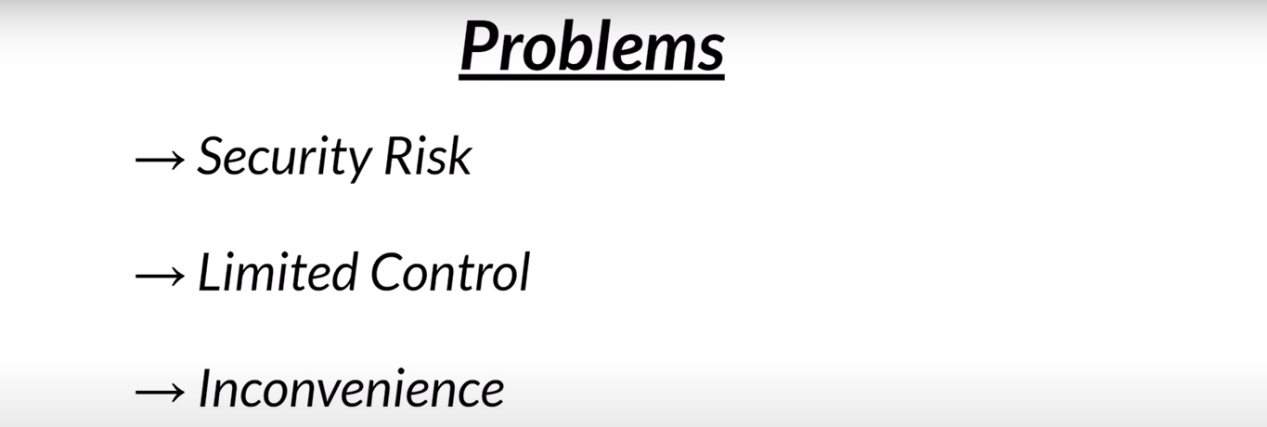
============================= OAuth2 ===============================

What Is OAuth2 and what is main purpose of OAuth2 ?

OAuth2 is mainly for authentication (logging in users securely). After successful authentication, we issue a **JWT token** that the user will use for subsequent API requests.

\*OAuth2 handles **user login** (Google/GitHub/etc.).  
\* JWT is used for **API authorization** (so users don't need to log in repeatedly).

So, OAuth2 and JWT **work together** in most modern applications.



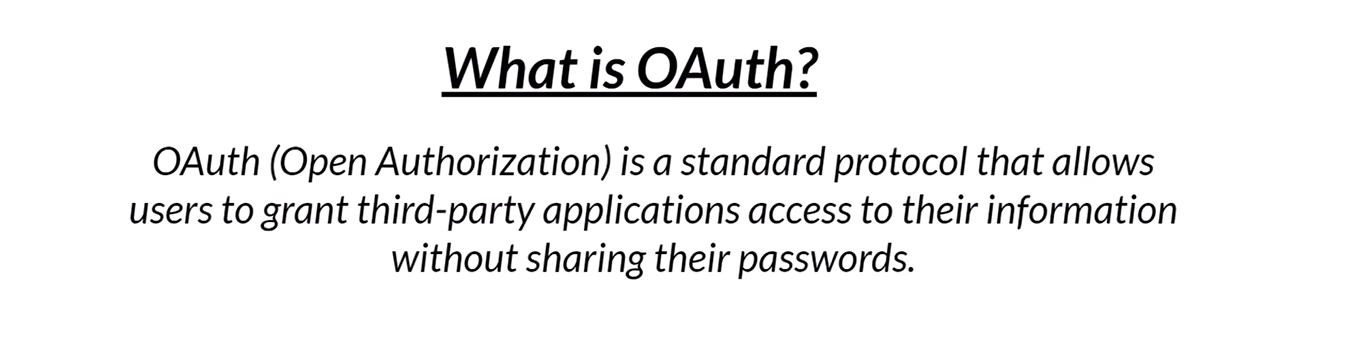
Before OAuth2 how things are working?

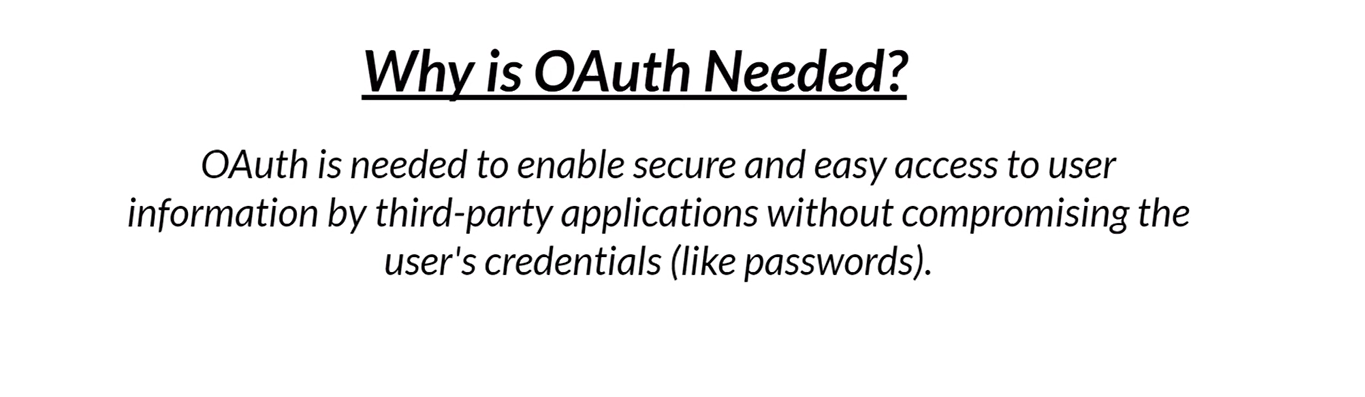
----> security risk : yedavina 3rd party application ni mana application lo use cheyali ante mana Credentials anevi share cheyali edi majore swcurity reason

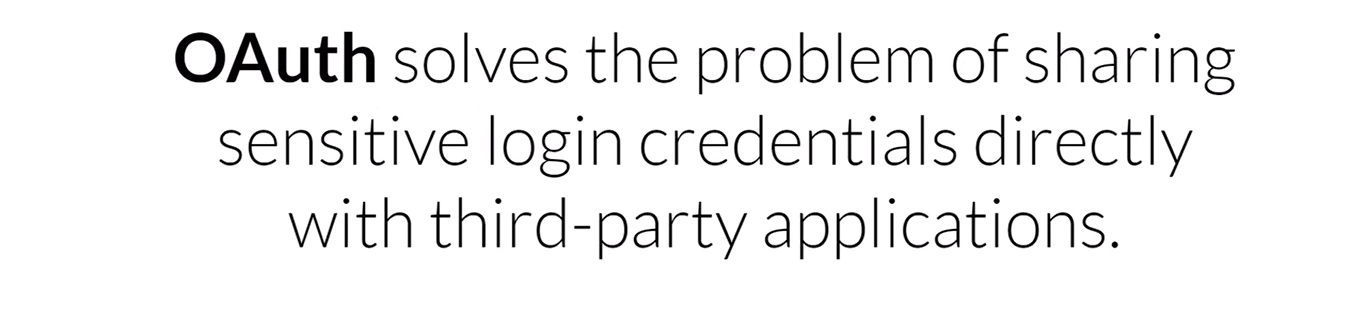
-----> Limited Control : Oka sari credentials share chesaka a 3rd party applications ki All access anevi vellevi, LImited Acess ni control anedi vundedi kadu

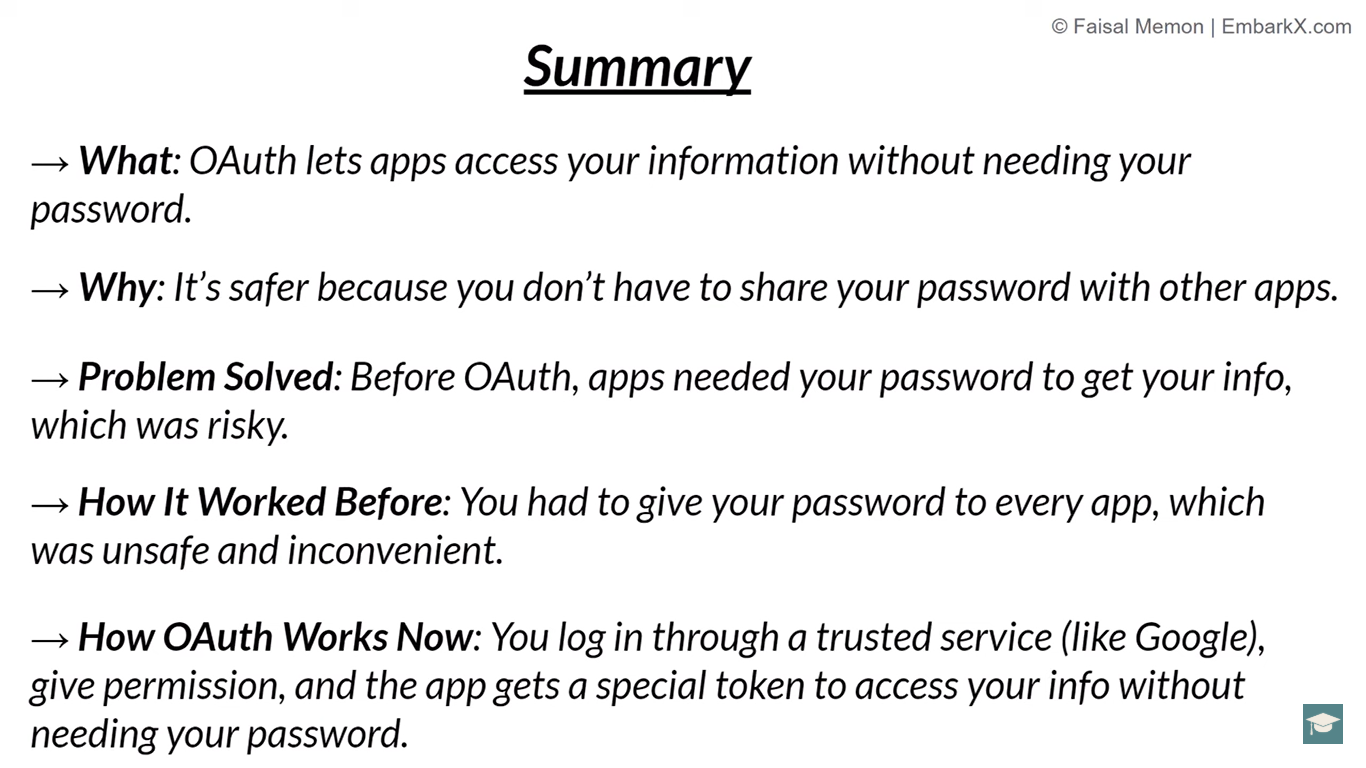
-------> Inconvenience : manam yedavian multiple 3rd party applications lo login avvi vundi yedavina oka danni nuodi access thiseyali anna leda vati access anedi pokunda password mathrame change cheyali anukunte kudaradu password chestahe login avvina applications ki access anedi pothundi

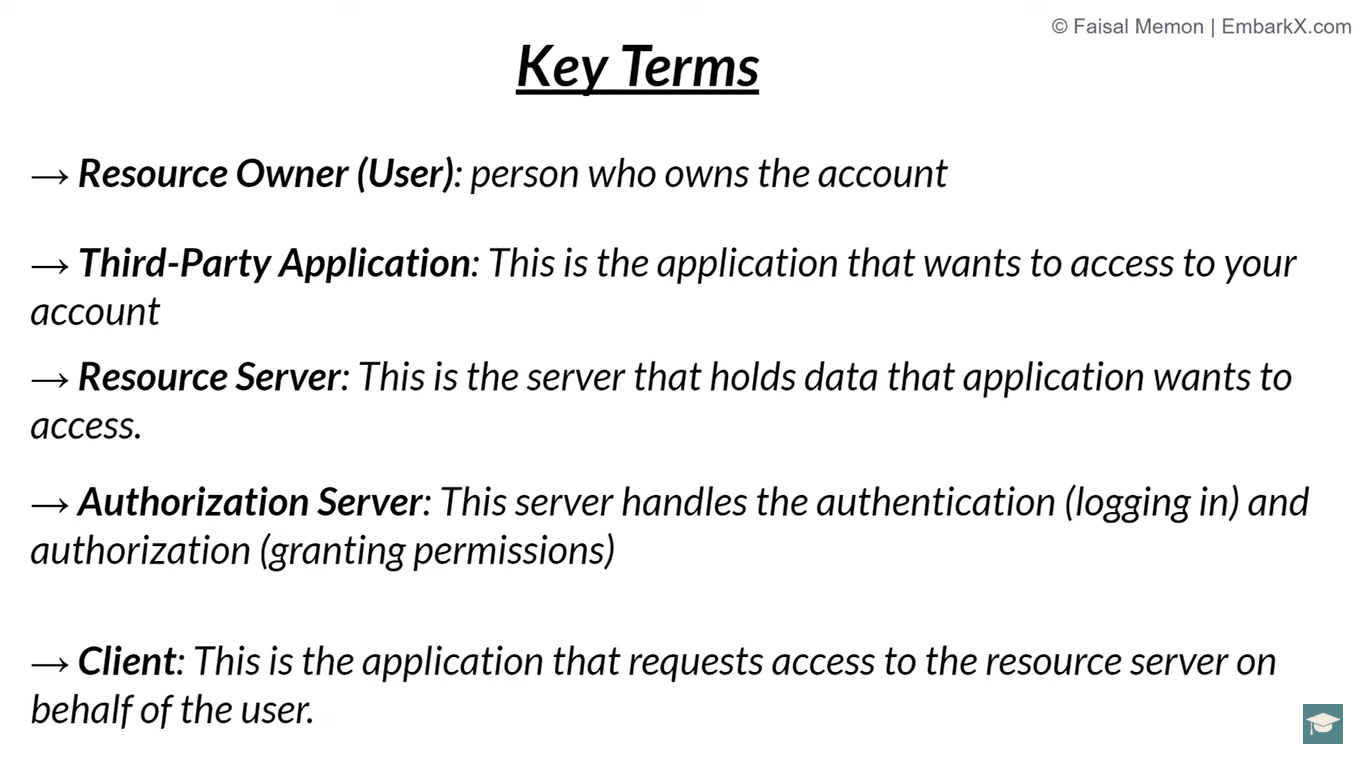
E prbs anevi OAuth2 lekamundu face chesevallam

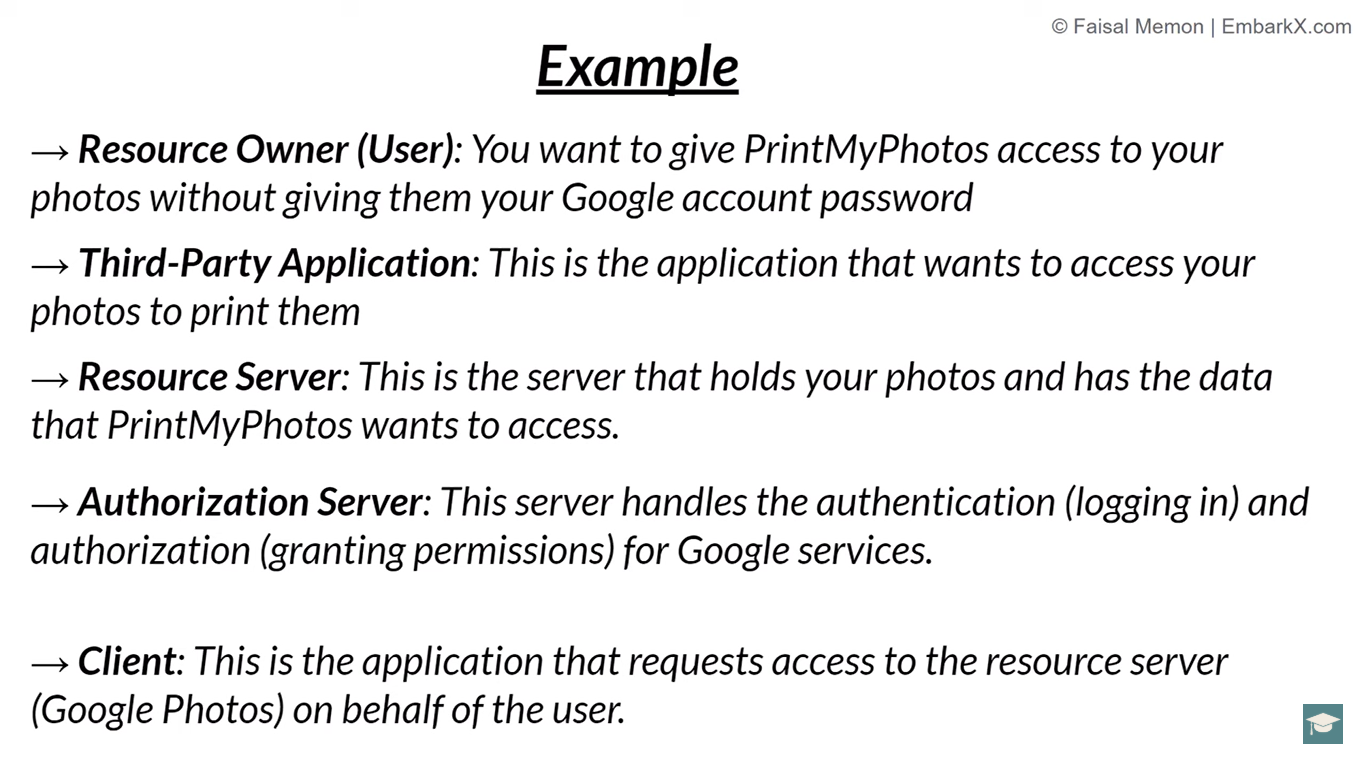












=============== doubts =============

How to create OAuth2 login setup ?

SecurityConfig class

@Configuration

@EnableWebSecurity

public class SecurityConfig {

public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {

http.authorizeHttpRequests(auth->auth.anyRequest().authenticated())

.oauth2Login(Customizer.withDefaults());

return http.build();

}

}

Controller Class

@RestController

//@RequestMapping("/Demo")

public class HelloController {

@GetMapping("/hello")

public String greet(){

return "Welcome to Telusko";

}

}

application.properties

spring.application.name=OAuth2\_Spring\_Demo

spring.datasource.url=jdbc:mysql://localhost:3306/spring\_security

spring.datasource.username=root

spring.datasource.password=Root@123

spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.jpa.database-platform=org.hibernate.dialect.MySQLDialect

spring.jpa.hibernate.ddl-auto=update

#GitHub Oauth2 credentials

spring.security.oauth2.client.registration.github.client-id=Ov23liMjRvArJ6HH6AAU

spring.security.oauth2.client.registration.github.client-secret=ab7ea344068593aa6a6177f8b8bc5a9194ec3bda

Eppudu ante Na Git Hub Account nuodi Git client-id & .client-secret ni genarate chesanu but real time project lo yela create chesthamu?

**Real-time lo OAuth2 Setup Ela Chestham?**

**Mee local lo** GitHub lo register chesi client-id & client-secret ni use chesaru kada? **Real-time lo same process untundi, but organization level lo chestham.**

#### ****Step 1: Organization Level lo GitHub OAuth App Create Cheyyadam****

**Company/Organization** ki **GitHub Enterprise Account** untundi. Ledu ante, regular GitHub account use chestharu.

**GitHub Developer Settings** ki vellali:

* 1. Open **GitHub**
  2. Go to **Settings** (profile icon click cheyyi -> "Settings")
  3. Left-side menu lo **Developer settings** click cheyyi
  4. **OAuth Apps** click cheyyi
  5. **New OAuth App** click cheyyi

**Application details enter cheyyali**

* 1. **Application Name:** Example – Company App
  2. **Homepage URL:** https://yourcompany.com
  3. **Authorization callback URL:** https://yourcompany.com/login/oauth2/code/github  
     (Spring Boot OAuth2 uses /login/oauth2/code/{provider} by default)
  4. **Register Application** click cheyyali

**GitHub miku Client ID & Client Secret istundi.**

* 1. **Client ID** – Direct ga kanipistadi
  2. **Client Secret** – Generate a new client secret click cheyyali, danini secure ga store cheyyali

#### ****Step 2: Ee Credentials ni Spring Boot lo Add Cheyyadam****

**Real-time projects lo secrets direct ga** application.properties **lo rayam. Instead, environment variables or secret vaults lo store chestham.**

##### ****Option 1: application.properties (Development/Testing purpose)****

spring.security.oauth2.client.registration.github.client-id=your-client-id

spring.security.oauth2.client.registration.github.client-secret=your-client-secret

##### ****Option 2: Environment Variables (Production best practice)****

Terminal lo:

export GITHUB\_CLIENT\_ID=your-client-id

export GITHUB\_CLIENT\_SECRET=your-client-secret

Spring Boot lo:

spring.security.oauth2.client.registration.github.client-id=${GITHUB\_CLIENT\_ID}

spring.security.oauth2.client.registration.github.client-secret=${GITHUB\_CLIENT\_SECRET}

### ****FAQs (Mee Doubts ki Samadhanam 😁)****

#### ❓ ****Real-time lo developers ki client-id & secret ela dorukuthai?****

➡️ Usually, **Admin** (DevOps team) create chestharu. Developers ki direct ga **properties lo access ivvakunda**, **Environment variables** lo store chestharu.

#### ❓ ****Company lo different environments ki (dev, test, prod) separate client-id & secret untaya?****

➡️ **Yes**, different environments ki different OAuth apps create chestharu.

* **Development OAuth App** (Local testing)
* **Staging OAuth App** (Pre-production testing)
* **Production OAuth App** (Live users)

#### ❓ ****Production lo client-secret security ela maintain chestham?****

➡️ **AWS Secrets Manager, Azure Key Vault, HashiCorp Vault** laanti secret management tools use chestharu.  
➡️ Direct ga .properties file lo petadam **security risk**.

#### ❓ ****Only GitHub OAuth2 kakunda, Google, Facebook kuda ala integrate chesthaama?****

➡️ **Yes!** Same process follow cheyyali:

* **Google OAuth2 App** – [Google Cloud Console](https://console.cloud.google.com/)
* **Facebook OAuth2 App** – [Meta Developer Console](https://developers.facebook.com/)
* **GitHub OAuth2 App** – [GitHub Developer Settings](https://github.com/settings/developers)

### ****Conclusion****

✅ **Mee local lo cheyyadam & real-time process almost same**, but real-time lo **Admin/DevOps teams maintain chestharu**.  
✅ **Environment variables or secret vaults use cheyyali, direct** .properties **lo client secrets store cheyyakudadu**.  
✅ **Different environments ki separate OAuth Apps create chestharu**.  
✅ **Same process Google, Facebook ki apply avuthundi**.

so realtime lo kuda eppudu nenu yela avithe na account tho create chesanoo...alane company account tho create chestahru right ?

**Yes, exactly! 🎯**  
Real-time lo **company GitHub/Google/Facebook account use chesi** OAuth2 app create chestharu.

🔹 **Nuvvu personal GitHub account tho create chesav kada?**  
➡️ **Company lo admin (DevOps team)** company account tho **GitHub Developer Settings** lo OAuth2 app create chestharu.

🔹 **Client ID & Client Secret kuda alane generate chestharu**  
➡️ But **production lo direct** .properties **lo petaru**, instead, **secret vaults** (AWS Secrets Manager, Azure Key Vault) lo store chestharu.

🔹 **Real-time lo different environments ki separate OAuth apps untai**  
➡️ Dev, Test, Production ki different **Client ID & Secret** untai.

**So, overall process same, just company account lo create chestharu!** 🚀

Project for OAuth2 with Git Login users need to save in DB

Yes, **absolutely right! ✅**

### 💡 This whole setup will handle ****both types of users**** in a single project:

### 🔐 ****1. Existing DB Users (Username + Password Login)****

* These users already exist in your users and authorities tables.
* They can log in via **username/password + JWT token generation**.
* Their role-based access control works dynamically via the DB.

### 🌐 ****2. GitHub OAuth2 Login Users****

* These users log in via GitHub using:  
  http://localhost:8080/oauth2/authorization/github
* On first login:
  + They get stored in users table with:
    - username = GitHub email or login
    - password = 'N/A' (or blank)
    - enabled = true
  + They are also inserted into authorities table with default GIT\_USER role.
* A JWT token is generated for them and sent back.
* They can use that token in **Postman or frontend** to call secured endpoints.

### ✅ Summary:

| **User Type** | **Auth Method** | **DB Storage** | **JWT Role** |
| --- | --- | --- | --- |
| Normal Users | Username/Password | Already in DB | From DB |
| GitHub Users | OAuth2 GitHub Login | Saved on 1st login | GIT\_USER |

So yes, your project will support **hybrid login and role-based access** — clean and real-world production ready.

Entire project Code

✅ Final Setup Goals:

* Existing DB users (from users & authorities) login via /JWT/signin and get JWT.
* GitHub users login via /oauth2/authorization/github, on first login their data is saved in users and authorities with default role GIT\_USER, and JWT is generated.
* No extra tables or entities required.
* Inline JWT generation after GitHub login (No extra handler class).
* Dynamic role-based authorization from DB via JdbcUserDetailsManager.

### 🔐 1. SecurityConfig\_mysql.java

@Configuration

@EnableWebSecurity

@EnableMethodSecurity

public class SecurityConfig\_mysql {

@Autowired

private AuthEntryPointJwt\_mysql unauthorizedHandler;

@Autowired

private JwtUtils\_mysql jwtUtils;

@Autowired

private DataSource dataSource;

@Bean

public AuthTokenFilter\_mysql authenticationJwtTokenFilter() {

return new AuthTokenFilter\_mysql();

}

@Bean

public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {

http.csrf(AbstractHttpConfigurer::disable)

.headers(headers -> headers.frameOptions().sameOrigin())

.sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))

.exceptionHandling(exception -> exception.authenticationEntryPoint(unauthorizedHandler))

.authorizeHttpRequests(auth -> auth

.requestMatchers("/h2-console/\*\*", "/JWT/signin", "/JWT/signup", "/public/\*\*").permitAll()

.anyRequest().authenticated()

)

.oauth2Login(oauth -> oauth.successHandler((request, response, authentication) -> {

OAuth2AuthenticationToken oauthToken = (OAuth2AuthenticationToken) authentication;

OAuth2User oauthUser = oauthToken.getPrincipal();

String username = oauthUser.getAttribute("login");

// Save to DB if not exists

JdbcTemplate jdbcTemplate = new JdbcTemplate(dataSource);

Integer count = jdbcTemplate.queryForObject(

"SELECT COUNT(\*) FROM users WHERE username = ?", Integer.class, username);

if (count != null && count == 0) {

jdbcTemplate.update("INSERT INTO users (username, password, enabled) VALUES (?, ?, true)", username, "N/A");

jdbcTemplate.update("INSERT INTO authorities (username, authority) VALUES (?, ?)", username, "GIT\_USER");

}

// Generate JWT

String jwt = jwtUtils.generateTokenFromUsername(username);

response.setContentType("application/json");

response.getWriter().write("{\"token\": \"" + jwt + "\"}");

}));

http.addFilterBefore(authenticationJwtTokenFilter(), UsernamePasswordAuthenticationFilter.class);

return http.build();

}

@Bean

public UserDetailsService userDetailsService() {

return new JdbcUserDetailsManager(dataSource);

}

@Bean

public AuthenticationManager authenticationManager(AuthenticationConfiguration config) throws Exception {

return config.getAuthenticationManager();

}

@Bean

public PasswordEncoder passwordEncoder() {

return new BCryptPasswordEncoder();

}

}

### 🛡 2. AuthTokenFilter\_mysql.java

@Component

public class AuthTokenFilter\_mysql extends OncePerRequestFilter {

@Autowired

private JwtUtils\_mysql jwtUtilsMysql;

@Autowired

private UserDetailsService userDetailsService;

private static final Logger logger = LoggerFactory.getLogger(AuthTokenFilter\_mysql.class);

@Override

protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain)

throws ServletException, IOException {

try {

String jwt = parseJwt(request);

if (jwt != null && jwtUtilsMysql.validateToken(jwt)) {

String username = jwtUtilsMysql.getUsernameFromToken(jwt);

UserDetails userDetails = userDetailsService.loadUserByUsername(username);

UsernamePasswordAuthenticationToken authentication = new UsernamePasswordAuthenticationToken(

userDetails, null, userDetails.getAuthorities());

authentication.setDetails(new WebAuthenticationDetailsSource().buildDetails(request));

SecurityContextHolder.getContext().setAuthentication(authentication);

}

} catch (Exception e) {

logger.error("Cannot set user authentication: {}", e);

}

filterChain.doFilter(request, response);

}

private String parseJwt(HttpServletRequest request) {

return jwtUtilsMysql.getJWTFromHeader(request);

}

}

### 🔐 3. JwtUtils\_mysql.java

@Component

public class JwtUtils\_mysql {

private static final Logger logger = LoggerFactory.getLogger(JwtUtils\_mysql.class);

@Value("${spring.app.jwtsecret}")

private String jwtsecret;

@Value("${spring.app.jwtExpirationMs}")

private int jwtExpirationMs;

public String getJWTFromHeader(HttpServletRequest request) {

String bearerToken = request.getHeader("Authorization");

if (bearerToken != null && bearerToken.startsWith("Bearer ")) {

return bearerToken.substring(7);

}

return null;

}

public String generateTokenFromUsername(String username) {

return Jwts.builder()

.subject(username)

.issuedAt(new Date())

.expiration(new Date(System.currentTimeMillis() + jwtExpirationMs))

.signWith(key())

.compact();

}

public String getUsernameFromToken(String token) {

return Jwts.parser()

.verifyWith((SecretKey) key())

.build()

.parseSignedClaims(token)

.getBody()

.getSubject();

}

public boolean validateToken(String token) {

try {

Jwts.parser().verifyWith((SecretKey) key()).build().parseSignedClaims(token);

return true;

} catch (Exception e) {

logger.error("JWT validation error: {}", e.getMessage());

return false;

}

}

private Key key() {

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

return Keys.hmacShaKeyFor(keyBytes);

}

}

### 🚫 4. AuthEntryPointJwt\_mysql.java

@Component

public class AuthEntryPointJwt\_mysql implements AuthenticationEntryPoint {

private static final Logger logger = LoggerFactory.getLogger(AuthEntryPointJwt\_mysql.class);

@Override

public void commence(HttpServletRequest request, HttpServletResponse response, AuthenticationException authException)

throws IOException, ServletException {

logger.error("Unauthorized error: {}", authException.getMessage());

response.setContentType(MediaType.APPLICATION\_JSON\_VALUE);

response.setStatus(HttpServletResponse.SC\_UNAUTHORIZED);

final Map<String, Object> body = new HashMap<>();

body.put("status", HttpServletResponse.SC\_UNAUTHORIZED);

body.put("error", "Unauthorized");

body.put("message", authException.getMessage());

body.put("path", request.getServletPath());

final ObjectMapper mapper = new ObjectMapper();

mapper.writeValue(response.getOutputStream(), body);

}

}

Controller Class

@RestController

@RequestMapping("/JWT")

public class JWTController\_mysql {

@Autowired

private JwtUtils\_mysql jwtUtilsMysql;

@Autowired

private AuthenticationManager authenticationManager;

@Autowired

private PasswordEncoder passwordEncoder;

@Autowired

private DataSource dataSource;

@PostMapping("/signin")

public ResponseEntity<?> authenticateUser(@RequestBody LoginRequest\_mysql loginRequestMysql) {

Authentication authentication;

try {

authentication = authenticationManager.authenticate(

new UsernamePasswordAuthenticationToken(loginRequestMysql.getUsername(), loginRequestMysql.getPassword()));

} catch (Exception e) {

return ResponseEntity

.badRequest()

.body("Invalid username or password");

}

SecurityContextHolder.getContext().setAuthentication(authentication);

UserDetails userDetails = (UserDetails) authentication.getPrincipal();

String jwt = jwtUtilsMysql.generateTokenFromUsername(String.valueOf(userDetails));

List<String> roles = userDetails.getAuthorities().stream()

.map(item -> item.getAuthority())

.collect(Collectors.toList());

LoginResponces\_mysql loginResponcesMysql = new LoginResponces\_mysql(jwt, userDetails.getUsername(), roles);

return ResponseEntity.ok(loginResponcesMysql);

}

@PostMapping("/signup")

public ResponseEntity<?> registerUser(@RequestBody LoginRequest\_mysql loginRequestMysql) {

JdbcUserDetailsManager jdbcUserDetailsManager = new JdbcUserDetailsManager(dataSource);

// Create user with the provided username and password

String userRole = loginRequestMysql.getRole();

jdbcUserDetailsManager.createUser(org.springframework.security.core.userdetails.User.withUsername(loginRequestMysql.getUsername())

.password(passwordEncoder.encode(loginRequestMysql.getPassword()))

.roles(userRole) // Use user-provided role instead of hardcoding

.build());

return ResponseEntity.ok("User registered successfully!");

}

}

✅ Now you can test:

* Existing DB users: /JWT/signin (your normal login API)
* GitHub login: http://localhost:8080/oauth2/authorization/github — JWT will be shown in browser after first login and you can use it in Postman.

After running Above code for git users working fine but existing DB users JWT token genarated but while using that token and if hit URL in post man I am getting Error "message": "Full authentication is required to access this resource",?

Perfect! You’ve understood it almost 100% right — let’s now summarize everything clearly and provide you with:

✅ Full issue summary  
✅ Root cause & explanation  
✅ JWT decode examples for **wrong token (object passed)** and **correct token (username passed)**  
✅ Final solution & takeaway

### ✅ ****ISSUE SUMMARY****

You were generating JWT token in /signin endpoint using this line:

String jwt = jwtUtilsMysql.generateTokenFromUsername(String.valueOf(userDetails)); // ❌ WRONG

Instead of this:

String jwt = jwtUtilsMysql.generateTokenFromUsername(userDetails.getUsername()); // ✅ CORRECT

So even though the token was **technically getting generated**, it was generated with the **wrong subject (username)** in the JWT payload, resulting in **authentication failure later** (401 Unauthorized).

### 🔍 ****ROOT CAUSE****

* generateTokenFromUsername(String username) expects a **plain username** like "john", "admin" etc.
* But String.valueOf(userDetails) gives you the **entire object’s** toString() **value**, like:  
  "org.springframework.security.core.userdetails.User@6d3f80ef"

So JWT was generated **with this object string as subject (sub)** instead of the real username.

Later, when request comes with that JWT:

String username = jwtUtilsMysql.getUsernameFromJwtToken(token);

UserDetails user = userDetailsService.loadUserByUsername(username);

This fails because that "org.springframework...User@xyz" is not a valid username in DB.

### ✳️ ****JWT Breakdown: Header, Payload, Signature****

A JWT token has 3 parts:

[HEADER].[PAYLOAD].[SIGNATURE]

1. **Header (Base64-encoded JSON)**:

{

"alg": "HS256",

"typ": "JWT"

}

1. **Payload (Base64-encoded JSON)**:  
   Contains user info like username (sub), issued time, expiry etc.

{

"sub": "john123", // username

"iat": 1710922000, // issued at

"exp": 1710925600 // expiry

}

1. **Signature** is generated using your secret key.

### 📌 ****Example 1: WRONG TOKEN (object passed)****

String jwt = jwtUtilsMysql.generateTokenFromUsername(String.valueOf(userDetails));

#### ➤ JWT Payload after decode:

{

"sub": "org.springframework.security.core.userdetails.User@6d3f80ef",

"iat": 1710922100,

"exp": 1710925700

}

* So getUsernameFromJwtToken(token) returns "org.springframework.security.core.userdetails.User@6d3f80ef" → Not found in DB → 401 Unauthorized

### 📌 ****Example 2: CORRECT TOKEN (username passed)****

String jwt = jwtUtilsMysql.generateTokenFromUsername(userDetails.getUsername());

#### ➤ JWT Payload after decode:

{

"sub": "john123",

"iat": 1710922200,

"exp": 1710925800

}

* Now getUsernameFromJwtToken(token) returns "john123" → Found in DB → Auth success ✅

### 💡 ****Solution Recap (Fix):****

🔸 **Always pass plain username** to generateTokenFromUsername():

String jwt = jwtUtilsMysql.generateTokenFromUsername(userDetails.getUsername());

🔸 Avoid using String.valueOf(userDetails) or userDetails.toString() — it's not the actual username, just a string representation of the object.

### 🧠 Final Takeaway:

| **Parameter Passed** | **JWT Payload sub** | **Auth Works?** |
| --- | --- | --- |
| String.valueOf(userDetails) | "org.springframework.security.core.userdetails.User@xyz" | ❌ No |
| userDetails.getUsername() | "john123" | ✅ Yes |

### 🔍 ****Optional Tip**** – Decode JWT tokens online:

Use [https://jwt.io](https://jwt.io/)   
Paste token → See payload → Quickly debug issues like this.

Conclustion : Manam “JwtUtils\_mysql “ lo rasian “generateTokenFromUsername” method anedi token ni genarate chesthundi

Controller class lo (“/signin) end point lo generateTokenFromUsername method ki String.valueOf(userDetails) eila pass cheyattam valana token anedi Genarate avvindi but entire invalid username tho Token genarate avvindi ala kakunda object ni just userDetails ni pass chesina work avvedi.

Ala avvatam vala token ni validate chesetappudu Usename header anedi a Token lo ledu andukani Error vacchindi

JWT Token decode chesthe header lo username correct ga vasthe validation gariddi.  
{

"sub": "kiran",

"iat": 1742413267,

"exp": 1742413367

}

Ala correct ga ravali ante “generateTokenFromUsername” method ki token genartion appudu only username ye pass cheyali

Conclusion of OAuth2 topic :

OAuth2 anedi users ki Hybrid mode lo login option evvatam( ex: Git,Google,Facebook…etc)

Diniki OAuth2 option evvali aukunna Apps lo Clien-id , Scerite - key ni register avvi mana application.poperties file lo evvali

Real time project lo Devoops team project related Account nuodi vatini eistharu avi manam use chesukovali

Best practies Oauth2 option dwara login avve users ki related configuration ni separate class lo rayatam and ala logina avina users detatils ni DB lo store chesi vallaki defaut ROLE evvatam best approch

Ala DB store cheyakunda and ROLE create cheyakunda kuda cheyachhu….

And einka chala validation cheyachhu… but as of now eve cover chesam