



API RESTful com Spring Boot e Java 8

Guia de referência no formato problema e solução
sobre API RESTful em 34 tópicos

{ REST API }

Márcio Casale de Souza
kazale.com

Introdução	2
Autor	4
Problemas e soluções	5
1. Como começar?	6
2. Instalando o Java 8	7
3. Instalando a IDE de desenvolvimento	9
4. Entendendo o Spring Boot	11
5. Criando um projeto com o Spring Boot	12
6. Versionando o código fonte com o GitHub	14
7. Integração contínua com o TravisCI	18
8. Instalando o MySQL para persistir os dados do sistema	20
9. Adicionando o JPA ao projeto Spring Boot	21
10. Adicionando o MySQL ao projeto Spring Boot	23
11. Adicionando o H2 ao projeto Spring Boot	25
12. Parâmetros de configuração (application.properties)	26
13. Profiles	27
14. Gerando senhas com o BCrypt	29
15. Criando Entidades JPA (Entity)	32
16. Criando Repositórios JPA (JpaRepository)	36
17. Versionamento de banco de dados com o Flyway	38
18. Configurando um pool de conexões de banco de dados com Hikari	41
19. Criando serviços (Service)	43
20. Criando controllers (Controller)	45
21. Utilizando o Postman	47
22. DTO (Data Transfer Object)	48
23. Encapsulando o retorno de uma chamada a API Restful (Response)	51
24. Validação de dados (Bean Validation)	54
25. Documentando API com Swagger	58
26. Versionamento de API	61
27. Autenticação e autorização com tokens JWT (Json Web Token)	64
28. Adicionando cache com EhCache	92
29. Teste de stress e performance com o Apache AB	95
30. Monitorando a JVM com o VisualVM	97
31. Instalando o MongoDB para persistir os dados do sistema	98
32. Adicionando o MongoDB ao projeto Spring Boot	99
33. Criando entities e repositórios com o MongoDB	100
34. Publicando a API no Heroku	105
Conclusão	107

Introdução

O mercado de desenvolvimento tem avançado e apresentado inúmeros recursos e padrões novos, que devem ser seguidos para atender a demanda de acessos e dados que precisamos manipular nos dias de hoje.

APIs Restful se tornaram peça chave para criação de aplicações robustas, seguindo padrões de micro serviços, além de trabalharem de modo standalone, ou seja, sem que uma requisição dependa de outra para executar uma determinada operação.

Este livro tem como objetivo apresentar de modo objetivo e de fácil entendimento, como criar APIs robustas utilizando o Java 8 com a suíte de frameworks que compõem o Spring, como o Spring Boot, Spring Data, Spring Rest, entre outros.

O conteúdo aqui apresentado é para quem deseja adquirir conhecimentos necessários para a criação de APIs Restful com Spring Boot, Java 8, e muitos outros frameworks e ferramentas complementares, seguindo padrões que são utilizados pelas maiores empresas de TI.

No livro serão abordados tópicos que não envolvem somente a parte do desenvolvimento da API Restful, mas também o processo de criação dela, que envolve versionamento de código fonte com Git e GitHub, integração contínua com Travis CI, análise de performance com o Apache AB, todos recursos que você precisará saber para ser um profissional excepcional.

O que será apresentado neste livro é muito difícil de ser encontrado em um único lugar, principalmente no que diz respeito a processos de autenticação e autorização utilizando Tokens JWT, recursos atualizados do Java 8, versionamento de APIs, cache, deploy na nuvem, dentre muitos outros.

O livro ensinará individualmente sobre todos os recursos a serem estudados para a criação de uma API Restful, no formato “problema”, “solução”, e “como fazer”.

Esse formato é uma excelente fonte de consulta, pois cada tópico poderá ser estudado individualmente e na ordem desejada, ou seja, servirá também como referência na criação de uma API Restful completa.

Seguramente este livro será de grande valor para o aperfeiçoamento de seus conhecimentos, e será um livro de cabeceira, no qual você sempre o utilizará para consultas, e como um guia de referência, sempre que uma dúvida surgir.

Bom estudos!

Autor



Sou Márcio Casale de Souza, engenheiro de software sênior, com conhecimentos em inúmeras tecnologias open source, como Java, PHP, Angular, Linux, MySQL, dentre muitas outras.

Atuo com TI desde 2007, e já trabalhei para inúmeras empresas no Brasil e também no exterior.

Adquiri muitos conhecimentos ao longo de minha carreira, tendo trabalhado em projetos de alto impacto e escalabilidade.

Atualmente trabalho na Rentalcars.com em Manchester - UK, que é o maior website de busca de locadoras de veículos existente.

Por ser o maior, temos que utilizar as últimas tecnologias e as melhores ferramentas de arquitetura para conseguir suprir todas as necessidades do negócios, que recebe milhares de visitas todos os dias.

Decidi escrever esse livro para compartilhar um pouco do que aprendi durante todos esses anos, como as empresas top existentes implementam seus sistemas, possibilitando assim que cada vez mais e mais pessoas avancem nessa área que está a cada dia mais desafiadora e cheia de oportunidades.

Espero que com esse trabalho você consiga adquirir os conhecimentos necessários para alavancar a sua carreira, e também que este livro seja utilizado como referência no seu dia a dia de trabalho para consultas de como executar determinada tarefa.

Bons estudos!

Problemas e soluções

1. Como começar?

⇒ **Problema:**

Gostaria de entender como o livro está organizado, e como devo fazer para tirar o máximo proveito dele.

⇒ **Solução:**

O livro está organizado em tópicos, onde serão apresentados problemas, soluções, e como executar as soluções na prática.

⇒ **Como fazer:**

Pelo fato do livro ser um guia de referência no formato de problemas e suas respectivas soluções, ele pode ser estudado na sequência desejada, começando pelos tópicos de maior interesse ou prioridade.

Caso um tópico em específico possua alguma dependência, você será notificado na descrição do mesmo.

Vamos lá, escolha o seu tópico de interesse e comece agora mesmo!

2. Instalando o Java 8

⇒ **Problema:**

Gostaria de configurar meu computador para poder desenvolver aplicações em Java.

⇒ **Solução:**

Para desenvolver aplicações com o Java, será necessário instalar a JDK (Java Development Kit), que como o próprio nome diz, é o kit de desenvolvimento Java.

Para criar códigos atualizados e utilizando seus últimos recursos, a versão a ser instalada será a versão 8, que possui muitas melhorias e recursos que tornam o desenvolvimento mais simples e eficiente.

⇒ **Como fazer:**

Para instalar o Java 8 em seu computador, você deverá acessar a seguinte URL:

<http://www.oracle.com/technetwork/pt/java/javase/downloads/index.html>

Clique na imagem do download do Java para ser redirecionado para a página de download.

Na página exibida, clique no link da última versão disponível do Java para o seu sistema operacional, para que o download seja iniciado.

Ao término do download, execute o arquivo de instalação e siga os passos na tela.

No caso do Linux, dependendo da distribuição, será necessário apenas extrair o arquivo baixado, e adicionar seu caminho na variável 'PATH' do sistema operacional, para que os comandos Java fiquem visíveis para o sistema.

Para validar a instalação do Java, abra o console/terminal e digite o seguinte comando:

```
java -version
```

Deverá ser exibida a versão atual do Java instalado em seu computador, de deverá ser a mesma que você realizou o download no site oficial.

3. Instalando a IDE de desenvolvimento

⇒ **Problema:**

Gostaria de ter um ambiente integrado de desenvolvimento, onde eu pudesse ter acesso a recursos que facilitasse o desenvolvimento de aplicações em Java 8 com Spring.

⇒ **Solução:**

O Java possui algumas boas opções de IDEs para desenvolvimento, e como você quer desenvolver em Java com o Spring, a IDE mais adequada é o STS (Spring Tool Suite), que é a IDE oficial desenvolvida e mantida pela própria equipe de desenvolvimento do Spring.

O STS é uma IDE que possui sua base em uma das mais populares IDEs existentes, que é o Eclipse, que além de ser muito completa e robusta, possui suporte para os mais variados sistemas operacionais existente.

⇒ **Como fazer:**

Para instalar o STS, acesse a URL a seguir:

<https://spring.io/tools/sts/all>

Nela serão exibidas todas as opções de downloads, para Windows, Mac, e Linux.

Selecione a versão referente ao seu sistema operacional e arquitetura, e clique no link para realizar o download.

Após o término do download, basta descompactar o arquivo e clicar no executável do STS.

Vale lembrar que o processo de instalação é o mesmo para todos os sistemas operacionais.

Ao abrir o STS, será solicitado um diretório para o armazenamento do projeto (workspace). Fique a vontade para escolher um de sua preferência.

É importante lembrar que esse diretório poderá ser alterado a qualquer momento.

4. Entendendo o Spring Boot

⇒ **Problema:**

Gostaria de simplificar o desenvolvimento de aplicações em Java, ter uma versão do meu sistema pronto para produção sem a necessidade de fazer inúmeras configurações e otimizações.

⇒ **Solução:**

O Spring Boot é um framework que permite ter uma aplicação rodando em produção rapidamente, além de seguir as melhores práticas de design, e com configurações já otimizadas.

Ele oferece uma série de templates, ferramentas previamente configuradas, segue convenções ao invés de configurações, ou seja, fornece a maioria dos recursos necessários já prontos para utilização.

⇒ **Como fazer:**

O modo mais simples de criar um projeto Spring Boot é utilizando o STS, pois ele possui um assistente de criação de projetos que simplifica muito o trabalho de configuração.

Vide o próximo item para aprender como criar um projeto com o Spring Boot.

5. Criando um projeto com o Spring Boot

⇒ **Problema:**

Gostaria de criar um projeto com Spring Boot, seguindo as melhores práticas para ter uma aplicação pronta para produção com o mínimo esforço possível.

⇒ **Solução:**

Para criar uma aplicação com o Spring Boot, nada melhor do que usar o STS, que é a IDE oficial para desenvolvimento de aplicações Java com o Spring.

Ela permite criar projetos Spring Boot através de templates e assistentes, assim você poderá criar um projeto já com todos os recursos que utilizará ao longo do desenvolvimento de seu sistema.

⇒ **Como fazer:**

Com o seu STS aberto, clique em:

File -> New -> Spring Starter Project

Como por padrão ele utiliza o Maven para gerenciamento de dependências, que será o gerenciador que utilizaremos aqui, será necessário preencher algumas informações relacionadas a ele.

Name: Nome do seu projeto, como “MeuPrimeiroProjeto”, sem espaços

Group: Domínio a qual pertence o seu projeto, como “com.meudominio”

Artifact: Nome do arquivo ao gerar o build, utilize nomes simples, sem espaços, como “meu-primeiro-projeto”

Version: Versão inicial da aplicação, pode ser deixado como o padrão

Description: Descrição da aplicação, descreva em uma sentença do que se trata sua aplicação

Package Name: Nome do pacote base de sua aplicação, como “com.meudominio.api”

Clique em “next”.

Na tela seguinte você poderá selecionar as dependências do seu projeto, como JPA, Web, Cache, Security, entre muitos outros.

Para propósitos de um primeiro projeto, selecione apenas o DevTools, que está dentro de Core. Esse pacote auxilia o desenvolvimento do projeto adicionando alguns recursos, como a reinicialização automática do servidor quando um arquivo é modificado.

Clique em “finish” para concluir a criação do projeto, que deverá ser exibido na aba da esquerda do STS.

6. Versionando o código fonte com o GitHub

⇒ **Problema:**

Gostaria de manter o meu código centralizado, seguro, criando versões entre os novos recursos adicionados, além que facilitar o desenvolvimento do meu sistema em equipe.

⇒ **Solução:**

Todo desenvolvimento profissional, mesmo que sendo somente criado por um único desenvolvedor, depende da utilização de um sistema de gerenciamento de código fonte.

O GitHub é o principal sistema de controle de versão de código fonte utilizado para projetos open source ou mesmo privados.

Ele é gratuito para hospedagem de projetos open source, é de fácil utilização, e utiliza como base o Git, que é um sistema totalmente estável e seguro .

⇒ **Como fazer:**

Para utilizar o Git com o GitHub, você primeiramente deverá instalar o Git em seu computador, para isso:

Acesse <https://git-scm.com/downloads>, selecione o seu sistema operacional, e siga as instruções de instalação.

Windows e Mac consistem basicamente em baixar um executável e realizar a instalação. Para Linux existem as instruções detalhadas para cada distribuição sobre como proceder para instalar.

Após a instalação do Git, crie uma conta no GitHub, acessando:

<http://github.com>

E para concluir a configuração, crie um repositório no GitHub, clicando no botão “New repository”.

Escolha um nome para o repositório, como “primeiro-projeto-sts”

Deixe ele como público, e selecione a opção “Initialize this repository with a README”.

Não adicione um arquivo ‘.gitignore’, pois o STS adiciona um por padrão ao criar um projeto.

Clique em “Create repository” para finalizar.

Com o Git e GitHub configurados, vamos versionar o projeto criado no tópico anterior.

Acesse via de comando (terminal) a raiz do projeto STS a ser versionado.

Execute o seguinte comando para inicializar o repositório localmente:

```
git init
```

Associe o projeto local com o repositório previamente criado:

```
git remote add origin https://github.com/seu-usuario/nome-repositorio
```

Sincronize ambos os projetos:

```
git pull origin master
```

Adicione os novos arquivos ao projeto:

```
git add .
```

Realize o commit dos novos arquivos:

```
git commit -m “Arquivos iniciais do projeto”
```

Para finalizar, envie os novos arquivos para o repositório do GitHub:

```
git push origin master
```

Pronto, agora sempre que realizar uma alteração, basta executar os comandos “git add”, “git commit”, e “git push”, conforme executados anteriormente.

Caso você precise importar o seu projeto novamente, siga os seguintes passos:

Execute no terminal o comando a seguir para clonar o repositório:

```
git clone https://github.com/seu-usuario/nome-repositorio
```

Acesse o repositório recém clonado, e execute o seguinte comando Maven (que deverá estar instalado no sistema <https://maven.apache.org/download.cgi>):

```
mvn eclipse:eclipse
```

Esse comando criará as configurações necessárias para que o projeto possa ser importado no STS.

Na sequência, abra o STS, e clique em:

```
File -> Import...
```

Na janela exibida, “Existing Maven Projects”, que está abaixo de “Maven”.

Clique em “next”, e depois no botão “Browse...” para selecionar onde seu projeto está.

Clique em “finish”. Pronto seu projeto já está pronto para uso.

Para saber mais sobre o Git e GitHub, verifique o mini curso gratuito disponível em <http://kazale.com/curso-basico-git-github/>

7. Integração contínua com o TravisCI

⇒ **Problema:**

Gostaria de certificar que minha alteração no código fonte não tenha inserido nenhum erro no código existente, e que todos os testes estejam passando.

⇒ **Solução:**

Para certificar que o suas alterações no código fonte estejam sempre funcionando, e executando e passando todos os testes, você deverá utilizar uma ferramenta de integração contínua.

Tal ferramenta será responsável por executar automaticamente todos os testes e processo de build de sua aplicação automaticamente.

O TravisCI é um serviço de integração contínua que funciona perfeitamente com repositórios hospedados no GitHub.

Sempre que uma alteração for enviada para o GitHub, o TravisCI automaticamente inicializará o processo de build de seu código fonte, e o notificará caso algum erro venha a ocorrer.

⇒ **Como fazer:**

Acesse o TravisCI em <http://travis-ci.org>.

Clique em “Sign in with GitHub” para associar ambas as contas.

Clique a esquerda no “+” para adicionar um repositório.

Ative o repositório desejado na tela exibida.

Crie na raiz do projeto um arquivo chamado `.travis.yml`

Adicione o seguinte conteúdo nele (utilize indentação de 2 espaços):

```
language: java  
jdk:  
  - oraclejdk8
```

Esse comando serve para informar que usaremos o Java 8 para executar a aplicação.

Realize o commit do arquivo conforme explicado no tópico anterior.

Acesse o painel do TravisCI para verificar o build do projeto.

8. Instalando o MySQL para persistir os dados do sistema

⇒ **Problema:**

Gostaria de persistir os dados do sistema no banco de dados MySQL.

⇒ **Solução:**

Para persistir os dados de sua aplicação no MySQL, será necessário instalar o servidor de banco de dados MySQL.

O MySQL pode ser instalado nos principais sistemas operações, e sua instalação pode variar de sistema para sistema.

Uma dica interessante seria instalar o MySQL em conjunto com o PHP, Apache HTTP e PhpMyAdmin, assim você terá uma suíte completa de gerenciamento do MySQL.

Tal suíte recebe o nome de WAMP para plataforma Windows, LAMP para plataforma Linux, e MAMP para plataforma Mac.

⇒ **Como fazer:**

A instalação em diferentes sistemas operacionais varia bastante, assim o melhor a fazer é acessar <https://dev.mysql.com/downloads/mysql/> e procurar pela versão desejada.

Para usuário Windows ou Mac, recomendo a instalação do MAMP (<https://www.mamp.info/en/downloads/>) para usuário do Mac, ou o WAMP (<http://www.wampserver.com/en/>) para usuários do Windows, assim você terá acesso a ferramentas extras para o gerenciamento de seu banco de dados.

Usuários Linux podem obter a mesma configuração, mas devem proceder com a instalação manualmente.

9. Adicionando o JPA ao projeto Spring Boot

⇒ **Problema:**

Gostaria de adicionar suporte JPA ao meu projeto Spring Boot, a fim de utilizar um banco de dados nele.

⇒ **Solução:**

É possível adicionar suporte ao JPA de dois modos distintos.

O primeiro e mais fácil é selecionando o Spring Data ao criar o seu projeto com o assistente de criação de projetos do STS.

O segundo é adicionando manualmente a dependência do Spring Data no arquivo de configuração do Maven ou Gradle.

⇒ **Como fazer:**

Caso você não tenha selecionado a opção JPA ao criar o seu projeto, basta abrir o arquivo pom.xml, encontrado na raiz do projeto, e adicionar o seguinte conteúdo nele:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.data</groupId>
    <artifactId>spring-data-jpa</artifactId>
  </dependency>
</dependencies>
<repositories>
  <repository>
    <id>spring-libs-release</id>
    <name>Spring Releases</name>
    <url>https://repo.spring.io/libs-release</url>
    <snapshots>
      <enabled>false</enabled>
    </snapshots>
  </repository>
</repositories>
```

```
</repository>  
</repositories>
```

Salve o arquivo e o STS automaticamente realizará o download das dependências necessárias.

10. Adicionando o MySQL ao projeto Spring Boot

⇒ **Problema:**

Gostaria de utilizar o MySQL em meu projeto Java com Spring Boot.

⇒ **Solução:**

Para que o Java consiga acessar o MySQL, será necessária a configuração do driver do MySQL para o Java, assim ele saberá como proceder para tal acesso.

Tal configuração é realizada adicionando a dependência do driver do MySQL ao projeto Spring Boot.

⇒ **Como fazer:**

O modo mais simples para adicionar o driver do MySQL ao projeto, é selecioná-lo ele ao executar o assistente de criação de projetos no STS.

Para adicionar o driver do MySQL ao projeto Spring Boot após sua criação, adicione a seguinte instrução ao arquivo pom.xml:

```
<dependencies>
  <dependency>
    <groupId>mysql</groupId>
    <artifactId>mysql-connector-java</artifactId>
  </dependency>
</dependencies>
```

Para configurar a aplicação com o MySQL instalado localmente, abra o seguinte arquivo:

```
src/main/resources/application.properties
```

E adicione nele as seguintes configurações (de acordo com sua instalação):

```
spring.jpa.hibernate.ddl-auto=create  
spring.datasource.url=jdbc:mysql://localhost:3306/nome-banco-de-dados  
spring.datasource.username=USUARIO_BANCO_DE_DADOS  
spring.datasource.password=SENHA_BANCO_DE_DADOS
```

11. Adicionando o H2 ao projeto Spring Boot

⇒ **Problema:**

Gostaria de utilizar o banco de dados H2 em meu projeto Java com Spring Boot, para que meus dados sejam persistidos em memória.

⇒ **Solução:**

Para que o Java em conjunto com o Spring consigam utilizar o H2, será necessário a configuração do driver do H2 para o Java.

Tal configuração é realizada adicionando a dependência do H2 ao projeto Spring Boot.

⇒ **Como fazer:**

O modo mais simples para adicionar o H2 ao projeto, é selecioná-lo ele ao executar o assistente de criação de projetos no STS.

Para adicionar o driver do H2 ao projeto Spring Boot após sua criação, adicione a seguinte instrução ao arquivo pom.xml:

```
<dependencies>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>
```

Salve o arquivo para que o STS realize o download das dependências necessárias.

Nenhuma configuração adicional é necessária, e o Spring se encarregará inclusive de garantir que as entidades sejam criadas automaticamente.

12. Parâmetros de configuração (application.properties)

⇒ **Problema:**

Gostaria que os parâmetros de configuração da aplicação, como dados de acesso ao banco de dados, urls de serviços, ficassem armazenado em um único lugar, e que eu pudesse acessá-los de qualquer serviço ou componente do Spring a qualquer momento.

⇒ **Solução:**

O Spring Boot oferece por padrão a configuração de um arquivo de propriedades, chamado 'application.properties', que serve para o armazenamento de configurações no formato chave e valor.

⇒ **Como fazer:**

Abra o arquivo de configurações:

```
src/main/resources/application.properties
```

E adicione a seguinte entrada nele:

```
paginacao.qtd_por_pagina=25
```

Para ter acesso ao valor adicionado acima, devemos utilizar a anotação '@Value' em um componente Spring.

O código do componente ficaria da seguinte forma:

```
@Value("${paginacao.qtd_por_pagina}")  
private int qtdPorPagina;
```

Dessa forma, o container IOC do Spring se encarregará de injetar e disponibilizar para nosso componente o valor informado, agora basta usá-lo em seu código.

13. Profiles

⇒ **Problema:**

Gostaria de ter arquivos de configurações com diferentes valores para determinadas ocasiões.

⇒ **Solução:**

O Spring Boot trabalha com o conceito de 'profiles', que permite criar perfis de execução.

Esses perfis podem estar associados a execução da aplicação em desenvolvimento, em teste, ou mesmo em produção, e para cada um desses perfis, é possível executar a aplicação com distintas configurações.

Os 'profiles' permitem associar um arquivo de configuração para cada um dos perfis, permitindo executar a aplicação com distintas configurações.

⇒ **Como fazer:**

Existem três formas principais para trabalhar com 'profiles' no Spring, via anotação, via linha de comando, ou via 'application.properties'.

Mas antes, devemos criar um novo arquivo properties para cada profile, que deverá seguir a seguinte nomenclatura:

```
application-NOME_DO_PROFILE.properties
```

Vamos então criar o profile 'test', para configurações de teste da aplicação, lembrando que você poderá quantos profiles desejar.

Crie o arquivo 'application-test.properties' no diretório de resources, juntamente com o 'application.properties'.

Adicione uma mesma entrada em cada arquivo, porém com valores diferentes.

Para habilitar o profile via anotação, adicione '@ActiveProfiles("test")' na classe desejada. Essa anotação é bastante útil para a criação de testes unitários.

Para executar a aplicação em um profile diferente, devemos fazer isso via linha de comando, adicionando o parâmetro '-Dspring.profiles.active=test', onde 'test' é o nome do profile, como por exemplo:

```
java -jar -Dspring.profiles.active=test meu-primeiro-projeto-0.0.1-SNAPSHOT.jar
```

A terceira forma consiste em adicionar o mesmo parâmetro acima diretamente no arquivo 'application.properties', conforme segue:

```
spring.profiles.active=test
```

Procure criar ao menos três profiles para suas aplicações, um para desenvolvimento, um para testes, e um para produção. O profile de produção deverá ser definido sempre como parâmetro por linha de comando, assim você não correrá o risco de acessar o ambiente de produção em desenvolvimento por exemplo.

14. Gerando senhas com o BCrypt

⇒ **Problema:**

Gostaria de armazenar as senhas dos usuários da aplicação de modo seguro no banco de dados.

⇒ **Solução:**

O BCrypt permite encriptar um determinado valor de forma irreversível, sendo o ideal para o armazenamento seguro de informações no banco de dados.

O que é mais interessante é que o BCrypt criar hashes diferentes para um mesmo valor se chamado mais de uma vez, o que torna sua encriptação altamente eficiente e segura.

⇒ **Como fazer:**

O BCrypt faz parte do módulo de segurança do Spring, então devemos adicionar sua dependência ao projeto, adicionando ao arquivo 'pom.xml':

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
</dependencies>
```

Salve o arquivo para o STS baixar as dependências.

Para utilizar o BCrypt, o recomendado é criar uma classe utilitária para gerenciar sua criação, para isso crie a classe 'SenhaUtils' com o seguinte conteúdo:

```
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;

public class SenhaUtils {
```

```

/**
 * Gera um hash utilizando o BCrypt.
 *
 * @param senha
 * @return String
 */
public static String gerarBCrypt(String senha) {
    if (senha == null) {
        return senha;
    }

    BCryptPasswordEncoder bCryptEncoder = new BCryptPasswordEncoder();
    return bCryptEncoder.encode(senha);
}

/**
 * Verifica se a senha é válida.
 *
 * @param senha
 * @param senhaEncoded
 * @return boolean
 */
public static boolean senhaValida(String senha, String senhaEncoded) {
    BCryptPasswordEncoder bCryptEncoder = new BCryptPasswordEncoder();
    return bCryptEncoder.matches(senha, senhaEncoded);
}
}

```

Com a classe criada, por ela conter métodos estáticos, basta chamá-los diretamente onde desejar em seu código, como por exemplo:

```
String senhaEncoded = SenhaUtils.gerarBCrypt("123456");
```

Ou para verificar uma senha:

```
boolean senhaValida = SenhaUtils.senhaValida("123456", senhaEncoded);
```

15. Criando Entidades JPA (Entity)

⇒ **Problema:**

Gostaria de mapear as tabelas da minha base de dados para serem acessadas em formato de objetos pelo Java.

⇒ **Solução:**

O Java possui o JPA (Java Persistence API), que é uma API de persistência para Java, que traduz um objeto Java para uma tabela no banco de dados, e vice versa.

O JPA é composto de uma série de anotações, que são responsáveis por mapear os atributos de uma classe Java em uma tabela de banco de dados.

Tais anotações são inclusive capazes de lidar com relacionamento entre tabelas, como o 'um para muitos', 'muitos para um'.

⇒ **Como fazer:**

Entidades JPA são classes normais Java contendo atributos e métodos.

Para criar uma entidade, crie uma classe Java e adicione a ela os atributos referente a seu conteúdo. Não deixe de adicionar os getters e setters para os atributos, e ao menos o construtor padrão.

O foco aqui não é explicar a fundo mapeamento JPA, e sim demonstrar como uma entidade é estruturada, para isso olhe o exemplo de uma entidade no código a seguir, lembrando que existe a dependência do Spring Data, que foi explicado no item 9:

```
import java.io.Serializable;
import java.util.Date;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.PrePersist;
```

```

import javax.persistence.PreUpdate;
import javax.persistence.Table;

@Entity
@Table(name = "empresa")
public class Empresa implements Serializable {

    private static final long serialVersionUID = 3960436649365666213L;

    private Long id;
    private String razaoSocial;
    private String cnpj;
    private Date dataCriacao;
    private Date dataAtualizacao;

    public Empresa() {
    }

    @Id
    @GeneratedValue(strategy=GenerationType.AUTO)
    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @Column(name = "razao_social", nullable = false)
    public String getRazaoSocial() {
        return razaoSocial;
    }

    public void setRazaoSocial(String razaoSocial) {
        this.razaoSocial = razaoSocial;
    }

    @Column(name = "cnpj", nullable = false)
    public String getCnpj() {
        return cnpj;
    }

    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }

    @Column(name = "data_criacao", nullable = false)
    public Date getDataCriacao() {

```

```

        return dataCriacao;
    }

    public void setDataCriacao(Date dataCriacao) {
        this.dataCriacao = dataCriacao;
    }

    @Column(name = "data_atualizacao", nullable = false)
    public Date getDataAtualizacao() {
        return dataAtualizacao;
    }

    public void setDataAtualizacao(Date dataAtualizacao) {
        this.dataAtualizacao = dataAtualizacao;
    }

    @PreUpdate
    public void preUpdate() {
        dataAtualizacao = new Date();
    }

    @PrePersist
    public void prePersist() {
        final Date atual = new Date();
        dataCriacao = atual;
        dataAtualizacao = atual;
    }

    @Override
    public String toString() {
        return "Empresa [id=" + id + ", razaoSocial=" +
            razaoSocial + ", cnpj=" + cnpj + ",
            dataCriacao=" + dataCriacao + ",
            dataAtualizacao=" + dataAtualizacao + "]";
    }
}

```

No código acima criamos uma entidade para representar uma empresa, e seguem as descrições das principais anotações utilizadas.

A '@Entity' informa ao JPA que essa classe se trata de uma entidade JPA.

A '@Table' é opcional, e permite definir o nome da tabela no banco de dados para a entidade.

A '@Id' informa que o campo será a chave primária da tabela.

A '@GeneratedValue' informa como a chave primária será incrementada, sendo que o modo automático apenas incrementará o valor em 1 a cada nova inserção.

A '@Column' permite definir um nome para o campo na tabela do banco de dados, assim como se ele pode ou não ser nulo.

As '@PrePersist' e '@PreUpdate' também são opcionais, e permitem executar uma ação antes de uma inserção ou atualização de um registro.

Existem outras anotações que serão abordadas na segunda parte, na criação de uma API na prática.

16. Criando Repositórios JPA (JpaRepository)

⇒ **Problema:**

Gostaria de executar de modo simples operações no Java para manipulação da minha base de dados, como inserir registros, atualizar, remover, listar dados.

⇒ **Solução:**

O Spring Data possui o 'JpaRepository', que permite executar as principais operações de acesso a uma base de dados de modo automático.

Essa interface somente precisa ser configurada para saber a qual entidade JPA ela se refere, assim a aplicação já terá disponível a implementação para os principais acessos a base de dados.

⇒ **Como fazer:**

Crie uma nova interface que estenda a interface 'JpaRepository', com isso você já terá acesso a diversas funcionalidades e acessos a tabela na base de dados, lembrando que existe a dependência do Spring Data, que foi explicado no item 9

É possível também criar métodos de acessos personalizados baseados em uma convenção do próprio Spring para a criação de métodos, conforme demonstrado no exemplo abaixo:

```
import org.springframework.data.jpa.repository.JpaRepository;

import com.kazale.api.entities.Empresa;

public interface EmpresaRepository extends JpaRepository<Empresa, Long> {

    Empresa findByCnpj(String cnpj);

}
```

O método 'findByCnpj' realiza uma busca por CNPJ diretamente na tabela, utilizando a convenção 'findBy' do Spring.

17. Versionamento de banco de dados com o Flyway

⇒ **Problema:**

Gostaria de criar as tabelas do meu banco de dados de modo automático, facilitando inclusive a criação do banco do mesmo modo para todos os membros da minha equipe de desenvolvimento.

⇒ **Solução:**

O Flyway é um framework que permite o versionamento e automatização no processo de criação de banco de dados.

Nele é possível configurar a criação de tabelas, dados iniciais que devem estar na base de dados, entre outros.

Ele possui um utilitário de linha de comando, que permite criar, atualizar, ou mesmo limpar uma base de dados, tornando o gerenciamento da base de dados simples e intuitiva.

⇒ **Como fazer:**

Vamos configurar o Flyway para trabalhar com o Spring Boot, então não entraremos em detalhes de como executá-lo via linha de comando.

Primeiramente adicione a dependência do Flyway no arquivo 'pom.xml':

```
<dependencies>
  <dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
  </dependency>
</dependencies>
```

No exemplo utilizaremos o banco de dados MySQL, então certifique de ter ele configurado para utilização, conforme demonstrado em tópicos anteriores.

Vamos configurar a conexão com o MySQL adicionando os seguintes valores ao 'application.properties':

```
spring.jpa.hibernate.ddl-auto=none
spring.datasource.url=jdbc:mysql://localhost:8889/flyway
spring.datasource.username=root
spring.datasource.password=root
```

É importante manter o 'ddl-auto' para 'none', para evitar que o Hibernate crie as tabelas automaticamente, uma vez que iremos delegar essa tarefa para o Flyway.

O Flyway possui algumas convenções de nomes e diretórios, então vamos criar a versão inicial de um banco de dados que contém uma tabela de empresas.

Crie a seguinte estrutura de diretórios na aplicação:

```
src/main/resources/db/migration/mysql
```

Crie o arquivo 'V1__tabelas.sql' dentro do diretório recém criado, e adicione o seguinte conteúdo nele.

```
CREATE TABLE `empresa` (
  `id` bigint(20) NOT NULL,
  `cnpj` varchar(255) NOT NULL,
  `data_atualizacao` datetime NOT NULL,
  `data_criacao` datetime NOT NULL,
  `razao_social` varchar(255) NOT NULL
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

ALTER TABLE `empresa`
ADD PRIMARY KEY (`id`);

ALTER TABLE `empresa`
MODIFY `id` bigint(20) NOT NULL AUTO_INCREMENT;
```

```
INSERT INTO `empresa` (`id`, `cnpj`, `data_atualizacao`, `data_criacao`, `razao_social`)  
VALUES (NULL, '82198127000121', CURRENT_DATE(), CURRENT_DATE(), 'Empresa  
ADMIN');
```

O código acima cria a tabela no banco, e adiciona uma empresa de teste.

Agora basta executar a aplicação para que o Spring Boot execute o Flyway automaticamente e gere nossa tabela na base de dados.

18. Configurando um pool de conexões de banco de dados com Hikari

⇒ **Problema:**

Gostaria de configurar um pool de conexões eficiente e personalizado para gerenciar a conexão da minha aplicação com a base de dados.

⇒ **Solução:**

Um dos gerenciadores de conexão mais utilizados, e com performance excelente é o Hikari, que pode ser adicionado a uma aplicação Java, ou pode ser configurado de modo bastante simples se adicionado a projetos Spring Boot, que já possui suporte para ele.

Um bom gerenciador de pool de conexões é fundamental para a performance da aplicação, uma vez que o banco de dados é um recurso que pode impactar drasticamente a execução de qualquer sistema.

⇒ **Como fazer:**

O Spring Boot possui suporte ao Hikari, então basta adicionar sua dependência ao arquivo 'pom.xml' e salvar o arquivo para que as dependências sejam instaladas, conforme código a seguir:

```
<dependencies>
  <dependency>
    <groupId>com.zaxxer</groupId>
    <artifactId>HikariCP</artifactId>
    <version>2.6.0</version>
  </dependency>
</dependencies>
```

Feito isso, você poderá personalizar o Hikari diretamente no arquivo 'application.properties', seguindo a convenção de configurações em 'spring.datasource.hikari'.

Um exemplo seria configurar o tamanho do pool de conexões, conforme exemplo a seguir:

```
spring.datasource.hikari.maximum-pool-size=25
```

Agora basta executar a aplicação para usar o novo pool de conexões.

19. Criando serviços (Service)

⇒ **Problema:**

Gostaria de manter a lógica da minha aplicação isolada e organizada em um só local, para que ela seja de fácil manutenção, e possa ser facilmente acessada por outros componentes.

⇒ **Solução:**

O Spring possui uma anotação chamada 'Service', que quando uma classe Java é anotada com ela, a mesma passará a ser um componente Spring.

Esse componente Spring, deverá conter uma lógica de negócio específica, e poderá ser injetada como dependência de qualquer outro componente usando a anotação 'Autowired'.

⇒ **Como fazer:**

Crie uma classe Java e adicione a anotação '@Service' a ela, conforme exemplo a seguir:

```
import org.springframework.stereotype.Service;

@Service
public class ExemploService {

    public void testarServico() {
        System.out.println("### Executando serviço de teste!!!");
    }

}
```

Com a classe criada, basta em qualquer componente ou serviço adicionar a anotação '@Autowired' para injetar um serviço, conforme o exemplo a seguir:

```
@Autowired
private ExemploService exemploService;
```

Pronto, agora basta chamar os métodos contidos no serviços.

20. Criando controllers (Controller)

⇒ **Problema:**

Gostaria de expor meus componentes Spring como sendo serviços Restful.

⇒ **Solução:**

O Spring Rest possui a anotação 'Controller', que uma vez adicionada a uma classe Java, aceitará um 'path' como parâmetro, tornando esse componente disponível para acesso HTTP para o 'path' adicionado.

Com os controllers, também é possível gerenciar os verbos HTTP (GET, POST, PUT, DELETE,...) para cada método da classe, permitindo criar todos os acessos Restful para a sua API.

⇒ **Como fazer:**

Para expor uma API com o Spring Boot, a primeira coisa a fazer é adicionar a dependência do Spring Boot Web, para isso adicione ao arquivo 'pom.xml':

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-web</artifactId>
</dependency>
```

Salve o arquivo para instalar as dependências, que incluem o Tomcat, Jackson, entre outras.

Depois, crie uma classe Java com o seguinte código:

```
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
```

```
@RestController
@RequestMapping("/api/exemplo")
public class ExemploController {

    @GetMapping(value =("/{nome}")
    public String exemplo(@PathVariable("nome") String nome) {
        return "Olá " + nome;
    }
}
```

No código acima, '@RestController' será o responsável por criar a API Rest, seguido do '@RequestMapping', que indicará o path do serviço.

Após isso, basta mapear os métodos dos controllers com a anotação '@GetMapping', seguido de um valor opcional como parâmetro.

A '@GetMapping' se refere a requisições HTTP GET, para outras como POST, PUT, DELETE, basta mudar a anotação para o formato desejado, como '@PostMapping', '@PutMapping', '@DeleteMapping', respectivamente.

A '@PathVariable' serve para obter um valor passado na URL.

Seguindo o mapeamento do exemplo acima, basta executar a aplicação e acessar a seguinte URL para testar o controller:

<http://localhost:8080/api/exemplo/NOME>

21. Utilizando o Postman

⇒ **Problema:**

Gostaria de ter uma interface gráfica para testar meus serviços Restful.

⇒ **Solução:**

O Postman é uma extensão para o navegador Google Chrome que permite fazer qualquer tipo de requisição Restful facilmente.

Ele é gratuito e fácil de instalar e utilizar, sendo uma ferramenta indispensável para o desenvolvimento de aplicações Restful.

⇒ **Como fazer:**

O Postman requer o navegador Google Chrome instalado, então certifique de ter ele em seu computador.

Após isso, acesse <https://www.getpostman.com>, faça o download da versão referente ao seu sistema operacional e execute o instalador.

Pronto, agora basta clicar no executável para ter acesso a sua interface, que é bastante intuitiva.

Basicamente selecione o verbo HTTP (GET, POST, ...), digite a URL e clique em 'send' para executar a requisição.

É possível também executar operações mais avançadas, como definir headers, parâmetros, entre outros.

22. DTO (Data Transfer Object)

⇒ **Problema:**

Gostaria de trafegar os dados das minhas requisições HTTP de modo padronizado.

⇒ **Solução:**

Requisições Restful exigem na maioria dos casos o envio de parâmetros, sejam eles de formulários, configuração.

Para que esses dados sejam facilmente manipulados e gerenciados pelo Spring, é recomendada a utilização do padrão de projetos DTO (Data Transfer Object), por permitir que os dados utilizados em uma requisição, sejam facilmente convertidos para uma classe Java.

Sua grande vantagem é permitir a fácil manipulação dos dados da requisição HTTP, e os DTOs consistem apenas de classes Java com atributos, que representam os parâmetros das requisições.

⇒ **Como fazer:**

Um DTO nada mais é que uma classe Java contendo atributos e seus assessores getters e setters.

A seguir está o código de uma classe DTO:

```
public class EmpresaDto {  
  
    private Long id;  
    private String razaoSocial;  
    private String cnpj;  
  
    public EmpresaDto() {  
    }  
}
```

```

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getRazaoSocial() {
        return razaoSocial;
    }

    public void setRazaoSocial(String razaoSocial) {
        this.razaoSocial = razaoSocial;
    }

    public String getCnpj() {
        return cnpj;
    }

    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }

    @Override
    public String toString() {
        return "EmpresaDto [id=" + id + ", razaoSocial=" + razaoSocial +
            ", cnpj=" + cnpj + "];"
    }
}

```

A classe acima representa um DTO de uma empresa, agora a seguir segue um controller que faz uso desse DTO:

```
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.kazale.api.dto.EmpresaDto;

@RestController
@RequestMapping("/api/empresas")
public class EmpresaController {

    @PostMapping
    public ResponseEntity<EmpresaDto> cadastrar(
        @RequestBody EmpresaDto empresaDto) {
        empresaDto.setId(1L);
        return ResponseEntity.ok(empresaDto);
    }
}
```

O controller acima é como o criado no tópico 20, e basicamente o que ele faz é obter os dados de um formulário enviado via requisição POST, definindo um ID, e depois retornando o mesmo DTO para o cliente.

Em um cenário real, o DTO seria persistido em banco de dados, mas para simplificar ele é apenas retornado.

A '@RequestBody' é uma classe do Spring responsável por converter os dados automaticamente de uma requisição HTTP em formato Java para o DTO.

O 'ResponseEntity' é uma classe utilitária também do Spring que nos permite gerenciar as requisições HTTP, por isso é utilizada para retornar o DTO para o cliente.

23. Encapsulando o retorno de uma chamada a API Restful (Response)

⇒ **Problema:**

Gostaria que todos os retornos de minha API Restful fossem padronizadas e tivessem a mesma estrutura.

⇒ **Solução:**

Para padronizar o retorno das requisições de uma API Restful, o indicado é a criação de uma classe responsável por encapsular os dados de retorno de um modo estruturado.

Por isso devemos criar uma classe 'Response', que conterá uma estrutura mínima para manipular os casos de sucesso ou erro em uma requisição, mantendo assim toda a estrutura da API padronizada.

⇒ **Como fazer:**

Primeiramente vamos criar uma classe para encapsular todas as requisições HTTP de nossa API.

```
import java.util.ArrayList;
import java.util.List;

public class Response<T> {

    private T data;
    private List<String> errors;

    public Response() {
    }

    public T getData() {
        return data;
    }
}
```

```

    public void setData(T data) {
        this.data = data;
    }

    public List<String> getErrors() {
        if (this.errors == null) {
            this.errors = new ArrayList<String>();
        }
        return errors;
    }

    public void setErrors(List<String> errors) {
        this.errors = errors;
    }
}

```

Na classe acima foram definidos basicamente dois atributos, 'data' para conter os dados em caso de sucesso, e 'errors' para armazenar mensagens de erros do sistemas.

Nesse caso, todas as requisições seguirão um mesmo padrão, verifique no código a seguir como ficaria o controller:

```

import javax.validation.Valid;

import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.kazale.api.dto.EmpresaDto;
import com.kazale.api.response.Response;

```

```
@RestController
@RequestMapping("/api/empresas")
public class EmpresaController {

    @PostMapping
    public ResponseEntity<Response<EmpresaDto>> cadastrar(
        @Valid @RequestBody EmpresaDto empresaDto,
        BindingResult result) {
        Response<EmpresaDto> response = new Response<EmpresaDto>();

        if (result.hasErrors()) {
            result.getAllErrors().forEach(error ->
                response.getErrors().add(error.getDefaultMessage()));
            return ResponseEntity.badRequest().body(response);
        }

        empresaDto.setId(1L);
        response.setData(empresaDto);
        return ResponseEntity.ok(response);
    }
}
```

Para detalhes da implementação do controller consulte os tópicos 22 e 24.

24. Validação de dados (Bean Validation)

⇒ **Problema:**

Gostaria de validar os dados de entrada em uma requisição HTTP de forma prática e automática.

⇒ **Solução:**

A biblioteca Hibernate Validator, em conjunto com o JPA fornecem uma solução completa para validação baseada em anotações, onde basta anotarmos os atributos de nossas entidades ou DTOs conforme necessário.

Tais anotações serão executadas assim que a API for chamada, e caso algum dos valores não atendam as regras impostas pelos validadores, um erro será lançado como retorno informando a causa do problema.

⇒ **Como fazer:**

Utilizando DTOs, é neles que iremos adicionar nossas regras de validação, segue um código de exemplo:

```
import org.hibernate.validator.constraints.Length;
import org.hibernate.validator.constraints.NotEmpty;
import org.hibernate.validator.constraints.br.CNPJ;

public class EmpresaDto {

    private Long id;
    private String razaoSocial;
    private String cnpj;

    public EmpresaDto() {
    }

    public Long getId() {
```

```

        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    @NotEmpty(message = "Razão social não pode ser vazia.")
    @Length(min = 5, max = 200,
        message = "Razão social deve conter entre 5 e 200 caracteres.")
    public String getRazaoSocial() {
        return razaoSocial;
    }

    public void setRazaoSocial(String razaoSocial) {
        this.razaoSocial = razaoSocial;
    }

    @NotEmpty(message = "CNPJ não pode ser vazio.")
    @CNPJ(message="CNPJ inválido.")
    public String getCnpj() {
        return cnpj;
    }

    public void setCnpj(String cnpj) {
        this.cnpj = cnpj;
    }

    @Override
    public String toString() {
        return "EmpresaDto [id=" + id + ", razaoSocial=" + razaoSocial +
            ", cnpj=" + cnpj + "]";
    }
}

```

Repare nas anotações '@NotEmpty', '@CNPJ', e '@Length', elas são as responsáveis por informar as regras de validação aplicadas para cada campo do DTO, e como suas descrições sugerem, elas validam um valor não nulo, CNPJ válido, e tamanho do texto, respectivamente.

Para que a validação seja executada, devemos adicionar em nosso controller a anotação '@Valid', e também fazer uso do 'BindingResult', que será responsável por conter o retorno da validação.

Veja a seguir como ficaria o controller validando a entrada de dados com um DTO:

```
import javax.validation.Valid;

import org.springframework.http.ResponseEntity;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.kazale.api.dto.EmpresaDto;
import com.kazale.api.response.Response;

@RestController
@RequestMapping("/api/empresas")
public class EmpresaController {

    @PostMapping
    public ResponseEntity<Response<EmpresaDto>> cadastrar(
        @Valid @RequestBody EmpresaDto empresaDto,
        BindingResult result) {
        Response<EmpresaDto> response = new Response<EmpresaDto>();

        if (result.hasErrors()) {
            result.getAllErrors().forEach(error ->
```

```
        response.getErrors().add(error.getDefaultMessage());  
        return ResponseEntity.badRequest().body(response);  
    }  
  
    empresaDto.setId(1L);  
    response.setData(empresaDto);  
    return ResponseEntity.ok(response);  
}  
}
```

Repare como o 'ResponseEntity' é utilizado para controlar o retorno das requisições através das chamadas a seus métodos 'badRequest' e 'ok', que retornam erro (código HTTP 400) ou sucesso (código HTTP 200) para cada tipo de requisição.

25. Documentando API com Swagger

⇒ **Problema:**

Gostaria de tornar mais simples a visualização dos serviços Restful da minha aplicação, de forma visual, para facilitar os testes, e também poder usá-la como documentação para ser exibida para terceiros.

⇒ **Solução:**

O Swagger é a solução ideal para documentar e criar ambientes para testes de uma API Restful.

Ele pode ser facilmente integrado com o Spring Boot, e de modo automático extrairá todas as informações da API do código fonte.

Em conjunto com o Swagger UI, uma interface funcional também será disponibilizada para a API.

⇒ **Como fazer:**

A integração do Swagger com o Spring Boot é bastante simples, e consiste primeiramente na adição de suas dependências no arquivo 'pom.xml', conforme segue:

```
<dependencies>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger2</artifactId>
    <version>2.6.1</version>
  </dependency>
  <dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.6.1</version>
  </dependency>
</dependencies>
```

A seguir, é preciso criar um arquivo de configuração para habilitar e instruir o Swagger a gerar a documentação para a API, conforme o código a seguir:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.service.ApiInfo;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;

@Configuration
@EnableSwagger2
public class SwaggerConfig {

    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2).select()
            .apis(RequestHandlerSelectors.basePackage("com.kazale.api"))
            .paths(PathSelectors.any()).build()
            .apiInfo(apiInfo());
    }

    private ApiInfo apiInfo() {
        return new ApiInfoBuilder().title("Swagger API")
            .description(
                "Documentação da API de acesso aos endpoints com Swagger")
            .version("1.0")
            .build();
    }
}
```

O Swagger usa o 'basePackage' para buscar por controllers, então altere ele para o local onde seus controllers estão implementados.

Com isso, basta executar a aplicação e acessar a seguinte URL:

<http://localhost:8080/swagger-ui.html>

Você deverá ver uma similar a seguinte:

The screenshot displays the Swagger UI interface. At the top, there is a green header bar with the Swagger logo on the left, a dropdown menu showing 'default (/v2/api-docs)' in the center, and an 'Explore' button on the right. Below the header, the main content area is titled 'Swagger API' with a subtitle 'Documentação da API de acesso aos endpoints com Swagger'. The primary section is for the 'empresa-controller : Empresa Controller', which includes links for 'Show/Hide', 'List Operations', and 'Expand Operations'. The selected operation is a POST request to '/api/empresas', labeled 'cadastrar'. The response is categorized as 'Response Class (Status 200)' with the status 'OK'. Under the 'Model' tab, an 'Example Value' is shown as a JSON object:

```
{  "data": {    "cnpj": "string",    "id": 0,    "razaoSocial": "string"  },  "errors": [    "string"  ]}
```

 At the bottom of the response area, there is a 'Response Content Type' dropdown menu currently set to '*/*'.

26. Versionamento de API

⇒ **Problema:**

Gostaria de versionar minha API Restful, assim eu poderia manter um maior controle sobre as alterações realizadas no meu código.

⇒ **Solução:**

O versionamento de APIs pode ser realizado de algumas formas distintas, e aqui apresentaremos duas das mais utilizadas.

A primeira delas consiste em adicionar a versão diretamente na URL requisitada, como por exemplo 'http://api.com/v1/listagem', 'http://api.com/v2/listagem'.

Essa é a forma mais fácil de gerenciar versões, pois somente exige modificação na URL, o que é algo simples de ser feito tanto no lado do cliente quanto na aplicação.

A segunda abordagem é adicionando um novo header na requisição HTTP, solicitando a ação para uma determinada versão da API.

Nesse caso, o header 'X-API-VERSION' é adicionado solicitando a versão a ser utilizada, o que torna a implementação no lado do cliente um pouco mais complexa do que apenas modificar a URL.

Não existe certo ou errado em nenhuma abordagem, e utilize a que melhor se encaixar a sua necessidade.

⇒ **Como fazer:**

O código a seguir implementa as duas abordagens de versionamento de APIs. Os dois primeiros métodos são referentes ao versionamento por URL, e os dois últimos são baseados em um parâmetro passado pelo header.

Verifique a seguir o código:

```

import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api")
public class VersionamentoApiController {

    /**
     * Versionamento de API pela url, define versão 'v1'.
     *
     * @param nome
     * @return ResponseEntity<Response<String>>
     */
    @GetMapping(value = "/v1/ola/{nome}")
    public ResponseEntity<String> olaNomeV1(
        @PathVariable("nome") String nome) {
        return ResponseEntity.ok(String.format("API v1: Olá %s!", nome));
    }

    /**
     * Versionamento de API pela url, define versão 'v2'.
     *
     * @param nome
     * @return ResponseEntity<Response<String>>
     */
    @GetMapping(value = "/v2/ola/{nome}")
    public ResponseEntity<String> olaNomeV2(
        @PathVariable("nome") String nome) {
        return ResponseEntity.ok(String.format("API v2: Olá %s!", nome));
    }

    /**

```

```

* Versionamento de API pelo Header 'X-API-Version', define versão 'v1'.
*
* @param nome
* @return ResponseEntity<Response<String>>
*/
@GetMapping(value = "/ola/{nome}", headers = "X-API-Version=v1")
public ResponseEntity<String> olaNomeHeaderV1(
    @PathVariable("nome") String nome) {
    return ResponseEntity.ok(String.format("API Header v1: Olá %s!", nome));
}

/**
* Versionamento de API pelo Header 'X-API-Version', define versão 'v2'.
*
* @param nome
* @return ResponseEntity<Response<String>>
*/
@GetMapping(value = "/ola/{nome}", headers = "X-API-Version=v2")
public ResponseEntity<String> olaNomeHeaderV2(
    @PathVariable("nome") String nome) {
    return ResponseEntity.ok(String.format("API Header v2: Olá %s!", nome));
}
}

```

O versionamento por URL é muito simples, basta adicionar a versão no mapeamento da anotação '@GetMapping'.

Já o versionamento por parâmetro do header implica na adição da referência da versão também na anotação '@GetMapping', mas em seu parâmetro 'header'.

27. Autenticação e autorização com tokens JWT (Json Web Token)

⇒ **Problema:**

Gostaria de controlar o acesso a minha aplicação, tanto no login, quanto nas ações a serem acessadas por determinados usuários dentro da aplicação.

Gostaria também que a autenticação fosse stateless, ou seja, que não dependesse de sessão, e que utilizasse tokens JWT (JSON Web Token) para o armazenamento das credenciais do usuário.

⇒ **Solução:**

APIs Restful eficientes não devem manter estado, e devem permitir que sejam escaláveis horizontalmente, para que assim sejam de alta performance.

Por ela não manter sessão em nenhum local, os dados de acesso devem estar armazenados em algum lugar que possa ser compartilhado entre requisições.

Para isso que existe o JWT, que é um formato de token seguro e assinado digitalmente, garantindo a integridade dos dados trafegados. Dessa forma manteremos as informações de autenticação no token, para que assim a aplicação seja capaz de validar o acesso a uma requisição.

As principais informações a serem armazenadas no token são dados do usuário, perfil e data de expiração do token.

⇒ **Como fazer:**

A parte de implementação da autenticação e autorização utilizando tokens JWT não vem implementada por padrão no Spring, então ela demanda uma série de códigos e implementação de interfaces do Spring Security.

Antes de tudo, adicione a dependência do Spring Security e JWT ao projeto, incluindo a seguinte entrada ao 'pom.xml':

```

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
  </dependency>
  <dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt</artifactId>
    <version>0.7.0</version>
  </dependency>
  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
</dependencies>

```

Salve o arquivo para que o STS faça o download das dependências.

Abaixo segue a implementação necessária para autenticar e autorizar uma aplicação Spring Boot utilizando tokens JWT.

Todo o código deverá ser executado passo a passo para que tudo funcione perfeitamente.

Esse código depende da classe de response estudada no tópico 23, e da 'SenhaUtils' estudada no tópico 14, portanto certifique-se de que elas estejam presente em seu projeto.

Vamos começar criando uma entidade 'Usuario' e um 'JpaRepository' para ele, pois essa entidade armazenará os dados dos usuários da aplicação. Também criaremos uma enum para manter os perfis de acesso, que serão dois, usuário comum e administrador (você poderá adaptar os perfis a suas necessidades depois).

Arquivo **PerfilEnum.java**:

```
package com.kazale.api.security.enums;

public enum PerfilEnum {
    ROLE_ADMIN,
    ROLE_USUARIO;
}
```

Sempre inicie o nome do perfil com o prefixo 'ROLE_', é uma convenção do Spring Security.

Arquivo **Usuario.java**:

```
package com.kazale.api.security.entities;

import java.io.Serializable;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.EnumType;
import javax.persistence.Enumerated;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.Table;

import com.kazale.api.security.enums.PerfilEnum;

@Entity
@Table(name = "usuario")
public class Usuario implements Serializable {

    private static final long serialVersionUID = 306411570471828345L;

    private Long id;
    private String email;
    private String senha;
```

```
private PerfilEnum perfil;

public Usuario() {
}

@Id
@GeneratedValue(strategy=GenerationType.AUTO)
public Long getId() {
    return id;
}

public void setId(Long id) {
    this.id = id;
}

@Column(name = "email", nullable = false)
public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

@Enumerated(EnumType.STRING)
@Column(name = "perfil", nullable = false)
public PerfilEnum getPerfil() {
    return perfil;
}

public void setPerfil(PerfilEnum perfil) {
    this.perfil = perfil;
}

@Column(name = "senha", nullable = false)
```

```

    public String getSenha() {
        return senha;
    }

    public void setSenha(String senha) {
        this.senha = senha;
    }
}

```

Entidade JPA conforme criada em tópicos anteriores.

Arquivo **UsuarioRepository.java**:

```

package com.kazale.api.security.repositories;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.transaction.annotation.Transactional;

import com.kazale.api.security.entities.Usuario;

@Transactional(readOnly = true)
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

    Usuario findByEmail(String email);
}

```

Repositório criado conforme tópicos anteriores, por ser somente leitura, utiliza a anotação '@Transactional' definida para tal, assim o acesso é mais rápido, uma vez que não existirá a necessidade de nenhum tipo de lock no banco.

Os códigos acima não apresentam nada do que foi estudado em outros tópicos, ele apenas faz a interface com o banco de dados, e em caso de dúvida consulte os tópicos relacionados a persistência.

Agora será criado um serviço que será responsável por chamar o repositório recém criado, para isso crie os seguintes arquivos:

Interface **UsuarioService.java**:

```
package com.kazale.api.security.services;

import java.util.Optional;

import com.kazale.api.security.entities.Usuario;

public interface UsuarioService {

    /**
     * Busca e retorna um usuário dado um email.
     *
     * @param email
     * @return Optional<Usuario>
     */
    Optional<Usuario> buscarPorEmail(String email);
}
```

E sua implementação, classe **UsuarioServiceImpl.java**:

```
package com.kazale.api.security.services.impl;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.kazale.api.security.entities.Usuario;
import com.kazale.api.security.repositories.UsuarioRepository;
import com.kazale.api.security.services.UsuarioService;
```

```

@Service
public class UsuarioServiceImpl implements UsuarioService {

    @Autowired
    private UsuarioRepository usuarioRepository;

    public Optional<Usuario> buscarPorEmail(String email) {
        return Optional.ofNullable(this.usuarioRepository.findByEmail(email));
    }
}

```

Esse serviço será utilizado para carregar os dados do usuário, e será utilizado no processo de autenticação e autorização.

Adicione dois parâmetros ao arquivo **application.properties**. Esses parâmetros controlarão a chave de assinatura do token, assim como seu tempo de expiração:

```

# JWT
jwt.secret=_@HRL&L3tF?Z7ccj4z&L5!nU2B!Rjs3_
# token com duração de 7 dias
jwt.expiration=604800

```

Fique a vontade para adaptar esses valores conforme sua necessidade.

Agora crie um arquivo utilitário para manipular o token JWT, criando o arquivo **JwtTokenUtil.java**:

```

package com.kazale.api.security.utils;

import java.util.Date;
import java.util.HashMap;
import java.util.Map;

import org.springframework.beans.factory.annotation.Value;
import org.springframework.security.core.userdetails.UserDetails;

```

```

import org.springframework.stereotype.Component;

import io.jsonwebtoken.Claims;
import io.jsonwebtoken.Jwts;
import io.jsonwebtoken.SignatureAlgorithm;

@Component
public class JwtTokenUtil {

    static final String CLAIM_KEY_USERNAME = "sub";
    static final String CLAIM_KEY_ROLE = "role";
    static final String CLAIM_KEY_CREATED = "created";

    @Value("${jwt.secret}")
    private String secret;

    @Value("${jwt.expiration}")
    private Long expiration;

    /**
     * Obtém o username (email) contido no token JWT.
     *
     * @param token
     * @return String
     */
    public String getUsernameFromToken(String token) {
        String username;
        try {
            Claims claims = getClaimsFromToken(token);
            username = claims.getSubject();
        } catch (Exception e) {
            username = null;
        }
        return username;
    }
}

```

```

/**
 * Retorna a data de expiração de um token JWT.
 *
 * @param token
 * @return Date
 */
public Date getExpirationDateFromToken(String token) {
    Date expiration;
    try {
        Claims claims = getClaimsFromToken(token);
        expiration = claims.getExpiration();
    } catch (Exception e) {
        expiration = null;
    }
    return expiration;
}

/**
 * Cria um novo token (refresh).
 *
 * @param token
 * @return String
 */
public String refreshToken(String token) {
    String refreshedToken;
    try {
        Claims claims = getClaimsFromToken(token);
        claims.put(CLAIM_KEY_CREATED, new Date());
        refreshedToken = gerarToken(claims);
    } catch (Exception e) {
        refreshedToken = null;
    }
    return refreshedToken;
}

```

```

/**
 * Verifica e retorna se um token JWT é válido.
 *
 * @param token
 * @return boolean
 */
public boolean tokenValido(String token) {
    return !tokenExpirado(token);
}

/**
 * Retorna um novo token JWT com base nos dados do usuários.
 *
 * @param userDetails
 * @return String
 */
public String obterToken(UserDetails userDetails) {
    Map<String, Object> claims = new HashMap<>();
    claims.put(CLAIM_KEY_USERNAME, userDetails.getUsername());
    userDetails.getAuthorities().forEach(
        authority -> claims.put(CLAIM_KEY_ROLE, authority.getAuthority());
    );
    claims.put(CLAIM_KEY_CREATED, new Date());

    return gerarToken(claims);
}

/**
 * Realiza o parse do token JWT para extrair as informações contidas no
 * corpo dele.
 *
 * @param token
 * @return Claims
 */
private Claims getClaimsFromToken(String token) {

```

```

        Claims claims;

        try {
            claims = Jwts.parser().setSigningKey(secret)
                .parseClaimsJws(token).getBody();
        } catch (Exception e) {
            claims = null;
        }

        return claims;
    }

    /**
     * Retorna a data de expiração com base na data atual.
     *
     * @return Date
     */
    private Date gerarDataExpiracao() {
        return new Date(System.currentTimeMillis() + expiration * 1000);
    }

    /**
     * Verifica se um token JWT está expirado.
     *
     * @param token
     * @return boolean
     */
    private boolean tokenExpirado(String token) {
        Date dataExpiracao = this.getExpirationDateFromToken(token);
        if (dataExpiracao == null) {
            return false;
        }
        return dataExpiracao.before(new Date());
    }

    /**
     * Gera um novo token JWT contendo os dados (claims) fornecidos.

```

```

    *
    * @param claims
    * @return String
    */
    private String gerarToken(Map<String, Object> claims) {
        return Jwts.builder().setClaims(claims).setExpiration(gerarDataExpiracao())
            .signWith(SignatureAlgorithm.HS512, secret).compact();
    }
}

```

O Spring Security depende de um usuário que implemente a interface `UserDetails`, pois é através desses dados que ele controlará quem está autenticado no sistema.

Crie então a classe **JwtUser.java**:

```

package com.kazale.api.security;

import java.util.Collection;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;

public class JwtUser implements UserDetails {

    private static final long serialVersionUID = -268046329085485932L;

    private Long id;
    private String username;
    private String password;
    private Collection<? extends GrantedAuthority> authorities;

    public JwtUser(Long id, String username, String password,
        Collection<? extends GrantedAuthority> authorities) {
        this.id = id;
        this.username = username;
    }
}

```

```
        this.password = password;
        this.authorities = authorities;
    }

    public Long getId() {
        return id;
    }

    @Override
    public String getUsername() {
        return username;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true;
    }

    @Override
    public boolean isAccountNonLocked() {
        return true;
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true;
    }

    @Override
    public String getPassword() {
        return password;
    }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
```

```

        return authorities;
    }

    @Override
    public boolean isEnabled() {
        return true;
    }
}

```

É preciso agora criar uma classe 'factory' para converter nosso usuário no usuário reconhecido pelo Spring Security, então crie a classe **JwtUserFactory.java**:

```

package com.kazale.api.security;

import java.util.ArrayList;
import java.util.List;

import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.authority.SimpleGrantedAuthority;

import com.kazale.api.security.entities.Usuario;
import com.kazale.api.security.enums.PerfilEnum;

public class JwtUserFactory {

    private JwtUserFactory() {

    }

    /**
     * Converte e gera um JwtUser com base nos dados de um funcionário.
     *
     * @param funcionario
     * @return JwtUser
     */
}

```

```

public static JwtUser create(Usuario usuario) {
    return new JwtUser(usuario.getId(), usuario.getEmail(),
        usuario.getSenha(),
        mapToGrantedAuthorities(usuario.getPerfil()));
}

/**
 * Converte o perfil do usuário para o formato utilizado pelo Spring Security.
 *
 * @param perfilEnum
 * @return List<GrantedAuthority>
 */
private static List<GrantedAuthority> mapToGrantedAuthorities(
    PerfilEnum perfilEnum) {
    List<GrantedAuthority> authorities = new ArrayList<GrantedAuthority>();
    authorities.add(new SimpleGrantedAuthority(perfilEnum.toString()));
    return authorities;
}
}

```

Teremos também que criar um serviço para manipular essa interface do UserDetails, para isso crie a classe **JwtUserDetailsServiceImpl.java**:

```

package com.kazale.api.security.services.impl;

import java.util.Optional;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.core.userdetails.UsernameNotFoundException;
import org.springframework.stereotype.Service;

import com.kazale.api.security.JwtUserFactory;
import com.kazale.api.security.entities.Usuario;

```

```

import com.kazale.api.security.services.UsuarioService;

@Service
public class JwtUserDetailsService implements UserDetailsService {

    @Autowired
    private UsuarioService usuarioService;

    @Override
    public UserDetails loadUserByUsername(String username)
        throws UsernameNotFoundException {
        Optional<Usuario> funcionario =
            usuarioService.buscarPorEmail(username);

        if (funcionario.isPresent()) {
            return JwtUserFactory.create(funcionario.get());
        }

        throw new UsernameNotFoundException("Email não encontrado.");
    }
}

```

Criaremos agora uma classe que será responsável por tratar um erro de autenticação, portanto crie a classe **JwtAuthenticationEntryPoint.java**:

```

package com.kazale.api.security;

import java.io.IOException;

import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

```

```

@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryPoint {

    @Override
    public void commence(HttpServletRequest request,
                        HttpServletResponse response,
                        AuthenticationException authException) throws IOException {
        response.sendError(HttpServletResponse.SC_UNAUTHORIZED,
            "Acesso negado. Você deve estar autenticado no sistema" +
            " para acessar a URL solicitada.");
    }
}

```

Antes de habilitar e configurar o Spring Security na aplicação, será preciso adicionar um filtro para verificar o acesso a cada requisição, ou seja, se existe no header do HTTP um token válido, autorizando o acesso. Crie a classe **JwtAuthenticationTokenFilter.java**:

```

package com.kazale.api.security.filters;

import java.io.IOException;

import javax.servlet.FilterChain;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.springframework.beans.factory.annotation.Autowired;
import
org.springframework.security.authentication.UsernamePasswordAuthenticationToken
;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import

```

```

org.springframework.security.web.authentication.WebAuthenticationDetailsSource;
import org.springframework.web.filter.OncePerRequestFilter;

import com.kazale.api.security.utils.JwtTokenUtil;

public class JwtAuthenticationTokenFilter extends OncePerRequestFilter {

    private static final String AUTH_HEADER = "Authorization";
    private static final String BEARER_PREFIX = "Bearer ";

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

    @Override
    protected void doFilterInternal(HttpServletRequest request,
                                    HttpServletResponse response, FilterChain chain)
        throws ServletException, IOException {
        String token = request.getHeader(AUTH_HEADER);
        if (token != null && token.startsWith(BEARER_PREFIX)) {
            token = token.substring(7);
        }
        String username = jwtTokenUtil.getUsernameFromToken(token);

        if (username != null &&
            SecurityContextHolder.getContext().getAuthentication() == null) {
            UserDetails userDetails =
                this.userDetailsService.loadUserByUsername(username);

            if (jwtTokenUtil.tokenValido(token)) {
                UsernamePasswordAuthenticationToken authentication =
                    new UsernamePasswordAuthenticationToken(userDetails,

```

```

        null, userDetails.getAuthorities());
        authentication.setDetails(new WebAuthenticationDetailsSource()
            .buildDetails(request));
        SecurityContextHolder.getContext()
            .setAuthentication(authentication);
    }
}

chain.doFilter(request, response);
}
}

```

Essa é a configuração básica do Spring Security, agora podemos avançar e habilitá-lo na aplicação Spring Boot, para isso crie a classe **WebSecurityConfig.java**:

```

package com.kazale.api.security.config;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import
org.springframework.security.config.annotation.authentication.builders.Authenticatio
nManagerBuilder;
import
org.springframework.security.config.annotation.method.configuration.EnableGlobalM
ethodSecurity;
import org.springframework.security.config.annotation.web.builders.HttpSecurity;
import
org.springframework.security.config.annotation.web.configuration.EnableWebSecurit
y;
import
org.springframework.security.config.annotation.web.configuration.WebSecurityConfi
gurerAdapter;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.core.userdetails.UserDetailsService;

```

```

import org.springframework.security.crypto.bcrypt.BCryptPasswordEncoder;
import org.springframework.security.crypto.password.PasswordEncoder;
import
org.springframework.security.web.authentication.UsernamePasswordAuthenticationF
ilter;

import com.kazale.api.security.JwtAuthenticationEntryPoint;
import com.kazale.api.security.filters.JwtAuthenticationTokenFilter;

@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

    @Autowired
    private JwtAuthenticationEntryPoint unauthorizedHandler;

    @Autowired
    private UserDetailsService userDetailsService;

    @Autowired
    public void configureAuthentication(
        AuthenticationManagerBuilder authenticationManagerBuilder)
        throws Exception {
        authenticationManagerBuilder.userDetailsService(
            this.userDetailsService).passwordEncoder(passwordEncoder());
    }

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }

    @Bean
    public JwtAuthenticationTokenFilter authenticationTokenFilterBean()

```

```

        throws Exception {
            return new JwtAuthenticationTokenFilter();
        }

        @Override
        protected void configure(HttpSecurity httpSecurity) throws Exception {
            httpSecurity.csrf().disable().exceptionHandling().
                authenticationEntryPoint(unauthorizedHandler).and()
                .sessionManagement()
                .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
                .and().authorizeRequests()
                .antMatchers("/auth/**")
                .permitAll().anyRequest().authenticated();
            httpSecurity.addFilterBefore(authenticationTokenFilterBean(),
                UsernamePasswordAuthenticationFilter.class);
            httpSecurity.headers().cacheControl();
        }
    }
}

```

Com isso todas as requisições serão validadas, e dependerão de um token válido para acessar qualquer ação no sistema.

Agora vamos implementar a ação de login na aplicação, que dependerá de três arquivos, um DTO para o usuário, um DTO para retornar o token, e o controller de autenticação.

Seguem ambos arquivos, começando pelo **JwtAuthenticationDto.java**:

```

package com.kazale.api.security.dto;

import org.hibernate.validator.constraints.Email;
import org.hibernate.validator.constraints.NotEmpty;

public class JwtAuthenticationDto {

    private String email;
}

```

```

private String senha;

public JwtAuthenticationDto() {
}

@NotEmpty(message = "Email não pode ser vazio.")
@email(message = "Email inválido.")
public String getEmail() {
    return email;
}

public void setEmail(String email) {
    this.email = email;
}

@NotEmpty(message = "Senha não pode ser vazia.")
public String getSenha() {
    return senha;
}

public void setSenha(String senha) {
    this.senha = senha;
}

@Override
public String toString() {
    return "JwtAuthenticationRequestDto [email=" + email +
        ", senha=" + senha + "]";
}
}

```

Arquivo **TokenDto.java**:

```

package com.kazale.api.security.dto;

```

```
public class TokenDto {  
  
    private String token;  
  
    public TokenDto() {  
    }  
  
    public TokenDto(String token) {  
        this.token = token;  
    }  
  
    public String getToken() {  
        return token;  
    }  
  
    public void setToken(String token) {  
        this.token = token;  
    }  
}
```

E por último o arquivo **AuthenticationController.java**:

```
package com.kazale.api.security.controllers;  
  
import java.util.Optional;  
  
import javax.servlet.http.HttpServletRequest;  
import javax.validation.Valid;  
  
import org.slf4j.Logger;  
import org.slf4j.LoggerFactory;  
import org.springframework.beans.factory.annotation.Autowired;  
import org.springframework.http.ResponseEntity;  
import org.springframework.security.authentication.AuthenticationManager;  
import
```

```

org.springframework.security.authentication.UsernamePasswordAuthenticationToken
;
import org.springframework.security.core.Authentication;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.CrossOrigin;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.kazale.api.response.Response;
import com.kazale.api.security.dto.JwtAuthenticationDto;
import com.kazale.api.security.dto.TokenDto;
import com.kazale.api.security.utils.JwtTokenUtil;

@RestController
@RequestMapping("/auth")
@CrossOrigin(origins = "*")
public class AuthenticationController {

    private static final Logger log =
        LoggerFactory.getLogger(AuthenticationController.class);

    private static final String TOKEN_HEADER = "Authorization";
    private static final String BEARER_PREFIX = "Bearer ";

    @Autowired
    private AuthenticationManager authenticationManager;

    @Autowired
    private JwtTokenUtil jwtTokenUtil;

```

@Autowired**private** UserDetailsService userDetailsService;

/**

* Gera e retorna um novo token JWT.

*

* @param authenticationDto

* @param result

* @return ResponseEntity<Response<TokenDto>>

* @throws AuthenticationException

*/

@PostMapping
public ResponseEntity<Response<TokenDto>> gerarTokenJwt(
 @Valid @RequestBody JwtAuthenticationDto authenticationDto,
 BindingResult result)
 throws AuthenticationException { Response<TokenDto> response = **new** Response<TokenDto>(); **if** (result.hasErrors()) {

log.error("Erro validando lançamento: {}", result.getAllErrors());

result.getAllErrors().forEach(error -> response.getErrors()

.add(error.getDefaultMessage()));

return ResponseEntity.badRequest().body(response);

}

log.info("Gerando token para o email {}", authenticationDto.getEmail());

 Authentication authentication = authenticationManager.authenticate(
 new UsernamePasswordAuthenticationToken(

authenticationDto.getEmail(), authenticationDto.getSenha());

SecurityContextHolder.getContext().setAuthentication(authentication);

 UserDetails userDetails = userDetailsService.loadUserByUsername(

authenticationDto.getEmail());

String token = jwtTokenUtil.obterToken(userDetails);

 response.setData(**new** TokenDto(token));

```

        return ResponseEntity.ok(response);
    }

    /**
     * Gera um novo token com uma nova data de expiração.
     *
     * @param request
     * @return ResponseEntity<Response<TokenDto>>
     */
    @PostMapping(value = "/refresh")
    public ResponseEntity<Response<TokenDto>> gerarRefreshTokenJwt(
        HttpServletRequest request) {
        log.info("Gerando refresh token JWT.");
        Response<TokenDto> response = new Response<TokenDto>();
        Optional<String> token = Optional.ofNullable(
            request.getHeader(TOKEN_HEADER));

        if (token.isPresent() && token.get().startsWith(BEARER_PREFIX)) {
            token = Optional.of(token.get().substring(7));
        }

        if (!token.isPresent()) {
            response.getErrors().add("Token não informado.");
        } else if (!jwtTokenUtil.tokenValido(token.get())) {
            response.getErrors().add("Token inválido ou expirado.");
        }

        if (!response.getErrors().isEmpty()) {
            return ResponseEntity.badRequest().body(response);
        }

        String refreshedToken = jwtTokenUtil.refreshToken(token.get());
        response.setData(new TokenDto(refreshedToken));
    }

```

```

        return ResponseEntity.ok(response);
    }
}

```

Com esses três arquivos implementados, agora já é possível obter um token de autenticação, para isso, adicione ao menos um usuário de teste ao banco de dados.

Existe mais um último recurso que é a autorização por perfil, conforme definido no enum anteriormente, assim é possível validar cada ação de um controle.

Para realizar tal validação, usamos a anotação '@PreAuthorize', e no exemplo a seguir é validado se o usuário possui o perfil de administrador para acessar a ação do controller:

```

package com.kazale.api.controllers;

import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
@RequestMapping("/api/exemplo")
public class ExemploController {

    @GetMapping(value =("/{nome}")
    @PreAuthorize("hasAnyRole('ADMIN')")
    public String exemplo(@PathVariable("nome") String nome) {
        return "Olá " + nome;
    }
}

```

A ação 'hasAnyRole' permite mais de um perfil, dando flexibilidade ao controle, e repare como o prefixo 'ROLE_' também é omitido dele, sendo usado como convenção pelo Spring Security.

Concluindo, certamente esse é o tópico mais complexo de todos os abordados aqui, pois o Spring não possui uma solução pronta para esse tipo de validação, portanto recomendo estudar cada classe em detalhes para entender todo o fluxo de autenticação e autorização de acesso.

28. Adicionando cache com EhCache

⇒ **Problema:**

Gostaria de aumentar a performance da minha aplicação utilizando cache, pois assim eu poderia armazenar resultados de processos lentos em memória, evitando que ele seja executado a todo momento.

⇒ **Solução:**

Muitas vezes precisamos executar tarefas que levam muito tempo para serem executadas, impactando na performance da aplicação.

O EhCache é um mecanismo de cache que se integra muito bem com o Spring Boot, permitindo armazenar dados processados na memória para posterior utilização.

Basicamente quando executamos uma operação lenta, podemos armazenar seu resultado no cache, ou seja, na memória (na maioria dos casos), para que da próxima vez que ela seja executada, a aplicação apenas retorne o valor previamente processado e armazenado no cache.

⇒ **Como fazer:**

O Spring Boot já possui suporte ao EhCache, sendo que para sua utilização basta apenas adicionar a seguinte dependência ao arquivo 'pom.xml':

```
<dependencies>
  <dependency>
    <groupId>net.sf.ehcache</groupId>
    <artifactId>ehcache</artifactId>
  </dependency>
</dependencies>
```

O cache trabalha como um banco de dados em memória no formato chave e valor.

Para marcar o resultado de uma operação a ser adicionada ao cache, você deverá utilizar a anotação '@Cacheable', passando como parâmetro o nome de como o resultado será armazenado no cache.

No código a seguir será criado um serviço que será adicionado ao cache, veja o código a seguir:

```
import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.cache.annotation.Cacheable;
import org.springframework.stereotype.Service;

@Service
public class ExemploCacheService {

    private static final Logger log =
        LoggerFactory.getLogger(ExemploCacheService.class);

    @Cacheable("exemploCache")
    public String exemploCache() {
        log.info("### Executando o serviço...");
        return "Teste de exemplo de cache";
    }
}
```

Com o serviço criado, precisaremos registrar essa entrada no cache, e fazemos isso no arquivo ehcache.xml, que deverá ficar no diretório 'resources' do projeto. Segue seu código:

```
<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:noNamespaceSchemaLocation="ehcache.xsd">
    <cache name="exemploCache"
        maxEntriesLocalHeap="200"
        timeToLiveSeconds="360">
    </cache>
</ehcache>
```

O uso do atributo 'timeToLiveSeconds' é importante para determinar por quanto tempo o resultado será mantido no cache, em segundos.

Para concluir, precisaremos habilitar o cache na aplicação Spring Boot, com o uso da anotação '@EnableCaching', que deverá ser adicionado ao arquivo principal da aplicação.

29. Teste de stress e performance com o Apache AB

⇒ **Problema:**

Gostaria de testar a performance de execução de minha API Restful, simular múltiplos acessos em paralelo, e obter algumas métricas sobre a execução.

⇒ **Solução:**

O Apache AB é um pequeno executável que vem junto com o servidor Apache HTTP, e é ideal para testes de performance em APIs Restful.

Ele permite via linha de commando realizar requisições HTTP, permitindo configurar a quantidade de requisições, quantidade de requisições em paralelo, espaço de tempo para realizar as requisições, entre outros.

No final, ele exibe um relatório com o plano de execução, demonstrando dados e métricas do que foi executado.

⇒ **Como fazer:**

Caso você não tenha instalado o MAMP, WAMP ou LAMP, você deverá baixar o Apache HTTP Server e descompactá-lo.

Para baixar acesse <https://httpd.apache.org/download.cgi>.

O Apache AB está localizado dentro do diretório 'bin' do arquivo baixado, e deverá ser acessado via linha de comando.

Abra o terminal e navegue até o diretório 'bin' dentro da sua instalação do Apache HTTP Server, e execute o seguinte comando:

```
ab -n 10000 -c 100 http://localhost:8080/api/exemplo/fulano
```

O comando acima executará 10000 requisições a URL informada, sendo que com uma concorrência de 100 ao mesmo tempo.

Ao término da execução será exibido um relatório similar ao exibido a seguir:

```

Server Software:
Server Hostname:      localhost
Server Port:          8080

Document Path:        /api/exemplo/Marcio
Document Length:      11 bytes

Concurrency Level:     100
Time taken for tests:  3.613 seconds
Complete requests:     10000
Failed requests:       0
Total transferred:     1440000 bytes
HTML transferred:     110000 bytes
Requests per second:   2767.81 [#/sec] (mean)
Time per request:      36.130 [ms] (mean)
Time per request:      0.361 [ms] (mean, across all concurrent requests)
Transfer rate:         389.22 [Kbytes/sec] received

```

Connection Times (ms)

	min	mean[+/-sd]	median	max
Connect:	0	13 12.6	11	244
Processing:	3	23 13.5	22	343
Waiting:	1	19 12.0	17	338
Total:	6	36 16.9	33	348

Percentage of the requests served within a certain time (ms)

50%	33
66%	39
75%	42
80%	43
90%	50
95%	65
98%	82
99%	108
100%	348 (longest request)

30. Monitorando a JVM com o VisualVM

⇒ **Problema:**

Gostaria de monitorar minha JVM (Java Virtual Machine) de modo visual, para entender mais sobre o uso da memória, recursos, e assim saber se minha aplicação está executando corretamente ou se apresenta alguma falha.

⇒ **Solução:**

O VisualVM é uma aplicação Java que consegue se conectar a JVM em execução no computador, e extrair métricas sobre ela.

Tais métricas são exibidas em forma de tabelas e gráficos, facilitando sua visualização para análise dos dados.

Sua análise pode necessitar conhecimentos avançados em Java, mas é sempre importante ter uma ferramenta como essa a disposição para uma análise mesmo que mais superficial do que está acontecendo.

⇒ **Como fazer:**

Para instalar o VisualVM você deverá acessar a seguinte URL:

<https://visualvm.github.io>

Clique em downloads e faça o download para o seu sistema operacional.

Execute a instalação, que pode variar de sistema para sistema.

Abra o VisualVM com sua aplicação em execução para que ela automaticamente reconheça a JVM e comece a exibir as métricas sobre ela.

31. Instalando o MongoDB para persistir os dados do sistema

⇒ **Problema:**

Gostaria de instalar o MongoDB em meu computador para utilização em meu projeto.

⇒ **Solução:**

Para instalar o MongoDB em um computador, basta efetuar o download dele em seu website oficial, descompactar e executar seu arquivo de inicialização.

Será importante criar um diretório em seu computador para que o MongoDB possa armazenar os dados, e tal diretório deverá ter direitos de acesso de escrita pelo MongoDB.

⇒ **Como fazer:**

Acesse a seguinte URL:

<https://www.mongodb.com/download-center#community>

Escolha o sistema operacional e faça o download.

Crie um diretório para armazenar os dados do MongoDB, que por padrão será '/data/db', e dê permissão de escrita para ele.

Consulte <https://docs.mongodb.com/manual/administration/install-community/> para maiores detalhes.

Agora acesse o diretório do MongoDB, e digite 'bin/mongod' para iniciá-lo, sempre lembrando que seu caminho padrão poderá ser adicionado ao 'path' do sistema para facilitar o acesso.

Para ter acesso ao prompt do MongoDB, digite '/bin/mongo' na raiz da instalação dele.

32. Adicionando o MongoDB ao projeto Spring Boot

⇒ **Problema:**

Gostaria de configurar meu projeto Spring Boot para que ele suporte o MongoDB.

⇒ **Solução:**

O Spring Boot possui suporte ao MongoDB, sendo apenas necessário adicionar o projeto Spring Data MongoDB ao projeto, para que ele passe a suportá-lo.

⇒ **Como fazer:**

Adicione a dependência do Spring Data MongoDB ao arquivo 'pom.xml', conforme código a seguir:

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-mongodb</artifactId>
  </dependency>
</dependencies>
```

Salve o arquivo para que o STS faça o download das dependências.

33. Criando entities e repositórios com o MongoDB

⇒ **Problema:**

Gostaria de criar entidades para mapear documentos do MongoDB, assim como repositórios de acesso em minha aplicação Spring Boot, para que ela execute operações no MongoDB.

⇒ **Solução:**

Com o Spring Data MongoDB adicionado ao projeto, criar entidades e repositórios é uma tarefa simples.

As entidades serão apenas classes Java normais, com apenas uma restrição, ela deve conter o atributo 'id', anotado com a anotação '@Id'.

Já para criar repositórios, basta criar uma interface que estenda 'MongoRepository', assim todas as principais funcionalidades de acesso ao MongoDB já estarão disponíveis para utilização.

⇒ **Como fazer:**

A entidade é uma classe Java normal contendo atributos e assessores getters e setters, e deve incluir a anotação '@Id' para identificar a chave primária, além da '@Document' para informar que se trata de um documento MongoDB, conforme exemplo abaixo:

```
import org.springframework.data.annotation.Id;
import org.springframework.data.mongodb.core.mapping.Document;

@Document(collection = "clientes")
public class Cliente {

    @Id
    private String id;
    private String nome;
    private Integer idade;
```

```
public Cliente() {  
}  
  
public Cliente(String nome, Integer idade) {  
    super();  
    this.nome = nome;  
    this.idade = idade;  
}  
  
public String getId() {  
    return id;  
}  
  
public void setId(String id) {  
    this.id = id;  
}  
  
public String getNome() {  
    return nome;  
}  
  
public void setNome(String nome) {  
    this.nome = nome;  
}  
  
public Integer getIdade() {  
    return idade;  
}  
  
public void setIdade(Integer idade) {  
    this.idade = idade;  
}
```

@Override

```

    public String toString() {
        return "Cliente [id=" + id + ", nome=" + nome + ", idade=" + idade + "]";
    }
}

```

O atributo 'collection' serve para nomear a coleção no MongoDB.

Para criar o repositório de acesso, estenda interface 'MongoRepository' conforme o código a seguir:

```

import java.util.List;

import org.springframework.data.mongodb.repository.MongoRepository;
import org.springframework.data.mongodb.repository.Query;

import com.kazale.documents.Cliente;

public interface ClienteRepository extends MongoRepository<Cliente, String> {

    Cliente findByNome(String nome);

    @Query("{ 'idade' : { $gt: ?0, $lt: ?1 } }")
    List<Cliente> findByIdadeBetween(int idadeInicial, int idadeFinal);
}

```

Repare que foram adicionados dois métodos, um para buscar por nome, seguindo as convenções do Spring, e o outro para executar uma query buscando por um intervalo de idades.

A query do segundo método usa a linguagem de queries do próprio MongoDB, então será necessário conhecimento sobre ela para criar queries mais complexas.

Para finalizar a configuração, faltou especificar o nome da coleção no MongoDB, e fazemos isso adicionando a seguinte entrada ao arquivo 'application.properties'.

```
spring.data.mongodb.database=spring-mongodb
```

No mesmo arquivo properties, seguindo o prefixo 'spring.data.mongodb', é possível acessar diversas outras configurações do MongoDB.

Por fim, segue um código de testes adicionado ao arquivo principal do projeto:

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.CommandLineRunner;
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

import com.kazale.documents.Cliente;
import com.kazale.repositories.ClienteRepository;

@SpringBootApplication
public class MongoDBApplication implements CommandLineRunner {

    @Autowired
    private ClienteRepository repository;

    public static void main(String[] args) {
        SpringApplication.run(MongoDbApplication.class, args);
    }

    @Override
    public void run(String... args) throws Exception {

        repository.deleteAll();

        repository.save(new Cliente("Alice", 20));
        repository.save(new Cliente("João", 30));
        repository.save(new Cliente("Maria", 40));

        System.out.println("Lista todos com o findAll():");
    }
}
```

```
System.out.println("-----");
repository.findAll().forEach(System.out::println);
System.out.println();

System.out.println("Busca um único cliente com o findByNome('Alice'):");
System.out.println("-----");
System.out.println(repository.findByNome("Alice"));
System.out.println();

System.out.println("Clientes com idade entre 18 and 35:");
System.out.println("-----");
repository.findByIdadeBetween(18, 35).forEach(System.out::println);
}
}
```

34. Publicando a API no Heroku

⇒ **Problema:**

Gostaria de publicar minha API Spring Boot na nuvem, usando um provedor de serviços eficiente e simples de configurar.

⇒ **Solução:**

O Heroku é a solução ideal e mais simples para publicar uma aplicação na nuvem em minutos.

Ele já oferece um suporte ao Spring Boot, facilitando todo o processo.

O Heroku utiliza o Git e seus comandos para a publicação, portanto é interessante um conhecimento básico sobre Git ao menos.

Ele possui também um cliente que auxilia na criação e publicação da aplicação, chamado 'Heroku CLI'.

Por fim, caso você precisa adicionar configurações específicas para a publicação, você pode criar o arquivo 'Procfile' na raiz do projeto com tais configurações, que são instruções de como ele deverá executar a sua aplicação em seus servidores na nuvem.

⇒ **Como fazer:**

Primeiramente crie uma conta gratuita no Heroku acessando <https://www.heroku.com/>.

A seguir acesse <https://devcenter.heroku.com/articles/getting-started-with-java#set-up>, e efetue o download do 'Heroku CLI', realizando sua instalação.

Abra o terminal e execute o comando 'heroku login', digite seu email e senha para logar.

Pelo fato do Heroku ser baseado no Git para efetuar o upload da aplicação para ele, tenha certeza de ter o Git instalado, conforme explicado no tópico 6.

Acesse pelo terminal a raiz do projeto a ser enviado para o Heroku, e execute os seguintes comandos:

```
git init
git add .
git commit -m "Primeiro commit"
heroku create
git push heroku master
heroku open
```

Pronto, sua aplicação está em funcionamento na nuvem!

Os comandos executados são para adicionar o código ao Git do Heroku, depois em ‘heroku create’ uma nova máquina virtual é criada para o deploy, seguido do envio dos arquivos para ele.

Para atualizar a aplicação, basta executar os mesmos passos, exceto o ‘heroku create’ por já existir uma máquina virtual criada.

Conclusão

Parabéns por ter concluído todos os tópicos apresentados neste livro!

Pode ter certeza que agora você possui os principais conhecimentos estruturais e arquiteturais para criar uma API Restful totalmente escalável e de alta performance.

Utilize o livro sempre que precisar tirar uma dúvida sobre um tópico em específico, e continue seus estudos colocando tudo o que foi aprendido aqui criando sua própria API Restful.

Boa sorte e nos vemos em breve.