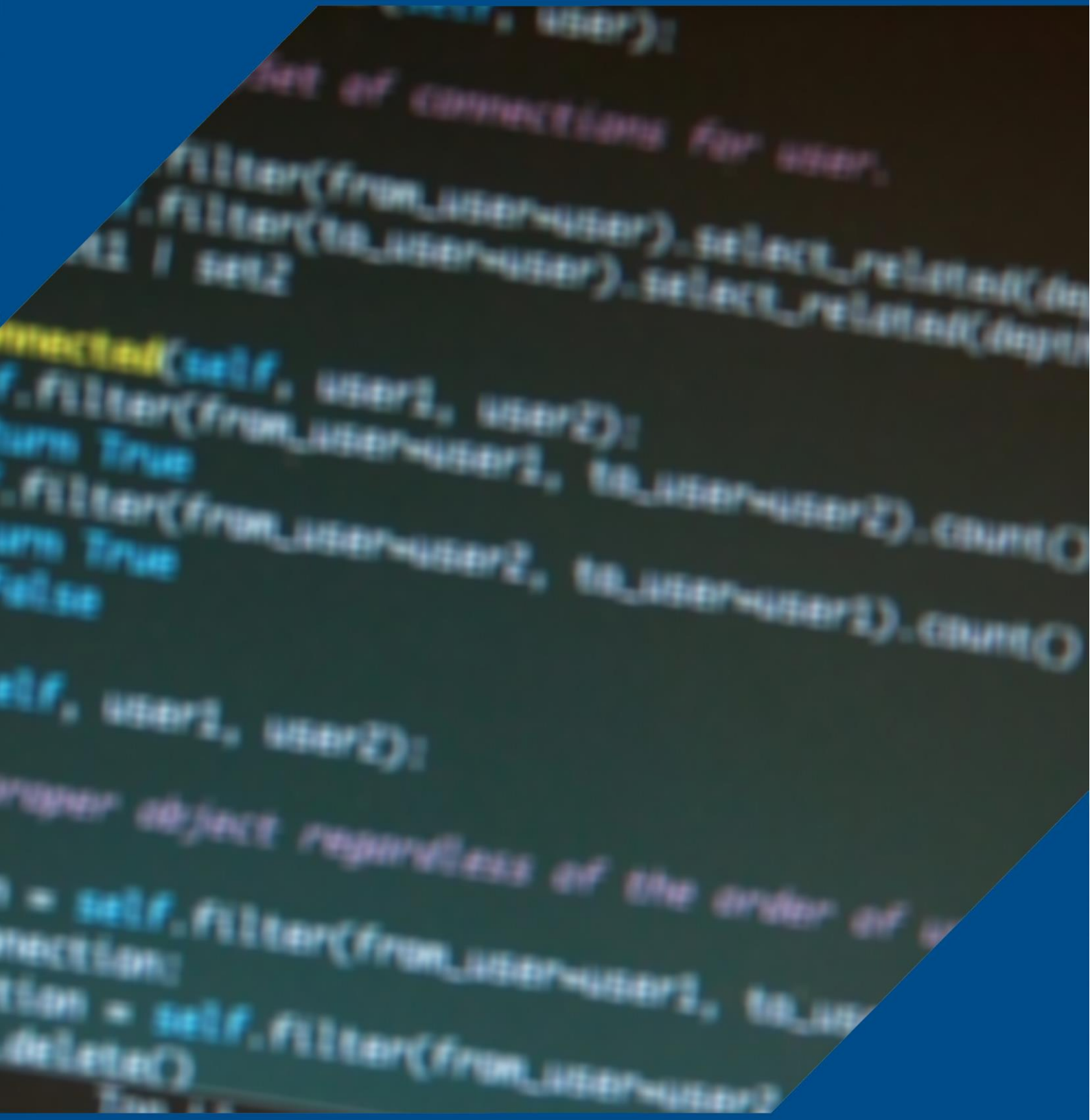
DISCIPLINA

DESENVOLVIMENTO BACK-END EM JAVA COM SPRING BOOT



Capítulo 1 - INTRODUÇÃO AO SPRING BOOT

Capítulo 1 - INTRODUÇÃO AO SPRING BOOT

Neste capítulo você irá:

- Compreender o que é o Spring Boot e a diferença que há entre ele, o Spring e o Spring MVC.
- Desenvolver uma aplicação simples com Spring Boot.
- Obter uma visão geral de como um sistema para web é arquitetado.

Criar uma aplicação com Spring Boot é fácil

O Objetivo do Spring Boot é facilitar a criação de aplicações baseadas em Spring, deixando-as prontas para serem utilizadas em produção sem configurações adicionais, você apenas as executa e pronto.

Para provar isso, antes de começar qualquer introdução à esta ferramenta, vamos criar uma aplicação simples com ela.

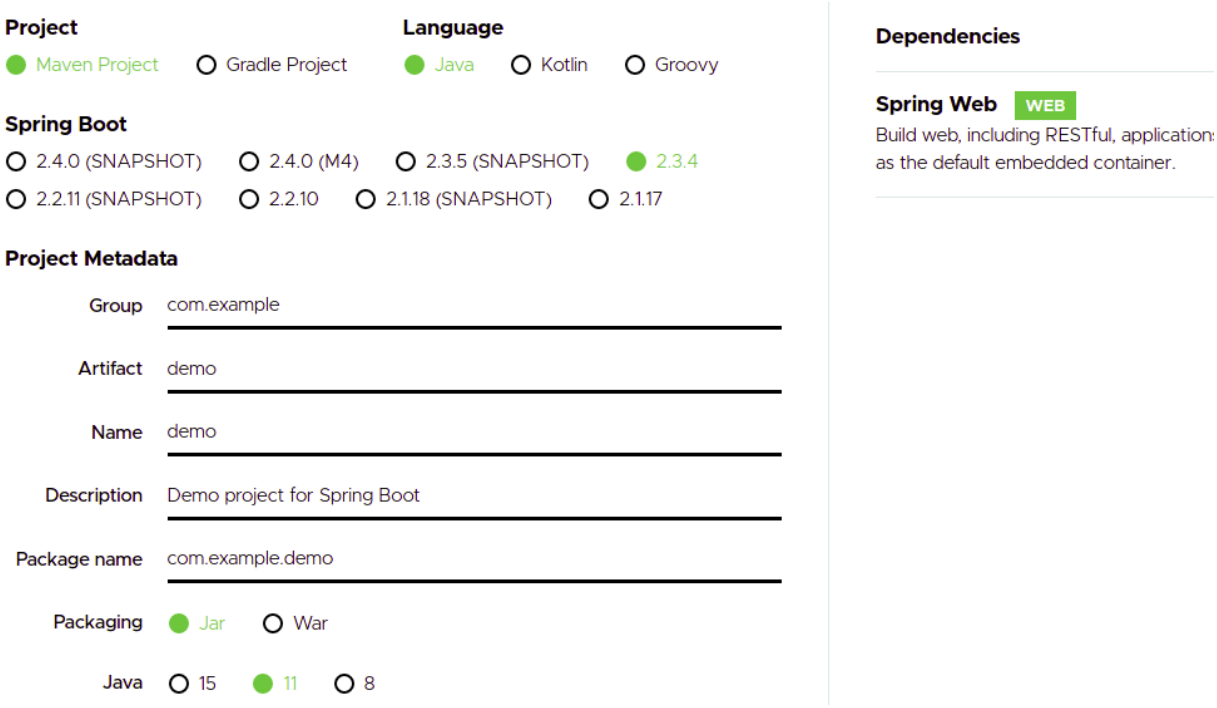
Spring Initializr

Ferramenta que oferece uma maneira rápida de configurar uma aplicação com tudo que ela precisa. Você seleciona os recursos que deseja utilizar e faz o download da estrutura inicial, pronta para ser importada no seu IDE. Para o primeiro exemplo vamos precisar apenas da dependência Spring Web.

1. Acesse o site http://start.spring.io
2. Selecione:
    a. Project: Maven Project
    b. Language: Java
    c. Spring Boot: 2.3.4
    d. Group: Nome da organização
    e. Artifact: nome da aplicação
    f. Packaging: Jar
    g. Java: 11
    h. Dependencies: Spring Web
3. Clique em Generate Project para ele gerar o projeto e disponibilizar um arquivo .zip.

4. Descompacte este arquivo em algum local do seu computador que seja de fácil acesso.
5. No eclipse, clique em File > Import > Maven > Existing Maven Projects. Clique em Next.

Figura 1.1 - Exemplo de Spring Initializr



Fonte: O autor

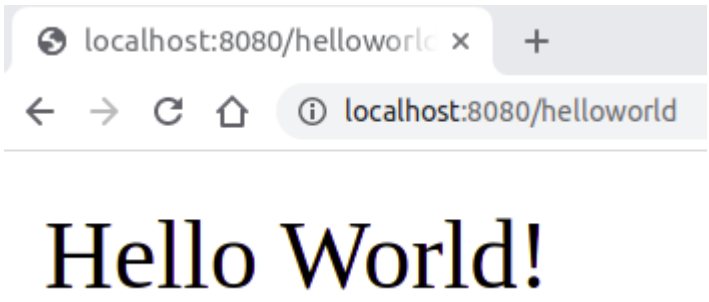
Criando uma Aplicação Web simples

Agora basta alterar a classe DemoApplication transformando-a em um Web Controller. Não se preocupe neste momento com as diversas anotações, pois as veremos com mais detalhes no decorrer do curso.

```
@SpringBootApplication
@RestController
public class DemoApplication {

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

  @GetMapping("/helloworld")
  public String hello() {
    return "Hello World!";
  }

}
```

Em seu navegador acesse localhost:8080/helloworld:

Figura 1.2 - Primeira Aplicação Web



Fonte: O autor

Simples e fácil, não? Mas o que aconteceu aqui? O que são as anotações () que foram adicionadas à classe? E o principal, como nossa aplicação está respondendo à requisição que fizemos no browser? São estas respostas que serão respondidas no decorrer do curso.

Spring Tool Suite

Se desejar, você pode criar os projetos diretamente de uma IDE com suporte para desenvolvimento de projetos Spring: https://spring.io/tools.

Visão geral de uma aplicação Web

Sistemas webs são desenvolvidos seguindo a arquitetura Servidor-Cliente. Nesta arquitetura existe uma aplicação rodando em uma máquina servidora que é responsável em atender à requisições que chegam a ela através de um protocolo em comum entre as duas partes. Esta aplicação servidora é então responsável em processar a requisição, executando as lógicas de negócios necessárias, e retorna uma resposta ao cliente.

Aplicações servidoras são executadas a partir de um servidor de aplicação ou conteiners de aplicação. Um dos servidores de aplicação mais comum é o Apache Tomcat. Ele é responsável em gerenciar todo contexto da aplicação, receber as requisições que vem dos clientes, executar a lógica dos programas que são configuradas nele e então retornar as respostas para os clientes.

Já os clientes são os consumidores dessas aplicações. Um exemplo comum de cliente são os navegadores web, como o Chrome ou Firefox. Estes navegadores são capazes de executar códigos em Javascript que fazem a comunicação com os servidores.

Neste cenário temos basicamente duas aplicações distintas. Uma sendo executando no servidor, desenvolvido com uma linguagem robusta, contendo toda lógica de negócio. A outra sendo executada no próprio navegador, utilizando uma linguagem dinâmica como o JS, que é capaz de enviar e receber dados do servidor e manipular páginas em html para apresentar o conteúdo de forma interativa.

Protocolo HTTP

Hypertext Transfer Protocol, ou HTTP, é o protocolo comum utilizado pelas aplicações clientes e servidoras que possibilitam a comunicação entre as duas partes. Ele é a base de toda comunicação que existe hoje na web.

Request e Response

A comunicação entre aplicações clientes e servidoras é realizada a partir de uma requisição (Request) originada por um cliente e pela resposta (Response) fornecida pelo servidor. São as mensagens trocadas entre as duas partes.

Partes de uma requisição HTTP:

- **HTTP Method:** ação executada (GET, POST, PUT, DELETE)
- **URL:** local a acessar
- **Form Parameters:** Similares aos argumentos de métodos Java.

Exemplo de uma requisição HTTP:

```
Request URL: https://www.google.com.br/?gfe_rd=cr
Request Method:GET
Status Code:200
Remote Address:216.58.222.3:443
```

Partes de uma resposta HTTP

Status Code: número inteiro que indica se a requisição teve sucesso ou não.Códigos mais conhecidos são o 200 (Success), 404 (Not Found) e 403 (Access Forbiden).

Content Type: text, html, image, pdf, etc. Também são chamados de MIME type.

Content: dados que devem ser renderizados pelo Web Client.

Exemplo de uma resposta HTTP

```
200 OK
Date: Fri, 01 Jul 2016 07:59:39 GMT
Server: gws
Content-Type: text/html;charset=UTF-8I
```

Entendendo a URL

URL é acrônimo para Universal Resource Locator e é usado para localizar um servidor e recursos. Todo recurso na web tem seu próprio endereço. Vamos ver as partes de uma URL:

`http://localhost:8080/Resource`

http:// - é a primeira parte da URL e informa qual o protocolo de comunicação que será usado entre servidor e cliente.

localhost - endereço único do servidor, na maioria das vezes é um endereço ip.

8080 - porta em que o servidor ouve as requisições.

Resource - Recurso requisitado do servidor (html, xml, json, pdf, imagens, etc).

Diferença entre Web Services, API e Microservices

A visão geral de uma aplicação web apresentada aqui considera casos onde um usuário esteja acessando algum site estático, onde as páginas, imagens e outros recursos são fornecidas do servidor para o cliente sem a necessidade de um processamento de dados por parte do servidor.

Mas vamos considerar um outro exemplo, onde uma aplicação tenha funcionalidade de cadastrar usuários em uma base de dados. Esta aplicação é então publicada em um servidor onde poderá ser acessada por outras aplicações (clientes) que desejam cadastrar usuários na mesma base de dados. Como estes clientes poderão acessar esta funcionalidade?

Uma aplicação cliente poderia acessar a funcionalidade cadastro de usuário através de uma *Application Programming Interface* (API), que é uma interface agnóstica de linguagem de programação que conecta dois programas, permitindo a comunicação e

a troca de dados entre eles. Ou seja, uma API é uma forma onde um sistema pode fornecer uma funcionalidade sua para ser utilizada por outro sistema. Esta troca de informação é realizada através de rotas e protocolos definidos, e os dados trafegados normalmente são em formato JSON e XML.

Um Web Service (WS) funciona praticamente da mesma maneira de uma API, porém a comunicação ocorre utilizando-se o protocolo HTTP. Um WS pode utilizar REST, SOAP ou XML-RPC como meios de comunicação. Aqui focaremos na implementação de um WS utilizando o estilo REST, que será introduzido mais a frente. Uma curiosidade é que todo Web Service é uma API, mas nem toda API é um Web Service.

E os Microservices? Trata-se de uma abordagem de arquitetura de software, onde você desenvolve vários WS com funcionalidade bem específicas. Nesta arquitetura você pode ter clientes que utilizam apenas os serviços que necessitam isoladamente. A vantagem aqui é ter WS menores e bem específicos. A desvantagem é ter uma arquitetura mais complexa para manter. Microservices não serão abordados aqui, mas é uma ótima sugestão de estudos a seguir ao término de nossos estudos.

Spring Framework

Como você deve ter imaginado, criar uma aplicação para funcionar na Web não deve ser tarefa fácil. Como fazer por exemplo que uma aplicação seja capaz de responder requisições através de uma URL digitada no navegador? Como fazer para lidar com todas particularidades de infraestrutura e protocolos? Como fazer para lidar com requisições que enviam dados em formato XML ou JSON e que precisam ser persistido em um banco de dados? E por falar em banco de dados, como fazer a comunicação com eles? É tanta coisa que deveríamos nos preocupar que o mais importante, a lógica da nossa aplicação ficaria de lado. Para nossa sorte existem frameworks que lidam com toda essas dificuldades para nós.

E o que são frameworks? São ferramentas que se encarregam de deixar pronto ou facilitar ao máximo o desenvolvimento de uma aplicação. Eles se encarregam de lidar com estas dificuldades que são corriqueiras a toda aplicação, deixando o programador livre para focar apenas no desenvolvimento das regras de negócio.

E o Spring é o framework que facilita a criação de aplicações corporativas em Java mais popular, reunindo recursos que são necessários em um único ambiente. É um framework que engloba vários outros módulos. Aplicações podem ser desenvolvidas

usando apenas os módulos necessários. Em seu core estão os módulos do container principal, incluindo o modelo de configuração, os mecanismo de injeção de dependência e o inversão de controle, suporte a Aspect Oriented Programming (AOP), acesso a dados, suítes de testes e suporte para desenvolvimento de aplicações de diferentes arquiteturas, como sistemas de mensageria, transacionais e web.

Aplicações Web Java que não usam o Spring devem ser executadas a partir de um servidor de aplicação, como o Apache Tomcat. Já o Spring traz seu próprio container alternativo que é responsável em gerenciar todos objetos que estão dentro da aplicação. Um change gamer importante neste cenário de aplicações corporativas.

Um dos grandes diferenciais do Spring é o fato de seu Container fornecer recursos avançados para desenvolvimento de aplicações corporativas que antes eram oferecidos apenas por servidores de aplicações pesados. Seu requisito básico de funcionamento é apenas a Java Virtual Machine (JVM).

Outra vantagem é o fato de possibilitar apenas a inclusão dos módulos necessários para uma aplicação, diferente de outros frameworks onde é necessário utilizar o framework inteiro, que pode tornar aplicação pesada. Com Spring a arquitetura fica leve e fácil de manter.

Ele também oferece um robusto suporte às tecnologias de persistência do mundo Java, como JDBC, ORMs como Hibernate, iBatis e JPA. Mais a frente veremos como é fácil acessar dados usando o suporte ao JPA do Spring. Ele contém um esquema de templates que reduzem drasticamente a quantidade de código que normalmente devemos escrever para acessar dados em um banco, como por exemplo abrir e fechar conexões, iniciar transações, fazer commit ou rollback. Aliás, este controle transacional libera nós desenvolvedores de toda esta responsabilidade, podemos deixar a cargo do Container que determinará quando deve ou não confirmar uma transação, quando abrir e fechar conexão com banco, etc.

Spring MVC

É um módulo do Spring Framework, que contém tudo o que é necessário para facilitar o desenvolvimento de aplicações Web sob o protocolo HTTP usando o padrão MVC (Model-View-Controller). O padrão MVC dita como devem ser separadas as lógicas de entrada através dos Controladores, as lógicas de negócios através dos Modelos e das lógicas de Visualizações.

Foi projetado em cima da Servlet API e usa o padrão de projeto Front Controller, onde um Servlet principal, nomeado de *DispatcherServlet* é responsável em receber toda requisição e direcionar para os controladores responsáveis em lidar com ela.

Spring MVC fornece um modelo de de programação baseado em anotações onde classes que são anotadas com *@Controller* e *@RestController* se tornam componentes que mapeiam requisições realizadas, tratam os dados de entrada e saída, manipulam exceções, etc. Controllers então pode ser entendido como as portas de entrada e saída da nossa aplicação com o mundo externo.

Os métodos das classes com estas anotações podem usar anotações específicas para mapear os métodos HTTP, como:

GET - *@GetMapping*;

POST - *@PostMapping*;

PUT - *@PutMapping*;

DELETE - *@DeleteMapping*

Cada uma destas anotações será vista em detalhes mais a frente.

Ele também traz suporte à biblioteca Jackson JSON, que possibilita a conversão de dados em formatos JSON para classes Java e vice-versa. Esta biblioteca é muito útil pois converte automaticamente os dados de entrada e saída dos métodos dos Controllers.

Contém também um manipulador de Exceções centralizado que facilita a conversão das mensagens de erros que estouram na aplicação em mensagens mais amigáveis para o cliente. Isso é um requisito comum em serviços REST, onde se deve incluir os detalhes dos erros no corpo da resposta. O Spring não pode fazer isso automaticamente, pois se trata de regras específicas de cada aplicação, mas traz mecanismos que permitem mapear tipos de exceções e retornar uma mensagem e um código HTTP mais adequado a cada erro ocorrido.

Listei aqui apenas alguns dos recursos oferecidos pelo Spring MVC, há muitos outros disponíveis responsáveis em auxiliar alguma parte específica do desenvolvimento para web. Mas toda esta gama trouxe também um problema, configurar cada um destes recursos em uma aplicação ficou complicado, pois era necessário escrever

cada vez mais arquivos xmls para fazer uma aplicação simples rodar. Isso fez com que surgisse uma ferramenta para facilitar nesta tarefa.

Spring Boot

É uma ferramenta interativa para você criar suas aplicações de maneira rápida, prontas para serem usadas em produção com a mínima ou nenhuma necessidade de configurações através de arquivos xml. As aplicações criadas a partir desta ferramenta podem ser executadas como se fossem aplicações *Stand-Alone* através do comando "java -jar" ou a partir de pacotes WAR tradicionais.

Conforme sua própria documentação os objetivos principais do Spring Boot são:

- Fornecer uma introdução rápida e acessível para os programadores iniciarem o desenvolvimento com Spring.
- Fornecer uma série de recursos não funcionais e de alta complexidade como servidores embarcados na própria aplicação, segurança, métricas, verificações, configuração simplificada, etc.

O gerenciamento de dependências é um aspecto crítico em projetos complexos, e fazê-lo manualmente pode trazer muitas dores de cabeça. Por isso o Spring Boot traz uma série de Starters, que são conjuntos de dependências preparadas para objetivos específicos e que podem ser adicionados aos sistemas.

Alguns dos starters mais utilizados são:

Web Starter

Para desenvolvimento de serviços REST, normalmente precisamos de bibliotecas como Spring MVC, Tomcat e Jackson, entre outras. O Starter "spring-boot-starter-web" adiciona todas estas bibliotecas na aplicação com apenas a adição de uma dependência no arquivo POM.xml:

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

Test Starter

Este starter adiciona as bibliotecas Spring Test, JUnit, Hamcrest, Mockito, etc, que são utilizadas para desenvolver testes unitários e de integração.

JPA Starter

Adiciona as bibliotecas necessária para manipular acesso a dados a partir da especificação JPA do Java.

Validation Starter

Adiciona as bibliotecas necessárias para fazer a validação dos dados a partir de anotações.

No decorrer do curso veremos muitos destes recursos na prática.

Links Úteis

Site oficial do Spring:

http://spring.io

Projeto Starter para criar projetos Spring Boot:

http://start.spring.io

Baixar a IDE Spring Tool Suite:

https://spring.io/tools

Considerações Finais

Vimos neste capítulo:

- Uma visão geral de como funcionam as aplicações para web.
- Como é fácil criar uma aplicação com Spring Boot.
- O que é o Spring Framework.
- O que é o Spring MVC.
- O que é Spring Boot.

No próximo capítulos veremos como criar REST API completa.

Capítulo 2 - PERSISTÊNCIA DE DADOS

Neste capítulo você irá:

- Aprender a criar as Entidades que serão mapeadas para as tabelas do banco de dados relacional com JPA.
- Conhecer o padrão repository do Spring Data.
- Aprender a fazer queries customizadas utilizando repository do Spring Data.
- Explorar os recursos de paginação e ordenação.

Java Persistence API (JPA)

Java Persistence API, ou JPA, é a especificação que dita como deve ser feito o Mapeamento Objeto Relacional (ORM em inglês), para armazenamento, acesso e gerenciamento de objetos Java em banco de dados relacionais.

Objetos Java no contexto de bancos de dados relacionais são definidos como Entidades. Estas entidades são mapeadas como tabelas no banco de dados. Os atributos das entidades são mapeados como colunas. Já os registros são a materialização de cada objeto.

No banco de dados, usamos *fereign keys* e *join tables* para definir o relacionamento entre as tabelas. Nas entidades usamos anotações específicas para determinar estes relacionamentos.

Entidades

Uma classe POJO pode ser definida como entidade através da anotação @Entity. Por exemplo:

```
@Entity
public class Tarefa {

}
```

Desta maneira, a classe Tarefa será mapeada como uma tabela chamada tarefa no banco de dados. Caso queira definir um nome alternativo para a tabela que será criada, utilize a anotação @Table. Esta possui propriedades para customizar a criação da tabela no banco de dados.

```
@Entity
@Table(name = "tarefas")
public class Tarefa {

}
```

Primary Key

Para determinar o atributo da classe que será utilizado como chave primária na tabela, fazemos uso da anotação @Id. Normalmente utilizamos Integer, Long ou String como tipo deste atributo.

```
@Entity
@Table(name = "tarefas")
public class Tarefa {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  private Long id;
```

A anotação @GeneratedValue indica que este campo será gerado automaticamente.

Atributos para colunas

Os atributos desta classe serão mapeados como colunas da tabela criada. Você também pode utilizar a anotação @Column para customizar detalhes da coluna:

```
@Entity
@Table(name = "tarefas")
public class Tarefa {

  @Id
  @GeneratedValue(strategy = GenerationType.AUTO)
  @Column(name = "codigo")
  private Long id;

  @Column(name = "ds_tarefa", nullable = false, length = 100)
  private String descricao;
```

Associação entre Entidades

O JPA define quatro anotações para tratar relacionamentos entre entidades:

@OneToOne

Utilizada para definir relações do tipo um para um, onde uma entidade tem outra entidade como um de seus atributos. Por exemplo, uma entidade Tarefa pode fazer ter um atributo do tipo Arquivo para armazenar uma foto no banco:

```
@Entity
@Table(name = "tarefas")
public class Tarefa {

  @Id
```

```
  @GeneratedValue(strategy = GenerationType.AUTO)
  @Column(name = "codigo")
  private Long id;

  @Column(name = "ds_tarefa", nullable = false, length = 100)
  private String descricao;

  @OneToOne(cascade = CascadeType.REMOVE)
  @JoinColumn(name = "arquivo_id")
  private Arquivo arquivo;
```

A opção cascade informa que o registro de Arquivo relacionado com o Tarefa deve ser apagado caso o curso seja apagado.

A anotação @JoinColumn define algumas opções do campo que será a chave estrangeira da tabela. Neste caso estamos definindo o nome do campo.

@OneToMany e @ManyToOne

Estas duas anotações são utilizadas para relacionamentos do tipo um para muitos. Considere o exemplo onde uma entidade Tarefa tenha apenas um Responsavel, mas o Responsavel pode ter vários Cursos. Este relacionamento seria mapeado assim:

```
@Entity
@Table(name = "tarefas")
public class Tarefa {

  // ...
  @ManyToOne
  @JoinColumn(name = "autor_id")
  private Responsavel responsavel;
```

Classe Responsavel:

```
@Entity
public class Responsavel {

@OneToMany(mappedBy = "responsavel")
private List<Tarefa> tarefas;
```

@ManyToMany

Esta anotação é utilizada para relações do tipo muitos para muitos. Considere o exemplo onde várias tarefas poderiam ter vários responsáveis.

```
@Entity
@Table(name = "tarefas")
public class Tarefa {

  @ManyToMany
  @JoinTable(name = "tarefa_responsavel",
      joinColumns = @JoinColumn(name="tarefa_id"),
      inverseJoinColumns = @JoinColumn(name="responsavel_id"))
```

```
  private List<Responsavel> responsavel;

// ...

}
```

Classe Responsavel:

```
@Entity
public class Responsavel {

@ManyToMany(mappedBy = "responsavel")
private List<Tarefa> tarefas;
```

Neste caso, a anotação @JoinTable faz com que uma tabela chamada tarefa_responsavel seja criada com duas colunas, tarefa_id e responsavel_id. As opções joinColumns e inverseJoinColumns diz como devem ser feitas as restrições das chaves estrangeiras.

Spring Data JPA

Spring Data JPA é um sub-projeto do Spring Data, que torna fácil a implementação de repositórios Spring utilizando JPA. Ele é destinado para implementação da camada de acesso a dados de uma aplicação. Estas camadas normalmente ficam poluídas com muito código para executar operações como paginação, auditoria, além de carregar um boilerplate de códigos referente a infraestrutura.

O Spring Data vem para resolver este problema introduzindo o conceito do padrão de projeto Repository.

Spring Data Repository

O objetivo principal dos repositórios do Spring Data é trazer uma abstração para reduzir significativamente a quantidade de códigos que precisam ser escritos para desenvolver a camada de acesso aos dados de uma aplicação.

A interface principal do Spring Data é a Repository. Ela se baseia na classe de domínio, que anotamos anteriormente com @Entity, bem como o tipo de dados do seu ID (Integer, Long,...) para fazer o gerenciamento da Entidade. Isso significa que ele será capaz de executar operações como Inser, Update, Select, etc.

Nós não precisamos implementar nada para esta interface para já sair usando. O Spring irá gerar uma implementação padrão em tempo de execução que fornecerá uma gama de métodos que podemos utilizar pela aplicação. A interface CrudRepository, por exemplo, provê as funcionalidades básicas das operações de

CRUD. Outra interface interessante é a JPARepository, que herda os comportamentos da interface CrudRepository, trazendo uma especialização deste repositório para trabalhar com JPA.

Além disso, CrudRepository herda de PagingAndSortingRepository, que provê funcionalidades de paginação de resultados e ordenação.

Query Methods

As interface mencionadas acima trazem uma gama enorme de métodos que podemos utilizar sem escrever uma linha de código sequer. Contudo, quando estamos desenvolvendo aplicações CRUD, precisaremos em alguma hora utilizar um método de consulta mais especializado. Com SpringData podemos escrever assinaturas de métodos em nossas interfaces que seguem um padrão de nomenclatura para executar operações no banco de dados.

Por exemplo, para buscar uma tarefa dado uma descrição, escrevemos a assinatura do método na interface:

```
public interface TarefaRepository
      extends JpaRepository<Tarefa, Long> {

  List<Tarefa> findByDescricao(String descricao);
}
```

É possível criar vários tipos de métodos que executam operações distintas no banco de dados simplesmente obedecendo esta convenção na nomenclatura dos métodos. A tabela 4.1 traz uma série destas convenções.

Tabela 4.1 - Padrão de nomes suportados pelo Spring Data JPA

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	… where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	… where x.lastname = ?1 or x.firstname = ?2
Is, Equals	findByFirstname,findByFirstnameIs,findByFirstnameEquals	… where x.firstname = ?1
Between	findByStartDateBetween	… where x.startDate between ?1 and ?2
LessThan	findByAgeLessThan	… where x.age < ?1
LessThanEqual	findByAgeLessThanEqual	… where x.age <= ?1
GreaterThan	findByAgeGreaterThan	… where x.age > ?1
GreaterThanEqual	findByAgeGreaterThanEqual	… where x.age >= ?1
After	findByStartDateAfter	… where x.startDate > ?1

Before	findByStartDateBefore	… where x.startDate < ?1
IsNull, Null	findByAge(Is)Null	… where x.age is null
IsNotNull, NotNull	findByAge(Is)NotNull	… where x.age not null
Like	findByFirstnameLike	… where x.firstname like ?1
NotLike	findByFirstnameNotLike	… where x.firstname not like ?1
StartingWith	findByFirstnameStartingWith	… where x.firstname like ?1 (parameter bound with appended %)
EndingWith	findByFirstnameEndingWith	… where x.firstname like ?1 (parameter bound with prepended %)
Containing	findByFirstnameContaining	… where x.firstname like ?1 (parameter bound wrapped in %)
OrderBy	findByAgeOrderByLastnameDesc	… where x.age = ?1 order by x.lastname desc
Not	findByLastnameNot	… where x.lastname <> ?1
In	findByAgeIn(Collection<Age> ages)	… where x.age in ?1
NotIn	findByAgeNotIn(Collection<Age> ages)	… where x.age not in ?1
TRUE	findByActiveTrue()	… where x.active = true
FALSE	findByActiveFalse()	… where x.active = false
IgnoreCase	findByFirstnameIgnoreCase	… where UPPER(x.firstame) = UPPER(?1)

Fonte: https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#reference

Este padrão já resolve boa parte das operações que precisamos fazer no banco de dados. Mas há momentos que serão necessários criar queries mais complexas. Para isso, o Spring Data traz a possibilidade de escrever suas próprias queries.

JPA Named Queries

Através de uma anotação na Entidade podemos definir uma queries customizadas e nomeá-las. Depois podemos utilizá-las fazendo referência a seu nome.

Por exemplo, vamos criar uma criar que recupera tarefas pelo seu status:

```
@Entity
@NamedQuery(name = "Tarefa.findByStatus",
query = "select t from Tarefa where t.status = ?1")
public class Tarefa extends BaseEntity {
//...
}
```

Definida a query, devemos então fazer a declaração na classe TarefaRepository:

```
public interface TarefaRepository extends JpaRepository<Tarefa, Long> {

  List<Tarefa> findByStatus(String status);
}
```

Spring Data tenta encontrar primeiro esses métodos anotadas com @NamedQuery, começando com o nome simples da classe de domínio configurada, seguido pelo nome do método separado por um ponto. Caso não encontre, ele tentará fornecer uma implementação padrão seguindo a convenção de nomes vista anteriormente.

Uma outra maneira de criar queries customizadas é através da anotação @Query.

Anotação @Query

Usar consultas nomeadas nas entidades funciona bem para um pequeno número de consultas. Mas existe uma outra possibilidade de declarar estas queries diretamente na classe Repository com a anotação @Query.

O exemplo a seguir mostra como ficaria a mesma query anotada diretamente no Repository:

```
public interface TarefaRepository
      extends JpaRepository<Tarefa, Long> {

  @Query("select t from Tarefa where t.status = ?1")
  List<Tarefa> findByStatus(String status);
```

Com esta anotação é possível utilizar queries nativas do banco de dados:

```
public interface TarefaRepository
      extends JpaRepository<Tarefa, Long> {

@Query(value = "select * from tarefas where status = ?1",
       nativeQuery = true)
  List<Tarefa> findByStatus(String status);
```

Desta maneira fica mais direta a criação de queries complexas no repositório.

Query by example

Existe uma outra maneira de criar queries de forma mais amigável através da técnica Query by Example, que permite a criação de queries dinâmicas a partir de um objeto de exemplo, sem a necessidade de escrever a query em si.

Para entender esta técnica vamos implementar um exemplo considerando a entidade Tarefa vista anteriormente. Vamos utilizá-la para criar nosso exemplo (Example). Por

padrão, campos com valores nulos são ignorados. Já para Strings precisamos definir alguns critérios de como ele fará a pesquisa.

```
Tarefa tarefa = new Tarefa();
tarefa.setDescricao("estudar java");

Example<Tarefa> exemplo = Example.of(tarefa);
```

Podemos então definir como ele devem considerar as Strings:

```
ExampleMatcher matcher = ExampleMatcher
      .matching()
      .withIgnoreCase()
      .withStringMatcher(
            ExampleMatcher.StringMatcher.CONTAINING);

Example<Tarefa> exemplo = Example.of(tarefa, matcher);
```

No código acima estamos informando que os valores dos campos do tipo String não precisam bater exatamente, apenas devem conter parte do texto informado. Neste caso, seria retornado todas tarefas que contivessem a descrição “estudar java”. Também pedimos para ignorar a diferenciação entre letras maiúsculas e minúsculas.

Estes objetos podem ser passados para o método de pesquisa padrão *findAll* contido na interface JpaRepository:

```
return repositorio.findAll(example);
```

Esta interface contém uma sobrecarga do método que recebe como parâmetro o exemplo que criamos e retorna uma lista de objetos da Entidade.

Existem outros recursos além dos que foram apresentados aqui e todos estão documentados na documentação oficial. O link para a documentação está na seção Links úteis deste capítulo e recomendo fortemente a leitura dela.

Conectado a um banco de dados externo

Os exemplos mostrados até agora pode ser executados no banco de dados H2, que é um banco que já vem embarcado no Spring Boot. Ele é muito útil no momento que estamos desenvolvendo nossa aplicação, mas quando a colocarmos para executar em produção, precisaremos utilizar um banco de dados mais robusto, com mais recursos. Um exemplo de banco muito utilizado é o MySql. Os passos a seguir mostram como configurar a conexão com um banco deste tipo.

No arquivo application.properties vamos adicionar algumas propriedades que serão resolvidas pelo Spring Boot e serão responsáveis em configurar a conexão com o

banco de dados externo. As propriedades abaixo são para o banco Mysql, mas você pode alterar para outro banco de dados.

```
spring.jpa.hibernate.ddl-auto=update
spring.datasource.url=jdbc:mysql://${MYSQL_HOST:localhost}:3306/db_example
spring.datasource.username=usuario
spring.datasource.password=senha
```

A primeira chave spring.jpa.hibernate.ddl-auto indica que o schema do banco de dados será atualizado quando ocorrer mudanças. A chave spring.datasource.url informa a url para conectar ao banco. As outras duas são respectivamente o nome do usuário e a senha para se conectar ao banco de dados.

Apenas esta configuração é o suficiente para fazer a aplicação se conectar a um banco de dados externo. Graças aos padrões seguidos pelo Spring, é possível até mesmo trocar um banco de dados sem ocorrer maiores impactos no código que foi desenvolvido.

Links Úteis

Documentação do Spring Data JPA:

https://docs.spring.io/spring-data/jpa/docs/current/reference/html

Tabela com as convenções dos nomes para Query Methods

https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#jpa.query-methods

Como acessar banco de dados MySql com Spring Boot:

https://spring.io/guides/gs/accessing-data-mysql/

Considerações Finais

Vimos neste capítulo:

- Como podemos obter grande facilidade nas tarefas que envolvam a camada de acesso a dados através do uso do Spring Data JPA.

No próximo capítulos veremos como aprimorar nossa aplicação isolando as regras de negócio.

Capítulo 3 - ISOLANDO REGRAS DE NEGÓCIO

Neste capítulo você irá:

- Entender a importância de separar sua aplicação em camadas para isolar as regras de negócio.
- Saber o motivo de se utilizar objetos DTO para separar o escopo das entidades com os objetos que serão retornados para o cliente.
- Conhecer os tipos de testes possíveis de serem escritos com as facilidades do Spring MVC.

Bean Validations

Bean Validation é a especificação padrão em Java para operações de validação de dados. Mais detalhes sobre ela pode ser consultado no site oficial https://beanvalidation.org. E o Spring Boot já fornece uma implementação para ela por padrão.

A grosso modo, o funcionamento do Bean Validation se dá pela definição de restrições adicionadas aos campos de uma classe. Fazemos isso adicionando algumas anotações específicas que ditam quais valores são aceitos para um certo atributo. Existem diversas anotações que definem restrições e uma lista completa com estas anotações pode ser encontrada em https://beanvalidation.org/2.0/spec/#builtinconstraints .

Depois de definida as restrições dos atributos, o objeto é testado por um validador que verifica se as restrições impostas aos atributos são satisfeitas. Caso algum valor de algum atributo não esteja válido, uma exceção será lançada.

Validando requisições no Controller

Existem três coisas que podemos validar em uma requisição HTTP:

- O corpo da requisição.
- Variáveis do path.
- Query parameters.

Validando corpo da requisição

Em métodos POST ou PUT, os objetos que são enviados pelos clientes podem ser verificados assim que chegam ao Controller. Então, podemos adicionar anotações nos atributos das classes que representam os recursos que são enviados, especificando as restrições que tornam aquele objeto válido. Por exemplo:

```
@Entity
public class Tarefa {

  @NotBlank
  private String descricao;

  ...
}
```

A anotação @NotBlank informa que o atributo nome não pode ser nulo e deve conter ao menos um caractere sem espaços vazios no início ou final do texto.

Na classe Controller, basta adicionar @Validated, para indicar que os métodos dela devem ter os parâmetros validados antes de serem executados.

```
@RestController
@RequestMapping("/tarefa")
@Validated
public class TarefaController {
   ...
}
```

E cada método que se deseja validar precisa ter a anotação @Valid antes dos parâmetros.

```
@PostMapping
public ResponseEntity<?> novaTarefa(
            @RequestBody @Valid TarefaRequest tarefaReq) {

  ...
}
```

Desta maneira, o Spring fará a validação dos dados do objeto antes de fazer qualquer outra coisa dentro do método.

Validando variáveis do path e query parameters

Este tipo de validação é um pouco diferente. Neste caso estaremos validando objetos simples, como inteiros ou strings. Não temos uma classe onde anotar um atributo, então adicionamos a anotação de restrição direto no parâmetro na classe Controller. Neste exemplo, a anotação @Min não permitirá valores menores que 1.

```
@GetMapping("/{id}")
public EntityModel<TarefaResponse> umaTarefa(
      @PathVariable @Min(1) Long id) {
  Tarefa tarefa = service.getTarefaPorId(id);
  return assembler.toModel(tarefa);
}
```

Caso um parâmetro esteja inválido, uma exceção ConstraintValidationException será lançada. O Spring não tem um exception handler padrão para este tipo de exceção, então ele acaba retornando para o cliente um HTTP Status 500 (Internal Server Error), quando o correto seria retornar 400 (Bad Request).

Para fazer isso, podemos implementar um método que seja capaz de tratar este tipo de exceção e registrá-lo como ExceptionHandler no classe anotada com @ControllerAdvice (Mais a frente veremos com mais detalhes a utilização de classes deste tipo. Por hora você precisa entender que são classes especiais que são capazes de interceptar requisições que resultaram em erro). Neste caso fazemos uma herança da nossa classe com a classe ResponseEntityExceptionHandler, que contém alguns métodos que podemos sobrescrever para tratar melhor alguns erros. Para tratar erros de validação, sobrescreveremos o método handleMethodArgumentNotValid:

```
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(
            MethodArgumentNotValidException ex,
            HttpHeaders headers, HttpStatus status, WebRequest request) {

  List<ValidationResponse> errors = ex.getBindingResult()
            .getFieldErrors()
            .stream()
            .map(x -> new ValidationResponse(x.getField(),
                  x.getDefaultMessage(), x.getRejectedValue()))
            .collect(Collectors.toList());

  Map<String, Object> body = new HashMap<>();
  body.put("errors", errors);

  return new ResponseEntity<>(
            body, headers, HttpStatus.BAD_REQUEST);
}
```

O código acima intercepta a execução de uma chamada a um método no controller que resultou em erro de validação, cria um objeto contendo as descrições dos erros de de cada campo e altera o Http Status para 400.

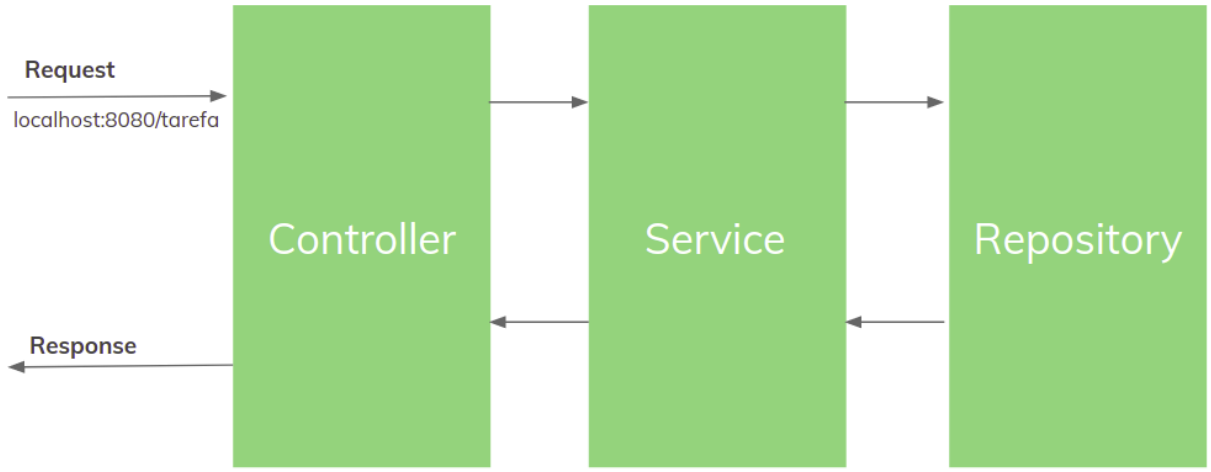
Service

É uma boa prática definir que classes do tipo Controllers não devem ter regras de negócio. Elas precisam ser simples, contendo apenas métodos para manipular dados de entrada das requisições, delegar ação para outra classe, que irá de fato executar as regras de negócio, e depois tratar o resultado obtido e formatar uma resposta a ser devolvida para o cliente.

Esta outra classe em questão são os chamados Services, que são classes especiais que desenvolvemos para centralizar as regras de negócio nelas. O Spring fornece a anotação @Service para transformar uma classe em um componente que poderá ser injetado em outras classes posteriormente.

De forma geral, uma aplicação é dividida em camadas, como ilustrado abaixo.

Figura 3.1 - Fluxo do dado em uma aplicação com Spring MVC



Fonte: O autor

Dividir a aplicação desta maneira é uma boa prática pois fornece maior isolamento entre as camadas, facilitando a manutenção do código e favorecendo o desenvolvimento de testes de unidade.

Para exemplificar, vamos imaginar o desenvolvimento de duas funcionalidades em um Web Service de gerenciamento de tarefas: completar e cancelar uma tarefa. Seu funcionamento se resume em alterar o status de uma tarefa, mas cada uma destas operações podem ter regras específicas, como por exemplo, não deixar que seu status seja alterado caso o status atual não seja “Em Andamento”.

```
@Service
public class TarefaService {
```

```
  @Autowired
  private TarefaRepository repository;

  public Tarefa cancelarTarefaPorId(Long id) {
    Tarefa tarefa = getTarefaPorId(id);

    if (!TarefaStatus.EM_ANDAMENTO.equals(tarefa.getStatus())) {

      throw new TarefaStatusException("Não é possível cancelar uma tarefa
                        completada");
    }

    tarefa.setStatus(TarefaStatus.CANCELADA);
    return salvar(tarefa);
  }

  public Tarefa completarTarefaPorId(Long id) {
    Tarefa tarefa = getTarefaPorId(id);

    if (!TarefaStatus.EM_ANDAMENTO.equals(tarefa.getStatus())) {
      throw new TarefaStatusException(
            "Não é possível completar uma tarefa cancelada");
    }

    tarefa.setStatus(TarefaStatus.TERMINADA);
    return salvar(tarefa);
  }

}
```

Como o desenvolvimento da regra foi feito na classe @Service, poderemos escrever testes para verificar se estão corretos sem a necessidade de executar a aplicação completa, não tendo a necessidade de subir todo o contexto do Spring. Isso agiliza muito o processo de desenvolvimento de software.

Injetando Service no RestController

Com o desenvolvimento da regra de negócio concluído, podemos fazer uso destes métodos na classe RestController. Para isso precisamos de uma instância da classe Service dentro do Controller. Vamos delegar para o Spring MVC fazer a instanciação do componente @Service e sua injeção na classe TarefaController.

```
@RestController
@RequestMapping("/tarefa")
@Validated
public class TarefaController {

  @Autowired
  private TarefaService service;
```

```
// ...
}
```

Veremos a técnica de Injeção de Dependência detalhadamente mais adiante. Mas basicamente, o que o Spring faz aqui é se encarregar de criar uma instância da classe TarefaService, carregá-la para seu contexto da aplicação em memória, e depois, irá atribuir esta instância dentro da classe TarefaController. Ele identifica que precisa fazer essa injeção através da anotação @Autowired.

Spring Tests

Uma boa prática no desenvolvimento de qualquer aplicação é a realização de testes automatizados. Estes tipos de testes são realizados escrevendo códigos na própria aplicação que servem para executar trechos isolados de códigos de negócio para verificar se ele está funcionando corretamente.

No Spring, podemos realizar estes testes utilizando as bibliotecas para testes que já são importadas no projeto no momento em que geramos o projeto pelo Spring Initializr.

Nós utilizaremos estas bibliotecas a partir de anotações em classes, como a @SpringBootTest e a @WebMvcTest.

A anotação @SpringBootTest faz o Spring procurar uma classe com a anotação @SpringBootApplication e usá-la para iniciar o contexto inteiro da aplicação.

O Spring traz recursos para que você possa testar seu código de maneira unitária (Unit Test), e testes de integração, testando códigos da camada Web (@WebMvcTest) e da camada de persistência (@DataJpaTest).

O teste mais básico que podemos fazer nesta camada serve para verificar se os componentes estão sendo carregados de forma correta durante a inicialização da aplicação. Este teste também é conhecido como Teste de Sanidade (Sanity Check). Apesar de simples, é extremamente útil, dando feedback rápido caso façamos algo no código que ocasione erro durante o deploy.

```
@SpringBootTest
class MinhasTarefasApplicationTests {

  @Autowired
  private TarefaController controller;

  @Test
  void contextLoads() {
    Assertions.assertThat(controller).isNotNull();
```

```
  }

}
```

Além deste teste, você pode escrever testes que verificam o comportamento da aplicação escrevendo um código que faça uma requisição HTTP e verificando se a resposta retornada é a esperada. O teste abaixo simula uma requisição com o método DELETE para o endpoint /tarefa/1.

```
@WebMvcTest(TarefaController.class)
public class TarefaControllerTest {

  @Autowired
  private MockMvc mvc;

  @MockBean
  private TarefaService service;

  @MockBean
  private TarefaModelAssembler assembler;

  @Test
  void quandoRequestForValidoRetornaSucesso() throws Exception {

    mvc.perform(
            MockMvcRequestBuilders.delete("/tarefa/1"))
       .andExpect(
            MockMvcResultMatchers.status().isNoContent());
  }
```

@SpringBootTest: Carrega todo contexto da aplicação. Mais lento.

@WebMvcTest: Não carrega todo contexto da aplicação, apenas a camada Web. Dentro do parênteses, podemos restringir ainda mais o contexto que é carregado, informando apenas o Controller que será testado.

@Autowired: Spring injeta uma instância do objeto.

@MockBean: cria e injeta um mock da classe Service na classe Controller. Sem esta anotação o contexto da aplicação não pode ser iniciado.

@Test: Indica que o método deve ser executado como teste.

Além destas anotações, usamos as classes MockMvcRequestBuilders e MockMvcResultMatchers que contém métodos que auxiliam na tarefa de fazer a requisição e verificar a resposta do serviço. Também precisamos injetar nesta classe as dependências para os objetos que são usados dentro da classe Controller. Isso é necessário para que o contexto do Spring possua uma instância delas na hora de

inicializar a classe. Como não queremos testar de fato o funcionamento destas dependências, as anotamos com @MockBean para que seja atribuído apenas um objeto fake.

Quando escrevemos testes de unidade, uma premissa importante a ser seguida é quanto ao tempo de execução de cada método. Eles precisam ser rápidos para gerar feedback o mais rápido possível para o desenvolvedor.

As anotações mostradas acima possuem características diferentes que interferem no modo em que os testes são executados, tornando-os mais lentos ou mais rápidos, por isso saber qual anotação utilizar em cada contexto pode tornar mais ou menos produtivo.

Se seu teste exige que todo contexto do Spring seja inicializado, neste caso você estará escrevendo um teste de integração entre as camadas Web, Service e Repository, por exemplo, então a anotação a ser utilizada é a @SpringBootTest. Esta anotação irá carregar todo contexto da aplicação, criando instância para cada componente anotado em seu código com @Service, @Component, @Repository, @Configuration, etc.

Agora você pode querer testar apenas a camada Web, onde ficam os controllers. Neste caso a anotação a ser utilizada é a @WebMvcTest. Com esta anotação o contexto será parcialmente carregado pelo Spring. Os componentes que são anotados com @Service, @Component, @Repository, @Configuration, etc. são ignorados. Por isso a necessidade de utilizar um recurso de Mock, para criar instâncias fakes destes componentes para o contexto do Spring injetar na classe que está sendo testada.

Outra observação importante a ser feita é quanto aos tipos de testes que devem ser escritos. No geral não faz muito sentido ficar escrevendo código de teste para testar operações como salvar entidade ou recuperar entidades do banco de dados utilizando os métodos padrões que são fornecidos pelo framework Spring Data por exemplo. Estas são responsabilidades dos próprio frameworks e seus desenvolvedores já fizeram toda esta carga de teste para certificar que operação de salvar estão funcionando corretamente. Você deve se atentar aos códigos que lidam com as regras de negócio do seu sistema, garantindo que eles continuem funcionando mesmo depois de alterações drásticas no código-fonte.

Isolar Entidades através de DTO's

Uma boa prática a se adotar no desenvolvimento de Web Service REST é a construção de objetos de transporte, ou Data Transfer Objects (DTO). Este objetos são utilizados para converter o dado que vem do banco de dados para um tipo de objeto contendo apenas os atributos que o cliente possa ver. Desta maneira, blindamos o Web Service de problemas relacionados à exposição de informação sigilosa por exemplo.

Para ilustrar, imagine uma entidade que representa os dados de um usuário. Digamos que esta entidade possua um atributo do tipo booleano que informa se ele é um administrador do sistema. Essa informação não deve ser exposta para o cliente, pois revela como a estrutura de dados está arquitetada. Se alguém souber desta informação, pode realizar requisições diretas para um endpoint tentando alterar o tipo de seu usuário.

Para resolver este problema, criamos duas classes DTO's, uma chamada UsuarioRequest e outra UsuarioResponse. Estas classes terão apenas os dados que podem ser recebidas e enviadas para o cliente.

```
public class UsuarioRequest {

  @NotBlank
  private String username;

  @NotBlank
  private String password;
...

public class UsuarioResonse {

  @NotBlank
  private String username;

  @NotBlank
  private String token;
...
```

Importante frisar que a responsabilidade em fazer as conversões de DTO para Entidade e vice-versa é da camada controller. A camada service deve continuar lidando com objetos do tipo Entidade mesmo.

Links úteis

Guia da documentação oficial mostrando como são desenvolvidos testes unitários com Spring MVC:

https://spring.io/guides/gs/testing-web/

Documentação do projeto Spring Test:

https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/testing.html#testing-introduction

Considerações Finais

Vimos neste capítulo:

- Como fazer validações de dados utilizando o Bean Validation.
- Como isolar as regras de negócio da aplicação através de componentes Service.
- Como testar uma aplicação utilizando o SpringTest.
- A importância de isolar o escopo das entidades dos objetos que são retornados para os clientes.

No próximo capítulos veremos como criar Web Services Restful com Spring MVC.

Capítulo 4 - RESTful COM SPRING MVC

Neste capítulo você irá:

- Entender o estilo de arquitetura RESTful.
- Aprender como desenvolver um Web Service RESTful com Spring MVC.

Estilo de arquitetura REST

REST é acrônimo para “REpresentational State Transfer”, ou Transferência de Estado Representativo em português. É um estilo de desenvolvimento de softwares para WEB. O termo foi cunhado por Roy Fielding em sua tese de doutorado, onde ele afirma que REST “enfatiza a escalabilidade das interações do componente, a generalidade das interfaces, a implementação independente de componentes, com componentes intermediários para diminuir a latência, garantindo segurança e encapsulando sistemas legados” (FIELDING, Roy - 2000).

Este estilo lista uma série de princípios que determinam uma maneira estruturada e sustentável para desenvolvimento de softwares para a chamada Web moderna. Utilizando REST, você tira proveito de toda infraestrutura já bem definida da Internet, fazendo melhor uso das especificações do protocolo HTTP, URI e métodos condizentes com cada operação que se deseja realizar. Essa padronização traz uma série de vantatens para os sistemas desenvolvidos.

URI e URL

URI vem de “Uniform Resource Identifier”, ou Identificador Uniforme de Recurso em português. Este é o mecanismo utilizado para dar um nome único às coisas na Internet. Por exemplo, https://www.amazon.com/index.html é o nome único da página inicial do site da Amazon.

Já URL, vem de “Uniform Resource Locator”, ou Localizador Uniforme de Recurso. Ele é utilizado para indicar a localização de coisas na Internet. Por exemplo, https://www.amazon.com é um endereço onde está localizado o site da Amazon.

Note que, toda URL pode ser considerado um URI, já que ele identifica um recurso, ou um local onde esteja o recurso. Mas nem toda URI pode ser considerada uma URL. Isso porque podem ser utilizadas para definir “namespaces” em arquivos XML por exemplo.

Recursos

Em REST tudo deve conter uma identificação, um ID. Como vimos no capítulo anterior, o mecanismo de identificação utilizado em sistemas Web é o URI. Portanto todas as coisas, ou recursos que serão disponibilizados, deverão ser alcançados através de uma URI.

JSON

"*JavaScript Object Notation*", ou simplesmente JSON, é um sintaxe textual baseado em JavaScript para armazenar e transportar dados. Por ser textual, tem a capacidade de ser compreendido tanto por humanos quanto por máquinas. É agnóstico à linguagem de programação, podendo ser convertido para objetos e vice-versa através de bibliotecas específicas.

Os dados no JSON são organizados através de pares chave/valor, sendo chave o nome de atributo e valor o conteúdo. Cada atributo é separado por vírgulas. Chaves delimitam quando começa e termina um objeto e colchetes é utilizado para arrays.

Um exemplo muito básico de JSON é: { "nome":"João da Silva" }

Note que tanto a chave quanto o valor são envoltos por aspas.

HATEOAS

Hypermedia as the Engine of Application State, ou HATEOAS. Em tradução livre seria algo como Hypermedia como o Motor de Estado da Aplicação.

E o que é Hypermedia? As páginas dos sites possuem textos e muitos outros links de recursos que são carregados automaticamente pelo navegador, como arquivos com scripts JavaScript, folhas de estilos CSS, imagens, sons, outras páginas, etc.

Estes links ditam para o navegador como esta página HTML deve ser renderizada, informando onde ele deve buscar os recursos necessários, como o ícone da página e a folha de estilo.

Figura 2.1 - Links em uma página html

```
<html>
<head>
  <link rel="icon" href="/img/favicon.ico"/>
  <link rel="stylesheet" href="/css/estilo.css" type="text/css"/>
</head>
```

Fonte: O autor

Estes links podem também serem utilizados nas respostas de seus serviços, para ditar aos clientes quais os próximos passos disponíveis para navegarem. Imagine que o cliente faça uma busca por um determinado livro (em um Web Service que retorna livros) a resposta do serviço irá trazer além dos dados do livro um link para a próxima ação disponível, que pode ser ou uma URI para que o usuário possa fazer a compra do livro, ou uma URI para que o usuário seja avisado quando o livro estiver disponível caso o exemplar já tenha sido esgotado. Este princípio traz muita dinâmica para a comunicação cliente servidor, pois o cliente não precisa saber de antemão quais passos ele pode seguir, o serviço irá informar.

HTTP Methods e Interface Uniforme

Este princípio do REST diz para utilizar a interface de métodos já disponibilizada pelo protocolo HTTP para fazer interações com seus serviços. Os quatro métodos mais utilizados são:

GET: para recuperar recurso.

POST: para criar um novo recurso.

PUT: para atualizar um recurso existente.

DELETE: para excluir um recurso.

Esses são os mais utilizados pois mapeiam muito bem as 4 operações básicas do banco de dados, ou CRUD (Create, Retrieve, Update e Delete).

Os métodos POST e PUT carregam em seu corpo de requisição os dados que serão inseridos ou atualizados no servidor. Os tipos de dados mais utilizados para este fim são o XML e o JSON. Todos os métodos retornam um response.

Códigos de respostas HTTP

Outro princípio de REST diz respeito sobre utilizar corretamente os códigos de resposta do HTTP. Quando uma requisição é feita para um serviço de maneira exitosa, um código 200 (“OK”) será retornado, ou qualquer outro da série 2xx. Mas se algo der errado, um código de erro das séries 3xx, 4xx ou 5xx será retornado.

O protocolo HTTP possui estas séries distintas de código para que o cliente saiba como lidar com a resposta do serviço. Cada série possui seu significado e deve ser utilizado de acordo com o resultado produzido em cada ação realizada.

Série 2xx

Códigos desta série indicam sucesso na requisição realizada. Os mais comuns são:

200 - OK

Operação realizada com sucesso.

201 - Created

Retornado após um POST, indica que o recurso foi criado corretamente. Retorna-se um cabeçalho Location juntamente com a resposta com a URI do recurso criado.

202 - Accepted

Utilizado para processamento assíncrono. Também retorna um cabeçalho Location.

204 - No content

Utilizado em POST e PUT quando o servidor não retorna dado algum.

Série 3xx

Esta série indica que deve se redirecionar a outro local para obter o recurso.

301 - Moved Permanently

Indica que o recurso procurado foi movido para outra URI. É retornado o cabeçalho Location indicando o local atualizado.

303 - See other

Indica que um processo assíncrono foi ou está sendo realizado e o recurso deve ser obtido em outro local. É retornado o cabeçalho Location indicando o local atualizado.

304 - Not modified

Utilizado em requisições GET, quando um recurso é cacheado. O serviço não retorna dados pois estes não sofreram modificações, indicando para o cliente utilizar o recurso pesquisado anteriormente.

307 - Temporary Redirect

Parecido com o 301, mas indica que o recurso foi movido temporariamente.

Série 4xx

Códigos desta série indicam que o cliente enviou dados errados na requisição.

400 - Bad Request

Código genérico indicando que ocorreu erro no processamento devido a qualquer erro nos dados enviados do cliente para o serviço.

401 - Unauthorized

Quando um cliente tenta acessar um recurso sem os dados de autenticação ou com autenticação inválida.

403 - Forbidden

Quando um cliente tenta acessar um recurso sem ter a permissão necessária.

404 - Not Found

Retornado quando o recurso solicitado não existe.

405 - Method Not Allowed

Quando o método utilizado na requisição não é suportado pelo serviço. Inclui o cabeçalho Allow indicando quais métodos podem ser utilizados.

409 - Conflict

Retornado em métodos POST, quando se tenta criar um recurso mas este recurso já existe. Retorna o cabeçalho Location indicando o local do recurso.

415 - Unsupported Media Type

Quando o cliente solicita ao serviço um tipo de dado que não é suportado.

Série 5xx

Códigos deste tipo referem-se a erros ocorridos por causa de problemas no servidor.

500 - Internal Server Error

Resposta genérica indicando que houve qualquer tipo de erro.

503 - Service Unavailable

Indica que o servidor está funcionando mas o serviço específico não está respondendo corretamente.

Uma lista completa dos códigos disponíveis no protocolo HTTP pode ser consultado através do link: https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html.

Por que usar RESTful?

O uso de estilo se popularizou devido aos benefícios trazidos pelo protocolo HTTP, como as ações bem definidas através de seus métodos (GET, POST, PUT, DELETE), sua segurança (criptografia e autenticação), desacoplamento entre cliente e servidor, entre outros.

Agora vamos ver como aplicar estes princípios na prática desenvolvendo um Web Service RESTful com Spring MVC. Será um serviço simples de controle de tarefas a fazer onde será possível demonstrar todos os princípios de RESTful escrito aqui.

RESTful com Spring MVC

O Spring MVC traz suporte para o desenvolvimento de Web Services RESTful através de componentes especializados. Nas próximas seções vamos ver cada um deles em ação no desenvolvimento de um serviço na prática.

@RestController

RestController é uma anotação fornecido pelo Spring MVC para facilitar a criação de Web Services RESTful. Ela é adicionada no topo da declaração de uma classe Java e faz com que a classe passe a ter a capacidade de responder à requisições feitas pelo protocolo HTTP.

Nesta classe, cada método precisa ser anotado com outras anotações para indicar qual método do HTTP o respectivo método será responsável em tratar. Cada

requisição tratada pelos métodos de um RestController serializam objetos automaticamente e retornam um HttpResponse.

```
@RestController
@RequestMapping("/tarefa")
public class TarefaController {

  @Autowired
  private TarefaService service;

  @PostMapping
  public Tarefa salvar(@RequestBody Tarefa tarefa) {
    return service.salvar(tarefa);
  }

  @GetMapping
  public List<Tarefa> pesquisa(Tarefa filtro) {
    return service.filtraPor(filtro);
  }

  @PutMapping("{id}")
  public Tarefa atualizar(@PathVariable Integer id,
                          @RequestBody Tarefa tarefa) {
    return service.atualizar(id, tarefa);
  }

  @DeleteMapping("{id}")
  public void excluir(@PathVariable Integer id) {
    service.excluir(id);
  }
}
```

O código acima mostra um exemplo de uma classe que foi anotada com @RestController tornando-a então um componente do Spring capaz de responder requisições como mencionado anteriormente.

Este código traz outras anotações importantes:

@RequestMapping

A anotação @RequestMapping, quando adicionada em uma classe, configura todos os métodos dela a responder às requisições realizadas para uma determinada rota, neste caso “/curso”. Os métodos das classes precisam ser anotados adequadamente para responder aos métodos específicos do HTTP.

@GetMapping

Anotação que indica que o método da classe irá mapear requisições do tipo HTTP GET.

@PostMapping

Anotação que indica que o método da classe irá mapear requisições do tipo HTTP POST.

@PutMapping

Anotação que indica que o método da classe irá mapear requisições do tipo HTTP PUT.

@DeleteMapping

Anotação que indica que o método da classe irá mapear requisições do tipo HTTP DELETE.

@Autowired

Anotação que permite ao Spring MVC identificar e injetar (em tempo de execução) instâncias de objetos em outros através do recurso de Injeção de Dependência.

Inversão de Controle e Injeção de dependência

Inversão de Controle é um princípio na engenharia de software onde o controle dos objetos utilizados por um software é controlado por um framework ou container. Este princípio traz algumas vantagens, como o desacoplamento, instanciação de diferentes implementações de classes, modulariza a aplicação, isolamento entre componentes, o que acaba facilitando a escrita e execução de testes de unidade, entre outros.

Esta inversão de controle pode ser realizada através de outro conceito conhecido como Injeção de Dependência.

A Injeção de Dependência pode ser entendido como a ação de conectar objetos com outros. Em uma classe normalmente temos relações com diversas outras classes, cada uma com sua responsabilidade. Normalmente compomos classes dependentes de outras para aproveitar a funcionalidade disponível em cada uma delas. Para utilizar estas classes relacionadas precisamos em algum momento instanciá-las, caso contrário teremos um erro ao acessar objeto nulo.

O exemplo mais simples de Injeção de Dependência é quando temos um construtor em nossa classe que recebe como argumento uma instância que será atribuída a um

atributo de nossa classe. No exemplo abaixo temos a representação de como ficaria a injeção de um objeto do tipo CursoService dentro do controller apresentado anteriormente.

```
public class CursoController {

  private CursoService service;

  public CursoController(CursoService service) {
    This.service = service;
  }

}
```

Neste exemplo CursoService pode ser uma interface, que aceita qualquer instância de sua implementação. O problema aqui é que quem for utilizar a classe CursoController, precisa antes de tudo instanciar um objeto do tipo CursoService para passar no construtor de CursoController no momento de sua utilização. Um trabalho um pouco mais complicado.

O Spring MVC facilita todo este processo através da anotação @Autowired. No momento da inicialização da aplicação, o Spring escaneia todas classes que estejam anotadas com @Component, @Service, @Configuration, @Repository, e outras, que são anotações que indicam se tratar de componentes do Spring. Quando Spring encontra estas classes, automaticamente cria uma instância para cada uma delas e deixa reservada na memória. Posteriormente, o Spring escaneia novamente as classes procurando pela anotação Autowired, quando encontra uma, verifica o tipo do atributo que está anotado, por exemplo, CursoService, e então injeta neste atributo uma instância correspondente de um objeto que ele criou no passo anterior.

Outras anotações importantes

O Spring traz uma série de anotações que facilitam muito a configuração da aplicação em vários aspectos, como por exemplo, tratar exceções ocorridas para mapear códigos de respostas mais adequados.

Tratando exceções com @ControllerAdvice

ControllerAdvice é uma anotação que faz uma classe compartilhar seus métodos com outras classes anotadas com @Controller.

Utilizamos @ControllerAdvice para centralizar o tratamento de exceções que podem ocorrer durante a execução de métodos dos Controllers. Dentro destas classes são

criados diversos métodos, cada um responsável em fornecer um tratamento adequado para cada exceção ocorrida na aplicação toda.

```
@ControllerAdvice
public class CustomGlobalExceptionHandler
extends ResponseEntityExceptionHandler {

  @ResponseBody
  @ExceptionHandler(EntityNotFoundException.class)
  @ResponseStatus(HttpStatus.NOT_FOUND)
  String entityNotFoundHandler(EntityNotFoundException ex) {
    return "Recurso não encontrado";
  }

  @ResponseBody
  @ExceptionHandler(TarefaStatusException.class)
  @ResponseStatus(HttpStatus.METHOD_NOT_ALLOWED)
  ResponseEntity<?> alteraStatusTarefaHandler(TarefaStatusException ex) {
    return ResponseEntity.status(HttpStatus.METHOD_NOT_ALLOWED)
            .header(HttpHeaders.CONTENT_TYPE,
                  MediaTypes.HTTP_PROBLEM_DETAILS_JSON_VALUE)
            .body(Problem.create().withTitle("Método não permitido")
            .withDetail("Você não pode realizar esta operação: " +
                  ex.getMessage()));
 }
}
```

A anotação **@ResponseBody** indica que o retorno do método deve ser associado ao corpo da resposta de uma requisição web. Ou seja, o objeto retornado será serializado em um tipo adequado para o cliente que fez a requisição a este recurso.

A anotação **@ExceptionHandler** indica qual exceção específica um determinado método irá tratar. No código acima, quando ocorrer uma exceção do tipo EntityNotFoundException, o método entityNotFoundHandler será invocado. Quando ocorrer uma exceção do tipo TarefaStatusException, o método alteraStatusTarefaHandler será invocdo. Dentro de cada método serão realizadas operações que irão criar uma resposta mais adequada para cada tipo de situação.

A anotação **@ResponseStatus** informa qual o código de status será retornado no cabeçalho de resposta do HTTP.

ResponseEntity é um tipo de objeto que representa a resposta a ser retornada formada por cabeçalho e corpo.

Links úteis

Lista completa dos HTTP Status Code:

https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html

Dissertação de Roy Fielding introduzindo os princípios REST

https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

Considerações Finais

Vimos neste capítulo:

- Como desenvolver um Web Service RESTful utilizando as facilidades do framework Spring MVC.

No próximo capítulos veremos como aprimorar nossa aplicação isolando as regras de negócio.

Capítulo 5 - SPRING SECURITY

Neste capítulo você irá:

- Compreender os processo de autenticação e autorização de um sistema.
- Conhecer o padrão OAuth.
- Aprender a gerar e validar tokens JWT.
- Desenvolver um fluxo de autenticação e autorização utilizando o Spring Security.

Autenticação e Autorização

Quando falamos em controle de acesso a uma aplicação, que é um entre vários requisitos de segurança que existem, estamos nos referindo a dois processos bases: autenticação e autorização. O primeiro está ligado a garantir a autenticidade da pessoa, ou seja, assegurar que ela é uma pessoa que está cadastrada previamente no sistema, enquanto o segundo verifica se uma pessoa já autenticada possui autorização para acessar o recurso que deseja.

Na autenticação, o usuário precisa fornecer suas credenciais (usuário e senha) para que o sistema possa verificar se ele existe no banco de dados. O sistema então consegue carregar dados adicionais deste usuário para um objeto que recebe o nome de *Principal*. Este, por sua vez, possui uma série de permissões de acesso associadas a ele, que são chamados de *Roles* ou *Authorities*. A partir destas informações que o sistema consegue realizar o processo de autenticação e autorização.

Estes processos são um dos mais críticos e importantes em qualquer aplicação. Em aplicações que possuem uma arquitetura cliente-servidor utilizando REST, usa-se autenticação baseada em Token seguindo a especificação OAuth2.

OAuth 2.0

OAuth é um protocolo aberto que permite fazer a segurança de aplicações web, mobile e desktop de maneira simplificada. Cria uma camada de segurança que separar a responsabilidade do cliente e do servidor no fluxo de autorização.

Um cliente precisa ser autenticado antes de acessar um recurso de um serviço que protegido. Para fazer a autenticação, o cliente envia suas credenciais que serão verificadas no banco de usuários do sistema. O sistema verificador então retorna um

token de acesso para o cliente, que deverá utilizá-lo toda vez que requisitar um recurso no serviço

Este token é assinado pelo servidor e possui um período de expiração, geralmente curto, para aumentar a segurança. O padrão mais utilizado de token utilizado atualmente é JWT.

JSON Web Token (JWT)

JSON Web Token (JWT) é um padrão aberto que define como deve ser transmitida informações entre duas partes de maneira segura utilizando objeto JSON. As informações contidas dentro destes objetos são confiáveis pois são assinadas digitalmente através de um algoritmo seguro.

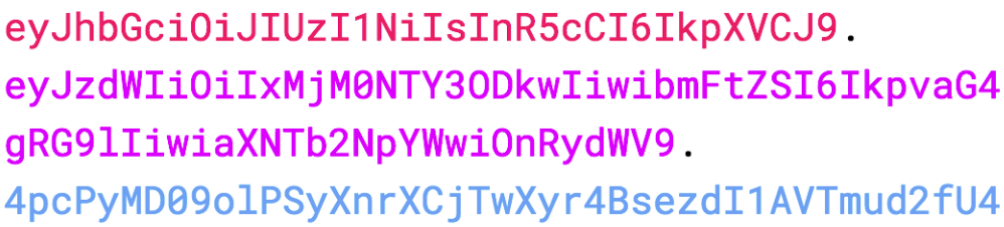
O Token JWT é dividido em três partes separados por pontos: **header.payload.signature**.

O header normalmente contém informações como o tipo do toke e o algoritmo utilizado para criptografar o token.

O payload é o corpo do token e carrega informações como dados do usuário.

O signature contém uma assinatura formada pelo header+payload criptografados (Base64) + uma chave secreta.

Figura 5.1 - JSON Web Token

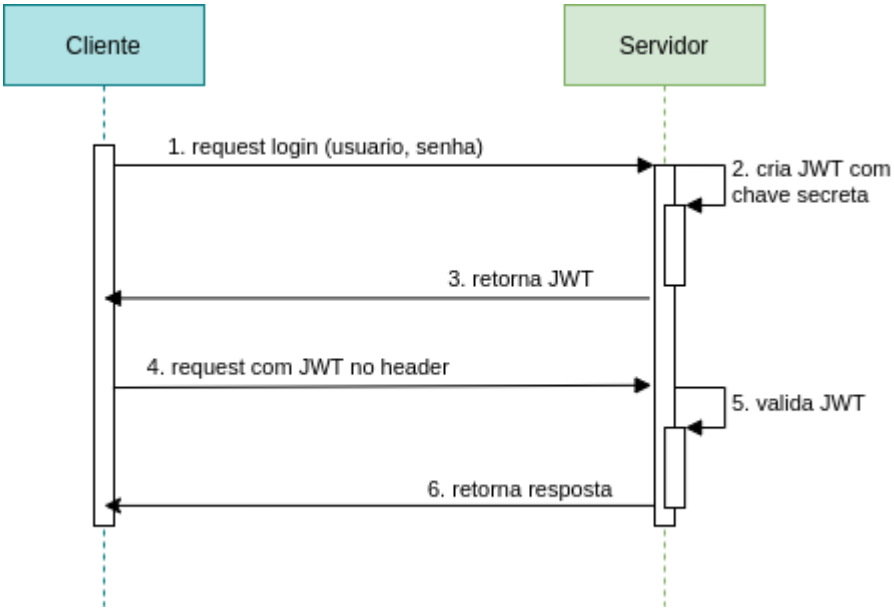


Fonte: https://jwt.io/introduction/

Fluxo de autenticação

O fluxo de autenticação com JWT inicia pelo cliente enviando ao servidor as credenciais do usuário, que irá vlidar se o usuário existe e que a senha está correta. Então o servidor então gera um token assinado com uma chave secreta que só ele conhece. Este token é enviado para o cliente e possui um tempo curto de expiração. Qualquer cliente é capaz de ver as informações deste token, mas só quem conhece a chave secreta consegue validá-lo.

Figura 5.2 - Fluxo de autenticação com JWT



Fonte: O autor

Nesse fluxo, o cliente fica responsável em manter o token obtido do servidor e enviar em cada requisição feita para algum recurso REST. O servidor não mantém o estado da sessão.

Spring Security na prática

O Spring Security é framework padrão de aplicações feitas em Spring para realizar autenticação de usuário e controle de acesso. Ele possui suporte aos padrões OAuth 2.0 e JWT vistos anteriormente.

O Spring Boot possui um starter que facilita adicionar as dependências necessárias no projeto. Uma vez adicionada, o sistema já estará protegido com uma configuração padrão, o que resta a ser feito são configurações personalizadas para permitir acesso às urls do serviço. A seguir será apresentada uma série de passos a seguir para adicionar o Spring Security na aplicação.

Dependências

Estas são as dependências necessárias para adicionar segurança na aplicação com Spring Boot.

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

<dependency>
  <groupId>io.jsonwebtoken</groupId>
```

```
  <artifactId>jjwt</artifactId>
  <version>0.9.1</version>
</dependency>
```

A primeira dependência é o starter do Spring Boot que contém todas dependências necessárias para desenvolver segurança na aplicação. A segunda é para manipular tokens JWT.

Configuração do Spring Security

Precisamos criar uma classe que centralizará as configurações do Spring Security. Vamos fazer uma extensão da classe WebSecurityConfigurerAdapter que já vem junto com o Spring Security, pois fornece alguns métodos que facilitam o desenvolvimento.

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

  @Autowired
  UserDetailsServiceImpl userDetailsService;

  @Autowired
  private AuthEntryPointJwt unauthorizedHandler;

  @Bean
  public AuthTokenFilter authenticationJwtTokenFilter() {
    return new AuthTokenFilter();
  }

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

  @Bean
  @Override
  public AuthenticationManager authenticationManagerBean() throws Exception
{
    return super.authenticationManagerBean();
  }

  @Override
  protected void configure(
      AuthenticationManagerBuilder authenticationManagerBuilder)
          throws Exception {

    authenticationManagerBuilder
      .userDetailsService(userDetailsService)
      .passwordEncoder(passwordEncoder());
  }

  @Override
  protected void configure(HttpSecurity http) throws Exception {
    http.csrf().disable()
```

```
      .exceptionHandling()
        .authenticationEntryPoint(unauthorizedHandler)
      .and()
        .sessionManagement()
        .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
      .and()
        .authorizeRequests()
        .antMatchers("/api/auth/**")
          .permitAll()
        .antMatchers("/tarefa/**")
          .hasAnyRole("USER", "ADMIN")
        .antMatchers("/h2-console/**")
          .permitAll()
        .antMatchers("/api/test/**")
          .permitAll()
        .anyRequest().authenticated()
      .and()
        .addFilterBefore(authenticationJwtTokenFilter(),
            UsernamePasswordAuthenticationFilter.class);
  }

}
```

@EnableWebSecuriy: Spring encontra e aplica a classe como configuração global.

@EnableGlobalMethodSecurity: permite utilizar as anotações @PreAuthorize e @PostAuthorize nos métodos.

O método configure(AuthenticationManagerBuilder) é responsável em verificar se o usuário existe no banco de dados e se a senha está correta, ou seja, fará o processo de autenticação do usuário. Já o método configure(HttpSecurity) é responsável pela autorização, já que define as regras de quais recursos o usuário que foi autenticado poderá acessar.

A próxima classe que vamos criar é a UserDetailsServiceImpl que implementa a interface do Spring UserDetailsService, que contém apenas um método, loadUserByUsername, onde vamos implementar a busca do usuário no repositório.

```
@Service
public class UserDetailsServiceImpl
      implements UserDetailsService {

  @Autowired
  UsuarioRepository userRepository;


  @Transactional
  public UserDetails loadUserByUsername(String username)
       throws UsernameNotFoundException {

    Usuario user = userRepository
            .findByUsername(username)
            .orElseThrow(() -> new UsernameNotFoundException(
```

```
                    "Usuário não encontrado"));

    return UserDetailsImpl.build(user);
  }

}
```

Para buscar o usuário no repositório vamos implementar a clase UsuarioRepository com o método findByUsername.

```
@Repository
public interface UsuarioRepository
            extends JpaRepository<Usuario, Integer> {

  Optional<Usuario> findByUsername(String username);

  Boolean existsByUsername(String username);

}
```

Vamos criar a Entidade Usuario para armazenar o usuário no banco de dados.

```
@Entity
@Table(name = "usuarios", uniqueConstraints =
{@UniqueConstraint(columnNames = "username")})
public class Usuario {

  @Id
  @GeneratedValue
  private Integer id;

  @NotBlank
  @Size(max = 20)
  private String username;

  @NotBlank
  @Size(max = 120)
  private String password;

  @ManyToMany(fetch = FetchType.LAZY)
  @JoinTable(name = "user_roles",
      joinColumns = @JoinColumn(name = "user_id"),
      inverseJoinColumns = @JoinColumn(name = "role_id"))
  private Set<Role> roles = new HashSet<Role>();

  public Usuario() {}

  public Usuario(String username, String password) {
    this.username = username;
    this.password = password;
  }

 // ...

}
```

Também vamos criar a entidade Role para armazenar as funções do usuário.

```
@Entity
@Table(name = "roles")
public class Role {
```

```
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @Enumerated(EnumType.STRING)
  @Column(length = 20)
  private ERole name;

  public Role() {}

  public Role(ERole name) {
    this.name = name;
  }

 // ...

}
```

O Enum ERole:

```
public enum ERole {

  ROLE_USER, ROLE_ADMIN

}
```

Voltando para classe UserDetailsServiceImpl, o método loadUserByUsername precisa retornar um objeto do tipo UserDetails. Então vamos criar a classe UserDetailsImpl que utilizará os dados do usuário que foi recuperado do banco de dados.

```
public class UserDetailsImpl implements UserDetails {

  private Integer id;

  private String username;

  @JsonIgnore
  private String password;

  private Collection<? extends GrantedAuthority> authorities;

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

  public static UserDetailsImpl build(Usuario user) {
    List<GrantedAuthority> authorities = user.getRoles().stream()
        .map(role -> new SimpleGrantedAuthority(role.getName().name()))
        .collect(Collectors.toList());

    return new UserDetailsImpl(
        user.getId(),
        user.getUsername(),
        user.getPassword(),
        authorities);
  }
```

```
  // ...
}
```

E então podemos fazer as configurações específicas para o JWT. Primeiro criando a classe JwtUtils que conterá os métodos para gerar token, validar um token e pegar o nome do usuário de um token.

```
@Component
public class JwtUtils {

  @Value("${app.jwt.SecretKey}")
  private String jwtSecret;

  @Value("${app.jwt.ExpirationMs}")
  private Integer jwtExpirationMs;

  public String generateJwtToken(Authentication authentication) {
    UserDetailsImpl userPrincipal =
        (UserDetailsImpl) authentication.getPrincipal();

    Date currentTime = new Date();
    Date expirationTime = new Date(
        currentTime.getTime() + jwtExpirationMs);

    return Jwts.builder()
        .setSubject(userPrincipal.getUsername())
        .setIssuedAt(currentTime)
        .setExpiration(expirationTime)
        .signWith(SignatureAlgorithm.HS512, jwtSecret)
        .compact();
  }

  public String getUserNameFromJwtToken(String token) {
    return Jwts.parser()
        .setSigningKey(jwtSecret)
        .parseClaimsJws(token)
        .getBody()
        .getSubject();
  }

  public boolean validateJwtToken(String authToken) {
    try {
      Jwts.parser().setSigningKey(jwtSecret).parseClaimsJws(authToken);
      return true;
    } catch (Exception e) {
      return false;
    }
  }
}
```

No arquivo application.properties adicionamos duas propriedades, para guardar a chave secreta e outra para determinar o tempo de expiração do token.

```
app.jwt.SecretKey= ChaveSecreta

app.jwt.ExpirationMs= 28800000
```

Precisamos agora criar um filtro Servlet que irá interceptar toda requisição realizada. Vamos chamá-lo de AuthTokenFilter e deve estender de OncePerRequestFilter.

```
public class AuthTokenFilter extends OncePerRequestFilter {

  @Autowired
  private JwtUtils jwtUtils;

  @Autowired
  private UserDetailsServiceImpl userDetailsService;

  @Override
  protected void doFilterInternal(
      HttpServletRequest request,
      HttpServletResponse response,
      FilterChain filterChain)
      throws ServletException, IOException {

    String jwt = parseJwt(request);

    if (jwt != null && jwtUtils.validateJwtToken(jwt)) {
      String username = jwtUtils.getUserNameFromJwtToken(jwt);

      UserDetails userDetails = userDetailsService
            .loadUserByUsername(username);

      UsernamePasswordAuthenticationToken authentication =
          new UsernamePasswordAuthenticationToken(
          userDetails, null, userDetails.getAuthorities());

      authentication.setDetails(new WebAuthenticationDetailsSource()
            .buildDetails(request));

      SecurityContextHolder.getContext().setAuthentication(authentication);
    }

    filterChain.doFilter(request, response);

  }

  private String parseJwt(HttpServletRequest request) {
    String headerAuth = request.getHeader("Authorization");

    if (StringUtils.hasText(headerAuth) && headerAuth.startsWith("Bearer"))
      return headerAuth.split(" ")[1];

    return null;
  }

}
```

Para finalizar a configuração do Spring Security, vamos adicionar a classe AuthEntryPointJwt que será um ExceptionHandler responsável em interceptar uma requisição de usuário não autenticado e mudar o HttpStatus para 401.

```
@Component
public class AuthEntryPointJwt implements AuthenticationEntryPoint {
```

```
  @Override
  public void commence(
      HttpServletRequest request,
      HttpServletResponse response,
      AuthenticationException authException)
            throws IOException, ServletException {

    response.sendError(
      HttpServletResponse.SC_UNAUTHORIZED, "Error: Unauthorized");
  }

}
```

As classes a seguir são para tratar do cadastro e autenticação do usuário.

UsuarioController:

```
@RestController
@RequestMapping("/api/auth")
public class UsuarioController {

  @Autowired
  private UsuarioService authService;


  @PostMapping("/signin")
  public ResponseEntity<?> signin(
      @Valid @RequestBody LoginRequest loginRequest) {

    JwtResponse jwtResponse = authService.authenticateUser(
        loginRequest.getUsername(), loginRequest.getPassword());

    return ResponseEntity.ok(jwtResponse);
  }

  @PostMapping("/signup")
  public ResponseEntity<?> signup(
      @Valid @RequestBody SignupRequest signupRequest) {
    authService.registerUser(
        signupRequest.getUsername(),
        signupRequest.getPassword(),
        signupRequest.getConfirmPassword(),
        signupRequest.getRoles());

    return ResponseEntity.ok("Usuário registrado com sucesso");
  }

}
```

Os DTO's:

```
public class LoginRequest {

  @NotBlank
  private String username;

  @NotBlank
  private String password;
 // ...
}

public class SignupRequest {
```

```
  private String username;

  private String password;

  private String confirmPassword;

  private Set<String> roles;
 //...
}


public class JwtResponse {

  private String token;
  private String type = "Bearer";
  private Integer id;
  private String username;
  private List<String> roles;

  public JwtResponse(String token, Integer id, String username,
List<String> roles) {
    this.token = token;
    this.id = id;
    this.username = username;
    this.roles = roles;
  }

  public boolean isAdmin() {
    return roles.contains(ERole.ROLE_ADMIN.name());
  }
 // ...
}
```

A classe UsuarioService que centraliza as regras para cadastrar usuário.

```
@Service
public class UsuarioService {

  @Autowired
  AuthenticationManager authenticationManager;

  @Autowired
  UsuarioRepository userRepository;

  @Autowired
  RoleRepository roleRepository;

  @Autowired
  PasswordEncoder passwordEncoder;

  @Autowired
  JwtUtils jwtUtils;

  public JwtResponse authenticateUser(String username, String password) {
    Authentication authentication = authenticationManager.authenticate(
        new UsernamePasswordAuthenticationToken(username, password));

    SecurityContextHolder.getContext().setAuthentication(authentication);
    String jwt = jwtUtils.generateJwtToken(authentication);

    UserDetailsImpl userDetails = (UserDetailsImpl)
```

```
        authentication.getPrincipal();

    List<String> roles = userDetails.getAuthorities().stream()
        .map(item -> item.getAuthority())
        .collect(Collectors.toList());

    return new JwtResponse(jwt, userDetails.getId(),
        userDetails.getUsername(), roles);
  }

  public void registerUser(String username,String password, String
confirmPassword, Set<String> strRoles) {

    if (userRepository.existsByUsername(username)) {
      throw new EntityExistsException("Usuário já existe");
    }

    Usuario user = new Usuario(username, passwordEncoder.encode(password));

    Set<Role> roles = new HashSet<Role>();

    if (strRoles == null) {
      Role userRole = roleRepository.findByName(ERole.ROLE_USER)
          .orElseThrow(() -> new EntityNotFoundException(
              "Role não encontrada"));
      roles.add(userRole);
    } else {
      strRoles.forEach(role -> {
        switch (role) {
        case "admin":
          Role adminRole = roleRepository.findByName(ERole.ROLE_ADMIN)
          .orElseThrow(() -> new EntityNotFoundException(
              "Role não encontrada"));
          roles.add(adminRole);
          break;

        default:
          Role userRole = roleRepository.findByName(ERole.ROLE_USER)
          .orElseThrow(() -> new EntityNotFoundException(
              "Role não encontrada"));
          roles.add(userRole);
        }
      });
    }

    user.setRoles(roles);
    userRepository.save(user);
  }

}
```

E um repositório para gerenciar as Roles.

```
public interface RoleRepository extends JpaRepository<Role, Integer> {

  Optional<Role> findByName(ERole name);

}
```

Configurar o Spring Security em uma aplicação REST não é tão simples como podemos ver, mas é um passo essencial para garantir a segurança.

Links úteis

Padrão OAuth:

https://oauth.net/2/

Mais sobre o JWT:

https://jwt.io/

Página do projeto Spring Security:

https://spring.io/projects/spring-security

Considerações Finais

Vimos neste capítulo:

- Como adicionar segurança a um Web Service RESTful utilizando o Spring Security, OAuth e JWT.

Referências

BOAGLIO, Fernando. Spring Boot - Acelere o desenvolvimento de microsserviços. São Paulo: Casa do Código, 2018. 154p (Livros para o programador).

FIELDING, Roy Thomas. Architectural Styles and the Design of Network-based Software Architectures. 2000. Disponível em: https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf.

WEISSMANN, Henrique Lobo. Vire o jogo com Spring Framework. São Paulo: Casa do Código, 2018. 314p (Livros para o programador).

ACCESSING DATA WITH JPA. Spring.io, 2020. Disponível em: https://spring.io/guides/gs/accessing-data-jpa. Acesso em: 17 out. 2020.

SPRING BOOT - REFERENCE DOCUMENTATION. Spring.io, 2020. Disponível em: https://docs.spring.io/spring-boot/docs/current/reference/html/. Acesso em: 17 out. 2020.

BUILDING REST SERVICES WITH SPRING. Spring.io, 2020. Disponível em: https://spring.io/guides/tutorials/rest/. Acesso em: 17 out. 2020.

EXCEPTION HANDLING IN SPRING MVC Spring.io, 2020. Disponível em: https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc. Acesso em: 17 out. 2020.

SPRING DATA JPA - REFERENCE DOCUMENTATION. Spring.io, 2020. Disponível em: https://docs.spring.io/spring-data/jpa/docs/current/reference/html. Acesso em: 17 out. 2020.

TESTING THE WEB LAYER. Spring.io, 2020. Disponível em: https://spring.io/guides/gs/testing-web. Acesso em: 17 out. 2020.

WEB ON SERVLET STACK. Spring.io, 2020. Disponível em: https://docs.spring.io/spring-framework/docs/current/spring-framework-reference/web.html. Acesso em: 17 out. 2020.

WORKING WITH SPRING DATA REPOSITORIES. Spring.io, 2020. Disponível em:https://docs.spring.io/spring-data/jpa/docs/current/reference/html/#repositories. Acesso em: 17 out. 2020.