

Anotações Spring Annotations

Inversão de controle:

É um padrão de projeto, é algo abstrato onde a gente define todas as dependências de um determinado objeto sem a necessidade de criar (gerenciar), pois passamos esse papel de gerenciamento para o framework no caso o **Spring** para o seu **Core**, então é o **Core** do **Spring** que vai ter toda essa responsabilidade de gerenciar todas essas dependências conforme necessidade.

E como o **Spring** faz isso então? Ele utiliza de uma implementação, a forma concreta que é a **injeção de dependências**, então **injeção de dependências** é a implementação concreta da **inversão de controle**, então assim ele consegue gerenciar todos esses **Beans** que são os objetos que vamos criando conforme vamos construindo as nossas aplicações ou vamos utilizando de bibliotecas externas por exemplo, vamos definindo esses **Beans** e assim o **Spring** ele vai cuidar das instâncias de todos esses objetos **Java**.

Tudo que envolve a base do Spring está contido no seu Core que fica dentro do projeto Spring Framework, que é aonde já traz então várias configurações prontas, muitas delas a gente já consegue usar de forma bem mais simples, a possibilidade de customizar e de criar todas as configurações necessárias para nossa aplicação.

Principais Anotações:

Stereotype:

As anotações de **Stereotypes** servem para nos mostrarmos as classes que o **Spring** deve gerenciar, e assim o **Spring** consegue verificar que essas **classes** vão ser os **Beans** que ele vai gerenciar.

@Component: Quando executarmos a aplicação o **Spring** vai detectar quais são os **objetos** que estão com essa anotação e esses objetos vão ser gerenciados por ele e também ele vai injetar essas dependências aonde for necessário e quando necessário. Ela é uma anotação genérica, então qualquer **Bean** que quisermos detalhar/especificar para o **Spring** podemos utilizar dessa anotação **@Component**, mas quando estamos desenvolvendo nossa aplicação fica muito mais sugestivo a gente já utilizar de **Stereotypes** específicos para cada responsabilidade (**@Service**, **@Controller**,

@Repository), função de determinada classe e assim também fica muito mais fácil para outros desenvolvedores quando forem trabalhar no nosso código ele quando ver a classe já entender qual a função dela.

@Service: Tem regras/logicas de negócios.

@Repository: Tem lógica de negócios do banco de dados, transações, lógica de banco, etc.

@Controller: Específica para **end-points**, para expormos os **end-points** da nossa aplicação web.

Core:

É a base *coração* do **Spring Framework** onde estão diversas das anotações que utilizamos para configuração deste framework e também para obtermos todo este suporte da inversão de controle com a injeção de dependências e definir todos esses **Beans** para que o **Spring** consiga então gerenciar todo o seu ciclo de vida entre outras coisas.

Beans:

@Autowired: Quando definimos que certas classes, objetos vão ser **Beans** para o **Spring** gerenciar ele já sabe quais vão ser as classes e quais São seus **Stereotypes**, já que ele sabe quais são essas definições de **Beans** temos que sinalizar de uma certa forma aonde que ele vai injetar essas instâncias quando necessárias, no **controller** por exemplo se usarmos algum **@Service** ou seja se usarmos uma classe **@Service** dentro do **controller** para obtermos por exemplo métodos de **findById**, **findAll**, temos que injetar o **Service** de alguma forma seja por **constructor** ou por **set**, porém tem um método melhor e menos verboso que é usar a anotação **@Autowired**.

Exemplo de injeção do **Service** no **controller** por meio de **constructor** para termos os métodos do **Service** nos nossos métodos **GET**, **PUT**, **POST**, **DELETE**:

```
final ParkingSpotService parkingSpotService;  
  
public ParkingSpotController(ParkingSpotService parkingSpotService) {  
    this.parkingSpotService = parkingSpotService;  
}
```

Exemplo de injeção do **Service** no **controller** por meio da anotação **@Autowired** para termos os métodos do **Service** nos nossos

métodos GET, PUT, POST, DELETE:

```
@Autowired  
private ParkingSpotService parkingSpotService;
```

Então agora sempre que necessário o **Spring** vai criar essa injeção, vai injetar o **Bean SERVICE** dentro do **Bean Controller** e assim a classe **Controller** vai conseguir utilizar de todos os métodos deste **Service**.

@Qualifier: Quando temos por exemplo um **Service** que é implementado por mais de uma classe **Service** o **Spring** acaba gerando um erro por não saber qual **Bean** ele vai injetar no **@Autowired** por exemplo do **controller** acima, logo para resolver este erro usamos da anotação **@Qualifier** com o valor do **Bean** sem a primeira letra ser maiúscula para mostrar para o **Spring** qual que é **Bean** que ele vai levar em consideração quando ele for injetar essa dependência.

```
@Autowired  
@Qualifier("parkingSpotServiceImpl")  
private ParkingSpotService parkingSpotService;
```

Com isso estamos dizendo para o **Spring** que na hora que ele for criar este ponto de injeção (injetar as dependências) neste ponto de injeção que estamos criando ele vai considerar a implementação **parkingSpotServiceImpl** da nossa interface **ParkingSpotService**.

@Value: Para explicar sobre o **@Value** podemos olhar para o **application.properties**, muitas vezes precisamos definir variáveis, propriedades que utilizamos dentro do código, mas ao invés de deixarmos isso fixo dentro de um **controller** por exemplo ou um **Service** temos a opção e boa prática de colocar (declarar) essas propriedades dentro de um **properties** assim fica muito mais fácil depois se no caso precisarmos alterar ou qualquer outra modificação, pois não precisamos mexer dentro do código somente apenas nesses arquivos de propriedades, e ainda mais se tivermos utilizando arquitetura de **micro-services** e estivermos contemplando o **pattern** que é o **global config** onde temos um serviço específico que faz o gerenciamento de todos esses arquivos de propriedades, então por exemplo podemos criar variáveis, propriedades, que vão estar com o **global config** para ele gerenciar e assim podemos na maioria dos casos

alterar o valor nos arquivos de propriedades sem precisar até de parar a aplicação.

Exemplo de propriedades e seus valores sendo definidos no properties:

```
app.name=Parking Control API
app.port=80
app.host=parkingcontrolapp
```

Eles sendo implementada em alguma classe:

```
@Value("${app.name}")
private String appName;

@Value("${app.port}")
private String appPort;

@Value("${app.host}")
private String appHost;
```

Ou seja esses atributos recebem o valor das propriedades que estão no **properties** por causa do **value** e podem fazer normalmente o que os atributos fazem só que estarão com o valor definido no **properties**.

Context:

@Configuration: Em classes de configuração usamos essa anotação, essas classes são aonde podemos configurar dados de forma global, declarar **Beans**, etc. O **Spring** sempre que inicia a aplicação ele vai olhar para essa classe e levar em consideração todas as customizações, configurações que fizemos dentro dela.

Por ela é possível utilizarmos da classe de configuração com essa anotação para declarar **Beans** que é uma forma de declarar **Beans** sem usar **Stereotypes**.

@ComponentScan: Essa anotação podemos não dar valor a ela que aí ela levará em conta o pacote raiz principal quando colocamos a anotação em uma classe raiz ou podemos definir determinados pacotes, tanto definir quanto excluir, mas o que vai ser essa definição ou exclusão desses pacotes que estamos passando dentro dessa anotação? Estamos mostrando para o **Spring** utilizando dessa anotação quais são os pacotes que ele vai fazer uma varredura ou seja vai buscar pelos **Beans** que ele vai gerenciar, então podemos colocar o pacote raiz que aí ele busca pela aplicação toda ou podemos especificar alguns pacotes que definimos certos **Beans** ou tem casos que podemos excluir pacotes,

ou seja determinados pacotes ele não vai fazer a varredura (se tiver **Bean** lá dentro ele não vai considerar). O **Spring** já tem essa anotação internamente ao iniciar, porém ela verifica o projeto todo.

@Bean: Usamos a anotação **@Bean** quando queremos declarar o **Bean** na classe de configuração, assim criamos ele e chamamos os métodos que gostaríamos dele usando a anotação em cima **@Bean**

```
@Configuration
public class BeanConfig {

    @Bean
    public MyBean myBean(){
        return new MyBean();
    }

    @Bean
    public IMapper mapper() {
        return new IMapper();
    }

}
```

Podemos fazer isso tanto com **Beans** criados por nós ou como **Beans** já prontos de outras bibliotecas.

Para injetar ele em alguma classe para uso, como no **controller** seria assim:

```
@Autowired
private IMapper mapper;
```

@Lazy: Quando criamos um **Bean** por padrão o **Spring** trata-o como um **Bean** ansioso, então ele sempre vai criar para deixar esse **Beans** disponíveis e aí eles são iniciados junto com toda a aplicação, mas tem casos que vamos querer um **start** preguiçoso, ou seja vamos falar para o **Spring** criar esse **Bean** apenas quando precisar de algum método ou uma classe precisar deste **Bean** e ele for injetado, então o **Spring** ele vai saber que ele não vai criar sempre que a aplicação subir, mas apenas quando aquele **Bean** for acionado e for necessário ser injetado em algum local.

Para isso vamos na classe do **Bean** e anotamos com **@Lazy**:

```

@Component
@Lazy
public class LazyBean {

    public LazyBean() {
        System.out.println("LazyBean started!!!");
    }

}

```

@Primary: Indica qual o **Bean** o **Spring** tem que considerar quando temos duas implementações de uma **interface** e **precisamos injetar ela em alguma outra classe**, então escolhemos a implementação que queremos como **Bean** e adicionamos a anotação **@Primary** nela. Esse é um **outro método de escolher o Bean** que vai ser levado em consideração pelo o **Spring** outro método para isso é a anotação **@Qualifier**.

@Scope: O **Scope** ele vai definir como vai ser criado/definido o ciclo de vida do **Bean** em questão (qual será o escopo em questão). **Tipos de Scope**

Singleton: Inicia o **Bean** de forma com que ele inicie com a aplicação e sempre estará disponível.

Prototype: Inicia somente quando o **Bean** em questão é chamado.

Request:

```
<bean id="loginAction" class="com.foo.LoginAction" scope="request"/>
```

Com a definição de **bean** acima em vigor, o contêiner **Spring** criará uma nova instância do **bean** **LoginAction** usando a definição de **bean** 'loginAction' para cada solicitação **HTTP**. Ou seja, o **bean** 'loginAction' será efetivamente definido no nível de solicitação **HTTP**. Você pode alterar ou sujar o estado interno da instância que é criada o quanto quiser, sabendo que **outras solicitações que também estão usando instâncias criadas na parte de trás da mesma definição de bean 'loginAction' não verão essas alterações no estado**, uma vez que são específicos de uma solicitação individual. Quando o processamento da solicitação terminar, o **bean** com escopo definido para a solicitação será descartado.

Session: Define o escopo de uma única definição de **bean** para o ciclo de vida de uma sessão **HTTP**. Válido apenas no contexto de um **Spring ApplicationContext** com reconhecimento da **Web**.

```
<bean id="userPreferences" class="com.foo.UserPreferences" scope="session"/>
```

Com a definição de **bean** acima em vigor, o contêiner **Spring** criará uma nova instância do **bean** **UserPreferences** usando a definição de **bean** 'userPreferences' para o tempo de vida de uma única sessão **HTTP**. Em outras palavras, o **bean** 'userPreferences' **será efetivamente definido no nível da sessão HTTP**. Assim como os **beans** com escopo de solicitação (**request**), você pode alterar o estado interno da instância que é criada o quanto quiser, sabendo que outras instâncias de sessão **HTTP** que também estão usando instâncias criadas na parte de trás do mesmo **bean** 'userPreferences' **A definição não verá essas alterações no estado, pois elas são específicas de uma sessão HTTP individual**. Quando a Sessão **HTTP** for finalmente descartada, o **bean** com escopo para essa Sessão **HTTP** específica também será descartado.

Global Session: Define o escopo de uma única definição de **bean** para o ciclo de vida de uma sessão **HTTP** global. Geralmente válido apenas quando usado em um contexto de **portlet**. Válido apenas no contexto de um **Spring ApplicationContext** com reconhecimento da **Web**.

O escopo da **global Session** é semelhante ao escopo da Sessão **HTTP** padrão (descrito imediatamente acima) e realmente só faz sentido no contexto de aplicativos da **web** baseados em **portlet**. A especificação do **portlet** define a noção de uma **Sessão global** que é compartilhada entre todos os vários **portlets** que compõem um único aplicativo da **web** de **portlet**. Os **beans** definidos no escopo da sessão global são definidos (ou vinculados) ao tempo de vida do **portlet global Session**.

(Observe que, se você estiver escrevendo um aplicativo da Web baseado em Servlet padrão e definir um ou mais beans como tendo escopo de sessão global, o escopo de sessão HTTP padrão será usado e nenhum erro será gerado.)

@PropertySource: Dentro da pasta **resources** dos nossos projetos em **Spring** temos o **application.properties** que nele temos as nossas **configurações**, mas se quisermos criar outro arquivo **properties** customizado temos que usar essa anotação, caso contrário não será

carregado o arquivo `properties` que criamos e somente o `application.properties` já que ele é inicializado e reconhecido pelo **Spring** por padrão.

@PropertySources: Faz a mesma coisa do **@PropertySource**, porém ele consegue indicar mais arquivos `properties` e não somente um.

@Profile: Muito usado para distinguirmos os valores/propriedades quando temos múltiplos ambientes, por exemplo temos o ambiente de desenvolvimento, produção, homologação, entre outros. Então para podermos distinguir os valores dessa propriedade ou os `Beans` de cada um desses ambientes podemos utilizar tanto a nível de arquivos de propriedades que é outra forma, mas temos essa anotação que é o **@Profile** para podermos especificar os perfis que vão ser ativos.

```
@Configuration
public class BeanConfig {

    @Profile("dev")
    @Bean
    public MyBean myBeanDevProfile(){
        System.out.println("Profile DEV Started!");
        return new MyBean();
    }

    @Profile("prod")
    @Bean
    public MyBean myBeanProdProfile(){
        System.out.println("Profile PROD Started!");
        return new MyBean();
    }
}
```

Se iniciarmos assim irá gerar um conflito, pois não especificamos para o **Spring** qual é o perfil que está ativo, para declarar basta adicionarmos o perfil que queremos deixar ativo no nosso `application.properties` com esse comando: `spring.profiles.active=(perfil)`.

Ou podemos usar isso a nível de classe (uma classe para cada perfil).


```
8
9 @Profile("prod")
10 @Configuration
11 public class ProdBeanConfig {
12
13     @Bean
14     public MyBean myBeanProdProfile(){
15         System.out.println("Profile PROD Started!");
16         return new MyBean();
17     }
18
19     @Bean
20     public ModelMapper modelMapper() { return new ModelMapper(); }
21
22
23
24 }
```

Boot:

@SpringBootApplication: Quando iniciamos um projeto Spring utilizando o **SpringBoot** ele já traz uma classe principal com o **void main()** e a anotação **@SpringBootApplication**, ela é uma combinação de outras 3 anotações sendo elas: **@ComponentScan**, **@Configuration**, **@EnableAutoConfiguration**.

@EnableAutoConfiguration: É utilizada para dizer para o **Spring** para ele utilizar de forma automática das suas configurações que então quando utilizamos do **SpringBoot** ele já faz várias configurações defaults por trás (**tomcats**, **mvc**, etc), sem precisarmos fazer muitas configurações personalizadas.

@ConfigurationProperties: Usamos essa anotação quando queremos pegar variáveis dos **properties** e atribuir seus valores em alguma classe, para não precisarmos definir o **@value** de cada uma na classe usamos essa anotação com o **prefixo** como **parâmetro**, assim ele vai disponibilizar todos os valores que são de variáveis com **aquele prefixo**, ficando então **muito mais fácil a injeção e melhor de ser reutilizado em outras classes**.

```

@ConfigurationProperties(prefix = "app")
@Component
public class AppProperties {

    private String name;
    private String port;
    private String host;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getPort() {
        return port;
    }
}

```

Agora podemos injetar essa classe em algum controller por exemplo e dar um console log `appProperties.getName()` que irá funcionar, ao invés de adicionar diretamente os atributos um por um no controller ou então se precisarmos de usar essas propriedades em outras classes fazer isso em cada uma.

WEB:

@RestController: É uma anotação derivada da `@Controller` que indica que usamos para deixar explícito para o `Spring` que a classe em que usarmos ela vai ser um `Bean` que vai ser gerenciado por ele, usamos ela

em APIs Rest por trabalharem com end-points, pois não temos a camada view.

@RequestMapping: Nela informamos como parâmetro qual que vai ser a URI que o cliente vai mandar para acessar os métodos deste end-point, ou seja com essa anotação a gente vai mostrar para o Spring qual que vai ser a URI que ele vai redirecionar essas requisições/solicitações para que nossos controllers possam responder dependendo do método que é acionado e todas as outras anotações de Mapping derivam dele.

@GetMapping: Deriva da anotação @RequestMapping, porém é melhor usarmos ele quando temos métodos que usam do GET, pois ele abrange melhor requisições GET, usamos em métodos para informar qual será o a requisição HTTP que será recebida que no caso é a GET.

```
@GetMapping  
@RequestMapping(value = "/parking-spot", method = RequestMethod.GET)
```

Então concluímos que em questão de colocar nos métodos a anotação @GetMapping é preferível.

@PostMapping: Usamos para informarmos para o Spring que o método em questão vai usar receber uma solicitação POST.

@DeleteMapping: Usamos para informarmos para o Spring que o método em questão vai usar receber uma solicitação DELETE.

@PutMapping: Usamos para informarmos para o Spring que o método em questão vai usar receber uma solicitação PUT.

@RequestBody: Quando temos que receber algo do corpo do site (JSON) usamos o @RequestBody para que ele possa mapear dos dados em JSON para o objeto Java (DTO por exemplo) a gente vai usar no parâmetro do método essa anotação.

```
@PostMapping  
public ResponseEntity<Object> saveParkingSpot(@RequestBody @Valid ParkingSpotDto parkingSpotDto){...}
```

O @Valid não é uma anotação WEB, mas é importante destaca-la por sua importância, além de adicionarmos anotações no DTO sobre validações (@NotNull, @Length, etc) precisamos adicionar o @Valid no parâmetro antes do objeto que vai ser recebido pelo método para que essas validações possam a ser realmente verificadas, então não basta colocar anotações de validações no nosso DTO se não colocarmos o @Valid.

@PathVariable: Quando temos um método que recebe um id por exemplo para podermos retornar um recurso específico para nós obtermos esse id e levamos em consideração na pesquisa do banco de dados precisamos recuperar uma parte da URI no caso o /{id} que é

uma **Path Variable** (variável dinâmica) ou seja ela muda de acordo com o **id** que o cliente envia, então para o obtermos a gente usa essa anotação **@PathVariable** que é esse **caminho variável**, então podemos recuperar dentro dele (aonde a gente passa o **value**) **oid** que corresponde ao **caminho variável** na **URI**.

```
@GetMapping("/{id}")
public ResponseEntity<Object> getOneParkingSpot(@PathVariable(value = "id") UUID id){
    Optional<ParkingSpotModel> parkingSpotModelOptional = parkingSpotService.findById(id);
    if (!parkingSpotModelOptional.isPresent()) {
        return ResponseEntity.status(HttpStatus.NOT_FOUND).body("Parking Spot not found.");
    }
    return ResponseEntity.status(HttpStatus.OK).body(parkingSpotModelOptional.get());
}
```

Então colocamos antes do objeto que vai ser recebido para indicar que ele tem o valor da **URI** que está chegando.

@RequestParam: Ele é um pouco diferente do **@PathVariable** para ela ser usada é da mesma forma, porém a **URI** vai ser enviada de forma diferente e é pouco utilizada, pois gera uma questão de vulnerabilidade de certa forma por causa que temos que passar o que estamos procurando

- <http://localhost:8080/parking-spot/testParam?id=f383ca40-5cbc-4bb4-9ce3-0f71fefe287>

@CrossOrigin: Com essa anotação estamos habilitando o **Cors** que é aonde a gente determina quais que vão ser as origens/domínios que vão poder acessar a nossa **API**, então **CrossOrigin** a gente pode definir quais são as origens/domínios específicos ou podemos deixar um ***** que significa que todas as origens são permitidas e nele também determinamos o **maxAge** que é o tempo que ele vai armazenar de resposta em segundos.

Json:

@JsonFormat: Por exemplo, imagine que precisamos receber a data/hora da consulta no seguinte formato: **dd/mm/yyyy hh:mm**. Para que isso seja possível, precisamos indicar ao **Spring** que esse será o formato ao qual a data/hora será recebida na **API**, sendo que isso pode ser feito diretamente no **DTO**, com a utilização da anotação

`@JsonFormat`:

```
@NotNull
@Future
@JsonFormat(pattern = "dd/MM/yyyy HH:mm")
LocalDateTime data
```

No atributo *pattern* indicamos o padrão de formatação esperado, seguindo as regras definidas pelo padrão de datas do **Java**. Você pode encontrar mais detalhes nesta [página do JavaDoc](#).

Essa anotação também pode ser utilizada nas classes **DTO** que representam as informações que a **API** devolve, para que assim o **JSON** devolvido seja formatado de acordo com o pattern configurado. Além disso, ela não se restringe apenas à classe `LocalDateTime`, podendo também ser utilizada em atributos do tipo `LocalDate` e `LocalTime`.