Spring

Spring boot é uma ferramenta/facilitador que padroniza todas as configurações pra gente e já evita para não precisarmos fazer as configurações manualmente, então ele abstrai muita coisa das configurações iniciais que precisaríamos de fazer em projetos Spring, ou seja o framework que utilizamos não é Spring Boot e sim o Spring, Spring Boot é apenas a ferramenta/facilitador que usamos para criar nossos projetos.
Antes em projetos WEB gerávamos arquivos WAR ou um EAR se for projeto mais enterprise, e a gente precisando fazer um deploy desse arquivo WAR ou EAR em um application container (servidor de aplicação com suporte Java) temos então o Tomcat, JBoss, weblogic, etc. E o Spring Boot no momento que estava ficando popular a parte de micro serviços ele implementou o Tomcat e é por isso que hoje as aplicações do Spring a gente consegue rodar num Docker, pois acaba gerando um arquivo JAR como pacote final que vai para a produção e esse arquivo JAR só precisa do Java para ser executado e é o que facilitou a adoção de micro serviços no mundo Java.

Application.properties:

Foi a forma de padronizar a forma que declaramos as propriedades em um projeto Spring.

Spring Core:

Parte principal do Spring que é onde temos a parte de injeção de dependências, bootstrap.

Spring Web/Webmvc:

Anotações @Controller, @Services que temos.

É todo baseado na especificação Servlet do Java, então quando criamos um @Controller no Spring, por trás é um Servlet do Java que a gente tem.

@ResponseBody:

O Spring trabalha muito com aquele modelo do MVC ou Model View, geralmente quando retornávamos algo o Spring esperava qual que nosso jsp que iriamos retornar/fazer o redirecionamento, no caso o @ResponseBody ele vai transformar por

exemplo uma String ele vai retornar apenas uma String, se for um Json/objeto vai retornar isso no formato Json ou formato XML.

@RestController:

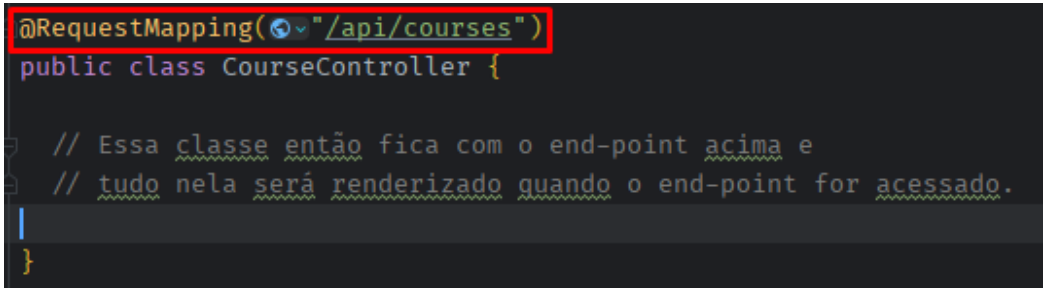
É apenas uma conveniência da gente poder utilizar @Controller e @ResponseBody e não precisar digitar os 2.

Que indica para o Spring que a classe ela contém um end-point (url) que nós vamos poder acessar a nossa API.

@RequestMapping:

Diz aonde é o end-point de determinada classe.

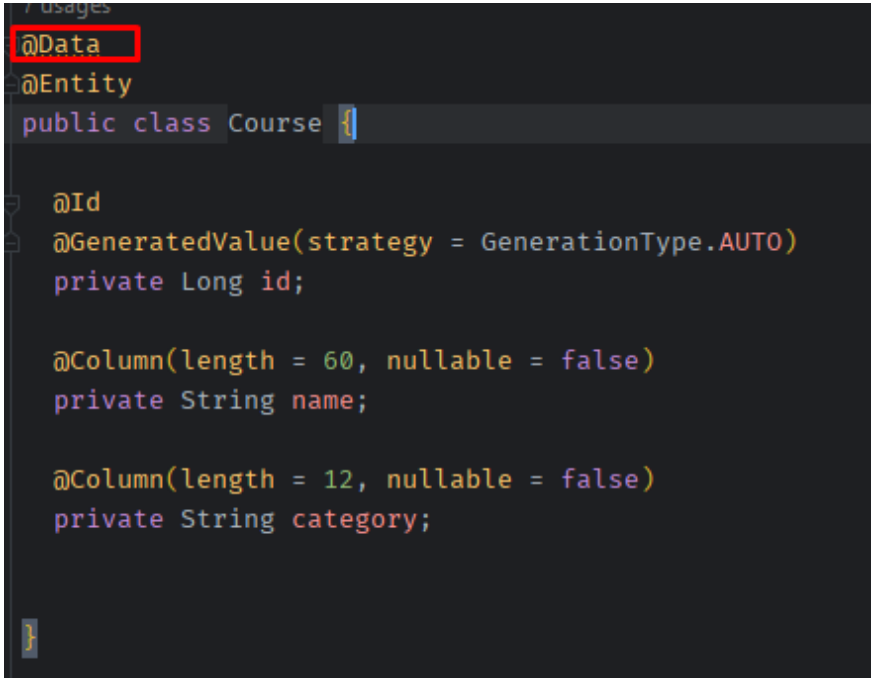
Exemplo:



LOMBOK:

Lombok é uma dependência que nos ajuda a ter "atalhos" na nossa aplicação, como por exemplo para gerar os getters and setters precisaríamos ter aquele monte de coisa escrita e gerar eles causaria "poluição" na nossa classe já que eles ocupam muitas linhas, com o Lombok nós temos as anotações @Getter, @Setter, @Data = cria os getters and setters e tem outras anotações também que ao colocado em cima ele gera sem precisar da poluição visual.

Exemplo:



Vemos então na imagem que não temos os getters and setters poluindo a classe visualmente, porém temos eles e podemos utiliza-los por causa do @Data do Lombok.

@GeneratedValue:

Informa como esse valor deve ser gerado, isso depende bastante do banco de dados, porém o mais comum MYSQL utiliza o AUTO (cria o dado automaticamente ele mesmo).

lenght: Informa o máximo de caracteres que é aceita na coluna.

nullable: Informa se a coluna aceita valores nulos ou não.

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

@Component:

Se você quer que o Spring crie uma instância e automaticamente gerencie o ciclo de vida dessa instância você coloca essa anotação.

Existem os components especiais como o @RestController que é um component que vamos expor uma API/end-point, @Repository que é um component especial que está falando para o Spring que é uma conexão (vai fazer o acesso ao banco de dados), @Service que é um component especial que é geralmente aonde colocamos a nossa lógica de negócios e também conseguimos fazer controlar as transações com o banco de dados.

@Repository:

Para termos acesso aos métodos do banco de dados a gente declara um repository como interface para podermos estender (extends) as interfaces que nós temos do próprio JPA no Spring Data que é um outro módulo que a gente adicionou no nosso POM.xml que possui interfaces para poder facilitar o acesso ao banco de dados, ao invés de fazer tudo manualmente com o ORM do Spring e fazer as conexões com o hibernate tudo manualmente, então o Spring criou essa outra camada para facilitar.

E ao fazer isso temos então acesso ao JpaRepository que iremos usar e nela temos que usar o tipo generics (<>) que temos que informar qual que é a nossa entidade (Entity) e qual é a chave primária dessa entidade, quando fazemos isso o Spring vai criar uma implementação dessa interface que já tem os métodos automaticamente para gente poder acessar (para ver isso podemos clicar com o ctrl+mouse1 no JpaRepository) que ele mostra os métodos.

Podemos também declarar métodos, por exemplo findbyName o Spring Data vai criar então um método para podermos acessar fazendo um SELECT * FROM (tabela) Where name.

```java
@Repository
public interface CoursesRepository extends JpaRepository<Course, Long> {
```

@Bean:

Estamos com ele falando para o Spring que queremos que o Spring gerencie todo ciclo de vida.

//

Quando temos um atributo que é obrigatório eu ter essa instância para que meus métodos funcionem, ou seja não vai funcionar se eu não tiver a instância dessa propriedade, por exemplo um repository, a gente considera isso como uma propriedade um atributo obrigatório, quando isso acontece a gente da preferência de fazer a injeção via constructor, porque quando o Spring for instanciar, o Spring vai falar "Essa classe aqui precisa dessa instância para poder funcionar", então no momento da criação (instância) (new CourseController) é que vamos passar essa instância, se fizermos isso via atributo (@Autowired) ou via setter a gente informa que precisamos disso em um 2° momento, então iremos sempre gerar o constructor.