FULL STACK DEV

Writing business & functional requirements and project management

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

FULL ENTERPRISE WORKFLOW (1)

- **Confluence**
  - Purpose: Document all business requirements, functional specs, wireframes, and meeting notes.
  - Actors: Team Leader / Product Owner
  - Output: Clear reference for developers, QA, and stakeholders.

- **Jira**
  - Purpose: Track project work using Epics → User Stories → Subtasks.
  - Actors: Team Leader creates Epics/User Stories; Developers are assigned subtasks.
  - Output: Organized, trackable tasks mapped to code/features.

FULL ENTERPRISE WORKFLOW (2)

- **Cucumber / Gherkin**
  - Purpose: Define BDD test scenarios based on user stories.
  - Actors: QA or Devs write feature files per Epic.
  - Output: Testable scenarios (acceptance criteria) for automated testing.

- **Development**
  - Purpose: Implement feature functionality.
  - Actors: Developers work on backend, frontend, or full-stack.
  - Output: Functional code per Jira tasks.

FULL ENTERPRISE WORKFLOW (4)

- **Unit Testing**
  - Junit: For testing Java code (backend, services, controllers)
  - Mockito: For mocking dependencies in unit tests
  - Purpose: Ensure code correctness and independent testing.

- **Commit & Git**
- Commit messages should reference Jira key, e.g.:
  - git commit -m "EMP-101: Implement employee registration form"
  - Branching strategy:
    - main → production
    - develop → integration
    - feature/<jira-key> → task-specific work

FULL ENTERPRISE WORKFLOW (5)

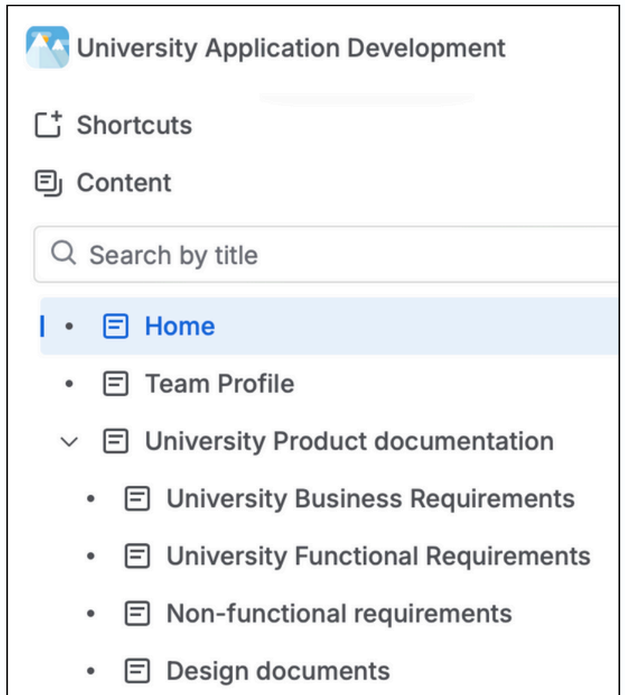
- **Pull Request (PR)**
  - Developers open PRs from feature branches to develop.
  - Team members or team leader review code and check automated tests.

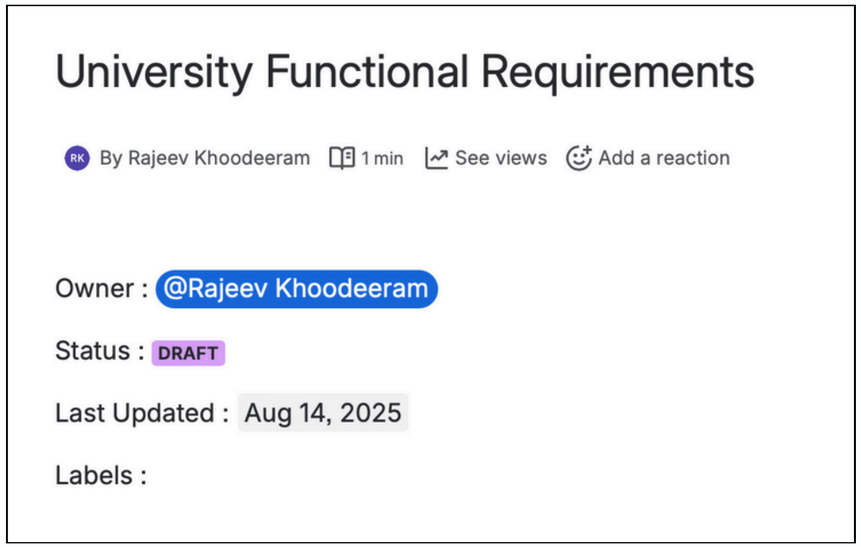
- **Merge to Main**
  - After develop passes CI/CD and QA tests, team leader merges develop into main for production deployment.

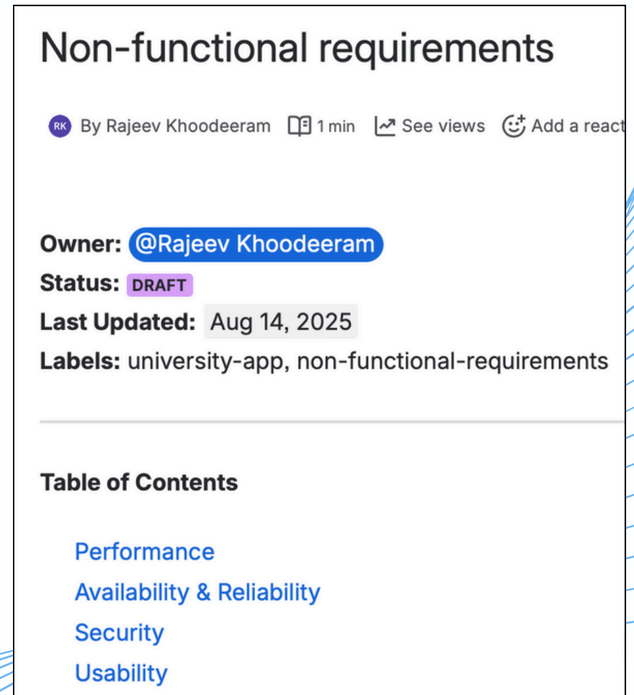
BUSINESS REQUIREMENTS - CONFLUENCE

- Identify core actors and their responsibilities
  - Student
  - Program coordinator
  - Lecturer, etc

- Write functional & non-functional requirements

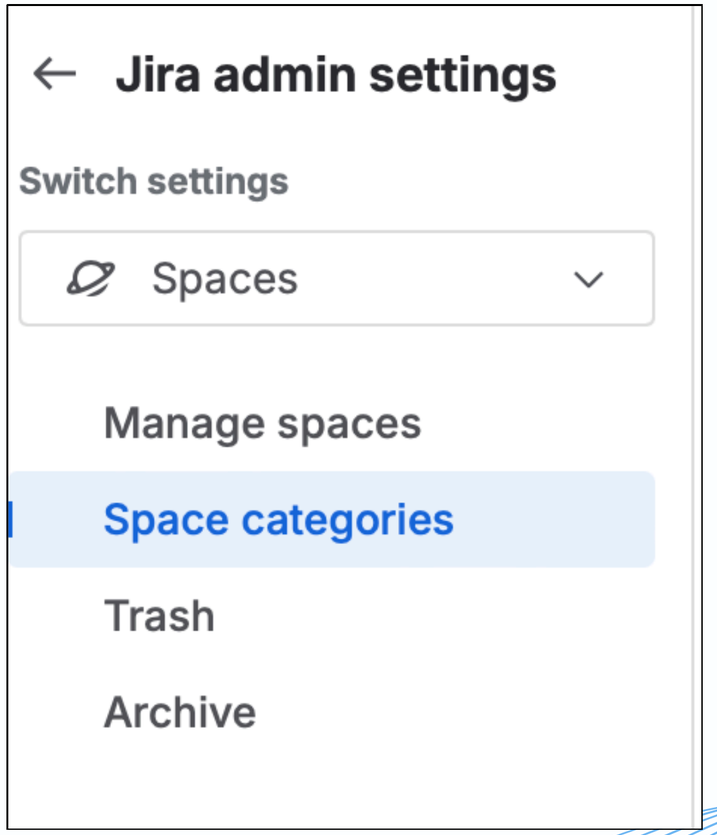


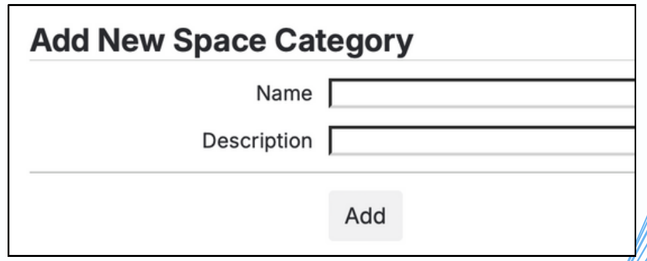




TRACK PROJECT WORK - JIRA (1)

- Works by organizing the whole project mainly as :
  - Category
    - Projects
      - Epics
      - User Stories
      - Subtasks





TRACK PROJECT WORK - JIRA (2)

- **Jira Space / Project** : STUDENT

- **Epic 1**: Student Registration & Authentication
  - Purpose: Onboard students and allow them to access the system.

- **User Stories under this Epic**:
  - STUDENT-1011: Student can submit registration form with personal details.
  - STUDENT-1012: Student can log in and check registration status (Pending, Approved, Rejected).
  - STUDENT-1013: Student receives confirmation email after registration.

CUCUMBER - GHERKINS

- **Epic 1** – Student Features
  - **User Story 1**: Student Self-Registration
  - **Feature**: Student Self-Registration - *gherkin*
  - **Scenario**: Successful registration
    - Given I am on the student registration page
    - When I enter valid personal details
    - And submit the registration form
    - Then I should see a confirmation message
    - And my registration status should be "Pending Review"
  - **Scenario**: Registration with missing fields
    - Given I am on the student registration page
    - When I leave required fields blank
    - And submit the form
    - Then I should see validation errors for each missing field

CUCUMBER - GHERKINS

- **Epic 1 – Student Features**
  - **User Story 2**: Status Check
  - gherkin

  - **Feature**: Student Registration Status Check
    - Scenario: View registration status
    - Given I am a logged-in student
    - When I navigate to the status page
    - Then I should see my registration status as either "Pending Review" or “Approved"

FULL STACK DEV

Building Entity Relationships

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

TYPICAL ENTERPRISE WORKFLOW

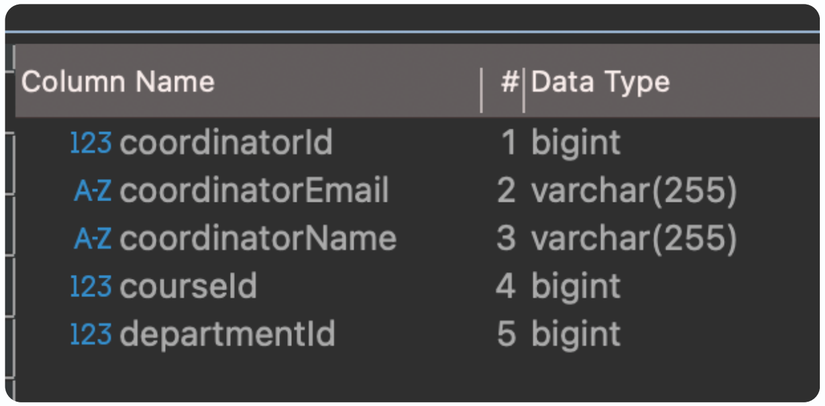
- Developer defines or updates entities in Java.
- Run Hibernate locally (**application-dev.properties** !!) with ddl-auto=create to test.
- Generate the SQL schema from Hibernate (e.g., via SchemaExport).
- Pass SQL to the DBA for review and modification.
- Apply via a migration tool (Flyway/Liquibase).
- Run app in production (USE OF **application-prod.properties** !!) with ddl-auto=validate.

ENTITIES

- **Coordinator**
  - A coordinator manages a course and belongs to a department
- **Course**
  - A course belongs to a department (a department has many courses !)
- **Department**
  - A department is found in a faculty and has a head who is a lecturer
- **Faculty**
  - A faculty has a dean who is a lecturer and it has many departments
- **Lecturer**
  - A lecturer belongs to a department
- **Module**
  - A module belongs to a course and has one lecturer assigned to it
- **ModuleEnrolment**
  - Students enrolled in modules
- **CourseEnrolment**
  - Students enrolled in one and only one course
- **Student**

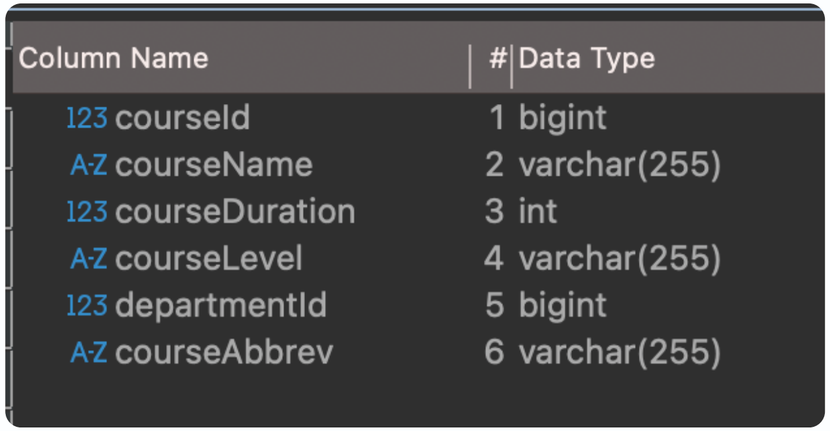
ENTITIES

- **Coordinator**
  - A coordinator manages a course and belongs to a department



Column Name	#	Data Type
123 coordinatorId	1	bigint
A-Z coordinatorEmail	2	varchar(255)
A-Z coordinatorName	3	varchar(255)
123 courseId	4	bigint
123 departmentId	5	bigint

- **Course**
  - A course belongs to a department (a department has many courses !)



Column Name	#	Data Type
123 courseId	1	bigint
A-Z courseName	2	varchar(255)
123 courseDuration	3	int
A-Z courseLevel	4	varchar(255)
123 departmentId	5	bigint
A-Z courseAbbrev	6	varchar(255)

ENTITIES

- **Department**
  - A department is found in a faculty and has a head who is a lecturer

Column Name	#	Data Type
123 departmentId	1	bigint
A-Z departmentName	2	varchar(255)
A-Z departmentCode	3	varchar(255)
123 facultyId	4	bigint
123 lecturerId	5	bigint

- **Faculty**
  - A faculty has a dean who is a lecturer and it has many departments

Column Name	#	Data Type
123 facultyId	1	bigint
A-Z facultyname	2	varchar(255)
A-Z facultycode	3	varchar(255)
A-Z facultyPhone	4	varchar(255)
123 facultyDean	5	bigint
A-Z facultyEmail	6	varchar(255)
A-Z facultyDescription	7	longtext

ENTITIES

- **Lecturer**
  - A lecturer belongs to a department

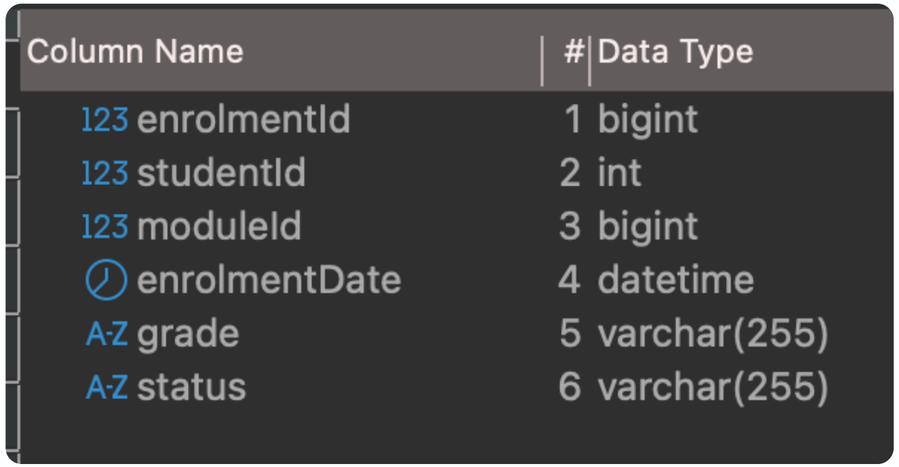
Column Name	#	Data Type
123 lecturerId	1	bigint
A-Z lecturerFirstName	2	varchar(255)
A-Z lecturerLastName	3	varchar(255)
A-Z lecturerEmail	4	varchar(255)
123 departmentId	5	bigint
lecturerHireDate	6	datetime
A-Z lecturerTitle	7	varchar(255)

- **Module**
  - A module belongs to a course and has one lecturer assigned to it

Column Name	#	Data Type
123 moduleId	1	bigint
A-Z moduleName	2	varchar(255)
A-Z moduleCode	3	varchar(255)
123 moduleCredits	4	int
123 courseId	5	bigint
123 lecturerId	6	bigint
123 moduleSemester	7	int

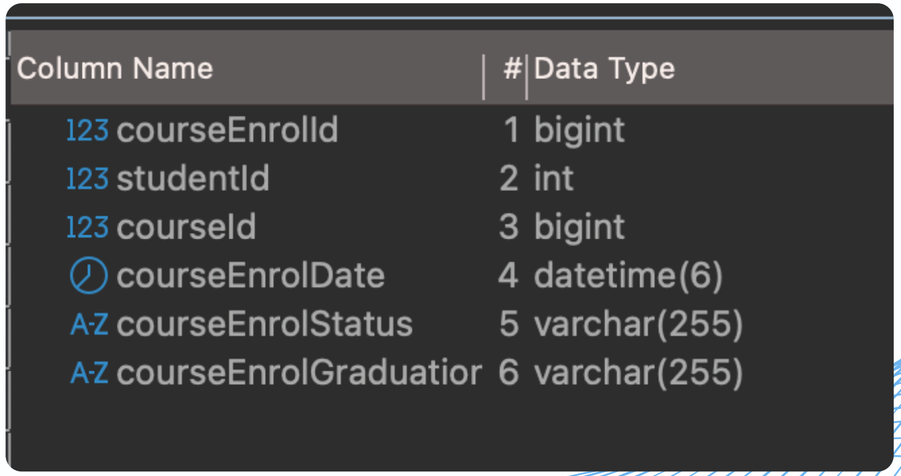
ENTITIES

- **ModuleEnrolment**
  - Students enrolled in modules



Column Name	#	Data Type
123 enrolmentId	1	bigint
123 studentId	2	int
123 moduleId	3	bigint
enrolmentDate	4	datetime
A-Z grade	5	varchar(255)
A-Z status	6	varchar(255)

- **CourseEnrolment**
  - Students enrolled in one and only one course



Column Name	#	Data Type
123 courseEnrolId	1	bigint
123 studentId	2	int
123 courseId	3	bigint
courseEnrolDate	4	datetime(6)
A-Z courseEnrolStatus	5	varchar(255)
A-Z courseEnrolGraduatior	6	varchar(255)

ENTITIES

- **Student**

Column Name	#	Data Type
123 student_id	1	int
A-Z student_firstname	2	varchar(255)
A-Z student_lastname	3	varchar(255)
A-Z student_dob	4	varchar(255)
A-Z student_gender	5	varchar(255)
A-Z student_phone	6	varchar(255)
A-Z student_address	7	varchar(255)
A-Z student_status	8	varchar(255)
A-Z student_email	9	varchar(255)
A-Z enrolment_date	10	varchar(255)
A-Z student_city	11	varchar(100)

DEFINING RELATIONSHIPS

- There are many types of relationships :
  - **one-to-one**
    - a coordinator can oversee a course, and a course has only one coordinator assigned to it
  - **one-to-many (many-to-one)**
    - one student can unenroll in only one course and a course has many students enrolled in it
  - **many-to-many**
    - a student can subscribe to many clubs
    - a club has many students as members

COURSE - COORDINATOR

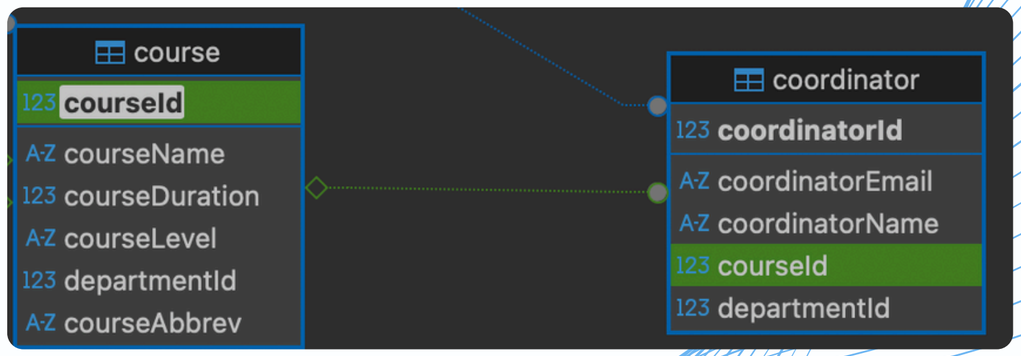
- Type: **One-to-One**

- Description: One Program Coordinator can oversee only one Course, and each Course is managed by only one Program Coordinator.

- Implementation: **Coordinator.courseId** is a Foreign Key referencing **Course.courseId** (here primary key).

```java
@OneToOne
@JsonBackReference
@JoinColumn(name = "courseId", nullable = false, unique = true)
private Coordinator coordinator;
```

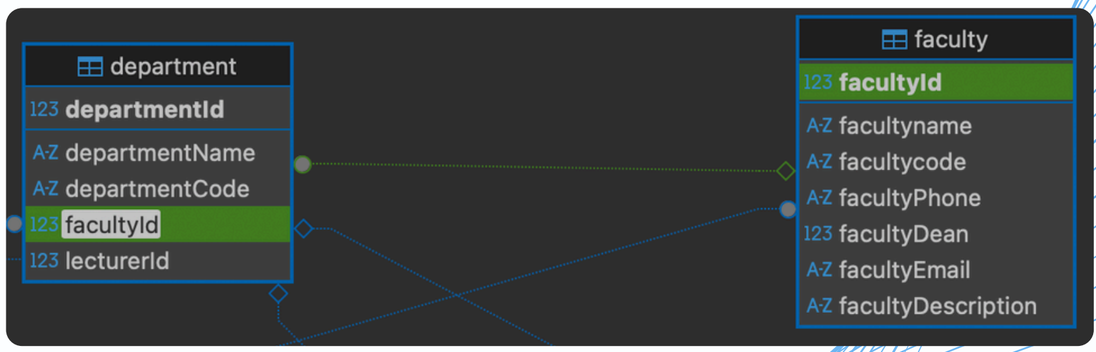
```java
@OneToOne
@JoinColumn(name = "courseId", nullable = false, unique = true)
private Course course;
```



DEPARTMENT - FACULTY

- Type: **Many-to-One**
- Description: One Department belongs to only one Faculty and one Faculty can have many Departments.
- Implementation: **Department.facultyId** is a Foreign Key referencing **Faculty.facultyId** (here primary key).

```java
// one department belongs to one faculty
@ManyToOne
@JoinColumn(name = "facultyId", nullable = false)
@JsonBackReference
private Faculty faculty;
```




```java
@OneToMany(mappedBy = "faculty", cascade = CascadeType.ALL, orphanRemoval = true)
private List<Department> departments = new ArrayList<>();
```

DEPARTMENT - LECTURER

- Type: **One-to-Many**

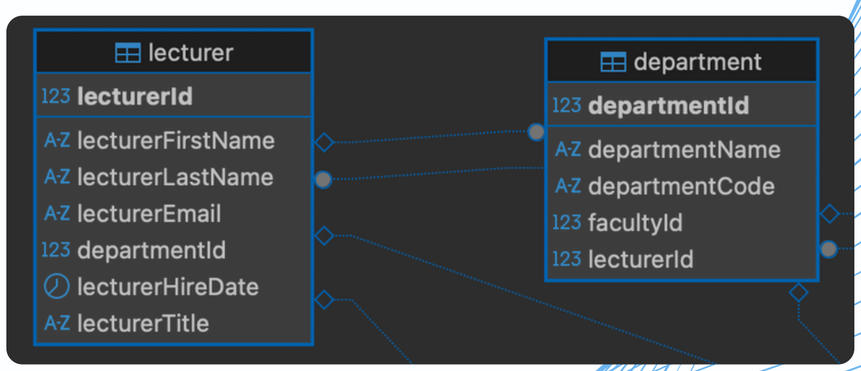
- Description: One Department has many lecturers and one lecturer belongs to only one Department.

- Implementation: **Department.lecturerId** is a Foreign Key referencing **Lecturer.lecturerId** (here primary key).
- **But, a department also has a head (who is a lecturer) --- complication !**

```java
// Department has many lecturers (regular members)
@OneToMany(mappedBy = "department", cascade = CascadeType.ALL, orphanRemoval = true)
@JsonManagedReference
private List<Lecturer> lecturers = new ArrayList<>();
```

```java
// one lecturer can head only one department
@OneToOne
@JoinColumn(name = "lecturerId", unique = true)
private Lecturer headOfDepartment;
```

```java
// many lecturers belong to one department
@ManyToOne
@JoinColumn(name = "departmentId", nullable = false)
@JsonBackReference
private Department department;
```



FULL STACK DEV

Creating the database - mySQL

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

FULL DATABASE OF UNVERSITYAPP

See ==> 2025-08-22-UniversityMySQLDB

FULL STACK DEV

Designing backend API for remaining entities

Presented by:

Rajeev Khoodeeram

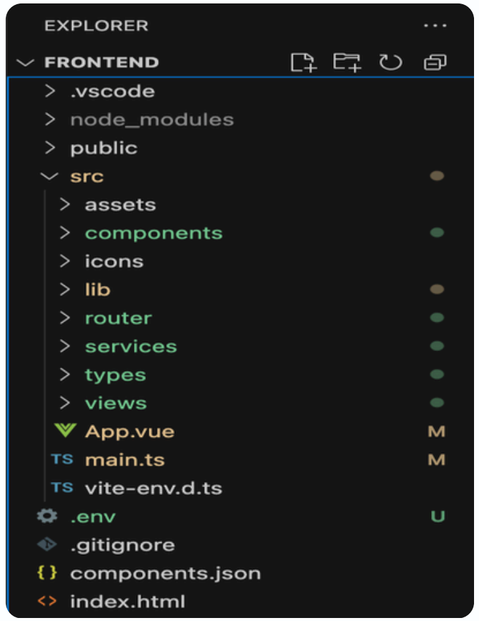
OCTOBER 2025

ARCHITECTURE

- **@CrossOrigin(origins = "http://localhost:5176")** → in controller !

- Create new directory inside your git repo - here it is frontend folder inside HumanResourceApp
  - Navigate inside this directory
  - Run : **>> npm create vue@latest frontend**
- Make sure you install extensions for Vue in Visual Studio Code
  - Check all options
  - Navigate into your new project directory (here it is frontend) and install dependencies
  - **>>npm install**
- Run the development server to verify everything works
  - **>>npm run dev**

DON'T FORGET THE CSS FILE



- Open frontend folder in VS Code
  - >>npm install

- Remember
  - index.html calls —> main.ts which calls —> App.vue

- Run npm install vue-router (will create folder router and add index.js (see github))
  - This provides all the routes for your app - if not generated then you create it manually !!

- Important to note here - We will have two interfaces :
  - one is the website the public views
  - the other one is the admin view for managing the database

MAIN.TS

We will modify the main.ts to cater for routing
main.ts

```
import { createApp } from "vue";
import App from "./App.vue";
import "./assets/index.css";
import router from "./router/index.js";

const app = createApp(App);
app.use(router);
app.mount("#app");
```

main.ts has been modified to take routing into consideration

MAIN.TS

We will modify the main.ts to cater for routing
main.ts

```
import { createApp } from "vue";
import App from "./App.vue";
import "./assets/index.css";
import router from "./router/index.js";

const app = createApp(App);
app.use(router);
app.mount("#app");
```

main.ts has been modified to take routing into consideration

UTILITIES FILE

- Centralized ts or utils files (it is in the lib folder)
  - Create a utils folder in src
  - Create your file (ex here formatDate.ts) or add your function in the same utils.ts

- Formatting date from database in Due
  - >>npm install dayjs

- import dayjs from "dayjs";

- export function formatDate(dateStr: string): string {
- return dayjs(dateStr).format("DD-MM-YYYY");
- }

FULL STACK DEV

Creating endpoints for API Testing

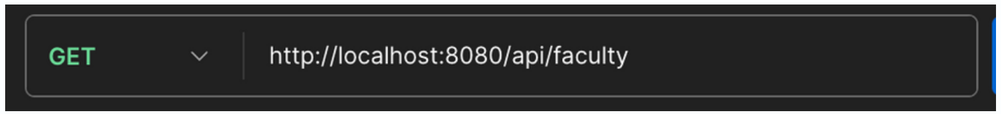
Presented by:

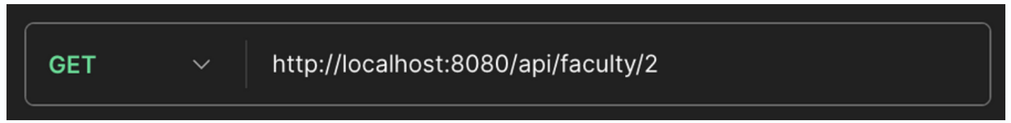
Rajeev Khoodeeram

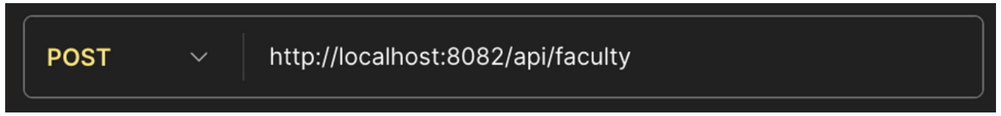
OCTOBER 2025

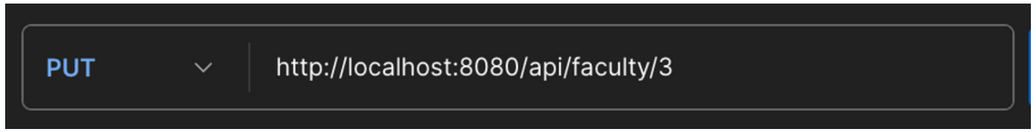
ENDPOINTS - ENTITY

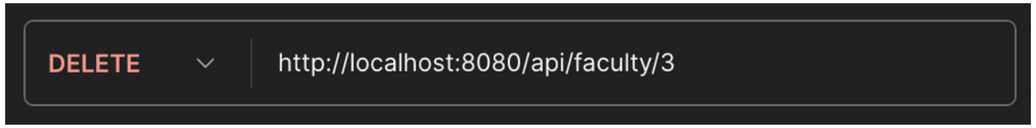
- This section outlines the REST API endpoints and the underlying business logic necessary to power the frontend features.

- The API will use standard REST principles (we use ***postman***)
  - GET
    - 

    - 

  - POST
    - 

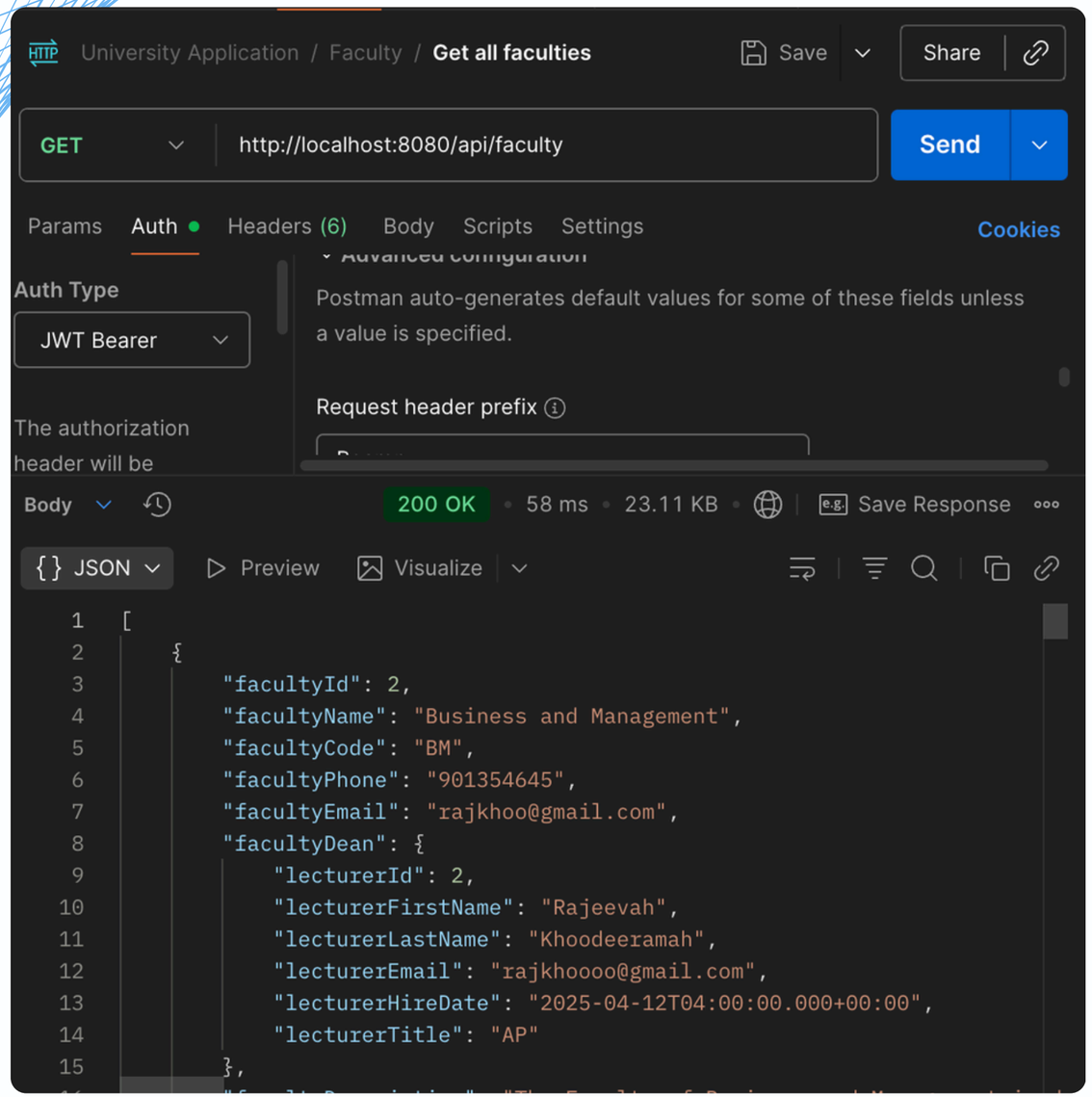
  - PUT
    - 

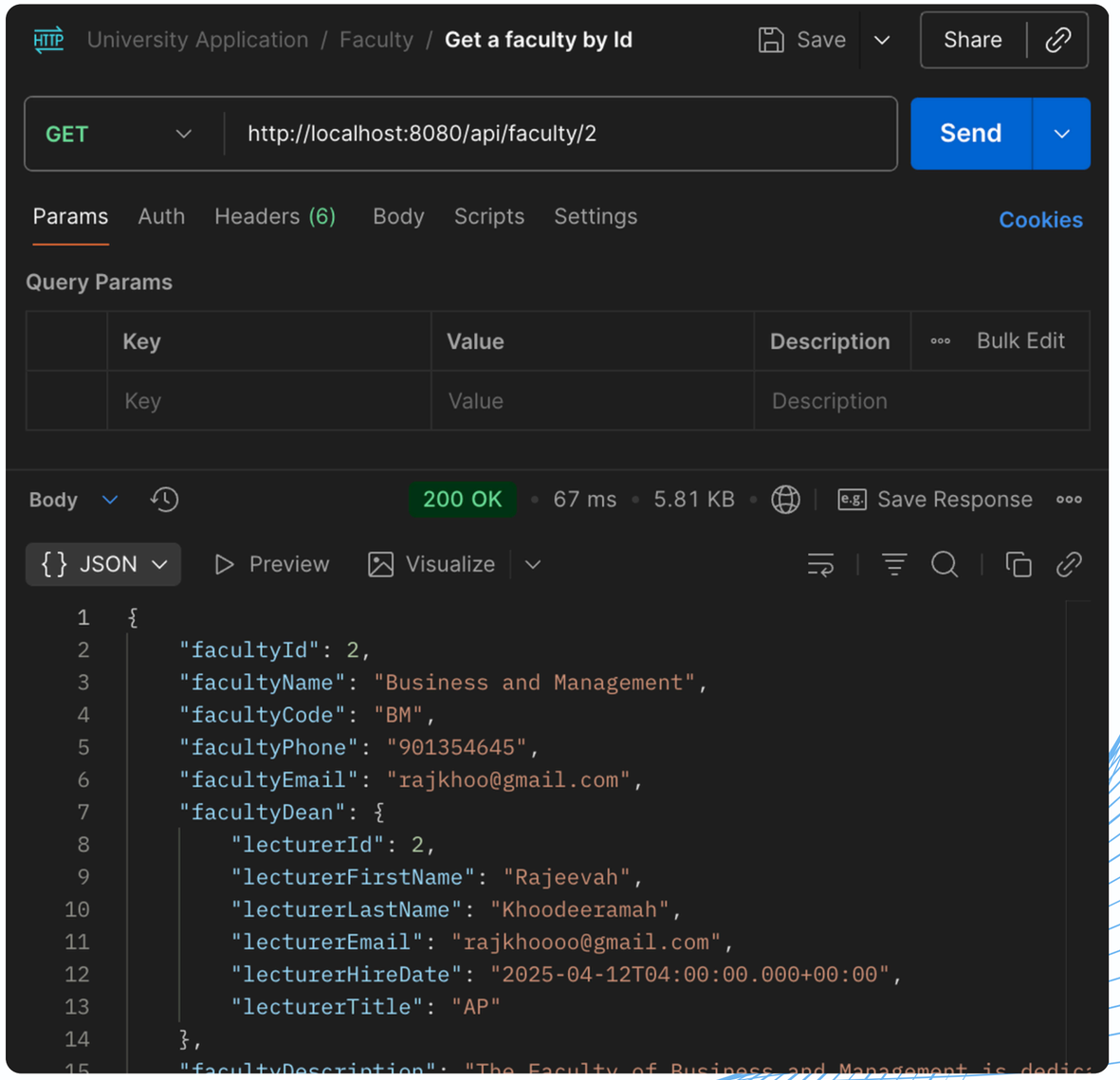
  - DELETE
    - 

FACULTY API ENDPOINTS (/API/FACULTY)

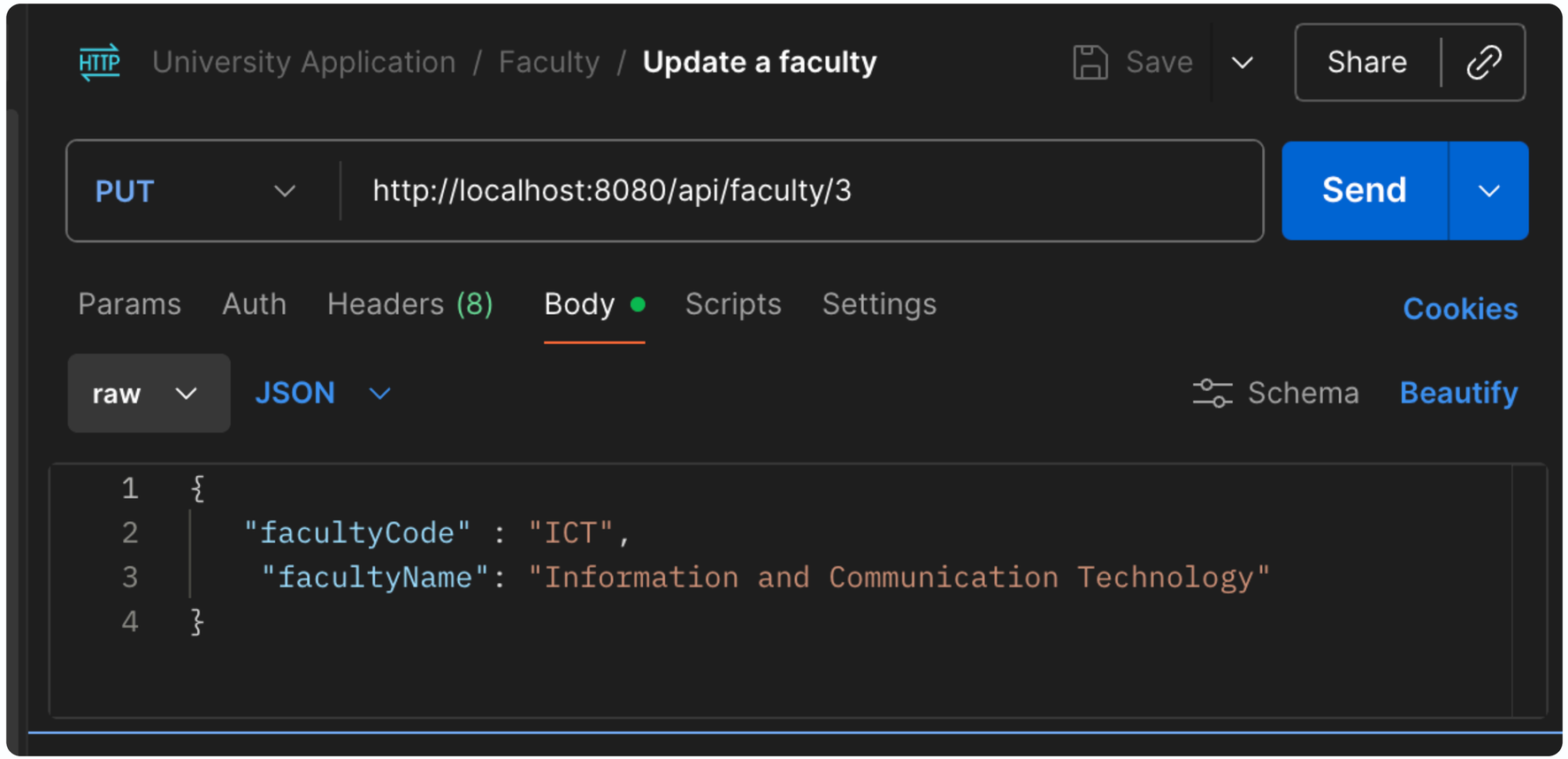
- GET /api/faculty: Retrieve a list of all students (with optional filtering/pagination parameters).
- GET /api/faculty/{id}: Retrieve a single student by their ID.
- POST /api/faculty: Create a new student.
- PUT /api/faculty/{id}: Update an existing student's details.
- DELETE /api/faculty/{id}: Delete a student.

FACULTY API ENDPOINTS (/API/FACULTY)

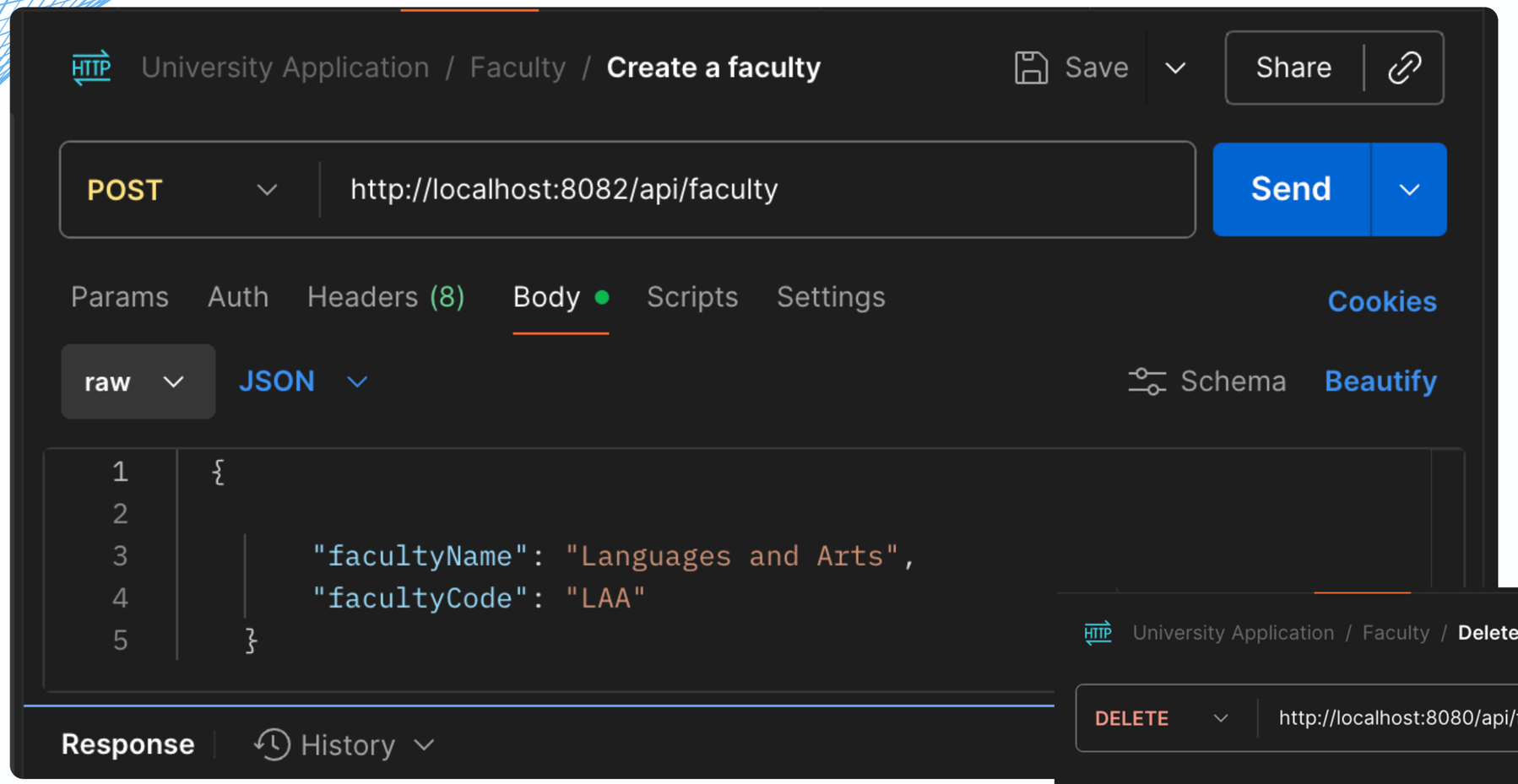


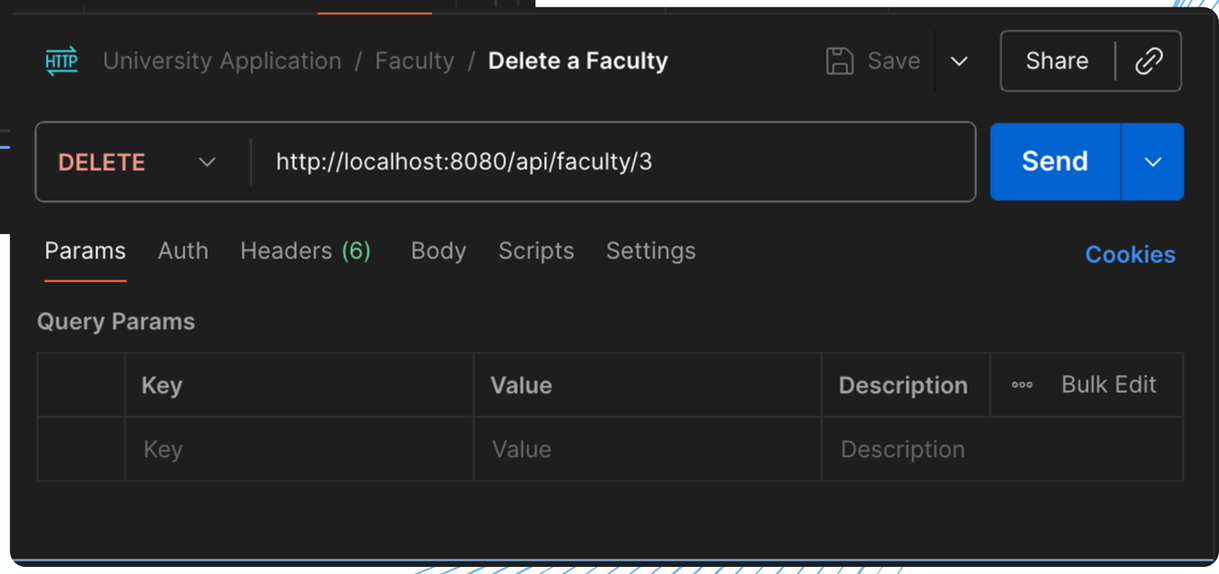


FACULTY API ENDPOINTS (/API/FACULTY)



FACULTY API ENDPOINTS (/API/FACULTY)





DEPARTMENT API ENDPOINTS (/API/DEPARTMENT)

- GET /api/department: Retrieve a list of all departments.
- GET /api/department/{id}: Retrieve a single department by their ID.
- POST /api/department: Create a new department.
- PUT /api/department/{id}: Update an existing department’s details.
- DELETE /api/department/{id}: Delete a department.

REMAINING ENDPOINTS

- @RequestMapping("/api/coordinators")
  - **http://localhost:8080/api/coordinators**
- @RequestMapping("/api/courses")
  - **http://localhost:8080/api/courses**
- @RequestMapping("/api/course-enrolments")
  - **http://localhost:8080/api/course-enrolments**
- @RequestMapping("/api/lecturers")
  - **http://localhost:8080/api/lecturers**
- @RequestMapping("/api/modules")
  - **http://localhost:8080/api/modules**
- @RequestMapping("/api/module-enrolments")
  - **http://localhost:8080/api/module-enrolments**
- @RequestMapping("/api/students")
  - **http://localhost:8080/api/students**

FULL STACK DEV

Add Basic HTTP Authentication using JWT and Redis

Presented by:

Rajeev Khoodeeram

OCTOBER 2025

WHY REDIS?

- In-memory blacklists (using a Set) are not suitable for production because:
  - They do not scale across multiple application instances.
  - Tokens are lost on application restart.
  - Redis provides a distributed, persistent, and performant solution to store blacklisted tokens with an automatic expiry (TTL) matching the token's lifespan.

- Prerequisites
  - Redis server running (local or cloud).
  - Spring Boot Redis dependency.
  - Add the necessary dependencies to pom.xml :

ADD DEPENDENCIES TO POM.XML :

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-redis</artifactId>
</dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
```

```
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
```

APPLICATION.PROPERTIES

- spring.data.redis.host=localhost
- spring.data.redis.port=6379
- spring.data.redis.password=
- spring.data.redis.timeout=60000
- jwt.secret=abcdefghijklmnopqrstuvwxyz123456
- jwt.expiration-ms=86400000

- logging.level.org.springframework.security=DEBUG

INSTALLING REDIS

- **On Mac :**
  - brew install redis
  - brew services start redis # to run as a background service

```
(base) rajeev@Rajeev-Khoodeeram ~ % redis-cli PING
PONG
(base) rajeev@Rajeev-Khoodeeram ~ %
```

- **On Linux**
  - sudo apt update
  - sudo apt install redis-server
  - sudo systemctl enable redis-server --now

IMPORTANT

- Spring Boot does not require Redis for JWT
  - *but Redis is often used with JWT in real-world applications to solve some key limitations of pure stateless JWT authentication.*

- JWT by Default is **Stateless**
- How JWT works:
  - The server signs a token and gives it to the client after login.
  - The client sends the token with every request.
  - The server verifies the signature and expiration without needing a database or session storage.

- Advantage: No need to store anything on the server → scales easily.

IMPORTANT

- **Problem**: No Central Control (Token Revocation)
  - If a user logs out or an admin wants to invalidate a token before it expires, pure JWT cannot handle this easily.
  - Tokens remain valid until their expiration time because the server has no record of them
- **Solution**: Redis as a Token Store / Blacklist

SPRING BOOT APPLICATIONS OFTEN INTEGRATE REDIS

- **Token Blacklisting**
  - When a user logs out, you can store the token's jti (JWT ID) or hash in Redis with a TTL (time-to-live).
  - During every request, check Redis to see if the token is blacklisted.
- **Refresh Token Management**
  - Store refresh tokens in Redis with an expiration time.
  - This allows issuing new access tokens securely.
- **Session-like Features**
  - Store user metadata (roles, permissions) in Redis for quick lookups.
  - Allows immediate role updates without waiting for JWT expiry.
- **High Performance**
  - Redis is in-memory, extremely fast, and supports TTL for automatic token expiration.

USING REDIS INSIGHT

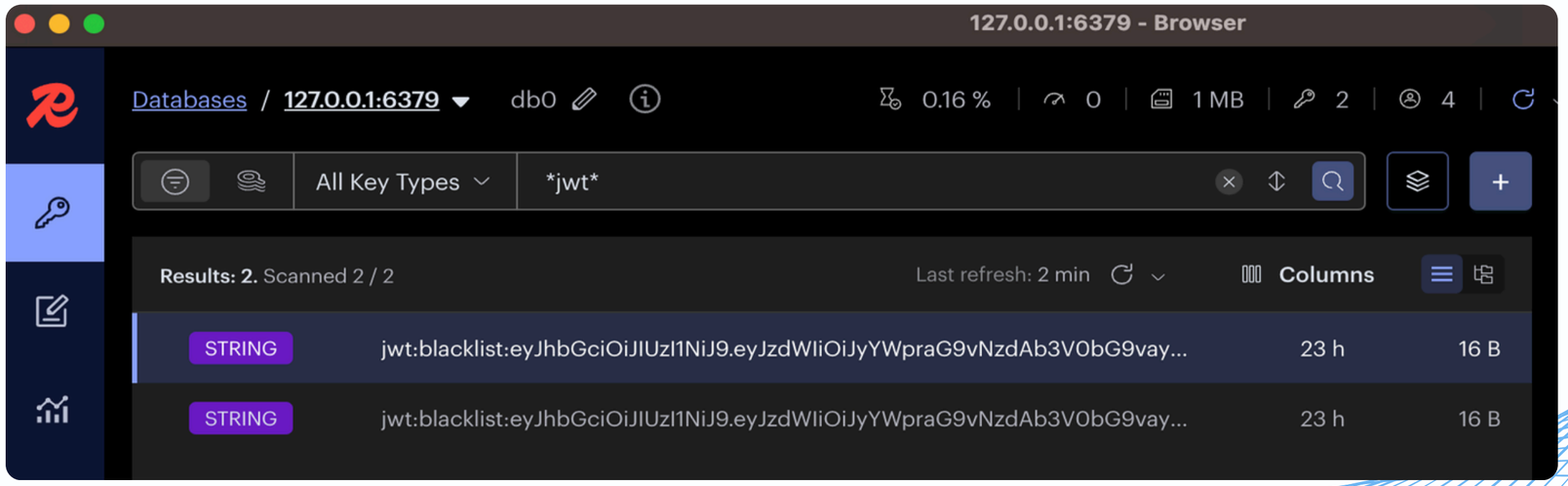
- You can use Redis Insight to view keys and other information managed by the Redis database:
- Download it from 👉 https://redis.io/insight/

From terminal (notice there are TWO) :

```
(base) rajeev@Rajeev-Khoodeeram ~ % redis-cli
127.0.0.1:6379> select 0
OK
127.0.0.1:6379>

127.0.0.1:6379> keys *
1)
"jwt:blacklist:eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJyYWpraG9vNzdAb3V0
bG9vay5jb20iLCJyb2xlcyI6W3siYXV0aG9yaXR5IjoiUk9MRV9zdHVkZW50In1d
LCJpYXQiOjE3NTg3NjA3ODcsImV4cCI6MTc1ODg0NzE4N30.n9I5j8OuWnxUMURa
QgdCAcPvvwkmjQz1F8DI8NgKIVs"
2)
“jwt:blacklist:eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJyYWpraG9vNzdAb3V0
bG9vay5jb20iLCJyb2xlcyI6W3siYXV0aG9yaXR5IjoiUk9MRV9zdHVkZW50In1d
LCJpYXQiOjE3NTg3NTcyOTgsImV4cCI6MTc1ODg0MzY5OH0.nMd9-5nhpScp8TlO
zAsHT6JZdm0dHRS0D5CwT_k6pe8"
```

HOW TO USE REDIS INSIGHT

- Connect to Your Redis Instance
- When you open RedisInsight:
- Click **Add Redis Database**.
- Fill in:
  - **Host**: localhost (or your server IP/domain if remote)
  - **Port**: 6379 (default)
  - **Password**: If your Redis instance requires one (e.g., requirepass).
- Click Connect.
  -



TYPICAL SPRING BOOT + REDIS JWT FLOW

- **User Logs In**
  - Spring Boot generates a JWT and stores a refresh token (or token ID) in Redis.

- **API Request**
  - User sends JWT in the Authorization header.
  - Spring Security verifies the signature and checks Redis for blacklisting or session validity.

- **Logout**
  - The JWT or its ID is added to Redis as a blacklist entry until it expires.

FULL STACK DEV

Backend authentication with Redis

Presented by:

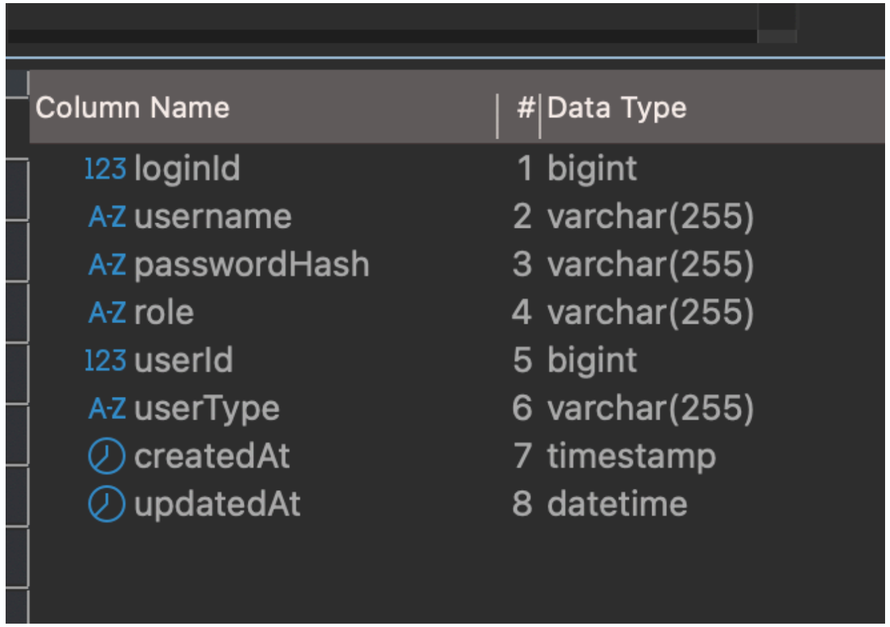
Rajeev Khoodeeram

OCTOBER 2025

STEP 1

- First create login table as shown previously.
- Then create Login model, LoginRepository and Login /AuthController

```java
@Entity
@Table(name = "login")
public class Login {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long loginId;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String passwordHash;

    @Column(nullable = false)
    private String role;

    @Column(nullable = false, unique = true)
    private int userId;
```



Column Name	#	Data Type
123 loginId	1	bigint
A-Z username	2	varchar(255)
A-Z passwordHash	3	varchar(255)
A-Z role	4	varchar(255)
123 userId	5	bigint
A-Z userType	6	varchar(255)
createdAt	7	timestamp
updatedAt	8	datetime

LOGIN TABLE

- CREATE TABLE login (
- loginId INT AUTO_INCREMENT PRIMARY KEY,
- username VARCHAR(100) NOT NULL UNIQUE,
- passwordHash VARCHAR(255) NOT NULL,
- role ENUM('STUDENT','LECTURER','COORDINATOR','ADMIN') NOT NULL,
- userId INT NOT NULL, -- ID of the actual user
- userType ENUM('STUDENT','LECTURER','COORDINATOR','ADMIN') NOT NULL,
- createdAt DATETIME DEFAULT CURRENT_TIMESTAMP,
- updatedAt DATETIME DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
-);

LOGIN TABLE

Column Name	#	Data Type
123 loginId	1	bigint
A-Z username	2	varchar(255)
A-Z passwordHash	3	varchar(255)
A-Z role	4	varchar(255)
123 userId	5	bigint
A-Z userType	6	varchar(255)
createdAt	7	timestamp
updatedAt	8	datetime

2	5	rajkhoo77@outlook.com	$2a$10$oqFr.YSzChyTFPrQOr9/oOyNrJpSdhOKeuO4tkV15	student
3	6	admin.uot@gmail.com	$2a$10$nyueFazbim8mcSrtBpjiLeP4xl93cU8ZnSLZkGXD7	Student
4	7	dev@outlook.com	$2a$10$6/HWdq.ckZ1Ohr4tA2b5f.nprDlJTjtCavnVd8Sn.nkl	student
5	8	olivia@yahoo.com	$2a$10$ZdsyQ1YoAfD.WsVPzcWpV.bGZPonFKY01yrqww2v	admin
6	9	jagpatibabu@yahoo.com	$2a$10$.rr9Ldmk3lOzMHcrB6ggguzOZr5LjPLUcZ6a4kNjh	admin
7	10	sdsd@yahoo.com	$2a$10$QJC3HeFmVuHc5f9z9F0GZ.K4EjVsDPsDm3rAO.D	admin
8	11	asdsyahoo.com	$2a$10$Exgl27o0tZKA4cYVN0vQzODAmvPtKnMzeSO1TV(	admin
9	12		$2a$10$2j9pgkxk8D/MirduMK9Wz.tTDz1PrRCgZrgWO.hAE	admin
10	13	test@admin.com	$2a$10$l0i4pIK5F3mpSbw5wGkE1eBJimuKsbj3nGJtxMz5F	admin
11	14	lect@osas.com	$2a$10$87XF0pu.bljiQ1GKmRXAAOSqv5SoypZVvzvyF/3aF	lecturer

LOGINCONTROLLER + REPOSITORY

- @RestController
- @RequestMapping("/api/auth")
- @CrossOrigin(origins = "http://localhost:4200")
- public class LoginController { }

```java
@PostMapping("/register")
public ResponseEntity<?> register(@RequestBody RegisterRequest req)
```

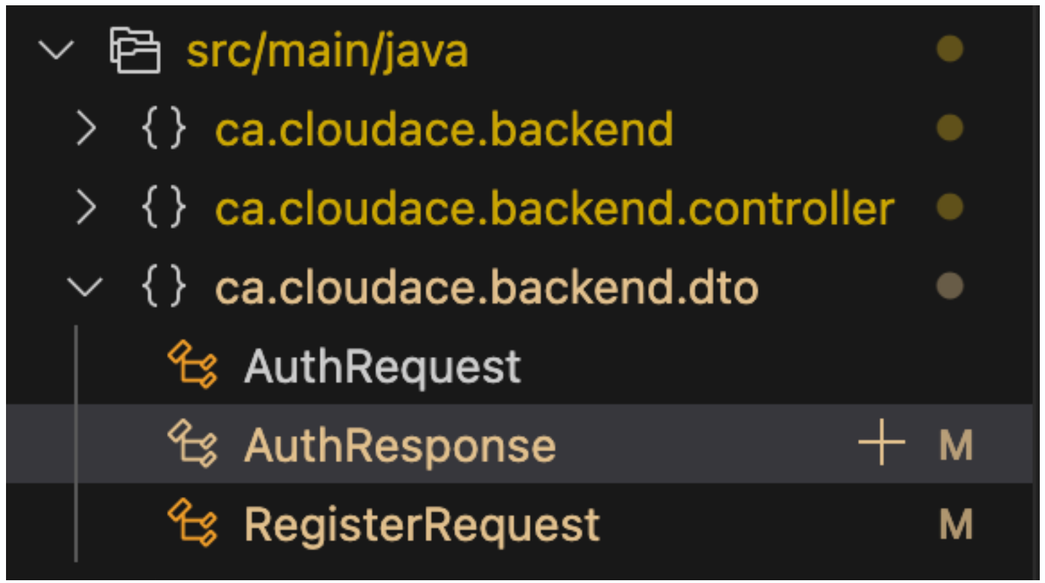
```java
@PostMapping("/login")
public ResponseEntity<?> login(@RequestBody AuthRequest req) {
```

```java
@PostMapping("/logout")
public ResponseEntity<?> logout(@RequestHeader("Authorization") String authHeader) {
```

```java
public interface LoginRepository extends JpaRepository<Login, Long> {

    Optional<Login> findByUsername(String username);

    Login findByUserId(int userId);

    Login findByUserType(String userType);

    boolean existsByUsername(String username);

    boolean existsByUserId(int userId);

}
```

STEP 2 - AUTH FILES

- Create the following files in **dto** folder
  - **AuthRequest**
  - **AuthResponse**
  - **RegisterRequest**




```java
public record AuthRequest(String username, String password) {
}
```

```java
public record AuthResponse(String token, String tokenType, String fullName, String role) {
}
```

```java
public record RegisterRequest(String username, String password, String role) {
}
```

WHY A CUSTOM USERDETAILSSERVICE IS NEEDED

- Spring Security does not know how you store users (MySQL, PostgreSQL, MongoDB, etc.).

- When a login request comes in, the AuthenticationManager will:
  - Call UserDetailsService.loadUserByUsername(username)
  - Expect a UserDetails object with:
    - username
    - password hash (e.g., bcrypt)
    - roles/authorities

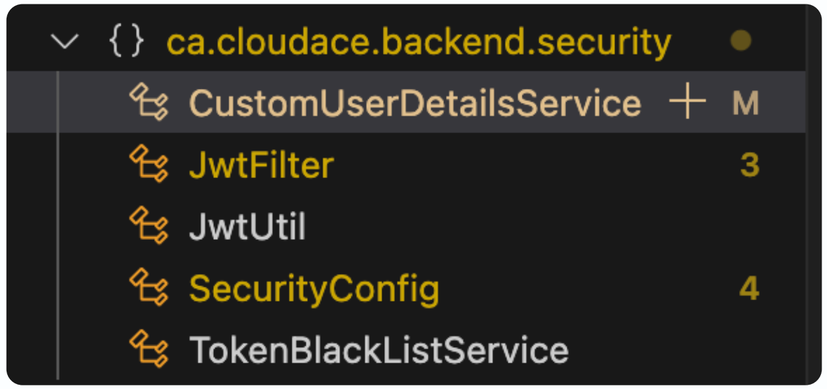
- Without a custom service, Spring cannot fetch your users.

STEP3 : WRITE CUSTOMUSERDETAILSSERVICE

```java
@Service
@Primary
public class CustomUserDetailsService implements UserDetailsService {
        @Autowired
        private LoginRepository loginRepository;
        Qodo Gen: Test this method | Qodo Gen: Test this method | Qodo Gen: Test this method
        @Override
        public UserDetails loadUserByUsername(String username)
                        throws UsernameNotFoundException {
                Optional<Login> loginOpt = loginRepository.findByUsername(username);
                Login user = loginOpt.orElseThrow(() -> new UsernameNotFoundException(msg:"User not found"));
                System.out.println("UserDetails password: " + user.getPasswordHash());
                System.out.println("Raw password: " + username);

                return User.builder()
                                .username(user.getUsername())
                                .password(user.getPasswordHash()) // BCrypt hashed
                                .roles(user.getRole()) // e.g., "USER
                                .build();
        }
}
```

CREATE SECURITY JAVA CLASSES

- Create the following files in **security** folder
  - **JwtFilter**
  - **JwtUtil**
  - **SecurityConfig**
  - **TokenBlackListService**



- ***JWT + Redis blacklisting protects your endpoints !!***

HOW IT WORKS WITH JWT

- **Login flow**
- AuthController calls AuthenticationManager.authenticate().
  - Verifies credentials using UserDetailsService
- AuthenticationManager delegates to your CustomUserDetailsService.
- Password is checked using the PasswordEncoder.
  - Hashes & validates passwords (e.g., BCrypt)
- On success, you generate a JWT and return it to the client.

- **Request validation**
  - JwtFilter extracts the username from the token.
  - Calls CustomUserDetailsService.loadUserByUsername to load authorities.
- Spring Security checks if the user is still valid.

JWT FILES

- **JwtFilter**
  - Uses UserDetailsService (provided by Spring) to load roles during requests
  - If you have a JwtFilter that checks the token in Redis for every request, it must skip the login and register endpoints, otherwise Spring Security blocks requests to /api/auth/login

- **SecurityConfig**

  With this setup:
    - CSRF is disabled for REST APIs
    - CORS is allowed for Angular dev server
    - /api/auth/** endpoints are public
    - All other endpoints remain secured

FULL STACK DEV

Backend authentication with Redis

Presented by:

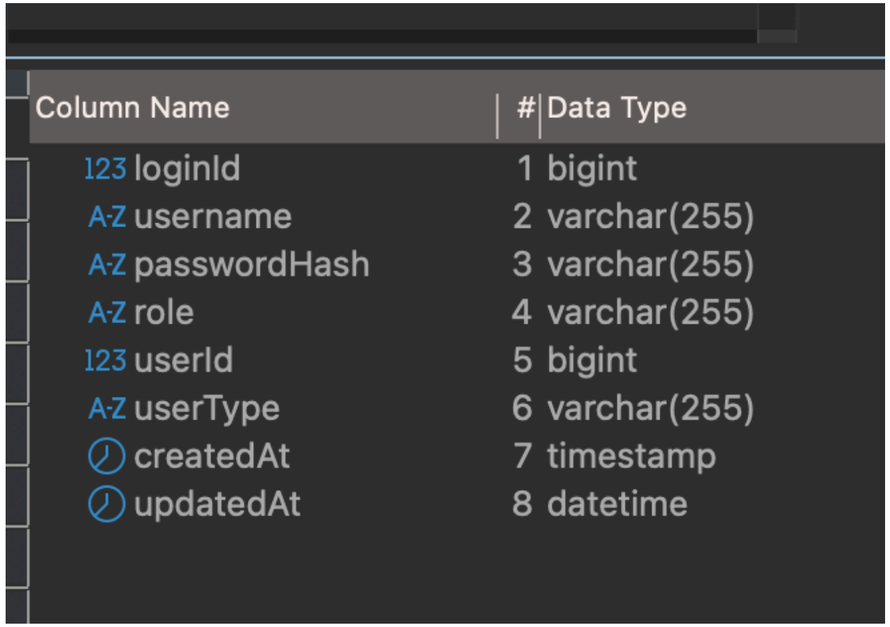
Rajeev Khoodeeram

OCTOBER 2025

STEP 1

- First create login table as shown previously.
- Then create Login model, LoginRepository and Login /AuthController

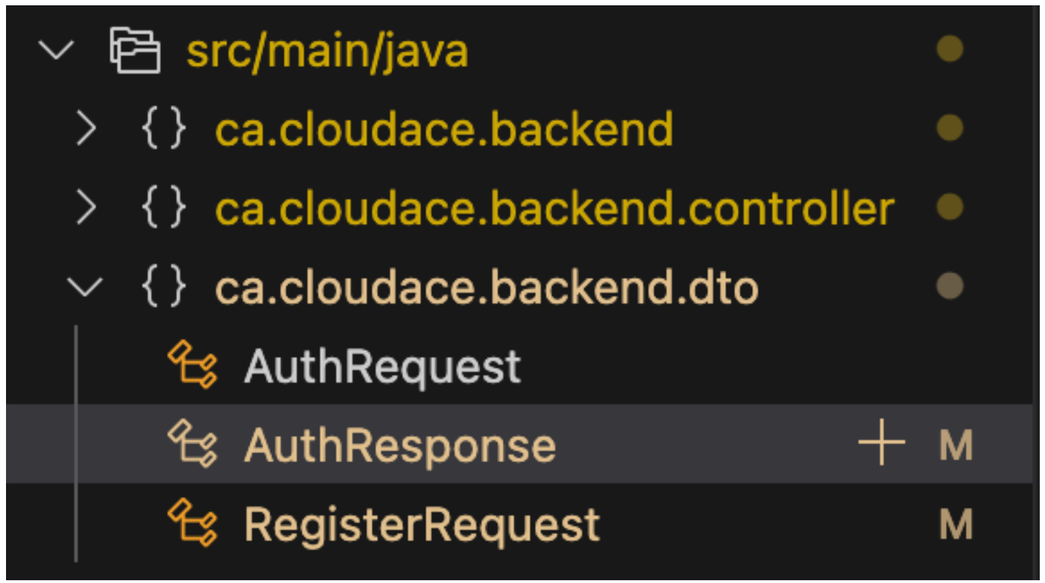
```java
@Entity
@Table(name = "login")
public class Login {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long loginId;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false)
    private String passwordHash;

    @Column(nullable = false)
    private String role;

    @Column(nullable = false, unique = true)
    private int userId;
```



Column Name	#	Data Type
123 loginId	1	bigint
A-Z username	2	varchar(255)
A-Z passwordHash	3	varchar(255)
A-Z role	4	varchar(255)
123 userId	5	bigint
A-Z userType	6	varchar(255)
createdAt	7	timestamp
updatedAt	8	datetime

STEP 2 - AUTH FILES

- Create the following files in dto folder
  - **AuthRequest**
  - **AuthResponse**
  - **RegisterRequest**




```java
public record AuthRequest(String username, String password) {
}
```

```java
public record AuthResponse(String token, String tokenType, String fullName, String role) {
}
```

```java
public record RegisterRequest(String username, String password, String role) {
}
```

WHY A CUSTOM USERDETAILSSERVICE IS NEEDED

- Spring Security does not know how you store users (MySQL, PostgreSQL, MongoDB, etc.).

- When a login request comes in, the AuthenticationManager will:
  - Call UserDetailsService.loadUserByUsername(username)
  - Expect a UserDetails object with:
    - username
    - password hash (e.g., bcrypt)
    - roles/authorities

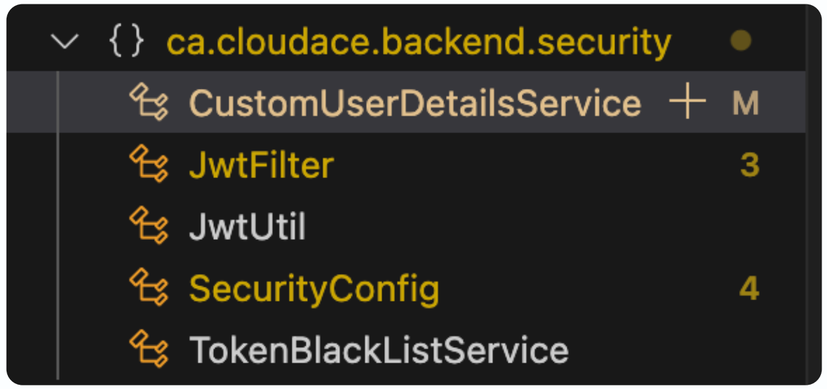
- Without a custom service, Spring cannot fetch your users.

STEP3 : WRITE CUSTOMUSERDETAILSSERVICE

```java
@Service
@Primary
public class CustomUserDetailsService implements UserDetailsService {
        @Autowired
        private LoginRepository loginRepository;
        Qodo Gen: Test this method | Qodo Gen: Test this method | Qodo Gen: Test this method
        @Override
        public UserDetails loadUserByUsername(String username)
                        throws UsernameNotFoundException {
                Optional<Login> loginOpt = loginRepository.findByUsername(username);
                Login user = loginOpt.orElseThrow(() -> new UsernameNotFoundException(msg:"User not found"));
                System.out.println("UserDetails password: " + user.getPasswordHash());
                System.out.println("Raw password: " + username);

                return User.builder()
                                .username(user.getUsername())
                                .password(user.getPasswordHash()) // BCrypt hashed
                                .roles(user.getRole()) // e.g., "USER
                                .build();
        }
}
```

CREATE SECURITY JAVA CLASSES

- Create the following files in **security** folder
  - **JwtFilter**
  - **JwtUtil**
  - **SecurityConfig**
  - **TokenBlackListService**



- ***JWT + Redis blacklisting protects your endpoints !!***

HOW IT WORKS WITH JWT

- **Login flow**
- AuthController calls AuthenticationManager.authenticate().
  - Verifies credentials using UserDetailsService
- AuthenticationManager delegates to your CustomUserDetailsService.
- Password is checked using the PasswordEncoder.
  - Hashes & validates passwords (e.g., BCrypt)
- On success, you generate a JWT and return it to the client.

- **Request validation**
  - JwtFilter extracts the username from the token.
  - Calls CustomUserDetailsService.loadUserByUsername to load authorities.
- Spring Security checks if the user is still valid.