Real Time Java Coding Standards and Best Practices

This document outlines standardized **Java coding conventions** and **best practices** to be followed in **real-time, production-grade projects**. It serves as a guideline for developers to write clean, maintainable, scalable, and secure code. It covers naming conventions, project structure, code formatting, exception handling, design principles. Adhering to these standards will ensure code consistency across the team, reduce technical debt, and improve collaboration in real-world enterprise applications.

🔹 1. Project Structure (Modular, Layered Architecture)

Structure your project by layers:

```
📁 projectname
├── 📁 src
│   ├── 📁 main
│   │   ├── 📁 java
│   │   │   └── 📁 com.companyname.projectname
│   │   │       ├── 📁 controller    → REST controllers
│   │   │       ├── 📁 service       → Business logic
│   │   │       ├── 📁 repository    → JPA repositories
│   │   │       ├── 📁 model         → Entities (JPA)
│   │   │       ├── 📁 dto           → Data Transfer Objects
│   │   │       ├── 📁 mapper        → MapStruct or custom mappers
│   │   │       ├── 📁 config        → Spring configs (e.g., `SecurityConfig`)
│   │   │       ├── 📁 exception     → Custom exceptions & handlers
│   │   │       ├── 📁 util          → Utility/helper classes
│   │   │       └── 📁 security      → Authentication & authorization
│   │   └── 📁 resources
│   │       ├── 📄 application.yml   → Main config file
│   │       ├── 📁 static            → Static assets (JS, CSS, etc.)
│   │       ├── 📁 templates         → Thymeleaf/FreeMarker templates
│   │       ├── 📁 i18n              → Localization files
│   │       └── 📁 log               → Log file storage (if configured
│   │                                    to write here)
│   └── 📁 test
│       └── 📁 java
│           └── 📁 com.companyname.projectname
│               ├── 📁 controller
│               ├── 📁 service
│               └── ... (tests matching main structure)
├── 📁 target                        → Compiled classes and build artifacts
│                                       (generated)
├── 📄 pom.xml                       → Maven build configuration
├── 📄 .gitignore                    → Git ignored files list
└── 📄 README.md                     → Project overview & instructions
```

🔹 **2. Package Naming**

- **Standard**: All **lowercase**, **dot-separated, hierarchical**.
- **Pattern**: **com.company.project.module**
- ✦ **Example**:

```
com.example.crm.user.service
com.example.crm.auth.controller
Com.example.crm.common.exception
```

🔹 **3. Class Naming**

- **Standard**: **PascalCase** (UpperCamelCase)
- **Suffixed types**:
  - → **Controller** : UserController
  - → **Service** : UserService, AuthServiceImpl
  - → **DTOs** : UserRequestDto, UserResponseDto
  - → **Entities** : User, Role
  - → **Repositories**: UserRepository
- ✦ **Example**:

```
public class UserServiceImpl implements UserService
```

🔹 **4. Method Naming**

Standard: **camelCase** (verbs first, descriptive)

Common patterns:

- → **getAllUsers()**
- → **findUserById()**
- → **saveUser()**
- → **deleteUserById()**
- → **assignRolesToUser()**

🔹 **5. Variable Naming**

- **Standard**: **camelCase**, meaningful names (no abbreviations)
- ✦ **Examples**:
  - → String **userName**;
  - → Long **userId**;
  - → List<Role> **assignedRoles**;
  - → UserResponseDto **userResponse**;

🔹 **6. Constant Naming (e.g., Roles like ROLE_AGENT)**

Standard: **UPPER_SNAKE_CASE**, usually in enums or constants classes.

Defined in Enum: Best practice in Spring Security

✦ **Example :**

```
public enum ERole {
    ROLE_USER,
    ROLE_AGENT,
    ROLE_ADMIN,
```

```
        ROLE_SUPERVISOR
    }
    Or in a constants class:

    public class SecurityConstants {
        public static final String ROLE_AGENT = "ROLE_AGENT";
    }
```

🔹 **7. Enum Naming**

Enum name: PascalCase

Enum constants: UPPER_SNAKE_CASE

✦ **Example:**

```
    public enum Status {
        ACTIVE,
        INACTIVE,
        BLOCKED
    }
```

🔹 **8. Configuration Naming**

Configuration classes: SecurityConfig, WebMvcConfig

Properties: application.yml

```
    server:
    port: 8080

    spring:
    datasource:
        url: jdbc:mysql://localhost:3306/mydb
```

🔹 **9. Code Readability & Formatting**

- **Indentation**: 4 spaces (no tabs)
- **Line length**: Max 120 characters
- **Braces**: Always use {} even for single-line if, for, etc.
- **Blank lines**: Separate logical blocks
- **Imports**: No wildcard imports (`import java.util.*;` ❌)
- Use an auto-formatter (like **IntelliJ default** or Google Java Style).

🔹 **10. Exception Handling (Robust & Predictable)**

- Use custom exceptions instead of RuntimeException.
- Use `@ControllerAdvice` for global exception handling.
- Return meaningful error messages to clients:

✦ **Example :**

- Creating custom exception **RoleNotFoundException**

```
import org.springframework.http.HttpStatus;
import org.springframework.web.bind.annotation.ResponseStatus;

@ResponseStatus(HttpStatus.NOT_FOUND)
public class RoleNotFoundException extends RuntimeException {
      public RoleNotFoundException(String message) { super(message); }
      }
 }
```

- If an Exception occurs, we should handle it **Globally**.

```
@Slf4j // logging purpose
@RestControllerAdvice
@AllArgsConstructor
public class GlobalExceptionController {

   @ExceptionHandler(RoleNotFoundException.class)
   public ResponseEntity<String> handleRoleNotFoundException
                                 (RoleNotFoundException ex)
   {
          return ResponseEntity.status(HttpStatus.NOT_FOUND)
                 .body(ex.getMessage());
   }
}
```

🔹 **11. Logging (Structured, Not System.out)**

- Use **SLF4J (@Slf4j)** instead of System.out.
- Log only meaningful data (**not passwords, sensitive info**).
- Use placeholders instead of string concatenation:

✦ **Example :**

```
log.info("User created: id={}, name={}", user.getId(), user.getName());
```

🔹 **12. Validation (Always Validate Input)**

- Use **javax.validation (@NotNull, @Size, etc.)** on DTOs

✦ **Example:**

```
 public class UserRequestDto {
      @NotBlank
      private String username;
      @Email
      private String email;
}
```

- Use `@Valid` in controller methods:

✦ **Example**

```
public ResponseEntity<?> createUser(@Valid
                                    @RequestBody UserRequestDto request) { ... }
```

🔹 **13. Security (Spring Security Best Practices)**

- Never hardcode **secrets.**
- Use **BCryptPasswordEncoder** for passwords.
- Store roles as **ROLE_ADMIN, ROLE_USER,** etc.
- Avoid exposing IDs (use **UUIDs or obscured values** for public APIs).

🔹 **14. DTOs vs Entities (Separation of Concerns)**

- Never expose **JPA entities directly** to controllers.
- Use **DTOs** to map request/response data.
- Use **MapStruct or ModelMapper** for clean mapping.

Real-world example :

✦ **1.JPA Entity (User.java)**

```
@Entity
@Table(name = "users")
public class User {

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

      private String username;
      private String email;

      @Enumerated(EnumType.STRING)
      private Status status;
            // Getters and setters (or use Lombok)
}
```

✦ **2.DTOs**

✦ UserRequestDto.java – for ***incoming requests***

```
      public class UserRequestDto {

            @NotBlank
            private String username;
```

```
        @Email
        private String email;
    }
```

✦ UserResponseDto.java – for outgoing responses

```
    public class UserResponseDto {

        private Long id;
        private String username;
        private String email;
        private String status;
    }
```

✦ 3.MapStruct Mapper(UserMapper.java)

```
@Mapper(componentModel ="spring")//tells Spring to manage this as a bean.
public interface UserMapper {
    User toEntity(UserRequestDto dto);
    UserResponseDto toDto(User user);
}
```

✦ 4.Service Layer (UserService.java)

```
@Service
@RequiredArgsConstructor
public class UserService {

    private final UserRepository userRepository;
    private final UserMapper userMapper;

    public UserResponseDto createUser(UserRequestDto dto) {
        User user = userMapper.toEntity(dto);
        user.setStatus(Status.ACTIVE);
        user = userRepository.save(user);
        return userMapper.toDto(user);
    }

    public List<UserResponseDto> getAllUsers() {
        return userRepository.findAll()
                .stream()
                .map(userMapper::toDto)
                .collect(Collectors.toList());
    }
```

```
    }
```

✦ 5. Controller Layer (UserController.java)

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

    private final UserService userService;

    @PostMapping
    public ResponseEntity<UserResponseDto> createUser(@Valid @RequestBody
    userRequestDto dto) {
            return  ResponseEntity.status(HttpStatus.CREATED).
                                body(userService.createUser(dto));
    }

    @GetMapping
    public ResponseEntity<List<UserResponseDto>> getAllUsers() {
            return ResponseEntity.ok(userService.getAllUsers());
    }
}
```

🔹 15. Immutability & Lombok Usage

- Use `@Value or final` for immutable objects.
- Use Lombok sparingly and only in DTOs or model layers:
- @Data, @Builder, @Getter, @Setter, @NoArgsConstructor, @AllArgsConstructor

✦ **Example**: DTO Using Lombok & Immutability

UserResponseDto.java

```
import lombok.Builder;
import lombok.Value;

@Value
@Builder
public class UserResponseDto {
    Long id;
    String username;
    String email;
    String status;
}
```

➤ **Note** : @Value makes the class:

• Final (cannot be extended)

- All fields are private and final
- Adds getters, constructor, equals(), hashCode(), and toString()
- No setters – truly immutable

✦ **Example**: Entity (Mutable, Use Lombok Carefully)

User.java

```java
@Entity
@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
public class User {

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

    private String username;
    private String email;

    @Enumerated(EnumType.STRING)
    private Status status;
}
```

➢ **Note** :⚠️ Avoid @Data on JPA entities because:

- It adds toString(), which can cause circular reference issues in bi-directional relationships.
- Adds equals() and hashCode(), which can break persistence behavior (especially with proxies).

✦ **Example**: Request DTO with @Builder (Mutable or Immutable)

UserRequestDto.java

```java
import lombok.Builder;
import lombok.Data;

@Data
@Builder
public class UserRequestDto {
    private String username;
    private String email;
}
```

➢ **Note** : Use @Data + @Builder for flexible and clean object creation in mutable DTOs.

✦ **Example** : Anti-pattern: @Data on Entity

❌ Not recommended!

```
@Data
@Entity
public class Role {
      @Id
      private Long id;
      private String name;
}
```

➤ **Note** : Why bad ?

- Adds toString(), equals(), hashCode() – can break Hibernate behavior or lead to infinite recursion.
- Less control over behavior.

🔹 **16. Testing (Unit + Integration)**

- Unit test services with @MockBean or Mockito.
- Use @WebMvcTest for controller tests.
- Use Testcontainers or H2 for integration tests.

🔹 **17. Build & Dependency Management**

- Use Maven or Gradle with proper dependency scopes.
- Keep dependencies updated.
- Don't overuse implementation – prefer specific scopes like testImplementation.

🔹 **18. API Documentation**

Use Swagger/OpenAPI:

- @Operation(summary = "Get all users")
- @ApiResponse(responseCode = "200", description = "OK")
- Swagger UI makes API testing easy and encourages good documentation.

🔹 **19. Code Quality Tools**

- SonarLint, Checkstyle, PMD, , and SpotBugs for static analysis
- Enable in your IDE or CI pipeline

THANK YOU