



Digital Business Enablement

Prof. Gilberto Alexandre das Neves
profgilberto.neves@fiap.com.br

Web Services Restful

O que é Rest?

A sigla **REST** (Representational State Transfer), em português, significa “Transferência de Estado Representacional”. Concebido como uma abstração da arquitetura da web, trata-se de um conjunto de princípios e definições necessários para a criação de um projeto com interfaces bem definidas.

A utilização da arquitetura **REST**, portanto, permite a comunicação entre aplicações. Ao abrir o navegador, ele estabelece uma conexão **TCP/IP** com o servidor de destino e envia uma requisição **GET HTTP**, com o endereço buscado.

O servidor, então, interpreta a requisição, retornando com uma resposta **HTTP** ao navegador. Essa resposta pode ser completa, com representações em formato **HTML**, ou apresentar erro, afirmando que o recurso solicitado não foi encontrado.

Esse processo é repetido diversas vezes em um período de navegação. Cada nova **URL** aberta ou formulário submetido refaz as etapas que descrevemos. Dessa forma, esses elementos permitem a criação de aplicações web, desenhando a forma como navegamos na internet.

O que é Rest?

Os *Web Services* que adotam **REST** são mais leves e perfeitos na busca da *metodologia ágil*.

Outro diferencial é a flexibilidade, sendo possível escolher o formato que melhor se encaixa para as mensagens do sistema. Os mais utilizados, além do texto puro, são **Json** e **XML**, dependendo da necessidade de cada momento.

REST e RESTful são a mesma coisa?

Agora que você já conheceu um pouco mais sobre o **REST**, está na hora de entender o que é **RESTful**. Embora possam gerar certa confusão, os dois termos revelam o mesmo propósito. Sendo assim, podemos dizer que sistemas que utilizam determinações **REST** são chamados de **RESTful**.

- **REST**: representa um apanhado de princípios de arquitetura,
- **RESTful**: representa a condição de um sistema específico em aplicar os conceitos de *REST*.

Fonte: <https://www.totvs.com/blog/developers/rest/>

Em uma aplicação **REST**, os métodos (**verbos**) mais utilizados são:

- O método **GET** é o método mais comum, geralmente é usado para solicitar que um servidor envie um recurso;
- O método **POST** foi projetado para enviar dados de entrada para o servidor. Na prática, é frequentemente usado para suportar formulários **HTML**;
- O método **PUT** edita e atualiza documentos em um servidor;
- O método **DELETE** que como o próprio nome já diz, deleta certo dado ou coleção do servidor.

Códigos de Respostas (Status code)

Cada resposta que a aplicação **REST** retorna, é enviado um código definindo o status da requisição. Por exemplo:

- **200 (OK)**, requisição atendida com sucesso;
- **201 (CREATED)**, objeto ou recurso criado com sucesso;
- **204 (NO CONTENT)**, objeto ou recurso deletado com sucesso;
- **400 (BAD REQUEST)**, ocorreu algum erro na requisição (podem existir inúmeras causas);
- **404 (NOT FOUND)**, rota ou coleção não encontrada;
- **500 (INTERNAL SERVER ERROR)**, ocorreu algum erro no servidor.

Fonte: <https://www.alura.com.br/artigos/rest-conceito-e-fundamentos>

Prosseguindo com o projeto...

Com a aplicação criada, vamos desenvolver as funcionalidades do nosso projeto (API Rest da aplicação de Reformas).

Vamos implementar o envio de dados para a API com o cadastro de prestadores. Observe abaixo a descrição dessa funcionalidade:

Cadastro de Prestadores:

O sistema deve possuir uma funcionalidade de cadastro de prestadores, na qual as seguintes informações deverão ser preenchidas:

Nome

E-mail (deve ser único)

Telefone

CNPJ (deve ser único)

Especialidade (ENCANADOR, PEDREIRO, ELETRICISTA, PINTOR ou DECORADOR)

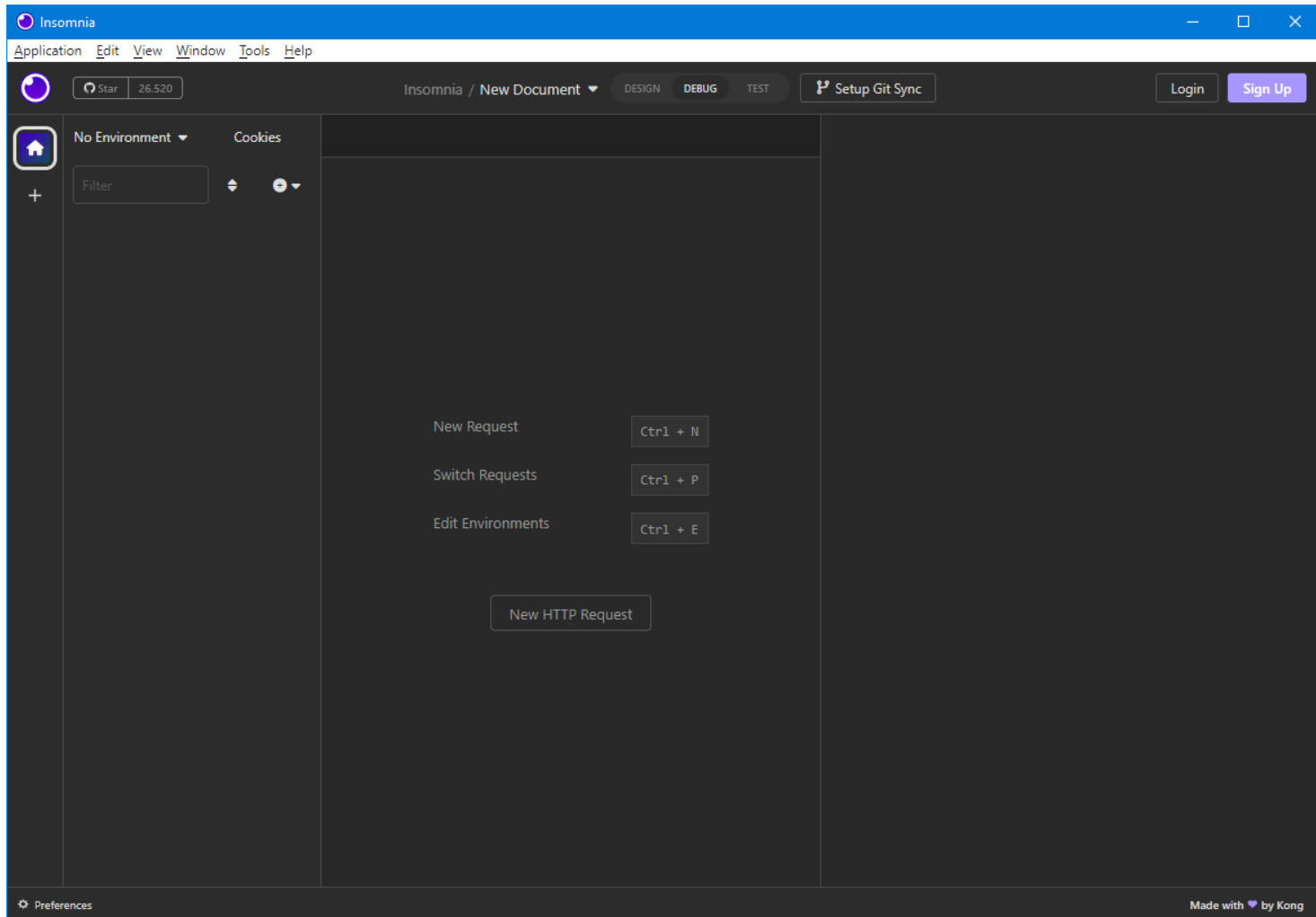
Endereço completo (Logradouro, número, complemento, bairro, cidade, UF e CEP)

Todas as informações são de preenchimento **obrigatório**, exceto número e o complemento de endereço.

Insomnia

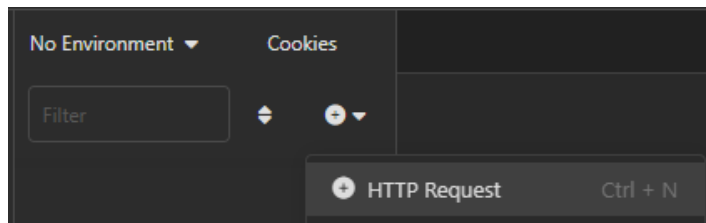
ferramenta de teste de API

Vamos utilizar a ferramenta para teste de API **Insomnia**, basta fazer download e instalar pelo link: <https://insomnia.rest/download>

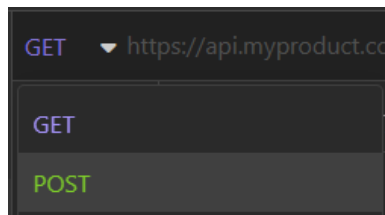


Vamos configurar:

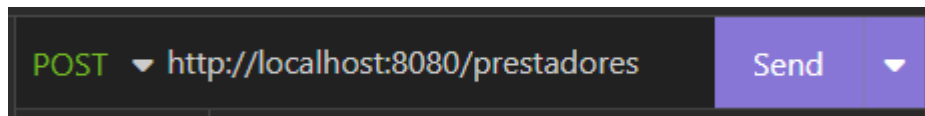
- Do lado direito clique no sinal de  e escolha **HTTP Request** (atalho **Ctrl+N**).



- Dê um duplo clique em *New Request* e renomeie para **Cadastro de Prestador**.
- Por ser um cadastro, não é um método *get*. Na parte central mude para **POST**.

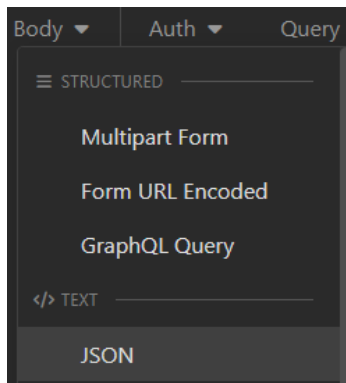


- Digite a **URL** <http://localhost:8080/prestadores> que iremos mapear em nosso projeto para enviar essa requisição **POST**.



Vamos configurar:

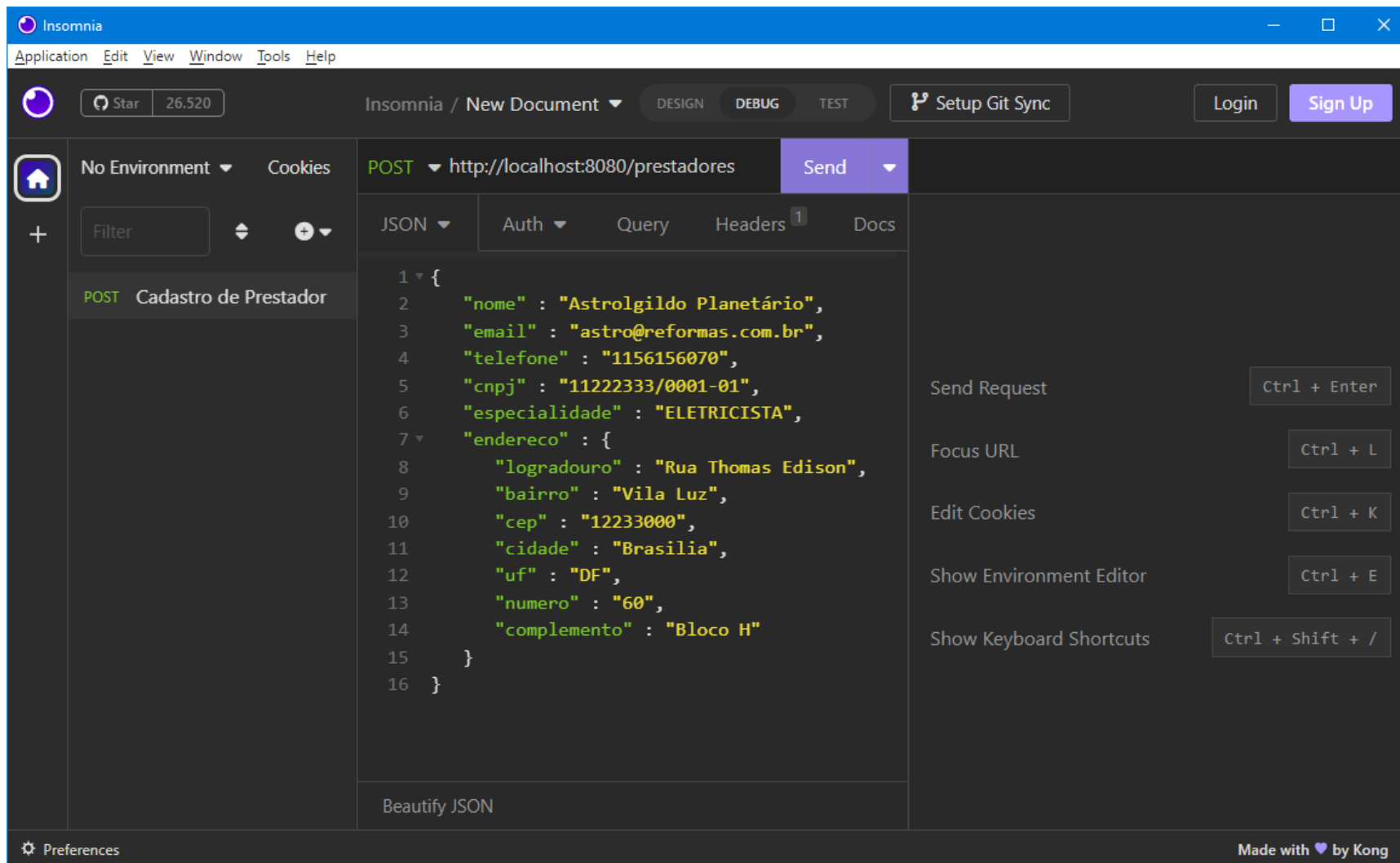
- Na parte central, em **Body** escolha **JSON**.



- E digite as informações:

```
{
  "nome" : "Astrolgildo Planetário",
  "email" : "astro@reformas.com.br",
  "telefone" : "1156156070",
  "cnpj" : "11222333/0001-01",
  "especialidade" : "ELETRICISTA",
  "endereco" : {
    "logradouro" : "Rua Thomas Edison",
    "bairro" : "Vila Luz",
    "cep" : "12233000",
    "cidade" : "Brasília",
    "uf" : "DF",
    "numero" : "60",
    "complemento" : "Bloco H"
  }
}
```

Confira como deve ficar o Insomnia:



JSON (*JavaScript Object Notation*) é um formato utilizado para representação de informações, assim como **XML** e **CSV**.

Uma API precisa receber e devolver informações em algum formato, que representa os recursos gerenciados por ela. O **JSON** é um desses formatos possíveis, tendo se popularizado devido a sua leveza, simplicidade, facilidade de leitura por pessoas e máquinas, bem como seu suporte pelas diversas linguagens de programação.

Um exemplo de representação de uma informação no formato **XML** seria:

```
<produto>
  <nome>Mochila</nome>
  <preco>89.90</preco>
  <descricao>Mochila para notebooks de até 17 polegadas</descricao>
</produto>
```

Já a mesma informação poderia ser representada no formato **JSON** da seguinte maneira:

```
{
  "nome" : "Mochila",
  "preco" : 89.90,
  "descricao" : "Mochila para notebooks de até 17 polegadas"
}
```

PrestadorController

Vamos criar a classe **PrestadorController** (no pacote **controller**).

Para comunicarmos o **Spring MVC** que é uma classe controller, acima dela incluiremos a anotação **@RestController** (pois estamos trabalhando com uma API Rest).

Outra anotação para incluir é a **@RequestMapping**. Isso informa qual a URL que esse controller vai responder, que será **/prestadores**. Assim, ao chegar uma requisição para **localhost:8080/prestadores** vai cair neste controller.

```
PrestadorController.java ×
1 package br.com.reformas.api.controller;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 @RequestMapping("/prestadores")
8 public class PrestadorController {
9
10 }
```

Vamos declarar um método chamado **cadastar()** para a funcionalidade de cadastro de prestadores, com o retorno **void**.

Acima do método, é necessário especificarmos o *verbo* do protocolo **HTTP** que ele vai lidar. No caso, estamos enviando as requisições via verbo **post**, por isso, incluiremos a anotação **@PostMapping**.

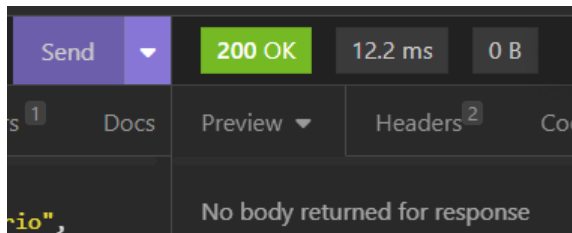
```
PrestadorController.java ×
1 package br.com.reformas.api.controller;
2
3 import org.springframework.web.bind.annotation.PostMapping;
4 import org.springframework.web.bind.annotation.RequestMapping;
5 import org.springframework.web.bind.annotation.RestController;
6
7 @RestController
8 @RequestMapping("/prestadores")
9 public class PrestadorController {
10
11     @PostMapping
12     public void cadastrar() {
13
14     }
15 }
```


Vamos testar:

- Adicione ao método **cadastrar()** um *parâmetro* do tipo **String**.
- Devemos adicionar a anotação **@RequestBody** neste parâmetro para indicar ao *Spring* que deve pegar as informações do corpo da requisição.
- E finalmente, vamos exibir no *Console* a informação recebida.

```
PrestadorController.java X
1 package br.com.reformas.api.controller;
2
3 import org.springframework.web.bind.annotation.PostMapping;
4 import org.springframework.web.bind.annotation.RequestBody;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 @RequestMapping("/prestadores")
10 public class PrestadorController {
11
12     @PostMapping
13     public void cadastrar(@RequestBody String json) {
14         System.out.println(json);
15     }
16 }
```

Clicando em **Send** no *Insomnia* temos como resposta o código de *status* **200 OK**.



E se conferirmos o *Console* do *Eclipse* vemos nosso **JSON**.

```
2023-02-10T14:20:45.301-03:00 INFO 8416 --- [nio-8080-exec
2023-02-10T14:20:45.302-03:00 INFO 8416 --- [nio-8080-exec
{
  "nome" : "Astrologildo Planetário",
  "email" : "astro@reformas.com.br",
  "telefone" : "1156156070",
  "cnpj" : "11222333/0001-01",
  "especialidade" : "ELETRICISTA",
  "endereco" : {
    "logradouro" : "Rua Thomas Edison",
    "bairro" : "Vila Luz",
    "cep" : "12233000",
    "cidade" : "Brasilia",
    "uf" : "DF",
    "numero" : "60",
    "complemento" : "Bloco H"
  }
}
```

DTO com Java Record



Conseguimos receber as informações enviadas pelo *Insomnia* no **controller**, mas precisamos encontrar uma maneira de não recebermos esses dados como *String* e sim receber o **JSON** inteiro como *String*.

Uma forma de recebermos cada campo isoladamente, é não usar uma *String* como *parâmetro* do método **cadastar** e sim uma classe. Nela, declaramos os **atributos** com os mesmos nomes que constam no **JSON**.

Por isso, no método **cadastar** vamos alterar o *parâmetro* de *String json* para uma classe que criaremos para representar os dados enviados pela requisição.

Chamaremos essa classe de **DadosCadastroPrestador**, nomearemos o parâmetro de **dados** e no *system.out* ao invés de **json** será **dados**.

```
PrestadorController.java ×
1 package br.com.reformas.api.controller;
2
3 import org.springframework.web.bind.annotation.PostMapping;
4 import org.springframework.web.bind.annotation.RequestBody;
5 import org.springframework.web.bind.annotation.RequestMapping;
6 import org.springframework.web.bind.annotation.RestController;
7
8 @RestController
9 @RequestMapping("/prestadores")
10 public class PrestadorController {
11
12     @PostMapping
13     public void cadastrar(@RequestBody DadosCadastroPrestador dados) {
14         System.out.println(dados);
15     }
16 }
```

Note que **DadosCadastroPrestador** está com erro de compilação, porque ainda não criamos essa classe

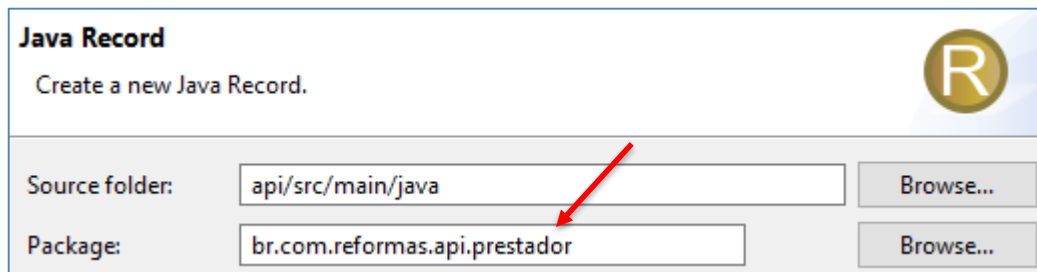
DTO com Java Record

Usaremos o recurso de **Record** (ao invés de uma classe tradicional). Este recurso funciona como se fosse uma classe imutável, para deixarmos o código simples.

Posicione o mouse sobre **DadosCadastroPrestador** (sem clicar) e escolha a opção que aparece no menu *pop-up*: **Create record 'DadosCadastroPrestador'** (segunda opção da lista).



Na janela, vamos alterar o pacote de *controller* para **prestador**.

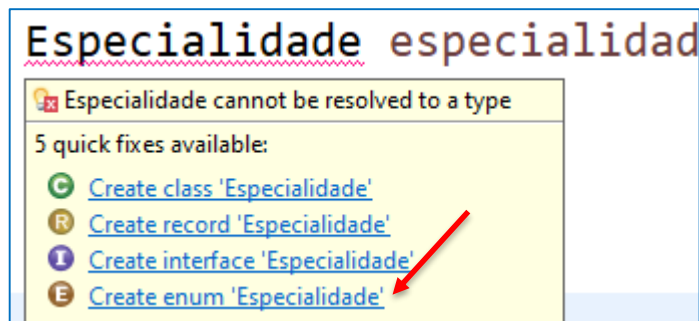


- No parênteses do método **record**, precisamos inserir os campos enviados pela requisição.
- Como o campo "**especialidade**" é fixo, não usaremos *String* e sim um **Enum**.
- No campo "**endereco**" temos contido diversos campos nele, por isso, criaremos outro **record** para representar os dados do endereço.

```
*DadosCadastroPrestador.java X
1 package br.com.reformas.api.prestador;
2
3 public record DadosCadastroPrestador(
4
5     String nome,
6     String email,
7     String telefone,
8     String cnpj,
9     Especialidade especialidade,
10    DadosEndereco endereco
11 ) {
12
13 }
```

Enum Especialidade

Posicione o mouse sobre **Especialidade** (sem clicar) e escolha a opção que aparece no menu *pop-up*: **Create enum 'Especialidade'** (quarta opção da lista).

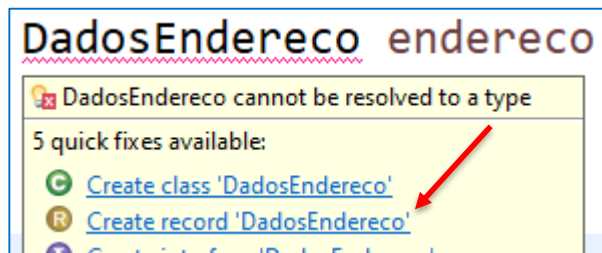


Na janela apenas clique em **Finish** (deixe no pacote *prestador* mesmo). E digite as especialidades neste **enum** (todas em maiúscula).

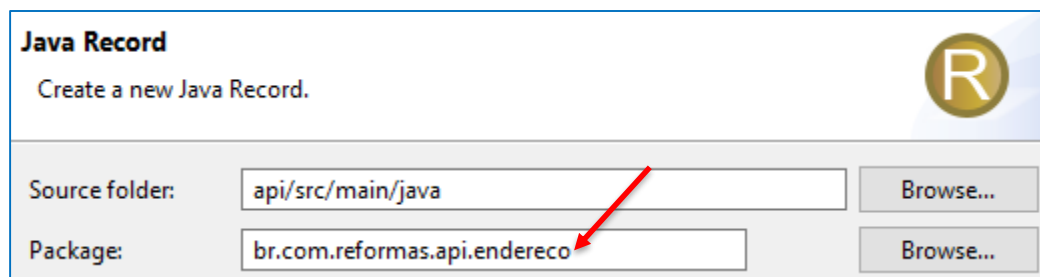
```
1 package br.com.reformas.api.prestador;
2
3 public enum Especialidade {
4
5     ENCANADOR,
6     PEDREIRO,
7     ELETRICISTA,
8     PINTOR,
9     DECORADOR;
10 }
```


Record DadosEndereco

Posicione o mouse sobre **DadosEndereco** (sem clicar) e escolha a opção que aparece no menu *pop-up*: **Create record 'DadosEndereco'** (segunda opção da lista).



Na janela, vamos alterar o pacote de *prestador* para **endereco**.

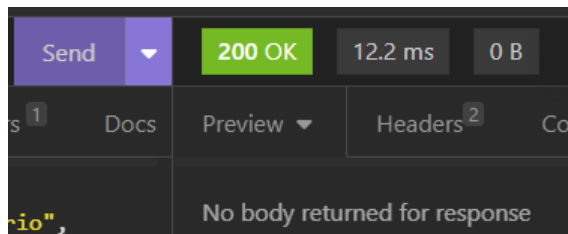


Record DadosEndereco

- No parênteses do método **record**, precisamos inserir os campos que compõe o endereço.

```
DadosEndereco.java X
1 package br.com.reformas.api.endereco;
2
3 public record DadosEndereco(
4     String logradouro,
5     String bairro,
6     String cep,
7     String cidade,
8     String uf,
9     String numero,
10    String complemento
11 ) {
12
13 }
```

Vamos testar novamente clicando em **Send** no *Insomnia* temos como resposta o código de *status* **200 OK**.



E se conferirmos o *Console* do *Eclipse* vemos nosso **JSON** (agora com cada campo separado com sua respectiva informação).

```
2023-02-10T15:21:11.739-05:00 INFO 5580 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed Initial  
DadosCadastroPrestador[nome=Astrolgildo Planetário, email=astro@reformas.com.br, telefone=1156156070, cnpj=11222333/0001-01,
```

Podemos até mesmo, exibir dados específicos. Basta no **system.out** colocar o objeto **dados** seguido do *método* com o nome da informação que deseja exibir. Faça o teste!

O **Record** é um recurso que permite representar uma classe imutável, contendo apenas atributos, construtor e métodos de leitura, de uma maneira muito simples e enxuta.

Esse tipo de classe se encaixa perfeitamente para representar classes **DTO** (*Data Transfer Object*), já que seu objetivo é apenas representar dados que serão recebidos ou devolvidos pela API, sem nenhum tipo de comportamento.

Para se criar uma classe **DTO** imutável, sem a utilização do **Record**, era necessário escrever muito código.

Praticando...

Vamos fazer o mesmo para os Clientes!

Vamos implementar o envio de dados para a API com o cadastro de clientes. Observe abaixo a descrição dessa funcionalidade:

Cadastro de Clientes:

O sistema deve possuir uma funcionalidade de cadastro de clientes, na qual as seguintes informações deverão ser preenchidas:

Nome

E-mail (deve ser único)

Telefone

CPF (deve ser único)

Endereço completo (Logradouro, número, complemento, bairro, cidade, UF e CEP)

Todas as informações são de preenchimento **obrigatório**, exceto número e o complemento de endereço.

1. Crie a classe **ClienteController** (no pacote **controller**) e construa nele o método **cadastrar()**. Mapeie a URL “/clientes” para receber as requisições.
2. Crie o Record **DadosCadastroCliente** (no pacote **cliente**).
3. Crie no *Insomnia* o **JSON** com as informações pertinentes para um Cliente e realize o teste para verificar se sua API está recebendo estas informações.

1. Em aplicações REST, qual a melhor definição para o método POST?
 - a) O método **POST** é o método mais comum, geralmente é usado para solicitar que um servidor envie um recurso.
 - b) O método **POST** foi projetado para enviar dados de entrada para o servidor. Na prática, é frequentemente usado para suportar formulários **HTML**.
 - c) O método **POST** edita e atualiza documentos em um servidor.
 - d) O método **POST** que como o próprio nome já diz, apaga certo dado ou coleção do servidor.

2. Para indicar ao *Spring* que deve pegar as informações do corpo da requisição vinda de uma ferramenta para teste de API , por exemplo. Devemos usar a anotação?
 - a) @RequestBody no JSON criado no Insomnia.
 - b) @RequestBody antes do nome da classe.
 - c) @RequestBody antes do nome do método.
 - d) @RequestBody antes do parâmetro do método criado.





Java com Spring, Hibernate e Eclipse. Anil Hemrajani. Pearson, 2013.

Spring MVC: Domine o principal framework web Java. Alberto Souza. Série Caelum.

Site: alura.com.br

Até breve!

1. Em aplicações REST, qual a melhor definição para o método POST?
 - a) O método **POST** é o método mais comum, geralmente é usado para solicitar que um servidor envie um recurso.
 -  b) O método **POST** foi projetado para enviar dados de entrada para o servidor. Na prática, é frequentemente usado para suportar formulários **HTML**.
 - c) O método **POST** edita e atualiza documentos em um servidor.
 - d) O método **POST** que como o próprio nome já diz, apaga certo dado ou coleção do servidor.

2. Para indicar ao *Spring* que deve pegar as informações do corpo da requisição vinda de uma ferramenta para teste de API , por exemplo. Devemos usar a anotação?
 - a) @RequestBody no JSON criado no Insomnia.
 - b) @RequestBody antes do nome da classe.
 - c) @RequestBody antes do nome do método.
 -  d) @RequestBody antes do parâmetro do método criado.