Code readablilty

Naming covention

SOLID Desing parttern.

Exception handling.

Null check

unit test cases

code duplication

Proper collection

logging practices

performance awareness

- Unnecessary database queries (`N+1` problem).
- Large object creation in loops.
- **Blocking calls in asynchronous methods.**

Security Checks

- Validate that sensitive endpoints are secured.
- Check for **SQL Injection**, **XSS**, and **CSRF** protections.
- **Make sure passwords are never logged or exposed.**

Code Review Best Practices for Java

✅ 1️⃣ Code Readability

- Code should be **clean, simple, and easy to understand**.
- Naming should be **clear and descriptive**:

```
java
CopyEdit
// Bad
int a = calculate(5);

// Good
int discountedPrice = calculateDiscountedPrice(5);
```

- Avoid hardcoded values — use constants.

✅ 2️⃣ Follow Java Naming Conventions

- Classes: `CamelCase` (Pascal Case).

- Methods and variables: `camelCase`.
- Constants: `UPPER_SNAKE_CASE`.

✅ 3️⃣ SOLID Principles

- Make sure the code follows **SOLID** principles:
  - `S` - Single Responsibility
  - `O` - Open/Closed
  - `L` - Liskov Substitution
  - `I` - Interface Segregation
  - `D` - Dependency Inversion

For example, check if a class is doing **too much** — break it into smaller services if necessary.

✅ 4️⃣ Error Handling

- Validate that:
  - Exceptions are being properly caught.
  - Custom exceptions are used meaningfully.
  - `try-catch` blocks aren't swallowing errors silently.
  - Logging is done in catch blocks.

Example:

```
java
CopyEdit
try {
    userService.saveUser(user);
} catch (DataIntegrityViolationException e) {
    log.error("Error saving user", e);
    throw new UserSaveException("User could not be saved");
}
```

✅ 5️⃣ Null Safety

- Use `Optional` where appropriate.
- Avoid `NullPointerException` risks.
- Prefer:

  ```
  java
  CopyEdit
  Optional.ofNullable(user).ifPresent(u -> process(u));
  ```

instead of raw null checks.

✅ 6️⃣ Unit Tests

- Code should be accompanied by meaningful **unit tests**.
- Coverage is important, but quality is more important than %.
- Prefer:

  ```
  java
  CopyEdit
  assertEquals(expectedValue, actualValue);
  ```

  over printing values for debugging.

✅ 7️⃣ Code Duplication

- Identify and remove duplicate logic.
- Extract common code into reusable private methods or utility classes.

✅ 8️⃣ Security Checks

- Validate that sensitive endpoints are secured.
- Check for **SQL Injection**, **XSS**, and **CSRF** protections.
- Make sure passwords are never logged or exposed.

✅ 9️⃣ Logging Best Practices

- Log meaningful, structured messages.
- Use proper log levels: `INFO`, `DEBUG`, `WARN`, `ERROR`.
- Never log sensitive information like passwords or tokens.

✅ 🔥 1️⃣0️⃣ Performance Awareness

- Watch out for:
  - Unnecessary database queries (`N+1` problem).
  - Large object creation in loops.
  - Blocking calls in asynchronous methods.

🏆 BONUS: Spring Boot Specific

- `@Autowired` should favor **constructor injection** over field injection.
- Use `@Value` and `@ConfigurationProperties` for config values.
- Avoid business logic in Controllers — push it to Service layer.

💡 Code Review Checklist — Quick Summary

Topic	Check
✅ Readability	Clear, descriptive names.
✅ Java Conventions	Correct naming, style guide.
✅ SOLID Principles	Good class responsibility split.
✅ Error Handling	Proper exception use.
✅ Null Safety	Use `Optional` where needed.
✅ Tests	Unit & integration tests exist.
✅ Duplicates	No copy-paste logic.
✅ Security	Validate auth, input, logs.
✅ Logging	Clean logs, no sensitive info.
✅ Performance	Watch for bottlenecks.

✅ Java Code Review Checklist

💡 1️⃣ Code Readability & Clean Code

- Descriptive and meaningful variable, method, and class names.
- Consistent indentation and formatting.
- Avoid deep nesting (keep methods small and clean).
- Avoid magic numbers — use constants or enums.

💡 2️⃣ Java Coding Conventions

- Class names: `PascalCase`.

- Method and variable names: `camelCase`.
- Constants: `UPPER_SNAKE_CASE`.
- Packages: `lowercase.without.underscore`.

💡 3️⃣ SOLID & OOP Principles

- Single Responsibility Principle: Each class has one job.
- Open/Closed Principle: Code is open for extension, closed for modification.
- Interface Segregation: Interfaces are client-specific.
- Dependency Injection used appropriately.
- Reuse through composition over inheritance.

💡 4️⃣ Error Handling

- Exceptions are caught and handled properly.
- Custom exceptions used where applicable.
- Meaningful error messages and logging.
- Avoid empty catch blocks.

💡 5️⃣ Null Safety

- Proper null checks or usage of `Optional`.
- Defensive programming against `NullPointerException`.
- External input is validated.

💡 6️⃣ Unit & Integration Tests

- Tests cover business logic, edge cases, and failure paths.
- Test names are clear and descriptive.
- Avoid hardcoded data in tests.
- Proper use of mocks and stubs.

💡 7️⃣ Code Duplication

- No repeated code blocks.
- Common logic is refactored into reusable methods or components.

💡 8️⃣ Security

- User inputs are validated and sanitized.
- Sensitive data is not logged.
- Proper authentication and authorization checks.
- CSRF, XSS, SQL Injection protections are in place.
- Secrets (keys, passwords) are never hardcoded.

💡 9️⃣ Logging

- Logs provide meaningful, structured information.
- Proper use of log levels (`DEBUG`, `INFO`, `WARN`, `ERROR`).
- Sensitive data is not printed in logs.
- Avoid `System.out.println()` in production code.

💡 🔄 1️⃣0️⃣ Performance & Scalability

- Avoid unnecessary object creation.
- Efficient use of collections and loops.
- Database queries optimized to prevent N+1 problems.
- Asynchronous processing is used where applicable.

💡 1️⃣1️⃣ Dependency Management

- Only necessary dependencies are included in `pom.xml` or `build.gradle`.
- Dependencies are up-to-date and secure.
- Avoid conflicting versions.

💡 1️⃣2️⃣ Microservice Specific (Spring Boot)

- Controller is thin; business logic in services.
- Use of constructor injection instead of field injection.
- Proper REST API status codes.
- Configuration values externalized using `@Value` or `@ConfigurationProperties`.
- Health checks and metrics enabled (`/actuator`).