Part IV

irian
Software Engineering Par Excellence

VO, DTO

POST, PUT, @RequestBody

Validation

Exception Handling

Spring Security, JWT

CORS, CSRF

OAuth2

by Georgia Leustean

VOs vs DTOs & CQRS

VOs as Read Models

- Represent data retrieved from queries
- Populate directly from database results
- Immutable, tailored to read-only use
- Shape fits query requirements (projections)

DTOs as Write Models

- Receive data from clients for persistence
- Map to domain entities to save changes
- Often mutable to support binding
- Include validation and transformation logic

VO vs DTO: Key Differences

- Role – VO: Read-only representation vs DTO: Write/input model
- Source – Populated from queries/projections vs Received from client requests
- Mutability – Immutable vs Mutable
- Use case – Query side vs Command side
- Validation – Assumed correct vs Explicit validation

Aligning with CQRS

- Query Model: Use VOs for read operations
- Command Model: Use DTOs for write operations
- Separate handlers/services for reads and writes
- Optimize models independently

Best Practices

- Keep VOs immutable and focused on read needs
- Design DTOs with clear validation rules
- Maintain separate query/write pipelines
- Document data contracts for each role

Writing POST & PUT Methods in Spring

HTTP POST vs PUT

- **POST**: Create new resource, not idempotent
- **PUT**: Update/replace resource, idempotent

POST Example

```java
@PostMapping
public ResponseEntity<ItemDto> create(@Valid @RequestBody ItemDto dto) {
    ItemDto created = service.create(dto);
    URI location = URI.create("/api/items/" + created.getId());
    return ResponseEntity.created(location).body(created);
}```
```

PUT Example

```java
@PutMapping("/{id}")
public ResponseEntity<ItemDto> update(@PathVariable Long id,
    @Valid @RequestBody ItemDto dto) {
    ItemDto updated = service.update(id, dto);
    return ResponseEntity.ok(updated);
}```
```

What is @RequestBody?

- Annotation to bind HTTP request body to a Java object
- Part of Spring MVC @Controller and @RestController
- Uses HttpMessageConverters (e.g., Jackson)
- Supports JSON, XML, and other formats

Basic Usage

```java
@PostMapping("/users")
public ResponseEntity<UserDto> createUser(
    @RequestBody UserDto userDto) {
    UserDto created = userService.create(userDto);
    return ResponseEntity.status(HttpStatus.CREATED).body(created);
}```
```

JSON Mapping

- Ensure `Content-Type: application/json` header
- Spring uses Jackson by default to deserialize JSON
- Unknown properties: configure FAIL_ON_UNKNOWN_PROPERTIES
- Customize with @JsonProperty, @JsonIgnore, etc.

Advanced Usage

- Use `@RequestBody(required = false)` for optional bodies
- Consume different media types with `consumes` attribute:
- `@PostMapping(consumes = MediaType.APPLICATION_XML_VALUE)`
- Implement custom HttpMessageConverter for new formats

Best Practices

- Use DTOs to decouple API from domain models
- Validate input early and clearly
- Handle missing or malformed bodies gracefully
- Document API with OpenAPI/Swagger annotations
- Limit body size to prevent abuse

Transactions & Flushing

- EntityManager requires active transaction for write operations
- em.flush(): pushes changes to the database without commit
- Commit automatically triggers flush
- em.clear(): detaches all entities

Transactions & @Transactional in Spring

What is a Transaction?

- A unit of work that is atomic, consistent, isolated, durable (ACID).
- Ensures all operations succeed or none take effect.
- Critical for data integrity in databases.

ACID Properties

- Atomicity – All-or-nothing execution.
- Consistency – Transition from one valid state to another.
- Isolation – Concurrent transactions do not interfere.
- Durability – Once committed, results are permanent.

Declarative vs Programmatic

- Declarative: Use @Transactional annotation at class/method level.
- Programmatic: Use TransactionTemplate or PlatformTransactionManager.
- Declarative preferred for simplicity and readability.

@Transactional Annotation

```java
@Service
public class OrderService {
    @Transactional
    public void placeOrder(OrderDTO orderDto) {
        // business logic
    }
}
```

Propagation Behaviors

- REQUIRED: join existing or create new.
- REQUIRES_NEW: suspend current, start new.
- MANDATORY: must run within existing.
- SUPPORTS: join if exists, else run non-transactional.

Isolation Levels

- READ_UNCOMMITTED: dirty reads allowed.
- READ_COMMITTED: prevents dirty reads.
- REPEATABLE_READ: prevents non-repeatable reads.
- SERIALIZABLE: full isolation, lowest throughput.

Rollback Rules

- Default: rollback on unchecked exceptions (RuntimeException).
- use rollbackFor / noRollbackFor to customize.
- `@Transactional(rollbackFor = Exception.class)`

Best Practices

- Keep transactions short to avoid locks and contention.
- Avoid database calls in loops within transactions.
- Use readOnly=true for read-only operations.
- Document transaction boundaries and behaviors.

Pitfalls of Missing @Transactional at Service Layer

Over-reliance on Spring Data JPA repository transactions

LazyInitializationException

- Occurs when accessing lazy-loaded associations outside a transaction
  - Example: Fetch entity, close repository scope, then access a collection
  - Results in org.hibernate.LazyInitializationException

Partial Persistence / Inconsistent State

- Multiple repository calls are not atomic
  - Service method calls save() on RepositoryA then RepositoryB
  - If second call fails, first change remains persisted
  - Leads to data inconsistency

Lack of Proper Rollback

- Without @Transactional, exceptions won't rollback multiple operations
  - No global rollback is applied
  - Requires manual compensation logic

Propagation & Isolation Issues

- Nested repository calls lack clear propagation
  - @Transactional allows setting propagation and isolation
  - Repository-level defaults may not suit complex flows

Code Example: Without @Transactional

```
public class PizzaService {
    private final PizzaRepository pizzaRepo;
    private final OrderRepository orderRepo;

    public PizzaService(PizzaRepository pizzaRepo, OrderRepository orderRepo) {
        this.pizzaRepo = pizzaRepo;
        this.orderRepo = orderRepo;
    }

    public void createOrder(OrderDto dto) {
        // Save pizza
        Pizza pizza = new Pizza(dto.getName(), dto.getSize());
        pizzaRepo.save(pizza);

        // If an exception occurs here, pizza remains persisted, order not

        // Save order
        Order order = new Order(dto.getCustomer(), pizza);
        orderRepo.save(order);
}
```

Code Example: With @Transactional

```
@Service
public class PizzaService {
    private final PizzaRepository pizzaRepo;
    private final OrderRepository orderRepo;

    public PizzaService(PizzaRepository pizzaRepo, OrderRepository orderRepo) {
        this.pizzaRepo = pizzaRepo;
        this.orderRepo = orderRepo;
    }

@Transactional
public void createOrder(OrderDto dto) {
        // Save pizza and order within one transaction
        Pizza pizza = new Pizza(dto.getName(), dto.getSize());
        pizzaRepo.save(pizza);
        Order order = new Order(dto.getCustomer(), pizza);
        orderRepo.save(order);

        // Exception here triggers full rollback of both operations
    }
```

Ex 01

- Create PizzaDTO, create save methods in Controller + Service and save new Pizzas to the database (using Postman)

OBS: If using data inserted via data.sql you will need to specify the next id to be generated like this:

ALTER TABLE pizza ALTER COLUMN id RESTART WITH 10;

Use the next value after the ones you inserted

Using ResponseEntity in Spring

What is ResponseEntity?

- Wrapper around HTTP response, including status, headers, and body
- Part of Spring MVC – org.springframework.http.ResponseEntity<T>
- Used in @RestController methods to craft full responses
- Provides fluent builders for flexibility

Creating ResponseEntity

```
new ResponseEntity<>(body, status)
```

Or use static builders:

```
ResponseEntity.ok(body)
ResponseEntity.status(HttpStatus.CREATED).body(body)
```

Common HTTP Status Codes

- 200 OK – Successful GET/PUT requests
- 201 Created – Successful POST with resource creation
- 204 No Content – Successful DELETE or no body
- 400 Bad Request – Validation or client errors
- 404 Not Found – Resource missing
- 500 Internal Server Error – Unhandled exceptions

Setting Headers

```
ResponseEntity.ok().header("X-Custom-Header", "value").body(body)
ResponseEntity.created(uri).headers(headers).body(body)
Use HttpHeaders for multiple headers
```

Example Code

```java
@PostMapping("/items")
public ResponseEntity<ItemDto> createItem(@RequestBody ItemDto dto) {
    ItemDto created = service.create(dto);
    URI location = URI.create("/api/items/" + created.getId());
    return ResponseEntity.created(location)
                         .header("X-Trace-Id", "12345")
                         .body(created);
}```
```

Best Practices

- Always return appropriate status codes
- Include Location header for newly created resources
- Handle errors and return meaningful messages
- Avoid exposing internal details in responses
- Use generics for type safety

What is @Valid and Bean Validation

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

- @Valid triggers JSR-303/JSR-380 validation on method arguments
- Supported via Hibernate Validator by default in Spring Boot
- Integrates with HttpMessageConverters for request bodies
- Ensures data integrity and reduces manual checks

Predefined Constraint Annotations

- @NotNull – Field must not be null
- @NotEmpty – String/List must not be empty
- @NotBlank – String must contain non-whitespace
- @Size(min, max) – Size constraints for String, Collection
- @Email – Valid email format
- @Pattern(regexp) – Matches regex pattern
- @Min/@Max – Numeric range constraints
- @Positive/@Negative – Numeric sign constraints

Annotating DTO Fields

```java
public class UserDto {
    @NotBlank
    private String username;

    @Email
    @NotNull
    private String email;

    @Size(min = 8, max = 20)
    private String password;
}
```

Using @Valid in Controller

```java
@RestController
@RequestMapping("/api/users")
public class UserController {

    @PostMapping
    public ResponseEntity<UserDto> createUser(
        @Valid @RequestBody UserDto userDto,
        BindingResult result) {
      if (result.hasErrors()) {
        // handle errors
      }
      // service call
    }
```

Ex 02

- Add validation on PizzaDTO to make sure name has only letter characters and space and the price is minimum 10 and maximum 100 and all fields are required
- Use Response Entity to return appropriate statuses

For ease of use:

ResponseEntity.created(location).build();

ResponseEntity.badRequest().body(result.getAllErrors());

Creating Custom Validation Annotations in Spring

1. Defining the Annotation

```java
@Documented
@Constraint(validatedBy = NameValidator.class)
@Target({ElementType.FIELD, ElementType.PARAMETER})
@Retention(RetentionPolicy.RUNTIME)
public @interface ValidName {
    String message() default "Invalid name";
    Class<?>[] groups() default {};
    Class<? extends Payload>[] payload() default {};
}
```

2. Implementing the Validator

```java
public class NameValidator implements ConstraintValidator<ValidName, String> {
    @Override
    public void initialize(ValidName constraint) { }

    @Override
    public boolean isValid(String value, ConstraintValidatorContext ctx) {
        return value != null && value.matches("[A-Za-z ]+");
    }
}
```

3. Wiring with @Constraint

- - The `validatedBy` attribute links to your validator class
- - Spring Boot auto-detects ConstraintValidator implementations
- - Ensure `hibernate-validator` is on the classpath

4. Using the Annotation

```java
public class UserDto {
    @ValidName
    private String fullName;

    // other fields/getters/setters
}
```

5. Customizing Messages

- - Use `message` attribute in the annotation
- - Reference messages in `ValidationMessages.properties`:
- `validname.invalid=Name must contain only letters and spaces`
- - Support i18n by locale-specific files

Global Validation Error Handling with @RestControllerAdvice

Manual Validation Drawbacks

- Controllers cluttered with validation logic
- Repetitive error checking across endpoints
- Inconsistent error response formats
- Harder to maintain and evolve

@RestControllerAdvice for Global Handling

- Centralizes exception handling logic
- Applies across all @RestController endpoints
- Keeps controllers focused on business logic
- Consistent response format for errors

Example: UniversalExceptionHandler

```java
@RestControllerAdvice
public class UniversalExceptionHandler {


    @ResponseStatus(HttpStatus.BAD_REQUEST)
    @ExceptionHandler(MethodArgumentNotValidException.class)
    public Map<String, String> handleValidationExceptions(MethodArgumentNotValidException ex) {
        Map<String, String> errors = new HashMap<>();
        ex.getBindingResult()
          .getAllErrors()
          .forEach(error -> {
              String field = ((FieldError) error).getField();
              String msg = error.getDefaultMessage();
              errors.put(field, msg);
          });
        return errors;
    }
}
```

Benefits of Global Handling

- DRY: No duplication across controllers
- Consistency: Uniform error structure
- Maintainability: Single place to update
- Cleaner Controllers: Focus on core logic

Best Practices

- Define a standard error response DTO
- Include error codes and user-friendly messages
- Support internationalization (i18n)
- Log exceptions appropriately
- Handle other exceptions (e.g., NotFound, AccessDenied)

Ex 03

- Add custom validation to check there are no duplicate pizza names

You can use existsBy keyword at repository level

- Add Global exception handling

Introduction to Spring Security

Overview, JWT Authentication, OAuth2, OpenID Connect, Session-based Auth

What is Spring Security?

- Authentication & authorization framework
- Part of the Spring ecosystem
- Protects applications at method and URL levels

Core Concepts

- SecurityContext & SecurityContextHolder
- Authentication & GrantedAuthority
- UserDetails & UserDetailsService

Key Classes & Interfaces

- WebSecurityConfigurerAdapter (or SecurityFilterChain)
- AuthenticationManager & Provider
- OncePerRequestFilter for custom filters

Security Filter Chain Architecture

- Chain of Servlet filters applied to incoming requests
- Built-in filters: UsernamePasswordAuthenticationFilter, BasicAuthenticationFilter, etc.
- Custom filters can be added before or after standard ones
- Filters handle authentication, authorization, CSRF, CORS, etc.

AuthenticationManager & AuthenticationProvider

- AuthenticationManager: delegates to a list of providers
- AuthenticationProvider: performs authentication logic
- Common providers: DaoAuthenticationProvider, JwtAuthenticationProvider
- Configurable provider list for flexible auth strategies

SecurityContext & SecurityContextHolder

- SecurityContext holds Authentication info for current user
- SecurityContextHolder stores context per thread (ThreadLocal)
- Access via SecurityContextHolder.getContext()
- Cleared automatically at request end

UserDetailsService

- Interface for loading user-specific data
- Method: loadUserByUsername(String username)
- Implementations: InMemoryUserDetailsManager, JdbcUserDetailsManager, custom
- Returns UserDetails with username, password, authorities

Password Encryption

- PasswordEncoder interface for hashing passwords
- BCryptPasswordEncoder: default secure implementation
- DelegatingPasswordEncoder for multiple encoding schemes
- Use strong work factor (e.g., BCrypt strength 10+)

JWT Authentication Overview

- Stateless token-based mechanism
- JSON Web Tokens contain claims
- Signed to verify integrity

Implementing JWT Auth - Steps

- 1. Add Spring Security & JWT dependencies
- 2. Create JwtUtil for token creation/validation
- 3. Implement Authentication filter
- 4. Configure SecurityFilterChain
- 5. Protect endpoints & test

Dependency Setup

- spring-boot-starter-security
- jjwt (io.jsonwebtoken) or auth0 java-jwt
- spring-boot-starter-web

Configuring Security

- Define SecurityFilterChain bean
- Permit PUBLIC endpoints, secure others
- Register JwtAuthenticationFilter before UsernamePasswordAuthFilter

JwtUtil Class

- Generate token with claims & expiration
- Validate token signature & expiry
- Extract username & roles from token

Filters & Authorization

- Authentication Filter: Validate JWT, set SecurityContext
- Authorization: @PreAuthorize or hasRole()
- Handle AccessDenied & AuthenticationEntryPoint

Dependencies (pom.xml)

```
<dependencies>
  <!-- Spring Boot starters -->
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
```

```
 <!-- JJWT -->
  <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>
</dependencies>
```

Configuration (application.properties)

- `jwt.secret=YourSuperSecretKeyOfMinLength32characters`
- `jwt.expirationMs=3600000`

JwtUtils.java

```
@Component
public class JwtUtils {
    @Value("${jwt.secret}") private String jwtSecret;
    @Value("${jwt.expirationMs}") private long jwtExpirationMs;
    private Key key;
    @PostConstruct
    public void init() {
        key = Keys.hmacShaKeyFor(jwtSecret.getBytes(StandardCharsets.UTF_8));
    }
    public String generateToken(String username) {
        Date now = new Date();
        Date expiryDate = new Date(now.getTime() + jwtExpirationMs);
        return Jwts.builder()
                .setSubject(username)
                .setIssuedAt(now)
                .setExpiration(expiryDate)
                .signWith(key, SignatureAlgorithm.HS256)
                .compact();
    }
    public String getUsernameFromToken(String token) {
        return Jwts.parserBuilder()
                .setSigningKey(key)
                .build()
                .parseClaimsJws(token)
                .getBody()
                .getSubject();
    }
```

```
public boolean validateToken(String token) {
        try {
            Jwts.parserBuilder()
                .setSigningKey(key)
                .build()
                .parseClaimsJws(token);
            return true;
        } catch (JwtException | IllegalArgumentException e) { return false; }
    }
}
```

SecurityConfig.java

```
@Configuration
@EnableMethodSecurity(securedEnabled = true)
public class SecurityConfig {
    @Bean
    public UserDetailsService userDetailsService(PasswordEncoder encoder) {
        UserDetails user = User.withUsername("user")
                               .password(encoder.encode("password"))
                               .roles("USER").build();
        UserDetails admin = User.withUsername("admin")
                                .password(encoder.encode("admin123"))
                                .roles("ADMIN").build();
        return new InMemoryUserDetailsManager(user, admin);
    }
    @Bean public PasswordEncoder passwordEncoder() { return new BCryptPasswordEncoder(); }
    @Bean public AuthenticationManager authenticationManager(
        AuthenticationConfiguration config) throws Exception {
        return config.getAuthenticationManager();
    }
    @Bean
    public JwtAuthenticationFilter jwtAuthenticationFilter(JwtUtils jwtUtils,
                                                           UserDetailsService uds) {
        return new JwtAuthenticationFilter(jwtUtils, uds);
    }
```

```
@Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http,
                                                   JwtAuthenticationFilter
jwtFilter) throws Exception {
        http.csrf(csrf -> csrf.disable())
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/auth/**").permitAll()
                .anyRequest().authenticated()
            )
            .sessionManagement(sess -> sess
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
            )
            .addFilterBefore(jwtFilter,
UsernamePasswordAuthenticationFilter.class);
        return http.build();
    }
}
```

JwtAuthenticationFilter.java

```
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private final JwtUtils jwtUtils;
    private final UserDetailsService userDetailsService;

    public JwtAuthenticationFilter(JwtUtils jwtUtils, UserDetailsService uds) {
        this.jwtUtils = jwtUtils;
        this.userDetailsService = uds;
    }

    @Override
    protected void doFilterInternal(HttpServletRequest req, HttpServletResponse res, FilterChain chain) throws ServletException, IOException {
        String header = req.getHeader("Authorization");
        if (header != null && header.startsWith("Bearer ")) {
            String token = header.substring(7);
            if (jwtUtils.validateToken(token)) {
                String username = jwtUtils.getUsernameFromToken(token);
                UserDetails user = userDetailsService.loadUserByUsername(username);

                UsernamePasswordAuthenticationToken auth = new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
                SecurityContextHolder.getContext().setAuthentication(auth);
            }
        }
        chain.doFilter(req, res);
    }
}
```

Defining Roles vs Authorities

- Roles are a type of authority prefixed with 'ROLE_'
- Authorities (GrantedAuthority) represent permissions
- SimpleGrantedAuthority example:

  new SimpleGrantedAuthority("ROLE_USER")
- Use roles for high-level grouping of permissions

Configuring Users with Roles/Authorities

In-Memory Configuration Example:

```
@Bean
public InMemoryUserDetailsManager userDetailsService() {
   UserDetails user = User.withDefaultPasswordEncoder()
      .username("user")
      .password("password")
      .roles("USER")
      .build();
   return new InMemoryUserDetailsManager(user);
}
```

```java
@RestController
@RequestMapping("/auth")
public class AuthController {
    private AuthenticationManager authManager;
    private JwtUtil jwtUtil;

    public AuthController(AuthenticationManager authManager, JwtUtil jwtUtil) {
        this.authManager = authManager;
        this.jwtUtil = jwtUtil;
    }

    @PostMapping("/login")
    public ResponseEntity<?> login(@RequestBody AuthRequest req) {
        authManager.authenticate(
                new UsernamePasswordAuthenticationToken(req.getUsername(), req.getPassword()));
        String token = jwtUtil.generateToken(req.getUsername());
        return ResponseEntity.ok(new AuthResponse(token));
    }
}
```

```java
public class AuthRequest {

    private String username;
    private String password;

    public AuthRequest() {
    }

    public AuthRequest(String username, String password) {
        this.username = username;
        this.password = password;
    }

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

```java
public class AuthResponse {

    private String token;

    public AuthResponse() {
    }

    public AuthResponse(String token) {
        this.token = token;
    }

    public String getToken() {
        return token;
    }

    public void setToken(String token) {
        this.token = token;
    }
}
```

Best Practices & Conclusion

- Prefer expression-based annotations for flexibility
- Keep authority definitions centralized
- Use principle of least privilege
- Regularly review and update roles and permissions

Enabling Method Security

In your configuration class, enable method security:

- @EnableGlobalMethodSecurity(securedEnabled = true, prePostEnabled = true, jsr250Enabled = true)
- Allows use of @Secured, @PreAuthorize, @RolesAllowed, etc.

@Secured

- Secures methods based on roles
- Example:

  ```
  @Secured("ROLE_ADMIN")
  public void adminOnlyMethod() { ... }
  ```

- Uses role prefix 'ROLE_' by default

@RolesAllowed

- JSR-250 annotation, similar to @Secured
- Example:

```
@RolesAllowed({"ROLE_USER", "ROLE_ADMIN"})
public void userOrAdminMethod() { ... }
```

- Requires jsr250Enabled = true

@PreAuthorize / @PostAuthorize

- Expression-based annotations for fine-grained control
- @PreAuthorize("hasRole('ADMIN') and #id == principal.id")

  Checks before method execution
- @PostAuthorize("returnObject.owner == principal.username")

  Checks after method execution

@PreFilter / @PostFilter

- Filter collections before or after method invocation
- @PreFilter("filterObject.owner == principal.username")

  Filters input collections
- @PostFilter("filterObject.public == true")

  Filters returned collections

Defining Roles vs Authorities

- Roles are a type of authority prefixed with 'ROLE_'
- Authorities (GrantedAuthority) represent permissions
- SimpleGrantedAuthority example:

  new SimpleGrantedAuthority("ROLE_USER")
- Use roles for high-level grouping of permissions

Ex 04

- Implement Security configuration with 3 users: 1 with role ROLE_CUSTOMER, 1 with ROLE_ADMIN and one with no roles
- Anyone who is logged in can see all the Pizzas
- Only users with ROLE_CUSTOMER can see all Orders
- Only users with ROLE_ADMIN can save a new Pizza

CORS & CSRF in Spring Security

Understanding Cross-Origin Resource Sharing and CSRF Protection

Agenda

- What is CORS?
- How CORS Works
- Configuring CORS in Spring
- What is CSRF?
- How CSRF Attacks Work
- CSRF Protection in Spring Security
- Best Practices & Examples

What is CORS?

- Stands for Cross-Origin Resource Sharing
- Browser security feature to control requests between different origins
- An origin is combination of protocol, domain, and port (e.g., https://api.example.com)
- Default same-origin policy blocks cross-origin requests

How CORS Works

- Client (browser) sends a request with Origin header
- Server responds with Access-Control-Allow-Origin header
- Preflight requests (OPTIONS) for complex requests
- Other headers: Access-Control-Allow-Methods, Access-Control-Allow-Headers
- Browser enforces rules based on these response headers

Configuring CORS in Spring

Method 1: Using @CrossOrigin

```
@CrossOrigin(origins = "http://example.com")
public class MyController { ... }
```

Method 2: Global Configuration

```
@Bean
public CorsConfigurationSource corsConfigurationSource() {
    CorsConfiguration config = new CorsConfiguration();
    config.setAllowedOrigins(Arrays.asList("http://example.com"));
    config.setAllowedMethods(Arrays.asList("GET","POST","PUT","DELETE"));
    config.setAllowedHeaders(Arrays.asList("*"));
    UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
    source.registerCorsConfiguration("/**", config);
    return source;
}
```

Method 3: Spring Security Configuration

```
http.cors().and()...
// Ensure corsConfigurationSource bean is picked up
```

CORS Configuration

Use a SecurityFilterChain bean instead of WebSecurityConfigurerAdapter

```
@Configuration
public class SecurityConfig {

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http
         .cors(Customizer.withDefaults()) // permit CORS
         .csrf().disable() // adjust as needed
         .authorizeHttpRequests(auth -> auth
            .anyRequest().authenticated()
         )
         .httpBasic(Customizer.withDefaults());
      return http.build();
   }

   @Bean
   public CorsConfigurationSource corsConfigurationSource() {
      CorsConfiguration config = new CorsConfiguration();
      config.setAllowedOrigins(List.of("https://frontend.example.com"));
      config.setAllowedMethods(List.of("GET", "POST", "PUT", "DELETE"));
      config.setAllowedHeaders(List.of("Authorization", "Content-Type"));
      config.setAllowCredentials(true);
      UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfigurationSource();
      source.registerCorsConfiguration("/**", config);
      return source;
   }
}
```

What is CSRF?

- Cross-Site Request Forgery (CSRF)
- Attack where a malicious site causes a user's browser to perform unwanted requests
- Relies on browsers sending stored credentials (cookies, basic auth) automatically
- Particularly relevant in session-based authentication schemes

Session-Based CSRF Explained

1. User logs into web app (Site A) -> Session cookie stored
2. Attacker’s site (Site B) crafts a form POST to Site A
3. Victim visits Site B while still authenticated to Site A
4. Browser includes Site A cookie with malicious request to Site A
5. Site A performs sensitive action, believing request is from authenticated user

→ No CSRF token means the server cannot distinguish legitimate vs forged request

CSRF in Modern Spring Security

- CSRF protection enabled by default for stateful sessions
- Relies on CsrfTokenRepository (e.g., HttpSessionCsrfTokenRepository or CookieCsrfTokenRepository)
- Generates a unique token per user session
- Token must be submitted with each state-changing request (POST, PUT, DELETE, PATCH)
- Spring Security validates token against stored value in session or cookie

SecurityFilterChain Configuration

Use SecurityFilterChain with HttpSecurity to configure CSRF:

```
@Configuration
public class SecurityConfig {

   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
      http
         .cors(Customizer.withDefaults())
         .csrf(csrf -> csrf
            .csrfTokenRepository(
               CookieCsrfTokenRepository.withHttpOnlyFalse()
            )
         )
         .authorizeHttpRequests(auth -> auth
            .antMatchers("/public/**").permitAll()
            .anyRequest().authenticated()
         )
         .formLogin(Customizer.withDefaults());
      return http.build();
   }
}
```

- `CookieCsrfTokenRepository` stores token in cookie accessible to JavaScript for AJAX

CSRF Token Repository Options

1. HttpSessionCsrfTokenRepository
   - Stores token in HTTP session (server-side)
   - Default if no repository specified
2. CookieCsrfTokenRepository
   - Stores token in a cookie (client-side, HttpOnly optional)
   - Facilitates SPAs or JS-heavy frontends

CSRF for REST APIs

- Stateless APIs often use token-based auth (JWT) instead of session cookies
- With JWT, CSRF risk is reduced if JWT sent in Authorization header
- If using cookies for JWT, still enable CSRF protection or use SameSite cookies
- Example: Disabling CSRF for token-based API endpoints

```
http
    .csrf(csrf -> csrf
        .ignoringAntMatchers("/api/**")
    )
    .authorizeHttpRequests(auth -> auth
        .antMatchers("/api/auth/**").permitAll()
        .anyRequest().authenticated()
    );
```

OAuth2 & Spring Security

Integrating OAuth2 in Modern Spring Applications

Agenda

- OAuth2 Overview
- Key Actors & Grant Types
- Spring Security OAuth2 Client (2025)
- Spring Security OAuth2 Resource Server
- Spring Authorization Server Basics
- Configuration Examples
- Securing Endpoints with Scopes
- Best Practices & 2025 Trends

OAuth2 Overview

- OAuth2 is an authorization framework
- Delegates user authentication to an Authorization Server
- Clients obtain an access token to access protected resources
- Focus on secure API access and delegation

Key Actors & Grant Types

Actors:

- Resource Owner (User)
- Client (App requesting access)
- Authorization Server (Issues tokens)
- Resource Server (API serving data)

Common Grant Types:

- Authorization Code (with PKCE for SPAs)
- Client Credentials (machine-to-machine)
- Refresh Token (obtain new access tokens)
- Password (legacy, discouraged)

OAuth2 Client in Spring Security

Use Spring Boot 3.x and Spring Security 6.x:

- Include dependency: spring-boot-starter-oauth2-client
- Configure clients in application.yml
- Spring Security auto-configures the OAuth2Login filter
- Example SecurityFilterChain:

```
@Configuration
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated()
            )
            .oauth2Login(Customizer.withDefaults());
        return http.build();
    }
}
```

Client Registration (application.yml)

Spring Boot 3.x YAML configuration:

```
spring:
  security:
    oauth2:
      client:
        registration:
          google:
            client-id: YOUR_CLIENT_ID
            client-secret: YOUR_CLIENT_SECRET
            scope: openid, profile, email
            redirect-uri: "{baseUrl}/login/oauth2/code/{registrationId}"
            client-authentication-method: client_secret_basic
            authorization-grant-type: authorization_code
        provider:
          google:
            issuer-uri: https://accounts.google.com
```

OAuth2 Resource Server

Use JWT-based tokens to secure APIs:

- Include dependency: spring-boot-starter-oauth2-resource-server
- Configure JWT decoder and issuers
- Example SecurityFilterChain:

```
@Configuration
public class ResourceServerConfig {

    @Bean
    public SecurityFilterChain resourceFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .antMatchers("/public/**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .jwt(Customizer.withDefaults())
            );
        return http.build();
    }

    @Bean
    JwtDecoder jwtDecoder() {
        return NimbusJwtDecoder.withJwkSetUri("https://auth-server.example.com/.well-
known/jwks.json").build();
    }
}
```

Spring Authorization Server Basics

Spring Authorization Server 1.x (2025):

- Official project replacing old auth server modules
- Built on Spring Security’s Authorization Server support
- Supports OAuth2, OIDC, custom grant types
- Provides UI for consent and login pages

Authorization Server Configuration

Example configuration in Spring Boot 3.x:

```
@Configuration
public class AuthServerConfig {

   @Bean
   public RegisteredClientRepository registeredClientRepository() {
      RegisteredClient registeredClient = RegisteredClient.withId(UUID.randomUUID().toString())
         .clientId("app-client")
         .clientSecret("{noop}secret")
         .clientAuthenticationMethod(ClientAuthenticationMethod.CLIENT_SECRET_BASIC)
         .authorizationGrantType(AuthorizationGrantType.AUTHORIZATION_CODE)
         .redirectUri("https://client.example.com/login/oauth2/code/app-client")
         .scope(OidcScopes.OPENID)
         .tokenSettings(TokenSettings.builder()
            .accessTokenTimeToLive(Duration.ofHours(1))
            .refreshTokenTimeToLive(Duration.ofDays(7))
            .build())
         .build();

      return new InMemoryRegisteredClientRepository(registeredClient);
   }

   @Bean
   public AuthorizationServerSettings authorizationServerSettings() {
      return AuthorizationServerSettings.builder().issuer("https://auth-server.example.com").build();
   }

   @Bean
   public SecurityFilterChain authServerSecurityFilterChain(HttpSecurity http) throws Exception {
      OAuth2AuthorizationServerConfiguration.applyDefaultSecurity(http);
      return http.formLogin(Customizer.withDefaults()).build();
   }
}
```

Securing Endpoints with Scopes & Authorities

Example of checking scopes in resource server:

```
@RestController
@RequestMapping("/api")
public class ApiController {

    @GetMapping("/data")
    @PreAuthorize("hasAuthority('SCOPE_read:data')")
    public ResponseEntity<String> getData() {
        return ResponseEntity.ok("Protected data");
    }

    @PostMapping("/data")
    @PreAuthorize("hasAuthority('SCOPE_write:data')")
    public ResponseEntity<String> postData() {
        return ResponseEntity.ok("Data created");
    }
}
```

Token Introspection & Revocation

OAuth2 Authorization Server provides endpoints:

- /oauth2/introspect - validate active tokens
- /oauth2/revoke - revoke tokens
- Spring Authorization Server auto-configures these if enabled
- Resource Server can use introspection for opaque tokens:

Opaque Token Support

Example Resource Server config for introspection:

```
@Configuration
public class ResourceServerConfig {

    @Bean
    public SecurityFilterChain resourceFilterChain(HttpSecurity http) throws Exception {
        http
            .authorizeHttpRequests(auth -> auth
                .anyRequest().authenticated()
            )
            .oauth2ResourceServer(oauth2 -> oauth2
                .opaqueToken(opaque -> opaque
                    .introspectionUri("https://auth-server.example.com/oauth2/introspect")
                    .introspectionClientCredentials("app-client", "secret")
                )
            );
        return http.build();
    }
}
```

Best Practices

- Prefer Authorization Code with PKCE for web & mobile clients
- Leverage JWT for stateless resource servers
- Use Spring Authorization Server for custom needs
- Secure admin endpoints with fine-grained roles/scopes
- Monitor token usage and rotate keys regularly

Session-based Authentication

- Stateful: server stores session data
- JSESSIONID cookie tracks sessions
- Built-in support via HttpSession & SecurityContextPersistenceFilter