Task Management System Using Microservices Architecture

Introduction:

The **Task Management System** is a backend application developed using a **Microservices Architecture** to manage tasks, user authentication, and task submissions in a scalable and modular manner.

The system allows users to register, log in securely, create tasks with deadlines, update task status, and submit task work. The project is designed following modern distributed system principles using Spring Boot and Spring Cloud technologies.

This application demonstrates real-world backend engineering practices such as service discovery, API gateway routing, inter-service communication, and secure authentication using JWT.

Purpose of the Project:

The primary objective of this project was to:

- Gain practical experience in designing and implementing **distributed systems**
- Understand **Microservices Architecture** in depth
- Learn service-to-service communication using **OpenFeign**
- Implement centralized routing using **Spring Cloud Gateway**

- Use **Eureka Server** for service discovery
- Build secure REST APIs using **JWT-based authentication**

This project was built not only as an academic exercise but as a **production-level backend system simulation** to prepare for real-world software engineering roles.

Problem Statement:

In traditional monolithic applications, all functionalities such as user management, task handling, and submissions are tightly coupled within a single codebase. This leads to several challenges:

- Difficult scalability
- Tight coupling between modules
- Harder debugging and maintenance
- Single point of failure
- Slow deployment cycles

The Task Management System solves these issues by separating responsibilities into independent services:

- User Service
- Task Service
- Submission Service

Each service operates independently while communicating through well-defined APIs

Why Microservices Architecture?

Microservices architecture was chosen for the following reasons:

1. Independent Deployment

Each service can be developed, tested, and deployed independently without affecting others.

2. Scalability

Services can be scaled individually based on load requirements (e.g., Task Service may require more instances than User Service).

3. Fault Isolation

Failure in one service does not crash the entire system.

4. Technology Flexibility

Each microservice can evolve independently and can potentially use different technologies if required.

5. Real-World Industry Relevance

Most modern large-scale applications (Netflix, Amazon, Uber) use microservices architecture. Building this project helped simulate real enterprise backend architecture.

Key Features of the System

- Secure user registration and login
- JWT-based authentication and authorization
- Task creation and management with status tracking
- Deadline assignment to tasks
- Task submission functionality

- Centralized API Gateway routing
- Service discovery using Eureka
- Inter-service communication using OpenFeign

System Architecture

Architecture Overview

The Task Management System is designed using a **Microservices Architecture Pattern**, where each core functionality is implemented as an independent service.

The system consists of five major components:

- Eureka Server (Service Registry)
- API Gateway
- User Service
- Task Service
- Submission Service

Each service runs independently on a separate port and communicates through REST APIs. Service discovery and routing are managed using Spring Cloud components.

Microservices Components

Eureka Server (Service Registry)

Port: 8070

Eureka Server acts as the **Service Discovery Server**.

Responsibilities:

- Registers all running microservices
- Maintains registry of service instances
- Enables dynamic service lookup
- Supports load-balanced communication via service names

All services (Gateway, User, Task, Submission) register themselves with Eureka at startup.

Without Eureka, services would need hardcoded URLs, making the system tightly coupled.

API Gateway

Port: 5000

The Gateway Service acts as the **single entry point** for all client requests.

Responsibilities:

- Routes incoming HTTP requests to appropriate services
- Performs load balancing using Eureka
- Handles global CORS configuration
- Acts as reverse proxy
- Centralizes request filtering and response handling

The client never directly communicates with internal services. All requests pass through the Gateway.

User Service

Port: 5001

Handles user-related functionalities.

Responsibilities:

- User registration
- User login (authentication)
- JWT token generation
- Fetch authenticated user profile
- Role-based validation (if implemented)

This service is responsible for security and identity management.

Task Service

Port: 5002

Handles task management operations.

Responsibilities:

- Create new tasks
- Assign deadlines
- Update task status
- Fetch task details
- Mark tasks as completed
- Validate user authorization via token

The Task Service may communicate with User Service for user validation.

Submission Service

Port: 5003

Handles task submission logic.

Responsibilities:

- Submit task solutions (GitHub link)
- Store submission time
- Fetch submissions by ID
- Fetch submissions by Task ID
- Accept or Decline submissions
- Notify Task Service to mark task as completed (via Feign Client)

This service communicates with:

- User Service → to validate user identity
- Task Service → to validate and update task status

Communication Pattern

The system uses:

- REST APIs for communication
- OpenFeign for inter-service communication
- Eureka for service discovery
- Load Balancer (lb://SERVICE-NAME) for dynamic routing

All inter-service calls are resolved using service names registered in Eureka instead of fixed URLs.

Request Flow Explanation

Example: User Login Flow

1. Client sends login request to:

   `http://localhost:5000/auth/signin`

2. Gateway Service receives the request.
3. Gateway checks route configuration and forwards request to:

   `USER-SERVICE (Port 5001)`

4. User Service:
   - Validates credentials
   - Generates JWT token
   - Returns authentication response
5. Gateway forwards response back to client.

Example: Task Submission Flow

1. Client sends submission request to:

   `http://localhost:5000/api/submission/submit`

2. Gateway routes request to:

   `SUBMISSION-SERVICE (Port 5003)`

3. Submission Service:
   - Validates JWT using User Service
   - Calls Task Service to verify task existence
   - Stores submission in database

- Saves submission time
- Returns response

4. Response flows back:

   `Submission Service → Gateway → Client`

Database Interaction

Each microservice maintains its own database layer.

- User Service → User table
- Task Service → Task table
- Submission Service → Submission table

Data is not shared directly between services.
All cross-service communication happens through REST APIs.

This ensures:

- Loose coupling
- Data ownership
- Service independence

Architectural Benefits Achieved

- Clear separation of concerns
- Independent scalability
- Better maintainability

- Centralized routing
- Dynamic service discovery
- Secure communication using JWT
- Production-level microservices simulation

Database Design

Database Architecture Strategy

The Task Management System follows the **Database per Service Pattern**, which is a core principle of microservices architecture.

Each service maintains its own independent database schema:

- User Service → User Database
- Task Service → Task Database
- Submission Service → Submission Database

No foreign key constraints are defined between services to maintain:

- Loose coupling
- Service independence
- Independent deployment
- Scalability

However, logical relationships are maintained using identifiers (IDs) passed between services.

User Table Design

Service: User Service

Purpose: Stores authentication and user identity details

Table: users

Column Name	Data Type	Description
id	Long (Primary Key)	Unique user identifier
full_name	String	User's full name
email	String (Unique)	User login email
password	String	Encrypted password
role	String	USER / ADMIN
created_at	Timestamp	Account creation time

Key Points:

- `id` is auto-generated.
- `email` is unique.
- Passwords are stored in encrypted format (BCrypt).
- No foreign keys to other services.

Task Table Design

Service: Task Service

Purpose: Stores task details created in the system

Table: tasks

Column Name	Data Type	Description
id	Long (Primary Key)	Unique task identifier
title	String	Task title
description	String	Task details
deadline	LocalDateTime	Task deadline
status	String	PENDING / COMPLETED
created_by	Long	User ID who created task
created_at	Timestamp	Task creation time

Key Points:

- `created_by` stores the User ID.
- No foreign key constraint to User table.
- User validation happens through REST call to User Service.
- Status is updated when submission is accepted.

Submission Table Design

Service: Submission Service
Purpose: Stores user task submissions

Table: submissions

Column Name	Data Type	Description

Column Name	Data Type	Description
id	Long (Primary Key)	Unique submission ID
task_id	Long	Associated task ID
user_id	Long	User who submitted
github_link	String	Submission repository link
submission_time	LocalDateTime	Time of submission
status	String	PENDING / ACCEPTED / REJECTED

Key Points:

- `task_id` references a Task logically.
- `user_id` references a User logically.
- No foreign key constraints are used.
- Validation happens via:
  - Task Service call (to check task exists)
  - User Service call (to validate user)

Logical Relationships (Application-Level Relationships)

Although there are no database-level foreign keys, the system maintains logical relationships:

Relationship 1:

One User → Many Tasks
(through created_by field)

Relationship 2:

One User → Many Submissions
(through user_id field)

Relationship 3:

One Task → Many Submissions
(through task_id field)

These relationships are enforced through:

- Service-to-service REST calls
- Business validation logic
- Feign client communication

API Documentation

Authentication APIs (User Service)

Base Path: `/auth`

These APIs handle user registration and login using JWT authentication.

□ POST `/auth/signup`

Registers a new user in the system.

Request Body

```
{
  "fullName": "Ramyasri",
  "email": "ramya@gmail.com",
  "password": "123456",
```

```
  "role": "USER"
}
```

Response

```
{
  "jwt":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "message": "Registration success",
  "status": true
}
```

Description

- Checks if email already exists.
- Encrypts password using BCrypt.
- Generates JWT token after successful registration.

□ POST `/auth/signin`

Authenticates an existing user.

Request Body

```
{
  "email": "ramya@gmail.com",
  "password": "123456"
}
```

Response

```
{
  "jwt":
"eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
  "message": "login success",
  "status": true
```

```
}
```

Description

- Validates credentials.
- Generates JWT token.
- Token must be used in Authorization header for protected APIs.

User APIs (User Service)

Base Path: `/api/users`

Requires: `Authorization: Bearer <token>`

□ GET `/api/users/profile`

Returns currently authenticated user details.

Header

```
Authorization: Bearer <JWT_TOKEN>
```

Response

```
{
  "id": 1,
  "fullName": "Ramyasri",
  "email": "ramya@gmail.com",
  "role": "USER"
}
```

□ GET `/api/users/allUsers`

Returns list of all users (Admin access).

Task APIs (Task Service)

Base Path: `/api/tasks`

Requires: `Authorization` header

□ POST `/api/tasks/save`

Creates a new task.

Request

```
{
  "title": "Build REST API",
  "description": "Develop task management
APIs",
  "deadline": "2026-03-01T23:59:00",
  "status": "PENDING"
}
```

Header

```
Authorization: Bearer <JWT_TOKEN>
```

Response

```
{
  "id": 1,
  "title": "Build REST API",
  "description": "Develop task management
APIs",
  "deadline": "2026-03-01T23:59:00",
  "status": "PENDING"
```

```
}
```

□ GET /api/tasks/{id}

Fetch task by ID.

Example:

```
GET /api/tasks/1
```

□ GET /api/tasks

Fetch all tasks.

Optional Query Parameter:

```
/api/tasks?status=PENDING
```

□ PUT /api/tasks/{taskid}/user/{userid}

Assign task to a user.

Example:

```
PUT /api/tasks/1/user/2
```

□ GET /api/tasks/user

Get tasks assigned to logged-in user.

Optional filter:

```
/api/tasks/user?status=PENDING
```

□ PUT `/api/tasks/{id}`

Update task details.

□ DELETE `/api/tasks/{id}`

Delete a task.

□ PUT `/api/tasks/{id}/complete`

Mark task as completed.

Submission APIs (Submission Service)

Base Path: `/api/submission`

Requires: `Authorization` header

□ POST `/api/submission/submit`

Submit task solution.

Request Parameters

```
taskid=1
githubLink=https://github.com/ramya/task-solution
```

Example:

```
POST /api/submission/submit?taskid=1&githubLink=https://github.com/ramya/task-solution
```

Response

```
{
  "id": 1,
  "taskId": 1,
  "userId": 2,
  "githubLink":
"https://github.com/ramya/task-solution",
  "status": "PENDING"
}
```

□ GET `/api/submission/{id}`

Get submission by ID.

□ GET `/api/submission`

Get all submissions.

□ GET `/api/submission/task/{taskid}`

Get all submissions for a specific task.

Example:

```
GET /api/submission/task/1
```

□ PUT `/api/submission/{id}`

Accept or decline submission.

Request Parameter

```
status=ACCEPTED
```

Example:

```
PUT /api/submission/1?status=ACCEPTED
```

Response

```
{
  "id": 1,
  "taskId": 1,
  "userId": 2,
  "status": "ACCEPTED"
}
```

Security Mechanism

- All protected APIs require:

```
Authorization: Bearer <JWT_TOKEN>
```

- Token is validated in each service.
- Gateway routes request to respective microservice.
- Services communicate using Feign Clients.

Service Communication

- Task Service → calls User Service to validate role
- Submission Service → calls:
  - User Service (to validate user)
  - Task Service (to validate task)

This ensures:

- Data consistency

- Security enforcement
- Role-based access control

Security Implementation

Security is one of the core components of the Task Management Microservices system. The application follows a **stateless authentication and authorization mechanism** using **JWT (JSON Web Token)** integrated with **Spring Security**.

The system ensures that:

- Only authenticated users can access protected APIs
- Role-based access control is enforced
- Passwords are securely encrypted
- Inter-service communication is secured using token forwarding

---

Authentication Mechanism – JWT Based Security

The system uses **JWT (JSON Web Token)** for authentication.

 What is JWT?

JWT is a compact, URL-safe token format used for securely transmitting information between parties. It contains:

- **Header** – Algorithm and token type
- **Payload** – User details (email, role, etc.)
- **Signature** – Used to verify token authenticity

Once a user logs in successfully, a JWT token is generated and returned to the client.

□ Authentication Flow

1. User sends login request to:

```
POST /auth/signin
```

2. Credentials are validated using `CustomerUserServiceImpl`
3. Password is verified using `BCryptPasswordEncoder`
4. If valid → JWT token is generated using `JwtProvider`
5. Token is returned in response
6. Client stores token and sends it in every request

Authorization Using Authorization Header

After login, the client must include the token in every protected API request.

□ Authorization Header Format

```
Authorization: Bearer <JWT_TOKEN>
```

Example:

```
Authorization: Bearer
eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...
```

Spring Security validates this token using:

```
JwtTokenValidator (Custom Filter)
```

This filter is added before:

```
BasicAuthenticationFilter
```

Configured in:

`ApplicationConfig.java`

Stateless Session Management

The system is configured as **STATELESS**:

```
.sessionManagement(
    management ->
management.sessionCreationPolicy(
        SessionCreationPolicy.STATELESS))
```

 Why Stateless?

- No session stored on server
- Each request must contain JWT
- Improves scalability in microservices
- Suitable for distributed systems

This is ideal for microservices architecture.

Securing API Endpoints

In `ApplicationConfig`:

```
.requestMatchers("/api/**").authenticated()
.anyRequest().permitAll()
```

 Meaning:

- All endpoints starting with `/api/**` require authentication
- Public endpoints like `/auth/signup` and `/auth/signin` are accessible without token

This ensures secure access control.

Password Encryption

Passwords are never stored in plain text.

The system uses:

`BCryptPasswordEncoder`

Benefits:

- Strong hashing algorithm
- Automatically salts passwords
- Resistant to brute-force attacks

During signup:

```
passwordEncoder.encode(user.getPassword())
```

During login:

```
passwordEncoder.matches(password,
userDetails.getPassword())
```

This ensures secure password verification.

Custom JWT Filter – JwtTokenValidator

A custom filter is added:

```
.addFilterBefore(new JwtTokenValidator(),
BasicAuthenticationFilter.class)
```

 Responsibilities of this filter:

- Extract JWT from Authorization header

- Validate token signature
- Extract user details
- Set authentication in SecurityContext

If token is invalid → request is rejected.

CORS Configuration

To allow frontend communication (React running on different port), CORS is configured:

```
cfg.setAllowedOrigins(Collections.singletonList("*"));
cfg.setAllowedHeaders(Collections.singletonList("*"));
cfg.setAllowedMethods(Collections.singletonList("*"));
```

Why CORS is required?

- Frontend runs on different origin (e.g., port 5173)
- Backend runs on different port (e.g., 5000+)
- Browser blocks cross-origin requests by default

CORS configuration enables secure cross-origin communication.

Why Feign Client Needs Authorization Header

In microservices architecture:

- Task Service calls User Service
- Submission Service calls Task Service
- Services communicate using Feign Clients

Example:

```
UserDto user =
userClient.getUserProfile(authHeader);
```

 Why pass Authorization header?

Because:

- Each microservice is independently secured
- Every `/api/**` endpoint requires authentication
- Without token → request will fail with 401 Unauthorized

So the original JWT token received from client must be forwarded to other services.

This ensures:

✓ Secure inter-service communication
✓ Consistent authentication validation
✓ Zero trust between services

Security Architecture Flow

```
Client (React)
     |
     | 1. Login
     v
User Service → Generates JWT
     |
     | 2. Send JWT in Authorization header
     v
Gateway
     |
     v
Task Service / Submission Service
```

```
     |
     | (Feign Client forwards same JWT)
     v
User Service validates token again
```

Security Best Practices Implemented

✓ Stateless authentication
✓ JWT-based authorization
✓ BCrypt password hashing
✓ Custom authentication filter
✓ Secure inter-service communication
✓ CORS configuration
✓ Role-based logic inside services
✓ No session storage on server

Challenges Faced & Solutions Implemented

During the development of the Task Management Microservices System, several real-world technical challenges were encountered. These challenges significantly enhanced my understanding of distributed systems, service discovery, security, and inter-service communication.

CORS Policy Errors

 Problem:

While integrating the React frontend with the Spring Boot Gateway, the browser blocked API calls with the error:

> “No 'Access-Control-Allow-Origin' header is present on the requested resource.”

This occurred because the frontend (Port 5173) and backend (Port 5000) were running on different origins.

✅Solution:

Implemented **Global CORS Configuration** in the Gateway:

- Allowed all origins
- Allowed all headers
- Allowed all HTTP methods
- Exposed Authorization header
- Enabled credentials

Configured using:

- `spring.cloud.gateway.globalcors`
- Custom `CorsConfigurationSource` in Spring Security

This resolved cross-origin communication between frontend and backend services.

– No Instance Available Error

□ Problem:

Gateway returned:

```
503 Unable to find instance for task-user-service
```

Even though the service was running.

 Root Cause:

Mismatch between:

- `spring.application.name`
- Gateway route `uri: lb://SERVICE-NAME`
- Eureka registered service name

Spring Cloud is **case-sensitive and name-sensitive**.

✅Solution:

Ensured:

```
spring:
  application:
    name: task-user-service
```

And in Gateway:

```
uri: lb://task-user-service
```

After correcting service name consistency, Gateway successfully routed requests.

Service Name Mismatch in Eureka

 Problem:

Services were showing as:

- `TASK USER SERVICE`
- `TASK-SERVICE`
- `task-user-service`

This caused routing failures.

✅Solution:

Standardized naming convention across all services:

- Used lowercase with hyphen
- Removed spaces
- Ensured consistency in:
  - application.yml
  - Gateway routes
  - Feign clients

This eliminated routing inconsistencies.

Enum Conversion Error

□ Problem:

When passing `TaskStatus` as request parameter:

```
?status=completed
```

Application threw conversion exception.

□ Cause:

Enum values must match exactly (case-sensitive).

✅Solution:

Defined enum clearly:

```
public enum TaskStatus {
    PENDING,
    IN_PROGRESS,
    COMPLETED
}
```

Passed correct uppercase values:

`?status=COMPLETED`

Ensured validation at controller level.

Duplicate Query Results

 Problem:

Some queries returned unexpected duplicate results.

 Cause:

Improper filtering logic in service layer.

✅**Solution:**

Refined repository queries and added:

- Proper filtering conditions
- Null checks
- Optional handling

Improved data consistency.

Feign Client Authorization Issue

 Problem:

Inter-service communication failed with 401 Unauthorized.

 Cause:

JWT token was not forwarded when Task or Submission service called User service.

✅Solution:

Passed `Authorization` header manually in Feign calls:

```
UserDto user =
userClient.getUserProfile(authHeader);
```

Ensured token propagation between microservices.

This maintained secure communication.

Internal Server Error (500)

□ Problem:

Gateway returned 500 error during `/auth/signin`.

□ Cause:

Exception handling was not properly structured in AuthController.

✅Solution:

Improved authentication logic:

- Added password match validation
- Handled BadCredentialsException
- Verified JWT generation logic

This stabilized authentication flow.

Technologies Used

The project was built using modern enterprise-grade technologies aligned with industry standards.

Backend Technologies

Technology	Purpose
Java 17	Core programming language
Spring Boot 3	Backend application framework
Spring Security	Authentication & authorization
Spring Cloud	Microservices architecture support
Eureka Server	Service discovery
Spring Cloud Gateway	API Gateway & routing
OpenFeign	Inter-service communication
JWT (JSON Web Token)	Stateless authentication
Hibernate / JPA	ORM for database operations

Database

Technology	Purpose
MySQL	Relational database
Separate DB per Service	Microservices isolation principle

Tools & DevOps

Tool	Usage
Maven	Dependency

Tool	Usage
	management
Postman	API testing
Git & GitHub	Version control
Eureka	Development IDE

Architecture Pattern

- Microservices Architecture
- Database Per Service Pattern
- API Gateway Pattern
- Stateless JWT Authentication
- Service Discovery Pattern