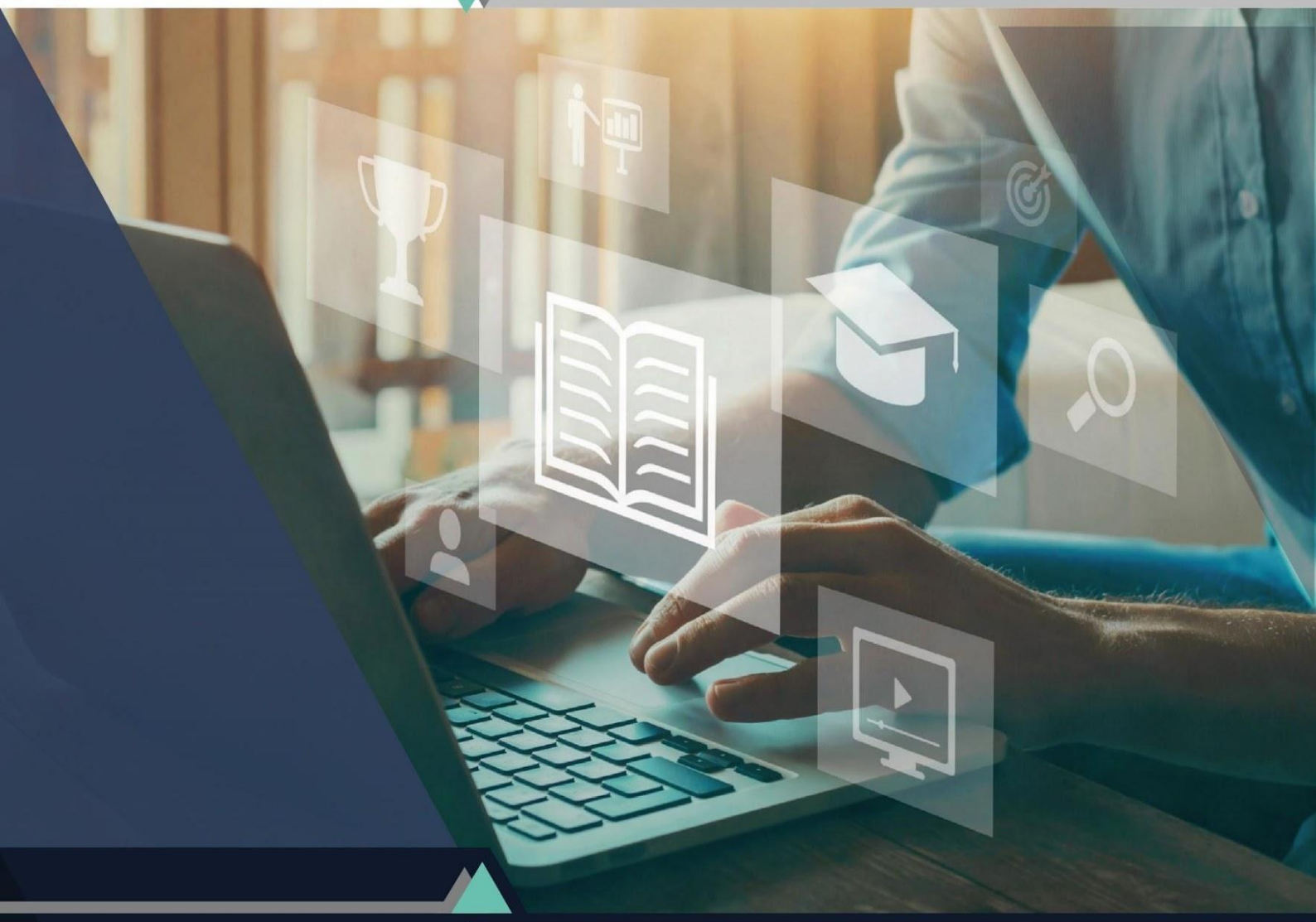

Desenvolvimento de Aplicações Back-End com Spring Boot e Quarkus

Autoria: Elder Moraes e Francisco Isidro

Introdução

Este eBook foi elaborado para oferecer uma compreensão aprofundada sobre dois dos principais frameworks utilizados no desenvolvimento de aplicações Java modernas: Quarkus e Spring Boot. Estruturado em dois módulos, ele visa atender às necessidades de profissionais que desejam se especializar em tecnologias cloud-native, microsserviços e arquitetura moderna de software.

O primeiro módulo é dedicado ao Quarkus, um framework que tem se destacado por sua eficiência, inicialização rápida e baixa pegada de memória. Ao longo dos capítulos, exploramos desde os fundamentos da plataforma até aspectos avançados como tolerância a falhas, persistência com Panache, segurança com JWT e observação com OpenTelemetry. O objetivo é proporcionar uma base sólida para o desenvolvimento de microsserviços resilientes e performáticos com Quarkus.

O segundo módulo foca no Spring Boot, consolidado como um dos frameworks mais utilizados no ecossistema Java para construção de APIs REST, integração com bancos de dados relacionais, segurança, testes e muito mais. Cada capítulo cobre uma etapa essencial do ciclo de vida de desenvolvimento, com ênfase em boas práticas, arquitetura em camadas, tratamento de exceções, validação de dados, uso de JWT para autenticação e integração com serviços externos.

Este material foi cuidadosamente organizado para manter a linguagem didática e fiel ao conteúdo ministrado nas aulas, sem perder o rigor técnico. A ausência de blocos de código é intencional, buscando valorizar a compreensão conceitual e a aplicação contextual dos recursos oferecidos por cada framework.

A leitura deste eBook oferece um panorama completo sobre o uso profissional de Quarkus e Spring Boot, capacitando o leitor para construir aplicações robustas, escaláveis e alinhadas às exigências do mercado atual de tecnologia.

Bons estudos!

MÓDULO DE QUARKUS:

Capítulo 1: Quarkus – Introdução ao Quarkus, Cloud Native e Kubernetes Native

O desenvolvimento moderno de aplicações back-end exige plataformas que combinem leveza, performance e compatibilidade com arquiteturas distribuídas e conteinerizadas. Neste cenário, o Quarkus surge como uma proposta disruptiva no ecossistema Java, alinhando-se aos conceitos de Cloud Native e Kubernetes Native. Este capítulo apresenta a história que levou ao surgimento do Quarkus, destacando os desafios superados pelo Java ao longo dos anos e como o Quarkus responde a essas necessidades com uma abordagem inovadora.

A jornada começa em 1996, com o lançamento do Java, uma linguagem orientada a objetos que se destacou pela promessa do "write once, run anywhere". Essa capacidade de portabilidade, aliada à simplicidade em relação a linguagens como C++ e Smalltalk, rápido posicionou o Java como uma das linguagens mais populares do mundo. Eventos como o JavaOne, que chegou a reunir 15 mil pessoas, são reflexos desse sucesso.

Porém, em 2013, um novo marco tecnológico mudaria os rumos do desenvolvimento de software: o lançamento do Docker. Embora a tecnologia de containers já existisse em nível de sistema operacional via cgroups no Linux, o Docker foi responsável por padronizar sua utilização, tornando containers acessíveis e reutilizáveis. Isso revolucionou o mercado, pois estabeleceu um formato comum para a distribuição de aplicações, promovendo escalabilidade, portabilidade e eficiência.

Com o surgimento do Docker, o desejo de rodar aplicações Java dentro de containers aumentou. Contudo, a JVM tradicional não estava preparada para esse ambiente. O principal problema era que, ao rodar dentro de um container, a JVM interpretava os recursos do host, não os limites impostos ao container. Isso levava a alocação excessiva de memória e falhas de execução, exigindo soluções alternativas, como scripts para informar manualmente os limites de recursos à JVM.

Somente a partir da build 1.2.1 do Java 8 é que surgem as primeiras funcionalidades para lidar com containers, e é com o Java 11, lançado em 2018, que o suporte se torna robusto. Paralelamente, em 2016, surge o Eclipse MicroProfile, um conjunto de especificações voltadas à construção de microsserviços em Java. Com APIs como CDI, JAX-RS, JSON-P e JSON-B, o MicroProfile visa romper com a dependência de servidores de aplicação tradicionais, como JBoss, WebLogic ou TomEE, promovendo aplicações mais leves e otimizadas para ambientes distribídos.

Mesmo com esse avanço, o Java ainda enfrentava desafios relacionados à leveza e ao tempo de inicialização, importantes em ambientes de containers e escalabilidade dinâmica. Foi nesse contexto que, em 2019, durante o evento DevNexus em Atlanta, ocorreu a primeira apresentação pública do Quarkus. Desenvolvido pela Red Hat, o Quarkus propunha um novo paradigma com o lema "Supersonic Subatomic Java": aplicações Java com performance extrema e baixo consumo de recursos.

Segundo sua própria documentação, o Quarkus é uma stack Java projetada para o modelo Kubernetes Native, compatível com OpenJDK e GraalVM, e baseada em bibliotecas e padrões amplamente adotados no universo Java. Ele se

diferencia por já incluir, de forma integrada, recursos para desenvolvimento de aplicações REST, Event Driven, Web Apps, CLIs e muito mais, oferecendo suporte nativo a containers e integração com tecnologias como Kafka, Camel, Hibernate, Kubernetes, OpenShift e provedores de nuvem como AWS, Google Cloud e Azure.

Uma das maiores inovações do Quarkus é a movimentação de várias tarefas do tempo de execução para o tempo de build. Isso inclui carregamento de classes, inicialização de dependências e configurações, reduzindo drasticamente o tempo de startup das aplicações. Esse modelo também permite imagens de container menores e maior eficiência no uso de memória e CPU.

Benchmarks divulgados pelo próprio projeto mostram diferenças impressionantes. Uma aplicação REST simples ocupa cerca de 136 MB em stacks tradicionais. Com Quarkus rodando em JVM, o consumo cai para 73 MB, e com compilação nativa usando GraalVM, para apenas 12 MB. O tempo de inicialização e primeira resposta também é significativamente menor: de 4,3 segundos em stacks tradicionais para 0,943 s com JVM e 0,016 s com nativo.

Esse ganho de performance está alinhado às exigências do paradigma Cloud Native, conforme definido pela Cloud Native Computing Foundation (CNCF). Segundo a CNCF, aplicações cloud native devem ser escaláveis, resilientes, gerenciáveis e observáveis. Devem operar eficientemente em ambientes modernos, dinâmicos, como nuvens públicas, privadas ou híbridas, utilizando containers, service mesh, microserviços, infraestrutura imutável e APIs declarativas. O Quarkus incorpora todos esses elementos, sendo projetado com foco em containers e Kubernetes desde sua concepção.

Outro destaque é a experiência do desenvolvedor. Com funcionalidades como Live Reload instantâneo, Dev Services para provisão automática de serviços como banco de dados e e-mail, e compilação nativa pronta para uso, o Quarkus torna o ciclo de desenvolvimento mais rápido e eficiente. Sua arquitetura favorece também a observabilidade, integração com ferramentas de tracing, health checks, configuração por ConfigMaps e Secrets, facilitando operação em ambientes de produção em larga escala.

Com isso, o Quarkus consolida-se como uma plataforma completa, não apenas um framework, atendendo plenamente as necessidades de aplicações modernas no ecossistema Java, especialmente em contextos onde performance, leveza e integração com Kubernetes são essenciais.

Capítulo 2: Quarkus – Configuração de Projetos e Extensões

Este capítulo apresenta uma abordagem prática e detalhada sobre como iniciar um projeto com o Quarkus, destacando as ferramentas disponíveis, o processo de criação de um projeto, o conceito de extensões, bem como funcionalidades que otimizam a experiência do desenvolvedor e facilitam o desenvolvimento de aplicações modernas.

Para iniciar um projeto com Quarkus, a principal ferramenta é o site code.quarkus.io, onde é possível gerar uma aplicação personalizada. A seleção pode ser feita entre Maven e Gradle, sendo o Maven frequentemente utilizado, inclusive com suporte a Kotlin. É possível escolher a versão do Quarkus, como as LTS 3.15 ou 3.20, e selecionar extensões que irão compor o projeto. Essas extensões são mais que dependências; elas adicionam código, configuram automaticamente partes da aplicação e integram funcionalidades.

As extensões são organizadas em categorias como REST, Event Driven, Command Line Tools, Web Apps, AI, Reactive, Observabilidade, Segurança, Serialização, entre outras. Um destaque importante é a compatibilidade com APIs do Spring. O Quarkus implementa diversas APIs do Spring (como Spring DI, Web, Security, Data, Boot Properties), permitindo que projetos Spring sejam portados para Quarkus com maior facilidade, aproveitando os ganhos de performance e eficiência.

Uma vez que o projeto é criado e baixado, ele inclui uma estrutura com Dockerfiles prontos para diferentes perfis de deploy (como JVM, nativo ou imagens reduzidas), um arquivo application.properties para configurações e,

caso tenha sido selecionada a extensão REST, uma classe com um endpoint exemplo já implementado. O Quarkus também traz por padrão extensões como ARC (para CDI), JUnit 5 e REST Assured para testes.

A execução em modo de desenvolvimento é feita com o comando mvn quarkus:dev, ativando o Developer Mode. Nesse modo, a aplicação inicia rapidamente (geralmente em menos de um segundo), e qualquer alteração no código é refletida instantaneamente através de Live Reload. O terminal interativo permite comandos para reiniciar a aplicação, editar configurações, visualizar exceções, entre outros.

Uma das ferramentas mais poderosas nesse contexto é o Dev UI, acessível via browser. Essa interface fornece uma visão completa do projeto em desenvolvimento: extensões ativas, bins CDI, endpoints REST, documentação via Swagger (OpenAPI), configurações disponíveis, workspace, testes, Dev Services, métricas, dependências e muito mais. Tudo centralizado em um único painel que aumenta consideravelmente a produtividade.

O conceito de Dev Services é outro diferencial. Ao detectar a necessidade de integração com um serviço externo (como PostgreSQL, Kafka ou servidor de e-mail) sem que haja configuração prévia no application.properties, o Quarkus automaticamente instancia uma imagem Docker correspondente, realiza a conexão com a aplicação e, se houver scripts de inicialização, também os executa. Isso elimina a necessidade de configurar manualmente ambientes de desenvolvimento complexos.

A execução em ambiente de produção é feita após a geração do pacote com mvn clean package, produzindo um JAR tradicional. Para compilação nativa,

usa-se mvn clean package -Pnative, utilizando o GraalVM. Essa geração é altamente otimizada, pois o bytecode produzido pelo Quarkus é projetado para maximizar a performance da compilação nativa, resultando em arquivos executáveis extremamente leves e rápidos.

Com a aplicação compilada nativamente, ela pode ser executada em qualquer sistema com a mesma arquitetura, sem necessidade de JVM instalada. Isso facilita o deploy em ambientes controlados e reduz significativamente o tempo de startup (menos de 30 ms em muitos casos). O Quarkus garante que tudo que está dentro de sua stack é compatível com compilação nativa, evitando os problemas históricos de incompatibilidades em frameworks Java.

Em resumo, o Quarkus oferece uma experiência de desenvolvimento moderna, eficiente e integrada, com ferramentas que simplificam desde a configuração inicial até a compilação e deploy. O suporte a extensões, o Dev UI, os Dev Services e a compatibilidade com Spring tornam o Quarkus uma escolha poderosa para aplicações cloud native de alta performance.

Capítulo 3: Quarkus – GraalVM

A evolução da plataforma Java nos últimos anos tem sido marcada por iniciativas que buscam maximizar a performance e a flexibilidade da linguagem. Nesse contexto, a GraalVM surge como um dos pilares fundamentais, oferecendo uma máquina virtual poliglota com capacidades avançadas de interoperabilidade, tooling unificado e compilação de alta performance. Este capítulo detalha os fundamentos, os diferenciais e o papel estratégico da GraalVM dentro do ecossistema Java moderno, especialmente em conjunto com o Quarkus.

A motivação principal para a criação da GraalVM pode ser compreendida em três pilares: performance, interoperabilidade entre linguagens e compartilhamento de ferramentas. No quesito performance, a proposta é oferecer alto desempenho não apenas para Java, mas para diversas linguagens. A GraalVM implementa um compilador Just-in-Time (JIT) chamado Graal Compiler, que substitui o compilador padrão da JVM (escrito em C++) por uma implementação em Java. Esse compilador também é utilizado em modo Ahead-of-Time (AOT) para compilação nativa, trazendo uma unificação entre os modos de execução e ganhos expressivos em eficiência.

Benchmarks realizados pelo time da GraalVM demonstram que linguagens como Ruby, quando executadas na GraalVM, podem operar com performance equiparável às linguagens mais rápidas do mercado, como C ou Java. Além disso, o uso combinado de Ruby e JavaScript em interoperabilidade mostra que é possível manter milhões de operações por segundo mesmo atravessando linguagens distintas. Isso é possível porque a GraalVM elimina a necessidade

de camadas intermediárias e de serialização ao compartilhar diretamente a implementação entre as linguagens.

O segundo pilar, interoperabilidade, é viabilizado por um componente essencial da GraalVM chamado Truffle. Truffle é um framework para implementação de linguagens na GraalVM. Ele permite criar interpretadores de linguagens que operam diretamente sobre a JVM com acesso ao Graal Compiler. Com isso, linguagens como Ruby, Python, R, JavaScript, e mesmo linguagens de baixo nível como C e C++ (via Sulong) podem executar dentro da GraalVM com suporte completo a otimizações de runtime.

Esse modelo torna possível executar um código em Java que invoque diretamente funções escritas em Ruby, por exemplo, sem necessidade de integração via API externa ou comunicação em rede. Mais ainda, o Truffle fornece uma API para criação de aplicações poliglotas, permitindo reutilização de bibliotecas escritas em outras linguagens dentro de um mesmo runtime.

O terceiro pilar é a unificação de tooling. No cenário tradicional, cada linguagem possui seu conjunto de ferramentas de profiling, debugging, monitoramento, etc. Com a GraalVM, ferramentas desenvolvidas sobre o Truffle tornam-se automaticamente compatíveis com todas as linguagens suportadas. Isso permite, por exemplo, que uma ferramenta escrita para monitorar aplicações Ruby seja utilizada também com aplicações Java, JavaScript ou Python, desde que executadas na GraalVM.

Essa arquitetura tem impactos significativos na manutenção e evolução de sistemas poliglotas. Ao padronizar a infraestrutura de execução, elimina-se a sobrecarga de manter toolings distintos para cada linguagem. Adicionalmente, a

capacidade de usar a melhor linguagem para cada tarefa específica aumenta a produtividade e qualidade do software.

Historicamente, o projeto GraalVM foi iniciado ainda na época da Sun Microsystems, no Sun Labs, e ganhou trânsito e maturidade dentro do Oracle Labs após a aquisição da Sun pela Oracle. A primeira release candidate foi lançada em 2018, mas o projeto já acumulava cerca de uma década de pesquisa e desenvolvimento.

Em termos de governança, embora a Oracle seja a mantenedora principal do projeto, a GraalVM conta com um Advisory Board composto por representantes de empresas como Bellsoft, Broadcom, Microdoc, Gluon, Shopify, Red Hat, Neo4j, Amazon, Alibaba, entre outras. Isso garante um modelo colaborativo e de maior transparência. A Red Hat, por exemplo, criou um fork do Native Image Builder chamado Mandrel, utilizado para garantir suporte e otimização do Quarkus com compilação nativa, com suas contribuições sendo incorporadas à versão community da GraalVM.

Portanto, a GraalVM não é apenas uma alternativa de execução para aplicações Java. Ela é um ambiente de runtime poderoso e versátil, que amplia as possibilidades de desenvolvimento com Java e outras linguagens. Em conjunto com o Quarkus, viabiliza a entrega de aplicações leves, de alto desempenho e altamente integradas, sendo uma das peças-chave no desenvolvimento de soluções cloud native modernas.

Capítulo 4: Quarkus – Compilação Nativa

A compilação nativa é um dos temas mais relevantes quando se discute a performance e a eficiência no uso do Quarkus em ambientes de produção. A técnica, baseada no conceito de Ahead-of-Time Compilation (AOT), permite transformar aplicações Java em executáveis nativos, dispensando a necessidade de uma JVM para execução e proporcionando ganhos significativos em tempo de startup e uso de memória.

A base dessa abordagem é o Native Image Builder, componente da GraalVM responsável por realizar uma análise estática de todas as classes e dependências do projeto, mapeando apenas o que é efetivamente utilizado em tempo de execução. A partir dessa análise, é gerado um executável nativo específico para o sistema operacional e arquitetura de hardware em questão.

O que compõe uma imagem nativa? As classes da aplicação, suas dependências diretas, bibliotecas utilizadas em runtime e um runtime embutido chamado Substrate VM. Este último é uma mini JVM que fornece funcionalidades essenciais como gerenciamento de memória, agendamento de threads e garbage collection, ainda que em versão simplificada.

Um dos principais atrativos da compilação nativa é a eliminação de vários componentes da execução tradicional via JVM. Isso inclui:

- Carregamento dinâmico de classes (class loading)
- Interpretação de bytecode
- Compilação Just-in-Time (JIT)
- Estruturas de profiling e otimização em tempo de execução

Com isso, as aplicações nativas alcançam tempos de inicialização extremamente reduzidos e um consumo de memória muito inferior ao tradicional, já que estruturas como metadata de classes, caches de código interpretado e dados de profiling simplesmente deixam de existir.

Contudo, nem tudo são vantagens. A compilação nativa traz limitações que devem ser cuidadosamente avaliadas:

- Incompatibilidade com a JVM Tool Interface (JVM TI)
- Ausência de suporte para Java Agents e JMX
- Dependência do garbage collector Serial GC, que pode impactar aplicações com heaps muito grandes
- Impossibilidade de otimizações dinâmicas durante a execução
- Limitação no uso de ferramentas como heap dump e thread dump (restritas à versão Enterprise da GraalVM)

Essas limitações fazem com que a compilação nativa não seja indicada para todos os cenários. Seu uso ideal se aplica a:

- Aplicações pequenas
- Códigos de execução curta e frequente
- Ferramentas de linha de comando (CLI)
- Funções serverless com curta duração

Nestes contextos, a ausência de um runtime pesado e a execução imediata são decisivas para eficiência e escalabilidade.

Vale destacar que o Quarkus é projetado desde sua concepção para aproveitar ao máximo a compilação nativa. Suas extensões são todas compatíveis com o

Native Image Builder, e o bytecode gerado pelo Quarkus já é otimizado para compilação AOT. Assim, usar o Quarkus com compilação nativa não exige ajustes adicionais na maioria dos casos, o que simplifica o processo para os desenvolvedores.

A Red Hat, em apoio a essa abordagem, criou o Mandrel, um fork do Native Image Builder que integra diretamente com o Quarkus e tem foco exclusivo na compilação nativa. As contribuições feitas via Mandrel retornam para o projeto GraalVM, fortalecendo o ecossistema como um todo.

Em suma, a compilação nativa é uma poderosa alternativa de execução para aplicações Java modernas, especialmente no contexto de containers e funções efêmeras. O Quarkus, com seu suporte completo a essa funcionalidade, consolida-se como uma escolha robusta e eficiente para o desenvolvimento de soluções cloud native.

Capítulo 5: Quarkus – Desenvolvimento de API RESTful com Quarkus

Neste capítulo, abordamos em profundidade o desenvolvimento de APIs RESTful utilizando o Quarkus. APIs REST (Representational State Transfer) são um dos pilares do desenvolvimento moderno de aplicações web, sendo o ponto de entrada para integração com aplicações externas, dispositivos móveis e outros sistemas distribuídos.

A implementação de APIs REST em Quarkus é feita por meio da extensão Quarkus REST, uma implementação da especificação Jakarta REST (antiga JAX-RS). Essa extensão fornece suporte tanto para endpoints imperativos (bloqueantes) quanto para reativos (non-blocking), sendo baseada no runtime Vert.x. Isso permite alta performance, mesmo em ambientes com alta concorrência e baixa latência.

Um endpoint é definido por uma URL que expõe um recurso ou funcionalidade da aplicação. Os métodos HTTP são utilizados para manipulação desses recursos: GET para leitura, POST para criação, PUT para atualização e DELETE para remoção. Esses métodos correspondem diretamente às operações CRUD.

O Quarkus REST permite anotar classes e métodos com @Path, @GET, @POST, @PUT, @DELETE, entre outras anotações, para mapear URLs e comportamentos. Adicionalmente, é possível especificar os formatos de entrada e saída com @Consumes e @Produces, permitindo controle preciso sobre os tipos MIME utilizados (por exemplo, application/json ou text/plain).

No contexto da evolução da plataforma Java, é importante entender a transição histórica do Java EE para Jakarta EE. O Java EE, rebatizado de Java 2 EE em 2006 e depois transferido para a Eclipse Foundation, passou a se chamar

Jakarta EE. Essa mudança também exigiu a substituição dos pacotes javax por jakarta nas APIs.

Dentro desse ecossistema, o Quarkus REST se integra com o MicroProfile, projeto criado em 2016 para padronizar o desenvolvimento de microsserviços em Java. APIs como CDI, JSON-P, JSON-B e JAX-RS compõem o núcleo do MicroProfile, e muitas dessas APIs são utilizadas diretamente em Quarkus.

Um diferencial importante do Quarkus REST é o suporte simultâneo a blocking e non-blocking handlers. Inicialmente, o Quarkus possuía duas extensões distintas para isso (REST e REST Reactive), mas ao longo do tempo, consolidou a REST Reactive como a implementação padrão, renomeando a antiga como REST Classic. Isso garante flexibilidade sem comprometer a simplicidade.

Para desenvolvimento, é possível configurar um caminho raiz global para os endpoints utilizando uma classe que estende javax.ws.rs.core.Application, anotada com @ApplicationPath. Dessa forma, todos os endpoints definidos na aplicação ficam agrupados sob um mesmo prefixo, como /api, facilitando a organização e o roteamento.

Outra funcionalidade essencial é o REST Client, utilizado para consumir APIs externas. Com o uso da anotação @RegisterRestClient, é possível definir interfaces que representam serviços remotos. Esses clientes são configurados com @Path e @GET, @POST etc., exatamente como nos recursos locais, e podem ser injetados com CDI para facilitar o consumo. O Quarkus integra esse modelo com MicroProfile Rest Client, garantindo compatibilidade e padronização.

A experiência do desenvolvedor é enriquecida pela presença do Swagger UI (via OpenAPI), que fornece uma documentação interativa das APIs em tempo de desenvolvimento. Isso permite testar e validar endpoints diretamente via navegador, otimizando o fluxo de desenvolvimento.

Em resumo, o desenvolvimento de APIs RESTful com Quarkus alia robustez, performance e praticidade. Ao integrar especificações consolidadas do Jakarta EE e do MicroProfile com ferramentas modernas e suporte a modelos reativos, o Quarkus se consolida como uma plataforma de escolha para soluções Java modernas e escaláveis.

Capítulo 6: Quarkus – Uso da API Fault Tolerance

Neste capítulo, exploramos a implementação da tolerância a falhas em aplicações Java com Quarkus, utilizando a especificação MicroProfile Fault

Tolerance. Esse recurso é essencial para sistemas distribuídos, onde a comunicação entre serviços pode estar sujeita a falhas de rede, lentidão ou indisponibilidade. O Quarkus integra esse suporte por meio da extensão SmallRye Fault Tolerance, baseada no MicroProfile.

A tolerância a falhas é aplicada especialmente em pontos de comunicação remota, onde se concentram os principais riscos operacionais. Três padrões fundamentais são abordados: Timeout, Fallback e Circuit Breaker.

O *Timeout* é o mecanismo que define o tempo máximo de espera por uma resposta. Caso esse limite seja ultrapassado, a execução é interrompida, evitando que o sistema fique bloqueado indefinidamente. Por padrão, o valor é de 1 segundo (1000 ms), mas pode ser ajustado. A unidade de tempo também é configurável. O uso adequado de timeouts impede a propagação de lentidão por todo o sistema.

O *Fallback* é uma estratégia que define um comportamento alternativo quando ocorre uma falha. Em vez de propagar a exceção ao usuário, um método de backup é chamado. Esse método deve ter a mesma assinatura do método original. Pode retornar um valor padrão, um objeto vazio ou qualquer lógica de contingência definida conforme a regra de negócio. Isso melhora a resiliência da aplicação.

O *Circuit Breaker* é inspirado no disjuntor elétrico. Ele monitora as chamadas para um serviço e, ao detectar uma taxa de falhas acima de um limiar, "abre o circuito" e impede novas chamadas por um tempo. Após esse período, novas chamadas são testadas. Se ocorrerem com sucesso, o circuito é "fechado" novamente. As configurações principais incluem:

- requestVolumeThreshold: número de requisições para considerar a análise.
- failureRatio: proporção de falhas para acionar o circuito.
- delay: tempo que o circuito permanece aberto.
- successThreshold: quantidade de sucessos para reativar o circuito.

Esses mecanismos permitem uma degradação controlada (graceful degradation), reduzindo o impacto de falhas localizadas.

Outro aspecto essencial é o *Health Check*, também padronizado pelo MicroProfile. Utilizado em ambientes Kubernetes, permite que o orquestrador monitore o estado da aplicação. Dois probes principais são utilizados:

- *Liveness*: indica se a aplicação está viva. Se falhar, o Kubernetes reinicia o pod.
- *Readiness*: indica se a aplicação está pronta para receber requisições. Se falhar, o pod continua existindo, mas não recebe tráfego.

A implementação consiste em criar classes que implementam HealthCheck, anotadas com @Liveness ou @Readiness. A métrica de readiness pode incluir verificações como acesso a banco de dados ou serviços externos. Um fallback definido pode ser reutilizado para verificar se um serviço remoto está responsivo.

Com os health checks corretamente configurados, o Kubernetes pode gerenciar melhor a disponibilidade da aplicação, direcionando tráfego apenas para instâncias saudáveis. Além disso, melhora a experiência do usuário final e a confiabilidade do sistema como um todo.

A API de Fault Tolerance, combinada com Circuit Breaker, Timeout, Fallback e Health Checks, representa uma abordagem robusta para construir aplicações resilientes e preparadas para operação em ambientes distribuídos e conteinerizados. O Quarkus, ao integrar essas funcionalidades de forma nativa e com suporte da especificação MicroProfile, reafirma seu papel como uma plataforma ideal para desenvolvimento cloud native em Java.

Capítulo 7: Quarkus – Persistência de Dados com Panache

Neste capítulo, abordamos o uso do Panache no Quarkus, uma extensão desenvolvida para simplificar a manipulação de dados utilizando o Hibernate ORM. O objetivo principal do Panache é oferecer uma abordagem pragmática

para operações de persistência, especialmente em cenários comuns de CRUD, sem abrir mão da flexibilidade do Hibernate.

O Panache surge como uma camada de abstração sobre o Hibernate ORM, adotando convenções e padrões que facilitam o desenvolvimento, especialmente para casos de uso simples. Enquanto o Hibernate foi projetado para lidar com cenários altamente complexos, o Panache visa atender aos 80% das necessidades mais frequentes de forma objetiva e com menos código boilerplate.

O ponto de partida para usar o Panache é adicionar as extensões adequadas ao projeto: hibernate-orm-panache e um driver de banco de dados como o H2, ideal para testes e demonstrações por ser um banco em memória com configuração simples. As propriedades necessárias, como quarkus.datasource.db-kind, quarkus.datasource.jdbc.url e quarkus.hibernate-orm.database.generation, devem ser configuradas no arquivo application.properties.

Para definir uma entidade, basta criar uma classe que estenda PanacheEntity e anotá-la com @Entity. Essa superclasse provê um atributo id e métodos como persist(), findById() e listAll() prontos para uso. Um ponto interessante é que, por padrão, os atributos podem ser public, o que, embora contrarie princípios clássicos de encapsulamento, é uma escolha deliberada para reduzir complexidade em aplicações simples. O Panache ainda oferece suporte caso o desenvolvedor prefira usar atributos privados com getters e setters.

A criação de uma API REST para interagir com essa entidade é direta. Com a extensão quarkus-resteasy-reactive, pode-se criar endpoints com métodos anotados com @GET, @POST, @PUT e @DELETE. A serialização e

desserialização de objetos JSON é tratada pela extensão quarkus-resteasy-reactive-jsonb, que deve ser adicionada ao projeto para evitar erros como "unsupported media type".

Para operações de leitura, listAll() retorna todos os registros, enquanto findById() recupera uma entidade específica. Para inserção de dados, persist() é utilizado, sendo importante garantir que o campo id esteja nulo antes da chamada, evitando sobrescrita acidental. A anotação @Transactional deve ser aplicada a métodos que realizam operações de escrita.

Atualizações são feitas ao recuperar a entidade existente, modificar seus campos e chamar novamente persist(). Para exclusão, o método deleteById() permite remover uma entidade diretamente por seu identificador. Também é possível criar um arquivo import.sql com dados de teste, facilitando a validação de funcionalidades sem necessidade de inserção manual.

O Panache ainda permite a criação de *custom finders*, métodos específicos para consultas com filtros personalizados. Por exemplo, um método estático findByAnoNascimento(int ano) pode ser definido utilizando o método find("anoNascimento", ano).list(), retornando uma lista de entidades conforme o critério informado. Esses métodos podem ser expostos como novos endpoints na API REST, permitindo queries mais ricas sem necessidade de escrever consultas SQL explícitas.

Em resumo, o Panache é uma poderosa extensão que torna o uso do Hibernate mais acessível e produtivo, sem sacrificar os recursos avançados que os desenvolvedores Java esperam. Ele é ideal para aplicações que necessitam de persistência simples e rápida, e se integra perfeitamente com o restante do

ecossistema Quarkus, promovendo agilidade no desenvolvimento de soluções cloud native.

Capítulo 8: Quarkus – Autenticação e Segurança com JWT e RBAC

Neste capítulo, exploramos os fundamentos da segurança em APIs no contexto do Quarkus, com foco em duas tecnologias essenciais: JSON Web Token (JWT) e Role-Based Access Control (RBAC). Ainda que não se trate de um curso de

segurança completo, os conceitos aqui apresentados são fundamentais para garantir a integridade e a confidencialidade de aplicações modernas.

O JWT é um padrão aberto amplamente adotado para autenticação e troca de informações entre sistemas. Um token JWT é composto por três partes: header, payload e assinatura. O header contém os metadados do token, como o algoritmo de criptografia. O payload contém as informações do usuário, chamadas de claims, que podem ser registradas (como issuer, subject, expiration), públicas (definidas pela aplicação) ou privadas (para troca exclusiva entre cliente e servidor). A assinatura garante a integridade do token, impedindo alterações indevidas.

O JWT é stateless, ou seja, não requer armazenamento no servidor. O processo de autenticação ocorre da seguinte forma: o cliente envia suas credenciais, a API valida e retorna um token, que será enviado em cada requisição subsequente no cabeçalho Authorization: Bearer. A API valida o token, verifica as claims e, se tudo estiver correto, autoriza o acesso.

No Quarkus, é necessário adicionar a extensão smallrye-jwt para utilizar JWT. Também é preciso configurar propriedades como o caminho para a chave pública usada na validação e o issuer permitido. Essa chave pública pode ser hospedada no GitHub ou outro repositório confiável. Com essas configurações, a aplicação passa a validar automaticamente os tokens recebidos.

Para acessar os dados do token dentro da aplicação, utilizamos a API JWTClaims, que permite recuperar claims específicas como preferred_username. Para que isso funcione corretamente, a classe que

consome o JWT deve estar anotada com @RequestScoped, garantindo que os dados sejam tratados no escopo de cada requisição.

A autenticação por si só não é suficiente; é preciso também implementar controle de acesso. Para isso, utilizamos o RBAC, que define permissões com base em roles (funções/perfis). Cada token pode conter uma ou mais roles, e o acesso aos recursos é condicionado à presença dessas roles. No Quarkus, isso é feito com a anotação @RolesAllowed, aplicada diretamente aos métodos REST. Essa anotação define quais roles têm permissão para executar determinado endpoint.

O processo funciona assim: o token JWT contém as roles do usuário; ao receber uma requisição, a aplicação verifica se o token é válido e se as roles do usuário são compatíveis com as exigidas pelo método. Caso positivo, o acesso é concedido; do contrário, o usuário recebe uma resposta de erro (401 Unauthorized ou 403 Forbidden).

No exemplo prático, uma classe SecureResource foi criada para demonstrar esse fluxo. Um método getClaim() retorna o preferred_username presente no token. Quando protegido com @RolesAllowed("Subscriber"), apenas usuários com a role Subscriber conseguem acessar esse método. Mudando a role exigida para uma diferente daquela presente no token, a aplicação retorna 403 Forbidden, indicando que o usuário não tem permissão suficiente.

Para testes, um token JWT pode ser gerado localmente com uma chave privada, e validado com a chave pública configurada na aplicação. O token pode ser passado nas chamadas via curl, simulando o comportamento real de um cliente autenticado.

Em resumo, a combinação de JWT e RBAC no Quarkus permite implementar autenticação e autorização de forma segura, flexível e eficiente. Essas tecnologias são fundamentais para proteger APIs e garantir que apenas usuários autorizados acessem os recursos apropriados, reforçando a postura de segurança de aplicações cloud native.

Capítulo 9: Quarkus – Observabilidade com OpenTelemetry

Neste capítulo, abordamos a implementação de observabilidade em aplicações Java utilizando o Quarkus, com foco nas tecnologias OpenTelemetry, Jaeger e Micrometer. A observabilidade é um pilar essencial em sistemas modernos, especialmente em arquiteturas de microsserviços, pois permite acompanhar o

comportamento da aplicação em execução, identificar gargalos e diagnosticar falhas com eficiência.

O OpenTelemetry é o principal projeto open source da CNCF (Cloud Native Computing Foundation) voltado para observabilidade. Ele fornece um conjunto de ferramentas e APIs para coletar, processar e exportar dados de telemetria, incluindo logs, métricas e traces. O Quarkus, como plataforma moderna cloud native, integra-se nativamente ao OpenTelemetry, permitindo instrumentação de forma simples e eficaz.

Para habilitar o suporte ao OpenTelemetry no Quarkus, adiciona-se a extensão correspondente ao projeto via comando mvn quarkus:add-extension. A partir disso, é possível coletar traçamentos distribuídos (distributed tracing), fundamentais em aplicações com fluxos complexos entre vários serviços.

Na demonstração prática, utilizou-se o Jaeger como ferramenta para visualização dos traces. O Jaeger foi iniciado localmente via container (podman ou docker), com configurações apropriadas de portas e variáveis de ambiente. Ao realizar chamadas HTTP para os endpoints da aplicação, é possível visualizar, na interface do Jaeger, cada etapa das requisições, com detalhes sobre duração, headers, spans e serviços envolvidos.

Para ampliar a visibilidade, foi adicionada a dependência opentelemetry-jdbc, que permite coletar traces também das interações com o banco de dados via JDBC. Uma configuração extra no application.properties (quarkus.datasource.jdbc.telemetry=true) ativa esse monitoramento. Com isso, os traces passam a incluir detalhes das queries executadas, conexões abertas e fechadas, proporcionando um diagnóstico mais completo.

Outro recurso apresentado foi o Dev Services, que detecta automaticamente a necessidade de serviços auxiliares (como bancos de dados) e sobe containers para prover esses serviços sem configuração manual. Ao mudar o banco H2 para PostgreSQL, o Quarkus detectou a mudança e iniciou automaticamente um container PostgreSQL, conectando-se a ele com configurações padrão.

Além do tracing, também foi demonstrada a coleta de métricas utilizando o Micrometer, biblioteca que também faz parte do ecossistema OpenTelemetry. A extensão micrometer-registry-prometheus foi adicionada para expor as métricas em formato compatível com o Prometheus, ferramenta popular para coleta e visualização de métricas.

Com isso, um endpoint /q/metrics foi disponibilizado, exibindo métricas padrão do runtime do Quarkus. Métricas personalizadas também foram adicionadas com anotações como @Counted, permitindo contabilizar o número de vezes que um endpoint foi chamado. Essas métricas podem ser utilizadas para acompanhar o comportamento da aplicação, detectar anomalias e realizar ajustes proativos.

Em resumo, a combinação de OpenTelemetry, Jaeger, Micrometer e Prometheus no Quarkus fornece uma solução poderosa e acessível para implantar observabilidade em aplicações cloud native. Essa visibilidade é essencial para garantir alta disponibilidade, identificar falhas rapidamente e melhorar continuamente a performance de sistemas distribuídos.

MÓDULO DE SPRING:

Capítulo 1: Spring – Introdução e Configuração do Projeto

Neste primeiro capítulo, iniciamos nosso módulo sobre desenvolvimento de APIs com Spring Boot. Para começar, é importante esclarecer algumas nomenclaturas fundamentais. Quando falamos em Spring, estamos nos referindo a um framework web amplamente conhecido e consolidado no ecossistema Java. Seu uso é dominante no mercado, estando presente em cerca

de 80 a 90% dos projetos de APIs em Java. Já o Spring Boot é uma abordagem que facilita o desenvolvimento dessas aplicações, eliminando a necessidade de configurações manuais extensas e servidores de aplicação externos como Tomcat ou JBoss.

Com Spring Boot, temos um servidor Tomcat embutido na aplicação, tornando-a autossuficiente. Isso significa que a aplicação pode ser executada diretamente, seja por linha de comando ou via um container Docker, facilitando o deploy e a escalabilidade, aspectos essenciais no desenvolvimento de microsserviços.

Para entender o papel do Spring, é preciso distinguir entre biblioteca, API e framework. Uma biblioteca é um conjunto de classes reutilizáveis de propósito geral. Uma API vai além disso, promovendo integração entre sistemas distintos. O framework, por sua vez, é de propósito específico, oferecendo não só bibliotecas, mas também uma estrutura e um modo de uso preestabelecido. No caso do Spring, ele define como devemos estruturar a aplicação, usando anotações para indicar ações do framework, como @RestController para classes que atendem requisições web e @GetMapping para métodos que respondem a requisições GET.

Utilizamos o Spring Initializr para gerar a estrutura inicial do projeto. Nessa plataforma, é possível definir metadados como tipo de projeto (Maven ou Gradle), linguagem (Java), versão do Spring Boot, grupo, artefato e versão do Java. Recomendamos sempre utilizar a versão estável sugerida pelo site. Para este projeto, selecionamos as dependências "Spring Web" e "Spring DevTools". A primeira inclui o Tomcat embutido e recursos para desenvolvimento de APIs, enquanto a segunda permite o hot reload durante o desenvolvimento.

Após gerar o projeto, extraímos o ZIP e importamos para a IDE. No caso do Eclipse, utilizamos a opção de importar como projeto Maven existente. O projeto é estruturado em pastas src/main/java para código fonte, src/main/resources para arquivos de configuração como o application.properties, e src/test/java para testes. A classe principal, anotada com @SpringBootApplication, contém o método main, ponto de entrada da aplicação.

Criamos um pacote controller e nele uma classe HelloController, anotada com @RestController. Dentro dessa classe, definimos um método sayHello() anotado com @GetMapping("/hello"), que retorna a mensagem "Hello World". Ao executar a aplicação e acessar http://localhost:8080/hello, o Spring Boot sobe o Tomcat embutido e responde corretamente à requisição.

É importante respeitar a hierarquia de pacotes: o Spring faz o scan das classes a partir do pacote base onde está localizada a classe principal da aplicação. Caso a estrutura de pacotes não esteja adequada, os controllers não serão identificados.

Criamos também uma classe Produto no pacote model, com atributos id, nome e preco, acompanhados de getters e setters. Na classe HelloController, adicionamos um novo endpoint @GetMapping("/produtos") que retorna um objeto Produto, automaticamente serializado em JSON pelo Jackson, biblioteca integrada ao Spring.

Para testar requisições POST, criamos o método addNewProduct() anotado com @PostMapping("/produtos"), que recebe um objeto Produto no corpo da requisição. Utilizando o Postman, enviamos um JSON com os dados do produto. Para que o Spring consiga fazer o binding corretamente, anotamos o parâmetro

do método com @RequestBody. Com isso, os dados são recebidos e processados corretamente.

Finalizamos esta aula com a base para o desenvolvimento de uma API REST em Spring Boot, entendendo a configuração inicial, estrutura do projeto, definição de endpoints e manipulação de objetos JSON. Este é o primeiro passo para construirmos aplicações robustas e escaláveis com Spring.

Capítulo 2: Spring – Criação de APIs Rest

Neste capítulo, vamos aprofundar o entendimento sobre as APIs REST e explorar a importância dos métodos HTTP no desenvolvimento de aplicações web modernas. Um ponto essencial é o conceito de statelessness das APIs REST. Diferente de aplicações web clássicas que mantinham informações de sessão (como no uso de JSP, Servlets ou PHP), uma API REST não armazena o histórico de requisições. Cada chamada é tratada de forma independente, como se fosse a primeira, sem qualquer dependência de estado anterior.

Outro ponto fundamental é o entendimento de que um endpoint em uma API REST não é apenas uma URL, mas sim a combinação entre URL e método HTTP. Assim, uma mesma URL como "/produtos" pode se referir a diferentes operações, dependendo se o método é GET, POST, PUT, PATCH ou DELETE:

- GET: recupera todos os produtos.
- POST: inclui um novo produto.

- PUT: altera todos os dados de um produto existente.
- PATCH: altera parcialmente um produto.
- DELETE: remove um produto do sistema.

Criamos então um ProdutoController, separando a lógica de produtos do HelloController. No ProdutoController, utilizamos uma ArrayList<Produto> como base de dados em memória, inicializada com alguns itens no construtor para simular um repositório.

Implementamos os seguintes endpoints:

1. **GET /produtos**: Retorna a lista completa de produtos.
2. **GET /produtos/{id}**: Retorna o produto com o ID informado. Se não encontrado, retorna 404.
3. **POST /produtos**: Adiciona um novo produto. O objeto é recebido via @RequestBody.
4. **PUT /produtos/{id}**: Atualiza um produto existente com base no ID. A substituição é total.
5. **DELETE /produtos/{id}**: Remove um produto com base no ID.
6. **GET /produtos/sort?order=asc**: Ordena os produtos por preço de forma ascendente ou descendente conforme o parâmetro order. Se for "asc", ordena do menor para o maior preço; se "desc", do maior para o menor; se inválido, retorna 400.

Cada operação está associada a uma resposta HTTP apropriada. Utilizamos a classe ResponseEntity para encapsular os retornos junto com os códigos de status HTTP:

- 200 OK: operação realizada com sucesso.

- 404 Not Found: recurso não encontrado.
- 400 Bad Request: requisição inválida.

Esses status ajudam o cliente da API a entender melhor o resultado da requisição e agir conforme esperado.

A estrutura de pacotes também foi ajustada, garantindo que cada parte da aplicação esteja organizada conforme a sua responsabilidade. Controllers, modelos, serviços (quando usados) e repositórios devem estar bem separados, facilitando a manutenção e evolução do projeto.

Com essa organização e entendimento dos métodos HTTP, conseguimos criar APIs REST mais semânticas, robustas e alinhadas às boas práticas do desenvolvimento moderno. Na próxima etapa, exploraremos a documentação dessas APIs para facilitar a integração com outras aplicações e sistemas.

Capítulo 3: Spring – Arquitetura Multicamadas e Documentação

Neste capítulo, exploramos dois pilares fundamentais do desenvolvimento de APIs com Spring Boot: a documentação automática dos endpoints e a arquitetura multicamadas baseada no padrão MVC (Model-View-Controller).

Começamos com a documentação da API. Em muitos cenários, especialmente quando os consumidores da API não têm acesso a ferramentas como o Postman, é essencial oferecer uma interface interativa e acessível. Para isso, utilizamos a extensão SpringDoc, mais especificamente o springdoc-openapi-starter-webmvc-ui. Essa dependência adiciona automaticamente uma interface Swagger acessível via http://localhost:8080/swagger-ui.html, permitindo explorar e testar todos os endpoints da aplicação.

Basta adicionar a dependência ao pom.xml, salvar e reiniciar o projeto. O Spring cuida do restante, gerando dinamicamente a documentação com base nas anotações já existentes nos controllers. Sem necessidade de HTML, sem configuração adicional. A interface Swagger permite não apenas visualizar os

endpoints, mas também executar chamadas HTTP diretamente do navegador, facilitando o teste e a validação da API.

Na segunda parte da aula, aprofundamos a arquitetura multicamadas baseada em MVC. A estruturação do projeto é dividida em pacotes com responsabilidades bem definidas:

- **Controller**: Contém as classes que expõem os endpoints. Responsável por receber requisições, encaminhar para o serviço correspondente e retornar respostas apropriadas.
- **Model (ou Entity)**: Representa os objetos de negócio e realiza o mapeamento entre as classes Java e as tabelas do banco de dados.
- **Repository**: Define interfaces de acesso ao banco. As operações como save, findById, deleteById, entre outras, são disponibilizadas via interfaces que extendem JpaRepository ou CrudRepository.
- **Service**: Camada intermediária onde são implementadas as regras de negócio. Essa separação promove a manutenção, a testabilidade e a reutilização de código.

Um aspecto essencial nessa arquitetura é a injeção de dependência. No Spring, definimos interfaces para os serviços (IMensagemService, por exemplo), que são utilizadas nos controllers. As implementações concretas são anotadas com @Service e injetadas via @Autowired. Isso permite, inclusive, trabalhar com múltiplas implementações de uma mesma interface utilizando @Qualifier, possibilitando alternar comportamentos em tempo de execução conforme a necessidade.

Discutimos também o princípio de alta coesão e baixo acoplamento. Alta coesão significa que cada classe tem uma responsabilidade clara e bem definida. Baixo acoplamento reduz a dependência entre os componentes, facilitando a substituição, evolução e manutenção do sistema.

Na prática, criamos uma interface IMessageService com um método getCustomMessage(String original). Implementamos duas versões:

- MessageServiceImplV1, que retorna a string em maiúsculas.
- MessageServiceImplV2, que substitui espaços por traços.

Ambas são anotadas com @Service e @Qualifier. No controller, especificamos via @Qualifier("v1") qual versão deve ser utilizada. Essa técnica permite alternar lógicas de negócio sem alterar o código do controller, favorecendo a extensibilidade e manutenção.

Além disso, destacamos o uso de **DTOs (Data Transfer Objects)**. DTOs são classes auxiliares criadas para representar dados trafegados entre as camadas. Podem ser utilizados para:

- Evitar exposição direta de entidades.
- Personalizar respostas de endpoints.
- Validar dados de entrada.
- Integrar com APIs externas.

A arquitetura proposta proporciona clareza nas responsabilidades de cada componente e prepara o projeto para futuras expansões, integrações e testes. Com uma API bem documentada via Swagger e uma estrutura bem definida, avançamos para os próximos tópicos com uma base sólida e profissional.

Capítulo 4: Spring – Integração com JPA

Neste capítulo, damos um passo decisivo rumo à persistência de dados no ecossistema Spring, com a integração do projeto à JPA (Java Persistence API) e ao banco de dados relacional. Para isso, utilizamos o Spring Data JPA com a implementação do Hibernate, criando um modelo completo de um sistema de inscrições para eventos com relacionamentos complexos entre entidades.

Inicialmente, estruturamos o projeto com os pacotes: controller, model, service e repository, mantendo a arquitetura multicamadas vista anteriormente. Utilizamos o Spring Initializr para gerar o projeto com as dependências: Spring Web, Spring DevTools, Spring Data JPA e o driver JDBC correspondente ao banco de dados (no caso, MySQL). O application.properties é configurado com os quatro parâmetros obrigatórios: spring.datasource.username, spring.datasource.password, spring.datasource.url e spring.jpa.properties.hibernate.dialect, além do spring.jpa.show-sql=true para exibir os comandos SQL gerados.

Adotamos um modelo de domínio para representação de conferências e inscrições. A entidade Conference possui atributos id, name e address, com mapeamento JPA via anotações @Entity, @Table, @Id, @GeneratedValue e

@Column. Cada conferência pode conter várias Session, formando uma relação de um para muitos.

A entidade Session inclui atributos como id, title, startDate e startTime, além da referência à conferência correspondente via @ManyToOne e @JoinColumn. Esse relacionamento garante que cada sessão está associada a uma conferência específica.

A entidade User é composta por id, name e email, com validação de unicidade e restrições como nullable = false e unique = true. Um aspecto importante é a decisão de usar tipos wrapper (Integer ao invés de int), permitindo diferenciar valores nulos e valores iniciais atribuídos automaticamente pela JVM, o que é fundamental para manter a integridade com os dados armazenados.

Avançamos para o relacionamento entre Session e User, inicialmente modelado como muitos para muitos com anotação @ManyToMany, @JoinTable, @JoinColumn e inverseJoinColumns. Essa estrutura permite mapear quais usuários estão inscritos em quais sessões, e vice-versa.

Porém, quando surge a necessidade de adicionar atributos à relação, como createdAt, level ou um uniqueId, o modelo puro muitos-para-muitos deixa de ser suficiente. A solução é transformar a associação em uma entidade intermediária chamada Subscription, com duas relações de um para muitos: uma entre User e Subscription, e outra entre Session e Subscription.

A entidade Subscription é definida como @Entity e mapeada para a tabela tbl_subscription. Como a chave primária é composta por User e Session, criamos a classe SubscriptionId, anotada com @Embeddable, contendo essas

referências com @ManyToOne e @JoinColumn. Essa classe é utilizada como ID composto em Subscription via @EmbeddedId.

A entidade Subscription também possui atributos adicionais, como createdAt (do tipo LocalDateTime), level (inteiro representando o nível de participação) e uniqueId (um identificador exclusivo). As colunas são mapeadas com @Column, permitindo definir nomes específicos, tamanhos (length) e restrições como nullable e unique.

Essa abordagem permite manter um modelo de dados relacional limpo e flexível, com alto grau de integridade e capacidade de evolução. Transformar uma relação muitos-para-muitos em duas relações um-para-muitos permite adicionar atributos específicos da interação entre as entidades, algo impossível no modelo direto.

Essa implementação reflete as boas práticas de modelagem orientada ao domínio com JPA, preparando o projeto para evoluir com segurança, performance e manutenção facilitada. No próximo capítulo, implementaremos os repositórios e serviços, finalizando a estrutura funcional da API baseada neste modelo de dados.

Capítulo 5: Spring – JPA - Services, Query By Method Name e conclusão da API

Neste capítulo, finalizamos a estrutura funcional da nossa API de eventos com a implementação completa das camadas de repositório e serviço. Com isso, garantimos a persistência e a lógica de negócio associada a entidades como Conferência, Sessão, Usuário e Inscrição (Subscription).

Criamos interfaces de repositório para cada uma dessas entidades usando ListCrudRepository, que estende as operações básicas de CRUD com suporte a listas diretamente. Cada repositório recebe como parâmetros o tipo da entidade e o tipo da chave primária. Por exemplo:

- ConferenceRepository extends ListCrudRepository<Conference, Integer>
- UserRepository extends ListCrudRepository<User, Integer>
- SessionRepository extends ListCrudRepository<Session, Integer>
- SubscriptionRepository extends ListCrudRepository<Subscription, SubscriptionId>

A vantagem do ListCrudRepository é o retorno direto de List<T> em operações como findAll, facilitando o uso e evitando conversões.

Na sequência, definimos interfaces de serviço como SubscriptionService, ConferenceService, SessionService e UserService. Cada uma dessas interfaces

declara os métodos específicos da lógica de negócio que manipulam as respectivas entidades. Por exemplo, o SubscriptionService define:

- addSubscription(Subscription subscription)
- getAllByUser(User user)
- getAllBySession(Session session)

A implementação dessas interfaces ocorre em classes como SubscriptionServiceImpl, ConferenceServiceImpl, SessionServiceImpl e UserServiceImpl, anotadas com @Service e usando injeção de dependência via construtor para obter acesso aos repositórios.

Para o método addSubscription, por exemplo, populamos os campos createdAt com LocalDateTime.now() e uniqueId com UUID.randomUUID().toString(), garantindo unicidade e marcação temporal. Em métodos de busca, como getAllByUser, utilizamos os recursos de query derivada do Spring Data, como findByUser(User user) e findBySession(Session session), eliminando a necessidade de escrever SQL manualmente.

Implementamos também um tratamento global de exceções com @ControllerAdvice. Criamos uma exceção personalizada NotFoundException que é lançada em casos de recurso inexistente. Um DTO ErrorDTO representa mensagens de erro e é retornado com código HTTP 404 quando aplicável. Isso elimina a necessidade de blocos try-catch repetitivos nos controllers.

Nos controllers, como SubscriptionController, UserController e ConferenceController, utilizamos a injeção de serviços e expomos endpoints REST para cadastro (@PostMapping) e consulta (@GetMapping) de dados. Os parâmetros de entrada são tratados com @RequestBody ou @PathVariable,

conforme o caso. A resposta é encapsulada em ResponseEntity, com status 200 (OK) ou 201 (Created) conforme a operação.

Durante os testes com o Postman, inserimos dados em tabelas como tbl_user, tbl_conference, tbl_session e tbl_subscription, e validamos a integração da API com o banco de dados MySQL. Também observamos o comportamento do pool de conexões do JPA, que por padrão cria múltiplas conexões simultâneas para atender a várias requisições.

Com isso, temos uma API funcional, organizada em camadas, com suporte a persistência, regras de negócio e tratamento de exceções, seguindo as boas práticas do ecossistema Spring.

Capítulo 6: Spring – Controle Transacional

Neste capítulo, exploramos o gerenciamento transacional no Spring, abordando tanto os fundamentos teóricos quanto sua aplicação prática em um cenário simulado de transferência bancária. Entender como funcionam as transações é essencial para garantir a integridade dos dados em sistemas que realizam múltiplas operações dependentes entre si.

Transações são blocos de operações que devem ser executadas de forma indivisível. Ou todas as operações são concluídas com sucesso, ou nenhuma delas é efetivada. As quatro propriedades fundamentais do gerenciamento transacional são resumidas pelo acrônimo ACID:

- **Atomicidade**: ou tudo é realizado ou nada é. Caso alguma operação falhe, realiza-se um rollback, retornando o sistema a um estado anterior.
- **Consistência**: a base de dados sai de um estado consistente e, após a transação, permanece consistente, respeitando todas as restrições e relacionamentos.
- **Isolamento**: transações simultâneas não interferem umas nas outras, garantindo integridade mesmo sob alta concorrência.
- **Durabilidade**: uma vez concluída a transação, suas alterações são persistidas e não serão perdidas, mesmo em caso de falha do sistema.

Na prática, desenvolvemos uma aplicação de transferência bancária com Spring Boot, estruturada em multicamadas, usando H2 como banco de dados local. O

diferencial é que, ao utilizar o modo file (e não mem), os dados são armazenados fisicamente em disco, simulando um ambiente mais próximo de produção.

Modelamos duas entidades principais:

- Account: com atributos number e balance, representa uma conta bancária.
- Transaction: com transactionNumber, timestamp, amount, debitAccount e creditAccount, representa uma transferência entre contas.

Cada entidade possui repositórios Spring Data JPA. Criamos exceções personalizadas como InvalidAccountException, InvalidBalanceAccountException e InvalidTransferException, tratadas por um handler global para retornar códigos HTTP apropriados (404 ou 400).

O TransferDTO encapsula os dados da transferência: conta de débito, conta de crédito e valor. Esse objeto é recebido pelo TransferController e repassado ao TransferService, onde toda a lógica da transação é centralizada.

No TransferService, injetamos os repositórios e aplicamos a anotação @Transactional ao método transferValue(TransferDTO dto). Essa anotação é a chave para que o Spring gerencie a transação automaticamente: se tudo ocorrer bem, os dados são persistidos; se houver qualquer exceção, todo o processo é revertido.

A lógica executa as seguintes etapas:

1. Recupera as contas de origem e destino usando os repositórios.
2. Atualiza o saldo da conta de destino somando o valor da transferência e salva.

3. Atualiza o saldo da conta de origem subtraindo o valor e salva.
4. Cria um objeto Transaction com os dados da operação.
5. Persiste a transação no banco de dados.

Caso o saldo da conta de origem fique negativo, uma exceção é lançada no setBalance, acionando o rollback automático. Isso garante que nenhuma parte da operação seja mantida, mesmo que a atualização na conta de destino já tenha ocorrido.

Realizamos testes utilizando o Postman, simulando transferências válidas e inválidas. Ao tentar transferir um valor superior ao saldo disponível, o sistema lança a exceção e a base de dados permanece inalterada, confirmando a eficácia do controle transacional.

O console do H2 foi utilizado para verificar o estado do banco antes e depois das operações. O comportamento foi o esperado: em operações bem-sucedidas, as contas foram atualizadas e a transação registrada; em casos de erro, nenhuma alteração foi salva.

Esse exemplo demonstra na prática como aplicar as propriedades ACID em uma aplicação Spring Boot. Com apenas uma anotação, garantimos que um conjunto complexo de operações interdependentes seja tratado com segurança e confiabilidade. O controle transacional é uma das funcionalidades mais importantes em sistemas empresariais, e o Spring fornece um suporte completo e intuitivo para sua aplicação.

Capítulo 7: Spring – Autenticação com JWT e Oauth

Neste capítulo, abordamos os fundamentos de segurança em APIs REST utilizando Spring Security, com foco na autenticação via token JWT. O objetivo é garantir que apenas usuários autenticados possam acessar determinados recursos da aplicação, mantendo o princípio de statelessness característico das APIs REST.

Inicialmente, criamos um projeto no Spring Initializr com as dependências: Spring Web, Spring Data JPA, Spring DevTools e H2. Em seguida, estruturamos a aplicação com uma entidade User contendo os campos id, username e password. O repositório UserRepository utiliza Query By Method Name com um método findByUsername para recuperar usuários com base no nome de usuário.

Com o Spring Security adicionado ao pom.xml, percebemos que automaticamente um mecanismo padrão de autenticação é aplicado. A API passa a exigir autenticação básica, e um log com senha gerada é exibido. Esse comportamento é útil para testes iniciais, mas inadequado para APIs REST. O ideal é utilizar JWT (JSON Web Token), que funciona como um crachá digital enviado em cada requisição.

Para configurar o controle de acesso, criamos uma classe WebSecurityConfig com as anotações @Configuration e @EnableWebSecurity. Implementamos um método que retorna um SecurityFilterChain, onde definimos que o endpoint /open será acessível a todos, enquanto os demais exigirão autenticação. Desabilitamos o CSRF e utilizamos authorizeHttpRequests para customizar as permissões.

Criamos um filtro customizado AuthFilter estendendo OncePerRequestFilter. Esse filtro é registrado com addFilterBefore para ser executado antes do

UsernamePasswordAuthenticationFilter. No AuthFilter, interceptamos todas as requisições e buscamos um cabeçalho HTTP chamado Authorization. Se ele contiver a string Bearer security123, consideramos a requisição válida.

Para isso, desenvolvemos a classe TokenUtil, com um método estático decode(HttpServletRequest request) que extrai o token do cabeçalho, valida seu valor, e, se correto, retorna um objeto UsernamePasswordAuthenticationToken. Esse objeto é então registrado no SecurityContextHolder, integrando o contexto de segurança do Spring.

Testamos com o Postman o acesso aos endpoints. O /open é acessível sem autenticação. O /restricted retorna 403 Forbidden quando acessado sem token, mas retorna 200 OK quando o header Authorization: Bearer security123 é incluído. Esse comportamento confirma a correta integração do filtro e do controle de acesso baseado em tokens.

Esse capítulo estabeleceu as bases para o uso de JWT no controle de autenticação de APIs REST com Spring Security. A próxima etapa será criptografar a senha dos usuários e gerar dinamicamente os tokens JWT após o login. Essa abordagem oferecerá um modelo seguro e escalável para proteger APIs em ambientes reais.

Capítulo 8: Spring – Geração de Token JWT com autenticação customizada

Neste capítulo, damos continuidade ao processo de segurança em APIs com Spring Boot, focando na geração e validação de tokens JWT com autenticação

customizada. A ideia central é substituir o token estático por um JWT gerado dinamicamente e criptografado, garantindo maior segurança e conformidade com padrões modernos de autenticação.

Inicialmente, ajustamos a funcionalidade de criação de usuários para que as senhas sejam salvas de forma criptografada usando o algoritmo Bcrypt, já embutido no Spring Security. Ao receber um novo usuário via POST /users, a senha é criptografada com BcryptPasswordEncoder.encode() antes de ser persistida no banco. Para liberar esse endpoint, ajustamos a classe de configuração de segurança, permitindo o acesso ao método POST em /users.

Em seguida, criamos o processo de login via endpoint POST /login. O login recebe um objeto User contendo username e password. O serviço valida se o usuário existe e, utilizando passwordEncoder.matches(), compara a senha fornecida com a senha criptografada armazenada. Em caso de sucesso, é gerado um token JWT.

Definimos um DTO MyToken, um record que encapsula o campo token, retornado como resposta em caso de login válido. Se o usuário não for encontrado ou a senha estiver incorreta, uma RuntimeException é lançada. Para tratar exceções, configuramos uma classe @ControllerAdvice que intercepta RuntimeException e retorna status 403 Forbidden com a mensagem de erro.

A geração do JWT é realizada na classe TokenUtil. Nela, criamos os métodos encode(User) e decode(HttpServletRequest). O token é gerado usando a biblioteca JJWT, com os seguintes atributos:

- **Subject**: username do usuário
- **Issuer**: identificador da aplicação (ex: "professor_isidro")

- **Expiration**: tempo de expiração em milissegundos (ex: 5 minutos)

A chave secreta é definida com pelo menos 32 caracteres, garantindo 256 bits de segurança. O token é assinado com essa chave e compactado em uma string. Essa string é o token JWT retornado ao cliente após o login.

A decodificação é feita no método decode(), que extrai o token do cabeçalho Authorization, verifica sua validade com base em subject, issuer e expiration, e retorna um UsernamePasswordAuthenticationToken se válido. Esse processo é usado no filtro customizado para autenticar requisições subsequentes.

Ao testar com o Postman, geramos um token via login e usamos esse token em requisições a endpoints protegidos. Se o token estiver correto e dentro do prazo de validade, o acesso é autorizado. Caso contrário, a resposta é 403 Forbidden.

Configuramos também uma expiração curta de 15 segundos para demonstrar a funcionalidade. Após esse tempo, o mesmo token é considerado inválido e a requisição é negada.

Finalizamos com uma introdução ao funcionamento do OAuth. Em aplicações com login via Google ou GitHub, o fluxo é iniciado no frontend, que redireciona para o provedor de identidade (ex: Google). Após autenticação, um token é retornado ao frontend, que então o envia para a API. A API valida o token (possivelmente usando um decodificador como JWTDecoder) e permite o acesso. Com isso, a autenticação e a gestão de credenciais são delegadas a terceiros, aumentando a segurança e simplificando o gerenciamento.

Esse processo completo oferece um modelo robusto de autenticação em aplicações Spring Boot, compatível com padrões modernos de segurança e ideal para ambientes distribuídos.

Capítulo 9: Spring – Exemplo de OAuth2 com Google

Neste capítulo, exploramos um exemplo prático de autenticação utilizando o protocolo OAuth2 com o provedor Google. A proposta é demonstrar como configurar uma aplicação para receber um token JWT diretamente da Google Identity Platform, de forma simples e eficaz, sem a necessidade de armazenar senhas localmente.

Para isso, iniciamos com a configuração no console da Google Cloud. A primeira etapa é acessar a seção "API e serviços" e, em seguida, "Tela de permissão OAuth". Definimos um novo projeto e criamos um cliente OAuth do tipo "aplicação web". Nessa etapa, é importante informar a URI autorizada para o login (por exemplo, http://localhost:5500) e a URI de redirecionamento (geralmente a mesma, quando em ambiente local).

Ao finalizar essa configuração, o Google fornece um client_id e um client_secret. Para o nosso exemplo, apenas o client_id é utilizado. Com isso em mãos, desenvolvemos uma página HTML básica que carrega os scripts da Google Identity, define o client_id e inclui a div do botão de login.

O botão de login possui um data-callback que aponta para uma função handleCredentialResponse. Essa função é responsável por capturar o token JWT enviado pelo Google após a autenticação do usuário.

Esse token JWT pode ser visualizado e analisado usando o site jwt.io, que permite verificar o payload decodificado. Entre as informações disponíveis estão: email, nome, domínio e até a URL da foto do usuário. Esses dados podem ser utilizados para realizar o cadastro ou login do usuário diretamente na API, sem a necessidade de armazenar senhas.

A ideia central é simples: o frontend realiza o login com o Google e recebe um JWT. Esse token é então enviado para a API, que pode:

1. Validar a assinatura do token.
2. Decodificar o payload.
3. Verificar se o email existe no sistema.
4. Criar ou autenticar o usuário automaticamente.

Com isso, a aplicação se beneficia de um modelo seguro e responsivo de autenticação, delegando a responsabilidade de login ao Google. Isso simplifica a gestão de credenciais e melhora a experiência do usuário.

Ao testar a aplicação, observamos que o login com o Google redireciona corretamente, e o token JWT é exibido no console. Ao decodificar esse token, é possível acessar todas as informações necessárias para identificar o usuário, como nome e email, e até mesmo sua foto.

Esse fluxo OAuth2, embora simples no frontend, requer que a API esteja preparada para validar e processar o token JWT. No contexto do Spring Boot, podemos usar bibliotecas como jjwt ou JWTDecoder para fazer essa validação.

Em resumo, esse capítulo mostrou como implementar um login com Google via OAuth2, demonstrando como capturar e processar um token JWT recebido no frontend, criando uma ponte segura e moderna entre usuário, Google Identity e nossa API.

Capítulo 10: Spring – WebFlux

Neste último capítulo do módulo de Spring, exploramos a programação reativa com Spring WebFlux, uma abordagem orientada à eficiência de recursos em aplicações que lidam com chamadas assíncronas e APIs externas de resposta lenta. O objetivo é evitar o bloqueio de requisições e melhorar a experiência do cliente ao liberar imediatamente a aplicação para outras tarefas.

Para contextualizar, foi apresentada uma API externa fictícia (Slow API), cuja resposta tem um atraso proposital de cinco segundos. Esse cenário simula chamadas para serviços governamentais, como a Receita Federal, que exigem

tempo para processamento de documentos como notas fiscais. A solução proposta é criar uma API reativa intermediária que consome a API lenta e responde de forma imediata ao cliente com um protocolo de rastreamento, permitindo a posterior consulta ao resultado.

A aplicação foi estruturada com as dependências: Spring Reactive Web, Web, DevTools, JPA e H2. A configuração do banco de dados foi feita para operar em modo reativo, utilizando o H2 com o nome reactiveDB, e ajustado o application.properties para exibir comandos SQL, habilitar o console web e gerar automaticamente as tabelas.

A entidade DocFiscal representa o documento fiscal e possui atributos id, protocolo e documento (um JSON armazenado como String). O repositório DocFiscalRepo expõe um método findByProtocolo, e o serviço DocFiscalService define duas operações principais:

- realizarAutorizacaoApiExterna(Long idCliente, Integer idServico, String protocolo): consome a API externa via WebClient.
- consultarPorProtocolo(String protocolo): retorna o documento com base no protocolo.

O componente WebClient, principal ferramenta de integração reativa, foi configurado em uma classe WebClientConfiguration com a anotação @Configuration, expondo um @Bean do tipo WebClient. Isso permite a injeção do cliente HTTP nas classes de serviço, promovendo desacoplamento e reutilização.

A chamada à API externa é feita via webClient.get().uri(...).retrieve().bodyToMono(String.class), com os dados

extraídos da URI. O resultado é processado em assinaturas doOnNext e doOnError. Caso a chamada seja bem-sucedida, o JSON retornado é armazenado no banco com o protocolo previamente gerado.

Para orquestrar essa lógica, foi criado o DocFiscalController, com dois endpoints principais:

1. POST /solicitar: recebe um DTO com idCliente e idServico, gera um protocolo aleatório (UUID), dispara a chamada reativa à API externa e retorna um 202 Accepted com o protocolo.
2. GET /consultar/{protocolo}: permite consultar o documento fiscal armazenado com base no protocolo fornecido.

O comportamento reativo é confirmado pelo uso de Mono.just(...) e a resposta imediata com ResponseEntity.accepted().body(...), indicando que a requisição foi recebida, mas seu processamento ainda está em andamento.

Testes com Postman demonstraram que:

- Ao disparar uma requisição para /solicitar, a resposta é imediata com o protocolo.
- A API externa continua processando a requisição em background.
- Posteriormente, o cliente pode consultar o resultado com o protocolo em /consultar/{protocolo}.

O console do H2 confirma que os dados são persistidos corretamente, com o documento armazenado como string e associado ao protocolo.

Essa abordagem ilustra claramente o poder da programação reativa no Spring Boot, permitindo escalar aplicações, liberar threads bloqueadas e construir sistemas responsivos mesmo em face de integrações com serviços lentos.

Concluímos assim o módulo de Spring, cobrindo desde configuração inicial e arquitetura multicamadas até temas avançados como autenticação JWT, OAuth2 e WebFlux. Com esse conhecimento, é possível construir aplicações robustas, seguras e escaláveis com o ecossistema Spring.