Spring Boot

Spring Framework

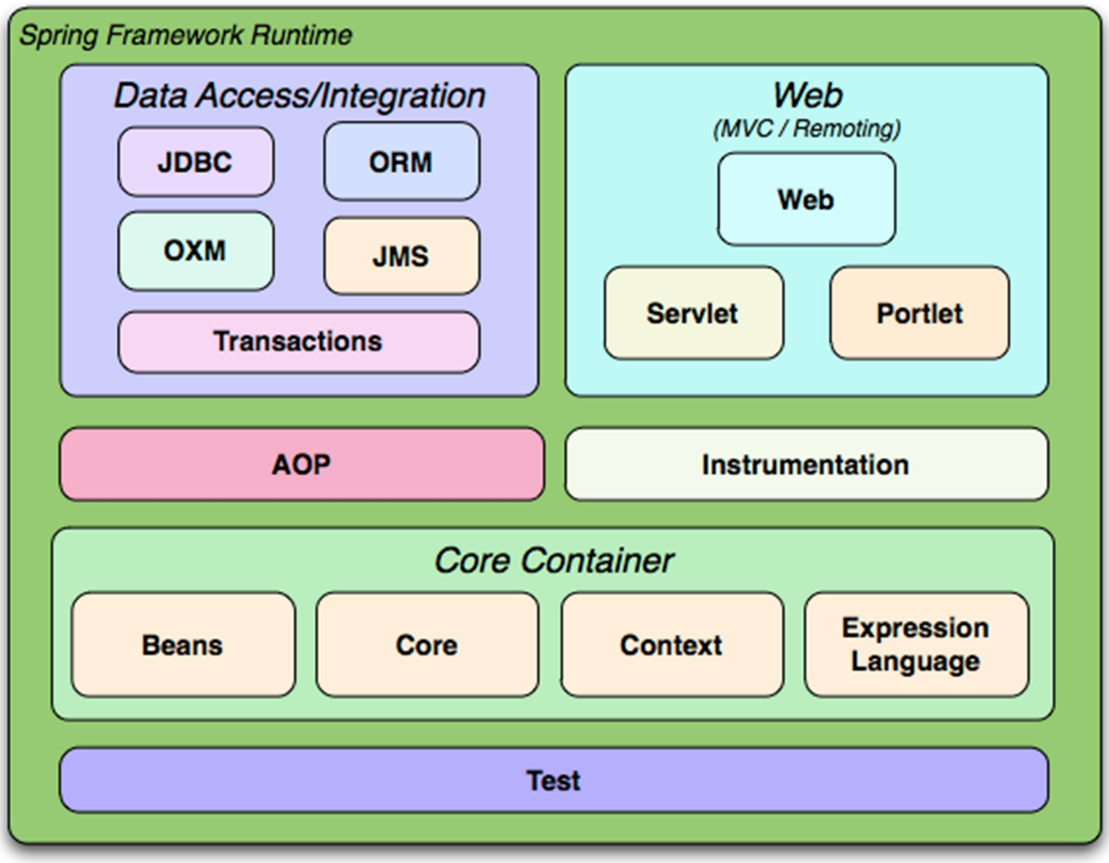
Todo framework é baseado em fundamentos , conceitos e padrões de boas práticas que são constantemente consolidados em programação

Spring Framework é um framework open source desenvolvido para plataforma java baseados nos padrões de projetos inversão de controle (atribuir responsabilidade a um container) e através da injeção de dependência atribuímos os recursos necessários para nossa aplicação.

Sua estrutura é composta por módulos afins de reduzir a complexidade no desenvolvimento de aplicações simples ou corporativa.

Existem módulos específicos para cada parte da nossa aplicação.

Ex. queremos trabalhar com dados, temos um módulo específico. Queremos trabalhar com segurança, temos um modulo pra isso.



Spring é baseado no módulo Core (núcleo), e através dele conseguimos iniciar um container spring.

O núcleo é o componente central do sistema operativo, ele é composto por um conjunto de rotinas, que estão disponíveis para ser usado pelo usuário e às suas aplicações.

Como citado acima, o núcleo (ou core) de um sistema é centro da aplicação. No Spring existe um conceito muito semelhante. Com o nome oficial de 'Core Container', o núcleo do Spring é responsável por criar os objetos, conectá-los, configurá-los e gerenciar seu ciclo de vida completo, desde a criação até a destruição. E se não bastasse tudo isso, o Spring Container ainda faz mais.

Dentro da arquitetura Spring, o Core Conteiner é dividido em componentes. O seus quatro componentes fundamentais são: Core, Beans, Expression Language (SpEL) e Context. Esses componentes são a base de todo o Core Container. Vamos entender um pouco mais sobre eles então:

Core

No Core se encontra toda a infraestrutura necessária que permite a implementação do container. São encontrados facilitadores no gerenciamento de classloaders, reflexão, tratamento de recursos, strings, etc.

Beans

Usando como base o Core temos o módulo Beans, onde encontramos o BeanFactory, que implementa do factory. O factory é um padrão de projetos permite às classes delegar para suas subclasses decidirem quais objetos criar. Este padrão de projeto que torna desnecessária a codificação manual de singletons (Padrão que garante a existência de instância de apenas uma classe) e nos permite o desacoplamento total do gerenciamento de dependências e ciclo de vida dos beans (Objeto de negócio) do código fonte através do mecanismo de configuração implementado neste módulo.

Expression Language (SpEL)

O componente de Expression Languege é uma linguagem baseada em expressões que é utilizado como modo de configuração do Spring. Esta linguagem nos permite definir valores e comportamentos aos nossos beans definidos usando XML.

Context

Finalmente, temos o módulo Context, onde se encontra a implementação mais avançada e comumente usada do container que é o ApplicationContext. O ApplicationContext é a interface central em um aplicativo Spring usado para fornecer informações de configuração ao aplicativo. Ele nos oferece recursos poderosos como por exemplo internacionalização, propagação de eventos, AOP, gerenciamento de recursos e acesso a funcionalidades básicas da plataforma Java EE.

Através dele criamos o beans, que são os objetos gerenciados pelo container spring, e ai teremos uma baixa dependência, um baixo acoplamento dentro das nossas aplicações.

Consequentemente conseguimos implementar um projeto baseado em interfaces , pra deixar as nossas implementações mais flexíveis e dinâmicas.

Também podemos ter interações com banco de dados via JDBC, na forma tradicional, ou partindo para ORM através de frameworks, além de outras interações.

Também podemos aumentar a produtividade no desenvolvimento web, através do módulo web.

E, sem deixar faltar, estruturas voltadas a testes.

Spring x Java EE

O spring foi um movimento diante da complexidade, da burocracia e de inúmeros processos das convenções da jEE.

E diante da necessidade de termos mais otimização, mais performance e atualizações recorrentes, surgiu o movimento spring.

Mas isso não significa que o java deixou de ser requisitado em alguns projetos.

Falando de história...

Olhando um pouco a história, há muito tempo atrás o java EE era realmente muito complicado e nem era necessário entrar numa discussão, muita burocracia, usar o spring era o caminho mais simples e mais fácil de evoluir.

Ai chegou a versão 5 do java e a discussão voltou a ficar quente.

Com a chegada do spring, a comunidade java percebeu que deveria ficar um pouco menos burocrática e assim aderiu a sugestões que a comunidade trazia para o projeto java.

Hoje podemos dizer que tanto o java EE como o spring possui features essenciais para um ambiente corporativo.

A sua escolha estará voltada a metodologia da empresa ou como você vem dominando todos os fundamentos.

Inversão de Controle:

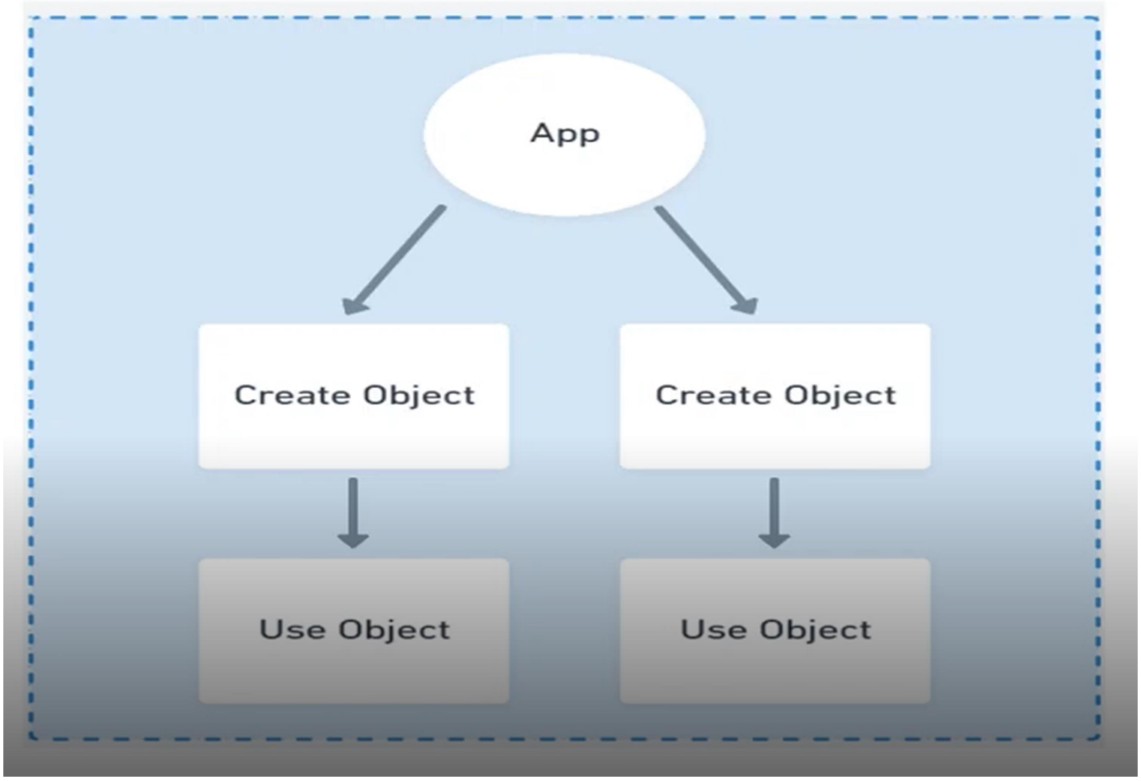
Inversion of Control ou IoC, trata-se do redirecionamento do fluxo de execução de um código retirando parcialmente o controle sobre ele e delegando-o para um container.

O principal propósito é minimizar o acoplamento do código.

A partir de agora temos um novo personagem na história: o container.

Quando inicializamos uma aplicação com o spring, um container é inicializado para obter todas as instâncias, todas as referências e configurações necessárias a nossa aplicação, pra que através da inversão de controle, consigamos não mais ter responsabilidade gerencial dos nossos objetos, mas deixar para o container fazer a gestão, a instanciação, performance, alocação de recurso em nossa memória e nos objetos existentes na nossa aplicação.

Aplicação sem o IoC

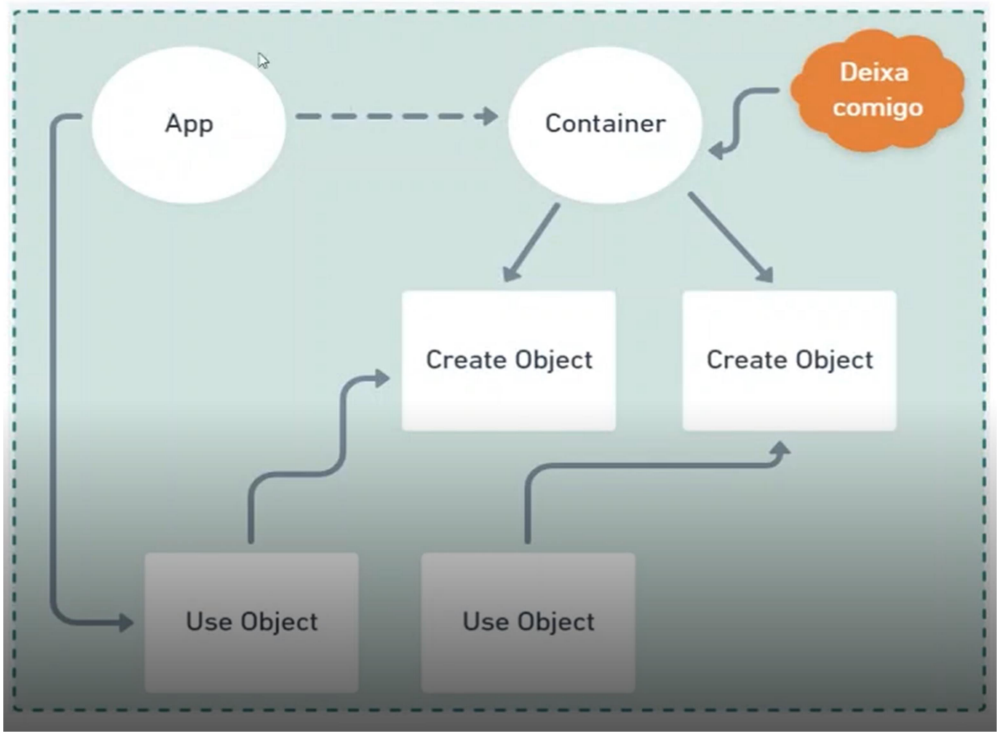


Aqui vimos uma ilustração da aplicação java sem o IoC.

Antes do IoC precisávamos criar o objeto para utilizar. Instanciar para poder utilizar.

Quando ele vai deixar de existir, ou quantos objetos existem na memória, nós não conseguimos fazer esse tipo de levantamento a olho nu sem a inversão de controle não conseguíamos fazer isso.

Com o IoC



Com o IoC, nós nós temos a aplicação, a aplicação é inicializada, consequentemente nosso container é configurado, ele é carregado, e aí todos os objetos que precisa previamente já estarão no container.

E quando a gente precisar usar algum abjeto, precisamos apenas requisitar ao conteiner as suas respectivas instâncias.

Então essa é a proposta da inversão de controle.

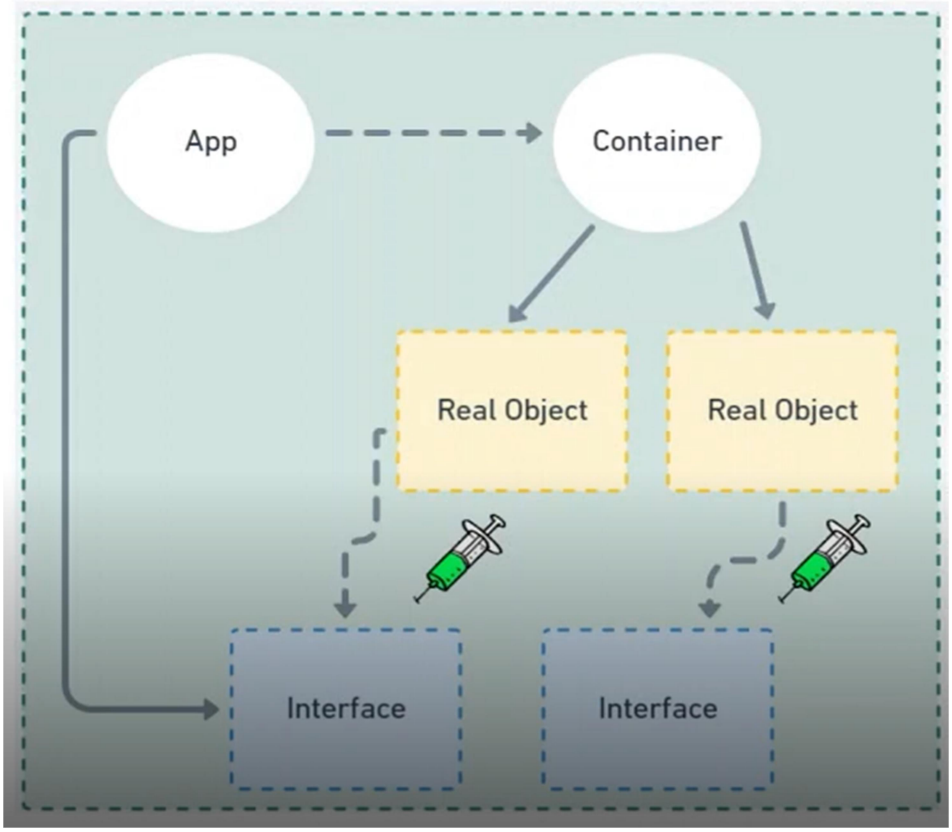
Injeção de Dependência

Injeção de Dependência é um padrão de desenvolvimento com a finalidade de manter baixo o nível de acoplamentos entre os módulos de um sistema.

Aqui não existe a necessidade muitas vezes até mesmo de ter a implementação, através da inversão de controle está que fortemente relacionada com a injeção de dependência, eu tenho uma interface, eu tenho o tipo correspondente, através do tipo que foi carregado pelo meu contêiner, eu consigo injetar uma especificação para as minhas interfaces.

Consequentemente a minha interface, a minha aplicação não tem ciência de qual o objeto, de qual dependência exija.

Mas o conceito de gestão de dependência é prover as nossas referências, os recursos necessários a aplicação sem tornar definitivamente tão explícitos, tão acoplados como a gente vem trabalhando fora de um container, fora de um contexto Spring.



E aqui temos a mesma analogia para injeção de dependência.

Mais uma vez a nossa aplicação ao ser inicializada, ela carrega o container.

O nosso Container que tem objetos reais,

(e colocando o máximo possível de boas práticas) as nossas implementações seriam fortemente associadas a interface.

Consequentemente a aplicação de não tem ciência de qual instância.

Ela tem um contrato, ela tem uma característica, e os objetos reais que atendem esta característica serão injetados através do recurso de gestão de dependência que é provida pelo contêiner.

Beans

Beans é um objeto que é instanciado (criado), montado e gerenciado por um container através do princípio da inversão de controle.

Como mostramos nas ilustrações anteriores, quando temos o conceito de inversão de controle e gestão de dependência, os nossos objetos serão criados pelo container e serão mantidos e gerenciados.

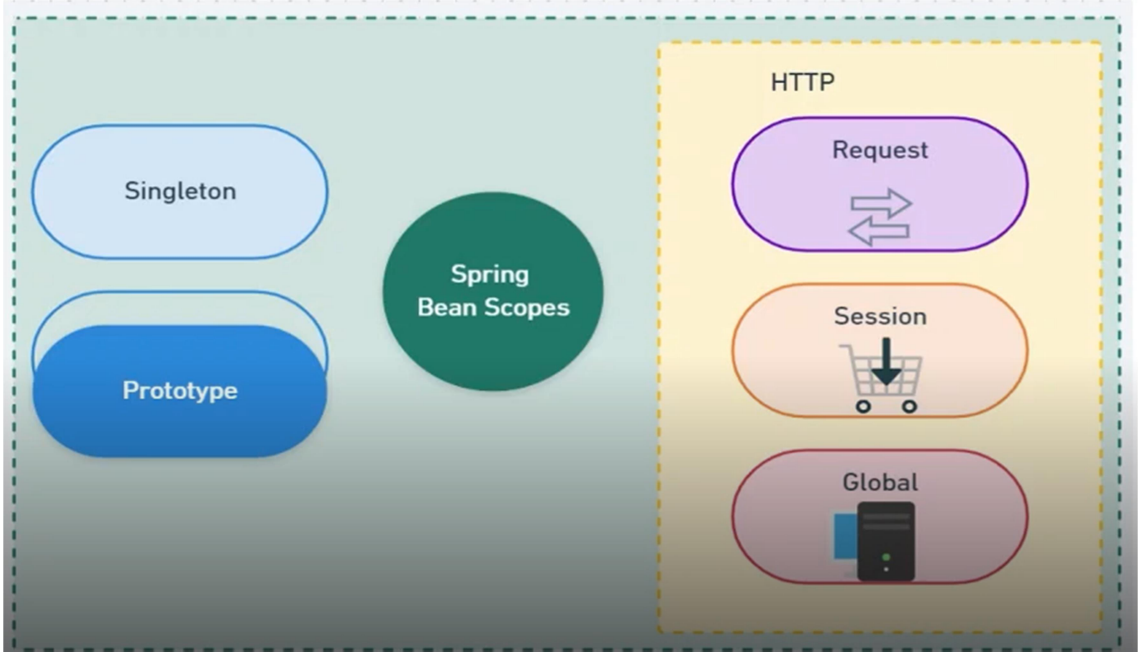
Isso quer dizer que: iniciamos a aplicação. Quantos objetos eu preciso ter diante de toda a minha aplicação, e também é gerenciado pelo conceito de bens do nosso contexto.

E assim sucessivamente.

Outras características, como por exemplo ser montado e gerenciado pelo contêiner.

O bean, nada mais é do que um objeto que é gerenciada pelo contêiner.

Scopes



Nós temos cinco tipos de scopes categorizado para conceitos standalone,e conceitos Http, conceitos web.

Iremos dar ênfase inicialmente aos socpes singleton e prototype, porque são padrão do spring.

Dentro do contexto standalone, ele é adotado em qualquer esfera de uma aplicação.

É como o próprio nome diz, singleton é um escopo onde vamos ter um único objeto compartilhado por toda minha aplicação, quando for solicitado.

Diferentemente nós temos o scope prototype, e esse é criado uma nova instância a cada requisisação de um objeto, de uma referência ao contêiner.

E o contêiner entende que eu não posso enviar a mesma referência ao mesmo objeto, para camadas ou cenários diferentes na minha aplicação.

Consequentemente nós temos os contextos http, que são escopo de requisições, de sessão e escopo Global.

Vamos conhecer mais desses escopos:

Singleton.

O contêiner do Spring IoC define apenas uma Instância de objeto para toda aplicação será criado um novo objeto na aplicação.

Prototype

Será criado um novo objeto a cada solicitação ao container, justamente para não inferir mudança de estrutura desses objetos.

http

Um bean será criado a cada requisição http.

Ficará de forma que os nossos objetos, nossos beans da aplicação, podem ter um esfera de requisição e esse objeto a ser transformados em toda a cadeia da aplicação, em todas as camadas da aplicação.

Porém quando a essa aplicação for finalizada, ou essa requisição esse bean será destruído.

Session

Um bean será criado para cada sessão de usuário.

Costumamos usar muito essa esfera de escopo pra sessão de usuário, para manter os dados dos usuários, o carrinho de compras e etc.

Global

Ou Application Scope cria um bean para o ciclo de vida do contexto da aplicação.

Enquanto a aplicação estiver no ar, esse bean de escopo global estará disponível para sua atalziação.

Autowired

Uma anotação (indicação) onde deverá ocorrer uma injeção automática de dependência.

- byName: é buscado um método set que corresponde ao nome do bean.
- byType: é considerado o tipo de classe para inclusão do Bean.
- byConstrutor: Usamos o construtor para incluir a dependência.

Em que momento, conseguimos delegar, que através de uma referência via interface, ou via classe, conseguimos buscar através da injeção de dependência, dentro do meu container qual dependência vou precisar obter para utilização.

E conseguimos ter esse autowired através de um nome, que vamos buscar um método que corresponde ao nome do meu bean.

Podemos buscar o tipo, que é o tipo da instância, tipo da Classeque estamos considerando como dependência, ou pelo Construtor.

Através do autowired, que não são necessariamente anotações, mas indicações de onde iremos injetar os nossos objetos.

Spring Boot Rest

Iremos agora falar do contexto web dentro do framwork.

O spring web possui o webmvc, que é um conjunto de módulos de model,view e controler, e temos a concepção de rest APi.

No spring boot web temos esses dois sistemas de finalidade de aplicação web.

Um é voltado a aplicação web, com interfaces gráficas, páginas, e o outro só a camada de disponibilização de recursos, ou apis de uma arquitetura rest.

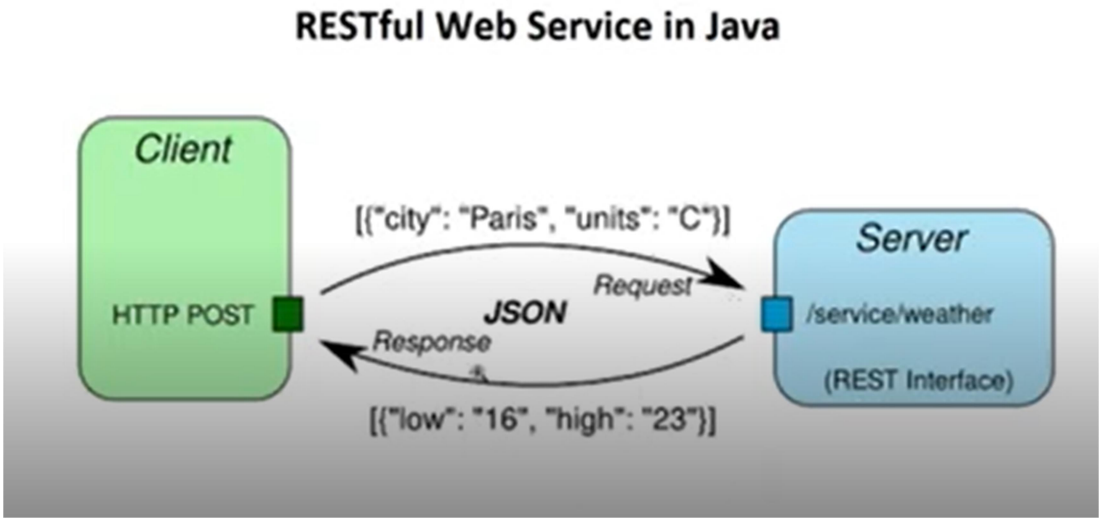
No curso iremos falar da arquitetura rest.
O que é rest, restfull, controller, documentação com Swagger e tratar algumas exceções ou fluxos inesperados com execption Handler.

Recursos Http

Iremos implementar recursos http, para intereação dos arquivos Json pelas aplicações.

(O formato **JSON** (JavaScript Object Notation) é um formato aberto usado como alternativa ao XML para a transferência de dados estruturados entre um servidor de Web e uma aplicação Web. Sua lógica de organização tem semelhanças com o XML, mas possui notação diferente.)

Iremos utilizar um arquitetura baseada em definições da camada de serviços rest.



Ou seja, temos o cliente, que irá solicitar o serviço, web, fazendo a requisição ao servidor e este retornará as respostas no formato json.

Deve ficar claro que não estamos falando de páginas web como estamos acostumados.

O que é uma API?

Uma API (interface application program) é um código programável que faz a “ponte” de comunicação entre duas aplicações distintas.

Através da Api conseguimos integrar a aplicação com outros serviços, que podem ser serviços internos ou fornecidos por outras empresas.

Rest e RESTful

A api rest (representational state transfer) é como um guia de boas práticas e RESTful é a capacidade de determinado sistema aplicar os princípios de rest.

Nem todas as APIs ou todos os webservices existentes no mercado estão aderentes as sistema rest;

Iremos conhecer as características, quais os princípios e níveis que nossa aplicação precisa aderir pra ser considerada uma API rest.

Princípios

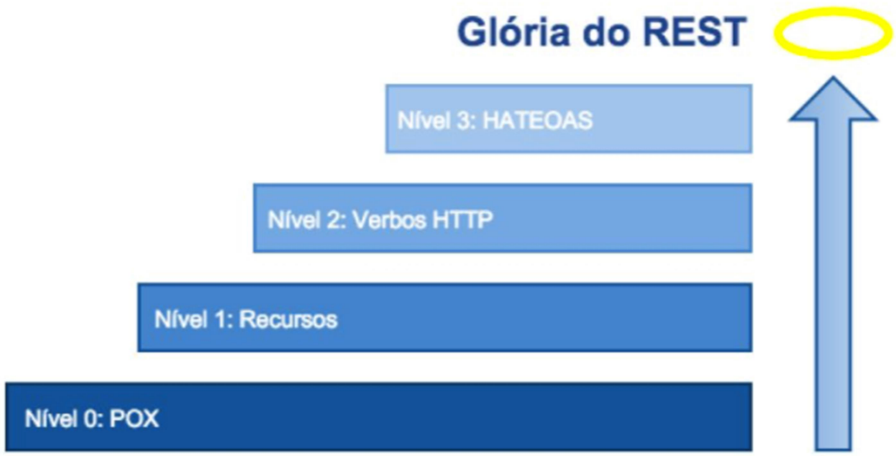
Para que uma arquitetura seja RESTful, é necessário ter uma série de princípios padrões:

- Cliente-servidor: significa aprimorar a portabilidade entre várias plataformas de interface do usuário e do servidor, permitindo uma evolução independente do sistema; conseguimos uma interface mais simples entre a aplicação e o servidor.
- Interface uniforme – representa uma interação uniforme entre cliente e servidor. Independente de qual cliente esteja utilizando minha aplicação, será sempre da mesma forma. Para isso é preciso ter uma interface que identifique e represente recursos, mensagens autodescritivas, bem como hypermedia (HATEOAS);
- Stateless – indica que cada interação via API tem acesso a dados completos e compreensíveis; todas interações teremos dados necessários ao seu contexto.
- Cache – necessário para reduzir o tempo médio de resposta, melhorar a eficiência, desempenho e escabilidade da comunicação;
- Camadas – permite que a arquitetura seja menos complexa e altamente flexível.

Também precisamos ter uma equalização mediante nossos recursos, voltado ao nível de maturidade da nossa API.

Nível de Maturidade

Para padronizar e facilitar o desenvolvimento de APIs Rest, Leonard Richardson propôs um modelo de maturidade para esse tipo de API, definido em 4 níveis.



Para que sua API atenda todas as características de uma API REST, ela precisa estar em um desses níveis.

Nível 0: Ausencia de Regras

Essse é considerado o nível mis básico e uma API que implementa apenas esse nível não pode ser considerada REST, pois não segue qualquer padrão.

Esse nível implementa todos os recursos, mas ele não segue nenhum padrão.

Ex.

Verbo HTTP	URL	Operações
POST	/getUsuario	Pesquisar Usuário
POST	/salvarUsuario	Salvar
POST	/alterarUsuario	Alterar
POST	/excluirUsuario	Deletar

Temos um único verbo.

Temos as URL, mas sem nenhuma padronização;

As operações também não possuem nenhum padrão.

Nível 1: Aplicação de Resources

Verbo HTTP	URL	Operações
GET	/Usuario/1	Pesquisar Usuário
POST	/Usuarios	Salvar
PUT	/Usuarios/1	Alterar
DELETE	/Usuarios/1	Deletar

Aqui os nomes dos recursos foram equalizados e para não gerar ambiguidades é necessário definir o verbo adequadamente.

A partir de agora é necessário sermos mais seletivos com os verbos, ou seja um verbo para cada operação.

Nível 2: Implementação de verbos HTTP

O nível 2 tem uma grande dependência do nível 1, pois a definição dos verbos já foi requisitada no Nível 1, o nível 2 se encarrega de validar a aplicabilidade dos verbos pra finalidades especificas como:

Isso quer dizer que se o método GET tem a finalidade de retornar dados, todas as nossas requisições que possuem a finalidade de retornar dados, devem ser do tipo GET, e assim sucessivamente.

Verbo HTTP	Função
GET	Retorna Dados
POST	Grava Dados

PUT	Altera Dados
DELETE	Remove Dados

Existe uma discussão quando precisamos retornar dados através de parâmetros via body recebidos pelo método post. Isso quando precisamos retornar dados, mas precisamos passar alguma informação, um filtro e ai não conseguimos ter esses recursos através do get.

Assim algumas requisições, alguns desenvolvedores aderem ao uso de post, quando não querem passar esses parâmetros via url.

Essa discussão entre usar post ou get se encaixam nas definições de um arquiteto.

Nível 3: HATEOAS

O nível 3 disponibiliza uma estrutura dentro das nossas respostas.

HATEOAS significa Hypermedia as the Engine of Application State.

Uma API que implementa esse nível fornece aos seus clientes links que indicarão como poderá ser feita a navegação entre seus recursos. Ou seja, quem for consumir a API precisará saber apenas a rota principal e a resposta dessa requisição terá todas as demais rotas possíveis.

O nível 3 é sem dúvida o menos explorado, muitas APIs existente no mercado não implementam esse nível.

Springboot REST API

Agora iremos criar nosso projeto com o rest api.

Criando o projeto:

Spring Initializr

Site que oferece os recursos para a criação de um projeto Spring Boot com uso do maven ou gradle.

- Acesse o site: https:start.spring.io/
- Selecione a opção maven Project
- Selecione a opção Language – Java

Abrindo o site temos as configurações básicas da nossa aplicação, e a versão do spring boot.

Preenchimento

- Group: Nome do grupo Organizacional - Funtec
- Artifact: Identificação do projeto - meu-primeiro-web-api
- Name: nome do projeto (igual ao artifact) - meu-primeiro-web-api
- Package name: web-api

Como usaremos que a aplicação seja uma aplicação web, adicionamos a dependência – spring web

Clica em gerar.

Salvar

Extrair

Importar, (principais:pasta src e pom.xml.

Abrir o projeto na ide

Vemos no arquivo as dependências do spring boot, e entre elas a dependência do spring boot start web, (linhas 20 a 23).

A estrutura do projeto fica a mesma, porem com a estrutura do springboot,

A classe ApiAplication terá a notação @springBooot Application

Temos o arquivo minha-primeira-api-application, com o método run.

Executar e ver o log da aplicação.

Vemos no log que o tomcat foi criado (servidor de aplicações web).

Vamos no browser.. localhost:8080

Whitelabel error page,....

Iremos agora disponibilizar recursos, serviços, camadas de serviços.

Dentro do spring web, esta camada é chamada de controller, que tem algumas características e finalidades especificas.

Criando nosso primeiro Controller

Um controller é um recurso que disponibiliza as funcionalidades de negócios da aplicação através do protocolo HTTP.

O controller em nossa aplicação, já obtém toda a logica de negocio.

Não é no controller que disponibilizamos as funcionalidades essenciais da aplicação.

Vemos o controller como uma camada que diz: Esse serviço funciona de forma local, eu quero agora prover para o mundo através de uma web api.

Qual a estrutura?

Quais as características imprescindíveis?

```java
import org.springframework.web.bind.annotation.RestController;
import org.springframework.web.bind.annotation.GetMapping;

@RestController
public class WelcomeController {
    @GetMapping("/welcome")
    public String welcome(){
        return "Welcome to a Spring Boot REST API";
    }
}
```

Vamos criar uma classe de exemplo.

Crie a classe BemVindoController.java, no pacote controller

```java
package Funtec.meuprimeirowebapi.Controller;
```

```
public class BemVindoController {

	public String bemvindo() {
		return "Bem vindo ao meu primeiro Spring Boot Web API";
	}
}
```

Até aqui temos um método que retorna alguma informação. (o texto)

Para que possamos transformar essa classe em um componente rest, o componente que irá gerenciar nossa aplicação, precisamos acrescentar o @RestController

```
package Funtec.meuprimeirowebapi.Controller;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class BemVindoController {

	public String bemvindo() {
		return "Bem vindo ao meu primeiro Spring Boot Web API";
	} }
```

A partir de agora essa classe é um restController e alguns de seus métodos serão serviços http.

Para isso também precisamos mapear esses métodos, porque nem todos os métodos se tornarão ou precisam ser recursos http.

Para isto informar que este método é um recurso http, utilizando a annotation @getMapping

```
package Funtec.meuprimeirowebapi.controller;

import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;

@RestController
public class BemvindoController {

	@GetMapping
	public String Bemvindo() {

		return "Bem vindo ao meu primeiro Spring Boot Web API";
	}}
```

Agora executar e no browser ver a mensagem de boas vindas!

Conseguimos executar o método através de um controller so spring com o mapeamento do tipo get, e esse método retorna um texto simples.

A ideia foi mostrar como podemos executar um restcontroller, através do mapeamento.

Iremos falar a seguir do Json, mas a proposta ate aqui vou mostrar como é fácil tornar uma classe em restController, e ele fazendo o mapeamento de nossos recursos

---

Agora iremos criar nosso primeiro controller para uma demonstração de uso de nossos serviços pela web.

A aplicação que já fizemos tem a finalidade de prover uma classe que denominamos de controller e um recursos get, que tem a responsabilidade de retornar uma informação através de recursos http.

Vemos que é uma aplicação spring boot simples, mas com a inclusão em nossa dependência do boot-starter, no arquivo pom.xml desta maneira conseguimos adicionar as anotations (@restController) tornando nossa classe um webservice.

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

Rest Controller

Um rest controller em Spring nada mais é que uma classe contendo anotações específicas para a disponibilização de recursos https baseados em nossos sérvios e regras de negócio.

Temos uma classe simples, e vamos mapeando através de algumas anotações e convenções os nossos métodos, para prover os recursos http.

Vamos agora conhecer essas anotações:

Anotações e configurações mais comuns:

- @RestController: Responsável por designar o bean de componente que suporta requisições HTTP com base na arquitetura rest.
- @RequestMapping(“prefix”): Determina qual a URL comum para todos os recursos disponibilizados pelo Controller. (convenção de nomenclatura).
- @GetMapping: Determina que o método aceitará requisições HTTP do tipo GET.
- @PostMapping: Determina que o método aceitará requisições HTTP do tipo POST.
- @PutMapping: Determina que o método aceitará requisições HTTP do tipo PUT.
- @DeleteMapping: Determina que o método aceitará requisições HTTP do tipo DELETE.
- RequestBody: Converte um JSON para o tipo do objeto esperado como parâmetro no método.
- PathVariable: Consegue determinar que parte da URL será composta por parâmetros recebidos nas requisições.

Para exploramos essas anotações, vamos fazer um sistema de controle de ususários.

Vamos disponibilizar as funcionalidades de CRUD da entidade User,java através da API.

Vamos simular um Fake Repository para exemplificar.

Criar o pacote Model, e a Classe Usuario, que vai representar o cadastro de usuários.

```
package Funtec.meuprimeirowebapi.model;

public class Usuario {
	private Integer id;
	private String login;
	private String password;

	public Usuario() {}
	public Usuario(String login, String password) {
		this.login = login;
		this.password = password;
	}
	@Override
	public String toString() {
		return "User {" +  "login=" + login + '\'' +
				", password=" + password + '\'' + "]";
	}
	public Integer getId() {
		return id;
	}
	public void setId(Integer id) {
		this.id = id;
	}
	public String getLogin() {
		return login;
	}
	public void setLogin(String login) {
		this.login = login;
	}
	public String getPassword() {
		return password;
	}
	public void setPassword(String password) {
		this.password = password;
	}


}
```

Agora iremos criar nosso repositório fake, pra representar a interação com a nossa API.

Criar pacote Repository, e a classe Usuario Repository

```
package Funtec.meuprimeirowebapi.repository;

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

import org.springframework.stereotype.Repository;

import Funtec.meuprimeirowebapi.model.Usuario;

//componente repository
@Repository
public class UsuarioRepository {

	public void save(Usuario usuario) {
		if(usuario.getId()==null)
		System.out.println("SAVE - Recebendo o usuário na camada de
repositoório ");
		else
			System.out.println("UPDATE -Recebendo o usuário na camada
de repositoório ");

		System.out.println(usuario);
	}

	public void deleteById(Integer id) {
		System.out.println(String.format("DELETE/id - Recebenso o id: %d
para delete"));
		System.out.println(id);
	}

	public List<Usuario> findAll(){
		System.out.println("LIST - Listando os usuários do sistema");
		List<Usuario> usuarios = new ArrayList<>();
		usuarios.add(new Usuario("shirlei", "123456"));
		usuarios.add(new Usuario("paulo", "654321"));
		return usuarios;
	}

	public Usuario findById(Integer id) {
		System.out.println(String.format("FIND/id - Recebenso o id: %d
para localizar um usuário"));
		return new Usuario("shirlei", "123456");
	}

	public Usuario findByUsername(String username) {
		System.out.println(String.format("FIND/username - Recebendo o
username: %s "));
		return new Usuario("shirlei", "123456");
	}}
```

Temos aqui aqui nosso repositório com algumas mensagens que buscam algumas requisições: save, update, delete, find all, findid, findusername.

Esses métodos veremos a utilidades deles qdo estivermos interagindo com o springJPA.
São nomes dos métodos que estão no framwork.

Vamos as etapas:

1 – listar nosso usuário:
Tendo o Controller (bemvindoController), cada controller pode estar associado a um serviço, ou a um contexto da aplicação.
Não necessariamente teremos um único controller.
Seguindo a convenção, também teremos um usuário controller

```
package Funtec.meuprimeirowebapi.controller;


import Funtec.meuprimeirowebapi.model.Usuario;
import Funtec.meuprimeirowebapi.repository.UsuarioRepository;

import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;


//o controller só da um direcionamento a nossa aplicação
//recomentado não colocar regras de negocio no controller, mas sim em outra camada da aplicação
@RestController
public class UsuarioController {
		//agora vamos buscar a lista que esta em nosso repositorio
		//colocamos o @Autowired, pra buscar nossa implementação, ele que contem a inteligencia
	@Autowired
	private UsuarioRepository repository; //o repository que tem a logica do negocio

	@GetMapping("/users") //verbo necessário para nossa interação
	//colocamos o users para diferenciar do GetMapping da outra classe. Precisamos nos preocupar com isso


	//metodo pra listar nossos usuarios
		public List<Usuario>getUsers(){
			repository.findAll();
			return repository.findAll();
	//o nosso controller, através do HTTP delegou para outra classe, que tem condição
		//de fazer a regra de negócio.


	}

}
```

Aqui rodar,

Localhost:8080/users

Aqui recebemos o retorno


```
[{"id":null,"login":"gleyson","password":"password"},
{"id":null,"login":"frank","password":"masterpass"}]
```


Esta é a estrutura mínima.

Vamos verificar os outros métodos,

Porem qdo estamos utilizando outros verbos, não http, como o post e put, precisaremos colocar alguns recursos que não precisam de clientes http.

Vamos mapear requisições que através da URL conseguimos ter uma interação.

Recebemos parâmetros integer e string.

E quando tivermos a necessidade de obter alguma informação dentro da URL podemos utilizar a @PathVariable

Para isto vamos alterar o UsuarioController:

```
@GetMapping("/users/{username}") //parametro q vier da url
//vamos buscar um usuário
public Usuario getOne(@PathVariable("username")String username){
	//com o @PathVariable veja a possição ou nome da variavel
		return repository.findByUsername(username);
	}
```

No browser localhost:8080/users/Shirlei

Agora incluindo o delete

```
@DeleteMapping("/users/{id}")
	public void deleteUser (@PathVariable("id") Integer id) {
```

```
        repository.deleteById(id);
    }
```

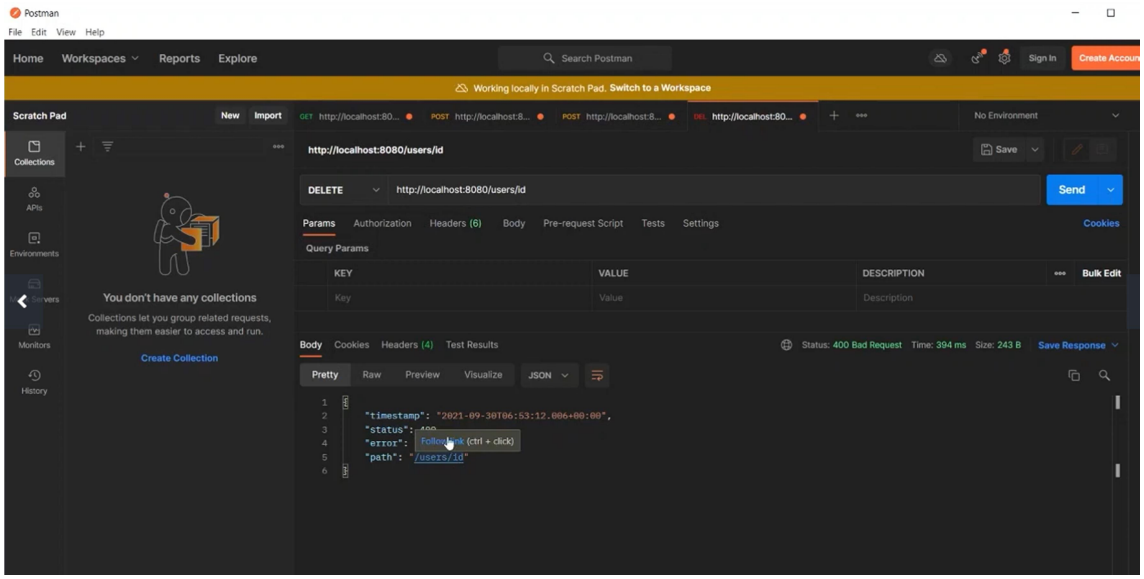
Porém as requisições HTTP, só se realizam dentro de um método Get.

Como fazer um delete? Para isso precisaremos de um Client HTTp.

Para isto temos vários no mercado.

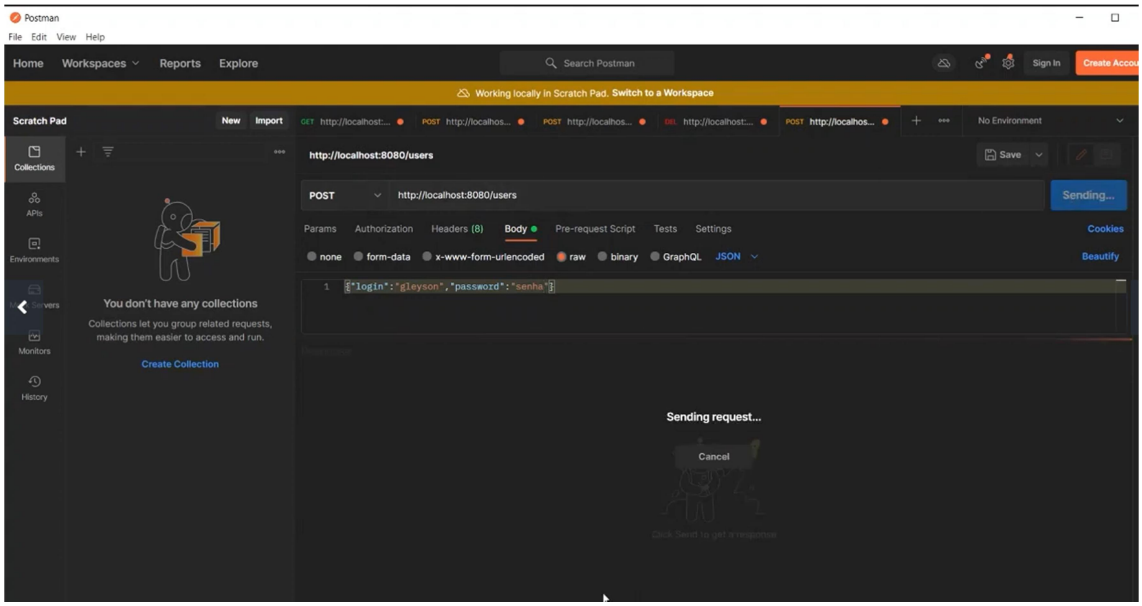
Vamos instalar o Postman

Copiamos a url, selecionamos delete



Agora já compreendemos que o mapeamento é mais de direcionamento da nossa requisição, vamos agora receber um parâmetro que seja do tipo corpo, que seja um objeto, dependendo de uma estrutura, no nosso caso o json, para que possamos realizar a interação de inclusão baseado no corpo que vamos receber nas nossas requisições.

```
    @PostMapping("/users")
    public void postUser(@RequestBody Usuario usuario) { //recebendo um
usuario
    repository.save(usuario);
    }
```



E para finalizar observamos que todas nossas requisições tem prefixo users , e como já colocamos no @requestMapping...podemos retirar..do @getMapping

---

Swagger

Com o Swagger documetanmos nossas classes e conseguimos realizar testes com nossas apis.

Swagger é uma linguagem de descrição para descrever APi RESTFull usando json. O Swagger é usado junto com um conjunto de ferramentas de software de código aberto para projetar, construir, documentar e usar serviços da web RESTful.

Habiliatando no nosso projeto spring temos uma interface gráfica, e o documento da nossa classe, além da capacidade de testar.

Com ela conseguimos testar nossas apis

Abra o arquivo pom.xml e adicione as dependências do Swagger conforme código abaixo:

Abaixo de: `<artifactId>spring-boot-starter.test</artifactId>`
`</dependency>`

```
<!-- Swagger documentação-->
	<dependency>
		<groupId>io.springfox</groupId>
		<artifactId>springfox-swagger2</artifactId>
		<version>2.9.2</version>
	</dependency>

	<dependency>
		<groupId>io.springfox</groupId>
		<artifactId>springfox-swagger-ui</artifactId>
		<version>2.9.2</version>
	</dependency>
```

E por boas práticas vamos criar arquivos com toda documentação da nossa aplicação.

Crie um pacote doc, e a classe SwaggerConfig:

```
package Funtec.meuprimeirowebapi.doc;

import org.springframework.context.annotation.Configuration;

import java.util.Arrays;
import java.util.HashSet;

import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import springfox.documentation.builders.ApiInfoBuilder;
import springfox.documentation.builders.PathSelectors;
import springfox.documentation.builders.RequestHandlerSelectors;
import springfox.documentation.spi.DocumentationType;
import springfox.documentation.spring.web.plugins.Docket;
import springfox.documentation.swagger2.annotations.EnableSwagger2;
import springfox.documentation.service.Contact;


@Configuration  //é uma classe de configuração
@EnableSwagger2 //habilitando a doc do swagger
public class SwaggerConfig { //classe de configuração

	//contato da nossa api
	private Contact contato() {
		return new Contact(
				"Seu nome",
				"http//seusite.com.br",
				"shirleifuntec@gmail.com");

	}

	//Informações da API
	private ApiInfoBuilder informacoesApi() {

		ApiInfoBuilder apiInfoBuilder = new ApiInfoBuilder();

		apiInfoBuilder.title("Title - Rest Api");
```

```
            apiInfoBuilder.description("Api exemplo de uso de SpringBoot
Rest Api");
            apiInfoBuilder.version("1.0");
            apiInfoBuilder.termsOfServiceUrl("Termo de usso: Open Source");
            apiInfoBuilder.license("Licença - sua empresa");
            apiInfoBuilder.licenseUrl("http://www.seusite.com.br");
            apiInfoBuilder.contact(this.contato());

            return apiInfoBuilder;

      }

      //como se trata de uma aplicação SpringBoot, vamos criar um
Docket(documento) em forma de @bean
      @Bean
      public Docket detalheApi() {
            Docket docket = new Docket(DocumentationType.SWAGGER_2);


            docket //temos um documento
            //esse docuemnto esta sendo configurado para consumir json
            .select()
            //aui precisamos informar o pacote principal que possui os nosos
controller "pacote.comseus.controllers")

      .apis(RequestHandlerSelectors.basePackage("Funtec.meuprimeirowebapi.co
ntroller"))
            .paths(PathSelectors.any())
            .build()
            .apiInfo(this.informacoesApi().build())
            .consumes(new
HashSet<String>(Arrays.asList("application/json")))
            .produces(new
HashSet<String>(Arrays.asList("application/json")));

                  return docket;
      }


}
```

Aqui rodar...

Aqui teremos uma única interface gráfica, que documenta, ilustra o recursos e permite a realização de testes.

No browser:

Localhost:8080/swagger-ui.html

Clicando em usuários vemos os verbos, e as informações e podemos realizar os testes.

No método post ele traz a estrutura que representa o corpo.

Preencher:

Login: “jose”

Password: “123456”

Executar e ver no terminal que recebemos o usuário.

Vamos ver os novos recursos:

Em ususario conrtoller acrescentar o método put:

```
@PutMapping() //update
	public void putUser(@RequestBody Usuario usuario) { //recebendo um
usuario
	repository.save(usuario);
	}
```

Executar novamente e teremos o verbo but, e se incluirmos o id sera update.

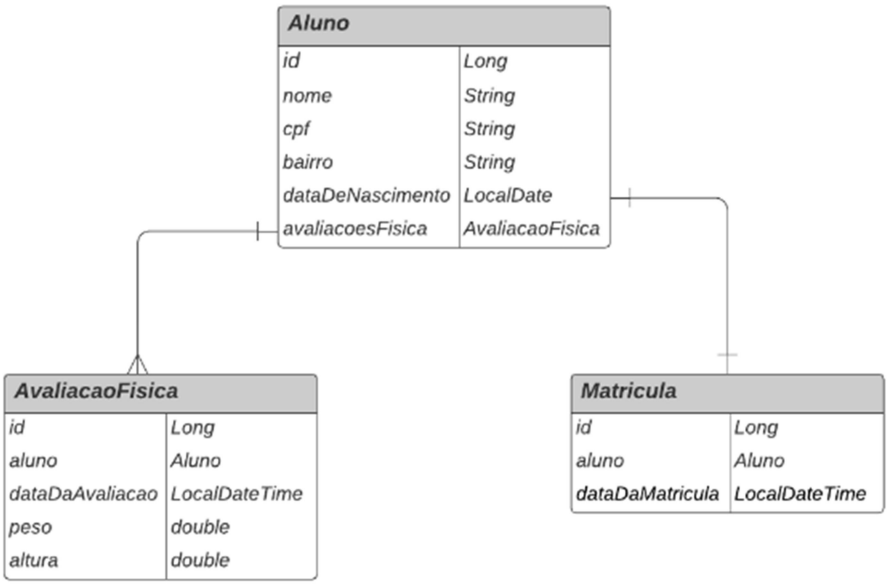
Esta é a proposta e o poder do Swagger.

---

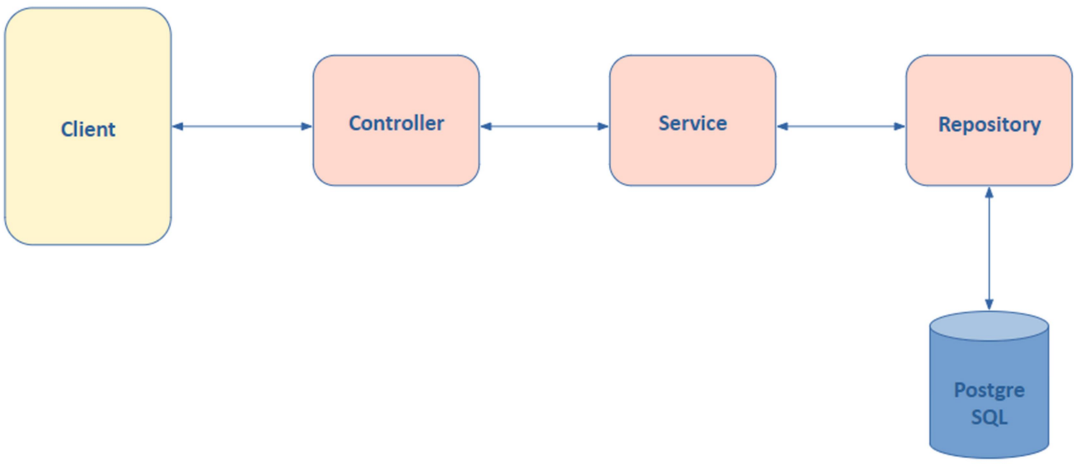
https://github.com/cami-la/academia-digital/blob/master/src/main/java/me/dio/academia/digital/repository/AvaliacaoFisicaRepository.java

Conectando com o banco postgress

Vamos criar uma aplicação de um cadastro de academia.



Fluxo



Links Uteis

- Spring Initializr
- Common application properties
- Spring Data JPA - Reference Documentation
- Validation Reference Implementation: Reference Guide

Anotações de Mapeamento

@Entity Usada para especificar que a classe anotada atualmente representa um tipo de entidade.

@Table Usada para especificar a tabela principal da entidade atualmente anotada.

@Id Especifica o identificador da entidade. Uma entidade deve sempre ter um atributo identificado.

@GeneratedValue Especifica que o valor do identificador de entidade é gerado automaticamente.

@Column Usada para especificar o mapeamento entre um atributo de entidade básico e a coluna da tabela de banco de dados.

@JoinColumn Usada para especificar a coluna FOREIGN KEY. Indica que a entidade é a responsável pelo relacionamento.

@OneToMany Usada para especificar um relacionamento de banco de dados um-para-muitos.

@OneToOne Usada para especificar um relacionamento de banco de dados um-para-um.

@ManyToOne Usada para especificar um relacionamento de banco de dados muitos-para-um.

cascade Realizar operações em cascata só faz sentido em relacionamentos Pai - Filho.

mappedBy Indica qual é o lado inverso ou não dominante da relação

Iniciando o projeto:

No arquivo POm, temos a dependência:

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

Clicando sobre ela vemos as outras dependências, incluindo a JBBD. A do hibernate entre outras.

Temos também a

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

Temos a do postgres

```
<dependency>
	<groupId>org.postgresql</groupId>
		<artifactId>postgresql</artifactId>
		<scope>runtime</scope>
	</dependency>
```

E outras dependências baixadas....

Indo para o sistema:

Pasta entity: salva as entidades do projeto, que tb pode ser chamada de model.

Nesta pasta fica o modelo do nosso sistema.

Temos 3 classes: aluno, AvaliacaoFisca e matricula, que serão transformadas em tabelas no nosso banco de dados.

Classe Aluno:

package me.dio.academia.digital.entity;

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

public class Aluno {

  private Long id;

  private String nome;
```

```
  private String cpf;

  private String bairro;

  private LocalDate dataDeNascimento;

  private List<AvaliacaoFisica> avaliacoes = new ArrayList<>();

}
```

Cada um desses atributos serão criados como coluna na tabela aluno. O mesmo pra AvaliacaoFisica e Matricula.

Como estamos trabalhando com o springBoot a pasta service é onde fica a logica do nosso programa.

As interfaces IAlunoService, IAvaliacaoFisicaService e IMatriculaService respresentam os métodos Crud.

mostrar os métodos.

Na pasta entity/form, temos os DTO, data transfer Object, serve para fazer a comunicação com nosso banco de dados.

Ex.: Quando formos criar um novo aluno, vamos criar através desse formulário:

```
Aluno create(AlunoForm form);
```

No update, não alteraemos o cpf, pois na inclusão tem a validação, então muito difícil termos que alteraro cpf.

Na pacote infra, temos a serealização e a deserelização para utilização nos programas, e trabalharmos com dados no sistema.

Em resources, application propertires temos as configurações do nosso banco de dados.

Se rodarmos o projeto teremos a resposta:

Failed to configure a DataSource: ‘url’ atribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class

Criando na pasta resources o arquivo: application.yml

jdbc:postgresql://localhost:5432/academia

porta do ppostgress 5432

nome do banco: academia

```
spring:
  datasource:
    url: //com timezone
jdbc:postgresql://localhost:5432/academia?useTimezone=true&serverTimezone=UTC
&useLegacyDate
    username: postgres
    password: treinamento
```

podemos rodar o projeto mas vai dar que o database academia não existe.

Configurando o JPA

```
jpa:
    show-sql: true  /mostra o sql
    hibernate:
      ddl-auto: update /configuração do hibernate, cada vez q inicia faz as
alterações
    properties:
      hibernate.format_sql: true
```

--

Criar o banco de dados academia

Create database academia;

Rodar a aplicação e o estará ok.

--

Agora vamos criar nossas tabelas, implementado aluno, matricula e avaliacaoFisicca

Aluno.java

```
package me.dio.academia.digital.entity;

import java.time.LocalDate;
```

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

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

//@Data serve pra implementar o get e set
//@NoargsContructor  cria um contrutor vazio e o hibernate precisa desse
contrutor
//@AllArgsConstructor possui todos os atributos
//@Entity para a conexão com o banco, ela precisa de uma chave primaria
// @Table dá nome a tabela
// @JsonIgnoreProperties({"hibernateLazyInitializer", "handler"}) -
inicialização lenta
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "tb_alunos")
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Aluno {

	@Id //chave primária
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	 private Long id;

  private String nome;

  @Column(unique = true) //notacao dizendo que é unico
  private String cpf;

  private String bairro;

  //no java escrevemos com camel case, mas no banco ele ficara
Data_de_nascimento
  private LocalDate dataDeNascimento;

//relacionando com a tabela avaliacaoFisica
//aluno, tera varias avaliações fisicas
  @OneToMany(mappedBy = "aluno", fetch = FetchType.LAZY) //Retorna as
informações de aluno, menos as avaliaçoes (lazy)
 @JsonIgnore
  private List<AvaliacaoFisica> avaliacoes = new ArrayList<>();

}
```

Avaliação Física

```
package me.dio.academia.digital.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import lombok.RequiredArgsConstructor;

import javax.persistence.*;
import java.time.LocalDateTime;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "tb_avaliacoes")
public class AvaliacaoFisica {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  //varias avaliações pra um aluno
  @ManyToOne(cascade = CascadeType.ALL) ////td q fizer em avaliacao vai
refletir em aluno
  @JoinColumn(name = "aluno_id")
  private Aluno aluno;

  private LocalDateTime dataDaAvaliacao = LocalDateTime.now();

  @Column(name="peso_atual")
  private double peso;

  @Column(name="altura_atual")
  private double altura;

}
```

Matricula

```
package me.dio.academia.digital.entity;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.persistence.*;
import java.time.LocalDateTime;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "tb_matriculas")
```

```
public class Matricula {

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

  @OneToOne(cascade = CascadeType.ALL) //uma matricula so tem um aluno
  //cascade  = ex se eu excluir uma maricula eu excluo o aluno
  @JoinColumn(name = "aluno_id")
  private Aluno aluno;

  private LocalDateTime dataDaMatricula = LocalDateTime.now();
}
```

Rodar o projeto e ver que a tabela foi criada.

Incluindo os dados:

Ir em Controller, AlunoController

```
package me.dio.academia.digital.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import me.dio.academia.digital.service.impl.AlunoServiceImpl;

@RestController
@RequestMapping("/alunos")
public class AlunoController {

  @Autowired //chamando o service
  private AlunoServiceImpl service; //temos que criar pacote /service/impl e
a classe AlunoServiceImpl

}
```

Necessário criar o pacote impl e Classe Service/impl/AlunoServiceImpl

Nesta classe é onde fica toda a lógica

Aqui precisamos extends a interface e implementar os métodos

```
package me.dio.academia.digital.service.impl;

import java.util.List;

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

```java
import me.dio.academia.digital.entity.Aluno;
import me.dio.academia.digital.entity.form.AlunoForm;
import me.dio.academia.digital.entity.form.AlunoUpdateForm;
import me.dio.academia.digital.service.IAlunoService;

@Service
public class AlunoServiceImpl implements IAlunoService {

	@Override
	public Aluno create(AlunoForm form) {
			return null;
	}

	@Override
	public Aluno get(Long id) {
			return null;
	}

	@Override
	public List<Aluno> getAll() {
				return null;
	}

	@Override
	public Aluno update(Long id, AlunoUpdateForm formUpdate) {
			return null;
	}

	@Override
	public void delete(Long id) {

	}

}
```

Voltar ao AlunoServiceImpl, que é onde fica toda a logica

```java
@Service
public class AlunoServiceImpl implements IAlunoService {

	@Autowired //chamando o repository
	private AlunoRepository repository;
```

Vai no pacote repository e na classe AlunoRepository

O repositor é quem chega no banco de dados.

Na JPA Repository ele tem os métodos delete, findAll, FindAllId e getFindAllID

Na classe alunoRepository por eqnto não usaremos nenhum método, pq o jpaRepository tem os métodos que precisamos

```java
package me.dio.academia.digital.repository;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;
import me.dio.academia.digital.entity.Aluno;


//aqui extendemos a Classe jparepository com os metodos do banco de dados

public interface AlunoRepository extends JpaRepository<Aluno, Long>{
}
```

Voltar ao AlunoServiceImpl, e implementamos o Public List

```java
@Override //retorna a lista de alunos
public List<Aluno> getAll() {
	return repository.findAll();
	}

}
```

Aqui tudo pronto na tabela.

Temos um controller que chama o service, com o método getAll,

O getAll chama o repository

E o repository traz nossa lista de alunos

Se rodarmos, localhost:8080/alunos vai retronar uma array vazia.

Agora iremos incluir dados no banco de dados

Em alunoController, create aluno

```java
  @PostMapping
  public Aluno create(@RequestBody AlunoForm form) {
    return service.create(form);
  }
```

Depois a logica em alunoserviceImpl

```java
@Override  //aqui com o metodo save iremos salvar no banco de dados
  public Aluno create(AlunoForm form) {
    Aluno aluno = new Aluno();
    aluno.setNome(form.getNome());
    aluno.setCpf(form.getCpf());
    aluno.setBairro(form.getBairro());
    aluno.setDataDeNascimento(form.getDataDeNascimento());
```

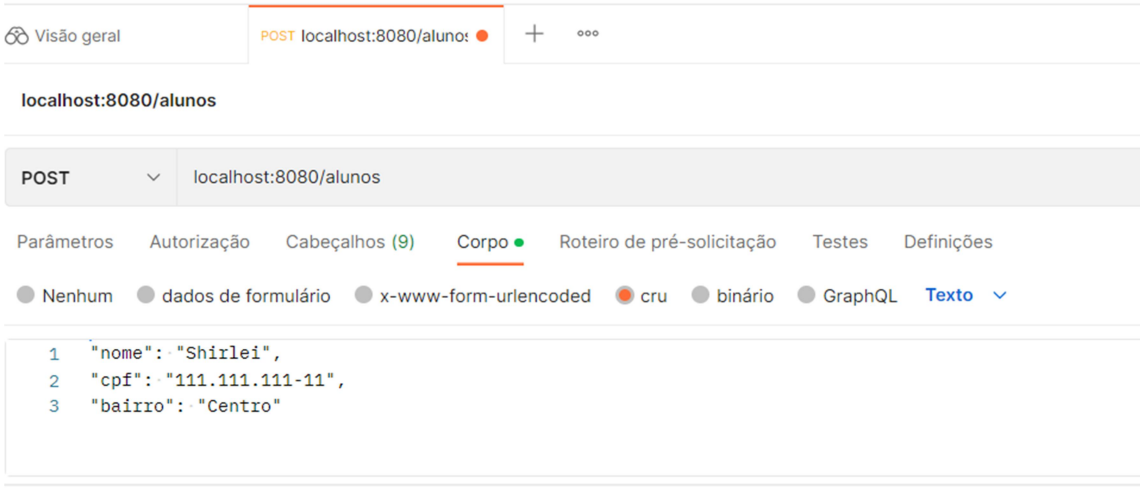
```
    return repository.save(aluno);
  }
```

Clicar em form para ir em AlunoForm e incluir:

```
package me.dio.academia.digital.entity.form;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;


@Data
@NoArgsConstructor
@AllArgsConstructor

public class AvaliacaoFisicaForm {

  private Long alunoId;

  private double peso;

   private double altura;
```

Testar no postman, os nomes dos campos devem ser iguais ao da classe java

Vai no banco de dados e os dados estarão la.



Em aluno controller, criar a lista.

```
  @GetMapping
```

```
  public List<Aluno> getAll(@RequestParam(value = "dataDeNascimento",
required = false)
                                    String dataDeNacimento){
    return service.getAll(dataDeNascimento);
  }
```

Criar métodos em AvaliaçãoFisicaController

```
@RestController
@RequestMapping("/avaliacoes")
public class AvaliacaoFisicaController {

  @Autowired
  private AvaliacaoFisicaServiceImpl service;
```

em service/impl/AvaliacaoFisicaServiceImpl

implementa a interface e os metodos

```
package me.dio.academia.digital.service.impl;

import java.util.List;

import org.springframework.stereotype.Service;

import me.dio.academia.digital.entity.AvaliacaoFisica;
import me.dio.academia.digital.entity.form.AvaliacaoFisicaUpdateForm;
import me.dio.academia.digital.service.AvaliacaoFisicaForm;
import me.dio.academia.digital.service.IAvaliacaoFisicaService;

@Service
public class AvaliacaoFisicaServiceImpl implements IAvaliacaoFisicaService  {

	@Override
	public AvaliacaoFisica create(AvaliacaoFisicaForm form) {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public AvaliacaoFisica get(Long id) {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public List<AvaliacaoFisica> getAll() {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public AvaliacaoFisica update(Long id, AvaliacaoFisicaUpdateForm
formUpdate) {
		// TODO Auto-generated method stub
		return null;
```

```
	}

	@Override
	public void delete(Long id) {
		// TODO Auto-generated method stub

	}

}
```

volta para avaliação fisicaController

```
package me.dio.academia.digital.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import me.dio.academia.digital.entity.AvaliacaoFisica;
import me.dio.academia.digital.entity.form.AvaliacaoFisicaForm;
import me.dio.academia.digital.service.impl.AvaliacaoFisicaServiceImpl;

@RestController
@RequestMapping("/avaliacoes")
public class AvaliacaoFisicaController {


	@Autowired
	  private AvaliacaoFisicaServiceImpl service;

	  @PostMapping //retorna uma avalicação fisica
	  public AvaliacaoFisica create(@RequestBody AvaliacaoFisicaForm form)
{
	    return service.create(form);
	  }
}
```

Vai em avaliacaoFisicaRepository para implementar a classe JPA

```
package me.dio.academia.digital.repository;

import me.dio.academia.digital.entity.AvaliacaoFisica;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface AvaliacaoFisicaRepository extends
JpaRepository<AvaliacaoFisica, Long> {

}
```

Volta em service/impl/AvaliacaoFisicaServiceImpl para implementar os métodos:

```
package me.dio.academia.digital.service.impl;

import java.util.List;

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

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import me.dio.academia.digital.entity.Aluno;
import me.dio.academia.digital.entity.AvaliacaoFisica;
import me.dio.academia.digital.entity.form.AvaliacaoFisicaForm;
import me.dio.academia.digital.entity.form.AvaliacaoFisicaUpdateForm;
import me.dio.academia.digital.repository.AlunoRepository;
import me.dio.academia.digital.repository.AvaliacaoFisicaRepository;
import me.dio.academia.digital.service.IAvaliacaoFisicaService;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Service
public class AvaliacaoFisicaServiceImpl implements IAvaliacaoFisicaService  {

        @Autowired  //chama a avaliação repository
        private AvaliacaoFisicaRepository avaliacaoFisicaRepository;

       @Autowired //para podermos trazer os alunos
       private AlunoRepository alunoRepository;


        public AvaliacaoFisica create(AvaliacaoFisicaForm form) {
          AvaliacaoFisica avaliacaoFisica = new AvaliacaoFisica();
          //precisamos buscar o aluno pelo id
          //findById é um metodo da JPA repository
          Aluno aluno = alunoRepository.findById(form.getAlunoId()).get();

          avaliacaoFisica.setAluno(aluno);
          avaliacaoFisica.setPeso(form.getPeso());
          avaliacaoFisica.setAltura(form.getAltura());

          //salvar no banco de dados
          return avaliacaoFisicaRepository.save(avaliacaoFisica);
        }


      @Override
      public AvaliacaoFisica get(Long id) {
            // TODO Auto-generated method stub
            return null;
      }


      @Override
      public List<AvaliacaoFisica> getAll() {
            // TODO Auto-generated method stub
            return null;
```

```
	}


	@Override
	public AvaliacaoFisica update(Long id, AvaliacaoFisicaUpdateForm
formUpdate) {
		// TODO Auto-generated method stub
		return null;
	}


	@Override
	public void delete(Long id) {
		// TODO Auto-generated method stub

	}


}  }
```

Em aluno Controller, criar métodos

```
package me.dio.academia.digital.controller;


import java.util.List;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.RestController;

import me.dio.academia.digital.entity.Aluno;
import me.dio.academia.digital.entity.form.AlunoForm;
import me.dio.academia.digital.service.impl.AlunoServiceImpl;

@RestController
@RequestMapping("/alunos")
public class AlunoController {

  @Autowired //chamando o service
  private AlunoServiceImpl service;  //temos que criar pacote /service/impl e
a classe AlunoServiceImpl


  @PostMapping
  public Aluno create(@RequestBody AlunoForm form) {
    return service.create(form);
  }

  @GetMapping
  public List<Aluno> getAll(@RequestParam(value = "dataDeNascimento",
required = false)
```

```
                                    String dataDeNacimento){
    return service.getAll();
  }

//retorna todas as avaliaçções fisicas do aluno
  @GetMapping("/avaliacoes/{id}")
  public List<AvaliacaoFisica> getAllAvaliacaoFisicaId(@PathVariable Long id)
{
    return service.getAllAvaliacaoFisicaId(id);
  } package me.dio.academia.digital.controller;

import me.dio.academia.digital.entity.Aluno;
import me.dio.academia.digital.entity.AvaliacaoFisica;
import me.dio.academia.digital.entity.form.AlunoForm;
import me.dio.academia.digital.service.impl.AlunoServiceImpl;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.List;

@RestController
@RequestMapping("/alunos")
public class AlunoController {

  @Autowired
  private AlunoServiceImpl service;

  @PostMapping
  public Aluno create(@Valid @RequestBody AlunoForm form) {
    return service.create(form);
  }

  @GetMapping("/avaliacoes/{id}")
  public List<AvaliacaoFisica> getAllAvaliacaoFisicaId(@PathVariable Long id)
{
    return service.getAllAvaliacaoFisicaId(id);
  }

  @GetMapping
  public List<Aluno> getAll(@RequestParam(value = "dataDeNascimento",
required = false)
                                    String dataDeNacimento){
    return service.getAll();
  }
}

}
```

Na interface IalunoService,

```
List<AvaliacaoFisica>getAllAvaliacaoFisicaId(Long id);
```

Em Aluno ServiceImpl vamos colocar a logica

```
@Override
	public List<me.dio.academia.digital.entity.AvaliacaoFisica>
getAllAvaliacaoFisicaId(Long id) {
		repository.findById(id).get();
		return null;
	}
```

Vamos no browser

localhost:8080/aluno9

no potman incluir:

“alunoId”: 1,

“peso”: 60.0,

“altura”: 175.5

Revisão:

Em applicarion.yml

Acrescentar:

```
logging:
  level:
    org:
      hibernate:
        type: trace
```

Em entidy temos as entidades que irão representar nossas tabelas no BD.

Criar tudo tb para Matricula

```
package me.dio.academia.digital.controller;

import me.dio.academia.digital.entity.Matricula;
import me.dio.academia.digital.entity.form.MatriculaForm;
import me.dio.academia.digital.service.impl.MatriculaServiceImpl;
```

```java
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import javax.validation.Valid;
import java.util.List;

@RestController
@RequestMapping("/matriculas")
public class MatriculaController {

  @Autowired
  private MatriculaServiceImpl service;

  @PostMapping
  public Matricula create(@Valid @RequestBody MatriculaForm form) {
    return service.create(form);
  }

  @GetMapping
  public List<Matricula> getAll(@RequestParam(value = "bairro", required =
false) String bairro) {
    return service.getAll(bairro);
  }

}
```

```java
package me.dio.academia.digital.entity.form;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Positive;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class MatriculaForm {

  @NotNull(message = "Preencha o campo corretamente.")
  @Positive(message = "O Id do aluno precisa ser positivo.")
  private Long alunoId;

public Long getAlunoId() {
      // TODO Auto-generated method stub
      return null;
}

}
```

```java
package me.dio.academia.digital.repository;

import me.dio.academia.digital.entity.Matricula;
```

```java
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface MatriculaRepository extends JpaRepository<Matricula, Long> {

  /**
   *
   * @param bairro bairro referência para o filtro
   * @return lista de alunos matriculados que residem no bairro passado como
parâmetro
   */
  @Query(value = "SELECT * FROM tb_matriculas m " +
      "INNER JOIN tb_alunos a ON m.aluno_id = a.id " +
      "WHERE a.bairro = :bairro", nativeQuery = true)
  //@Query("FROM Matricula m WHERE m.aluno.bairro = :bairro ")
  List<Matricula> findAlunosMatriculadosBairro(String bairro);

  //List<Matricula> findByAlunoBairro(String bairro);


}
```

Agora iremos criar uma matricula, e um aluno

```java
package me.dio.academia.digital.entity.form;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import org.hibernate.validator.constraints.br.CPF;

import javax.validation.constraints.*;
import java.time.LocalDate;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class AlunoForm {

  @NotEmpty(message = "Preencha o campo corretamente.")
  @Size(min = 3, max =50, message = "'${validatedValue}' precisa estar entre
{min} e {max} caracteres.")
  private String nome;

  @NotEmpty(message = "Preencha o campo corretamente.")
  @CPF(message = "'${validatedValue}' é inválido!")
  private String cpf;

  @NotEmpty(message = "Preencha o campo corretamente.")
  @Size(min = 3, max =50, message = "'${validatedValue}' precisa estar entre
{min} e {max} caracteres.")
  private String bairro;

  @NotNull(message = "Prencha o campo corretamente.")
```

```
  @Past(message = "Data '${validatedValue}' é inválida.")
  private LocalDate dataDeNascimento;
}
```

```
package me.dio.academia.digital.entity.form;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

import javax.validation.constraints.NotNull;
import javax.validation.constraints.Positive;

@Data
@NoArgsConstructor
@AllArgsConstructor
public class MatriculaForm {

  @NotNull(message = "Preencha o campo corretamente.")
  @Positive(message = "O Id do aluno precisa ser positivo.")
  private Long alunoId;


}
```

Ir no potman

Id: 1,

Nome: “Jose”;

Cpf: “111.111.111-11”,

Bairro: “Campos Eliseos”,

dataDeNascimento: “03/04/1980”

se digitarmos algo invalido, no terminal a ultima msg mostra o erro.

Consultas avançadas não podem ser feitas pela JPA

```
Aqui ira listar todos os alunos com a data de nasciemtno informada.

Posso trocar por exemplo o parâmetro por bairro
```

```java
package me.dio.academia.digital.repository;

import me.dio.academia.digital.entity.Aluno;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.time.LocalDate;
import java.util.List;

@Repository
public interface AlunoRepository extends JpaRepository<Aluno, Long> {

  /**
   *
   * @param dataDeNascimento: data de nascimento dos alunos
   * @return lista com todos os alunos com a data de nascimento passada como
parâmetro da função
   */
  List<Aluno> findByDataDeNascimento(LocalDate dataDeNascimento);

}
```

E no aluno controller tenho o método

```java
  @GetMapping
  public List<Aluno> getAll(@RequestParam(value = "dataDeNascimento",
required = false)
                                  String dataDeNacimento){
    return service.getAll();
  }
}
```

E em service/alunoserviceImpl

```java
@Override
              public List<Aluno> getAll(String dataDeNascimento) {


                if(dataDeNascimento == null) {
                  return repository.findAll();
                } else {
                  LocalDate localDate = LocalDate.parse(dataDeNascimento,
            JavaTimeUtils.LOCAL_DATE_FORMATTER);
                  return repository.findByDataDeNascimento(localDate);
                }


              }
```

Executar no navegador localhost:8080/alunos

No potman passar os parâmetros

Get

Parâmetros

Key dataDeNascimento

Value : a data

Outra forma

MatriculaController

Aqui passo os alunos

```
@GetMapping
                public List<Matricula> getAll(@RequestParam(value = "bairro",
              required = false) String bairro) {
                   return service.getAll(bairro);
                }
```

Matricula repository

```
/**
           *
           * @param bairro bairro referência para o filtro
           * @return lista de alunos matriculados que residem no bairro passado como
        parâmetro
           */
          @Query(value = "SELECT * FROM tb_matriculas m " +
              "INNER JOIN tb_alunos a ON m.aluno_id = a.id " +
              "WHERE a.bairro = :bairro", nativeQuery = true)
          //@Query("FROM Matricula m WHERE m.aluno.bairro = :bairro ")
          List<Matricula> findAlunosMatriculadosBairro(String bairro);
```

```
        //List<Matricula> findByAlunoBairro(String bairro);
```

No postman

Key bairro

Valor nomedobairro

Vantagens de usar Spring Boot

Permite criar APIs REST com configurações mínimas . Alguns benefícios do uso do Spring Boot para suas APIs REST incluem: Não há necessidade de configurações XML complexas. Servidor Tomcat incorporado para executar aplicativos Spring Boot.

O que é Maven e para que serve?

O Apache **Maven** é uma ferramenta de automação e gerenciamento de projetos Java, embora também possa ser utilizada com outras linguagens. Ela fornece às equipes de desenvolvimento uma forma padronizada de automação, construção e publicação de suas aplicações, agregando agilidade e qualidade ao produto final.

Tom cat

O Tomcat é um servidor web Java, mais especificamente, um container de servlets. O Tomcat implementa, dentre outras de menor relevância, as tecnologias Java Servlet e JavaServer Pages e não é um container Enterprise JavaBeans. Desenvolvido pela Apache Software Foundation, é distribuído como software livre.

Sobre json

https://www.w3schools.com/js/js_json_intro.asp

https://blog.geekhunter.com.br/xml-vs-json-entenda-como-fazer-a-melhor-escolha/

O que faz o Hibernate?

Hibernate é uma ferramenta para mapeamento objeto/relacional para ambientes Java. O termo mapeamento objeto/relacional (ORM) refere-se à técnica de mapeamento de uma representação de dados em um modelo de objetos para um modelo de dados relacional baseado em um esquema E/R.

```java
package me.dio.academia.digital.entity;

import java.time.LocalDate;
import java.util.ArrayList;
import java.util.List;

import javax.persistence.Column;
import javax.persistence.Entity;
import javax.persistence.FetchType;
import javax.persistence.GeneratedValue;
import javax.persistence.GenerationType;
import javax.persistence.Id;
import javax.persistence.OneToMany;
import javax.persistence.Table;

import com.fasterxml.jackson.annotation.JsonIgnore;
import com.fasterxml.jackson.annotation.JsonIgnoreProperties;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;

//@Data serve pra implementar o get e set
//@NoargsContructor  cria um contrutor vazio e o hibernate precisa desse
contrutor
//@AllArgsConstructor possui todos os atributos
//@Entity para a conexão com o banco, ela precisa de uma chave primaria
// @Table dá nome a tabela
// @JsonIgnoreProperties({"hibernateLazyInitializer", "handler"}) -
inicialização lenta
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "tb_alunos")
@JsonIgnoreProperties({"hibernateLazyInitializer", "handler"})
public class Aluno {

	@Id //chave primária
	@GeneratedValue(strategy = GenerationType.IDENTITY)
	 private Long id;

  private String nome;

  @Column(unique = true) //notacao dizendo que é unico
  private String cpf;

  private String bairro;

  //no java escrevemos com camel case, mas no banco ele ficara
Data_de_nascimento
  private LocalDate dataDeNascimento;

//relacionando com a tabela avaliacaoFisica
//aluno, tera varias avaliações fisicas
  @OneToMany(mappedBy = "aluno", fetch = FetchType.LAZY) //Retorna as
informações de aluno, menos as avaliaçoes (lazy)
 @JsonIgnore
  private List<AvaliacaoFisica> avaliacoes = new ArrayList<>();
```

```
}
```

```
package me.dio.academia.digital.entity.form;

import lombok.AllArgsConstructor;
import lombok.Data;
import lombok.NoArgsConstructor;
import me.dio.academia.digital.entity.Aluno;

import javax.persistence.*;
import java.time.LocalDateTime;

@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "tb_avaliacoes")
public class AvaliacaoFisicaForm {
  @Id
  @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Long id;

  //varias avaliações pra um aluno
  @ManyToOne(cascade = CascadeType.ALL) ////td q fizer em avaliacao vai
refletir em aluno
  @JoinColumn(name = "aluno_id")
  private Aluno aluno;

  private LocalDateTime dataDaAvaliacao = LocalDateTime.now();

  @Column(name="peso_atual")
  private double peso;

  @Column(name="altura_atual")
  private double altura;

}
```