Preprints.org

Article

Not peer-reviewed version

Architectural Patterns in REST API withSpring Framework

Galina Kim *, Ruslan Isaev, Gulzada Esenalieva

Posted Date: 10 January 2025

doi: 10.20944/preprints202501.0813.v1

Keywords: MVC; CQRS; microservices; patterns; architecture; REST API; Spring

Preprints.org is a free multidisciplinary platform providing preprint service that is dedicated to making early versions of research outputs permanently available and citable. Preprints posted at Preprints.org appear in Web of Science, Crossref, Google Scholar, Scilit, Europe PMC.

Copyright: This open access article is published under a Creative Commons CC BY 4.0 license, which permit the free download, distribution, and reuse, provided that the author and preprint are cited in any reuse.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 January 2025 doi:10.20944/preprints202501.0813.v1

Disclaimer/Publisher's Note: The statements, opinions, and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions, or products referred to in the content.

Article

Architectural Patterns in REST API with Spring Framework

Galina Kim *, Ruslan Isaev and Gulzada Esenalieva

Ala-Too International University, Ankara 1/8, Bishkek, Kyrgyzstan

* Correspondence: galina.kim@alatoo.edu.kg

Abstract: This article discusses the most popular architectural patterns used in the development of REST APIs based on the Spring Framework, including MVC (Model-View-Controller), microservice architecture, CQRS (Command Query Responsibility Segregation), and other patterns. For each pattern, an analysis of its structure, principles, and implementation approaches is provided. The advantages and disadvantages of each are discussed. The study concludes that the choice of architectural pattern depends on the project's specifics, scale, and performance requirements, suggesting possible approaches for hybrid use of these patterns to improve the flexibility and scalability of REST APIs with the Spring Framework.

Keywords: MVC; CQRS; microservices; patterns; architecture; REST API; Spring

---

1. Introduction

With advancing technology and increasing performance requirements for web applications, REST API development has become a crucial component in the architecture of modern software systems. REST APIs provide an efficient interface for interaction between various services and clients, making them essential for building flexible and scalable applications. However, to design these interfaces effectively, it is important to apply architectural patterns that address various needs, such as scalability, performance, and simplified development.

The Spring Framework, being one of the most popular frameworks for Java application development, provides powerful tools for implementing REST APIs.

Developers often use a variety of architectural patterns to solve different challenges in REST API design. Among these are approaches such as MVC (Model-View-Controller), microservice architecture, and CQRS (Command Query Responsibility Segregation). Each of these patterns has its own characteristics, strengths and weaknesses, which must be considered when selecting a solution for a specific project.

The goal of this article is to review the most popular architectural patterns used in the development of REST APIs with the Spring Framework. We will analyze their structure, principles, and possible implementation approaches, as well as discuss the pros and cons of each. In addition, emphasis will be placed on how pattern selection depends on project specifics, scalability requirements and performance needs.

2. MVC (Model-View-Controller)

Model-View-Controller (MVC) is an architectural pattern commonly used in the development of REST APIs. The Spring MVC framework provides the architecture for the Model-View-Controller pattern. The MVC pattern separates the concerns of an application (input logic, business logic, and data display logic), and by communicating with each other, they create a cohesive application.

• Model: Encapsulates (or combines) the application data, which generally consists of POJOs ("Plain Old Java Objects,"or beans). [1]

© 2025 by the author(s). Distributed under a Creative Commons CC BY license.

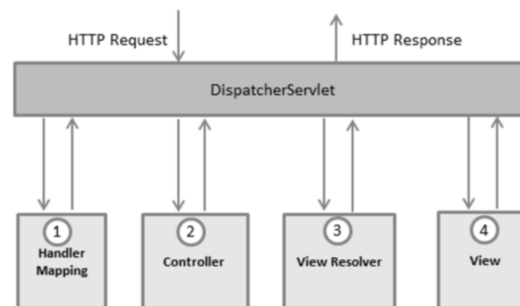
Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 January 2025 doi:10.20944/preprints202501.0813.v1

• View: Responsible for displaying the Model data — typically generating HTML that we see in our browser. [1]

• Controller: Handles the user's request, creates the appropriate Model and passes it for display in the View. [1]

2.1. DispatcherServlet

1. Client Request: When a client sends an HTTP request, DispatcherServlet receives the request.
2. Finding the Appropriate Handler: DispatcherServlet passes the request to HandlerMapping, which finds the appropriate handler (controller).
3. Calling the Handler: DispatcherServlet invokes the controller method associated with the request.
4. Getting the Data Model: The handler can return data, which is passed to the view.
5. Rendering the View: DispatcherServlet passes the model to ViewResolver, which decides which view (e.g., JSP or Thymeleaf) to use for rendering the response.
6. Sending the Response to the Client: Once rendering is complete, DispatcherServlet sends the final response to the client.



2.2. Some Code

Here is an example of a controller that handles requests on the route "/hello" and returns the HTML template "hello.html". Before sending the view, the controller adds data to the model in the form of a key-value pair:

• "message" = "Hello Spring MVC Framework!"

Below is the code of the controller:

```
@Controller
@RequestMapping("/hello")
public class HelloController {
   @RequestMapping(method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

Next, to display data from the model in the view, you can use the Thymeleaf templating engine. Specifically, to output the value of the variable message, added to the model, in the HTML page, the following syntax is used.

Example of an HTML page code that displays model data using Thymeleaf:

```
<!DOCTYPE html>
<html xmlns:th="http://www.thymeleaf.org">
<head>
    <title>Hello Spring MVC</title>
</head>
<body>
    <h1 th:text="${message}">Message will be displayed here</h1>
</body>
</html>
```

In this example: - The th:text attribute is used to inject a value from the model into an HTML element. In this case, the value of message, added in the controller, will be displayed inside the <h1> tag. - If the message value is absent, the fallback text "Message will be displayed here"will be shown.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 January 2025 doi:10.20944/preprints202501.0813.v1

Thus, Thymeleaf allows for easy integration of model data into an HTML template, enabling dynamic content rendering on the page.

2.3. Advantages of Using MVC

• Separation of concerns: Spring MVC helps to separate different parts of the application into three main components: Model, View, and Controller. This simplifies the maintenance and scaling of the application, as each component is responsible for only its specific logic.
• Integration with other Spring components: Spring MVC is part of the broader Spring Framework, allowing for easy integration with other modules, such as Spring Security, Spring Data, and Spring AOP.
• Support for validation and error handling:

2.4. Disadvantages of Using Spring MVC

• Overhead in large applications: In large web applications, especially those with many complex interfaces, the number of controllers and views can increase significantly, leading to difficulties in managing the project. In such cases, approaches like Microservices are often used to break the system into smaller parts, and Spring MVC might not be as suitable for microservice architectures.
• Scaling difficulties in microservices: Spring MVC is ideal for monolithic applications, but its use in microservices architecture may be less efficient compared to other approaches.
• Scalability issues in web applications with many users: In some cases, using Spring MVC for traditional web applications with heavy user sessions can lead to scalability issues if the application is not properly configured to handle high request volumes.

2.5. Conclusion

Spring MVC is well-suited for small traditional web applications where a visual component is essential.

3. Microservices

All applications that are developed can be divided into two architectural camps:
• Monoliths: These are large, complex applications that combine all the system's functionality. This approach has several drawbacks: monoliths are hard to maintain due to tight internal dependencies, they are difficult to scale, and if one part fails, the entire system might stop, as the application is a single entity.
• Microservices: Microservice architecture is the opposite of monolithic. It is based on dividing the system into separate services, each of which performs a specific task. Together, these services work as a unified system, but each can be modified, updated, or restored independently. In case of failure, only one service will be affected, not the entire application, which increases the system's resilience.

3.1. Key Features of Microservice Architecture

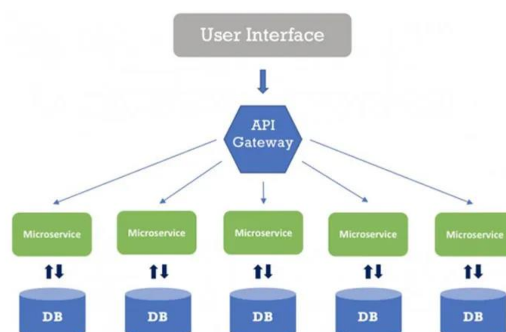
Modularity: Each microservice is responsible for a specific functionality. For example, services can be divided by business domains, such as "users "orders and "payments".
• Isolation and Independence: Each microservice is autonomous and isolated from others, making management, updating, and scaling easier.

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 January 2025 doi:10.20944/preprints202501.0813.v1

• REST API: REST (Representational State Transfer) is a popular protocol for communication between microservices. It uses HTTP methods (GET, POST, PUT, DELETE) to perform operations on resources.
• Communication: REST API provides communication between microservices, offering a unified and easily accessible interface.
• Scalability: Microservices can be scaled independently, allowing the system's performance to be increased in the areas where it is needed the most.
• Technology Independence: Since each microservice is self-contained, it can be developed and deployed on different languages and platforms, if necessary.

3.2. Microservices in Spring

The primary tool for creating microservices in Spring is Spring Cloud, which consists of various components for easy microservice development:
• Eureka: Provides service discovery, allowing services to register and find each other.
• Ribbon: Client-side load balancing that works with Eureka and distributes traffic across service instances.
• Hystrix: Ensures resilience and fault tolerance by providing a "circuit breaker" to handle failures in communication between services.
• Spring Cloud Config: Allows managing configurations for all microservices from a single location.
• Gateway: An API gateway that handles client requests and routes them to the appropriate services.



3.3. Advantages of Microservices Architecture

• Fault tolerance: If one service fails, the whole application won't crash.

• Service autonomy: It is easier to fix individual services rather than dealing with the entire application. The testing process is also simplified.

3.4. Disadvantages of Microservices Architecture

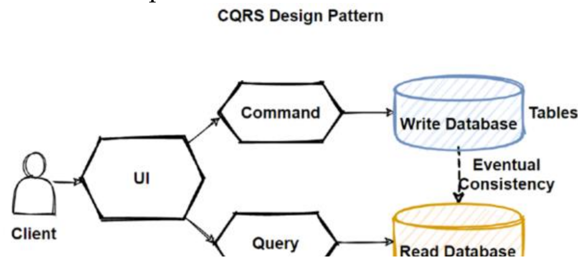
• Breaking a monolith into microservices: Although it may seem easy at first to divide a logically unified monolithic application into separate services, it can often be a complex technical challenge.
• Monitoring challenges: A monolith is one unit, so tracking its performance is easier. Microservices can number in the hundreds or even thousands, making it physically impossible to monitor each one. This requires a strong focus on management and monitoring systems.[4]
• Each service can be written in completely different technologies, languages, and by different developers: Configuring all microservices to work together and maintaining a "zoo"of technologies can be challenging.[4]
• Deployment complexities: To meet fault tolerance requirements, microservices must be deployed on separate servers. The approach of "take the app, install it, and run it"does not work here. Orchestration and deployment systems are needed.[4]

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 January 2025 doi:10.20944/preprints202501.0813.v1

4. CQRS (Command Query Responsibility Segregation)

CQRS is a design pattern that separates read (Query) and write (Command) operations. Its main idea is that commands and queries are handled differently, as they often have different performance, data structure, and security requirements.

This pattern is based on the CQS (Command Query Separation)pattern.



Queries return results and do not alter the observable state of the system.Commands change the state of the system but do not necessarily return a value. [2]

4.1. Main Components

• Command: A command is responsible for changing the system's state and is responsible for creating, modifying or deleting resources.

• Query: A query is an operation for retrieving data without changing the system — reading data.

• Command Handler: Executes commands, leading to changes in the system.

• Query Handler: Processes queries, retrieves data from the system, and presents it in the required format.

4.2. CQRS in Spring Boot

Entity :

```
@Entity
public class Task {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String title;
    private boolean completed;
    // Constructors, getters, and setters
}
```

CommandModel:

```
public class CreateTaskCommand {
    private String title;

    // Getter and setter for title
}
```

CommandHandler:

```
@Service
public class TaskCommandHandler {
    @Autowired
    private TaskRepository taskRepository;

    @Transactional
    public Long handle(CreateTaskCommand command) {
        Task task = new Task();
        task.setTitle(command.getTitle());
        task.setCompleted(false);
        taskRepository.save(task);
        return task.getId();
    }
}
```

Query Model:

```
public class TaskDTO {
    private Long id;
    private String title;
    private boolean completed;

    // Constructors, getters, and setters
}
```

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 January 2025 doi:10.20944/preprints202501.0813.v1

Query Handler:

```
@Service
public class TaskQueryHandler {
    @Autowired
    private TaskRepository taskRepository;

    public List<TaskDTO> getAllTasks() {
        List<Task> tasks = taskRepository.findAll();
        return tasks.stream()
                .map(task -> new TaskDTO(task.getId(),
                    task.getTitle(), task.isCompleted()))
                .collect(Collectors.toList());
    }
}
```

TaskController:

```
@RestController
@RequestMapping("/tasks")
public class TaskController {
    @Autowired
    private TaskCommandHandler commandHandler;
    @Autowired
    private TaskQueryHandler queryHandler;

    @PostMapping
    public Long createTask(@RequestBody CreateTaskCommand command)
        {
        return commandHandler.handle(command);
    }

    @GetMapping
    public List<TaskDTO> getAllTasks() {
        return queryHandler.getAllTasks();
    }
}
```

4.3. Advantages of CQRS

• Scalability: CQRS allows independent scaling of read and write operations. This is especially useful in systems where read and write loads significantly differ. You can allocate more resources to optimize query performance while maintaining the efficiency of write operations. [3]

• Improved Security: CQRS enables the implementation of different security policies for reading and writing operations. You can apply stricter controls for commands, ensuring only authorized users can perform modifications..

• Improved Maintainability: CQRS simplifies the codebase by separating concerns. This separation leads to cleaner and more maintainable code, as commands and queries do not interfere with each other. [3]

• Greater Flexibility: With commands and queries being distinct, you can independently fine-tune your data storage and retrieval approaches. This is particularly advantageous when using various data storage solutions or aiming to improve system performance.

4.4. Disadvantages of CQRS

• Increased Complexity: Implementing CQRS can complicate your system. You need to manage the flow of data between command and query models, potentially duplicating data for different models. [3]

• Additional Learning: Developers who are new to CQRS might face a learning curve when adopting this pattern. Grasping the concepts and implementing them accurately can be difficult.

5. Summary and Conclusions

Of course, these are not all the patterns that exist; there are more and more emerging over time. They are indispensable, as they play a key role in creating high-quality products in today's IT market. It is important to note that when selecting the right patterns for creating a REST API, the main goal

Preprints.org (www.preprints.org) | NOT PEER-REVIEWED | Posted: 10 January 2025 doi:10.20944/preprints202501.0813.v1

becomes simplifying the code and optimizing development processes, as well as ensuring the system's flexibility and scalability. It is essential to remember that each pattern addresses specific problems and can be applied in different contexts.

However, it is crucial to maintain balance: excessive use of patterns can lead to a complicated architecture and an unnecessary amount of abstractions, making the project's support and development more difficult. Therefore, the choice of patterns should be thoughtful, based on the specifics of the task, as well as the maturity of the team and the project's architecture.

Thus, the right selection and use of architectural patterns is an integral part of successfully creating a REST API, and understanding and applying them will be an important step for any development team aiming to create high-quality and efficient solutions.

References

1. Habr. "Статья: Spring MVC — основные принципы," https://habr.com/ru/articles/336816/, accessed 10th November 2024.
2. Baeldung. "CQRS and Event Sourcing in Java," https://www.baeldung.com/cqrs-event-sourcing-java, accessed 25th December 2024.
3. JackyNote. "Article: Understanding CQRS Pattern: Pros, Cons, and a Spring Boot Example," https://dev.to/jackynote/understanding-cqrs-pattern-pros-cons-and-a-spring-boot-example-3flb, accessed 9th November 2024.
4. Habr. "Микросервисы: плюсы, минусы, когда и зачем внедрять," https://habr.com/ru/companies/slurm/articles/674600/, accessed 25th December 2024.
5. Vlad Mishustin. "Микросервисы Простыми Словами," YouTube, https://www.youtube.com/watch?v=XtOJZ1T3qw4, accessed 25th December 2024.

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.