Ravi Yasas Follow

Sep 21, 2022 · 8 min read · Listen

Save

Spring Boot Best Practices for Developers

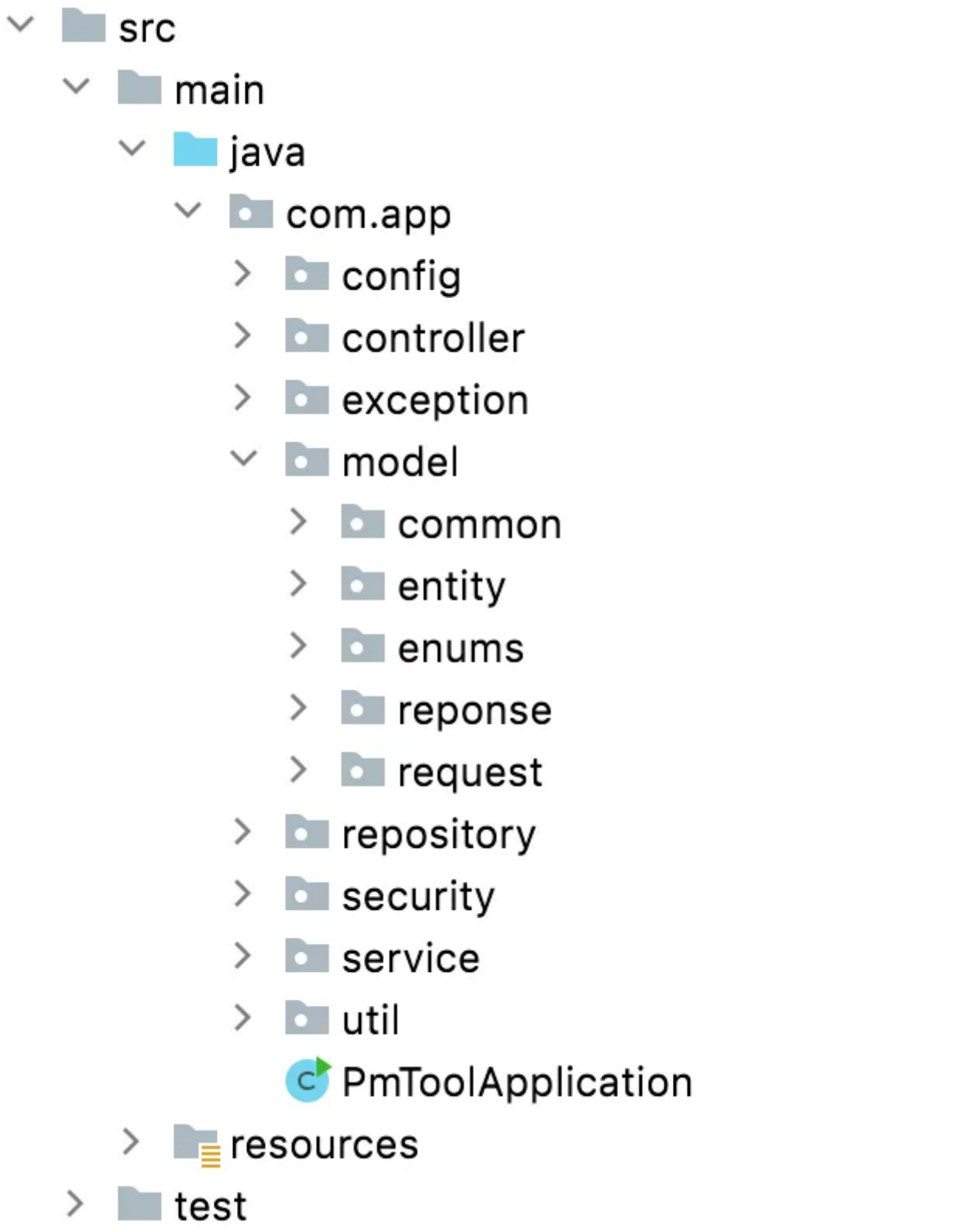
Spring Boot is a widely used and very popular enterprise-level high-performance framework. Here are some best practices and a few tips you can use to improve your Spring Boot application and make it more efficient. This article will be a little longer, and it will take some time to completely read the article.

Proper packaging style

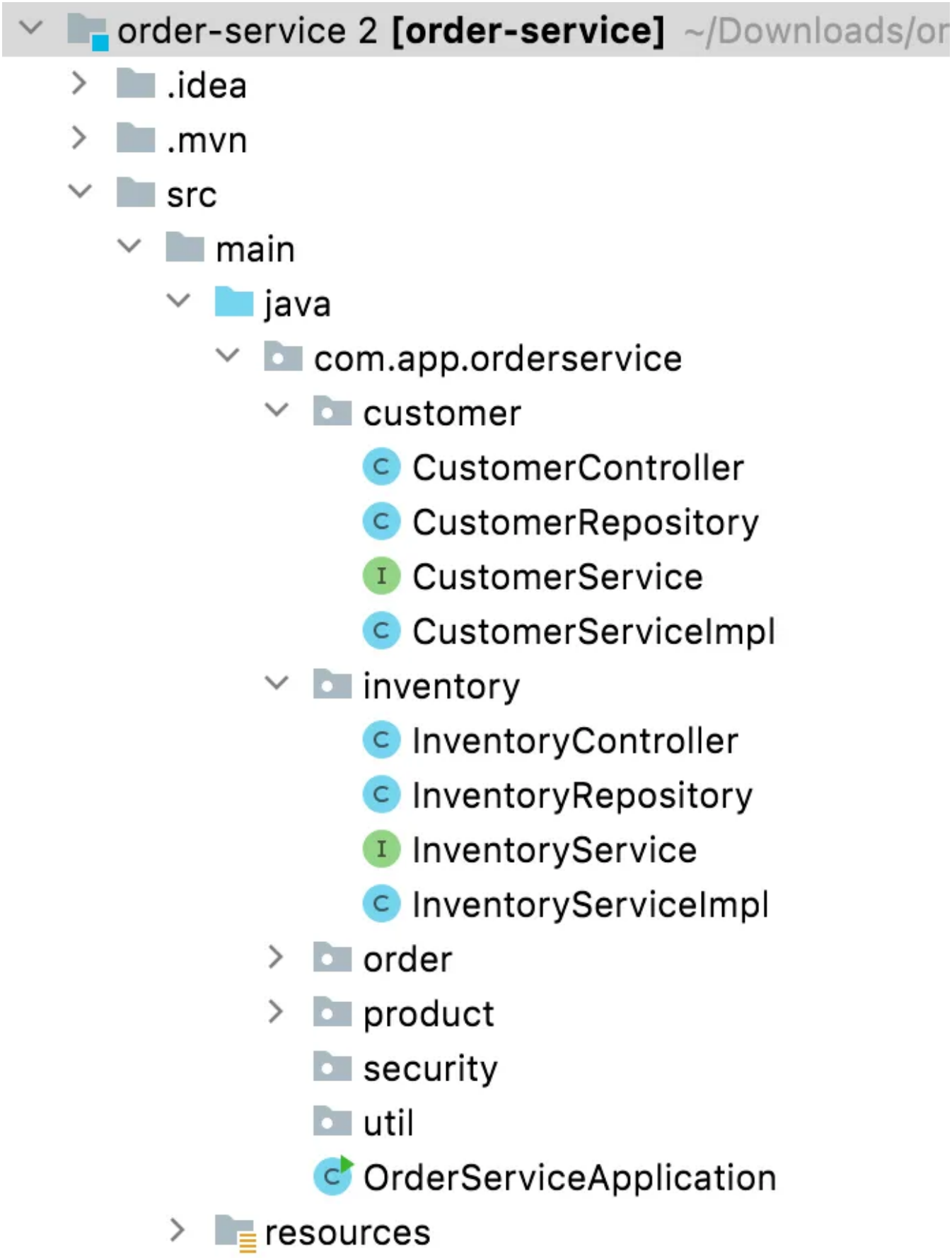
- Proper packaging will help to understand the code and the flow of the application easily.
- You can structure your application with meaningful packaging.

- You can include all your controllers into a separate package, services in a separate package, util classes into a separate package…etc. This style is very convenient in small-size microservices.
- If you are working on a huge code base, a feature-based approach can be used. You can decide on your requirement.

Based on type



Based on feature



Use design patterns

- No complaints. Design patterns are already best practices.
- But you must identify the place where you can use them.

- Please check this article to understand "*how to use the Builder design pattern*" in our Spring Boot applications.

Use Spring Boot starters

Image: https://www.merriam-webster.com/words-at-play/names-of-appetizers

- This is a cool feature of Spring Boot.
- We can very easily use starter dependencies without adding single dependencies one by one. These starter dependencies are already bundled with the required dependencies.
- For example, if we add **spring-boot-starter-web** dependency, by default it is bundled with **jackson, spring-core, spring-mvc, and spring-boot-starter-tomcat dependencies**.
- So we don't need to care about adding dependencies separately.
- And also it helps us to avoid version mismatches.

Use proper versions of the dependencies

- It is always recommended to use the latest stable **GA** versions.

- Sometimes it may vary with the Java version, server versions, the type of the application…etc.
- Do not use different versions of the same package and always use <properties> to specify the version if there are multiple dependencies.

```
<properties>
    <java.version>11</java.version>
    <jjwt.version>0.11.2</jjwt.version>
</properties>

<dependencies>

    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-api</artifactId>
        <version>${jjwt.version}</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-impl</artifactId>
        <version>${jjwt.version}</version>
    </dependency>
    <dependency>
        <groupId>io.jsonwebtoken</groupId>
        <artifactId>jjwt-jackson</artifactId>
        <version>${jjwt.version}</version>
    </dependency>
```

Use Lombok

- As a Java developer, you have probably heard of the **Lombok project.**
- Lombok is a Java library that can be used to reduce your codes and allow you to write clean code using its annotations.
- For example, you may use plenty of lines for getters and setters in some classes like entities, request/response objects, dtos…etc.
- But if you use Lombok, it is just one line, you can use **@Data, @Getter** or **@Setter** as per your requirement.
- You can use ***Lombok*** logger annotations as well. ***@Slf4j*** is recommended.
- Check this ***file*** for your reference.

Use constructor injection with Lombok

```
@RestController
@RequestMapping("api/v1")
@CrossOrigin(origins = "http://localhost:4200")
@RequiredArgsConstructor
public class IssueController {

    private static final Logger logger = LoggerFactory.getLogger(IssueController.class);

    private final IssueService issueService;
```

- When we talk about dependency injection, there are two types.
- One is "**constructor injection**" and the other is "**setter injection**". Apart from that, you can also use "**field injection**" using the very popular **@Autowired** annotation.
- But we highly recommend using **Constructor injection** over other types. Because it allows the application to initialize all required dependencies at the initialization time.
- This is very useful for unit testing.
- The important thing is, that we can use the **@RequiredArgsConstructor** annotation by Lombok to use constructor injection.

- Check this *sample controller* for your reference.

Use slf4j logging

```
logger.debug("Retrieving project data for projectId: {} - data: {}", id, project);
```

- Logging is very important.
- If a problem occurs while your application is in production, logging is the only way to find out the root cause.
- Therefore, you should think carefully before adding loggers, log message types, logger levels, and logger messages.
- Do not use System.out.print()
- Slf4j is recommended to use along with logback which is the default logging framework in Spring Boot.
- Always use slf4j { } and avoid using String interpolation in logger messages. Because string interpolation consumes more memory.
- Please check *this file* for your reference to get an idea about, implementing a logger.
- You can use Lombok ***@Slf4j*** *annotation to create a logger very easily.*
- If you are in a micro-services environment, you can use the ELK stack.

Use Controllers only for routing

```java
@RestController
@RequestMapping("api/v1")
@CrossOrigin(origins = "http://localhost:4200")
@RequiredArgsConstructor
public class ProjectController {

    private static final Logger logger = LoggerFactory.getLogger(ProjectController.class);

    private final ProjectService projectService;

    @PostMapping("/projects")
    public ResponseEntity<ApiResponse> createProject(@RequestBody ProjectRequest projectRequest) {
        logger.info("Create project API: {}", projectRequest);
        return projectService.saveProject(projectRequest);
    }

    @GetMapping("/projects")
    public ResponseEntity<ApiResponse> getProjects() {
        logger.info("Retrieve all projects API");
        return projectService.getAll();
    }
}
```

- Controllers are dedicated to routing.
- It is **stateless** and **singleton.**
- The DispatcherServlet will check the *@RequestMapping* on *Controllers*
- Controllers are the ultimate target of requests, then requests will be handed over to the service layer and processed by the service layer.
- The business logic **should not be** in the controllers.

Use Services for business logic

- The **complete business logic goes here** with validations, caching…etc.
- Services communicate with the persistence layer and receive the results.
- Services are also singleton.

Bonus article: Manage stress as a Software Engineer

Avoid NullPointerException

- To avoid **NullPointerException** you can use **Optional** from java.util package.
- You can also use null-safe libraries. Ex: **Apache Commons StringUtils**
- Call **equals()** and **equalsIgnoreCase()** methods on known objects.
- Use **valueOf()** over **toString()**
- Use IDE-based **@NotNull** and **@Nullable** annotations.

Use best practices for the Collection framework

- Use appropriate collection for your data set.
- Use **forEach** with Java 8 features and avoid using legacy for loops.
- Use **interface type** instead of the implementation.
- Use **isEmpty()** over **size()** for better readability.
- Do not return null values, you can return an empty collection.
- If you are using objects as data to be stored in a hash-based collection, you should override equals() and hashCode() methods. Please check this article "*How does a HashMap internally work*".

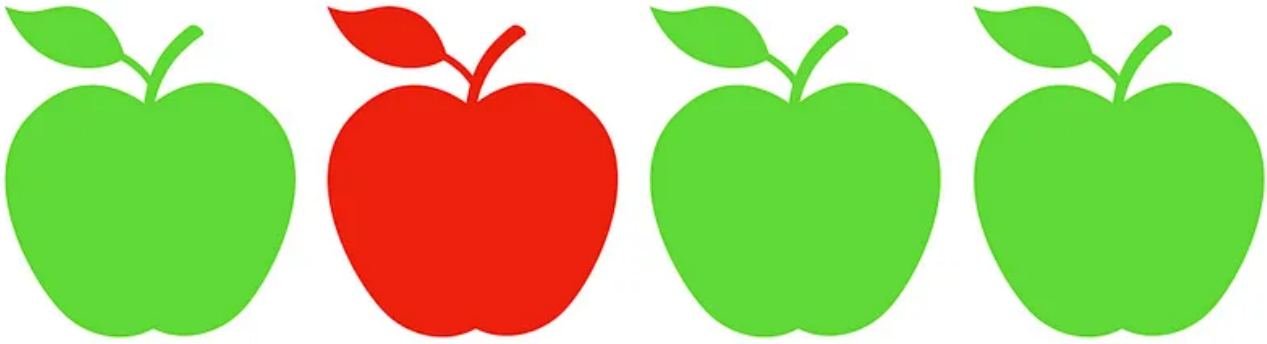
Use pagination

- This will improve the performance of the application.
- If you're using **Spring Data JPA**, the *PagingAndSortingRepository* makes using pagination very easy and with little effort.

Use caching

- Caching is another important factor when talking about application performance.
- By default Spring Boot provides caching with **ConcurrentHashMap** and you can achieve this by **@EnableCaching** annotation.
- If you are not satisfied with default caching, you can use **Redis**, **Hazelcast,** or any other distributed caching implementations.
- Redis and Hazelcast are **in-memory** caching methods. You also can use database cache implementations as well.

Use custom exception handler with global exception handling



- This is very important when working with large enterprise-level applications.
- Apart from the general exceptions, we may have some scenarios to identify some specific error cases.

- Exception adviser can be created with **@ControllerAdvice** and we can create separate exceptions with meaningful details.
- It will make it much easier to identify and debug errors in the future.
- Please check *this* and *this* for your reference.

Bonus article: What is serverless Architecture?

Use custom response object

- A custom response object can be used to return an object with some specific data with the requirements like HTTP status code, API code, message…etc.
- We can use the **builder design pattern** to create a custom response object with custom attributes.
- Please check ***this article*** for your reference.

Remove unnecessary codes, variables, methods, and classes.

```
// check the project is exists or not
/*
Optional<Project> project = projectRepository.findById(issueRequest.getProjectId());
if (project.isEmpty()) {
    throw new DataNotFoundException("Project ID is null or not existing");
}
*/
```

commented code in the production

- **Unused variable** declarations will acquire some memory.
- Remove unused methods, classes…etc because it will impact the performance of the application.

- Try to avoid nested loops. You can use maps instead.

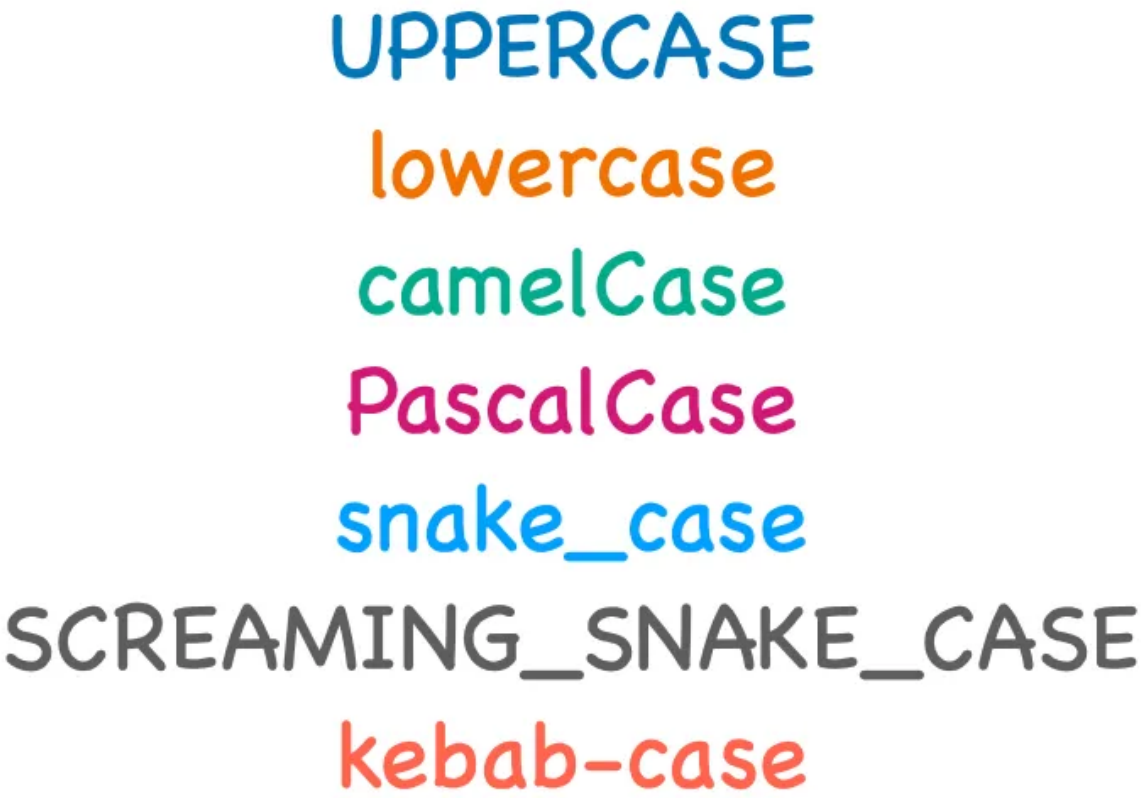
Using comments

- Commenting is a good practice unless you abuse it.
- DO NOT comment on everything. Instead, you can write descriptive code using meaningful words for classes, functions, methods, variables...etc.
- Remove commented codes, misleading comments, and story-type comments.
- You can use comments for warnings and explain something difficult to understand at first sight.

Use meaningful words for classes, methods, functions, variables, and other attributes.

- This looks very simple, but the impact is huge.
- Always use proper **meaningful and searchable** naming conventions with proper case.
- Usually, we use **nouns or short phrases** when declaring **classes**, **variables**, and **constants**. Ex: String firstName, const isValid
- You can use **verbs and short phrases with adjectives** for **functions** and **methods**. Ex: readFile(), sendData()
- Avoid using **abbreviating variable** names and **intention revealing** names. Ex: int i; String getExUsr;
- If you use this meaningfully, declaration comment lines can be reduced. Since it has meaningful names, a fresh developer can easily understand by reading the code.

Use proper case for declarations



- There are many different cases like **UPPERCASE, lowercase, camelCase, PascalCase, snake_case, SCREAMING_SNAKE_CASE, kebab-case**…etc.
- But we need to identify which case is dedicated to which variable.
- Usually, I follow,

classes — **PascalCase**

methods & variables— **camelCase**

constants — **SCREAMING_SNAKE_CASE**

DB-related fields— **snake_case**

- This is just an example. It can be different from the standard you follow in the company.

Be simple

- Always try to write simple, readable codes.
- The same simple logic can be implemented using different ways, but it is difficult to understand if it is not readable or understandable.
- Sometimes complex logic consumes more memory.
- Try to use **KISS**, **DRY**, and **SOLID** principles when writing codes. I will explain this in a future article.

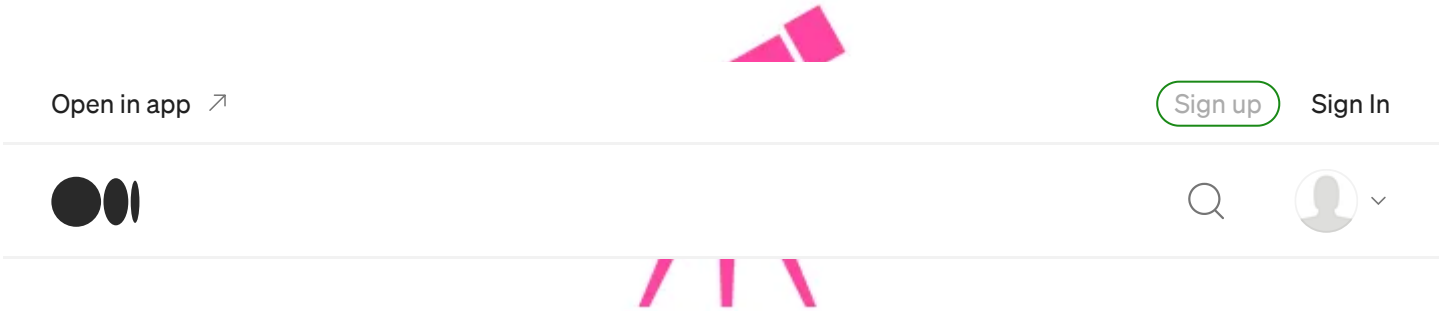
Use a common code formatting style

```
@GetMapping("/issues")
public ResponseEntity<ApiResponse> getIssues() {
    logger.info("Retrieve all issues API");
    return issueService.getAll();
}

@GetMapping("/issues/{id}")
public ResponseEntity<ApiResponse> getIssue(@PathVariable Integer id)
{
    logger.info("Retrieve issue API for issueId: {}", id);

    return issueService.getIssue(id);
}
```

- Formatting styles vary from developer to developer. Coding style changes are also considered a change and can make code merging very difficult.

- To avoid this, the team can have a common coding format.

Use SonarLint



- This is very useful for identifying small bugs and best practices to avoid unnecessary bugs and code quality issues.
- You can install the plugin into your favorite IDE.

This is not the end. There are a lot of small tips and practices. Let's look at them in the next articles. Happy coding !!!

Spring Boot Best Practices Backend Spring Java