

Linguagens de Programação Dinâmica

CRUD REST using Spring Boot 2, Hibernate, JPA, and MySQL

Por Loiane Groner ACE

Publicado em Fevereiro 2019

Revisado por Elder Moraes

Nesse artigo será demonstrado como desenvolver uma API REST para um CRUD (Create, Read, Update e Delete - em português Criar, Ler, Atualizar e Remover) utilizando Spring Boot 2, Hibernate, JPA e MySQL.

Para completar o desenvolvimento deste tutorial será necessário as seguintes ferramentas:

- Java JDK (v8+) (<https://www.oracle.com/technetwork/java/javase/downloads/index.html>)
- Maven (v3+) (<https://maven.apache.org/download.cgi>)
- MySQL + MySQL Workbench (última versão disponível) (<https://dev.mysql.com/downloads/workbench/>)

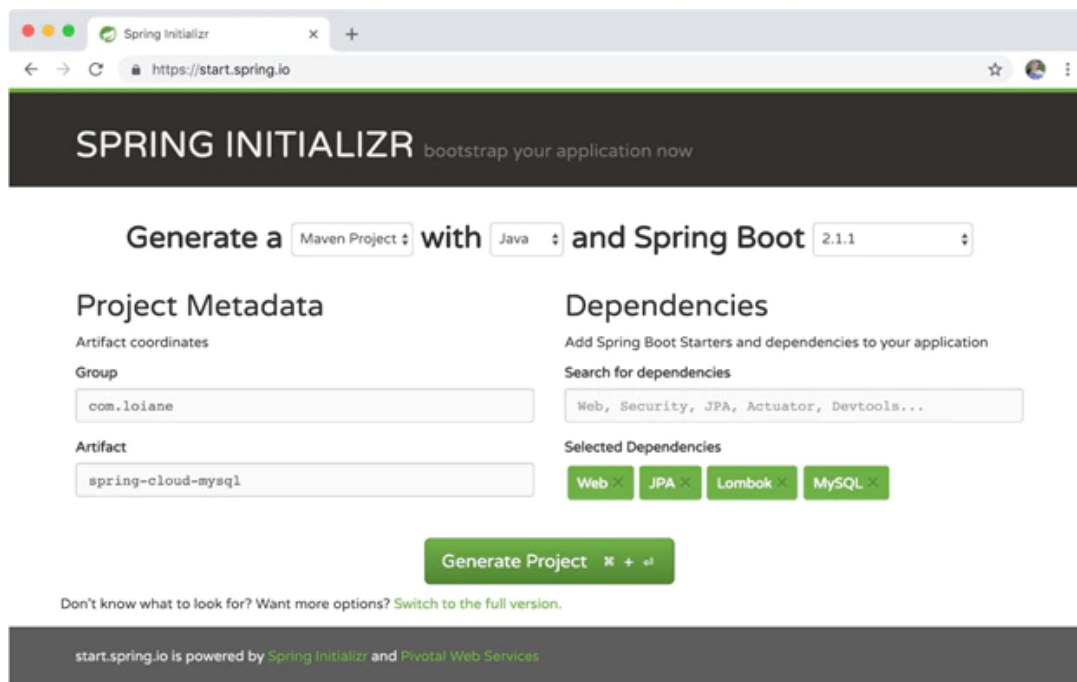
A API que será desenvolvida

Será criado um Controller para uma API de Contatos que irá expor cinco métodos HTTP (URIs RESTful) definidos abaixo:

- Listar todos os contatos - `@GetMapping("/contacts")`
- Obter um contato específico pelo ID - `@GetMapping("/contacts/{id}")`
- Remover um contato pelo ID - `@DeleteMapping("/contacts/{id}")`
- Criar um novo contato - `@PostMapping("/contacts")`
- Atualizar detalhes de um contato - `@PutMapping("/contacts/{id}")`

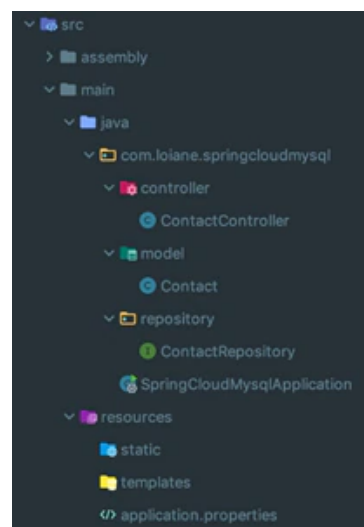
Criando o projeto com Spring Boot

O primeiro passo é criar o projeto com Spring Boot. Pode-se utilizar o serviço do <https://start.spring.io>, que fornece um projeto pronto para ser importado por uma IDE, além de uma classe main e arquivo pom.xml do Maven com as dependências. Para este exemplo, as dependências necessárias são: Web, JPA, Lombok e MySQL como demonstrado na imagem abaixo:



Após entrar todos os detalhes, clique em "Generate Project", faça download do arquivo .zip, extraia o conteúdo para um workspace e abra o projeto em uma IDE.

A imagem abaixo mostra a estrutura do projeto que será criado:



Criando a classe de entidade JPA

A primeira classe que será criada é a entidade JPA, ou seja, a classe que representa a tabela que está no banco de dados. A classe se chamará Contact com uma chave primária id:

```
@AllArgsConstructor
@NoArgsConstructor
@Data
@Entity
public class Contact {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
```



```
private String name;  
private String email;  
private String phone;  
}
```

As seguintes anotações (annotations) são do projeto Lombok que ajuda a manter o código mais limpo e enxuto já que não é necessário gerar os métodos getters e setters, além dos construtores (esse código será gerado e estará presente nos arquivos .class quando o código for compilado).

- **AllArgsConstructor**: cria automaticamente um construtor com todas os atributos da classe como argumento.
- **NoArgsConstructor**: cria automaticamente um construtor vazio (sem argumentos).
- **Data**: cria automaticamente os métodos toString, equals, hashCode, getters e setters.

É necessário configurar o projeto Lombok na IDE para que essas anotações funcionem corretamente, senão o código apresentará problemas de compilação quando se tentar usar algum método getter ou setter (por exemplo). Caso esteja usando o Eclipse siga os passos descritos no link (<https://projectlombok.org/setup/eclipse>) e caso esteja usando IntelliJ IDEA instale o plugin descrito no link (<https://plugins.jetbrains.com/plugin/6317-lombok-plugin>). No link do projeto (<https://projectlombok.org/>) também pode encontrar os passos para outras IDEs e editores.

A anotação **@Entity** pertence ao JPA e isso significa que a classe será automaticamente mapeada à tabela com o mesmo nome (classe **Contact** e tabela **Contact**). Todos os atributos dessa classe também serão mapeados com as respectivas colunas. Podemos omitir a anotação **@Column** para cada atributo da classe desde que o nome do atributo seja o mesmo nome da coluna. Caso a coluna tenha o nome diferente do atributo precisamos especificar assim (supondo que o nome da coluna seja **contact_name**):

```
@Column(name="contact_name")  
private String name;
```



Para cada entidade do JPA, também é necessário identificar qual atributo é a chave primária. Isso é feito através da anotação **@Id**. E com a anotação **@GeneratedValue** serve para identificar como a coluna id será gerada. Nesse caso será definido pelo próprio banco de dados.

O script SQL a ser usado para criar a tabela correspondente encontra-se abaixo:

```
CREATE TABLE `mydatabase`.`contact` (  
  `id` INT NOT NULL,
```

```
`name` VARCHAR(255) NULL,  
`email` VARCHAR(255) NULL,  
`phone` VARCHAR(45) NULL,  
PRIMARY KEY (`id`));  
  
ALTER TABLE `mydatabase`.`contact`  
CHANGE COLUMN `id` `id` INT(11) NOT NULL AUTO_INCREMENT ,  
ADD UNIQUE INDEX `id_UNIQUE` (`id` ASC);
```

Através do AUTO_INCREMENT, não é necessário fornecer o id para criação de um

Criando o repositório JPA

Com a classe modelo criada, o próximo passo é criar o repositório (ou DAO: Data Access Object) que irá fornecer os métodos para as operações CRUD. Uma forma fácil de fazer isso é criar uma interface que estende a interface JpaRepository (do Spring Data):

```
@Repository  
public interface ContactRepository extends JpaRepository<contact, Long> {  
    </contact,>
```

 Copy

A interface JpaRepository precisa de dois parâmetros do tipo Generics: o primeiro é a entidade JPA que representa a tabela e o segundo é o tipo da chave primária (o mesmo tipo do atributo id).

Criando o Controller

O Controller é a classe responsável por expor cada URI que estará disponível na API. O código inicial está listado abaixo:

```
@RestController  
@RequestMapping("/contacts")  
public class ContactController {  
  
    private ContactRepository repository;  
  
    ContactController(ContactRepository contactRepository) {  
        this.repository = contactRepository;  
    }  
    // métodos do CRUD aqui  
}
```

 Copy

A anotação **@RestController** contém as anotações **@Controller** e **@ResponseBody** (pode omitir essas duas para deixar o código mais limpo). A anotação **@Controller** representa uma classe com endpoints (URLs que serão expostas pela API) e a classe indica que o valor retornado pelos métodos devem ser vinculados ao corpo da resposta (response body).

A anotação **@RequestMapping("/contacts")** indica que a URL da API desse controller começa com **/contacts**, isso significa que pode-se acessar usando a URL `http://localhost:8080/contacts` (acesso local).

Como o controller irá acessar o repositório diretamente (como este é um exemplo simples, a camada de serviço está sendo omitida, porém é sempre uma boa prática usar uma classe Service que evoca o repositório e contém a lógica de negócio do projeto para deixar o código da classe controller enxuto e mais limpo), é necessário declarar o repositório como atributo.

O Spring automaticamente fornece a injeção de dependência. Este exemplo não está usando a anotação **@Autowired** pois não é mais considerado uma boa prática para injeção de dependência de atributos obrigatórios. Desde a versão 4 do Spring a prática recomendada é o uso de injeção de dependência por construtor (as IDEs mais modernas inclusive apresentam um alerta quando fazemos o uso do **@Autowired**).

Cada método do controller será declarado a seguir.

Listando todos os contatos (GET /contacts)

O código para listar todos os contatos está listado abaixo:

```
@GetMapping
public List findAll(){
    return repository.findAll();
}
```

 Copy

O método `findAll` é direto ao ponto: utiliza o método **findAll** da interface `JpaRepository` que faz um `select * from contacts`.

Como esta é uma API RESTful, pode-se omitir o código `@RequestMapping(value="/contacts", method=RequestMethod.GET)` e utilizar somente a anotação **@GetMapping**

Por padrão, o formato do resultado será um JSON.

Obtendo um contato específico pelo ID (GET /contacts/{id})

O código para listar apenas um contato buscando pelo seu ID está listado abaixo:

```
@GetMapping(path =("/{id}"))
public ResponseEntity findById(@PathVariable long id){
    return repository.findById(id)
        .map(record -> ResponseEntity.ok().body(record))
        .orElse(ResponseEntity.notFound().build());
}
```



```
}
```

Seguindo os conceitos RESTful, é necessário passar na URL o ID do registro. A anotação **@PathVariable** vincula o parâmetro passado pelo método com a variável do path. Note que o parâmetro long id tem o mesmo nome do path declarado em **@GetMapping(path = {"/{id}"})**.

A lógica para obter um contato específico é utilizar o método **findById** da interface JpaRepository (que irá fazer um select * from contacts where id = ?). Caso o registro seja encontrado, é retornado o status HTTP 200 (ResponseEntity.ok()) e no corpo da resposta é enviado o registro. Caso o registro não seja encontrado é retornado o status HTTP 404 - Não Encontrado (ResponseEntity.notFound()).

Um ponto importante é notar a diferença entre os métodos **findAll** e **findById** da interface JpaRepository. O método **findAll** retorna uma lista de entidades. Caso não exista nenhum registro na tabela, é retornada uma lista vazia ([]). Já o método **findById** retorna um **Optional**. O classe Optional existe desde o Java 8 e representa um container que pode ou não conter um valor não nulo (diferente de null). Essa classe é bem interessante para evitar exceções do tipo NullPointerException, e também permite fazer o uso dos métodos da classe **Optional** que simulam a programação funcional. Com o retorno do método do método findAll da interface JpaRepository podemos fazer o uso do método **map** caso o valor retornado seja diferente de nulo. O método map transforma (mapeia) o registro retornado em um tipo ResponseEntity.

Existe também a diferença no tipo do retorno dos métodos no controller. Enquanto to **findAll** retorna uma lista diretamente, o método **findById** retorna um **ResponseEntity** para indicar sucesso ou não.

Criando um novo contato (POST /contacts)

O código para criar um novo contato está listado abaixo:

```
@PostMapping
public Contact create(@RequestBody Contact contact){
    return repository.save(contact);
}
```



O método **create** também é bem direto ao ponto: apenas chama o método **save** da interface JpaRepository. Após criar o registro na tabela, retorna o contato com o atributo id populado e o registro é retornado no corpo de resposta.

A anotação **@RequestBody** indica que o parâmetro contact será vinculado do corpo da requisição. Isso significa que o método espera o seguinte conteúdo do corpo da requisição (em formato JSON):

```
{
```

```
"name": "Java",  
"email": "java@email.com",  
"phone": "(111) 111-1111"  
}
```

Com o uso dessa anotação, o Spring é inteligente e consegue ler e transformar o conteúdo em uma instância da classe Contact.

Atualizando um contato (PUT /contacts)

O código para atualizar um contato existente está listado abaixo:

```
@PutMapping(value="/{id}")  
public ResponseEntity update(@PathVariable("id") long id,  
                             @RequestBody Contact contact) {  
    return repository.findById(id)  
        .map(record -> {  
            record.setName(contact.getName());  
            record.setEmail(contact.getEmail());  
            record.setPhone(contact.getPhone());  
            Contact updated = repository.save(record);  
            return ResponseEntity.ok().body(updated);  
        }).orElse(ResponseEntity.notFound().build());  
}
```

 Copy

Para atualizar um registro, é necessário informar seu ID no caminho da URL (similar ao processo de obter um registro específico). Caso deseje usar um nome de variável diferente do que foi utilizado também pode utilizar o seguinte código `@PathVariable("recordID") long id`, desde que `otherID` também seja o nome em `@PutMapping(value="/{otherID}")`. Além do ID, também é necessário passar o objeto com os dados atualizados.

O próximo passo é encontrar o registro a ser atualiza que está na base de dados. Se o registro for encontrado, pode-se fazer as atualizações necessárias e assim chamar o método `save` e retornar os dados do registro atualizados. Note que o método `save` também foi utilizado na criação do registro. Caso o objeto tenha sido recuperado da base tenha um ID, será realizado um `update` e não um `insert` na tabela.

Caso o registro não seja encontrado, é retornado um erro HTTP 404.

Um ponto importante para esse método (e também para o processo de criação de registros) é que a própria classe de entidade JPA está sendo utilizada como objeto do tipo parâmetro. Não é uma prática recomendada utilizar a entidade JPA como um objeto de transferência (ou DTO: Data Transfer Object). É sempre bom evitar expor o modelo da base diretamente para o cliente da API. Para esse caso, pode-se criar uma classe com todos os atributos da classe Contact, exceto o atributo `id` (ou uma classe com atributos que facilite a manipulação dos dados por um front-end por exemplo).

Pode-se ainda desenvolver uma série de validações para melhorar esse código. Por exemplo, pode-se adicionar uma validação para garantir que o id do registro passado como parâmetro é o mesmo id passado na URL. Pode-se também utilizar a API Java Beans para aplicar validações de tamanho de campo, obrigatoriedade de atributos, etc. É aqui que entrar toda a lógica de negócio necessária para a aplicação funcionar da forma que se é esperado.

Removendo um contato pelo ID (DELETE /contacts/{id})

O código para remover um contato pelo ID está listado abaixo:

```
@DeleteMapping(path = "{/id}")
public ResponseEntity <?> delete(@PathVariable long id) {
    return repository.findById(id)
        .map(record -> {
            repository.deleteById(id);
            return ResponseEntity.ok().build();
        }).orElse(ResponseEntity.notFound().build());
}
```

 Copy

Para remover um contato pelo ID, utiliza-se o **id** que foi passado como parâmetro para procurar se o registro existe na base. Caso exista, utiliza-se o método **deleteById** da interface **JpaRepository** e retorna o status HTTP 200 para indicar sucesso. Caso o registro não exista, retorna um erro HTTP 404.

E a API RESTful CRUD está pronta! Um ponto importante é a anotação que foi utilizada em cada método que foi desenvolvido:

- Listar todos os contatos - **@GetMapping("/contacts")**
- Obter um contato específico pelo ID - **@GetMapping("/contacts/{id}")**
- Remover um contato pelo ID - **@DeleteMapping("/contacts/{id}")**
- Criar um novo contato - **@PostMapping("/contacts")**
- Atualizar detalhes de um contato - **@PutMapping("/contacts/{id}")**

Apesar da URL ser a mesma (/contacts), o que garante os métodos HTTP diferentes são cada uma das anotações usadas em cada método.

Inicializando a base de dados

Um último passo que é opcional é inserir alguns registros na tabela para teste na classe principal da aplicação (o nome da classe pode variar dependendo do nome que configurou para o projeto no start.spring.io):

```
@SpringBootApplication
public class SpringCloudMysqlApplication {

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




```
@Bean
CommandLineRunner init(ContactRepository repository) {
    return args -> {
        repository.deleteAll();
        LongStream.range(1, 11)
            .mapToObj(i -> {
                Contact c = new Contact();
                c.setName("Contact " + i);
                c.setEmail("contact" + i + "@email.com");
                c.setPhone("(111) 111-1111");
                return c;
            })
            .map(v -> repository.save(v))
            .forEach(System.out::println);
    };
}
```

Como último passo, pode-se criar um método e adicionar a anotação `@Bean` que indicar que o container do Spring deve gerenciar esse método. O repositório é passado como parâmetro para esse método (e o Spring irá cuidar da injeção de dependência) para que possa ser acessado de dentro do método. O método retorna o tipo `CommandLineRunner` (que é uma interface) que indica que o mesmo deverá ser executado quando a aplicação foi executada dentro do contexto do Spring. Esse interface implement o método `run`. No caso do exemplo, está utilizando uma função lambda que retorna os argumentos que serão passados para o método `run`.

Primeiro, remove-se todos os registros da base, e logo após, cria-se um stream de 1 até 11 (não inclusivo), e para cada iteração, insere um registro de teste na base e imprime (`toString` da classe `Contact`). No total serão inseridos 10 registros na base.

O código fonte está pronto. O próximo passo é configurar o projeto para acessar o banco de dados.

Configurando o acesso ao MySQL

Para que o código consiga se conectar com sucesso na base de dados MySQL, é necessário informar os detalhes de conexão. Esses detalhes são declarados no arquivo `src/main/resources/application.properties`:

```
## Spring DATASOURCE (DataSourceAutoConfiguration & DataSourceProperties)
spring.datasource.url=jdbc:mysql://localhost:3306/mydatabase?useSSL=false
spring.datasource.username=root
spring.datasource.password=root

# Dialeto SQL melhorar o SQL gerado pelo Hibernate
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL5InnoDBDialect
```

Caso não deseje executar o script para criar a tabela manualmente, também pode-se adicionar essa configuração para que o Hibernate crie a tabela automaticamente:

```
# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto=update
```

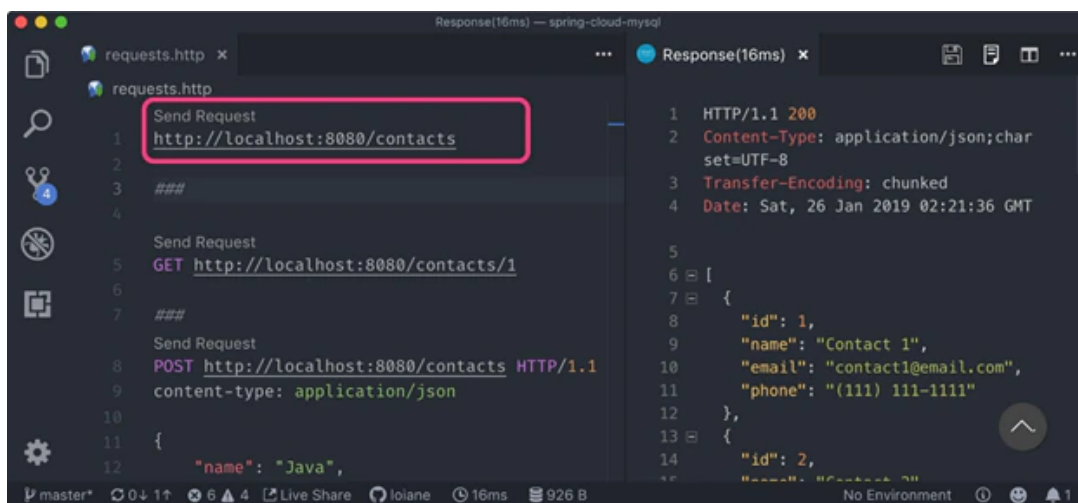
[Copy](#)

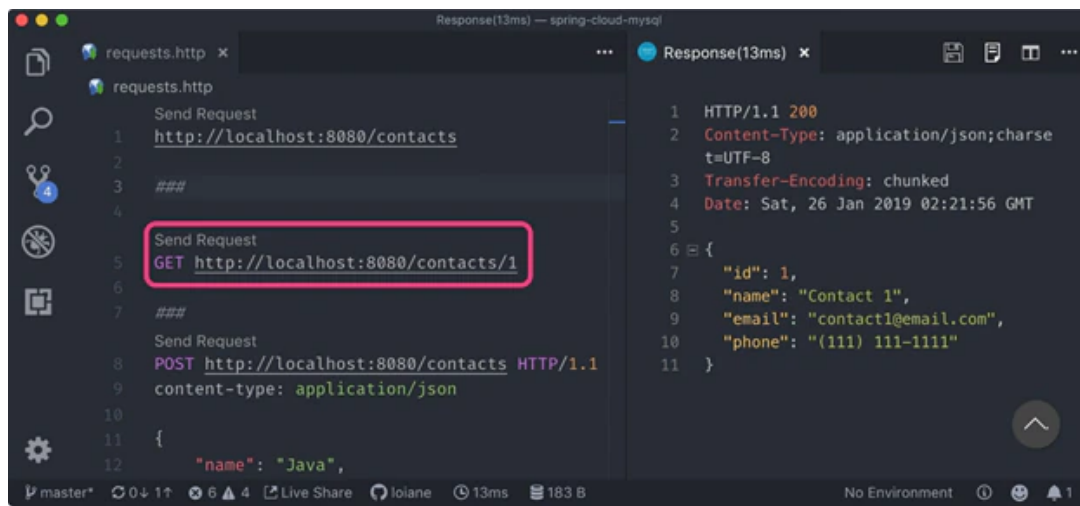
Apesar da configuração de geração de tabelas simplificar o processo, não é uma prática recomendada para bases de produção. Para produção, pode ser que queira criar um ID funcionar para poder executar apenas scripts DLM (Data Manipulation Language) como select, insert, update e delete e não poder executar nenhuma operação DDL (Data Definition Language) como create table, alter, drop table, etc.

Por enquanto, os dados de conexão estão declarados diretamente no arquivo (url da base, usuário e senha). Isso também não é uma boa prática. No desenvolvimento moderno, essas informações ficarão no container que irá executar a aplicação e irá passar essas informações dinamicamente. Iremos aprender como fazer isso em um artigo futuro.

Testando a API manualmente

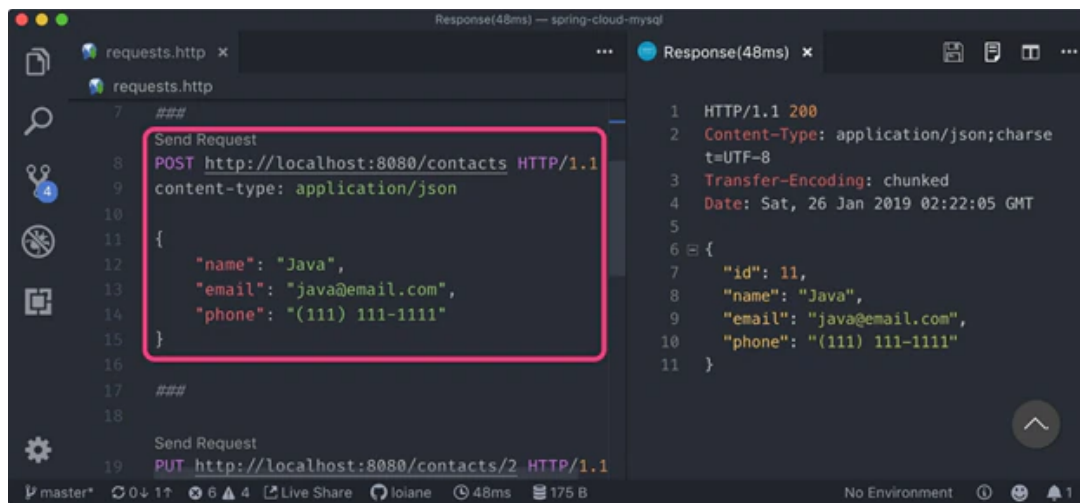
Pode-se ver com a API se comporta na prática utilizando o Postman (<https://www.getpostman.com/>). As imagens a seguir utilizam o plugin REST Client (<https://marketplace.visualstudio.com/items?itemName=humao.rest-client>).





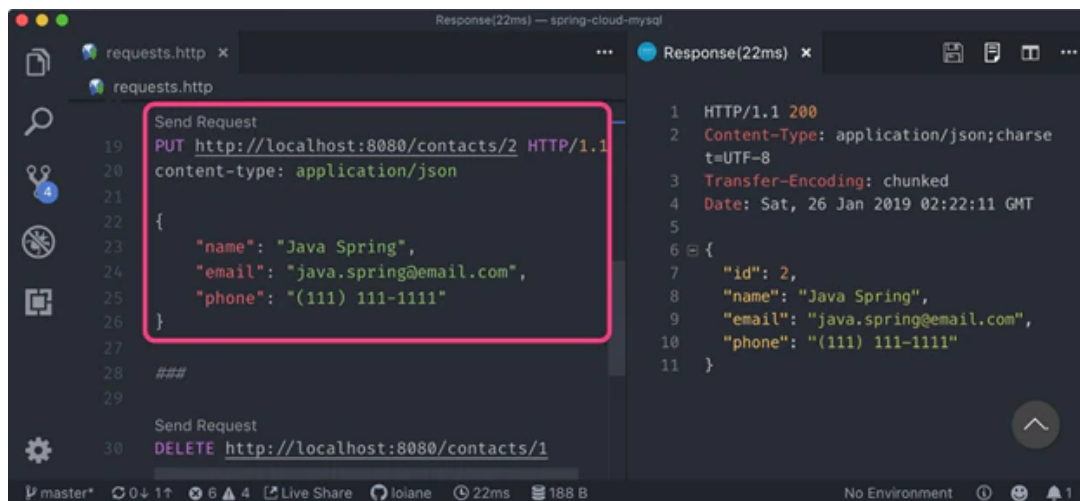
The screenshot shows the Postman interface with a GET request to `http://localhost:8080/contacts/1` highlighted in a red box. The response is a JSON object with the following details:

```
1 HTTP/1.1 200
2 Content-Type: application/json;charse
  t=UTF-8
3 Transfer-Encoding: chunked
4 Date: Sat, 26 Jan 2019 02:21:56 GMT
5
6 {
7   "id": 1,
8   "name": "Contact 1",
9   "email": "contact1@email.com",
10  "phone": "(111) 111-1111"
11 }
```



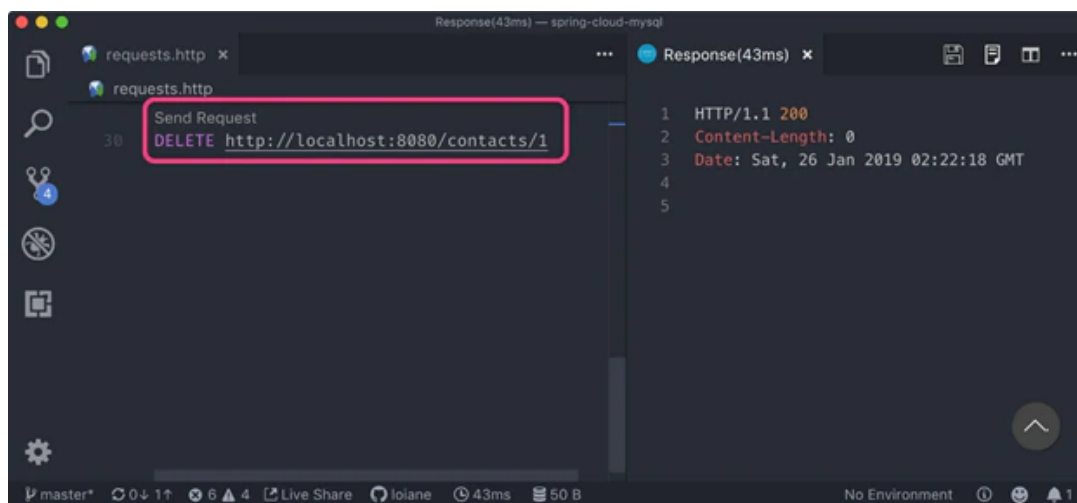
The screenshot shows the Postman interface with a POST request to `http://localhost:8080/contacts` highlighted in a red box. The request body is a JSON object. The response is a JSON object with the following details:

```
1 HTTP/1.1 200
2 Content-Type: application/json;charse
  t=UTF-8
3 Transfer-Encoding: chunked
4 Date: Sat, 26 Jan 2019 02:22:05 GMT
5
6 {
7   "id": 11,
8   "name": "Java",
9   "email": "java@email.com",
10  "phone": "(111) 111-1111"
11 }
```



The screenshot shows the Postman interface with a PUT request to `http://localhost:8080/contacts/2` highlighted in a red box. The request body is a JSON object. The response is a JSON object with the following details:

```
1 HTTP/1.1 200
2 Content-Type: application/json;charse
  t=UTF-8
3 Transfer-Encoding: chunked
4 Date: Sat, 26 Jan 2019 02:22:11 GMT
5
6 {
7   "id": 2,
8   "name": "Java Spring",
9   "email": "java.spring@email.com",
10  "phone": "(111) 111-1111"
11 }
```



Conclusão

Esse artigo mostrou como desenvolver uma API CRUD conectando num banco de dados local. Os próximos passos são criar um banco de dados em um ambiente cloud além de fazer o deploy da aplicação na nuvem também para que outros usuários também possam utilizar essa API. Esses passos serão mostrados em artigos futuros.

Loiane Groner trabalha com desenvolvimento de software há mais de 10 anos e atualmente trabalha como analista de negócios e desenvolvedor sênior de Java / HTML5 / JavaScript em uma instituição financeira americana. Loiane é autora de livros para Packt Publishing, e teve a oportunidade de falar em algumas conferências sobre desenvolvimento Java, JavaScript, Sencha, Ionic, Angular e moderno. Em seu tempo livre, ela publica artigos em seu blog <https://loiane.com> e tutoriais em vídeo em <https://loiane.training>.

Este artigo foi revisado pela equipe de produtos Oracle e está em conformidade com as normas e práticas para o uso de produtos Oracle

Recursos para

Carreiras
Desenvolvedores
Investidores
Parceiros
Startups
Estudantes e Educadores

Por que Oracle

Relatórios de Analistas
Gartner MQ para ERP Cloud
Economia da Nuvem
Responsabilidade Corporativa
Diversidade e Inclusão
Práticas de Segurança

Aprenda

O que é computação em nuvem?
O que é CRM?
O que é Docker?
O que é Kubernetes?
O que é Python?
O que é SaaS?

Novidades

Experimente o Oracle Cloud - Modo Gratuito (Free Tier)
Oracle Arm Processors
Oracle e Premier League
Oracle e Red Bull Racing Honda
Plataforma de Experiência do Funcionário

Oracle Support
Rewards

Entre em Contato

Vendas: 0800-891-4433

Como podemos ajudar?

Inscreva-se para receber e-mails

Eventos

Notícias

Blogs

	©	Mapa	Termos de	Preferências	Opções	Carreiras
País/Região	2021	do Site	Uso e	de Cookies	de	
	Oracle		Privacidade		Anúncios	