Day 9 Project Uber

SecurityConfig.java

```java
package com.yasif.project.uber.Uber.backend.system.configs;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManager;
import org.springframework.security.config.annotation.authentication.configuration.AuthenticationConfiguration;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;

@Configuration
public class SecurityConfig {

    @Bean
    PasswordEncoder passwordEncoder(){
        return new BCryptPasswordEncoder();
    }

    @Bean
    AuthenticationManager authenticationManager(@NotNull AuthenticationConfiguration configuration)
            throws Exception
    {
        return configuration.getAuthenticationManager();
    }

}
```

SecurityConfig

1. @Configuration

This annotation positions the class as a **centralized configuration unit**.

- It signals Spring that this class contains **bean definitions**
- Spring will process it at startup and register the declared beans into the **Application Context**
- Think of it as a **security enablement blueprint** rather than business logic

In short, this class exists to **wire security infrastructure**, not to execute workflows.

2. PasswordEncoder Bean

```
@Bean
PasswordEncoder passwordEncoder(){
   return new BCryptPasswordEncoder();
}
```

What this achieves:

- Establishes **BCrypt** as the standard password hashing algorithm across the system
- Ensures passwords are **never stored or compared in plain text**
- Enables Spring Security to automatically:
- Hash passwords during registration
- Validate hashed passwords during login

Why BCrypt is a best-practice choice:

- It is **one-way and irreversible**
- Includes built-in **salting**
- Designed to be **computationally expensive**, which mitigates brute-force and rainbow-table attacks
- Widely adopted and battle-tested in enterprise systems

From a governance perspective, this ensures **compliance-grade credential security** with minimal configuration overhead.

3. AuthenticationManager Bean

```
@Bean
AuthenticationManager authenticationManager(
      AuthenticationConfiguration configuration) throws Exception {
   return configuration.getAuthenticationManager();
}
```

What this does:

- Exposes the **AuthenticationManager** as a Spring Bean
- This is the **core engine** responsible for validating user credentials
- It orchestrates:
- UserDetailsService (to load user data)
- PasswordEncoder (to compare passwords)
- Authentication providers (DAO, JWT, etc.)

Why this bean is important:

- Allows authentication to be triggered **programmatically**
- For example, inside a login API or JWT authentication flow
- Without this bean, manual authentication (common in REST APIs) becomes cumbersome

This design cleanly decouples **authentication logic** from **controllers and services**, improving maintainability and testability.

4. How everything works together (end-to-end flow)

At runtime, the system behaves like this:

1. A login request comes in
2. AuthenticationManager is invoked
3. It fetches user details via UserDetailsService
4. It compares the incoming password using BCryptPasswordEncoder
5. If validation passes, authentication succeeds
6. The security context is populated for downstream access control

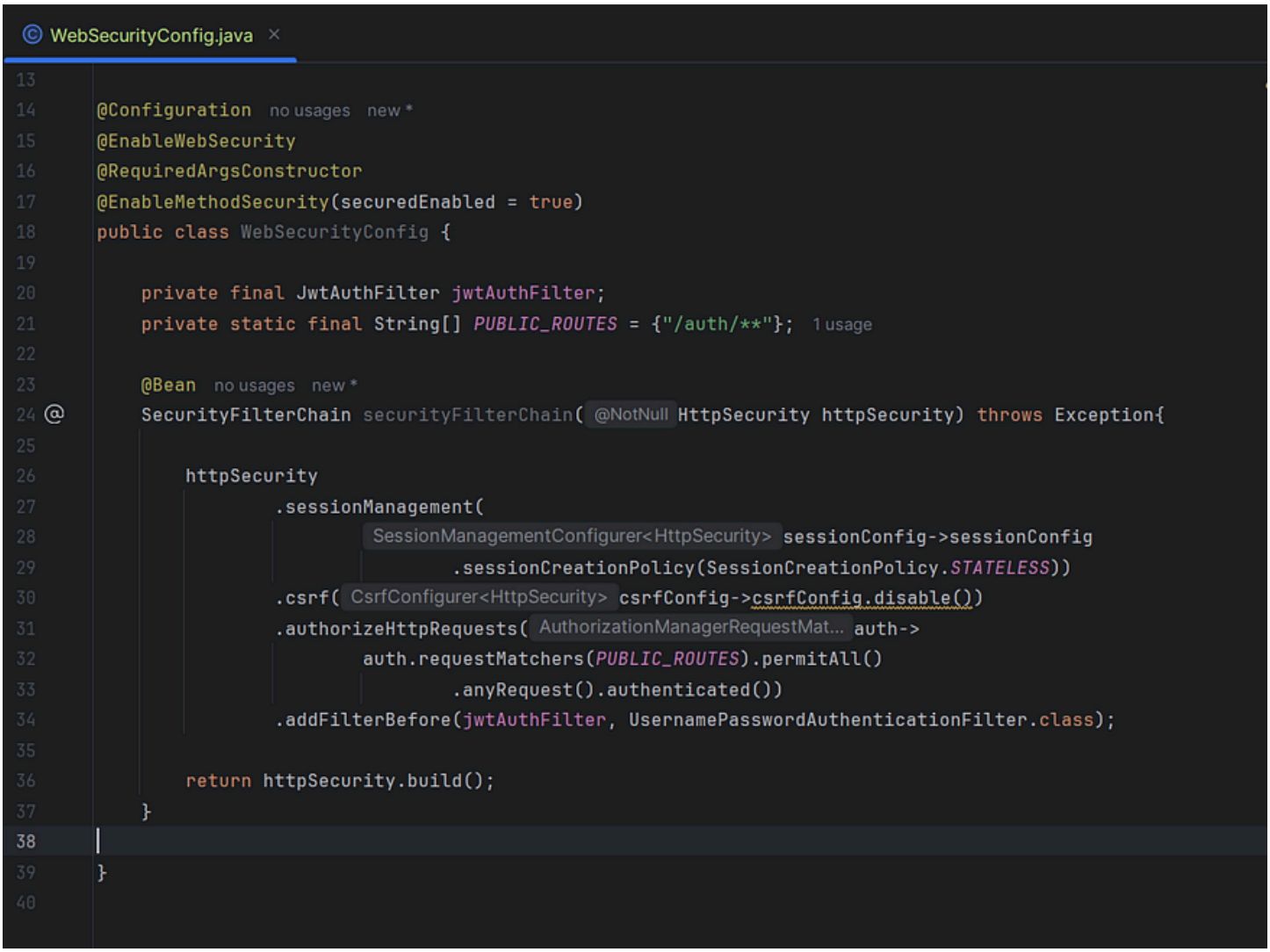
This setup provides a **plug-and-play authentication backbone** that scales well as more security layers (JWT, MFA, RBAC) are introduced.

Executive Summary

- This configuration establishes **secure password management**
- It exposes the **authentication engine** needed for login flows
- It follows **Spring Security best practices**
- It creates a future-ready foundation for advanced security features

In essence, this class is not doing anything flashy — but it is **strategically critical**. It ensures that authentication is secure, standardized, and extensible as the application matures.

Press enter or click to view image in full size



WebSecurityConfig

At an architectural level, this class is defining the **end-to-end web security posture** of the application. It controls *how requests are authenticated, which endpoints are exposed, and how Spring Security enforces access decisions* — all aligned with a stateless, token-driven strategy.

1. Class-level intent

```
@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
@EnableMethodSecurity(securedEnabled = true)
public class WebSecurityConfig {
```

This combination establishes the class as the **single source of truth for HTTP security**.

- **@Configuration**
  Declares this as a configuration module loaded at startup.
- **@EnableWebSecurity**
  Activates Spring Security's web layer and allows full customization of request security.
- **@RequiredArgsConstructor**
  Lombok generates a constructor for final fields, ensuring:
- Immutable dependencies
- Clean constructor-based dependency injection
- **@EnableMethodSecurity(securedEnabled = true)**
  Enables method-level access control such as:
- @Secured("ROLE_ADMIN")
- Fine-grained authorization inside service layers

  This means authorization is enforced **both at the API boundary and inside the business layer**, reducing risk exposure.

2. Dependency injection: JwtAuthFilter

```
private final JwtAuthFilter jwtAuthFilter;
```

This filter is the **core JWT enforcement mechanism**.

Strategically, it is responsible for:

- Extracting JWT tokens from incoming requests
- Validating token integrity and expiry
- Setting the authenticated user into the Security Context

  By injecting it here, the system guarantees that **every request is evaluated for token validity before reaching controllers**.

3. Public route definition

```
private static final String[] PUBLIC_ROUTES = {"/auth/**"};
```

This creates a **clear contract for open endpoints**.

- All authentication-related APIs (login, register, refresh token) are publicly accessible
- Every other endpoint defaults to secured access

This white-list approach is safer and easier to govern than selectively blocking endpoints.

4. SecurityFilterChain bean

```
@Bean
SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
```

This method defines the **entire HTTP security pipeline**.

From a systems perspective, this is where:

- Session strategy
- CSRF policy
- Authorization rules
- Custom filters
  are all stitched together.

5. Stateless session management

```
httpSecurity
  .sessionManagement(sessionConfig ->
    sessionConfig.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
```

This explicitly tells Spring Security:

- Do **not** create HTTP sessions
- Do **not** store authentication on the server

This aligns perfectly with **JWT-based authentication**, where:

- Each request is self-contained
- Scalability is improved
- Horizontal scaling becomes trivial

6. CSRF disabled

```
.csrf(csrfConfig -> csrfConfig.disable())
```

This is a **deliberate and correct choice** for stateless REST APIs.

- CSRF protection is mainly required for session-based browser apps
- With JWT passed in headers, CSRF attacks are not applicable
- Disabling it avoids unnecessary complexity

This decision is aligned with REST security best practices.

7. Authorization rules

```
.authorizeHttpRequests(auth ->
  auth.requestMatchers(PUBLIC_ROUTES).permitAll()
    .anyRequest().authenticated())
```

This defines a **clear access control strategy**:

- /auth/** → open access
- All other endpoints → require authentication

Operationally, this creates:

- Predictable security behavior
- Minimal risk of accidentally exposing sensitive APIs
- A default-deny posture, which is enterprise-grade security hygiene

8. JWT filter placement

```
.addFilterBefore(jwtAuthFilter, UsernamePasswordAuthenticationFilter.class);
```

This line is critical.

It ensures:

- JWT validation runs **before** Spring's default authentication logic
- Requests are authenticated via token **without using username/password each time**
- The Security Context is populated early in the filter chain

This sequencing guarantees that downstream authorization checks work correctly.

9. Final build

```
return httpSecurity.build();
```

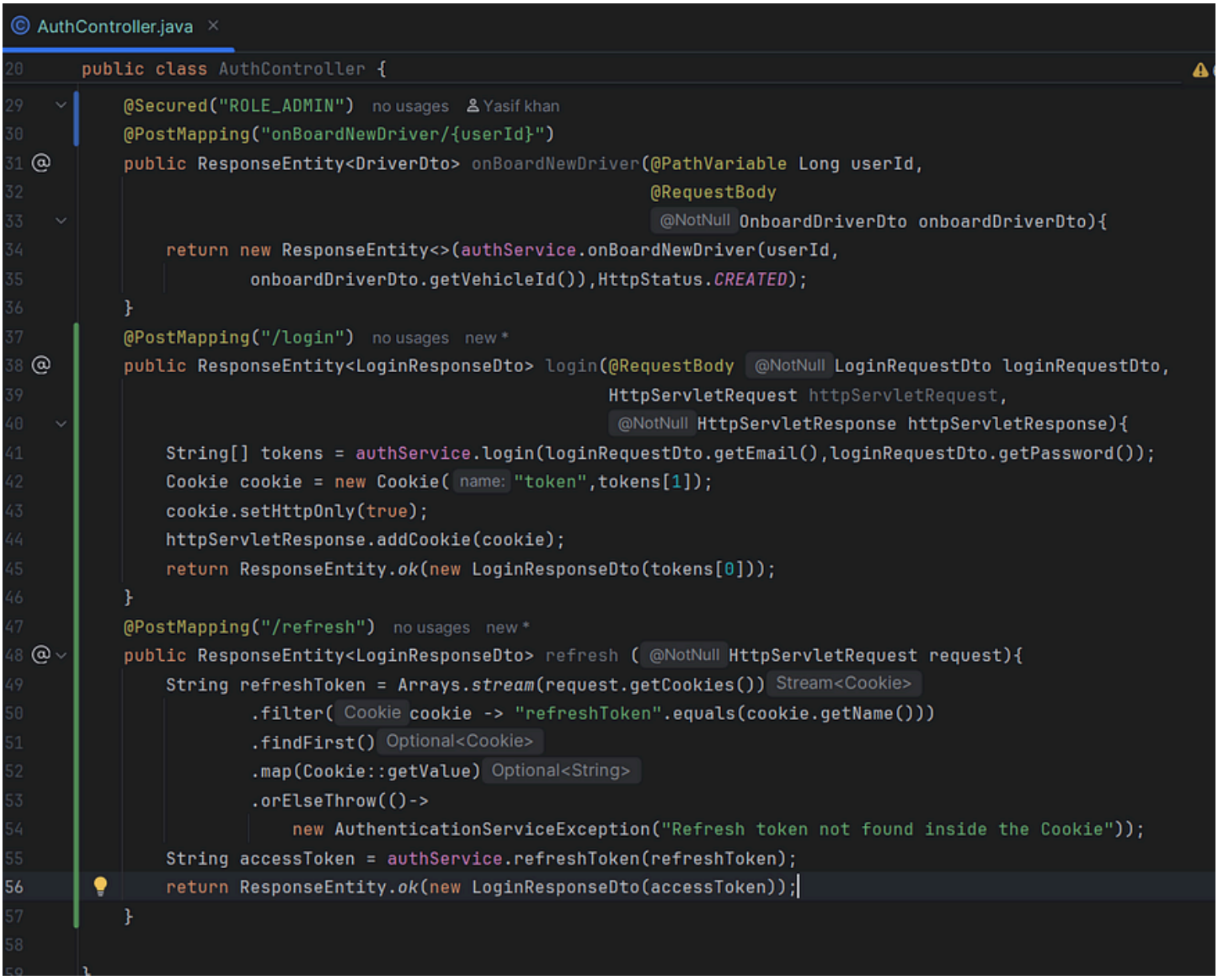
This compiles all configurations into a **production-ready security filter chain** that Spring Security enforces at runtime.

Executive-level summary

- Establishes **stateless JWT-based security**
- Cleanly separates **public vs secured endpoints**
- Enforces authentication at both **HTTP and method levels**
- Optimized for **scalability, maintainability, and enterprise readiness**

In short, this configuration positions the application with a **modern, cloud-native security model** that is robust today and extensible for future enhancements like RBAC, MFA, or API rate limiting.

Press enter or click to view image in full size



AuthController

From an enterprise security and API governance perspective, this AuthController is the **external-facing contract for authentication and**

driver onboarding operations. It enforces role-based access, manages token lifecycle, and ensures secure communication with clients.

1. Onboarding a new driver

```
@Secured("ROLE_ADMIN")
@PostMapping("onBoardNewDriver/{userId}")
public ResponseEntity<DriverDto> onBoardNewDriver(
    @PathVariable Long userId,
    @RequestBody OnboardDriverDto onboardDriverDto
) {
  return new ResponseEntity<>(
      authService.onBoardNewDriver(userId, onboardDriverDto.getVehicleId()),
      HttpStatus.CREATED
  );
}
```

Strategic intent

- **@Secured("ROLE_ADMIN")**
  Ensures that only admin users can call this endpoint. Unauthorized access is blocked at the method level.
- **@PostMapping("onBoardNewDriver/{userId}")**
  Binds this operation to an HTTP POST request and uses a path variable for the target user ID.
- **Flow:**

1. Admin calls API with a userId and vehicle information.
2. Delegates onboarding to the AuthService.
3. Returns the created driver as a DriverDto with **HTTP 201 CREATED**.

Business outcome: Only authorized admins can escalate a user to a driver and assign vehicles, maintaining operational security.

2. User login

```
@PostMapping("/login")
public ResponseEntity<LoginResponseDto> login(
    @RequestBody LoginRequestDto loginRequestDto,
    HttpServletRequest httpServletRequest,
    HttpServletResponse httpServletResponse
) {
```

Execution flow

1. Authenticates the user via authService.login(email, password).
2. Receives an **access token** and **refresh token**.
3. Stores the refresh token in an **HTTP-only cookie**, protecting it from JavaScript-based attacks (XSS).

```
Cookie cookie = new Cookie("refreshToken", tokens[1]);
cookie.setHttpOnly(true);
httpServletResponse.addCookie(cookie);
```

1. Returns the **access token** in the response body.

Security considerations

- **Separation of tokens:**
  Access token is short-lived and used in headers.
  Refresh token is long-lived but stored securely in an HTTP-only cookie.
- **Stateless authentication:**
  Backend doesn't need to track session state; JWT handles identity.
- **HTTP 200 response:**
  Provides a standard response for successful login.

3. Refreshing access tokens

```
@PostMapping("/refresh")
public ResponseEntity<LoginResponseDto> refresh(HttpServletRequest request) {
```

Execution flow

1. Extracts the **refresh token from cookies**:

```
String refreshToken = Arrays.stream(request.getCookies())
    .filter(cookie -> "refreshToken".equals(cookie.getName()))
    .findFirst()
    .map(Cookie::getValue)
    .orElseThrow(() -> new AuthenticationServiceException(
        "Refresh token not found inside the Cookie"
    ));
```

1. Delegates to authService.refreshToken(refreshToken) to generate a new access token.
2. Returns the new access token in the response body.

Business and security impact

- Supports **stateless session management** with JWT.
- Ensures that expired or missing refresh tokens immediately fail.
- Provides seamless token rotation without forcing users to re-authenticate frequently.
- Enhances security by not exposing refresh tokens in response bodies.

4. How it fits into the overall authentication flow

1. **Login:** User submits credentials → receives access token in body and refresh token in cookie.
2. **Subsequent requests:** Access token sent in Authorization header → validated via JwtAuthFilter.
3. **Token expiration:** When access token expires, client calls /refresh → receives a new access token without re-login.
4. **Admin operations:** Endpoints like onBoardNewDriver enforce RBAC via @Secured.

Executive summary

- Exposes **secure, stateless authentication endpoints** with token lifecycle management.
- Enforces **role-based access control** for critical operations.
- Leverages **HTTP-only cookies for refresh tokens** to reduce attack surface.
- Delegates business logic to service layer, maintaining **clean separation of concerns**.
- Supports enterprise-ready workflows for both riders and drivers while maintaining security and compliance.

Net result: AuthController serves as a **robust, secure, and maintainable gateway** for authentication, authorization, and onboarding, designed to scale safely in production environments.

```java
@Data  1 usage  new *
public class LoginRequestDto {

    private String email;
    private String password;

}
```

Press enter or click to view image in full size

```java
@Data  4 usages  new *
@NoArgsConstructor
@AllArgsConstructor
public class LoginResponseDto {

    private String accessToken;

}
```

Login Request,Response Dtos

From a design and delivery standpoint, these two DTOs form a **clean, well-bounded authentication contract** between the client and the backend. They intentionally separate **input credentials** from **output security artifacts**, which is exactly how a mature authentication flow should be modeled.

1. LoginRequestDto – inbound authentication payload

```
@Data
public class LoginRequestDto {
private String email;
   private String password;
}
```

Strategic role

This DTO represents **what the client must provide** to initiate authentication.

- email
  Acts as the **principal identifier** (username equivalent)
- password
  The secret used for credential validation

Why this abstraction matters

- Keeps controller method signatures clean
- Decouples API contracts from persistence entities
- Allows validation and evolution (e.g. add CAPTCHA, device info) without breaking controllers

Lombok @Data

This generates:

- Getters and setters
- toString() *(note: careful in logs with password fields)*
- equals() and hashCode()

From a governance angle, this is a **minimal, purpose-built DTO** — no excess fields, no leakage of internal structures.

2. LoginResponseDto – outbound authentication result

```
@Data
@NoArgsConstructor
@AllArgsConstructor
public class LoginResponseDto {
```

```
private String accessToken;
}
```

Strategic role

This DTO defines **what the system returns after successful authentication**.

- accessToken
  A JWT that:
- Represents authenticated identity
- Is sent by the client on every subsequent request
- Enables stateless authorization

Why only the token is returned

- Prevents accidental exposure of user data
- Keeps the response lightweight and secure
- Aligns with REST and JWT best practices

This design also makes it trivial to extend later with:

- refreshToken
- expiresIn
- tokenType

without disrupting existing consumers.

3. End-to-end login flow (business view)

At runtime, the interaction looks like this:

1. Client sends LoginRequestDto (email + password)
2. Backend authenticates using AuthenticationManager
3. On success, a JWT is generated
4. Backend responds with LoginResponseDto containing accessToken
5. Client stores the token and sends it in the Authorization header

This creates a **clear, auditable, and scalable authentication lifecycle**.

4. Key strengths of this approach

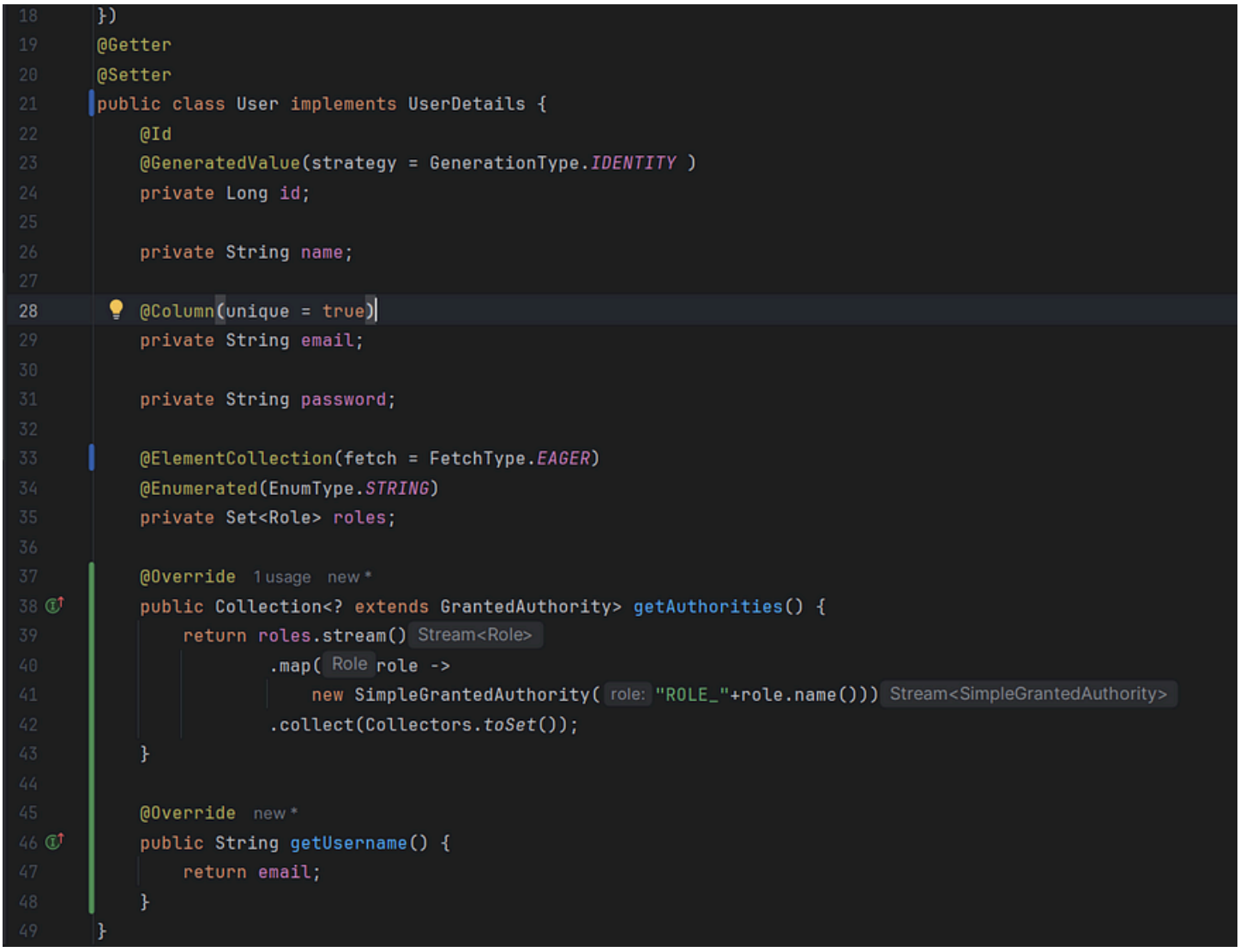
- Clear separation of concerns
- No entity leakage into API layer
- Stateless and cloud-ready
- Easy to extend for enterprise needs (MFA, RBAC, token rotation)

Executive summary

These DTOs may look simple, but they are **foundational to a secure authentication strategy**. They establish a well-defined contract, reduce coupling, and set the stage for future enhancements without architectural churn.

In short: **lean by design, secure by default, and future-proof by intent**.

Press enter or click to view image in full size



```
})
@Getter
@Setter
public class User implements UserDetails {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY )
    private Long id;

    private String name;

    @Column(unique = true)
    private String email;

    private String password;

    @ElementCollection(fetch = FetchType.EAGER)
    @Enumerated(EnumType.STRING)
    private Set<Role> roles;

    @Override 1 usage new *
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return roles.stream() Stream<Role>
                .map( Role role ->
                    new SimpleGrantedAuthority( role: "ROLE_"+role.name())) Stream<SimpleGrantedAuthority>
                .collect(Collectors.toSet());
    }

    @Override new *
    public String getUsername() {
        return email;
    }
}
```

User Entity

From a security architecture lens, this change elevates the **User entity into a first-class security principal**. By implementing UserDetails, the entity now integrates seamlessly with Spring Security's authentication and authorization pipeline, eliminating translation layers and tightening governance.

1. Implementing UserDetails in the User entity

By implementing UserDetails, the User entity becomes **directly consumable by Spring Security**.

What this unlocks operationally:

- Spring Security can use the entity **as-is** during authentication
- No separate adapter or wrapper class is required
- Authorization decisions are driven straight from persisted user data

This is a pragmatic, production-ready choice for most applications.

2. Role storage using @ElementCollection

```
@ElementCollection(fetch = FetchType.EAGER)
@Enumerated(EnumType.STRING)
private Set<Role> roles;
```

What this configuration achieves

- **@ElementCollection**
  Indicates that roles is a value collection (not a separate entity).
  Hibernate will create a **join table** linking users to roles.
- **fetch = FetchType.EAGER**
  Ensures roles are loaded immediately during authentication.
  This avoids lazy-loading failures when Spring Security evaluates authorities.
- **@Enumerated(EnumType.STRING)**
  Stores enum values as readable strings (e.g. ADMIN, USER) rather than ordinals.
  This prevents data corruption if enum ordering changes.

From a risk-management perspective, this setup prioritizes **correctness and predictability** over premature optimization.

3. Mapping roles to GrantedAuthority

```
@Override
public Collection<? extends GrantedAuthority> getAuthorities() {
  return roles.stream()
        .map(role ->
          new SimpleGrantedAuthority("ROLE_" + role.name()))
        .collect(Collectors.toSet());
}
```

Why this is critical

- Spring Security authorizes access based on **authorities**, not raw roles
- Prefixing with "ROLE_" is a Spring Security convention
- This enables seamless use of:
- @Secured("ROLE_ADMIN")
- @PreAuthorize("hasRole('ADMIN')")

This mapping ensures that **database roles cleanly translate into runtime access control** without ambiguity.

4. Overriding getUsername()

```
@Override
public String getUsername() {
  return email;
}
```

Strategic implication

- Declares email as the **principal identifier**
- Spring Security will authenticate users by email instead of a username
- Aligns with modern UX expectations and SaaS standards

This also keeps the authentication flow intuitive and business-aligned.

5. How this fits into the authentication flow

End-to-end execution looks like this:

1. UserDetailsService loads the User entity by email
2. Spring Security calls:
- getUsername()

- getPassword()
- getAuthorities()

3. Authorities are derived from persisted roles
4. Access decisions are enforced at API and method levels

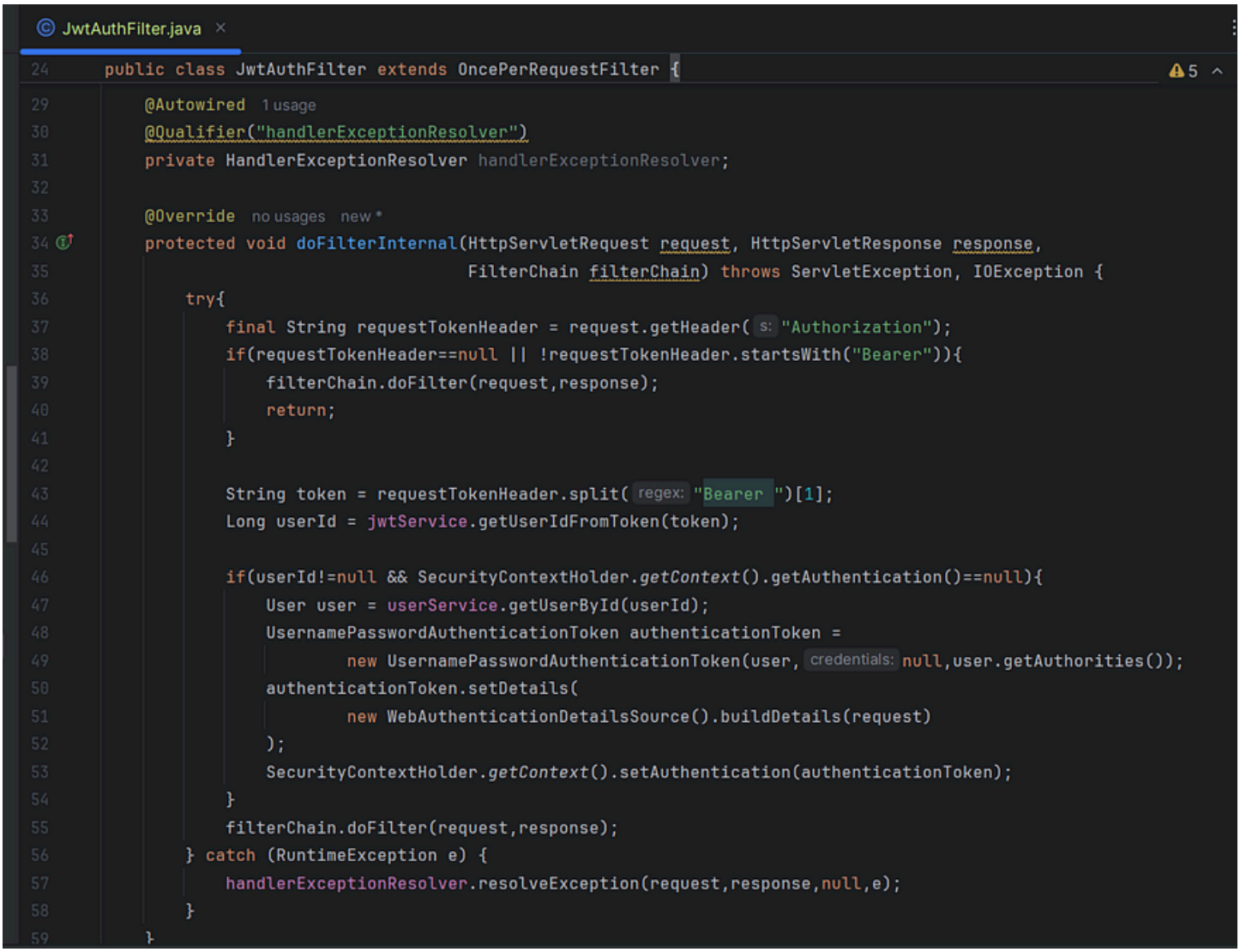
This delivers a **tight, deterministic security lifecycle** with minimal moving parts.

Executive summary

- User entity is now **security-aware**
- Roles are persisted safely and loaded reliably
- Authorization is standardized via ROLE_ conventions
- Email-based authentication is fully supported

Net impact: **reduced complexity, improved clarity, and enterprise-grade authorization alignment**. This setup is clean today and scales confidently as role models or access policies evolve.

Press enter or click to view image in full size



JwtAuthFilter

From a platform security perspective, this component is the **operational backbone of JWT-based authentication**. It ensures that every inbound request is evaluated for identity, validated, and contextually enriched *before* it reaches the business layer.

1. Class role and positioning

`@Service`
`@RequiredArgsConstructor`
public class JwtAuthFilter extends OncePerRequestFilter {

- **@Service**
  Registers this filter as a managed Spring component, making it injectable and lifecycle-aware.
- **@RequiredArgsConstructor**
  Enforces constructor-based dependency injection for required

collaborators, promoting immutability and testability.

- **OncePerRequestFilter**
  Guarantees the filter executes **exactly once per HTTP request**, eliminating duplicate authentication checks and inconsistent state.

  This positions the filter as a **reliable security gatekeeper** in the request pipeline.

2. Core dependencies

```
private final JWTService jwtService;
private final UserService userService;
```

These services split responsibilities cleanly:

- **JWTService**
  Owns token parsing, validation, and claim extraction (single responsibility)
- **UserService**
  Retrieves authoritative user data from persistence

  This separation reduces coupling and improves long-term maintainability.

3. Centralized exception handling strategy

```
@Autowired
@Qualifier("handlerExceptionResolver")
private HandlerExceptionResolver handlerExceptionResolver;
```

This is a **production-grade design decision**.

- Routes filter-level exceptions to the **global exception handling layer**
- Ensures consistent error responses across:
- Controllers
- Filters
- Interceptors

Without this, exceptions in filters often bypass @ControllerAdvice, leading to fragmented error handling.

4. Authorization header validation

```
final String requestTokenHeader = request.getHeader("Authorization");
if(requestTokenHeader == null || !requestTokenHeader.startsWith("Bearer")){
  filterChain.doFilter(request,response);
  return;
}
```

What this accomplishes

- Checks for the presence of a JWT
- Skips authentication logic for:
- Public endpoints
- Requests without tokens

This avoids unnecessary processing and keeps the filter lightweight.

5. Token extraction and identity resolution

```
String token = requestTokenHeader.split("Bearer ")[1];
Long userId = jwtService.getUserIdFromToken(token);
```

Operationally:

- Extracts the raw JWT
- Derives the **user identifier** from the token claims

This step treats the token as a **stateless identity carrier**, which is fundamental to scalable systems.

6. Authentication context population

```
if(userId != null && SecurityContextHolder.getContext().getAuthentication() == null){
  User user = userService.getUserById(userId);
```

This condition ensures:

- Authentication is established only once per request
- Existing security context is not overridden

Then:

```
UsernamePasswordAuthenticationToken authenticationToken =
  new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
```

- Principal → User
- Credentials → null (password is not re-validated)

- Authorities → derived from roles

  This is a **token-trust model**: the system trusts the JWT and only enriches the request with identity and permissions.

7. Request metadata binding

```
authenticationToken.setDetails(
   new WebAuthenticationDetailsSource().buildDetails(request)
);
```

This attaches contextual request information (IP, session metadata) to the authentication object, which can be valuable for:

- Auditing
- Logging
- Security analytics

8. Security context activation

```
SecurityContextHolder.getContext().setAuthentication(authenticationToken);
```

This is the decisive step:

- Marks the request as authenticated
- Enables downstream authorization checks (@PreAuthorize, @Secured)
- Makes the authenticated user accessible throughout the request lifecycle

9. Filter chain continuation

```
filterChain.doFilter(request,response);
```

This hands control to the next filter or controller, now with a fully populated security context.

10. Exception containment and resolution

```
} catch (RuntimeException e) {
   handlerExceptionResolver.resolveException(request,response,null,e);
}
```

This ensures:

- Token parsing or user lookup failures are handled gracefully

- Clients receive structured, predictable error responses
- The application avoids unhandled security exceptions

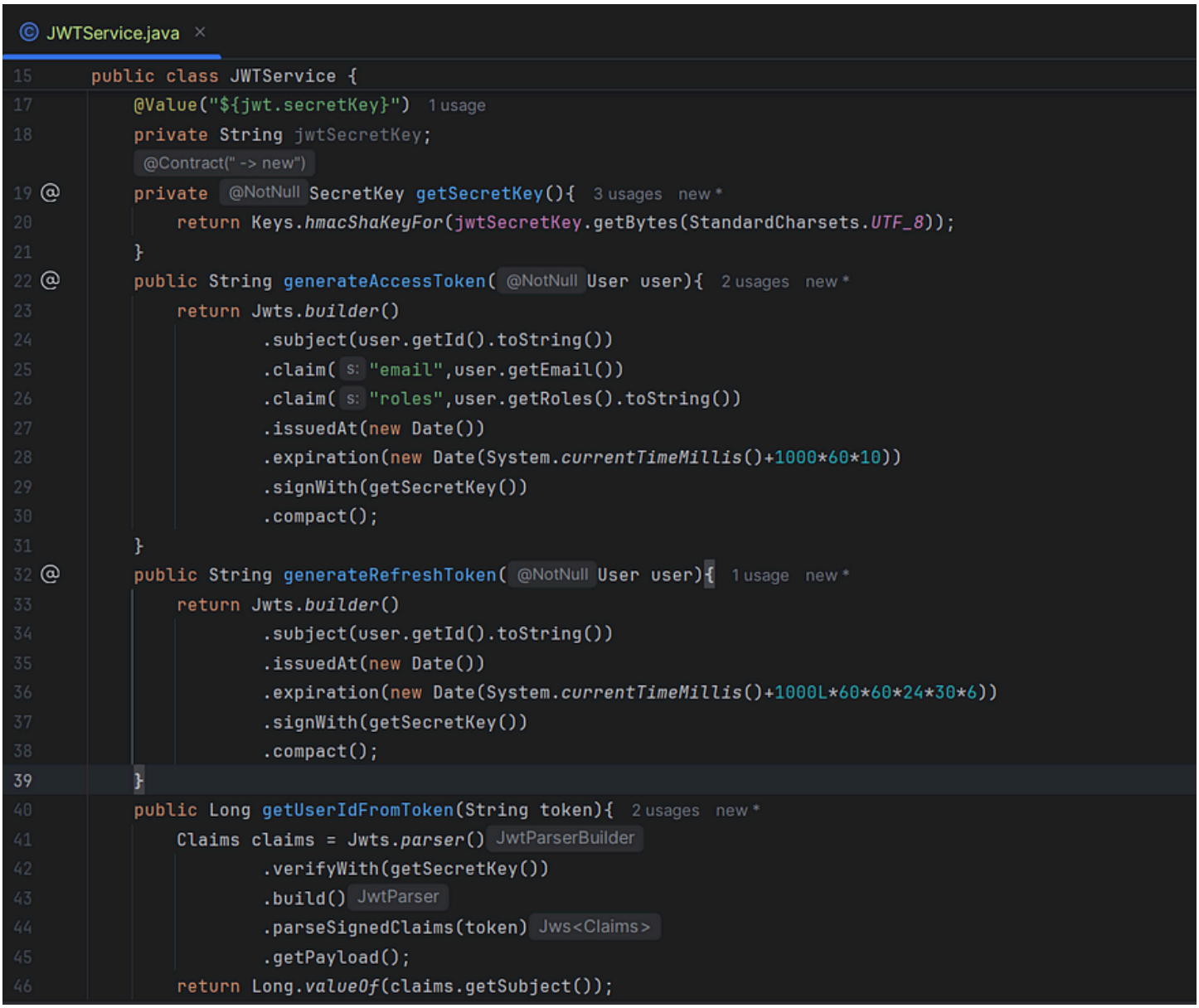
This is an **enterprise-grade resilience pattern**.

Executive summary

- Enforces JWT authentication at the request boundary
- Maintains stateless, scalable security
- Cleanly separates token logic from user retrieval
- Integrates seamlessly with global exception handling
- Enables consistent authorization across APIs

Net result: this filter acts as a **high-confidence security control point**, ensuring that every authenticated request enters the system with verified identity, enriched context, and enforceable permissions — ready for scale and future evolution.

Press enter or click to view image in full size



JWTService

From a security platform standpoint, this service is the **token lifecycle authority** of the application. It centralizes JWT creation and validation, ensuring identity, trust, and expiry are governed consistently across the system.

1. Service role and responsibility

`@Service`
`public class JWTService {`

This class is a **dedicated security utility service** responsible for:

- Generating access tokens
- Generating refresh tokens

- Parsing and validating incoming tokens
- Extracting trusted identity data

By isolating this logic, the application avoids token sprawl and maintains a **single source of truth** for JWT behavior.

2. Secret key injection

```
@Value("${jwt.secretKey}")
private String jwtSecretKey;
```

Strategic intent

- Externalizes the signing key to configuration
- Enables environment-specific secrets (dev, QA, prod)
- Avoids hard-coding sensitive material

From a governance perspective, this supports **secure secret rotation** and operational compliance.

3. Secret key construction

```
private SecretKey getSecretKey(){
  return Keys.hmacShaKeyFor(
    jwtSecretKey.getBytes(StandardCharsets.UTF_8)
  );
}
```

Why this matters

- Uses HMAC-SHA based signing (industry standard)
- Converts the configured secret into a cryptographic SecretKey
- Guarantees consistent signing and verification across all tokens

This is foundational to **token integrity and non-repudiation**.

4. Access token generation

```
public String generateAccessToken(User user){
  return Jwts.builder()
    .subject(user.getId().toString())
    .claim("email", user.getEmail())
    .claim("roles", user.getRoles().toString())
    .issuedAt(new Date())
```

```
        .expiration(new Date(System.currentTimeMillis() + 1000 * 60 * 10))
        .signWith(getSecretKey())
        .compact();
    }
```

Business and technical intent

- **Subject** → user ID
  Acts as the canonical identity reference
- **Custom claims** → email, roles
  Enables downstream authorization and auditing
- **IssuedAt & Expiration**
  Access token lifespan is intentionally short (10 minutes) to limit blast radius
- **Signature**
  Cryptographically seals the token to prevent tampering

This token is optimized for **frequent validation and short-term trust**.

5. Refresh token generation

```
public String generateRefreshToken(User user){
    return Jwts.builder()
        .subject(user.getId().toString())
        .issuedAt(new Date())
        .expiration(
            new Date(System.currentTimeMillis() + 1000L * 60 * 60 * 24 * 30 * 6)
        )
        .signWith(getSecretKey())
        .compact();
    }
```

Why this design works

- Contains **minimal data** (only subject)
- Has a **longer lifespan** (~6 months)
- Used exclusively to mint new access tokens

This separation ensures:

- Reduced exposure of sensitive claims
- Controlled re-authentication without re-entering credentials

A textbook **token rotation strategy**.

6. Token parsing and validation

```
public Long getUserIdFromToken(String token){
   Claims claims = Jwts.parser()
      .verifyWith(getSecretKey())
      .build()
      .parseSignedClaims(token)
      .getPayload();
   return Long.valueOf(claims.getSubject());
}
```

What happens here

- Verifies the token signature
- Validates token integrity and expiration
- Extracts trusted claims
- Converts subject back to a system-usable user ID

If validation fails, the parser throws an exception — effectively blocking unauthorized access.

This method is the **trust gateway** used by your JWT filter.

7. How this fits into the overall security flow

End-to-end execution:

1. User logs in successfully
2. Access + refresh tokens are generated
3. Client sends access token on every request
4. JwtAuthFilter validates the token via JWTService
5. User identity is reconstructed from token claims
6. Authorization is enforced downstream

This flow is **stateless, scalable, and cloud-native**.

Executive summary

- Centralizes JWT creation and verification
- Enforces short-lived access tokens with long-lived refresh tokens
- Keeps secrets externalized and rotatable

- Integrates cleanly with Spring Security filters
- Designed for scale, security, and future enhancements

Net impact: this service provides a **robust, enterprise-grade token strategy** that balances security risk with user experience and operational efficiency.

Press enter or click to view image in full size

```
    @Override  1 usage  new *
    public String refreshToken(String refreshToken) {

        // Extracts userId from refresh token.
        Long userId = jwtService.getUserIdFromToken(refreshToken);

        // Fetches user associated with the token.
        User user = userRepository.findById(userId).orElseThrow(
                () -> new ResourceNotFoundException(
                        "User not found with id:" + userId
                )
        );

        // Generates and returns a new access token.
        return jwtService.generateAccessToken(user);
    }
}
```

Press enter or click to view image in full size

```java
@Override  1 usage  new *
public String[] login(String email, String password) {

    // Authenticates user credentials using Spring Security.
    Authentication authentication = authenticationManager.authenticate(
            new UsernamePasswordAuthenticationToken(email, password)
    );

    // Retrieves the authenticated user principal.
    User user = (User) authentication.getPrincipal();

    // Generates short-lived access token.
    String accessToken = jwtService.generateAccessToken(user);

    // Generates long-lived refresh token.
    String refreshToken = jwtService.generateRefreshToken(user);

    // Returns both tokens as an array.
    return new String[]{accessToken, refreshToken};
}
```

AuthServiceImpl

From a solution-delivery and governance standpoint, this service is the **central orchestration layer for authentication and user onboarding**. It consolidates login, signup, role evolution, and token lifecycle management into a single, transactionally consistent workflow.

1. Class-level posture

```
@RequiredArgsConstructor
@Service
@Transactional
public class AuthServiceImpl implements AuthService {
```

- **@Service**
  Positions this class as a core business service in the authentication domain.
- **@RequiredArgsConstructor**
  Ensures all dependencies are injected via constructor, promoting

immutability and clean dependency management.

- **@Transactional**
  Wraps public methods in transactions, guaranteeing **data consistency** across multi-step operations like signup and driver onboarding.

  This establishes the class as a **reliable orchestration engine**, not just a utility.

2. Dependency landscape

```
private final UserRepository userRepository;
private final ModelMapper modelMapper;
private final RiderService riderService;
private final WalletService walletService;
private final DriverService driverService;
private final PasswordEncoder passwordEncoder;
private final AuthenticationManager authenticationManager;
private final JWTService jwtService;
```

Each dependency has a clear accountability:

- Persistence → UserRepository
- DTO ↔ Entity translation → ModelMapper
- Domain onboarding → Rider, Wallet, Driver services
- Security primitives → PasswordEncoder, AuthenticationManager, JWTService

  This separation of concerns keeps the service **focused, testable, and extensible**.

3. Login flow

```
@Override
public String[] login(String email, String password) {
```

Execution logic

```
Authentication authentication = authenticationManager.authenticate(
    new UsernamePasswordAuthenticationToken(email, password)
);
```

- Delegates credential validation to Spring Security
- Ensures password comparison is handled by configured PasswordEncoder

- Avoids custom authentication logic (low risk, high reliability)

```
User user = (User) authentication.getPrincipal();
```

- Retrieves the authenticated security principal
- The user entity doubles as UserDetails

```
String accessToken = jwtService.generateAccessToken(user);
String refreshToken = jwtService.generateRefreshToken(user);
```

- Issues short-lived access token
- Issues long-lived refresh token

```
return new String[]{accessToken, refreshToken};
```

This delivers a **stateless, scalable authentication response**.

4. Signup flow

```
@Override
@Transactional
public UserDto signup(SignupDto signupDto) {
```

Step 1: Duplicate user check

```
User user = userRepository.findByEmail(signupDto.getEmail()).orElse(null);
if (user != null) {
  throw new RuntimeConflictException(
    "User already exist with email " + signupDto.getEmail()
  );
}
```

- Enforces email uniqueness
- Prevents identity collisions early

Step 2: User creation

```
User mapUser = modelMapper.map(signupDto, User.class);
mapUser.setRoles(Set.of(Role.RIDER));
```

- Maps request DTO to entity
- Assigns a default role (RIDER)

```
mapUser.setPassword(passwordEncoder.encode(mapUser.getPassword()));
```

- Ensures password is stored securely

- Eliminates plain-text password risk

Step 3: Persistence and onboarding

```
User savedUser = userRepository.save(mapUser);
```

After persistence, related domain entities are provisioned:

```
riderService.createNewRider(savedUser);
walletService.createNewWallet(savedUser);
```

This guarantees:

- Domain completeness
- No partially created users

Step 4: Response mapping

```
return modelMapper.map(savedUser, UserDto.class);
```

- Returns a safe, external-facing representation
- Avoids exposing internal security fields

5. Driver onboarding flow

```
@Override
public DriverDto onBoardNewDriver(Long userId, String vehicleId) {
```

Step 1: User validation

```
User user = userRepository.findById(userId).orElseThrow(
    () -> new ResourceNotFoundException("User not found with id:" + userId)
);
```

- Ensures user existence before role escalation

Step 2: Role conflict prevention

```
if (user.getRoles().contains(DRIVER)) {
    throw new RuntimeConflictException(
        "User with id:" + userId + " already exists"
    );
}
```

- Prevents duplicate driver creation
- Preserves role integrity

Step 3: Driver creation and role escalation

```
Driver createDriver = Driver.builder()
    .user(user)
    .rating(0.0)
    .vehicleId(vehicleId)
    .available(true)
    .build();

user.getRoles().add(DRIVER);
userRepository.save(user);
```

- Promotes the user to DRIVER
- Persists role change before downstream operations

```
Driver savedDriver = driverService.createNewDriver(createDriver);
```

- Delegates driver persistence to domain service

Step 4: Response mapping

```
return modelMapper.map(savedDriver, DriverDto.class);
```

6. Refresh token flow

```
@Override
public String refreshToken(String refreshToken) {
```

Execution logic

```
Long userId = jwtService.getUserIdFromToken(refreshToken);
```

- Validates refresh token
- Extracts trusted identity

```
User user = userRepository.findById(userId).orElseThrow(
    () -> new ResourceNotFoundException("User not found with id:" + userId)
);

return jwtService.generateAccessToken(user);
```

- Issues a new access token
- Maintains stateless re-authentication

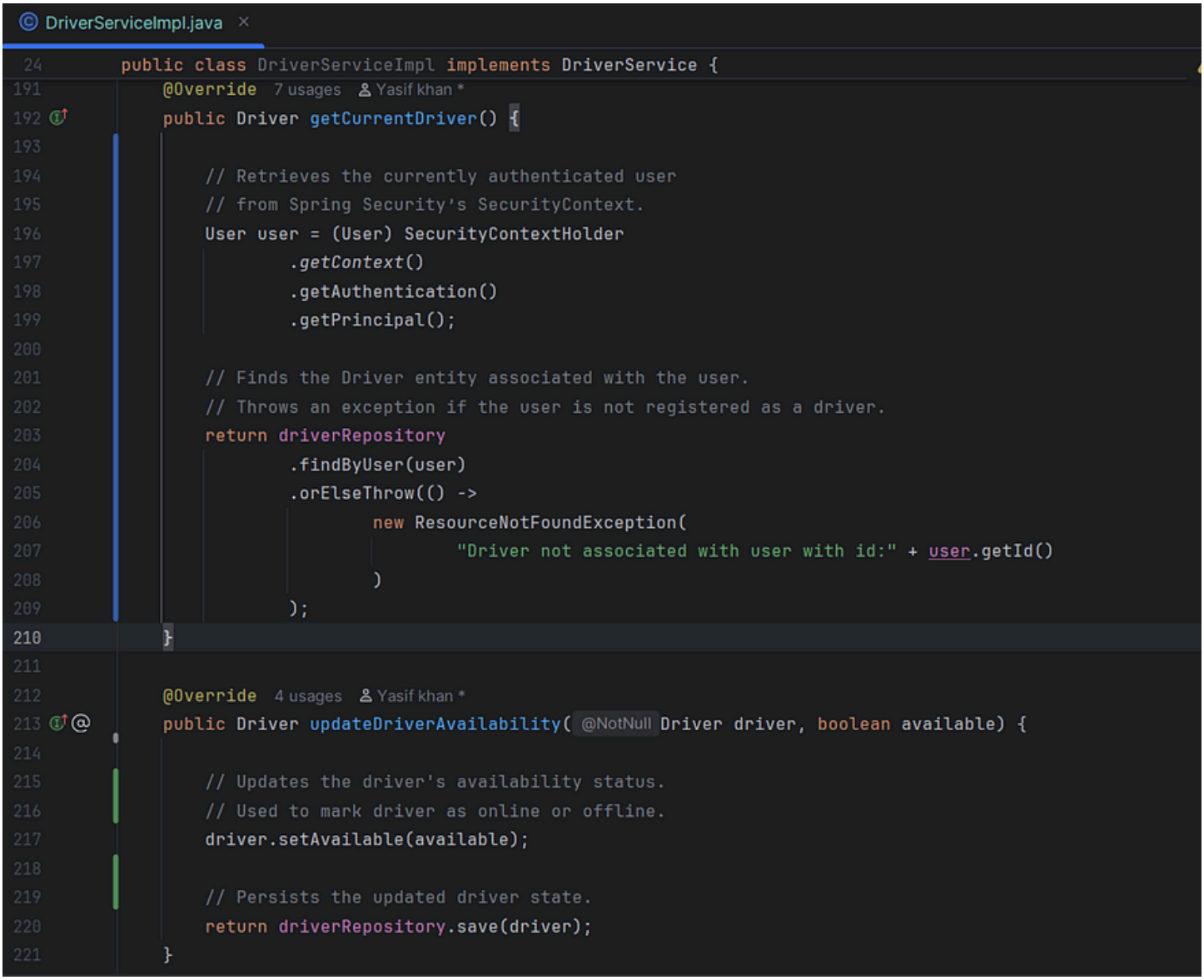
Executive summary

- Centralizes authentication and onboarding workflows

- Delegates security-critical operations to Spring Security
- Maintains transactional integrity across complex flows
- Enforces role evolution cleanly and safely
- Implements a robust JWT access/refresh token strategy

Net outcome: this service operates as a **high-confidence authentication orchestration layer**, balancing security, scalability, and clean domain ownership — well-positioned for enterprise growth and future enhancements like MFA, audit logging, or token revocation.

Press enter or click to view image in full size



DriverServiceImpl

From an operating-model perspective, this segment of DriverServiceImpl is **closing the loop between security context,**

domain identity, and persistence. It ensures driver-specific operations are always executed in the context of the authenticated user, while keeping state transitions explicit and auditable.

1. Resolving the current driver from the security context

```
@Override
public Driver getCurrentDriver() {
```

Execution flow

```
User user = (User) SecurityContextHolder
    .getContext()
    .getAuthentication()
    .getPrincipal();
```

- Pulls the authenticated principal directly from Spring Security
- Assumes JWT authentication has already populated the context
- Eliminates the need to pass user identifiers through APIs

This creates a **secure, identity-driven execution model**.

```
return driverRepository.findByUser(user)
  .orElseThrow(() -> new ResourceNotFoundException(
    "Driver not associated with user with id:" + user.getId()
  ));
```

- Maps the authenticated user to the Driver domain entity
- Fails fast if the user is not onboarded as a driver

This protects downstream logic from operating on invalid assumptions.

2. Updating driver availability

```
@Override
public Driver updateDriverAvailability(Driver driver, boolean available) {
```

Business intent

- Toggles whether a driver is available for ride matching
- Explicitly models a **state transition** rather than implicit behavior

```
driver.setAvailable(available);
return driverRepository.save(driver);
```

- Updates in-memory state
- Persists the change in a single, clear operation

This method is intentionally minimal, ensuring **predictable side effects** and easy traceability.

3. Creating a new driver

```
@Override
public Driver createNewDriver(Driver driver) {
   return driverRepository.save(driver);
}
```

Why this exists

- Encapsulates persistence logic inside the service layer
- Keeps controllers and upstream services decoupled from repositories
- Enables future hooks:
- Validation
- Event publishing
- Audit logging

Right now it's lean; strategically, it's **future-proofed**.

4. How this fits into the overall system

End-to-end perspective:

1. JWT filter authenticates request
2. SecurityContextHolder holds the authenticated User
3. getCurrentDriver() resolves the domain-specific identity
4. Driver state is read or mutated safely
5. Persistence is handled transactionally

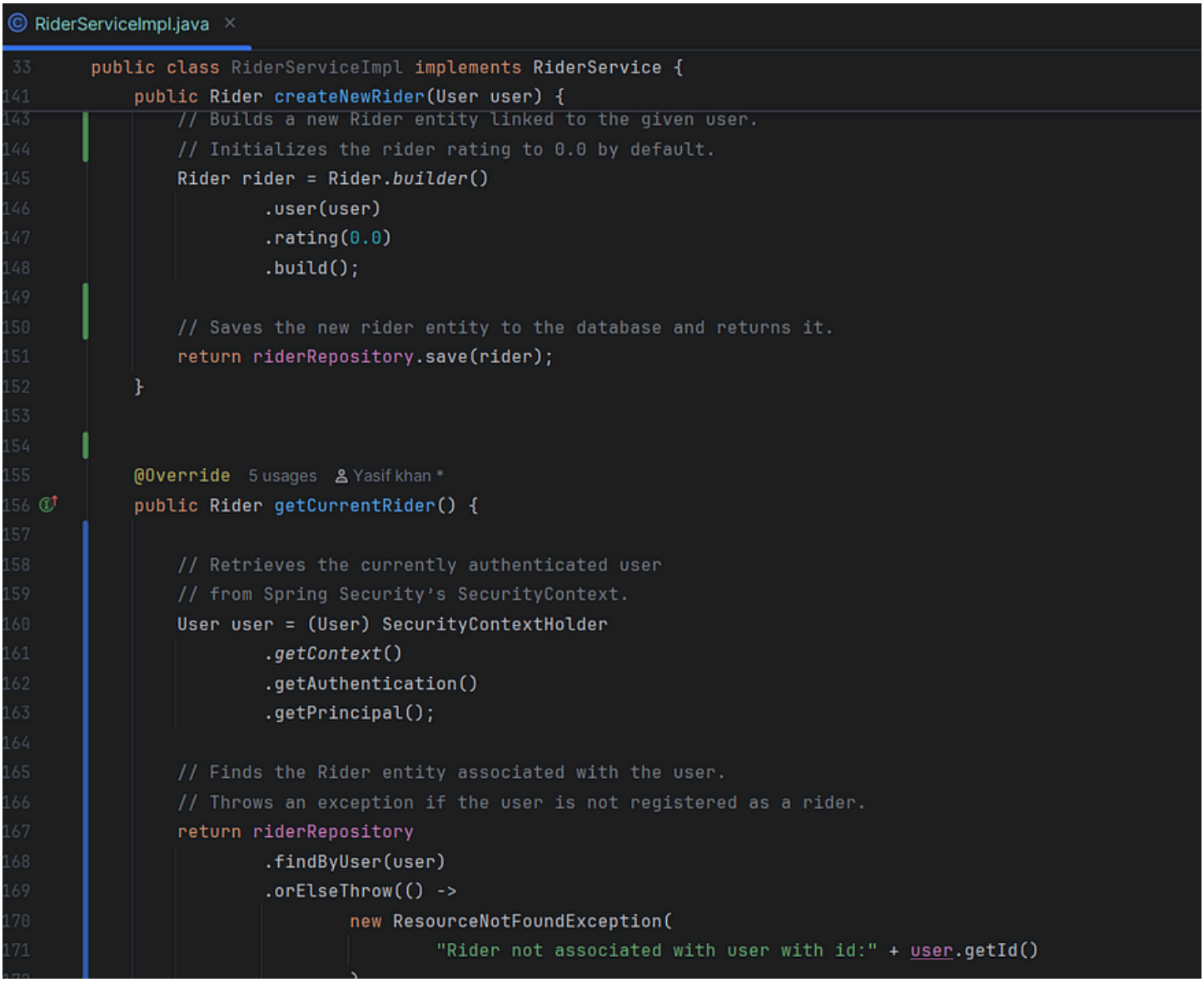
This design ensures **identity integrity** across all driver operations.

Executive summary

- Anchors driver actions to the authenticated user
- Avoids passing sensitive identifiers through request payloads
- Enforces clean separation between security, domain, and persistence
- Provides explicit, low-risk state transitions

Net result: this implementation delivers a **secure, context-aware driver service** that scales operationally while remaining easy to reason about and extend.

Press enter or click to view image in full size



RiderServiceImpl

From a solution and governance perspective, this RiderServiceImpl implementation mirrors the driver service's approach, providing a **secure, identity-driven, and transactionally consistent way to manage riders**. It ensures rider-specific operations are always executed in the context of the authenticated user.

1. Creating a new rider

```
@Override
public Rider createNewRider(User user) {
   Rider rider = Rider.builder()
         .user(user)
         .rating(0.0)
         .build();
   return riderRepository.save(rider);
}
```

Execution flow & intent

- **User linkage:** Associates the new rider with an existing user.
- **Initial state:** Sets a default rating of 0.0.
- **Persistence:** Saves the rider entity via riderRepository.

Strategic rationale:

- Ensures that **rider creation is atomic and consistent**.
- Keeps business logic centralized, avoiding scattered persistence calls.
- Provides a foundation for future enhancements such as adding default preferences or rewards during rider creation.

2. Resolving the current rider from security context

```
@Override
public Rider getCurrentRider() {
   User user = (User) SecurityContextHolder.getContext().getAuthentication().getPrincipal();
```

- Retrieves the authenticated User from the security context.
- Guarantees that **all rider operations are executed in the context of the logged-in user**, reducing risk of unauthorized access.

```
return riderRepository.findByUser(user)
      .orElseThrow(() -> new ResourceNotFoundException(
         "Rider not associated with user with id:" + user.getId()
      ));
```

- Maps the authenticated user to the corresponding rider entity.
- Fails fast if no rider is associated, preventing downstream errors.

Operational benefits:

- Enforces identity integrity across rider operations.
- Reduces risk of **unauthorized access** or invalid operations.

- Simplifies controller logic by abstracting user-to-rider mapping.

3. Integration with the larger system

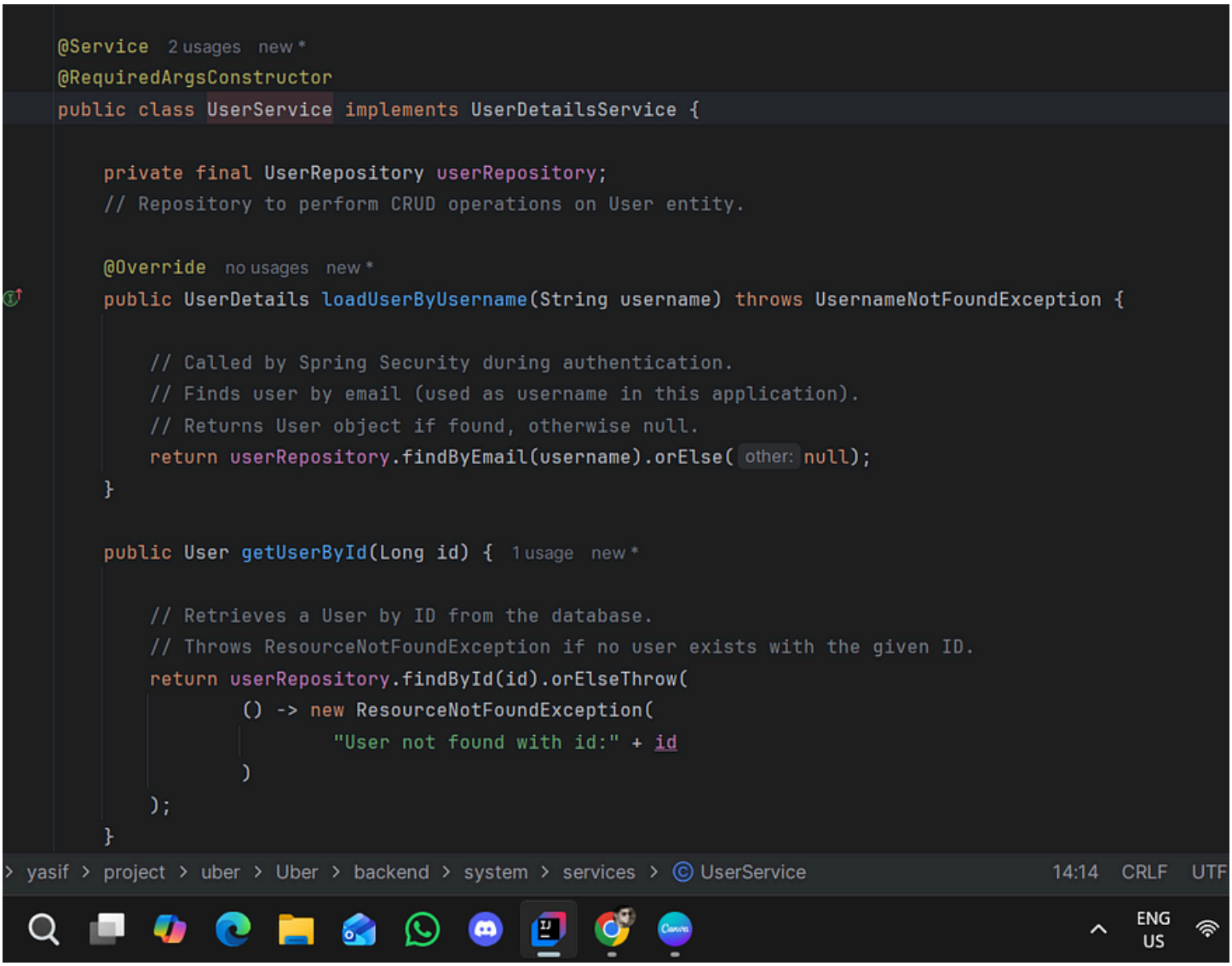
End-to-end workflow:

1. A new user signs up → RiderServiceImpl.createNewRider() creates a linked rider.
2. Authenticated rider requests data → getCurrentRider() fetches the correct rider entity automatically.
3. All rider-specific operations (booking, rating, wallet updates) operate on **context-aware entities**, ensuring security and consistency.

Executive summary

- Anchors rider operations to **authenticated users**.
- Maintains **clean separation of concerns**: security context, domain logic, and persistence.
- Provides **transactionally safe creation** of riders.
- Offers a **scalable, maintainable foundation** for additional rider features (e.g., loyalty, analytics, preferences).

Net outcome: This service ensures **secure, reliable, and context-aware management of riders**, fully aligned with enterprise-grade authentication and domain modeling best practices.



UserService

From a security and architectural standpoint, this UserService is the **bridge between Spring Security and your persistence layer**. It ensures that authentication, authorization, and user retrieval are handled consistently and securely across the application.

1. Class-level positioning

```
@Service
@RequiredArgsConstructor
public class UserService implements UserDetailsService {
```

- **@Service**
  Marks the class as a Spring-managed service, suitable for injection and lifecycle management.

- **@RequiredArgsConstructor**
  Enforces constructor-based dependency injection, ensuring immutability and clean code.
- **implements UserDetailsService**
  Connects the service directly to Spring Security.
  This interface is required for the authentication manager to retrieve user details during login.

2. Loading users by username (email)

```
@Override
public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
   return userRepository.findByEmail(username).orElse(null);
}
```

Strategic intent

- Retrieves a User entity by email (used as the username in your system).
- The returned User entity implements UserDetails, so Spring Security can use it for authentication and authorization.
- Returning null triggers Spring Security's default behavior for invalid users (could also throw UsernameNotFoundException for clarity).

Operational benefits:

- Decouples authentication logic from controllers.
- Makes user retrieval consistent for all authentication flows.
- Ensures roles and authorities are automatically available via the UserDetails implementation.

3. Retrieving user by ID

```
public User getUserById(Long id) {
   return userRepository.findById(id).orElseThrow(
      () -> new ResourceNotFoundException("User not found with id:" + id)
   );
}
```

- Provides a **reliable way to fetch a user entity** for internal service operations (e.g., token validation, onboarding).
- Throws a domain-specific exception if the user does not exist, which is crucial for predictable downstream logic.

- Supports workflows like refresh token validation or linking drivers/riders to users.

4. Integration with the broader security flow

1. **Authentication:** Spring Security calls loadUserByUsername during login.
2. **JWT generation:** AuthService retrieves the User via getUserById to generate tokens.
3. **Domain services:** Driver/Rider services can use getUserById to link domain entities to the correct user.
4. **Security context population:** JWT filters leverage UserService to retrieve validated users.

   This ensures **consistency, security, and traceability** throughout the application.

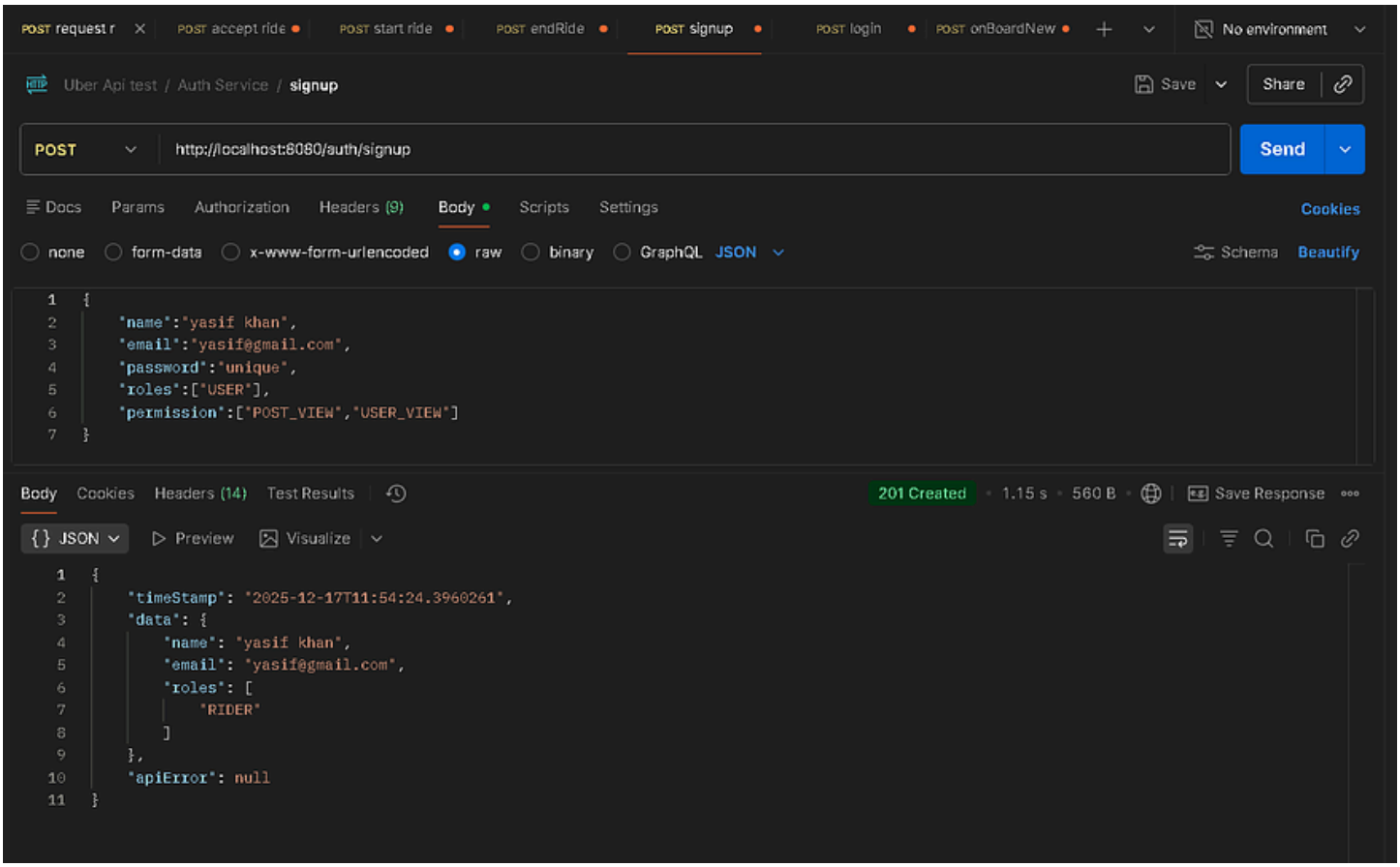
Executive summary

- Acts as the **gateway between Spring Security and persistence**.
- Provides **robust user retrieval mechanisms** for both authentication and domain services.
- Ensures **role and authority mapping** flows naturally via UserDetails.
- Supports enterprise-level flows like JWT authentication, onboarding, and role management.

Net impact: This service establishes a **secure, centralized, and maintainable approach** to user identity management, enabling authentication and authorization to work seamlessly across the application.

Outputs: SignUp

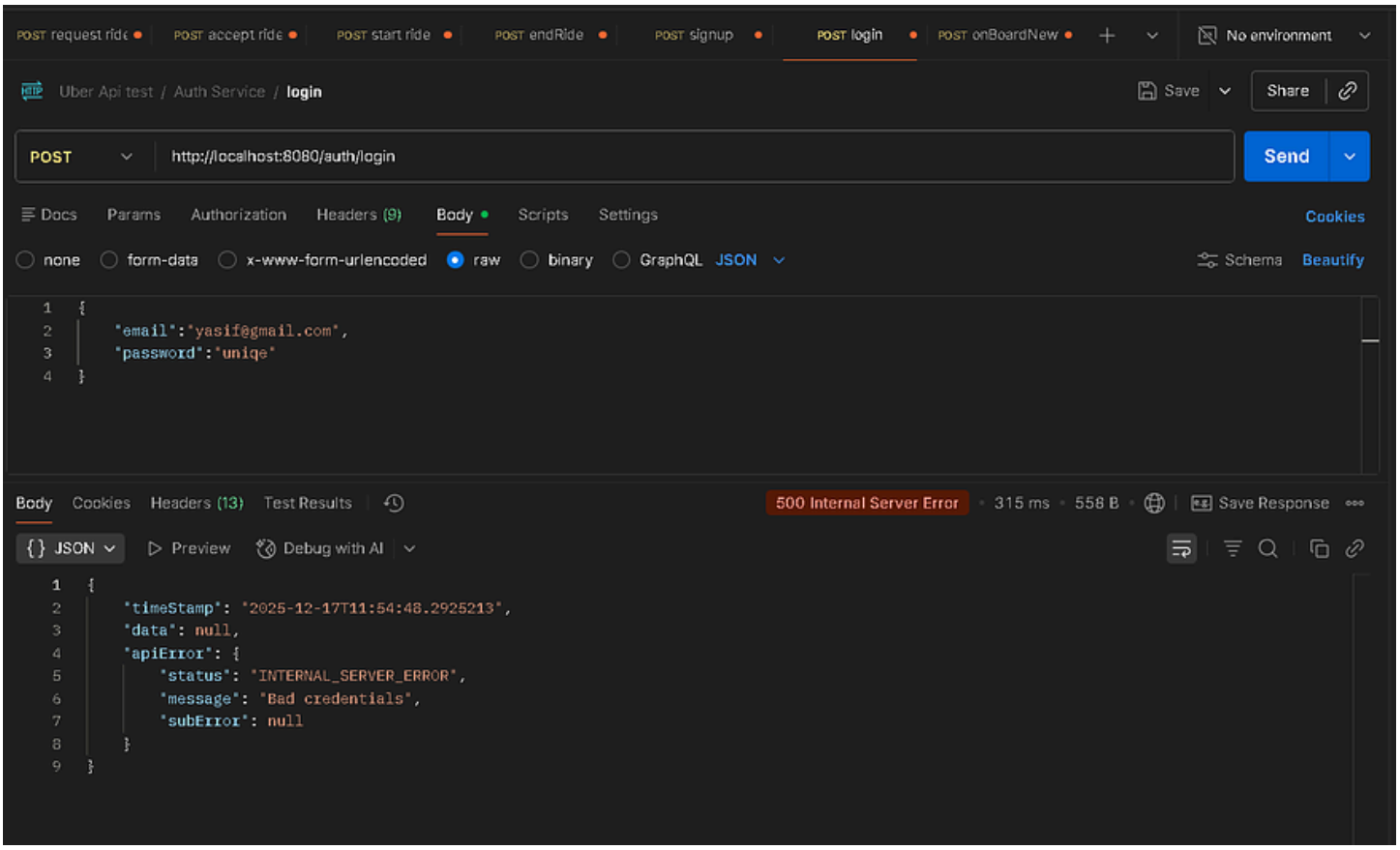
Press enter or click to view image in full size



singup

Login with wrong password

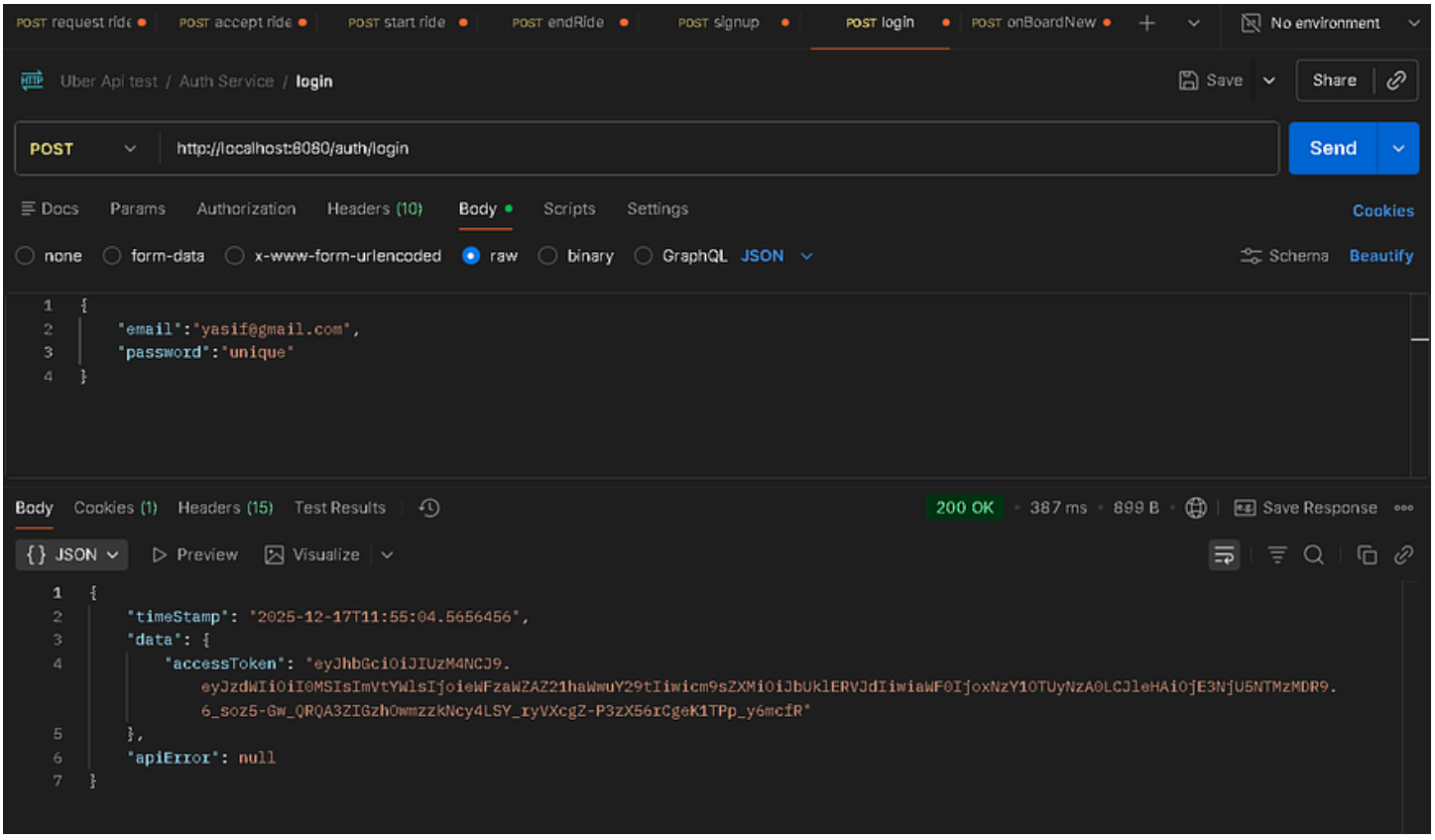
Press enter or click to view image in full size



incorrect password

Login with correct password

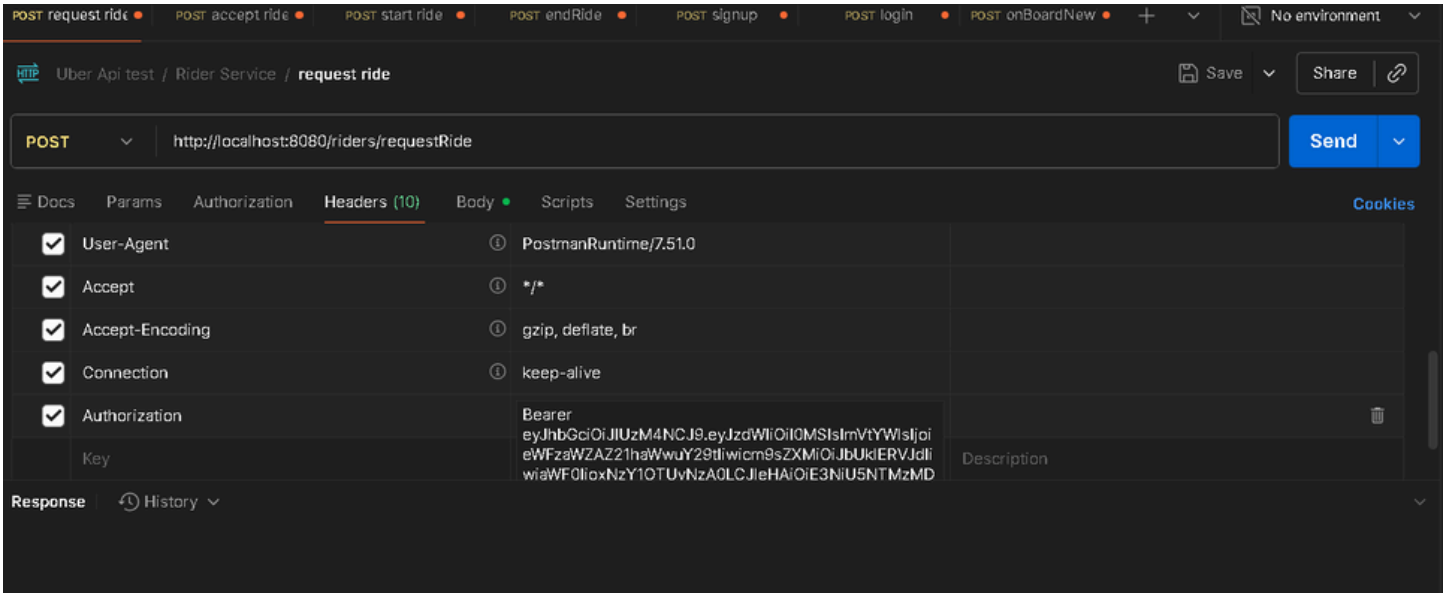
Press enter or click to view image in full size



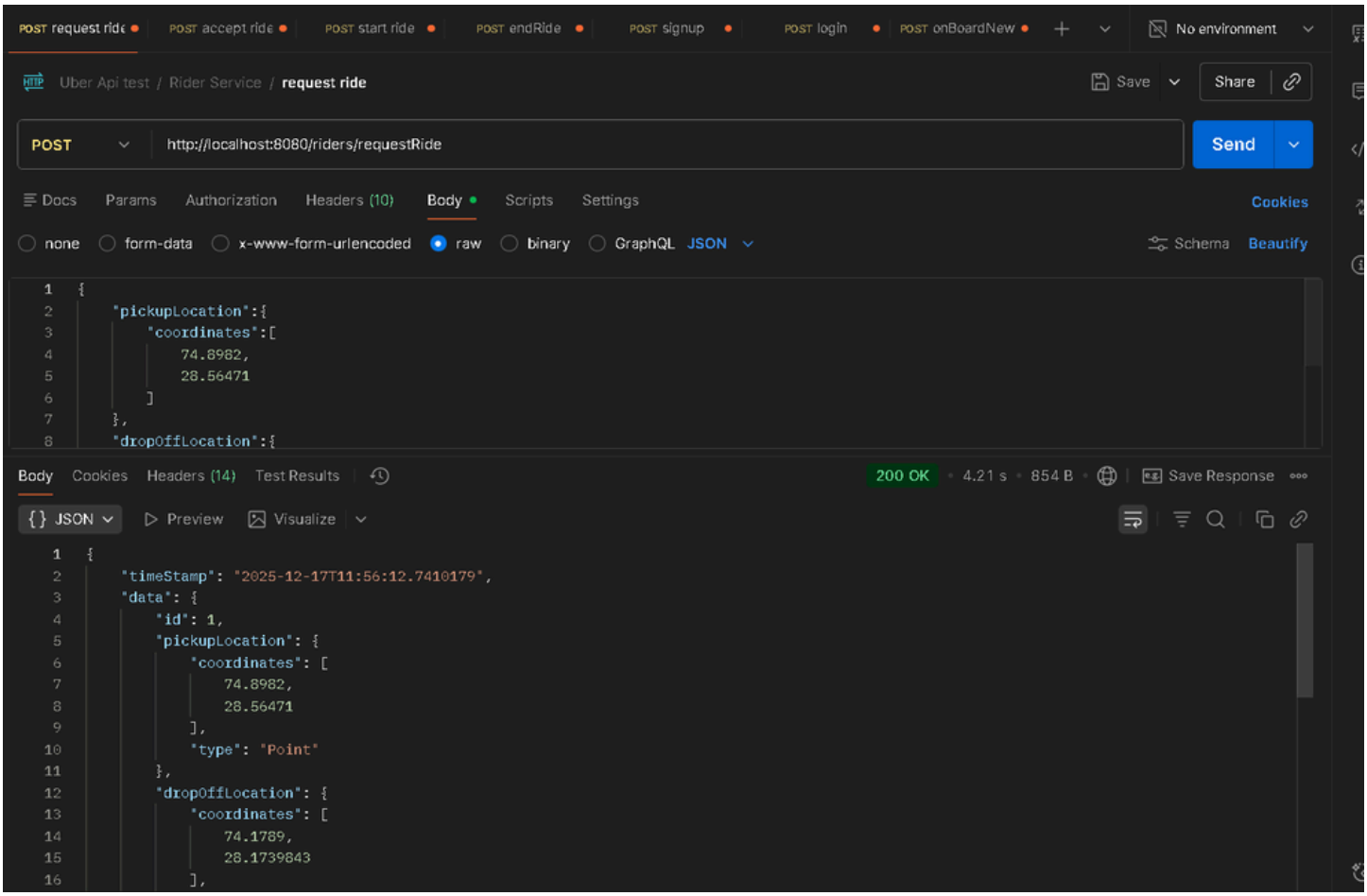
correct password

Request Ride

Press enter or click to view image in full size



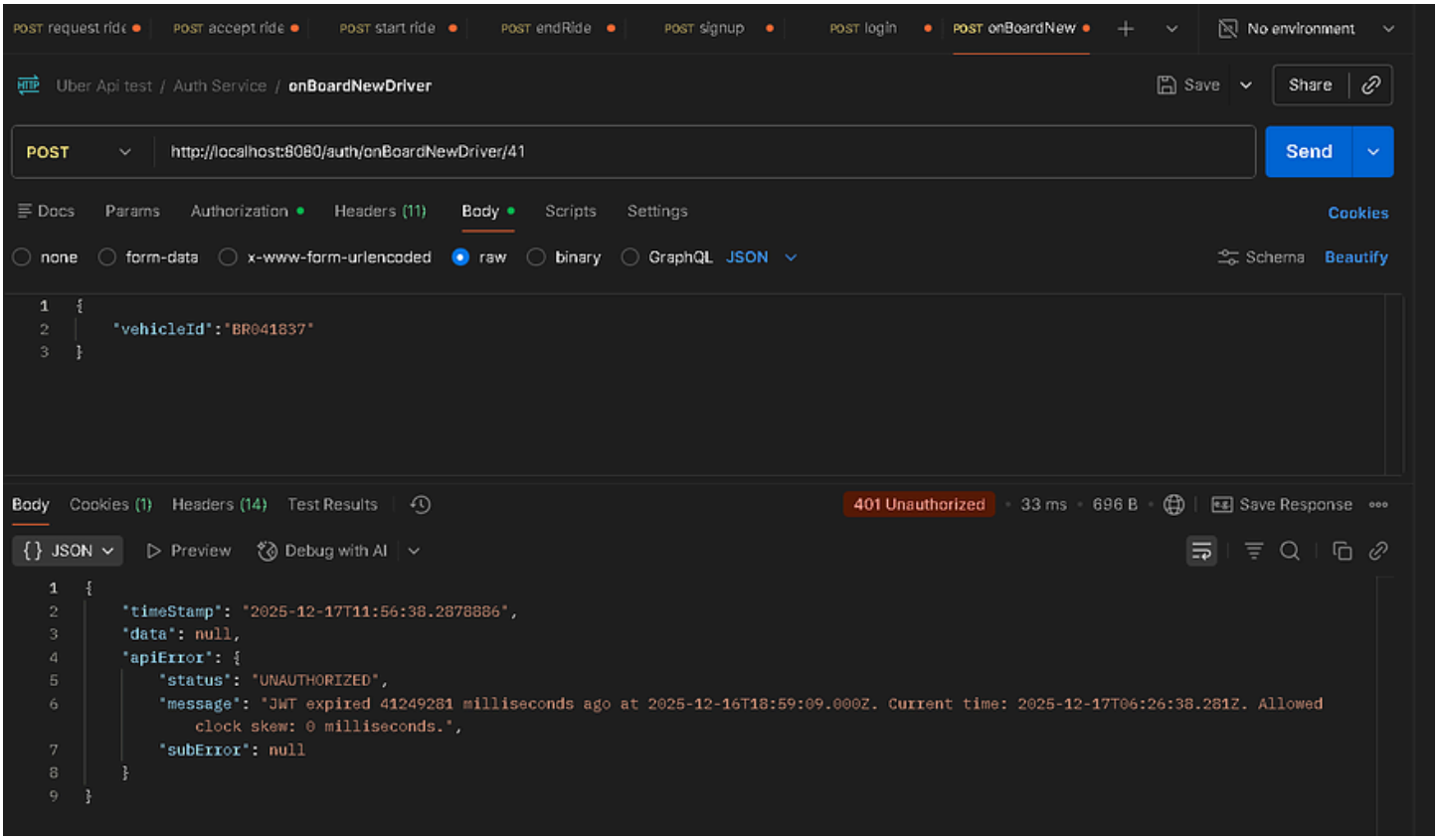
Press enter or click to view image in full size



Request a ride

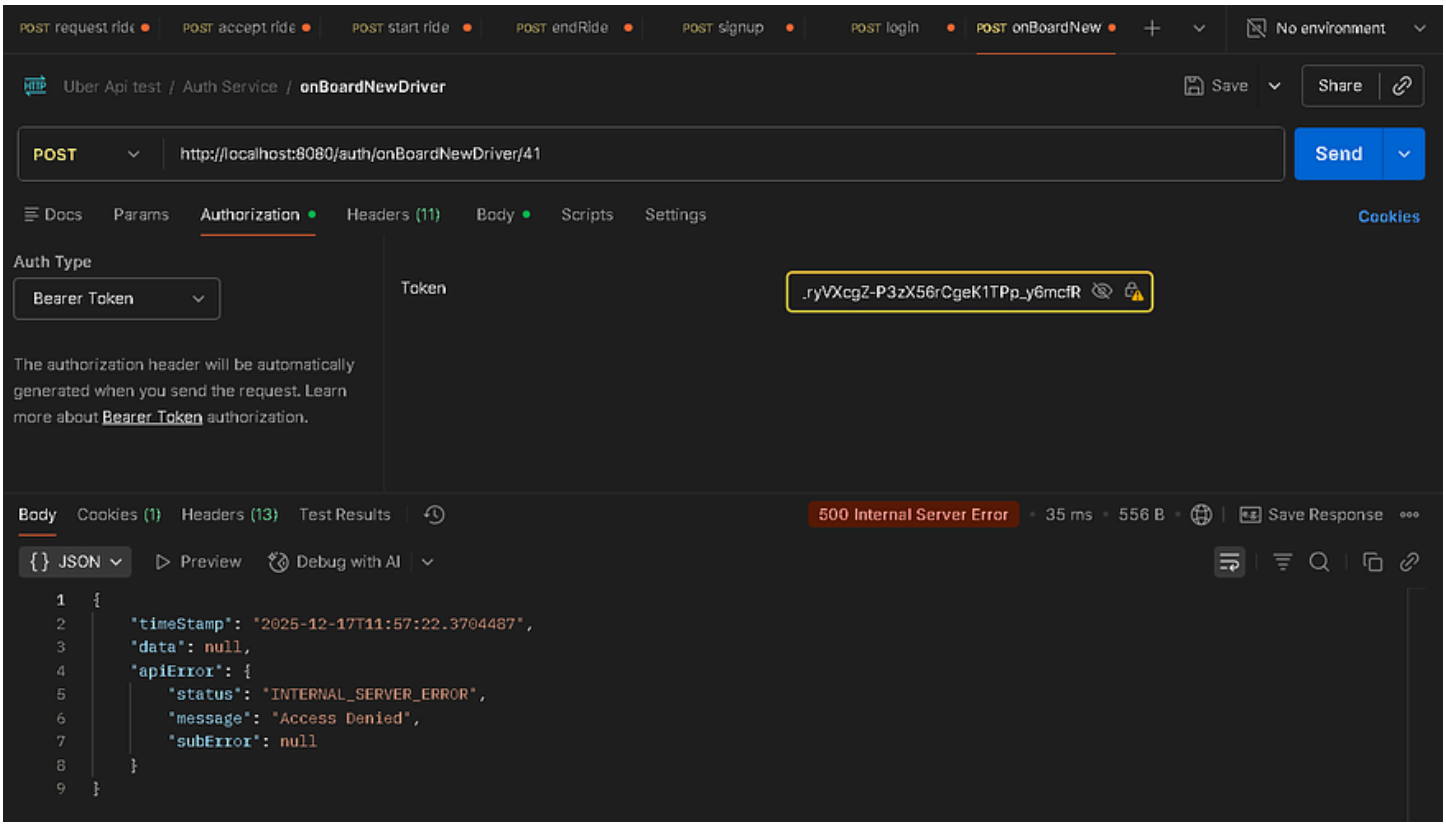
JWT token expiration

Press enter or click to view image in full size



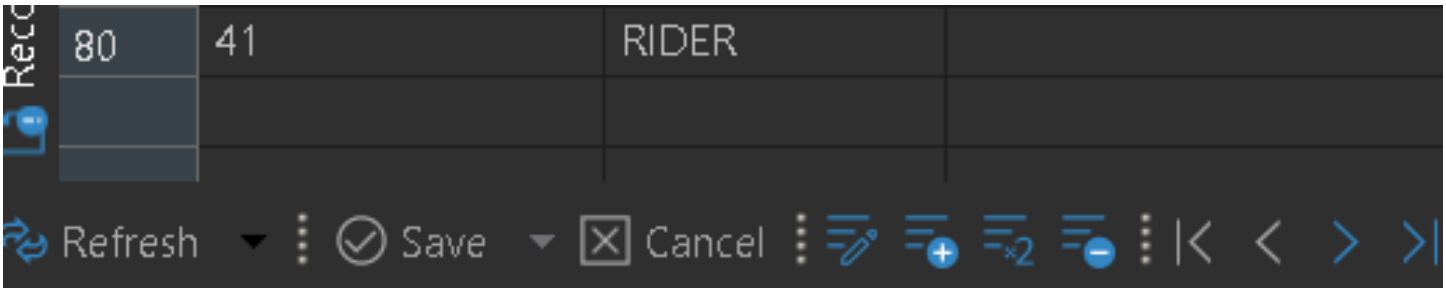
OnBoardNewDriver(ADMIN only) without permission

Press enter or click to view image in full size

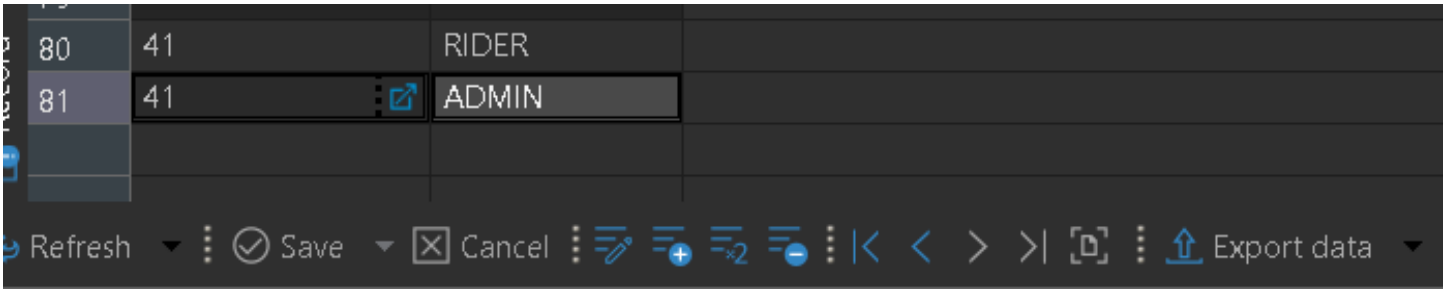


Now i change the User into ADMIN also

Press enter or click to view image in full size



Press enter or click to view image in full size



user with id 41 change into ADMIN

Now OnBoardNewDriver

Press enter or click to view image in full size