THE BEST JAVA CODING PRACTICES

To Boost Microservices Productivity

MUAATH BIN ALI

Best Java Coding Practices to Boost Microservices Productivity

mezocde

This book is for sale at
http://leanpub.com/best-java-microservices-coding-practices

This version was published on 2024-07-05

Contents

Introduction

Welcome to Best Java Coding Practices to Boost Microservices Productivity a concise and practical guide designed to help Java developers optimize their microservices.

Microservices have become increasingly popular in recent years due to their ability to break down complex systems into smaller, more manageable components. However, developing microservices in Java requires a deep understanding of best practices to ensure maximum productivity and efficiency.

This mini-book covers five critical areas that every Java developer should master to create robust and efficient microservices.

Best Practices for API Design

APIs are the backbone of microservices communication. This section will guide you through the principles of designing clean, intuitive, and well-documented APIs that facilitate seamless integration between services.

Best Practices for Handling Null

Null values can be a major source of bugs and performance issues in Java applications. Learn how to effectively handle null values and avoid common pitfalls that can lead to unexpected behavior.

Best Practices for Writing Logs

Logging is crucial for monitoring and debugging microservices. Discover best practices for writing informative, structured logs that

provide valuable insights into your system's behavior.

Best Practices for Handling Exceptions

Exceptions can disrupt the flow of your microservices and impact overall system stability. Master the art of exception handling to gracefully manage errors and maintain a smooth user experience.

Best Practices for Database Handling

Efficient database management is vital for microservices performance. Learn how to optimize your database interactions, leverage connection pooling, and ensure data consistency across services.

By following the best practices outlined in this mini book, Java developers can significantly boost their microservices productivity, write cleaner code, and build more resilient systems. Whether you're a seasoned developer or just starting your microservices journey, this resource will provide you with the knowledge and tools necessary to take your skills to the next level.

Chapter 2 - Best Practices for API Design

Understanding The best practices of API Design in Java is crucial for Java developers aiming to create robust, scalable Microservices. In this section, we dive into 11 essential best practices for API development in Java, which will guide you toward professional and efficient API creation. By exploring Java-specific examples, you'll learn to distinguish between effective and inefficient approaches to API development.

2.1 - Adherence to RESTful Principles

RESTful architecture is characterized by statelessness, cacheability, a uniform interface, and a client-server architecture. When APIs follow these principles, they ensure predictable and standardized interactions.

Good Example

A GET request to retrieve a resource by ID.

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

    @GetMapping("/{id}")
    public ResponseEntity<User> getUserById(@PathVariable\
 Long id) {
        User user = userService.findById(id);
        if (user == null) {
            return ResponseEntity.notFound().build();
        }
        return ResponseEntity.ok(user);
    }
}
```

Avoid Example

Performing an update action using a GET request goes against the principle that GET requests should be safe and idempotent.

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

    @GetMapping("/forceUpdateEmail/{id}")
    public ResponseEntity<Void> forceUpdateUserEmail(
           @PathVariable Long id,
           @RequestParam String email) {
        // This is bad, GET should not change state.
        userService.updateEmail(id, email);
        return ResponseEntity.ok().build();
    }
}
```

Avoid Example

Performing an update action using a GET request goes against the principle that GET requests should be safe and idempotent.

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

    @GetMapping("/forceUpdateEmail/{id}")
    public ResponseEntity<Void> forceUpdateUserEmail(@Pat\
hVariable Long id,
                                                     @Req\
uestParam String email) {
        userService.updateEmail(id, email); // This is ba\
d, GET should not change state.
        return ResponseEntity.ok().build();
    }
}
```

2.2 - Utilization of Meaningful HTTP Status Codes

HTTP status codes are a critical component of client-server communication, providing immediate insights into the outcome of an HTTP request.

Good Example

Using 201 Created for successful resource creation.

```
    @PostMapping("/users")
    public ResponseEntity<User> createUser(@RequestBody U\
ser user) {
        User savedUser = userService.save(user);
        return new ResponseEntity<>(savedUser, HttpStatus\
.CREATED);
    }
```

Avoid Example

Returning 200 OK for a request that fails validation, where a client error code (4xx) would be more appropriate.

```
@PostMapping("/users")
    public ResponseEntity<User> createUser(@RequestBody U\
ser user) {
        if (!isValidUser(user)) {
            return new ResponseEntity<>(HttpStatus.OK); /\
/ This is bad, should be 4xx error.
        }
        User savedUser = userService.save(user);
        return new ResponseEntity<>(savedUser, HttpStatus\
.CREATED);
    }
```

2.3 - Strategic API Versioning

Versioning APIs is essential to manage changes over time without disrupting existing clients. This practice allows developers to introduce changes or deprecate API versions systematically.

Good Example Explicitly versioning the API in the URI.

```
@RestController
@RequestMapping("/api/v1/users")
public class UserController {
// RESTful API actions for version 1
}
```

Avoid Example Lack of versioning, leading to potential breaking changes for clients.

```
@RestController
@RequestMapping("/users")
public class UserController {
// API actions with no versioning.
}
```

2.4 - Graceful Exception Handling

Robust error handling enhances API usability by providing clear, actionable information when something goes wrong.

Good Example A specific exception handler for a resource not found scenario, returning a 404 Not Found.

```
    @ExceptionHandler(UserNotFoundException.class)
    public ResponseEntity<Object> handleUserNotFound(User\
NotFoundException ex) {
        return new ResponseEntity<>(ex.getMessage(), Http\
Status.NOT_FOUND);
    }
```

Avoid Example Exposing stack traces to the client, can be a security risk and unhelpful to the client.

```
    @ExceptionHandler(Exception.class)
    public ResponseEntity<Object> handleAllExceptions(Exc\
eption ex) {
        return new ResponseEntity<>(ex.getStackTrace(), H\
ttpStatus.INTERNAL_SERVER_ERROR); // This is bad, don't e
xpose stack trace.
    }
```

2.5 - Ensuring API Security

Security is paramount, and APIs should implement appropriate authentication, authorization, and data validation to protect sensitive data.

Good Example

Incorporating authentication checks within the API endpoint.

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUserById(@PathVariable Lon\
g id) {
    if (!authService.isAuthenticated(id)) {
        return new ResponseEntity<>(HttpStatus.UNAUTHORIZ\
ED);
    }
    // Fetch and return the user
}
```

Avoid Example

Omitting security checks, leaving the API vulnerable to unauthorized access.

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUserById(@PathVariable Lon\
g id) {
    // No authentication check, this is bad.
    // Fetch and return the user
}
```

2.6 - Comprehensive API Documentation

Well-documented APIs facilitate ease of use and integration, reducing the learning curve for developers and encouraging adoption.

Good Example

Using annotation-based documentation tools like Swagger.

```
@Api(tags = "User Management")
@RestController
@RequestMapping("/api/v1/users")
public classUserController {
    // RESTful API actions with Swagger annotations for d\
ocumentation
}
```

Avoid Example

Non-existent documentation makes it difficult to discover the usability of APIs.

```
@RestController
@RequestMapping("/users")
public class UserController {
    // API actions with no comments or documentation anno\
tations
}
```

2.7 - Effective Use of Query Parameters

Query parameters should be employed for filtering, sorting, and pagination to enhance the API's flexibility and prevent the transfer of excessive data.

Good Example

API endpoints that accept query parameters for sorting and paginated responses.

```
@GetMapping("/users")
public ResponseEntity<List<User>> getUsers(
            @RequestParam Optional<String> sortBy,
            @RequestParam Optional<Integer> page,
            @RequestParam Optional<Integer> size) {
    // Logic for sorting and pagination
    return ResponseEntity.ok(userService.getSortedAndPagi\
natedUsers(
    sortBy, page, size));
}
```

Avoid Example

Endpoints that return all records without filtering or pagination, potentially overwhelming the client.

```
@GetMapping("/users")
public ResponseEntity<List<User>> getAllUsers() {
    // This is bad, as it might return too much data
    return ResponseEntity.ok(userService.findAll());
}
```

2.8 - Leveraging HTTP Caching

Caching can significantly improve performance by reducing server load and latency. It's an optimization that experts should not overlook.

Good Example

Implementing ETags and making use of conditional requests.

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUserById(@PathVariable Lon\
g id,
                                        @RequestHeader(va\
lue = "If-None-Match", required = false) String ifNoneMat
ch) {
    User user = userService.findById(id);
    String etag = user.getVersionHash();

    if (etag.equals(ifNoneMatch)) {
        return ResponseEntity.status(HttpStatus.NOT_MODIF\
IED).build();
    }

    return ResponseEntity.ok().eTag(etag).body(user);
}
```

Avoid Example

Ignoring caching mechanisms leads to unnecessary data transfer and processing.

```
@GetMapping("/users/{id}")
public ResponseEntity<User> getUserById(@PathVariable Lon\
g id) {
    // No ETag or Last-Modified header used, this is bad \
for performance
    return ResponseEntity.ok(userService.findById(id));
}
```

2.9 - Maintaining Intuitive API Design

An API should be self-explanatory, with logical resource naming, predictable endpoint behavior, and consistent design patterns.

Good Example

Clear and concise endpoints that immediately convey their functionality.

```
@PostMapping("/users")
public ResponseEntity<User> createUser(@RequestBody User \
user) {
    // Endpoint clearly indicates creation of a user
}

@GetMapping("/users/{id}")
public ResponseEntity<User> getUserById(@PathVariable Lon\
g id) {
    // The action of retrieving a user by ID is clear
}
```

Avoid Example

Confusing or convoluted endpoint paths and actions that obfuscate their purpose.

```
@PutMapping("/user-update")
public ResponseEntity<User> updateUser(@RequestBody User \
user) {
    // This is bad, as the path does not indicate a resou\
rce
}
```

2.10 - Enable Response Compression

To optimize network performance, enabling response compression is a smart move. It reduces the payload size, which can significantly decrease network latency and speed up client-server interactions.

Good Example

Configuring your web server or application to use gzip or Brotli compression for API responses.

```
// In Spring Boot, you might configure application.proper\
ties to enable response compression.

server.compression.enabled=true
server.compression.mime-types=application/json,applicatio\
n/xml,text/html,text/xml,text/plain
```

This configuration snippet tells the server to compress responses for specified MIME types.

Avoid Example

Sending large payloads without compression leads to increased load times and bandwidth usage.

```
// No configuration or code in place to handle response c\
ompression.

@GetMapping("/users")
public ResponseEntity<List<User>> getAllUsers() {
    // This could return a large JSON payload that isn't \
compressed, which is inefficient.
    return ResponseEntity.ok(userService.findAll());
}
```

2.11 - Embrace Asynchronous Operations

Asynchronous operations are essential for handling long-running tasks, such as processing large datasets or batch operations. They free up client resources and prevent timeouts for operations that take longer than the usual HTTP request-response cycle.

Good Example Using asynchronous endpoints that return a 202 Accepted status code with a location header to poll for results.

```
@PostMapping("/users/batch")
public ResponseEntity<Void> batchCreateUsers(@RequestBody\
 List<User> users) {
    CompletableFuture<Void> batchOperation = userService.\
createUsersAsync(users);
    HttpHeaders responseHeaders = new HttpHeaders();
    responseHeaders.setLocation(URI.create("/users/batch/\
status"));

    return ResponseEntity.accepted().headers(responseHead\
ers).build();
}
```

This example accepts a batch creation request and processes it asynchronously, providing a URI for the client to check the operation's status.

Avoid Example Blocking operations for batch processing that keep the client waiting indefinitely.

```
@PostMapping("/users/batch")
public ResponseEntity<List<User>> batchCreateUsers(@Reque\
stBody List<User> users) {
    // This is a synchronous operation that may take a lo\
ng time to complete.
    List<User> createdUsers = userService.createUsers(use\
rs);
    return ResponseEntity.ok(createdUsers);
}
```

This example performs a synchronous batch creation, which could lead to a timeout or a poor user experience due to the long wait time.

By incorporating response compression and embracing asynchronous operations, API developers can greatly improve performance and user experience. These practices are essential when dealing with modern web applications and services that require efficient real-time data processing and transmission.

Summary

The following guidelines help ensure your Java APIs are robust, secure, and user-friendly, enhancing both performance and developer experience.

Best Practices for Effective API Design in Java

Follow RESTful Principles

- Utilize statelessness, cacheability, and uniform interfaces.
- **Do**: Use `GET` for retrieving data.
- **Don't**: Use `GET` for actions that change state.

Use Meaningful HTTP Status Codes

- Choose status codes that accurately reflect the API response.
- **Do**: `404 Not Found` for non-existent resources.
- **Don't**: Misuse `200 OK` for errors.

Implement API Versioning

- Manage changes without breaking existing functionality.
- **Do**: Version your APIs in the URI.
- **Don't**: Deploy APIs without version control.

Handle Exceptions Gracefully

- Provide clear error messages and actionable feedback.
- **Do**: Customize error handling based on exception type.
- **Don't**: Expose stack traces to end-users.

Secure Your API

- Ensure authentication, authorization, and data validation are in place.
- **Do**: Protect endpoints using security checks.
- **Don't**: Leave APIs open to unauthorized access.

Document APIs Thoroughly

- Make integration and usage straightforward with clear documentation.
- **Do**: Use tools like Swagger.
- **Don't**: Leave APIs undocumented.

Utilize Query Parameters Effectively

- Support filtering, sorting, and pagination through query parameters.
- **Do**: Allow extensive query options.
- **Don't**: Overwhelm clients with unfiltered data.

Enable HTTP Caching

- Improve performance with caching strategies like ETags.
- **Do**: Implement caching to reduce load.
- **Don't**: Overload clients and networks with unnecessary data re-fetching.

Ensure Intuitive Design

- Maintain logical, predictable API endpoints.
- **Do**: Design endpoints that clearly indicate their function.
- **Don't**: Confuse users with misleading paths.

Support Asynchronous Operations

- Facilitate handling of long-running tasks without blocking clients.
- **Do**: Provide endpoints for status checks on asynchronous requests.
- **Don't**: Force clients to wait for prolonged operations.

Best Practices for Handling Null

As a Java engineer, dealing with null references can be quite challenging. A mistake can cause the entire application to crash due to the dreaded `NullPointerException`, especially while dealing with third parties and Microservices architecture. In this section we will provide a roadmap for null-safe programming in Java, enabling you to write more stable and bug-free code.

Understanding Null in Java

Before we delve into handling nulls, let's understand the implications. Null references were introduced in Java to represent the absence of a value. However, Tony Hoare, the inventor of the null reference, has called it his "billion-dollar mistake" due to the number of errors it can cause in programming.

Strategies for Null Handling

Let's explore the strategies that can help Java Engineers write null-safe code, ensuring applications are more reliable and maintainable.

1. Explicit Null Checks: The First Line of Defense

When to use: Use explicit null checks when dealing with code that interacts with external libraries or systems where you have little

control over the input.

The simplest way to guard against null is an explicit check:

```
if (user != null) {
   user.updateProfile();
}
```

2. Embracing Optional: A Modern Approach

When to use: Optional is best used as a return type where there might not be a value to return, and when you want to avoid null checks.

Java 8 introduced the Optional class, which can help you express a variable that might be null more explicitly:

```
Optional<OptionalUser> optionalUser = userRepository.find\
ById(userId);
optionalUser.ifPresent(User::updateProfile);
```

3. Assertions: Catching Bugs Early

When to use: Assertions are ideal in the development and testing phases but should not be relied upon for public API argument checking.

Assertions are a way to document assumptions and catch bugs during development:

```
assert user != null;
```

4. Annotations: Self-Documenting Code

When to use: Annotations are a great way to communicate nullability contracts and help static analysis tools identify potential null-related errors.

Annotations like `@NonNull` and `@Nullable` can be used to indicate when a method parameter, return type, or field can or cannot be null:

```
public void updateUserProfile(@NonNull User user) {}
```

5. Null Object Pattern: Avoiding Null Checks

When to use: This pattern is useful when you want to avoid multiple null checks and when it makes sense to have default behavior.

The Null Object Pattern involves creating an object with default behavior or no behavior:

```
public class NullUser extends User {
  @Override
  public void updateProfile(ProfileInfo info) {}
}

User user = userRepository.findById(userId).orElse(new Nu\
llUser());
user.updateProfile(profileInfo);
```

6. Libraries: Third-Party Helpers

When to use: Turn to libraries when you want robust, tried-and-tested methods to handle null values and are open to adding third-party dependencies.

Libraries like Apache Commons Lang provide utility methods for handling null. For example:

```
String value = StringUtils.defaultIfEmpty(getStringMayBeN\
ull(), "defaultString");
```

7. Design by Contract: Enforcing Usage

When to use: Design by contract is essential for public APIs, as it enforces the proper use of methods and helps prevent errors caused by improper arguments.

Define clear contracts for your methods:

```
public void updateUserProfile(User user) {
  Objects.requireNonNull(user, "User cannot be null");
}
```

8. Safe Navigation Operator: A Future Prospect

While Java does not currently have the Elvis operator (?.), it's worth looking out for in future versions. It's popular in languages like Groovy for handling nulls gracefully.

9. Java 8+ Stream API: Functional Null Handling

When to use: The Stream API is best when working with collections or streams of data where you want to filter out null values in a clean, functional style.

Java's Stream API provides a functional approach to handle possible null values in collections:

```
public void updateMultipleUsers(List<String> userIds) {
 userIds.stream()
        .filter(Objects::nonNull)
        .map(userRepository::findById)
        .filter(Optional::isPresent)
        .map(Optional::get)
        .forEach(this::updateUserProfile);
}
```

10. Failing Fast: Early Detection

When to use: Failing fast is a general best practice that applies across many programming scenarios, not just null handling.

Throw exceptions as soon as a null value is detected:

```
public void updateUserProfile(User user) {
 if (user == null) {
    throw new IllegalArgumentException("User cannot be nu\
ll");
 }
}
```

Effective Logging with SLF4J and Logback

Effective Logging is an essential aspect of any Java application, providing insights into its operational state. It is especially crucial in production environments, where it aids in debugging, monitoring, and incident response. In this comprehensive guide, we will explore the effective practices for using SLF4J with Logback, ensuring a reliable and maintainable logging strategy.

By following these best practices, developers and operations teams can leverage SLF4J and Logback to turn logs into strategic resources for application management and incident resolution. Embracing these guidelines will lead to improved observability, quicker troubleshooting, and a deeper understanding of system behavior, establishing a solid foundation for application reliability and performance.

Key Benefits of Effective Logging

- Improved observability: Logs provide a detailed record of application behavior, making it easier to understand how the system is operating and identify potential issues.
- Faster troubleshooting: Well-structured and informative logs enable developers to quickly pinpoint the root cause of problems and resolve them efficiently.
- Enhanced incident response: Logs are invaluable during incident response, providing a chronological account of events leading up to and during an issue.
- Compliance and security: Logs can serve as evidence of compliance with regulations and help identify security breaches

or suspicious activities.

Choosing SLF4J and Logback

SLF4J (Simple Logging Facade for Java) is a popular logging facade that provides a consistent API for logging across different logging frameworks. Logback is a widely used logging framework that offers a rich set of features and customization options. By combining SLF4J with Logback, you can benefit from the flexibility and power of both tools.

In this guide, we will cover 14 essential best practices for using SLF4J and Logback effectively in your Java applications. These practices will help you achieve reliable, maintainable, and informative logging that supports your application's operational needs.

1. Use SLF4J as the Logging Facade

Good Practice:

Choose SLF4J as your application's logging facade to decouple your logging architecture from the underlying logging library implementation. This abstraction allows you to switch between different logging frameworks without major code changes.

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;

public class MyClass {
  private static final Logger logger = LoggerFactory.getL\
ogger(MyClass.class);
  // ...
}
```

Avoid Practice:

Hardcoding a specific logging framework implementation in your application code can lead to difficulties when needing to switch libraries.

```
import org.apache.log4j.Logger;

public class MyClass {
  private static final Logger logger = Logger.getLogger(M\
yClass.class);
  // ...
}
```

2. Configure Logback for Efficient Logging

Good Practice:

Externalize your Logback configuration and use ‘PatternLayout‘ for improved performance and flexibility. Define different configurations for development, staging, and production environments to better manage the verbosity and detail of logs.

```
<configuration>

  <appender name="STDOUT" class="ch.qos.logback.core.Cons\
oleAppender">
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logge\
r{36} - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="debug">
    <appender-ref ref="STDOUT" />
  </root>

</configuration>
```

Avoid Practice:

Using an outdated or non-performant layout class and hardcoding configuration settings in the code can make it difficult to adapt to different environments.

```
<configuration>

  <appender name="STDOUT" class="ch.qos.logback.core.Cons\
oleAppender">
    <layout class="ch.qos.logback.classic.PatternLayout">
      <!-- Non-recommended layout configuration -->
    </layout>
  </appender>

  <!-- ... -->

</configuration>
```

3. Use Appropriate Log Levels

Good Practice:

Log at the correct level to convey the importance and intention of the message. Use ‘INFO‘ for general events, ‘DEBUG‘ for detailed information during development, and ‘ERROR‘ for serious issues that need attention.

```
logger.info("Application has started.");
logger.debug("The value of X is {}", x);
logger.error("Unable to process the request.", e);
```

Avoid Practice:

Logging everything at the same level, can overwhelm the log files with noise and make it difficult to spot critical issues.

```
logger.error("Application has started."); // Incorrect us\
e of log level
logger.error("The value of X is " + x); // Inefficient st\
ring concatenation
// ...
```

4. Log Meaningful Messages

Good Practice:

Include relevant information such as transaction or correlation IDs in your log messages to provide context. This is especially helpful in distributed systems for tracing requests across services.

```
logger.info("Order {} has been processed successfully.", \
orderId);
```

Avoid Practice:

Vague or generic log messages that do not provide sufficient context to understand the event or issue.

```
logger.info("Processed successfully."); // No context pro\
vided
```

5. Use Placeholders for Dynamic Content

Good Practice:

Utilize placeholders to avoid unnecessary string concatenation when the log level is disabled, saving memory and CPU cycles.

```
logger.debug("User {} logged in at {}", username, LocalDa\
teTime.now());
```

Avoid Practice:

Concatenating strings within log statements is less efficient.

```
logger.debug("User " + username + " logged in at " + Loca\
lDateTime.now());
```

6. Log Exceptions with Stack Traces

Good Practice:

Always log the full exception, including the stack trace, to provide maximum context for diagnosing issues.

```
try {
  // some code that throws an exception
} catch (Exception e) {
  logger.error("An unexpected error occurred", e);
}
```

Avoid Practice:

Logging only the exception message without the stack trace can omit critical diagnostic information.

```
try {
  // some code that throws an exception
} catch (Exception e) {
  logger.error("An unexpected error occurred: " + e.getMe\
ssage());
}
```

7. Use Asynchronous Logging for Performance

Good Practice:

Implement asynchronous logging to improve application performance by offloading logging activities to a separate thread.

```
<configuration>

  <appender name="ASYNC" class="ch.qos.logback.classic.As\
yncAppender">
    <appender-ref ref="FILE" />
  </appender>

  <appender name="FILE" class="ch.qos.logback.core.FileAp\
pender">
    <file>application.log</file>
    <encoder>
      <pattern>%d{HH:mm:ss.SSS} [%thread] %-5level %logge\
r{36} - %msg%n</pattern>
    </encoder>
  </appender>

  <root level="INFO">
    <appender-ref ref="ASYNC" />
  </root>

</configuration>
```

Avoid Practice:

Synchronous logging in performance-critical paths without considering the potential for log-related latency.

```
logger.info("A time-sensitive operation has completed.");
```

8. Log at the Appropriate Granularity

Good Practice:

You should balance between logging too much and too little. Log at the appropriate granularity based on the specific requirements of your application. Avoid excessive logging that clutters the logs and makes it difficult to identify important information.

```
public void processOrder(Order order) {

  logger.info("Processing order: {}", order.getId());

  // Logging at a finer granularity for debugging purposes
  logger.debug("Order details: {}", order);

  // Process the order
  orderService.save(order);

  logger.info("Order processed successfully");
}
```

Avoid Practice:

Excessive logging at a high granularity in production, can lead to performance issues and log flooding.

```
public void processOrder(Order order) {

  logger.trace("Entering processOrder method");
  logger.debug("Received order: {}", order);
  logger.info("Processing order: {}", order.getId());

  // Logging every step of order processing
  logger.debug("Step 1: Validating order");
  // ...
  logger.debug("Step 2: Calculating total amount");
  // ...
  logger.debug("Step 3: Updating inventory");
  // ...
```

```
14
15    logger.info("Order processed successfully");
16    logger.trace("Exiting processOrder method");
17  }
```

9. Monitor and Rotate Log Files

Good Practice:

Configure log file rotation based on size or time to prevent logs from consuming excessive disk space. Set up monitoring for log files to trigger alerts when nearing capacity.

Avoid Practice:

Letting log files grow indefinitely, can lead to disk space exhaustion and potential system failure.

10. Secure Sensitive Information

Good Practice:

Implement filters or custom converters in your logging framework to redact or hash sensitive data before it's written to the logs.

```
1  log.info("Processing payment with card: {}", maskCreditCa\
2  rd(creditCardNumber));
3
4  public String maskCreditCard(String creditCardNumber) {
5    int length = creditCardNumber.length();
6    if (length < 4) return "Invalid number";
7    return "****-****-****-" + creditCardNumber.substring(l\
8  ength - 4);
9  }
```

Avoid Practice:

Logging sensitive information such as passwords, API keys, Credit Cards, or personally identifiable information (PII).

```
log.info("Processing payment with card: {}", creditCardNu\
mber);
```

11. Structured Logging

Good Practice:

Adopt structured logging to output logs in a machine-readable format like JSON, facilitating better searching and indexing in log management systems.

```
<configuration>

  <appender name="JSON_CONSOLE" class="ch.qos.logback.cor\
e.ConsoleAppender">
    <encoder class="net.logstash.logback.encoder.LoggingE\
ventCompositeJsonEncoder">
      <providers>
        <timestamp>
          <timeZone>UTC</timeZone>
        </timestamp>
        <version />
        <logLevel />
        <threadName />
        <loggerName />
        <message />
        <context />
        <stackTrace />
      </providers>
```

```
    </encoder>
  </appender>

  <root level="info">
    <appender-ref ref="JSON_CONSOLE" />
  </root>

</configuration>
```

Let's take a look at an example log message that is printed in JSON format:

```
logger.info("Order has been processed");
```

The output of the above log message will be printed as below:

```
{"@timestamp":"2024-03-26T15:52:00.789Z","@version":"1","\
message":"Order has been processed","logger_name":"Applic
ation","thread_name":"main","level":"INFO"}
```

Avoid Practice:

Using unstructured log formats that are difficult to parse and analyze programmatically.

12. Integration with Monitoring Tools

Good Practice:

Link your logging with monitoring and alerting tools to automatically detect anomalies and notify the concerned teams.

Avoid Practice:

Ignoring the integration of logs with monitoring systems can delay the detection of issues.

13. Log Aggregation

Good Practice:

In distributed environments, use centralized log aggregation to collect logs from multiple services, simplifying analysis and correlation of events.

Avoid Practice:

Allowing logs to remain scattered across various systems, complicates the troubleshooting process.

14. Smart Logging

We have great content here for implementing Smart Logging using AOP.

Best Practices for Handling Exceptions

Java exception handling is an absolute must if you want to develop reliable applications. Neglecting to handle exceptions properly can cause instability and ultimately ruin the user's experience especially when dealing with third-party services and Microservices. But don't worry, we've got you covered! This article will show you how to handle exceptions like a Pro.

Understanding the Exception Hierarchy

Java's exception hierarchy is designed to categorize and handle different types of exceptions effectively. It is essential to understand this hierarchy when designing exception-handling mechanisms for a health management system. The two main categories of exceptions are:

1. Checked Exceptions: These exceptions represent anticipated error conditions that can occur during the normal execution of the program. They are typically recoverable and require explicit handling or declaration in the method signature. Examples include `IOException`, `SQLException`, and custom exceptions specific to the health management domain, such as `PatientNotFoundException` or `MedicalRecordAccessException`. Checked exceptions ensure that the developer is aware of potential issues and

can implement appropriate error handling and recovery mechanisms.

2. Unchecked Exceptions: These exceptions indicate programming errors or unexpected runtime conditions. They are subclasses of `RuntimeException` and do not require explicit handling or declaration. Examples include `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `IllegalArgumentException`. Unchecked exceptions often signify bugs or improper usage of APIs and should be prevented through careful programming practices and input validation.

Understanding the distinction between checked and unchecked exceptions allows developers to design exception-handling strategies that align with the nature of the exceptions and the specific requirements of the health management system.

Java Exception Practices:

- Catch Specific Exceptions
- Avoid Swallowing Exceptions
- Utilize Finally Blocks or Try-With-Resources
- Document Exceptions
- Avoid Using Exceptions for Flow Control
- Throw Specific and Meaningful Exceptions
- Prefer Checked Exceptions for Recoverable Conditions
- Wrap Exceptions When Appropriate
- Log Exceptions with Relevant Details
- Handle Exceptions at the Appropriate Layer

Now let's dive into all of them:

1. Catch Specific Exceptions

When catching exceptions, it is recommended to catch the most specific exception type possible. This practice enables precise error handling and avoids masking potential bugs or unintended behavior.

Bad Practice:

```
try {
  patientService.updatePatientRecord(patientRecord);
} catch (Exception e) {
  // Generic exception handling
  // Masks specific errors and hinders effective debuggin\
g
}
```

Good Practice:

```
try {
  patientService.updatePatientRecord(patientRecord);
} catch (PatientNotFoundException e) {
  // Handle specific exception when patient record is not\
 found
  // Log and propagate the exception or perform necessary\
 recovery steps
} catch (DatabaseAccessException e) {
  // Handle specific exception related to database access\
 issues
  // Implement appropriate error handling and recovery me\
chanisms
}
```

By catching specific exceptions, developers can provide targeted error handling, improve code clarity, and facilitate effective debugging and maintenance.

2. Avoid Swallowing Exceptions

Swallowing exceptions, i.e., catching an exception without properly handling or logging it, is a dangerous practice that can lead to silent failures and difficult-to-diagnose issues. In a health management system, ignoring exceptions can have severe consequences, such as data inconsistencies or improper patient care.

Bad Practice:

```
try {
  prescriptionService.fillPrescription(prescription);
} catch (DrugNotFoundException e) {
  // Swallowing the exception without proper handling or \
logging
  // Potential issues remain unnoticed and unresolved
}
```

Good Practice:

```
try {
  prescriptionService.fillPrescription(prescription);
} catch (DrugNotFoundException e) {
  // Log the exception with relevant details
  logger.error("Failed to fill prescription. Drug not fou\
nd: {}", prescription.getDrugId(), e);

  // Propagate the exception or perform necessary error h\
andling
  throw new PrescriptionFillException("Failed to fill pre\
scription", e);
}
```

By logging exceptions and properly handling or propagating them, developers can ensure that errors are visible, traceable, and addressed promptly.

3. Utilize Finally Blocks or Try-With-Resources

Resource management is critical, especially when dealing with external resources such as file handles, database connections, or network sockets. Proper resource cleanup ensures that resources are released promptly, preventing resource leaks and maintaining system stability.

Bad Practice:

```
BufferedReader reader = null;
try {
  reader = new BufferedReader(new FileReader("patient_dat\
a.txt"));
  // Process patient data

} catch (IOException e) {
  // Exception handling

} finally {
  if (reader != null) {
    try {
      reader.close();
    } catch (IOException e) {
      // Exception handling during resource cleanup
    }
  }
}
```

Good Practice:

```
try (BufferedReader reader = new BufferedReader(new FileR\
eader("patient_data.txt"))) {
  // Process patient data

} catch (IOException e) {
  // Exception handling
}
```

By using try-with-resources (available since Java 7), the resource is automatically closed when the try block exits, eliminating the need for explicit cleanup code in the finally block. This approach ensures proper resource handling and reduces the likelihood of resource leaks.

4. Document Exceptions

Documenting exceptions is crucial for maintaining a clear and maintainable codebase. By providing meaningful and accurate documentation, developers can communicate the expected behavior of methods, the exceptions they may throw, and any preconditions or postconditions.

Good Practice:

```
/**
 * Updates the patient's health record with the provided \
information.
 *
 * @param record The health record to update.
 * @throws PatientNotFoundException if the patient record\
 does not exist.
 * @throws DatabaseAccessException if there is an error a\
ccessing the database.
 */
```

```
public void updateHealthRecord(HealthRecord record) throw\
s PatientNotFoundException, DatabaseAccessException {
  // Implementation
}
```

Documenting exceptions using Javadoc or inline comments helps other developers understand the possible exceptional conditions and how to handle them appropriately. It promotes code clarity, maintainability, and collaboration among team members.

5. Avoid Using Exceptions for Flow Control

Exceptions should not be used as a means of normal flow control in a program. Overusing exceptions for non-exceptional scenarios can lead to complex and hard-to-understand code, reduced performance, and decreased maintainability.

Bad Practice:

```
try {
  return patientList.get(patientId);
} catch (IndexOutOfBoundsException e) {
  return null;
}
```

Good Practice:

```
if (patientId >= 0 && patientId < patientList.size()) {
  return patientList.get(patientId);
}
return null; // or throw a specific exception if necessary
```

By using conditional statements and proper input validation, developers can handle expected situations without relying on exceptions. Exceptions should be reserved for truly exceptional scenarios that disrupt the normal flow of the program.

6. Throw Specific and Meaningful Exceptions

When throwing exceptions, it is important to use specific and meaningful exception types that accurately represent the nature of the error or exceptional condition. Generic exceptions like 'Exception' or 'RuntimeException' provide little information about the cause of the problem and make it harder to handle exceptions appropriately.

Bad Practice:

```
public void savePatientData(PatientData data) throws Exce\
ption {
  // Throwing a generic exception
  // Lacks specificity and hinders effective exception ha\
ndling
}
```

Good Practice:

```
public void savePatientData(PatientData data) throws Data\
PersistenceException {
  // Throwing a specific exception related to data persis\
tence
  // Provides clear indication of the nature of the probl\
em
}
```

By throwing specific exceptions, developers can convey the precise error condition, making it easier to handle and diagnose issues. Custom exception classes can be created to represent domain-specific exceptional scenarios, providing additional context and facilitating targeted exception handling.

7. Prefer Checked Exceptions for Recoverable Conditions

Checked exceptions are suited for representing recoverable error conditions that the caller is expected to handle. They enforce explicit exception handling and make the exceptional conditions visible in the method signature.

Bad Practice:

```
public void calculateDosage(Patient patient) {
  if (patient.getWeight() <= 0) {
    throw new IllegalArgumentException("Patient weight mu\
st be positive");
  }
  // Dosage calculation logic
}
```

Good Practice:

```
public void calculateDosage(Patient patient) throws Inval\
idPatientWeightException {
  if (patient.getWeight() == 0) {
    throw new InvalidPatientWeightException("Patient weig\
ht cannot be zero.");
  }
}
```

8. Wrap Exceptions When Appropriate

When propagating exceptions across different layers or modules of the application, it may be necessary to wrap the original exception inside a more appropriate exception type. Wrapping exceptions allows for providing additional context, hiding implementation details, and presenting a consistent exception interface to the caller.

Good Practice:

```
public void retrievePatientHistory() throws PatientDataAc\
cessException {
  try {
    // Retrieve patient history from the database
    // ...

  } catch (SQLException e) {
    // Wrap the SQLException in a more appropriate except\
ion type
    throw new PatientDataAccessException("Failed to retri\
eve patient history", e);
  }
}
```

Wrapping exceptions helps maintain a clean and abstracted exception hierarchy, encapsulating the underlying implementation exceptions and providing more meaningful exceptions to the higher layers of the application.

9. Log Exceptions with Relevant Details

Logging exceptions is crucial for monitoring, debugging, and troubleshooting issues in a health management system. When logging exceptions, it is important to include relevant details such as the exception message, stack trace, and any additional contextual information that can aid in problem resolution.

Good Practice:

```
try {
  // Perform database operation
  // ...

} catch (SQLException e) {
  // Log the exception with relevant details
  logger.error("Database operation failed. Patient ID: {}\
, Operation: {}", patientId, operation, e);

  // Rethrow the exception or handle it appropriately
  throw new DatabaseAccessException("Failed to perform da\
tabase operation", e);
}
```

By logging exceptions with meaningful details, developers and support teams can quickly identify and diagnose issues, reducing the time required for problem resolution and improving the overall maintainability of the system.

10. Handle Exceptions at the Appropriate Layer

Exception handling should be performed at the appropriate layer of the application, based on the responsibility and scope of each layer. The goal is to handle exceptions at a level where they can be effectively managed and where appropriate actions can be taken.

Good Practice:

```
// Data Access Layer
public Patient getPatientById(int patientId) throws Patie\
ntNotFoundException {
  try {
    // Retrieve patient from the database
    // ...

  } catch (SQLException e) {
    // Log the exception and throw a specific exception
    logger.error("Failed to retrieve patient from the dat\
abase. Patient ID: {}", patientId, e);
    throw new PatientNotFoundException("Patient not found\
 with ID: " + patientId);
  }
}

// Service Layer
public void updatePatientProfile(Patient patient) throws \
ProfileUpdateException {
  try {
    // Perform business logic and update patient profile
    // ...
    patientRepository.updatePatient(patient);

  } catch (PatientNotFoundException e) {
```

```
26      // Handle the exception thrown by the data access lay\
27  er
28      logger.warn("Patient not found while updating profile\
29  . Patient ID: {}", patient.getId(), e);
30      throw new ProfileUpdateException("Failed to update pa\
31  tient profile", e);
32    }
33  }
```

In this example, the data access layer handles the low-level SQLException and throws a more specific PatientNotFoundException. The service layer catches the PatientNotFoundException and takes appropriate action, such as logging a warning and throwing a higher-level ProfileUpdateException.

By handling exceptions at the appropriate layer, the system can provide more meaningful error messages, maintain a clear separation of concerns, and ensure that exceptions are handled at a level that can be effectively managed.

Chapter 6 - Best Practices for Handling Database

Hello Java developers, accessing the Database is crucial but what are the best practices? In this article, we will dive into the Java database best practices that are essential for every developer aiming to master database interactions and ORM (Object-Relational Mapping) in Java. By adopting these best practices, you'll enhance not only the performance of your applications but also their maintainability. We'll cover the key techniques and practices such as abstraction, transaction management, lazy loading, and null handling, which will help you effectively manage common challenges in database access and ORM (Object-Relational Mapping).

1. Using Repository Abstraction

When using frameworks such as Spring, Micronaut, or Quarkus it is recommended to use the Data repository interfaces, such as JpaRepository, which allows for a more abstract approach to database access, leading to cleaner and more maintainable application code.

Good: Interface-based repositories enable a cleaner separation of concerns

```
1  public interface UserRepository extends JpaRepository<Use\
2  r, Long> {
3    Optional <User> findByEmail(String email);
4  }
```

```
@Service
public class UserService {
    private final UserRepository userRepository;

    @Autowired
    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserByEmail(String email) {
        return userRepository.findByEmail(email)
                .orElseThrow(() -> new UserNotFoundExcept\
ion("User not found"));
    }
}
```

Explanation: This is good because it uses the framework Data's repository abstraction to create a query method that is clear and concise. The findByEmail method follows the framework Data JPA conventions and automatically provides the implementation. The use of Optional is a good practice to avoid null checks and to handle the absence of a result in a clean, expressive way.

Avoid: Direct use of EntityManager results in error-prone and less manageable code

```
@Service
public class UserService {

    @Autowired
    private EntityManager entityManager;

    public User getUserByEmail(String email) {
        // Bad: Using EntityManager directly for somethin\
g repositories could do
        Query query = entityManager.createQuery("SELECT u\
```

```
11   FROM User u WHERE u.email = :email");
12
13           query.setParameter("email", email);
14           return (User) query.getSingleResult();
15       }
16
17   }
```

Explanation: This is bad because it bypasses the simplicity and safety provided by repository abstractions. Direct use of EntityManager for this type of query is unnecessary and verbose. It also opens up the potential for JPQL injection if not handled properly and the method does not handle the case when the result is not found, potentially throwing a NoResultException.

##2. Transaction Management

Good: Proper use of @Transactional at the service layer

```
1   @Service
2   @Transactional
3   public class UserService {
4       private final UserRepository userRepository;
5
6       public UserService(UserRepository userRepository) {
7           this.userRepository = userRepository;
8       }
9
10      public User createUser(User user) {
11          // The transaction is automatically managed by Sp\
12  ring
13          return userRepository.save(user);
14      }
15  }
```

Explanation: This is good because it uses the @Transactional annotation at the service level, which is the recommended way to handle

transactions in a Spring application. This ensures that the entire method execution is wrapped in a transactional context, which provides consistency and integrity in the database operations.

Avoid: Annotating individual repository methods with @Transactional is unnecessary

```
public interface UserRepository extends JpaRepository<Use\
r, Long> {
  @Transactional
  <extends User S> save(S entity);
}
```

Explanation: This is bad because adding @Transactional to a method in a repository interface is not necessary since Spring Data repositories are already transactional by nature. Moreover, this can lead to confusion, as transaction management should typically be handled at the service layer, not the repository layer. The service layer often represents business transactions that can span multiple repository calls, requiring a broader transactional context.

3. Handling Lazy Initialization

Good: Load lazy associations within a transaction using Hibernate.initialize()

```
@Service
public class UserService {
    private final UserRepository userRepository;

    @Transactional(readOnly = true)
    public User getUserWithOrders(Long userId) {
        User user = userRepository.findById(userId)
                .orElseThrow(() - new UserNotFoundExcepti\
on("User not found"));
        // Initialize lazy collection
        Hibernate.initialize(user.getOrders());
        return user;
    }
}
```

Explanation: This is good because it handles the lazy loading within a transactional context, ensuring that the Hibernate.initialize() method can load the lazy collection before the session is closed. The use of readOnly = true is also good practice for read operations as it can optimize performance.

Avoid: Accessing a lazily loaded collection outside a transaction can lead to exceptions

```
@Service
public class UserService {
    @Autowired
    private UserRepository userRepository;

    public User getUserWithOrders(Long userId) {
        User user = userRepository.findById(userId).get();

        // May throw LazyInitializationException
        int orderCount = user.getOrders().size();

        return user;
```

```
    }
}
```

Explanation: This is bad because it attempts to access a lazily loaded collection outside of an open session, which can result in a LazyInitializationException. There's no @Transactional annotation, meaning that the session may be closed before the lazy collection is accessed.

4. Using Pagination

Good: Implementing pagination with Spring Data's Pageable Pagination helps in fetching data in manageable chunks, thus saving resources.

```
public interface UserRepository extends JpaRepository<Use\
r, Long> {
    PageUser findAll(Pageable pageable);
}

@Service
public class UserService {
    private final UserRepository userRepository;

    public Page<User> getUsersWithPagination(int page, in\
t size) {
        Pageable pageable = PageRequest.of(page, size, So\
rt.by("lastName").ascending());
        return userRepository.findAll(pageable);
    }
}
```

Explanation: This is good practice because it takes advantage of Spring Data's built-in support for pagination, which is important

for performance and usability when dealing with large datasets. The Pageable parameter encapsulates pagination information and sorting criteria, which the repository infrastructure uses to generate the correct query.

Avoid: Retrieving all entries can lead to memory and performance issues

```
@RestController
public class UserController {
    private final UserRepository userRepository;

    @GetMapping("/users")
    public <List> User getAllUsers() {
    }
}
```

5. Handling Null Values with Optional

Check out here for more details dealing with null values.

Good: Using orElseThrow() to handle absent values elegantly

```
@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public User getUserById(Long id) {
        return userRepository.findById(id)
```

```
                .orElseThrow(() -> new UserNotFoundExcept\
ion("User with id " + id + " not found"));
    }
}
```

Explanation: This is good because it makes use of Optional, a feature introduced in Java 8, which is designed to provide a better alternative to null. It forces the developer to think about the case when the User might not be found and handle it accordingly, possibly throwing a custom exception. Avoid: Using get() without checking if the value is present can cause exceptions

```
@Service
public class UserService {
    private final UserRepository userRepository;

    public User getUserById(Long id) {
        return userRepository.findById(id).get();
    }
}
```

Explanation: This is bad because it assumes that the findById method will always return a non-null value, which is not guaranteed. Using get() directly on the Optional returned by findById may throw a NoSuchElementException if the Optional is empty (i.e. if the user is not found). This approach fails to handle the potential absence of a User with the given ID in a clean, safe manner.

Summary

The “Good” examples follow the best practices of Java framework Data JPA and Java, promoting code readability, maintainability, and proper error handling. In contrast, the “Avoid” examples

show common pitfalls that can lead to bugs, inefficient database operations, and code that is harder to maintain and understand.

Appendix

A. Additional Resources

Books

- **"Effective Java" by Joshua Bloch**: A comprehensive guide to writing robust, maintainable, and efficient Java code.

B. Tools and Libraries

Logging

- **Logback**[1]: A powerful and flexible logging framework for Java applications.
- **SLF4J**[2]: A simple facade for logging frameworks, allowing the end user to plug in the desired logging framework at deployment time.

Exception Handling

- **Vavr**[3]: Functional programming library for Java that provides immutable collections and functional control structures.
- **Exception handling like a Pro**[4]: Built-in support for handling exceptions in Spring Boot applications.

[1] https://logback.qos.ch/
[2] http://www.slf4j.org/
[3] https://github.com/vavr-io/vavr/
[4] https://mezocode.com/exception-handling-in-java-like-a-pro/

API Design

- **Swagger/OpenAPI**[5]: A powerful toolkit for API design and documentation.
- **Postman**[6]: An API platform for building and using APIs, enabling easier testing and collaboration.
- **RESTful web API design**[7]: An API platform for building and using APIs, enabling easier testing and collaboration.

C. Code Samples and Templates

For more coding examples refer to my GitHub repo[8]

[5] https://swagger.io/
[6] https://www.postman.com/
[7] https://learn.microsoft.com/en-us/azure/architecture/best-practices/api-design
[8] https://github.com/mezocode