User Navigation Bar**

****

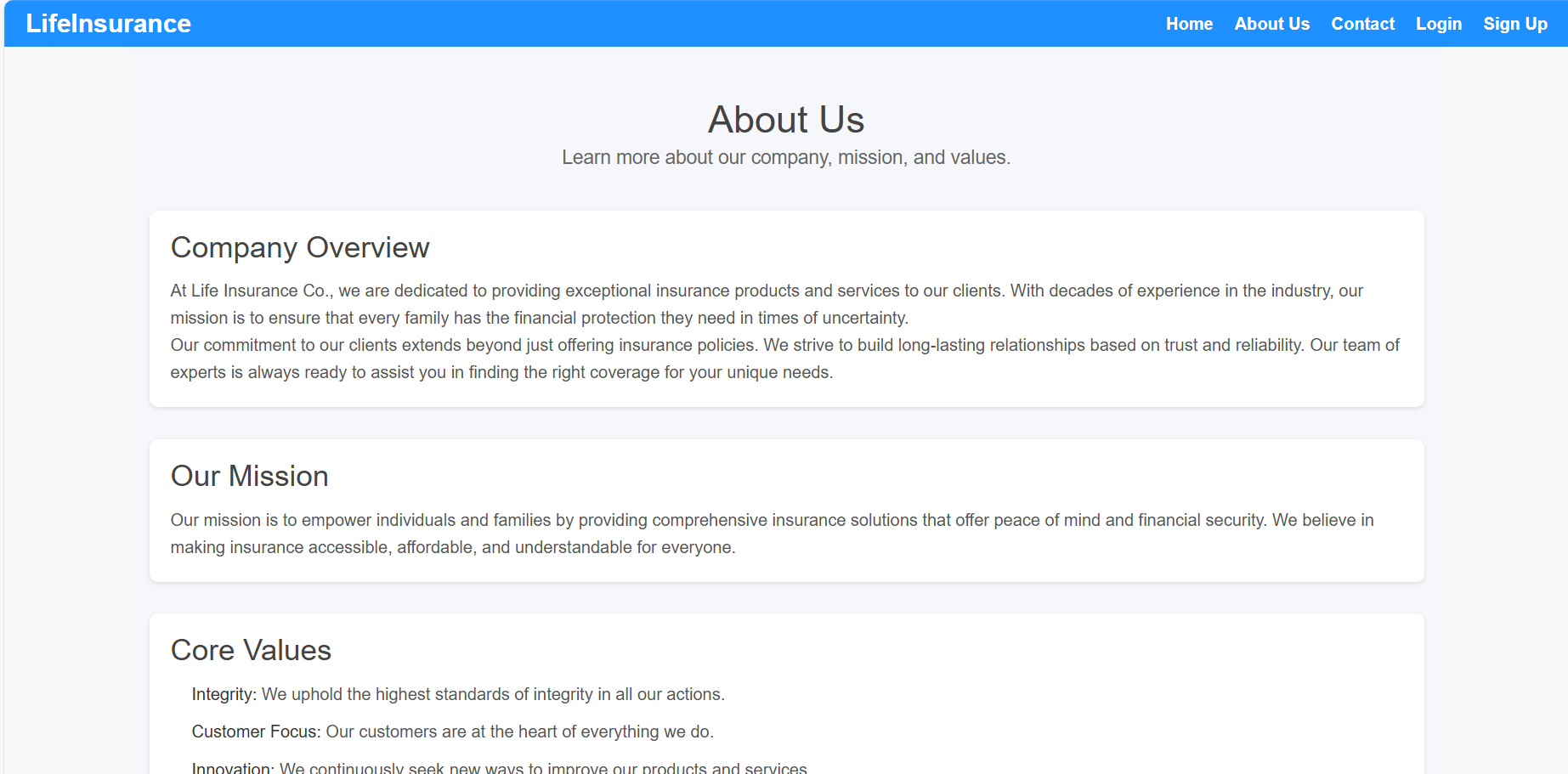
**Insurance Scheme**

****

**Contact Info**



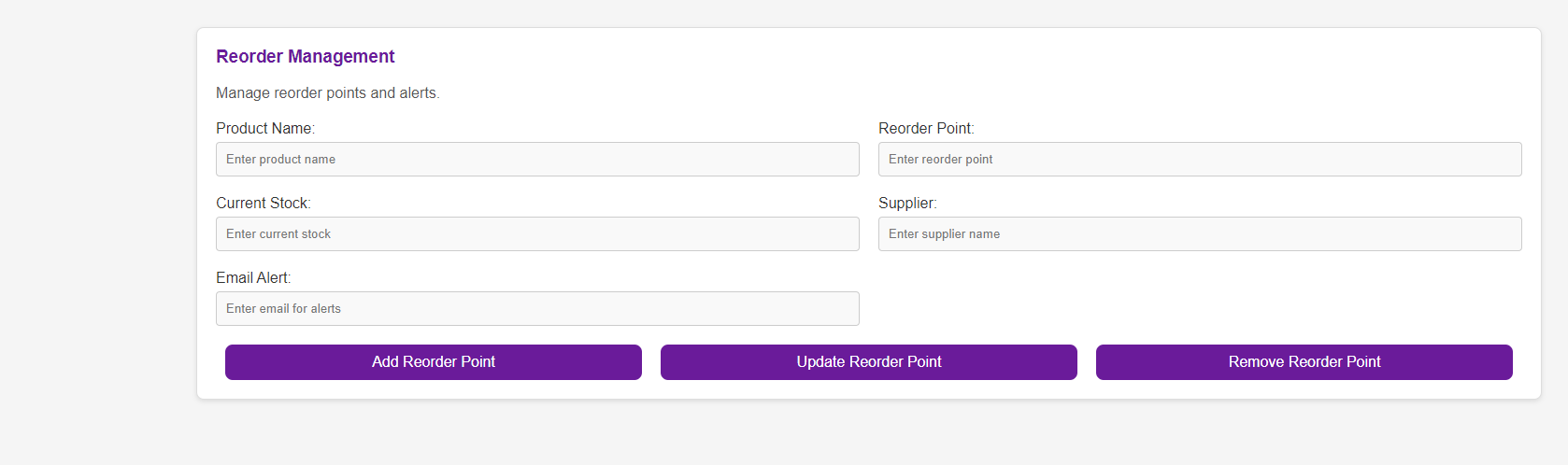
About us



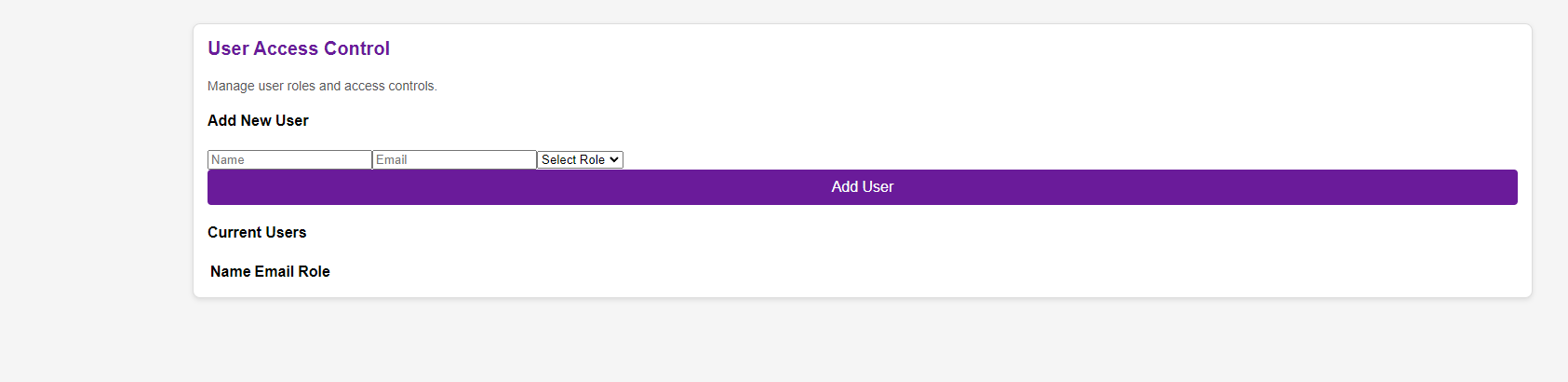
User Details



**Record Management**



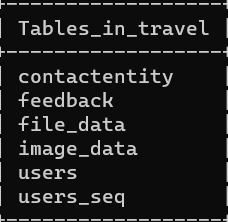
**User Access**



**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.

### **POSTGRESQL**



#### Fig2.1 MySQL

Local storage is a type of web storage for storing data on the client side ofaweb 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 informationsuch 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 JavaScriptAPI. 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 storageis 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.

### **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 andserver to provide additional functionality, such as load balancing, caching, or security.

### **SPRINGBOOT**

Spring Boot is an open-source Java framework that simplifies the developmentof 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.

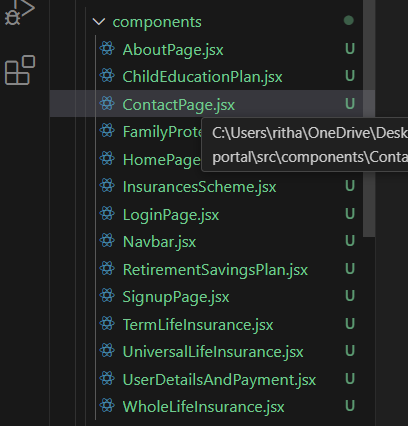
Actuator: Spring Boot Actuator provides out-of-the-box monitoring and management endpoints for the application. It offers metrics, health checks, logging, and other management features, making it easier to monitor and manage the application in production environments.

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.

# **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.

### **BACKEND**

****

#### Fig3.1 Backend System Architecture

Chess academy :Customized Gif application, the backend layer 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 photograph details management. Furthermore, the backend is equipped to handle user management and authentication, ensuring a secure and personalized user experience for both administrators and customers. In pursuit of enhanced security and modularity, the backend is strategically structured according to the principles of springboot 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 Open Library system 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 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 responses.

#### 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 orders, 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. 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 customize gift and orders-related data from the database, ensuring the persistence and accessibility of vital information.

#### Data Transfer Objects(DTOs):

Data Transfer Objects (DTOs) play a pivotal role in enabling data exchange between the frontend and backend layers of the application. These objects define the structure and format of data shared in API requests and responses. DTOs are employed to represent destination and order details, user information, and other relevant data that is transferred between the frontend and backend, ensuring seamless .

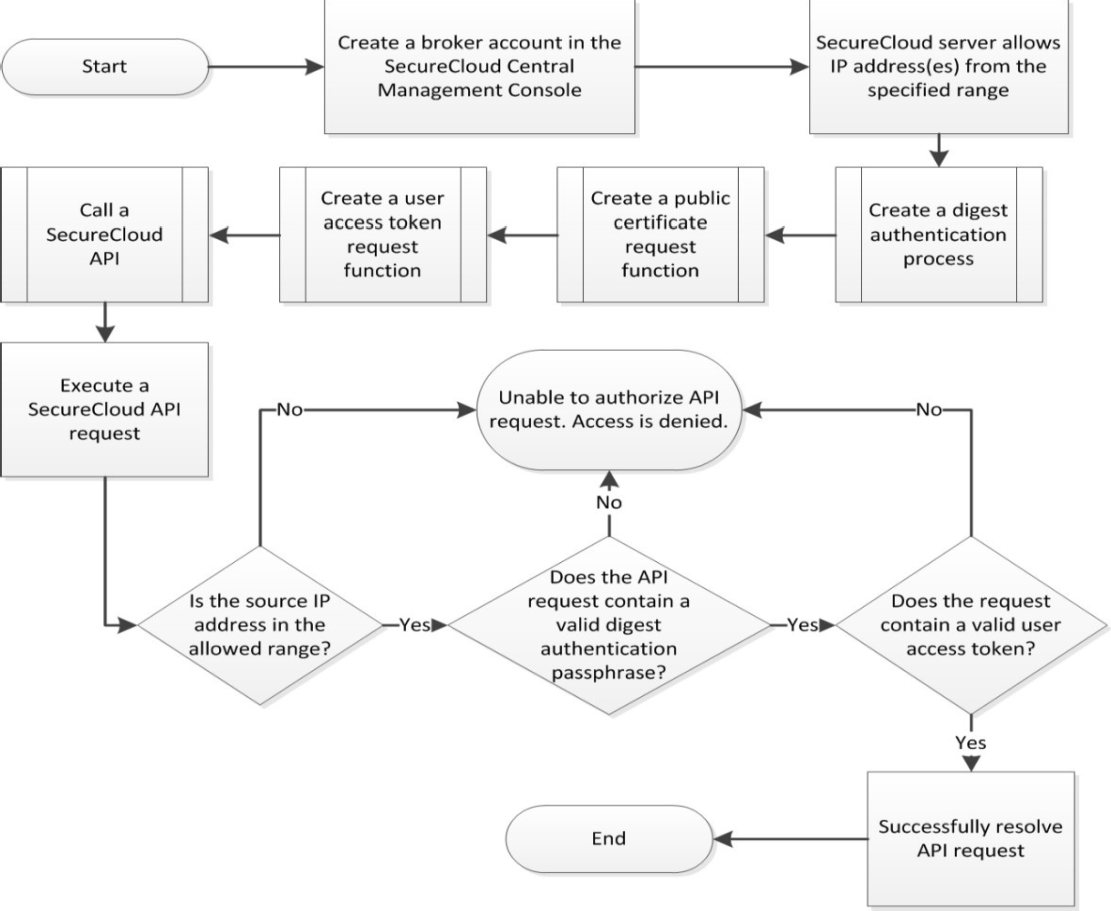
#### Security:

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.

# **IMPLEMETATION AND FUNCTIONALITY**

The Chess academy :Customized Gif application's backend is the linchpin of effective place details management. It provides administrators with a comprehensive set of tools to seamlessly add, update, and remove places. The user-friendly interface simplifies their tasks, reducing the learning curve. Security is at priority ,with stringent access controls and user authentication measures protecting sensitive data.

#### API Request



**Fig4.1 REST API flow chart**

A Representational State Transfer (REST) API plays a pivotal role in the architecture of the Application, providing a structured and efficient means of communication between the frontend and backend components. In this context, the REST API serves as the intermediary that enables the exchange of data and requests, making it a cornerstone of the application's functionality.

The REST API of the Application adheres to RESTful principles, which are centered around a set of stateless operations for creating, retrieving, updating,anddeletingdata.Itdefinesaclearstructurefortheendpoints,with each endpoint corresponding to a specific resource or action. For instance, endpoints might include "GET /user" to retrieve a list of available places, "POST /user" to add places, and "DELETE /auth/{user ID}" to remove places.

By adopting RESTful design, the API simplifies interactions with theapplication, ensuring that users can easily access place information and complete transactions. It provides data in a format that is widely understood, typically in JSON, allowing for seamless integration with various client applications, including web and mobile interfaces.

In addition to its role in enabling user interactions, the REST API is a fundamental component for potential future developments. It opens the door to third- party integrations, such as payment gateways, external inventory systems, or analytics services, that can enhance the application's functionality and expand its capabilities. Furthermore, the API empowers the application to be scalable, ensuring that it can accommodate growing user bases and evolving feature sets.

The REST API in the Application is not merely a technical component; it's the conduit through which the application's core functionalities are exposed and extended, ultimately contributing to a seamless and versatile user experience.

### **CRUD OPERATION**

In the application ,the implementation of CRUD(Create, Read, Update, Delete) operations is fundamental to the efficient management of the product inventory. The "Create" operation allows administrators to add new places by entering comprehensive details ,which are then validated for accuracy and completeness before being securely stored in the database. "

The "Update" operation empowers administrators to modify place information, and a user-friendly interface ensures that this process is intuitive. Stringent data validation criteria are maintained to guarantee the accuracy and reliability of the edited information, which is subsequently updated in the database.

The "Delete" operation provides administrators with the means to remove places from the inventory, incorporating a confirmation prompt to prevent accidental deletion. Once confirmed, the place’s information is securely deleted from the database.

#### Coding:

**Controller Class:**

#### package com.example.giftcraft.controller;

#### import org.springframework.beans.factory.annotation.Autowired;

#### import org.springframework.http.ResponseEntity;

#### import org.springframework.web.bind.annotation.GetMapping;

#### import org.springframework.web.bind.annotation.PostMapping;

#### import org.springframework.web.bind.annotation.RequestBody;

#### import org.springframework.web.bind.annotation.RequestMapping;

#### import org.springframework.web.bind.annotation.RestController;

#### import com.example.giftcraft.dto.Request.AuthenticationRequest;

#### import com.example.giftcraft.dto.Request.RegisterRequest;

#### import com.example.giftcraft.dto.Response.AuthenticationResponse;

#### import com.example.giftcraft.entity.UserRegister;

#### @RestController

#### @RequestMapping("/auth")

#### @RequiredArgsConstructor

#### public class AuthenticationController {

#### @Autowired

#### private UserRegisterRepository userRegisterRepository;

#### private final AuthenticationService service;

#### @PostMapping("/register")

#### public ResponseEntity<String> register(

#### @RequestBody RegisterRequest request

#### ) {

#### service.register(request);

#### return ResponseEntity.ok("Registered Successfully");

#### }

#### @GetMapping("/getregister")

#### public List<UserRegister> getregister()

#### {

#### return userRegisterRepository.findAll();

#### 

#### }

#### @PostMapping("/authentication")

#### public ResponseEntity<AuthenticationResponse> authenticate(@RequestBody AuthenticationRequest request)

#### {

#### return ResponseEntity.ok(service.authenticate(request));

#### 

#### }

#### 

#### Entity Class:

#### package com.example.giftcraft.entity;

#### import com.example.giftcraft.entity.enumerate.Role;

#### import jakarta.persistence.Entity;

#### import jakarta.persistence.EnumType;

#### import jakarta.persistence.Enumerated;

#### import jakarta.persistence.GeneratedValue;

#### import jakarta.persistence.GenerationType;

#### import jakarta.persistence.Id;

#### import jakarta.persistence.Table;

#### import lombok.AllArgsConstructor;

#### import lombok.Builder;

#### import lombok.Data;

#### import lombok.NoArgsConstructor;

#### @Entity

#### @Data

#### @AllArgsConstructor

#### @NoArgsConstructor

#### @Builder

#### @Table(name = "\_user")

#### public class UserRegister {

#### @Id

#### @GeneratedValue(strategy = GenerationType.IDENTITY)

#### private Long userid;

#### private String username;

#### private String emailid;

#### private String password;

#### private String mobileno;

#### @Enumerated(EnumType.STRING)

#### private Role roles;

#### }

#### Service Class:

package com.example.giftcraft.service;

import org.springframework.beans.factory.annotation.Autowired;

import org.springframework.security.authentication.AuthenticationManager;

import org.springframework.security.authentication.UsernamePasswordAuthenticationToken;

import org.springframework.security.crypto.password.PasswordEncoder;

import org.springframework.stereotype.Service;

import com.example.giftcraft.dto.Request.AuthenticationRequest;

import com.example.giftcraft.dto.Request.RegisterRequest;

import com.example.giftcraft.dto.Response.AuthenticationResponse;

import com.example.giftcraft.entity.UserRegister;

import com.example.giftcraft.entity.enumerate.Role;

import com.example.giftcraft.repository.UserRegisterRepository;

import lombok.RequiredArgsConstructor;

@Service

@RequiredArgsConstructor

public class AuthenticationService {

@Autowired

private final UserRegisterRepository userRegisterRepository;

private final PasswordEncoder passwordEncoder;

private final JwtService jwtService;

private final AuthenticationManager authenticationManager;

@SuppressWarnings("null")

public AuthenticationResponse register(RegisterRequest request) {

Role role;

if (request.getRoles() != null) {

role = request.getRoles();

} else {

role = Role.USER;

}

UserRegister user = UserRegister.builder()

.username(request.getUsername())

.emailid(request.getEmailid())

.mobileno(request.getMobileno())

.password(passwordEncoder.encode(request.getPassword()))

.roles(role)

.build();

userRegisterRepository.save(user);

String jwtToken = jwtService.generateToken(request.getUsername());

return AuthenticationResponse.builder()

.token(jwtToken)

.build();

}

public AuthenticationResponse authenticate(AuthenticationRequest request) {

authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(

request.getUsername(),

request.getPassword()

)

);

String jwtToken = jwtService.generateToken(request.getUsername());

return AuthenticationResponse.builder()

.token(jwtToken)

.build();

}

}

#### Repository Interface:

package com.example.giftcraft.repository;

import java.util.Optional;

import org.springframework.data.jpa.repository.JpaRepository;

import org.springframework.data.jpa.repository.Query;

import org.springframework.stereotype.Repository;

import com.example.demo.entity.Users;

import java.util.List;

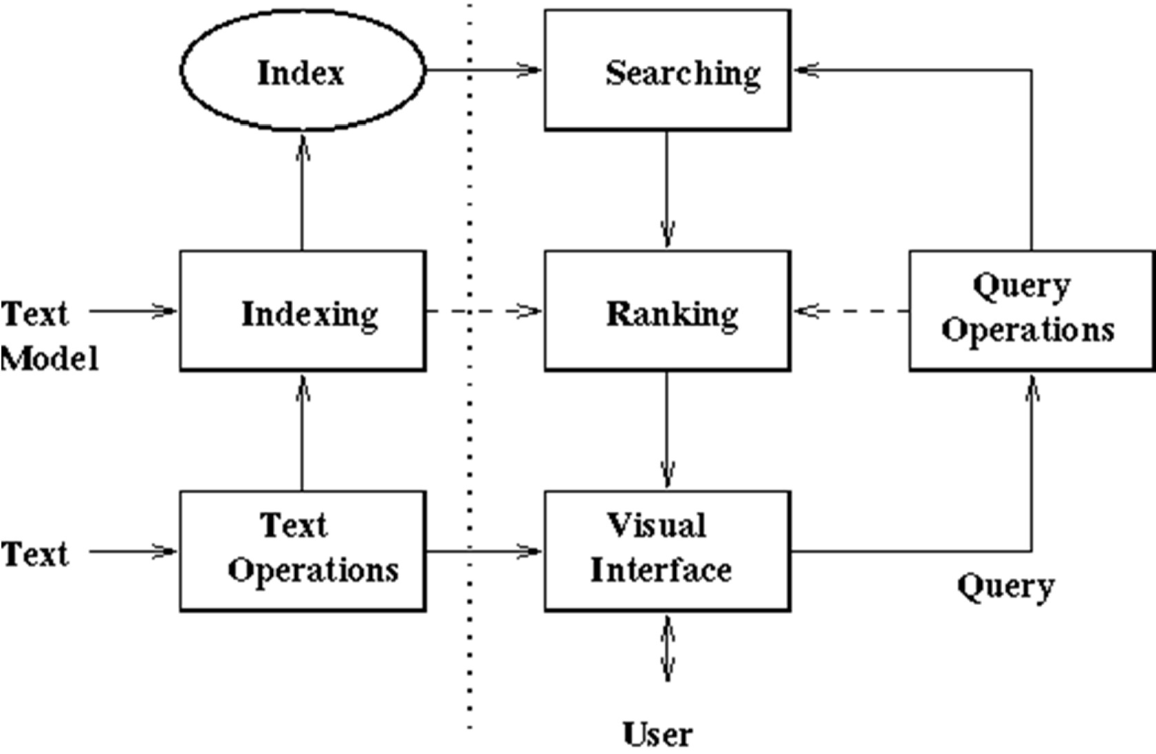
@Repository

public interface UsersRepository extends JpaRepository<Users, Integer> {

Optional<Users>findByEmail(String email);

}

### **DATA RETRIEVEL PROCESS**

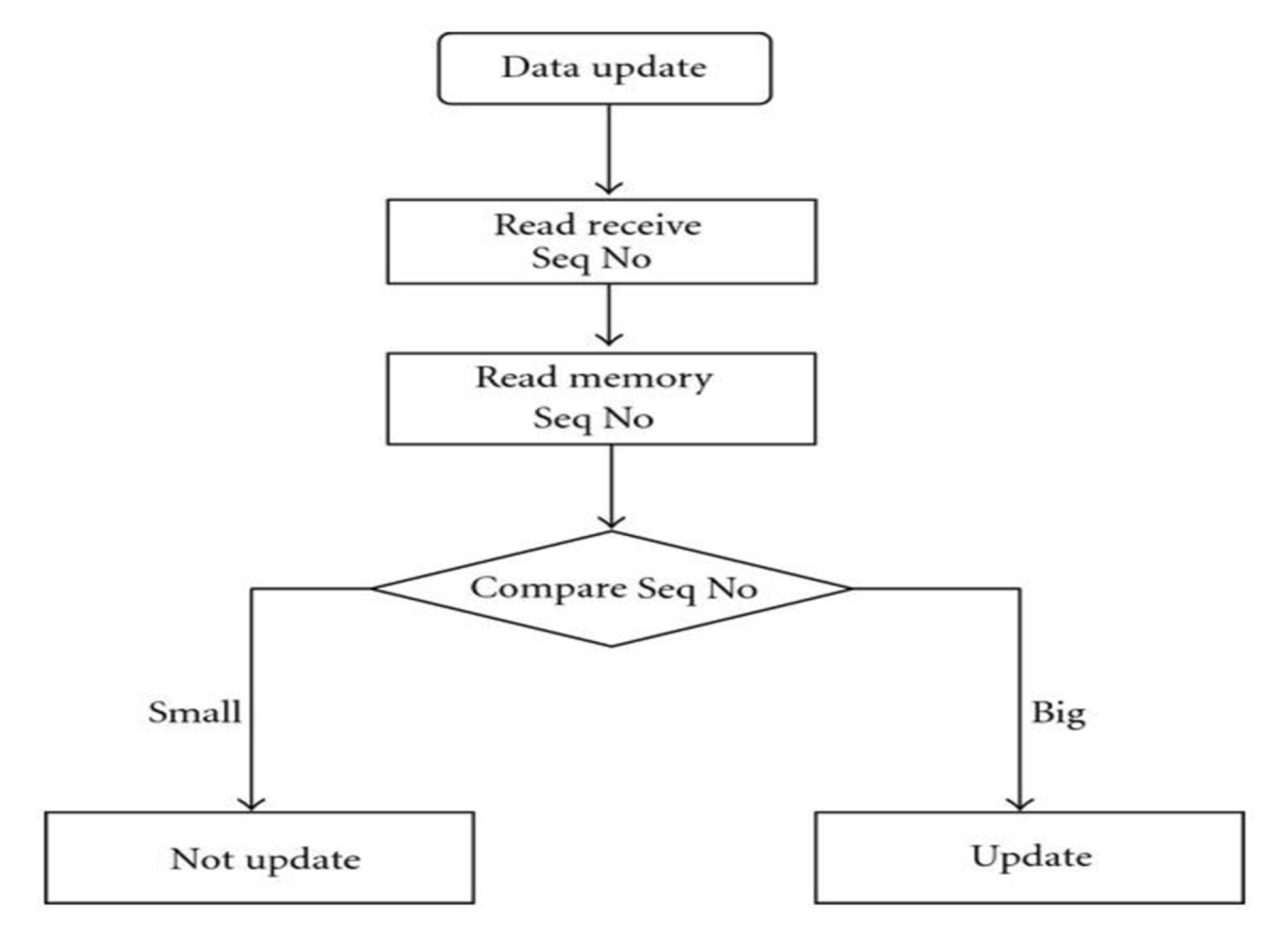


#### Fig4.2 Data Retrieval Process

The data retrieval process in the backend of the Application is a pivotal component that facilitates the efficient and secure retrieval of data from the database and its transmission to the frontend or client applications. This process begins with a client request made to a specific API endpoint on the backend. Upon receiving the request, the back ends routing system directs it to the appropriate endpoint handler based on the URL and HTTP method.

Before proceeding, the backend verifies user authentication and confirms if the necessary permissions exist to access the requested data. Following authorization, a database query is generated based on the request, specifying the criteria for data retrieval.

* 1. **DATA UPDATE PROCESS**



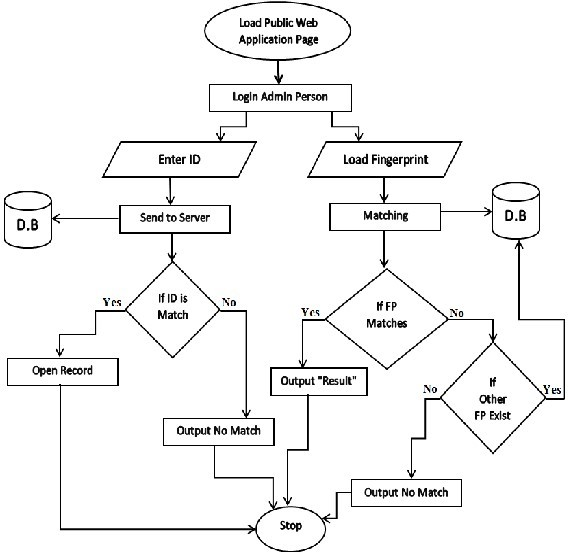
## **Fig4.3 Data Update Flowchart**

The data update process within the backend of Application is a crucial mechanism that allows users and administrators to enact changes in the application's database.

Authentication and authorization are fundamental checkpoints in this process. The backend confirms the user's identity and verifies if the user possesses the necessary permissions to execute the data update operation. Unauthorized requests are diligently restricted.

Data validation is a subsequent step, whereby the backend scrutinizes the provided data to ensure that it adheres to the correct format and complies with established business rules and constraints .This phase plays a pivotal role in maintaining data integrity.

### **SECURITY AND AUTHENTICATION**



#### Fig4.4 Security And Authentication Flowchart

Security and authentication lie at the heart of the application's robust infrastructure. To safeguard sensitive data and uphold the integrity of the system .

User authentication is a fundamental pillar, allowing users, including administrators and customers, to register securely, leveraging email and strong, hashed passwords. A robust login system verifies user credentials and controls access to the application.

Data encryption, both in transit and at rest, is a core component of the security framework. Secure communication channels protect data during interactions, while encryption of sensitive data in the database safeguards information in the event of a breach. Session management maintains secure user sessions, preventing unauthorized access or data exposure.

Utilizing security libraries and frameworks, like Spring Security, enhances the efficiency of authentication and access control. Additionally, protection against common security threats, such as cross-site scripting and SQL injection, is in place.

User awareness and education contribute to the overall security posture, ensuring that users are informed and capable of recognizing potential threats. Regular security audits and vulnerability assessments are conducted to proactively identify and address potential weaknesses, keeping the application resilient against emerging security risks. In sum, these measures collectively create a secure and trustworthy environment within the the application, protecting user data and maintaining the application's integrity.

#### Coding:

**Jwt Authentication Filter Class:**

#### package com.example.giftcraft.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.giftcraft.service.JwtService;

#### import jakarta.servlet.FilterChain;

#### import jakarta.servlet.ServletException;

#### import jakarta.servlet.http.HttpServletRequest;

#### import jakarta.servlet.http.HttpServletResponse;

#### @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(username);

#### if (jwtService.validateToken(token, userDetails)) {

#### UsernamePasswordAuthenticationToken authToken = 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.giftcraft.service;

import java.util.Date;

import java.util.HashMap;

import java.util.Map;

import java.util.function.Function;

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;

@Component

public class JwtService {

@Value("${application.jwt.secret-key}")

private String secretKey;

@Value("${application.jwt.token-expiration:1800000}") // Default token expiration: 30 minutes

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<>();

// You can add additional claims here if needed

return createToken(claims, username);

}

private String createToken(Map<String, Object> claims, String username) {

return Jwts.builder()

.setClaims(claims)

.setSubject(username)

.setIssuedAt(new Date())

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

.signWith(getSignKey(), SignatureAlgorithm.HS256)

.compact();

}

private Key getSignKey() {

byte[] keyBytes = Decoders.BASE64.decode(secretKey);

return Keys.hmacShaKeyFor(keyBytes);

}

}

**CONCLUSION:**

The development of the life insurance portal using React has been a transformative journey, showcasing the power and flexibility of modern web technologies. This project has demonstrated the ability to create a seamless, user-friendly interface that meets the needs of both clients and insurance providers. By leveraging React's component-based architecture, we've built a scalable and maintainable platform that can easily adapt to future enhancements and integrations.

Through this project, we've not only provided users with an intuitive experience for exploring and purchasing insurance products but also ensured that the backend services are robust and secure. The implementation of responsive design principles has made the portal accessible across various devices, enhancing the overall user experience.

As we conclude this phase, the life insurance portal stands as a testament to the effective use of cutting-edge technologies in addressing real-world needs. The lessons learned and the technical expertise gained during this project will undoubtedly be valuable for future endeavors. We look forward to continuously improving the portal, adding new features, and providing even greater value to our users.