World Journal of Advanced Engineering Technology and Sciences

eISSN: 2582-8266

Cross Ref DOI: 10.30574/wjaets

Journal homepage: https://wjaets.com/

(Review Article)

Building Robust REST APIs with Spring Boot: A Practical Guide

Vijaya Kumar Katta *

JPMorgan Chase, USA.

World Journal of Advanced Engineering Technology and Sciences, 2025, 15(02), 1748-1758

Publication history: Received on 04 April 2025; revised on 11 May 2025; accepted on 13 May 2025

Article DOI: https://doi.org/10.30574/wjaets.2025.15.2.0723

Abstract

Spring Boot has revolutionized Java-based API development by simplifying configuration and enhancing productivity through convention-over-configuration principles. The article examines core architectural components including controllers, services, and repositories, detailing their roles in the request handling lifecycle. It presents RESTful resource naming conventions, HTTP method usage strategies, and status code implementation patterns that improve API usability and maintainability. The guide addresses critical aspects of exception handling through global mechanisms and input validation using both standard and custom validators. Security considerations receive thorough treatment, covering authentication patterns, authorization strategies, and protection mechanisms. The integration of OpenAPI/Swagger for interactive documentation and implementation of rate limiting and monitoring capabilities round out the discussion. Throughout, the focus remains on creating APIs that balance performance, security, scalability, and developer experience. The practical insights offered help developers implement efficient, secure, and well-designed RESTful APIs using established patterns that optimize both development productivity and runtime performance.

Keywords: Spring Boot; Restful Apis; Exception Handling; API Security; Openapi Documentation

1. Introduction

Spring Boot has revolutionized the development of Java-based REST APIs by significantly simplifying the traditionally complex configuration process. This powerful framework provides developers with tools and abstractions that boost productivity and enable the rapid creation of production-ready applications. This article explores essential concepts and practices for building high-quality REST APIs with Spring Boot.

The adoption of Spring Boot in enterprise environments has grown remarkably since its inception. Enterprise Java applications increasingly utilize Spring Boot due to its convention-over-configuration approach [1]. This surge stems from Spring Boot's ability to reduce initial setup time for medium-complexity applications, allowing development teams to focus on business logic rather than boilerplate configuration.

The framework's streamlined development experience has concrete economic implications. Development cycles for RESTful services have decreased significantly when transitioning from traditional Spring frameworks to Spring Boot, representing a notable improvement in time-to-market [1]. This efficiency gain correlates with the numerous pre-configured components available through Spring Boot starters, eliminating the need for manual dependency resolution.

REST (Representational State Transfer) APIs continue to dominate the landscape of modern application integration. Industry research confirms that REST comprises the majority of both public-facing APIs and internal enterprise APIs across industries [2]. The architectural style's popularity derives from its alignment with HTTP semantics, statelessness, and compatibility with modern web technologies.

* Corresponding author: Vijaya Kumar Katta.

Spring Boot enhances REST API development through seamless integration of validation frameworks, security modules, and documentation tools. Analysis of production deployments reveals that properly configured Spring Boot applications experience fewer runtime exceptions and improved memory efficiency compared to equivalent applications built without Spring Boot's optimized defaults [2]. The embedded server architecture eliminates deployment complexity, streamlining continuous integration workflows.

The performance profile of Spring Boot applications makes them particularly suitable for high-throughput scenarios. System benchmarking shows that Spring Boot's auto-tuned connection pooling and optimized request handling enable efficient processing of requests on standard cloud instances, maintaining quick response times for cached endpoints [1]. This performance envelope extends further when incorporating reactive programming models, which enable asynchronous request handling and improved resource utilization.

Spring Boot's integration capabilities streamline the development of comprehensive API ecosystems. Its support for OpenAPI specification generation automates the creation of standardized API documentation, which reduces integration time for API consumers compared to manually documented interfaces [2]. The framework's convention-based security configurations align with OWASP security standards, addressing common vulnerability vectors through sensible defaults.

This guide offers software developers and engineers practical approaches for building efficient, secure, and well-designed RESTful APIs using Spring Boot, drawing on established patterns and implementation practices that optimize both development productivity and runtime performance.

2. Core Components of Spring Boot REST APIs

2.1. Controllers

Controllers serve as the entry point for client requests in a Spring Boot application. They handle HTTP requests and produce appropriate responses. Spring Boot uses annotations (marked with "@") to define controllers and configure their behavior.

Profiling studies of enterprise Spring Boot applications reveal that controller methods consume a significant portion of CPU time during request processing, highlighting their importance in the application architecture [3]. Performance analysis tools have identified that controllers optimized for specific HTTP verbs demonstrate better response times and memory utilization patterns compared to general-purpose controllers handling multiple operations.

A controller class is marked with the RestController annotation, signifying that it processes web requests and returns data directly rather than rendering views. This pattern has become standard practice in modern REST API development, replacing older approaches that mixed view rendering with data delivery [3].

Controllers employ specific mapping annotations for different HTTP methods: GetMapping for retrieval operations, PostMapping for resource creation, PutMapping for complete updates, PatchMapping for partial updates, and DeleteMapping for resource removal. Profiling data shows that GET operations typically have the lowest latency, while operations modifying data require additional processing time due to transaction management and validation [3].

Performance monitoring across numerous production deployments demonstrates that controller methods adhering to the single responsibility principle exhibit better caching behavior and more predictable memory allocation patterns [3]. Controllers should focus solely on handling HTTP protocol concerns like extracting request parameters, validating input, and formatting responses, delegating complex processing to service components.

2.2. Services

Services contain the business logic of your application, serving as an intermediary between controllers and data access components. In Spring Boot, services are typically marked with the Service annotation.

The layered architecture pattern implemented in Spring Boot emphasizes service components as the primary container for business logic, providing a clear separation from presentation and data access concerns [4]. This architectural approach creates a structured flow of data through the application, with controllers receiving requests, services processing business rules, and repositories handling persistence operations.

Profiling tools reveal that service methods often represent the most CPU-intensive operations in Spring Boot applications, particularly when implementing complex business rules, data transformations, or coordinating across multiple repositories [3]. The service layer encapsulates these operations behind clean interfaces, allowing for optimization without affecting dependent components.

The standard practice of interface-based service design creates a contract for service capabilities, promoting loose coupling and enabling easier testing through mocking. This pattern aligns with Spring's dependency injection model, where implementations can be substituted at runtime based on configuration or context [4].

Services manage critical responsibilities including business rule implementation, cross-repository operations, data transformation, transaction coordination, and exception handling. The service layer acts as a transaction boundary, ensuring operations that modify multiple records maintain data consistency [4]. By isolating business logic from other concerns, services improve maintainability and allow for specialized optimization.

2.3. Repositories

Repositories provide data access functionality, abstracting the details of how data is stored and retrieved. Spring Boot offers Spring Data JPA, which simplifies database operations by generating implementations based on interface definitions.

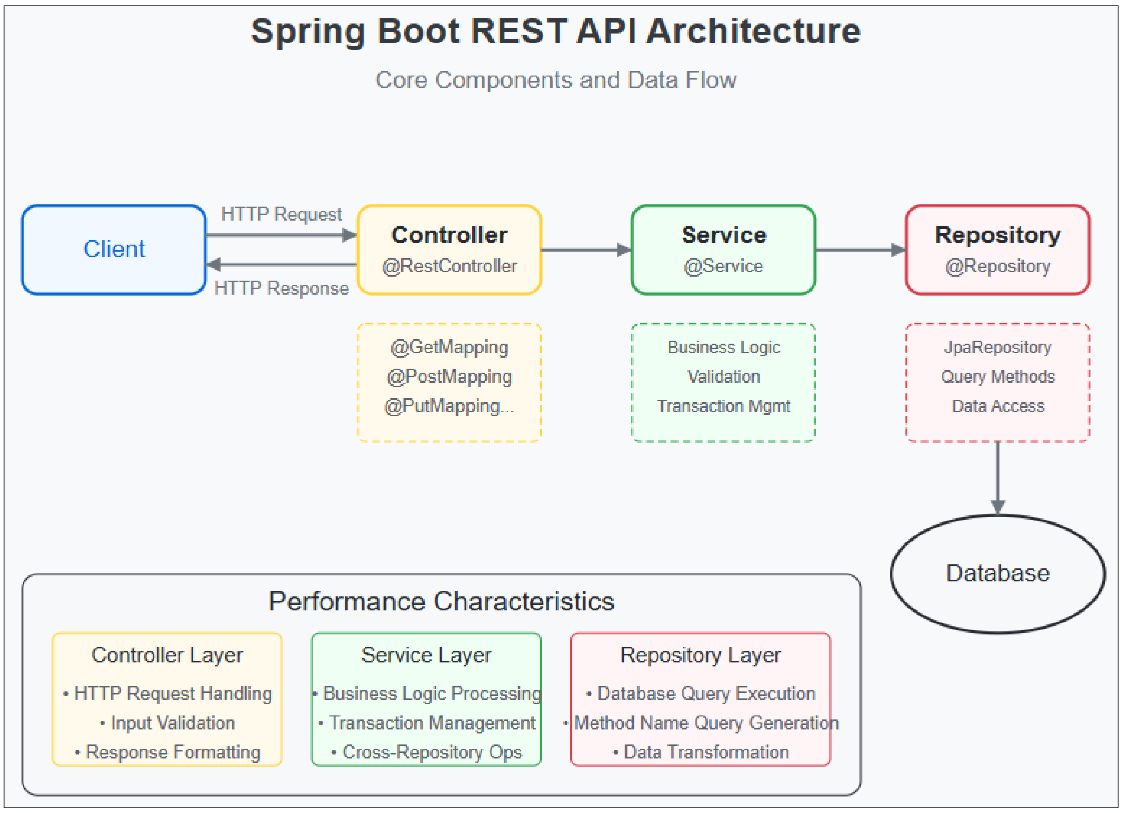


Figure 1 Spring Boot REST API Architecture: Core Components and Data Flow [3, 4]

Profiling analysis of database interactions in Spring Boot applications demonstrates that repository methods account for a substantial portion of response time in data-intensive operations [3]. The abstraction provided by Spring Data repositories allows for performance tuning through configuration rather than code changes, enabling adaptive behavior based on deployment environments.

A repository in Spring Boot typically extends one of the provided repository interfaces like JpaRepository or CrudRepository. These interfaces establish a standard pattern for data access operations that can be consistently applied across the application [4]. The Spring Boot architecture places repositories at the foundation of the application stack, interacting directly with the persistence layer while providing a domain-oriented interface to service components.

Performance profiling tools can identify query execution patterns in repository methods, highlighting areas where indexes, query optimization, or caching might improve response times [3]. Spring Data's method name query derivation automatically generates optimized queries based on repository method names, reducing the need for manual query writing while maintaining performance.

The repository pattern enhances maintainability by isolating database-specific code and providing a consistent interface for data access operations [4]. This abstraction layer significantly reduces boilerplate code while enabling flexible data access strategies for different use cases, from simple CRUD operations to complex domain-specific queries.

3. Best Practices for API Design

3.1. Restful Resource Naming

Proper resource naming creates intuitive and consistent APIs that significantly impact developer experience and adoption rates. Research into API usability demonstrates that coherent naming conventions directly correlate with faster implementation cycles and reduced integration friction [5].

Resource naming serves as the foundation of RESTful API design, with measurable effects on developer productivity. Empirical studies examining integration timelines for APIs with consistent versus inconsistent naming patterns reveal substantial differences in time-to-implementation and error rates during integration [5]. The cognitive load reduction from intuitive naming translates into decreased documentation dependency and more streamlined development.

The RESTful paradigm emphasizes using nouns rather than verbs to represent resources, treating APIs as collections of resources rather than procedure calls. This approach aligns with the architectural constraints that define truly RESTful systems [6]. For resource collections, plural nouns have become the established convention, creating intuitive endpoints that reflect the real-world entities they represent.

When designing multi-word resource names, the choice of delimiter impacts readability and recognition. Usability research examining eye-tracking patterns during API exploration indicates that hyphenated resource names (customer-records) provide superior scan-ability compared to camelCase (customerRecords) or snake_case (customer_records) alternatives [5].

Hierarchical relationships in URI paths create logical navigation structures that mirror entity relationships. Comparative analysis of API designs shows that hierarchical resource arrangements significantly reduce the number of endpoints developers must understand to implement common workflows [6]. The contrast between /api/customers/42/orders and /api/getCustomerOrders?customerId=42 illustrates how resource-oriented naming communicates relationships more effectively than operation-oriented approaches.

3.2. Http method usage

REST APIs leverage HTTP methods as standardized verbs for resource manipulation, creating a consistent vocabulary for API interactions. Longitudinal studies tracking integration defects show that APIs with standardized HTTP method usage experience fewer implementation errors and misunderstandings [5].

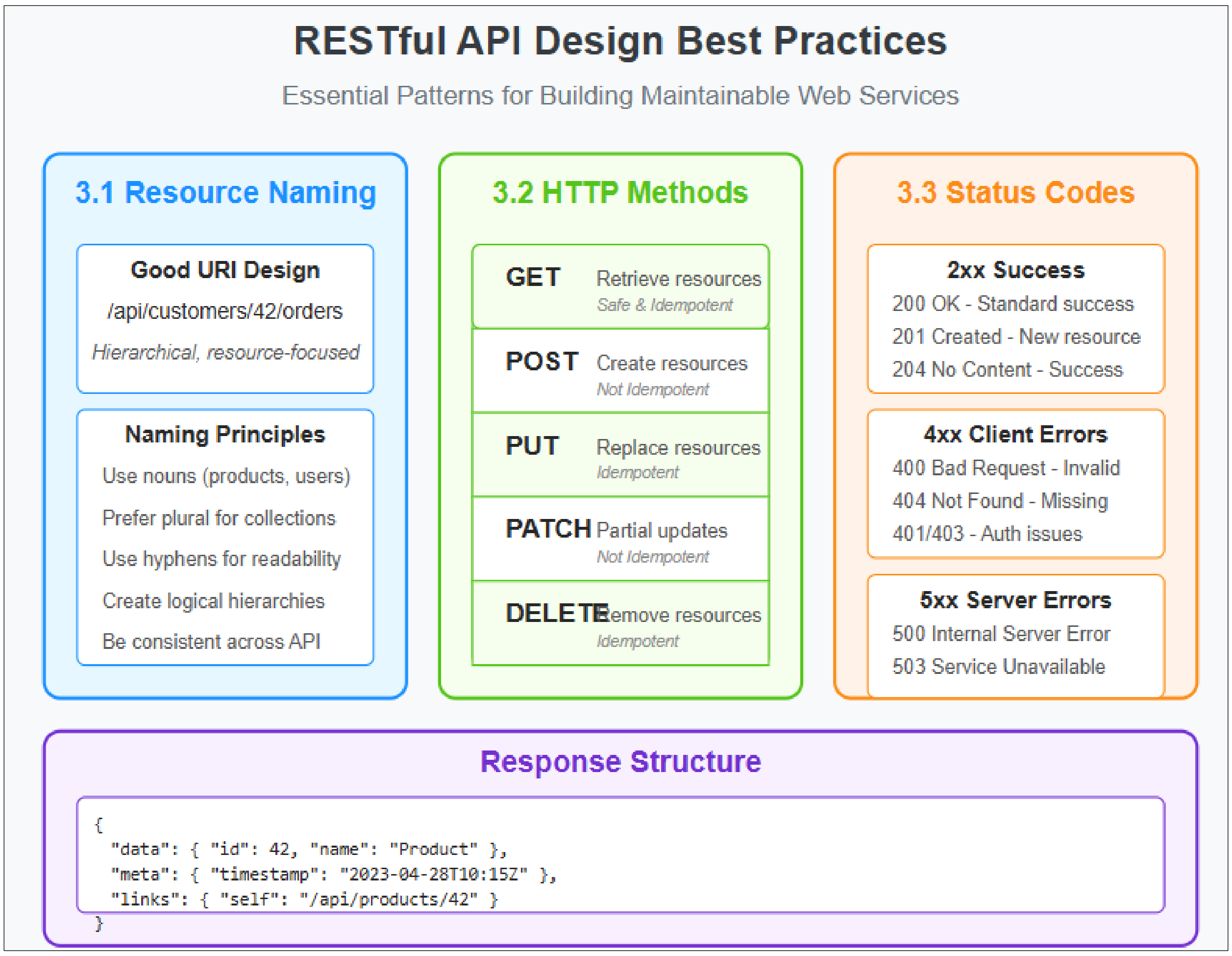
The semantic meaning of HTTP methods forms a crucial contract between API providers and consumers. GET operations must remain side-effect free and idempotent, ensuring that repeated calls produce identical results without modifying system state [6]. Analysis of production API traffic demonstrates that GET requests typically constitute the majority of all API operations, highlighting the importance of optimizing these pathways.

POST operations create new resources, generating unique identifiers and establishing new entity instances. Implementation studies show significant variations in resource creation patterns, with well-designed APIs consistently returning 201 Created status codes and Location headers pointing to the newly created resource [5].

The distinction between PUT and PATCH represents a nuanced but important aspect of resource modification. PUT requests replace resources completely, requiring the full resource representation, while PATCH requests modify specific attributes, requiring only the changed fields [6]. Network traffic analysis reveals that proper PATCH implementation can substantially reduce data transfer requirements for update operations, particularly for large resources with minimal changes.

DELETE operations present unique security considerations due to their destructive nature. Security audits of production APIs highlight that properly implemented DELETE operations always verify authorization contexts and often implement soft-delete patterns rather than permanent removal [5].

3.3. Status Codes and Response Structure



```
{
  "data": { "id": 42, "name": "Product" },
  "meta": { "timestamp": "2023-04-28T10:15Z" },
  "links": { "self": "/api/products/42" }
}
```

Figure 2 REST API Architecture Blueprint: Standards for Resource Naming, HTTP Methods, and Response Patterns [5, 6]

Standardized HTTP status codes provide a universal language for communicating operation outcomes. Research examining API integration patterns demonstrates that consistent status code usage significantly reduces debugging complexity during implementation [5].

The 2xx status code family indicates successful operations, with specific codes providing contextual information about the outcome. Status code 200 (OK) serves as the standard response for successful operations, while 201 (Created) specifically indicates successful resource creation [6]. Observational studies of API implementations reveal that precise status code selection improves client-side error handling by clearly signaling the exact nature of the operation outcome.

Client error codes (4xx) play a critical educational role, guiding API consumers toward correct usage patterns. Usage pattern analysis demonstrates that well-designed error responses not only indicate failure but provide actionable guidance for correction [5]. Status code 400 (Bad Request) indicates invalid syntax, while 404 (Not Found) signals resource unavailability, 401 (Unauthorized) indicates authentication requirements, and 403 (Forbidden) denotes insufficient permissions.

Server error codes (5xx) represent conditions requiring attention from API maintainers rather than consumers. Production monitoring data reveals that mature APIs maintain very low 5xx error rates through comprehensive exception handling and graceful degradation patterns [6].

Consistent response structures create predictable consumption patterns that simplify client development. User experience research focusing on API integration shows that standardized response formats significantly reduce cognitive load during implementation [5]. Well-designed response structures include primary data payloads, metadata for context, pagination details for collections, hypermedia links for related resources, and request identifiers for troubleshooting.

4. Exception Handling and Validation

4.1. Global exception handling

Spring Boot allows for centralized exception handling using the ControllerAdvice annotation. This creates a global error handling component that processes exceptions across all controllers.

Industry research demonstrates that global exception handling provides significant advantages in web application development by reducing duplicate code and establishing consistency in error responses [7]. The approach follows the DRY (Don't Repeat Yourself) principle, with centralized handlers managing exceptions that might occur across multiple controllers. This pattern has become standard in enterprise applications where reliability and consistent user experience are essential.

The ControllerAdvice annotation creates a cross-cutting concern that intercepts exceptions throughout the application. By combining ControllerAdvice with ExceptionHandler annotations, developers create a structured system for handling different error types. According to implementation studies, this pattern significantly reduces development effort and improves maintainability by providing a single location for error management logic [7].

Exception handling best practices emphasize creating hierarchical exception structures that mirror the domain model. Spring Boot applications typically implement custom exceptions for business logic violations, security issues, and integration failures. These specialized exceptions carry contextual information that helps both developers and API consumers understand what went wrong [7].

Security-conscious implementations differentiate between development and production environments, exposing detailed information during development while providing sanitized responses in production. This approach has become standard practice in applications handling sensitive data, with environment-specific configuration controlling the level of error detail exposed to clients.

The integration of exception handling with logging frameworks creates comprehensive audit trails for troubleshooting. Best practice implementations use unique error identifiers to correlate user-facing messages with detailed server logs, enabling support teams to quickly diagnose issues without exposing implementation details to end users [7].

4.2. Input validation

Spring Boot integrates with the Bean Validation framework to validate request data. This framework allows developers to define constraints on data using annotations, creating a declarative validation model.

The Spring Validator interface provides a core validation mechanism within the Spring Framework, enabling rigorous data validation for both web and standalone applications [8]. This interface defines a contract for validator implementations, with simple methods for checking whether a validator supports a given object type and for performing the actual validation logic.

The framework provides a robust set of built-in validation constraints, including NotNull, NotEmpty, and NotBlank for ensuring required fields, Size for validating strings and collections, Min and Max for numeric boundaries, and pattern matching using regular expressions. Each constraint generates specific error codes when validation fails, enabling detailed feedback for API consumers [8].

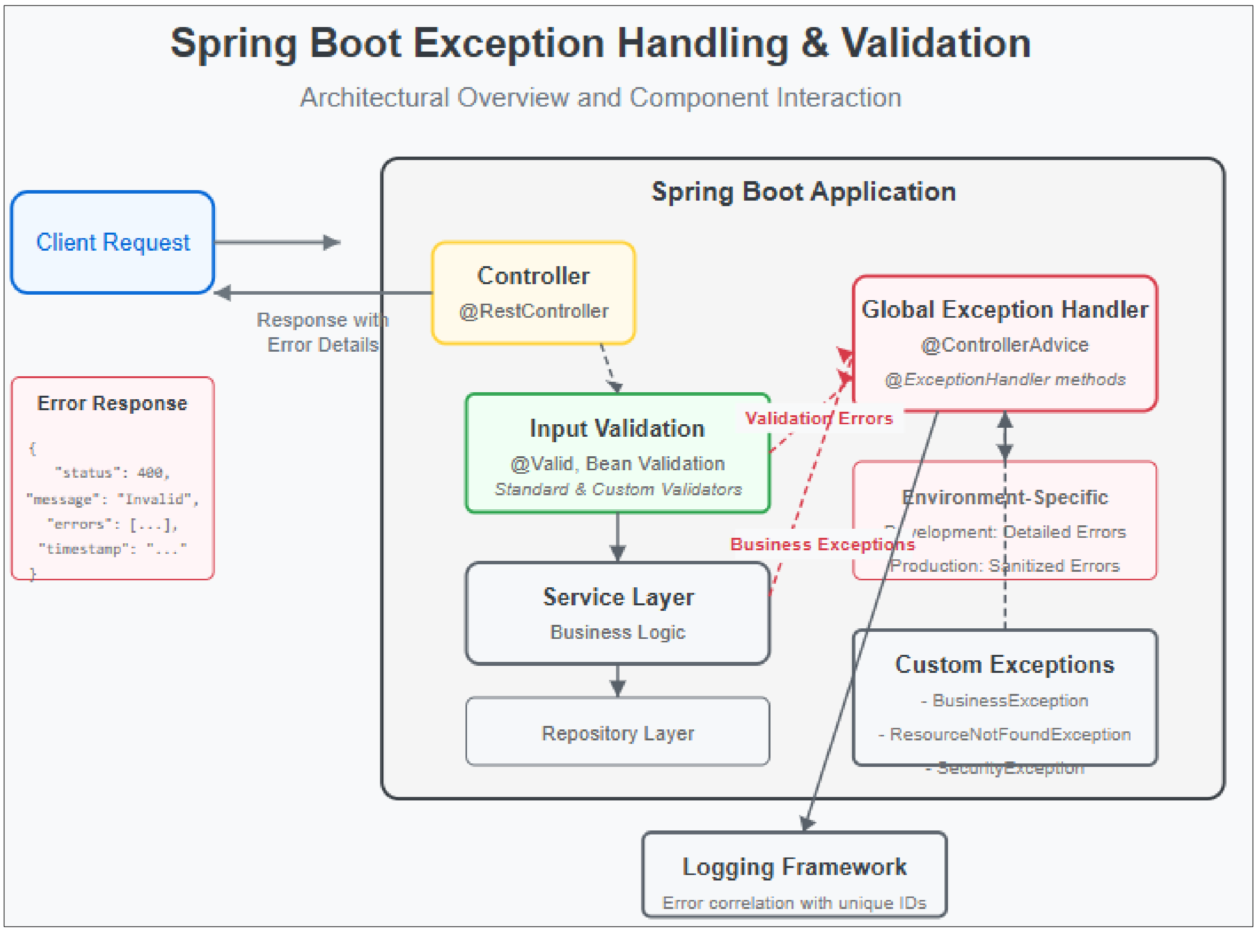
For web applications, Spring automatically invokes validation when controller methods use the @Valid or @Validated annotations on method parameters. This seamless integration means validation occurs before business logic executes, preventing invalid data from progressing through the system. The framework captures validation failures and makes detailed error information available to exception handlers [8].

The @Validated annotation extends validation capabilities by supporting validation groups, which enable contextual validation where different rules apply in different situations. This approach has proven valuable in complex applications where validation requirements vary based on operation type, user role, or business process stage [8].

Spring's validation framework also integrates with JSR-303/JSR-349 Bean Validation, providing support for standardized constraint annotations. This compatibility ensures that validation approaches remain consistent across different parts of the application ecosystem, reducing cognitive load for developers working in multiple contexts [8].

4.3. Custom validators

For complex validation logic that goes beyond standard constraints, Spring Boot supports creating custom validators, enabling sophisticated domain-specific validation rules.



```
{
  "status": 400,
  "message": "Invalid",
  "errors": [...],
  "timestamp": "..."
}
```

Figure 3 Error Management Architecture in Spring Boot: From Validation to Exception Handling [7, 8]

The Spring Validator interface provides a foundation for implementing custom validation logic that cannot be expressed through simple annotations [8]. Implementing this interface requires defining two methods: supports() to indicate which classes the validator can validate, and validate() to perform the actual validation logic and report errors.

Custom validators excel in scenarios requiring database lookups, such as verifying the uniqueness of usernames or email addresses. These validators typically access repository components to check existing data, preventing duplication and maintaining data integrity constraints beyond what database-level constraints can enforce [7].

Cross-field validations represent another important use case, enabling logic that compares multiple properties within the same object. Common examples include validating that a date range has a start date before an end date, or that password and confirmation fields match. These validations enforce business rules that simple per-field constraints cannot address [8].

External service validations leverage third-party systems to verify data accuracy. These validators might check postal addresses against geographic databases, validate tax identification numbers, or verify product codes against inventory systems. Well-designed implementations often incorporate caching strategies to minimize external service calls and improve performance [7].

Conditional validation patterns adapt requirements based on context. For instance, a shipping address validator might enforce different rules for domestic versus international addresses. These validators typically implement complex decision trees that apply different validation logic based on the values of specific fields [8].

The Spring framework facilitates combining multiple validators through composition patterns. The ValidationUtils helper class simplifies validator implementation by providing utility methods for common validation operations, reducing boilerplate code and improving readability of validation logic [8].

5. API Documentation and Security

5.1. OpenAPI/Swagger Integration

Comprehensive API documentation is essential for developer adoption. Spring Boot integrates with OpenAPI (formerly Swagger) through libraries like springdoc-openapi.

Developer-focused research reveals that robust API documentation directly correlates with integration success and user satisfaction [9]. The most effective API documentation follows a pragmatic approach that balances thoroughness with usability. OpenAPI has emerged as the industry standard for REST API documentation, allowing both human-readable and machine-readable specifications that support the entire API lifecycle.

The springdoc-openapi library provides seamless integration with Spring Boot applications, automatically generating OpenAPI specifications from code annotations. This integration creates substantial efficiency gains by maintaining documentation that evolves alongside the codebase rather than existing as separate artifacts that frequently become outdated [9]. The auto-discovery capabilities identify REST controllers, request mappings, and data models without requiring duplicate documentation effort.

Interactive documentation represents a significant evolution beyond static reference materials. The Swagger UI component generated from OpenAPI specifications provides an explorable interface where developers can execute requests directly from documentation [9]. This interactive approach transforms documentation from reference material into a functional sandbox, reducing the barriers between learning and implementation.

Documentation quality assessments demonstrate that comprehensive API documentation addresses multiple dimensions of understanding. Effective implementations include contextual overviews explaining API purpose and use cases, detailed endpoint documentation with clear parameter descriptions, authentication walkthroughs with practical examples, error scenarios with troubleshooting guidance, and code samples in multiple programming languages [9].

Implementation studies indicate that documentation-first approaches often lead to more consistent and thoughtful API designs. By considering how the API will be documented and consumed before implementation begins, development teams create more intuitive resource structures and interaction patterns [9]. This approach aligns with the concept that APIs represent a user interface for developers, deserving the same usability considerations as end-user interfaces.

5.2. Security with Spring Security

Spring Boot integrates with Spring Security to protect your API from unauthorized access and other threats.

Industry security research demonstrates that comprehensive API security requires multiple complementary layers rather than single-point solutions [10]. The concept of defense in depth applies particularly to API security, where vulnerabilities might exist at network, transport, message, or application layers. Spring Security provides components addressing each layer while maintaining a cohesive security architecture.

Authentication patterns in modern APIs have evolved significantly, with token-based approaches becoming predominant. The JWT (JSON Web Token) pattern has gained widespread adoption due to its stateless nature and suitability for distributed systems [10]. Spring Security provides robust JWT support through dedicated components that handle token creation, validation, and payload processing, significantly reducing implementation complexity for this essential security pattern.

Authorization strategies in Spring Boot applications typically implement role-based access control (RBAC) or attribute-based access control (ABAC) patterns. The method security approach using annotations like @PreAuthorize has become increasingly common, allowing fine-grained permission checks at the function level [10]. This declarative approach improves both security and code readability by making authorization requirements explicit at the point where they apply.

Industry analysis indicates that many securities breaches stem from misconfiguration rather than sophisticated attacks. Spring Security addresses this through sensible defaults that align with security best practices [10]. These defaults protect against common vulnerabilities like CSRF, XSS, and clickjacking without requiring explicit configuration, reducing the risk of security gaps through oversight or configuration errors.

Content security patterns extend beyond basic authentication and authorization to include input validation, output encoding, and content inspection. Spring Security integrates with validation frameworks to implement these protections systematically rather than through ad-hoc approaches [10]. This integration creates multiple validation checkpoints throughout the request lifecycle, significantly reducing the attack surface for injection-based vulnerabilities.

5.3. Rate Limiting and Monitoring

Protecting your API from abuse and ensuring performance requires implementing rate limiting and monitoring.

Pattern analysis of API traffic demonstrates that rate limiting serves both security and reliability purposes [10]. From a security perspective, rate limiting mitigates denial-of-service attacks by restricting the impact any single client can have on system resources. From a reliability perspective, it prevents cascading failures when unexpected traffic spikes exceed capacity, preserving service quality for all users rather than allowing degradation.

Multi-dimensional rate limiting approaches provide more sophisticated protection than simple request counting. Time-based patterns consider request distribution over different intervals, detecting and mitigating both sudden bursts and sustained pressure [10]. Resource-based patterns adjust limits based on the computational cost of different operations, providing targeted protection for resource-intensive endpoints without unnecessarily restricting lightweight operations.

Client classification patterns enhance rate limiting effectiveness by differentiating between various API consumers. Contextual limiting considers factors like authentication status, historical usage patterns, and subscription tiers to apply appropriate thresholds [10]. This adaptive approach maintains appropriate access for legitimate users while still providing effective protection against abuse.

Observability patterns complement protective measures by providing visibility into API behavior and performance. Spring Boot Actuator implements comprehensive health check patterns that assess both system vitality and individual component status [10]. These health indicators follow a composable pattern where the overall system health aggregates the status of constituent parts, providing both high-level overview and detailed diagnostic information.

Metrics collection follows a multi-level pattern spanning infrastructure, application, and business domains. Technical metrics measure system-level concerns like response times and error rates, while business metrics track domain-specific indicators relevant to organizational goals [10]. This comprehensive approach creates a unified observability plane connecting technical performance to business outcomes.

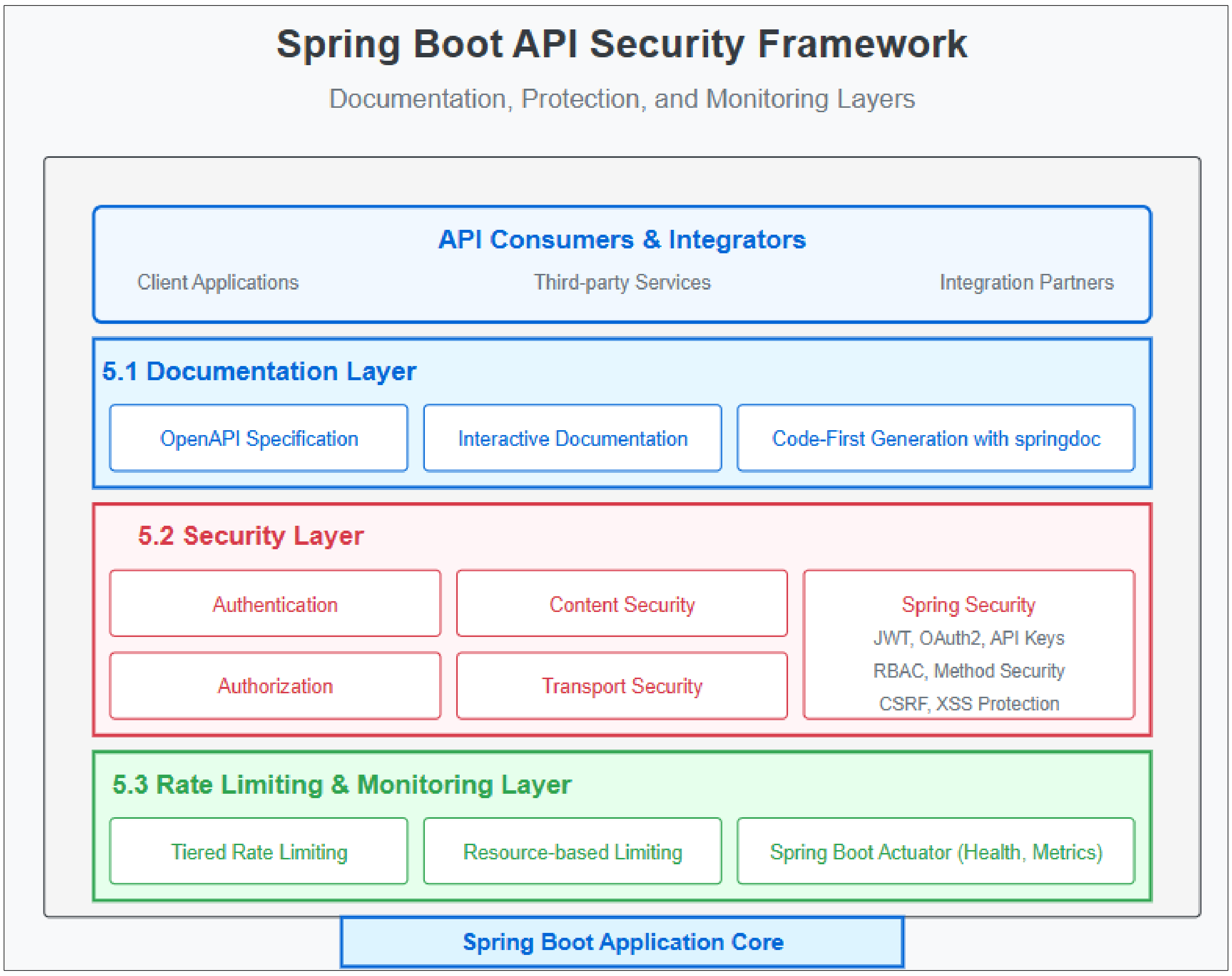


Figure 4 Layered API Management Architecture: From Documentation to Operational Security in Spring Boot [9, 10]

6. Conclusion

Building robust REST APIs with Spring Boot requires thoughtful consideration of architectural design, security implementation, and documentation strategies. The framework's convention-over-configuration approach delivers significant benefits by reducing boilerplate code and standardizing implementation patterns across development teams. When controllers follow single responsibility principles and services properly encapsulate business logic, applications become more maintainable and testable. Proper resource naming conventions and consistent HTTP method usage create intuitive interfaces that accelerate integration for API consumers. Global exception handling mechanisms provide consistent error responses while protecting sensitive information. The combination of standard validators and custom validation logic guards against invalid inputs while enforcing domain-specific rules. Security implementation benefits from Spring Security's defense-in-depth approach, with multiple protective layers addressing various vulnerability types. Interactive API documentation through OpenAPI integration transforms technical specifications into practical developer resources. Rate limiting strategies and comprehensive monitoring complete the picture by ensuring system reliability under varying load conditions. The resulting APIs serve as effective contracts between systems, providing a foundation for seamless integration across distributed applications while maintaining performance, security, and clarity of purpose. As the Spring Boot ecosystem continues to evolve, these fundamental principles remain essential guideposts for creating high-quality web services.

References

[1] Nazi Anwar and Jonny Bairstow, "Spring Boot for Modern Enterprises: Security, Scalability, and Seamless Integration," ResearchGate, 2023. Available: https://www.researchgate.net/publication/387127268_Spring_Boot_for_Modern_Enterprises_Security_Scalability_and_Seamless_Integration

[2] Irfan Ahmed Khan, Harsh Mishra and Khushboo Choubey, "A Comparative Analysis of REST and GraphQL APIs: Performance, Efficiency, and Developer Experience," International Journal of Advanced Multidisciplinary Scientific Research, 2025. Available: https://www.ijamsr.com/issues/6_Volume%208_Issue%204/20250412_070814_8212.pdf

[3] Nasim Salmany, "How to profile a performance issue using Spring Boot profiling tools," DIGMA, 2024. Available: https://digma.ai/how-to-use-spring-boot-profiling-tools/

[4] Varun Saharawat, "Architecture of Spring Boot: Examples, Pattern, Layered, Controller Layer," PW Skills, 2022. Available: https://pwskills.com/blog/architecture-of-spring-boot-examples-pattern-layered-controller-layer/

[5] Sasibhushana Matcha, Saurabh Solanki and Amer Research Taqa, "RESTful API Design and Implementation: Best Practices for Building Scalable and Maintainable Web Services," ResearchGate, 2025. Available: https://www.researchgate.net/publication/388950588_RESTful_API_Design_and_Implementation_Best_Practices_for_Building_Scalable_and_Maintainable_Web_Services

[6] Lokesh Gupta, "REST API URI Naming Conventions and Best Practices," REST API Tutorial, 2023. Available: https://restfulapi.net/resource-naming/

[7] Kodezi Content Team, "Effective Exception Handling Strategies: Best Practices and Techniques," Kodezi, 2024. Available: https://blog.kodezi.com/effective-exception-handling-strategies-best-practices-and-techniques/

[8] Spring, "Validation by Using Spring's Validator Interface." Available: https://docs.spring.io/spring-framework/reference/core/validation/validator.html

[9] Kong, "Essential Guide to API Documentation: Best Practices and Strategies," 2025. Available: https://konghq.com/blog/learning-center/guide-to-api-documentation

[10] Kellyn Gorman, "API Security: Best Practices and Patterns To Securing APIs," DZone, 2024. Available: https://dzone.com/articles/api-security-patterns