Spring Boot Mastery — Progressive Project Roadmap

A step-by-step project-based roadmap to master the Spring ecosystem. Each project builds on previous topics; code is intentionally omitted.

Generated: 2025-10-06 17:42:15

How to use this document

This roadmap lists progressive projects (from basic → advanced), each with:
- Goal and short description
- Functional requirements (what to implement)
- Non-functional requirements (quality attributes to practice)
- Topics covered (Spring projects, patterns, and external technologies)
- Prerequisites and how it builds on earlier projects
- Recommended official guides and docs (links)

Use this as a study & project checklist. Build, test, document, and push each project to a public GitHub repository. Aim to add automated tests, CI, containerization, and monitoring for later projects.

1. Spring Boot Basics — Personal Notes (CRUD)

A minimal, well-documented CRUD application to learn Spring Boot fundamentals, REST controllers, Spring Data JPA, validation, and testing.

Goal: Create a Notes app with REST endpoints for creating, reading, updating, deleting notes and simple tagging/search.

Functional Requirements:

- User registration & authentication (basic form or in-memory users)
- CRUD endpoints for Notes (title, body, tags, createdAt, updatedAt)
- Search by tag and full-text search on title/body (simple)
- Pagination & sorting for notes list
- API documentation (OpenAPI/swagger) and README

Non-functional Requirements:

- Clear project structure and README
- Automated unit & integration tests (JUnit, Spring Test)
- Use H2 in-memory for dev; profile-based properties
- Input validation and meaningful error responses
- Basic logging and exception handling

Topics Covered:

- Spring Boot starters and auto-configuration (Spring Boot).
- Spring MVC (REST controllers, request/response mapping).
- Spring Data JPA & repositories for persistence. (JPA/Hibernate).
- Validation with Jakarta Bean Validation (javax/jakarta.validation).
- Testing with Spring Boot Test, JUnit, MockMvc.

Prerequisites: Java 17+, IDE (IntelliJ/Eclipse), basic Java SE knowledge.
How it builds on earlier projects: — (starting project)

Recommended official resources:

```
- Spring Boot overview: https://docs.spring.io/spring-boot/docs/current/reference/html/
- Accessing Data with JPA (guide): https://spring.io/guides/gs/accessing-data-jpa
- Spring Guides (REST): https://spring.io/guides
```

2. Secure Notes — Add Spring Security (Form-login & JWT)

Add authentication and authorization to the Notes app. Support both session/form login (for browser) and JWT-based token auth for REST clients.

Goal: Protect endpoints, add roles (USER, ADMIN), and implement a JWT token flow for API clients.

Functional Requirements:

- Form-based login for a small web UI or Postman-driven testing
- JWT-based authentication endpoints: /auth/login -> issues JWT, /auth/refresh
- Role-based access: USER can manage own notes, ADMIN can manage all notes
- Password hashing (BCrypt) and secure user storage
- Logout and token revocation (basic)

Non-functional Requirements:

- Secure configuration best-practices (no secrets in source)
- Secure headers, CSRF protection for browser endpoints
- Basic rate-limiting (per endpoint) to mitigate abuse
- Add security-focused tests and integration tests
- Documentation of security design and threat model

Topics Covered:

- Spring Security: authentication, authorization, password encoding, method security.
- JWT patterns and stateless APIs.
- Security testing and common vulnerabilities (CSRF, XSS basics).

Prerequisites: Project 1, understanding of HTTP, cookies, and tokens.
How it builds on earlier projects: Project 1 (adds security on top of CRUD app).

Recommended official resources:

```
- Securing a Web Application (guide): https://spring.io/guides/gs/securing-web
- Spring Security reference:
https://docs.spring.io/spring-security/reference/index.html
```

3. RESTful E■Commerce Backend (Orders, Products, Users)

A more realistic backend: users, products, carts, orders, payments (gateway simulation). Focus on transactional integrity, DTO mapping, caching, validation, and documentation.

Goal: Implement a production-ready API for products and orders, including transactions and idempotency for order creation.

Functional Requirements:

- Product CRUD, search, categories, inventory count
- Cart & order flow: add to cart, checkout, order status lifecycle
- Simulated payment gateway integration (sandbox flow)
- Admin endpoints for inventory management & reports
- API versioning and HATEOAS-enhanced endpoints

Non-functional Requirements:

- Transactional integrity for checkout (ACID where required)
- Caching for product read endpoints (Redis/Caffeine)
- DTOs and API contracts with OpenAPI
- Idempotency for critical operations (order submission)
- Good logging, error handling, and observability

Topics Covered:

- Spring Data JPA advanced: transactions, locking, optimistic/pessimistic locking.
- Spring Cache abstraction and caching providers.
- API design: versioning, HATEOAS, DTOs and mapping (MapStruct optional).
- Integration with external APIs via RestTemplate/WebClient.

Prerequisites: Projects 1–2, knowledge of transactions and database isolation.
How it builds on earlier projects: Adds complexity and real-world concerns to Projects 1–2.

Recommended official resources:

```
- Spring Data JPA reference:
https://docs.spring.io/spring-data/jpa/reference/index.html
- REST HATEOAS guide: https://spring.io/guides/gs/rest-hateoas
- Caching guide: https://spring.io/guides/gs/caching
```

4. Payment & Integration Pipeline (WebClient, Secrets, Idempotency)

Make payment flows robust and safe: async callbacks, secure secret handling, idempotency keys, and integration patterns.

Goal: Implement payment simulation, asynchronous callbacks (webhooks), and resilient external API calls.

Functional Requirements:

- Payment initiation with idempotency keys
- Webhook receiver for asynchronous payment updates
- Retry & backoff for transient API failures
- Secure secret management (external config or a secrets manager)
- Audit logs for payment events

Non-functional Requirements:

- Secure handling of secrets (do NOT store API keys in source)
- Reliable retries and dead-letter handling for failed webhook processing
- High observability for payment flows
- Strong contract tests / consumer-driven contract tests for integrations

Topics Covered:

- Spring WebClient for non-blocking HTTP calls and timeouts.
- Resilience patterns (retries, backoff) — Resilience4j recommended.
- Externalized configuration and secrets (Spring Cloud Config / Vault).

Prerequisites: Project 3, familiarity with async processing and callbacks.
How it builds on earlier projects: Payment flow added to the E-commerce backend (Project 3).

Recommended official resources:

```
- Resilience4j with Spring Boot (docs/tutorials): https://resilience4j.readme.io/
- Spring Cloud Config: https://spring.io/projects/spring-cloud-config
- WebClient guide: https://spring.io/guides/gs/consuming-reactive-rest-service
```

5. Microservices — Product, Order, User + Gateway & Discovery

Split the monolith into microservices. Add an API Gateway, service discovery, centralized config, and inter-service communication with OpenFeign.

Goal: Deploy multiple small services communicating via REST/RPC, register with discovery server, route through gateway, and externalize config.

Functional Requirements:

- Separate Product, Order, User service deployments, each with its own DB
- Service registration & discovery (Eureka or alternative)
- API Gateway routing with authentication & rate limits
- Centralized configuration repository and refresh
- Inter-service calls via OpenFeign/REST with fallback strategies

Non-functional Requirements:

- Service isolation and independent deployability
- Resilience with circuit breakers and bulkheads
- Distributed tracing and correlation IDs
- CI/CD pipelines for individual services

Topics Covered:

- Spring Cloud: service discovery, config server, gateway, distributed patterns.
- OpenFeign/RestTemplate/WebClient for inter-service calls.
- Resilience4j and circuit breaker patterns.
- Distributed tracing and observability (Micrometer Tracing / Sleuth legacy notes).

Prerequisites: Projects 1–4. Docker and basic CI knowledge.
How it builds on earlier projects: Refactors Project 3 into multiple services and adds infra patterns.

Recommended official resources:

```
- Spring Cloud projects overview: https://spring.io/projects/spring-cloud
- Building a Gateway (guide): https://spring.io/guides/gs/gateway
- Spring Cloud Reference:
https://docs.spring.io/spring-cloud/docs/current/reference/html/
```

6. Reactive Social Feed — WebFlux & Reactive Data

Build a reactive social feed (posts, live updates, likes) using Spring WebFlux and reactive data stores.

Goal: Create a highly-concurrent, non-blocking social feed backend with streaming endpoints.

Functional Requirements:

- Reactive endpoints for posting and streaming feed updates
- Reactive repositories (R2DBC or Reactive MongoDB)
- Backpressure-aware streaming to clients (Server-Sent Events or WebFlux streams)
- Reactive security considerations

Non-functional Requirements:

- High concurrency with low thread usage (non-blocking model)
- Performance testing with load tools (wrk, Gatling)
- Monitor Reactor metrics and resource usage
- Document differences vs imperative stack (guides)

Topics Covered:

- Spring WebFlux and Project Reactor (Flux/Mono, operators).
- Reactive persistence with R2DBC or reactive drivers for NoSQL.
- Non-blocking WebClient and streaming responses.

Prerequisites: Comfort with asynchronous programming and Projects 1–5.
How it builds on earlier projects: Parallel reactive alternative to previous imperative services; reuse domain logic.

Recommended official resources:

```
- Reactive RESTful Web Service guide: https://spring.io/guides/gs/reactive-rest-service
- Spring WebFlux docs:
https://docs.spring.io/spring-framework/reference/web/webflux.html
```

7. Batch ETL Pipeline — Scheduled Data Jobs

Extract–transform–load jobs for processing large datasets (imports, nightly reports) using Spring Batch.

Goal: Implement resilient batch jobs with chunking, restartability, and reporting.

Functional Requirements:

- Jobs to ingest CSV/JSON into DB with chunk processing
- Retry, skip, and job restart behavior for failures
- Partitioning or parallel step execution for scale
- Reporting & job dashboard (history of runs, success/failure)

Non-functional Requirements:

- High-throughput processing and resource isolation
- Idempotent job design and checkpointing
- Observability of job metrics and durations
- Secure handling of input files and sensitive data

Topics Covered:

- Spring Batch core concepts: job, step, chunk, reader, processor, writer.
- Scaling batch jobs using partitioning and parallel steps.

Prerequisites: Projects 1–4, familiarity with I/O and databases.
How it builds on earlier projects: Adds offline/background processing skills to your backend.

Recommended official resources:

```
- Batch Processing guide: https://spring.io/guides/gs/batch-processing
- Spring Batch project page: https://spring.io/projects/spring-batch
```

8. Event-driven System — Kafka / Spring Cloud Stream

Make the system event-driven using Kafka (or RabbitMQ) and Spring Cloud Stream for decoupled services and scalable event processing.

Goal: Replace direct synchronous calls with events for order updates, inventory changes, and user notifications.

Functional Requirements:

- Publish domain events (OrderCreated, InventoryReserved) to topics
- Consumers that react to events and update read models
- Idempotent consumers and dead-letter handling
- Schema/versioning strategy for evolving events

Non-functional Requirements:

- High-throughput, durable messaging with at-least-once or exactly-once semantics (where possible)
- Monitoring consumer lag and throughput
- Backpressure and partitioning strategies
- Secure communication to broker (TLS, SASL if required)

Topics Covered:

- Spring Cloud Stream and Kafka binder (topics, bindings, consumers).
- Event-driven design, schema evolution, and idempotency.
- Monitoring Kafka consumer lag and broker health.

Prerequisites: Projects 3–6, basic knowledge of messaging systems.
How it builds on earlier projects: Decouples services in the microservices architecture (Project 5).

Recommended official resources:

```
- Spring Cloud Stream guide: https://spring.io/guides/gs/spring-cloud-stream
- Kafka binder reference: https://docs.spring.io/spring-cloud-stream/docs/current/refer
ence/html/spring-cloud-stream-binder-kafka.html
```

9. Real-time Chat — WebSockets & STOMP

Real-time messaging features (chat, presence) using Spring WebSocket support with STOMP or a broker-backed messaging layer.

Goal: Implement multi-user chat with rooms, presence notifications, and message persistence.

Functional Requirements:

- WebSocket endpoints for real-time messaging
- Room management, message broadcasting, and private messages
- Presence (online/offline) and typing indicators
- Optional message persistence for history

Non-functional Requirements:

- Scale-out strategy (sticky sessions or external broker like RabbitMQ)
- Authentication/authorization for socket endpoints
- Resilience for dropped connections and reconnection
- Performance & load testing for concurrent sockets

Topics Covered:

- Spring WebSocket and STOMP messaging (broker options).
- Socket security and session management.
- Scaling real-time features.

Prerequisites: Projects 1–6, WebSockets basics.
How it builds on earlier projects: Adds real-time layer often required in social or collaboration apps.

Recommended official resources:

```
- Messaging with STOMP+WebSocket guide:
https://spring.io/guides/gs/messaging-stomp-websocket
- WebSocket docs:
https://docs.spring.io/spring-framework/reference/web/websocket/stomp.html
```

10. GraphQL Service — Flexible Querying

Add a GraphQL endpoint for flexible, client-driven queries using Spring for GraphQL.

Goal: Provide a GraphQL API for the product catalog and user profile, with efficient data fetching (DataLoader).

Functional Requirements:

- Expose GraphQL schema for products, users, and orders
- Implement resolvers and DataLoader for N+1 problem
- Support subscriptions (real-time) for updates if needed
- Versioning and schema evolution practices

Non-functional Requirements:

- Schema-first design and good documentation
- Query complexity limits and depth limiting to prevent abuse
- Observability of GraphQL execution times and errors

Topics Covered:

- Spring for GraphQL and schema-first design.
- Efficient data fetching patterns (DataLoader).
- GraphQL subscriptions for live updates.

Prerequisites: Projects 1–6, understanding of REST vs GraphQL differences.
How it builds on earlier projects: Alternative API layer to support flexible clients (mobile).

Recommended official resources:

```
- Building a GraphQL service (guide): https://spring.io/guides/gs/graphql-server
- Spring for GraphQL docs: https://spring.io/projects/spring-graphql
```

11. Search & Analytics — Elasticsearch Integration

Enable full-text search, facets, and analytics over products and content using Elasticsearch via Spring Data Elasticsearch.

Goal: Index data, implement search endpoints, and provide analytics queries for admin dashboards.

Functional Requirements:

- Index product and content documents to Elasticsearch
- Implement text search, filters, and aggregations
- Near-realtime indexing (async sync strategy)
- Admin analytics for popular queries and product metrics

Non-functional Requirements:

- Search performance and shard/index tuning considerations
- Index lifecycle and mapping/versioning strategy
- Secure access to Elasticsearch and credentials management

Topics Covered:

- Spring Data Elasticsearch repository and operations.
- Search indexing strategies and analyzers.
- Connecting application models with search models.

Prerequisites: Projects 3–5; knowledge of data modeling.
How it builds on earlier projects: Adds search capabilities to Product/Content services.

Recommended official resources:

```
- Spring Data Elasticsearch: https://spring.io/projects/spring-data-elasticsearch
- Elasticsearch operations docs:
https://docs.spring.io/spring-data/elasticsearch/reference/index.html
```

12. Observability & Production Readiness

Make your services production-ready: Actuator, metrics, tracing, centralized logging, and dashboards (Prometheus & Grafana).

Goal: Expose health, metrics, and traces; integrate with Prometheus/Grafana and set up alerting basics.

Functional Requirements:

- Expose Actuator endpoints for health and metrics
- Instrument custom metrics and business metrics
- Connect app metrics to Prometheus and visualize in Grafana
- Add tracing for distributed requests and logs correlation

Non-functional Requirements:

- Low-overhead monitoring and sampling strategy
- Secure actuator endpoints (auth & IP restrictions)
- Retention policy for metrics and logs
- On-call runbook / basic SLOs for critical services

Topics Covered:

- Spring Boot Actuator and production-ready features.
- Micrometer metrics and Prometheus/Grafana integration.
- Distributed tracing and log correlation.

Prerequisites: Any of Projects 1–11; familiarity with containerization.
How it builds on earlier projects: Adds ops/observability to all services.

Recommended official resources:

```
- Actuator docs: https://docs.spring.io/spring-boot/reference/actuator/index.html
- Observability with Spring Boot (blog):
https://spring.io/blog/2022/10/12/observability-with-spring-boot-3
- Micrometer Prometheus docs: https://docs.micrometer.io/
```

13. Cloud-Native Deployment — Docker, Kubernetes & CI/CD

Containerize services, build images and deploy using Kubernetes. Add CI/CD pipelines and practice canary/rolling deployments.

Goal: Prepare production deployment artifacts and pipelines, including readiness/liveness probes and secrets management.

Functional Requirements:

- Docker images for each service and local docker-compose for dev
- Helm charts or Kubernetes manifests for deployments/services
- CI pipeline for build/test/docker-publish (GitHub Actions/GitLab CI)
- Secrets/config via Kubernetes Secrets or external store

Non-functional Requirements:

- Zero-downtime deploy strategies (rolling, canary)
- Resource requests/limits and autoscaling
- Cluster-level monitoring & logging (ELK/EFK or Grafana stack)
- Cost-conscious architecture (right-sizing)

Topics Covered:

- Docker and Cloud Native Buildpacks for Spring Boot.
- Kubernetes basics: deployments, services, probes, config maps, secrets.
- CI/CD pipelines with GitHub Actions/GitLab CI.

Prerequisites: Projects 5–12; basic infra knowledge.
How it builds on earlier projects: Prepare microservices for production hosting and scale.

Recommended official resources:

```
- Spring Boot and Cloud Native Buildpacks:
https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#buildpack
- Kubernetes docs: https://kubernetes.io/docs/home/
```

14. AI-Enabled Assistant — Spring AI & LLM integration

Add AI-powered features to your app: summarization, recommendations, embeddings search, or a chat assistant — built using Spring AI.

Goal: Integrate LLM-backed features in a modular, testable way while handling prompts, safety, caching, and costs.

Functional Requirements:

- Endpoint that summarizes content using an LLM-backed service
- Search-augmentation using embeddings (local or cloud)
- Conversational assistant endpoint with context and safety filters
- Caching and rate-limiting for LLM calls; usage tracking

Non-functional Requirements:

- Cost, latency, and token usage awareness (budgeting)
- Rate limiting and usage quotas for LLM APIs
- Safety checks on LLM output (filters, fallback content)
- Logging and audit of prompts and responses (privacy considerations)

Topics Covered:

- Spring AI abstractions and clients for integrating LLMs and embeddings.
- Prompt-engineering patterns, caching, and streaming outputs.
- Privacy, safety, and cost-control when using LLMs.

Prerequisites: Projects 1–13; understanding of external APIs and async flows.
How it builds on earlier projects: Adds intelligence to product and search features (projects 3 & 11).

Recommended official resources:

```
- Spring AI project page: https://spring.io/projects/spring-ai
- Spring AI reference docs: https://docs.spring.io/spring-ai/reference/index.html
- Spring AI GitHub: https://github.com/spring-projects/spring-ai
```

15. CQRS & Event Sourcing — Scalable Read/Write Separation

Introduce CQRS and Event Sourcing for complex domains requiring auditability, high scalability, and eventual consistency.

Goal: Implement separate read and write models, persist events, and create projections for queries.

Functional Requirements:

- Command model for writes (immutable events emitted)
- Event store for persisting events (append-only)
- Projections/denormalized read models for queries
- Sagas for long-running transactions across services

Non-functional Requirements:

- Strong versioning and migration strategies for events
- Snapshotting for performance
- Event replay and recovery procedures
- Monitoring of event store and projection pipelines

Topics Covered:

- CQRS patterns, event sourcing best practices.
- Axon Framework (optional) or custom event-store implementations.
- Sagas, eventual consistency, and projection maintenance.

Prerequisites: Advanced: Projects 5, 8, 11; experience with distributed systems.
How it builds on earlier projects: Provides an architecture for high-scale domains requiring audit and complex business flows.

Recommended official resources:

```
- Axon Framework docs (example): https://axoniq.io/
- Event-driven and CQRS resources:
https://learn.microsoft.com/azure/architecture/patterns/cqrs
```

Learning Strategy & Checklist

Study approach — For each project:
1) Read the official guides and docs listed. 2) Design the API and data model. 3) Implement incremental features and tests. 4) Add CI, containerization, and monitoring. 5) Push to GitHub with clear README, diagrams and sample data.
Iterate: after shipping a minimal version, rework for non-functional requirements (resilience, caching, observability).

Selected official references (core)

```
- Spring Boot reference:
https://docs.spring.io/spring-boot/docs/current/reference/html/
- Spring Framework reference:
https://docs.spring.io/spring-framework/reference/index.html
- Spring Security reference:
https://docs.spring.io/spring-security/reference/index.html
- Spring Cloud documentation:
https://docs.spring.io/spring-cloud/docs/current/reference/html/
- Spring AI project page: https://spring.io/projects/spring-ai
- Spring Guides: https://spring.io/guides
```