



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



GYM TRAINER AND PROGRESS TRACKER APPLICATION

A PROJECT REPORT

Submitted by

GIRISHKANTH D : 727822TUCS037

in partial fulfilment for the award of the degree

Of

BACHELOR OF TECHNOLOGY

IN

Computer Science & Engineering

JULY - 2024

GYM TRAINER AND PROGRESS TRACKER APPLICATION

INTRODUCTION:

The **Gym Trainer and Progress Tracker Application** is designed to revolutionize personal fitness management by providing a comprehensive platform for users to track their workout routines, monitor progress, and receive personalized training guidance. This application is aimed at fitness enthusiasts, personal trainers, and gym-goers who seek an organized and efficient way to manage their fitness goals.

Key features include user-friendly interfaces for logging workouts, visual progress dashboards, and tailored exercise programs. The app also offers administrative tools for managing users and tracking overall usage statistics. Built with a robust backend and intuitive frontend, the Gym Trainer and Progress Tracker Application aims to enhance user experience and support effective fitness tracking and goal achievement.

This project leverages modern web technologies and secure authentication methods to ensure data integrity and user privacy, making it a reliable and essential tool for anyone committed to improving their physical fitness.

OBJECTIVE:

The backend of the **Gym Trainer and Progress Tracker Application** is designed to achieve several key objectives. At its core, it provides secure user authentication and authorization using JWT for token-based access, ensuring robust role management for both admin and user functionalities. The backend focuses on efficient data management by storing and handling user profiles, workout plans, and progress logs, with comprehensive CRUD operations for workouts.

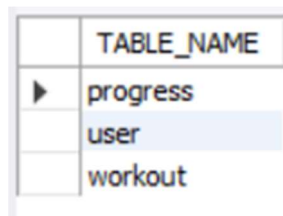
One of the primary objectives is to facilitate seamless integration with the frontend through RESTful APIs, ensuring data consistency and integrity. This allows users to interact with their workout plans and track their progress effortlessly. Furthermore, the backend emphasizes security by protecting sensitive user data through encryption and secure handling practices, thereby ensuring user privacy and data integrity.

Additionally, the backend aims to support scalable and maintainable code architecture, allowing for future enhancements and modifications. By providing a reliable and secure backend, the application ensures a smooth user experience, enabling users to focus on their fitness goals without concerns about data security or application performance.

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.

1.1 POSTGRESQL



	TABLE_NAME
▶	progress
	user
	workout

Fig1.1 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 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 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.

1.2 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.

1.3 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.

SYSTEM ARCHITECTURE

The "GYM TRAINER AND PROGRESS TRACKER 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.

2.1 BACKEND



Fig2.1 Backend System Architecture

The backend layer of the "GYM TRAINER AND PROGRESS TRACKER APPLICATION" 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 gym trainer and progress tracking. The backend handles various functionalities such as managing workout plans, tracking progress metrics, scheduling training sessions, and providing analytics and reports.

Furthermore, the backend is equipped to handle user management and authentication, ensuring a secure and personalized user experience for both trainers and clients. This includes features like user registration, login, role-based access control, and session management. 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 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.

IMPLEMENTATION AND FUNCTIONALITY

The GYM TRAINER AND PROGRESS TRACKER APPLICATION application's backend is crucial for effective fitness tracking and user management. It offers a robust set of tools that enable administrators to efficiently manage user accounts, workouts, and progress data. The backend's intuitive design ensures that administrators can add, update, and remove user profiles and workout information with ease, minimizing the learning curve.

3.1 API Request

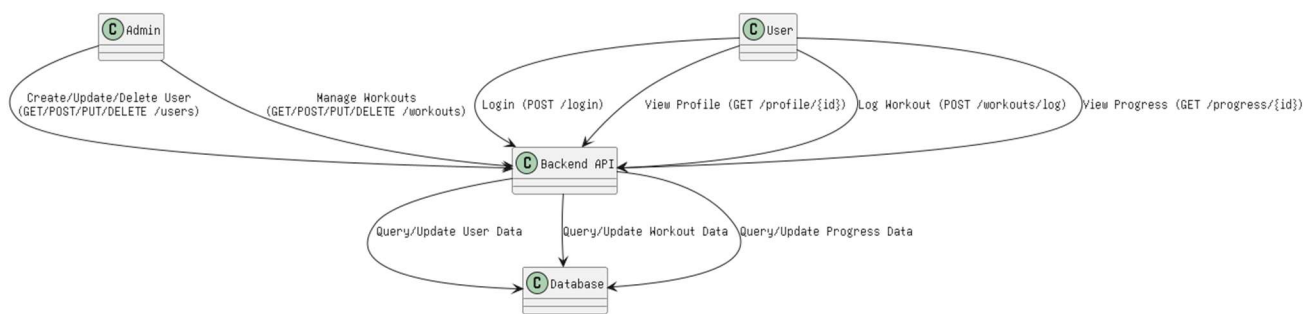


Fig3.1 REST API flow chart

In the FitClub Tracker Analyzer application, API requests facilitate interaction between the frontend interface and the backend services, ensuring seamless management of user and workout data.

When administrators perform user management tasks, they utilize specific RESTful API endpoints to handle operations. For example, creating a new user involves sending a POST request to /users with the user's details. To update existing user information, an administrator would send a PUT request to /users/{id}, where {id} specifies the user to be updated. Deleting a user is accomplished via a DELETE request to /users/{id}. These operations are crucial for maintaining accurate and up-to-date user records in the system.

For authentication, users send login credentials in a POST request to /login. This request generates a JSON Web Token (JWT) upon successful authentication, which is used for secure, token-based access to protected resources throughout the application.

Workout management is similarly handled through the API. Administrators can view, add, modify, or remove workout exercises using GET, POST, PUT, and DELETE requests to the /workouts endpoint. This allows for comprehensive control over the workout data available to users. Users themselves can log their workouts by sending a POST request to /workouts/log, which records their exercise activities.

Progress tracking for users is accessed through GET requests to endpoints like /progress/{id}, where {id} corresponds to the user's unique identifier. This functionality allows users to view their exercise history and track their progress over time.

Each API request triggers the backend to process the data and interact with the database to perform the necessary operations. For instance, a request to view a user's profile will prompt the backend to query the database for that user's information and return it to the frontend. Similarly, when data needs to be updated or deleted, the backend modifies the database accordingly.

Security is paramount, with JWT authentication ensuring that only authorized users and administrators can perform specific actions. The API's design not only ensures efficient data management but also protects sensitive information, maintaining the integrity and security of the application.

3.2 CRUD OPERATION

In the FitClub Tracker Analyzer application, CRUD operations are essential for managing user and workout data through the API. For Create operations, administrators can add new users by sending a POST request to /users, providing necessary details like

name and email. Similarly, new workouts are added via a POST request to /workouts, including information such as workout name and description. Users can log their workouts through a POST request to /workouts/log, which records their exercise activities.

Read operations allow for data retrieval. Administrators can fetch specific user details by sending a GET request to /users/{id}, while workout information is accessible through a GET request to /workouts. Users can review their progress by sending a GET request to /progress/{id}, where {id} represents their unique identifier.

For Update operations, administrators can modify existing user profiles by sending a PUT request to /users/{id} with updated information. Workouts can be updated similarly by sending a PUT request to /workouts/{id}, adjusting details as needed. Users can also update their workout logs through a PUT request to /workouts/log/{id}.

Delete operations enable data removal. Administrators can delete a user account with a DELETE request to /users/{id}, while specific workouts can be removed via a DELETE request to /workouts/{id}. To remove workout log entries, a DELETE request to /workouts/log/{id} is used. Each CRUD operation involves the backend processing the request, interacting with the database, and providing a response to ensure accurate and secure data management.

Coding:

Entity Class:

```
package com.sp.backend.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;

import com.sp.backend.model.User;
import com.sp.backend.service.UserService;

import java.util.Optional;

@RestController
@RequestMapping("/api/users")
```

```

public class UserController {
    @Autowired
    private UserService userService;

    @PostMapping("/signup")
    public ResponseEntity<?> signUp(@RequestBody User user) {
        if (userService.existsByEmail(user.getEmail())) {
            return ResponseEntity.badRequest().body("Account already exists");
        }
        User savedUser = userService.saveUser(user);
        return ResponseEntity.ok(savedUser);
    }

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody User user) {
        Optional<User> foundUser = userService.findByEmail(user.getEmail());
        if (foundUser.isPresent() &&
            foundUser.get().getPassword().equals(user.getPassword())) {
            return ResponseEntity.ok(foundUser.get());
        }
        return ResponseEntity.badRequest().body("Invalid credentials");
    }

    @PutMapping("/update")
    public ResponseEntity<?> updateUser(@RequestBody User user) {
        Optional<User> foundUser = userService.findByEmail(user.getEmail());
        if (foundUser.isPresent()) {
            User existingUser = foundUser.get();
            existingUser.setName(user.getName());
            existingUser.setPhoneNumber(user.getPhoneNumber());
            existingUser.setAge(user.getAge());
            existingUser.setHeight(user.getHeight());
            existingUser.setWeight(user.getWeight());
            existingUser.setBloodGroup(user.getBloodGroup());
            existingUser.setMedicalCondition(user.getMedicalCondition());
            existingUser.setDateOfJoining(user.getDateOfJoining());
            existingUser.setCity(user.getCity());
            existingUser.setCountry(user.getCountry());
            existingUser.setGoal(user.getGoal());
            existingUser.setRole(user.getRole());
            userService.saveUser(existingUser);
            return ResponseEntity.ok(existingUser);
        }
    }
}

```

```

        return ResponseEntity.badRequest().body("User not found");
    }

    @GetMapping("/{email}")
    public ResponseEntity<?> getUserByEmail(@PathVariable String email) {
        Optional<User> foundUser = userService.findByEmail(email);
        if (foundUser.isPresent()) {
            return ResponseEntity.ok(foundUser.get());
        }
        return ResponseEntity.notFound().build();
    }

    @GetMapping("/all")
    public ResponseEntity<?> getAllUsers() {
        return ResponseEntity.ok(userService.findAllUsers());
    }

    @DeleteMapping("/delete/{id}")
    public ResponseEntity<?> deleteUser(@PathVariable Long id) {
        userService.deleteUserById(id);
        return ResponseEntity.ok().build();
    }
}

```

Service Class:

```

package com.sp.backend.service;

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

import com.sp.backend.model.User;
import com.sp.backend.repository.UserRepository;

import java.util.List;
import java.util.Optional;

@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;
}

```

```

    public Optional<User> findByEmail(String email) {
        return userRepository.findByEmail(email);
    }

    public User saveUser(User user) {
        return userRepository.save(user);
    }

    public boolean existsByEmail(String email) {
        return userRepository.findByEmail(email).isPresent();
    }

    public List<User> findAllUsers() {
        return userRepository.findAll();
    }

    public void deleteUserById(Long id) {
        userRepository.deleteById(id);
    }

    public Optional<User> findById(Long id) {
        return userRepository.findById(id);
    }
}

```

Repository Interface:

```

package com.sp.backend.repository;

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

import com.sp.backend.model.User;

import java.util.Optional;

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
}

```


3.3 DATA RETRIEVAL PROCESS

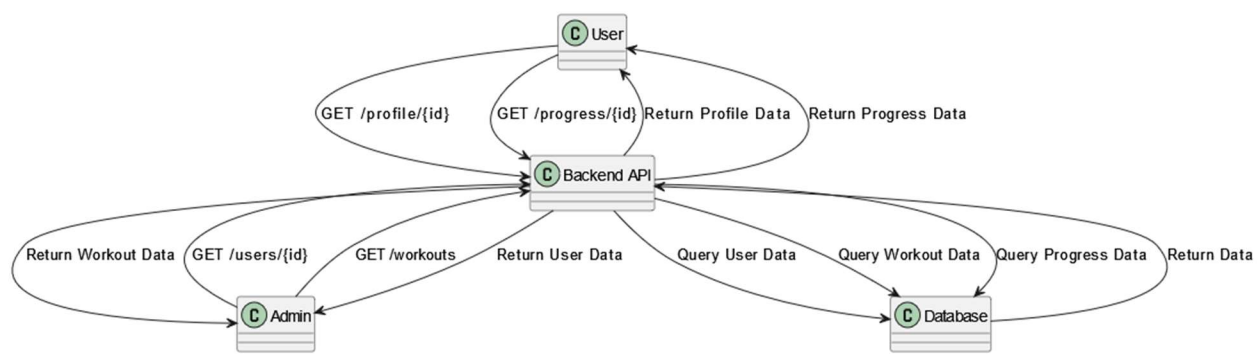


Fig3.2 Data Retrieval Process

In the **FitClub Tracker Analyzer** application, data retrieval begins when a **User** or **Admin** initiates a request to the **Backend API**. For example, a user might request their profile information or progress data, while an admin might request details about other users or workout exercises. Upon receiving the request, the **Backend API** queries the **Database** for the relevant information. The database processes this query and returns the requested data to the API. Finally, the **Backend API** sends the retrieved data back to the requester—whether that be the user or the admin—ensuring they receive the accurate and up-to-date information they need. This flow ensures that data retrieval is efficient and secure, maintaining the integrity of the application's data management.

3.4 DATA UPDATE PROCESS

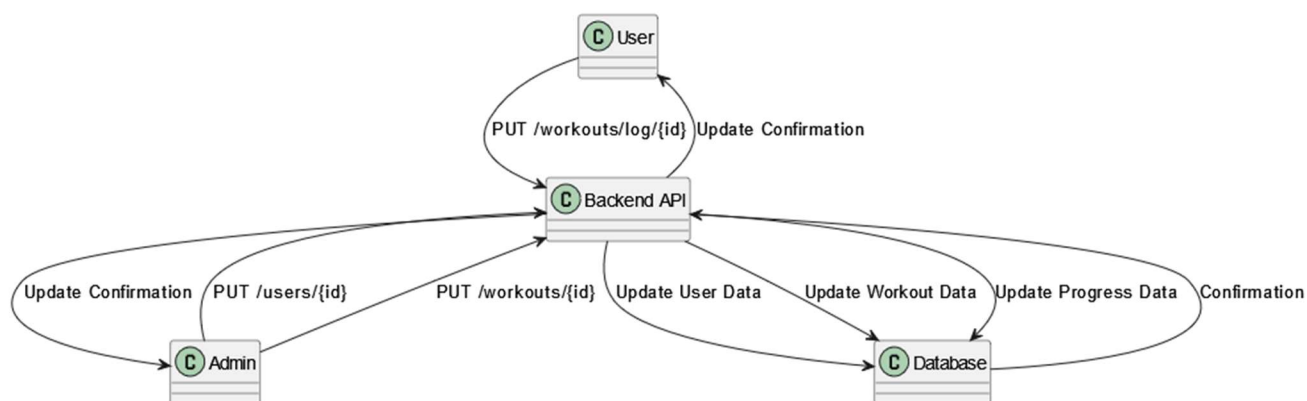


Fig3.3 Data Update Flowchart

In the FitClub Tracker Analyzer application, the data update process begins when an Admin or User initiates an update request through the Backend API. For example, an admin might update user profiles or workout details, while a user might update their workout logs.

When an update request is made, such as a PUT request to `/users/{id}` or `/workouts/{id}`, the Backend API receives the request and processes the update instructions. The API then communicates with the Database to modify the relevant data based on the request parameters. Once the database performs the update, it sends a confirmation back to the API. The Backend API then responds to the requester with the outcome of the update operation, such as a success message or the updated data itself.

This process ensures that data updates are accurately applied and reflected in the system, maintaining data consistency and integrity.

3.5 SECURITY AND AUTHENTICATION

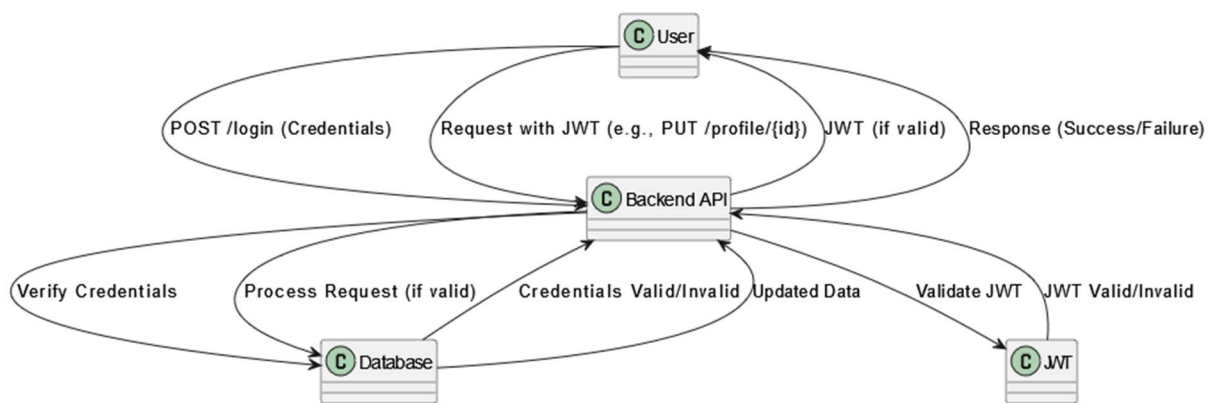


Fig3.4 Security And Authentication Flowchart

In the FitClub Tracker Analyzer application, the security and authentication process is crucial for protecting user data and ensuring that only authorized individuals can access or modify sensitive information.

Process Description:

1. User Authentication:

- When a user attempts to log in, they send their credentials (username and password) via a POST request to the /login endpoint of the **Backend API**.
- The **Backend API** verifies the credentials against stored records in the **Database**. If the credentials are valid, the API generates a JSON Web Token (JWT) and returns it to the user. This token is used for authentication in subsequent requests.

2. Token-Based Authentication:

- For accessing protected resources, users include their JWT in the HTTP Authorization header of their requests. For example, to update their profile or log a workout, they would send requests with the JWT token to the respective API endpoints.
- The **Backend API** validates the JWT to ensure it is authentic and has not expired. If the token is valid, the API processes the request and performs the necessary actions, such as updating user data or logging workouts.

3. Access Control:

- The API checks the user's role and permissions to ensure they are authorized to perform specific actions. For instance, only admins can add or delete users, while regular users can only update their own profiles.
- The API responds with appropriate messages based on the outcome of the authentication and authorization checks. If the token is invalid or the user lacks permission, the API returns an error message.

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:

In the **FitClub Tracker Analyzer** application, the integration of robust backend functionality, comprehensive CRUD operations, and stringent security measures ensures a reliable and secure platform for managing fitness data. The backend efficiently handles user and workout data through a structured set of CRUD operations, allowing for the seamless creation, retrieval, updating, and deletion of records. This functionality supports administrators in maintaining accurate user profiles and workout details, while also enabling users to log their activities and track their progress.

Security and authentication are paramount, with the system employing JWT (JSON Web Token) for secure, token-based access. This approach safeguards sensitive information by ensuring that only authenticated and authorized users can access or modify data. The use of JWT enhances the security of the application, protecting user data and preventing unauthorized access.

Overall, the FitClub Tracker Analyzer application delivers a well-rounded solution for fitness management, combining efficient data handling with robust security features. This design ensures that users and administrators can interact with the system confidently, knowing their data is managed securely and efficiently.