Projeto Sumo

IDE: Spring Tool Suite

* Criar o banco no MySQL (create database sumo).
* Criar o projeto no Spring - Create new Spring Starter Project

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Maven: gerenciador de dependências e automação de build (versão compilada do software aqui, arquivo Jar)

Cria na nuvem

Cria na nuvem

* Selecionar os starters:

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Starter (dependência): uma das principais funcionalidades do spring boot. Cada starter carrega consigo várias outras dependências transitivas

Verificar compatibilidade com a jdk 11 em installed JRE´s

Interface gráfica do usuário

Descrição gerada automaticamente

pom.xml: é um arquivo do Maven. Contém informações e configurações do projeto usadas pelo Maven para fazer o build do projeto.

* Configurar o arquivo application.properties

Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

Src/main/resources pasta que contém arquivos estáticos/de configuração

spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5InnoDBDialect

spring.jpa.hibernate.ddl-auto = create-drop

spring.sql.init.mode=always

spring.datasource.url = jdbc:mysql://localhost:3306/sumo?serverTimezone=UTC

spring.datasource.username = root

spring.datasource.password = admin

* As classes que serão criadas nos diferentes pacotes a partir daqui, devem ser criadas a partir do pacote principal (pai) criado pelo Spring

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

A classe criada pelo Spring (ProjetoSumoApplication.java) é a classe que faz o start da aplicação Spring Boot.

Não é necessário rodar essa classe, mas se quiser, pode. Outra maneira é clicar no start da janela Boot Dashboard.Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

Ao fazer isso a classe levanta o servlet container que é o TomCat.

* Criar a classe HomeController com as anotações e um syso para saber se funcionou no console.

Texto

Descrição gerada automaticamente

* Rodar a aplicação a partir da classe padrão criada pelo Spring. Se aparecer “funcionou” no console é pq....funcionou!
* Criar a entidade Rikishi no pacote entity:

Tabela:

Logotipo

Descrição gerada automaticamente

Chave primaria:

Uma imagem contendo Texto

Descrição gerada automaticamente

Colunas:

Texto

Descrição gerada automaticamente

+ construtor e getters e setters. Salvar e verificar no banco se foi criada corretamente. Se tudo certo, alterar o arquivo application.properties para “update” e colocar never na linha do “Always”

* Criar a classe RikishiController

Essa classe que trata as requisições HTTP e devolve uma resposta. A anotação @RestController significa que é uma classe para desenvolver os endpoints rest. A anotação @RequestMapping vai mapear os endpoints adicionados nesta classe para receber requisições iniciadas com /rikishi. Quando chegar uma requisição /rikishi é nessa classe que serão tratadas.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

* Criar a interface RikishiRepository no pacote repositor.

Essa classe é responsável por interagir com o banco de dados com uma determinada entidade. Não é necessário implementar nada pois o spring data JPA já fornece a implementação em tempo de execução.



Injetar na classe RikishiController com a anotação @Autowired. Essa anotação vai injetar uma instância de RikishiRepository no atributo RikishiRepository criado.Interface gráfica do usuário, Texto, Aplicativo

Descrição gerada automaticamente

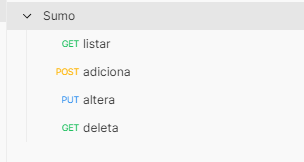
* Criar um file na pasta src/main/resources com a extensão .sql e popular a tabela rikishi

INSERT INTO RIKISHI (shikona, heya, pos\_rank) values ('Shodai', 'Tokitsukaze', 'Ozeki');

POSTMAN: é uma ferramenta para testar a API fazendo requisições HTTP.

Criar uma coleção para deixar todos os requests juntos.

Interface gráfica do usuário, Aplicativo

Descrição gerada automaticamente

GET (PEGAR): dar um nome a request com o verbo GET, adicionar o endereço criado na tabela entidade, salvar e send.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente

POST (INSERIR): criar a request no mesmo esquema. Para inserir o registro: colar os atributos na caixa do post, selecionando body –> opções raw e JSON. Apagar a linha do ID pois é criada pelo banco. Alterar as informações para a nova inclusão, salvar e enviar.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente



PUT (ALTERAR): Para alterar alguma informação a linha de id deve ser especificada. Alterar o que for necessário para aquele id e salvar

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente



DELETE: o método foi construído para ser deletado por id então na caixa da request digitar somente o número correspondente ao id a ser deletado.

Interface gráfica do usuário, Texto, Aplicativo, Email

Descrição gerada automaticamente



Camada service: camada intermediaria entre o controller e o repository. Quando houver chamada do controller em vez de ele chamar o repository ele vai solicitar ao service. Por isso injetar o repository no service.

- o controller aciona o service que vai acionar o repository

Dto rikishi: todos os campos que o cliente vai enviar para salvar

Métodos:

@PostMapping

**public** ResponseEntity<Object> saveRikishi (@RequestBody @Valid RikishiDto rikishiDto) {

**var** rikishiModel = **new** RikishiModel();

BeanUtils.*copyProperties*(rikishiDto, rikishiModel);

**return** ResponseEntity.*status*(HttpStatus.***CREATED***).body(rikishiService.save(rikishiModel));

}

Traduzindo:

responseEntity para montar a resposta com o status e o corpo da resposta. <Object> pq vai mudar o tipo de retorno dependendo das verificações que fizer.

saveRikishi = nome do método

() = vamos receber como parâmetro o DTO que virá como um json. @valid é para as validações serem realizadas.

**new** RikishiModel() = iniciada uma instancia do rikishi model

beanUtils = conversão do DTO em model. Método copyProperties faz isso

return = método save recebe um rikishiModel, salva, e retorna esse model com o id inclusive. Esse método save é feito pela ide em service, só precisa construir o retorno.

Interface gráfica do usuário, Texto

Descrição gerada automaticamente

A anotação transactional é importante em métodos que fazem delete ou create em cascata pq se der algo errado é garantido fazer o rollback.

Método get:

Uma imagem contendo Aplicativo

Descrição gerada automaticamente

Traduzindo:

Vai ter como retorno um objeto pq se o rikishi não existir o retorno será que o objeto procurado não existe mas se existir nos enviamos a resposta.

Para obter o id do url nos utilizamos o @pathVariable (= valor aleatório dependendo do id)

Através do id a info vai ser buscada na base de dados pra isso criamos um novo método (findById) dentro do service que retorna um optional de rikishi model (criar o findbyid pela ide)

Verificação: se o id do rikishi não estiver presente na base retornar: riishi noa encontrado, senão, retorna o optional com o método get.