Curso: Análise e Desenvolvimento de Sistemas – ADS

Ano: 2023/1

Orientação Técnica (OT) – 19

Cadastro REST - Programação

Introdução

Neste momento, temos o formulário pronto para fazer seu papel: permitir o preenchimento de todos os dados referentes a um novo produto. Agora nos falta apenas fazer a programação que realizará a inserção desse produto no BD, e é justamente esse o papel da OT que está lendo. Vamos começar então!

Enviando os dados do formulário ao servidor

Novamente, começaremos do lado frontend. Abra o arquivo **product/index.html** e encontre o botão “Cadastrar”. Nele, adicione o seguinte código:

```
<button type="button" onclick="COLDIGO.produto.cadastrar()">Cadastrar</button>
```

Assim, ao clicar neste botão, chamaremos a função JS responsável pelo cadastro de um novo produto. Como ainda não temos essa função, vamos até o arquivo **product.js** para criá-la, conforme a imagem abaixo:

```
    COLDIGO.produto.carregarMarcas();

    //Cadastra no BD o produto informado
    COLDIGO.produto.cadastrar = function(){

        var produto = new Object();
        produto.categoria = document.frmAddProduto.categoria.value;
        produto.marcaId = document.frmAddProduto.marcaId.value;
        produto.modelo = document.frmAddProduto.modelo.value;
        produto.capacidade = document.frmAddProduto.capacidade.value;
        produto.valor = document.frmAddProduto.valor.value;

        if((produto.categoria=="")||(produto.marcaId=="")||(produto.modelo=="")
                ||(produto.capacidade=="")||(produto.valor=="")){
            COLDIGO.exibirAviso("Preencha todos os campos!");

        } else {

        }
    }

});
```

Veja que não há nada de muito especial: criamos a função **dentro do objeto Coldigo.produto** e, dentro dela, criamos um **objeto produto**, recebemos nele todos os **valores do formulário** através de atributos criados aqui mesmo, e fazemos uma **validação básica** de campos vazios. Aliás, se quiser, pode melhorar essa validação, contanto que deixe íntegra a situação de que "se tudo estiver certo, o bloco "else" deve ser executado", *talquei*?
Agora que temos o código que nos permite **saber se os dados foram preenchidos corretamente**, podemos fazer a parte que irá **enviar esses dados ao servidor**. Para isso, implemente o seguinte código **dentro do "else"**, como indicado.

```
} else {

    $.ajax({
        type: "POST",
        url: "/ProjetoTrilhaWeb/rest/produto/inserir",
        data:JSON.stringify(produto),
        success: function (msg) {
            COLDIGO.exibirAviso(msg);
            $("#addProduto").trigger("reset");
        },
        error: function (info) {
            COLDIGO.exibirAviso("Erro ao cadastrar um novo produto: "+ info.status + " - " + info.statusText);
        }
    });

}
```

Dessa vez, fizemos de uma vez só **todo o código Ajax** necessário para essa etapa. Detalhando, executamos a **função ajax do jQuery**, passando como parâmetros:

- **POST** como método de requisição;
- a **URL** desejada;
- transformamos o **objeto produto** numa **String em formato JSON** e colocamos no **parâmetro data**;
- na função **success**, recebemos em **msg** o valor que o servidor nos retornar, exibimos esse valor em nossa **modal de aviso** através do chamado da função **COLDIGO.exibirAviso**, e com o comando **jQuery trigger** executamos o **reset** no formulário (é dele o **ID addProduto**) para limparmos seus dados;
- na função **error**, exibimos o status e sua descrição na nossa **modal de aviso**.

RESPONDA: Por que o método deve ser POST?

Ainda sobre o **parâmetro do Ajax chamado data**, grande novidade do momento: Este parâmetro tem como função **indicar os dados a serem enviados do lado cliente para o servidor**. Nesse caso, como queremos gravar os dados do formulário, são eles que colocaremos nesse parâmetro (mas em formato Json) para que o servidor os receba e faça a gravação.

RESPONDA: Por que precisamos converter o objeto produto para JSON antes de enviá-lo ao servidor?

Com isso, finalizamos as edições no lado frontend. Quer dizer, quase...

Otimização da programação frontend com indicação estática do caminho PATH

Dê uma olhada nos **atributos url dos dois Ajax** que fizemos até agora. Reparou que ambos começam do mesmo jeito: **/ProjetoTrilhaWeb/rest/** ? Pois é, e como esse é o caminho padrão de nossas REST (e será até o fim do projeto), vamos ter que repetir isso em todos os outros momentos em que usarmos Ajax...

Agora imagine a seguinte situação: você desenvolveu um software inteiro em sua empresa, e chegou a hora de implantá-lo em um servidor para seu cliente. Com essa mudança, a URL será diferente: www.coldigogeladeiras.com.br. Assim, teremos que alterar TODAS as nossas funções Ajax para atender o novo padrão! Fazer isso um por um levaria muito tempo, e ainda poderíamos ter lugares esquecidos, não é? Para evitarmos problemas assim, vamos indicar de maneira **estática** essa parte que é padrão em todas nossas chamadas ao servidor.

Abra o arquivo **admin.js** e adicione nele o código exibido na imagem, entre os destaques em amarelo:

```
$(document).ready(function() {

    //Cria uma constante com o valor da URI raiz do REST
    COLDIGO.PATH = "/ProjetoTrilhaWeb/rest/";

    $("header").load("/ProjetoTrilhaWeb/pages/admin/general/header.html");
    $("footer").load("/ProjetoTrilhaWeb/pages/admin/general/footer.html");
```

No objeto **COLDIGO**, adicionamos um **atributo chamado PATH** (em maiúsculas, por se tratar de um valor **constante**, não variável) com o valor raiz de nossas RESTs.

Agora, em nossas duas chamadas **Ajax**, vamos usar essa constante. Para isso, altere as **url em ambos Ajax** para que fiquem assim:

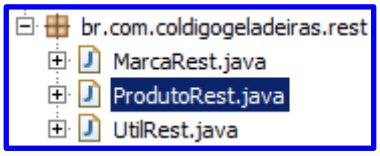
```
url: COLDIGO.PATH + "marca/buscar",
url: COLDIGO.PATH + "produto/inserir",
```

Pronto! Como não finalizamos a parte de inserir produto ainda, **teste atualizando a página e vendo se as marcas aparecem** ainda corretamente no formulário.

ATENÇÃO: se não aparecerem as marcas, corrija o que estiver errado antes de prosseguir!

Codificação backend para registro de produtos

Primeiro, crie o arquivo REST relativo a produtos:



Mais uma vez, criamos um REST, dessa vez para lidar com todas as requisições relativas a produtos. Assim, podemos dizer que, pensando no padrão **MVC**, **ProdutoRest** é uma classe que atua na camada **Controller**.
Vamos agora à sua codificação inicial. Adicione nela as partes destacadas:

```
package br.com.coldigogeladeiras.rest;

import javax.ws.rs.Consumes;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.core.Response;

@Path("produto")
public class ProdutoRest extends UtilRest{

    @POST
    @Path("/inserir")
    @Consumes("application/*")
    public Response inserir(String produtoParam){

    }

}
```

Veja que fazemos os **imports** iniciais, indicamos com uma **anotação o caminho dessa REST** e fazemos com que ela **herde os métodos de UtilRest**, igual fizemos na classe MarcaRest. Porém, o método em si é bem diferente. Vamos explicar em detalhes as 4 linhas dele (até porque explicar pra que serve um "}" a essa altura é sacanagem.

- Anotamos que o método de acesso a ele é POST;
- Anotamos que seu caminho é **produto/inserir**;
- Anotamos que ele **aguarda (ou, na tradução, *consome*) alguma informação do lado cliente**, e que ela deve estar em formato genérico de aplicação;
- Criamos um **método público**, que deve **retornar um objeto de Response**,

com **nome inserir** e que **recebe como parâmetro** uma **String chamada produtoParam**.

PERGUNTAS: Sabe dizer por que ProdutoRest estende UtilRest? E o motivo de usarmos POST neste método, saberia dizer?

Explicando um pouco mais sobre o **Consumes** e o parâmetro **produtoParam**: estamos agora fazendo um processo de cadastro, certo? Para que ele dê certo, o usuário deve **preencher um formulário** com certas **informações**, que devem ser **salvas no BD**. Como o usuário **preenche esses dados no lado cliente**, precisamos fazer com que os dados sejam enviados **pelo** lado cliente **para** o servidor, e que o **servidor saiba como os receber e os tratar** para que sejam armazenados corretamente. Mas em detalhes, como isso acontece?

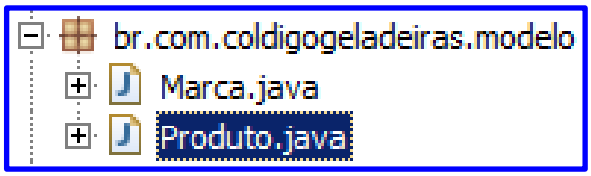
No lado **cliente**, o **envio dos dados** para o servidor é feito no **data do Ajax** criado na OT anterior, onde nesse caso usamos uma função para converter os dados do formulário para JSON (se não lembrar bem, veja no arquivo **product.js** o Ajax mencionado).

Já no lado **servidor**, o método que deve gravar os dados precisa, obviamente, **receber os dados**. Por isso, esse método precisa da **anotação @Consumes**, para indicar que **algo deve ser recebido para ele funcionar corretamente**. Mas **onde** esses dados serão recebidos? **No parâmetro do método!** Assim, nesse caso, o **produtoParam** vai conter exatamente o que nós enviamos lá no **data do Ajax**, um JSON com todos os dados do formulário! Legal, né?

Pois bem, vamos seguir adiante. Dentro do novo método, vamos colocar um **try-catch** básico para o que virá depois:

```
public Response inserir(String produtoParam){

    try{

    }catch(Exception e){
        e.printStackTrace();
        return this.buildErrorResponse(e.getMessage());
    }

}
```

Assim, já cobrimos os possíveis erros desse método. Agora, precisamos criar nossa **classe modelo** para lidarmos com um produto. Conforme imagem abaixo, crie-a!



Abra-a e digite nela o código destacado a seguir:

```
package br.com.coldigogeladeiras.modelo;

import java.io.Serializable;

public class Produto implements Serializable {

    private static final long serialVersionUID = 1L;

    private int id;
    private String categoria;
    private int marcaId;
    private String modelo;
    private int capacidade;
    private float valor;

}
```

Veja que indicamos que a classe **Produto** deve **implementar** a **interface Serializable** e a **importamos** para o arquivo, e dentro da classe criamos os **atributos de um produto e o serialVersionUID**. Só falta uma coisa nessa classe agora: os **getters e setters**! Então, **implemente-os**!

PERGUNTA: para que serve mesmo o serialVersionUID?

Criada a classe modelo de Produto, agora podemos receber os dados enviados lá do frontend e armazená-los em um objeto de Produto. Para isso, vamos adicionar o código abaixo na **ProdutoRest**:

```java
import javax.ws.rs.core.Response;

import com.google.gson.Gson;

import br.com.coldigogeladeiras.modelo.Produto;

@Path("produto")
public class ProdutoRest extends UtilRest{

    @POST
    @Path("/inserir")
    @Consumes("application/*")
    public Response inserir(String produtoParam){

        try{
            Produto produto = new Gson().fromJson(produtoParam, Produto.class);

        }catch(Exception e){
            e.printStackTrace();
            return this.buildErrorResponse(e.getMessage());
        }

    }
```

Adicionamos assim as **importações** das classes necessárias, e usamos o **método fromJson da classe Gson**, através de seus dois parâmetros, **convertemos** os dados enviados do lado cliente em **formato Json** (**produtoParam**) para o modelo da **classe Produto** (**Produto.class**). Entendeu?

OK. Essa explicação do **fromJson** está meio superficial, né? Vamos melhorá-la então... Na verdade, este método **converte** um conteúdo **do formato Json** (primeiro parâmetro) **para uma classe modelo especificada** (segundo parâmetro). Para isso, ele usa um processo de **comparação das chaves no Json e dos atributos da classe**: para cada chave Json, ele procura um atributo na classe que possua exatamente o mesmo nome, e **se o encontrar**, guarda o valor contido na chave Json no atributo homônimo da classe indicada!

Trazendo para nosso caso: no lado cliente, enviamos no **data do Ajax** o **objeto produto**, **transformado em Json** pelo stringify. Veja a imagem abaixo, que mostra este objeto no arquivo **product.js**:

```
var produto = new Object();
produto.categoria = document.frmAddProduto.categoria.value;
produto.marcaId = document.frmAddProduto.marcaId.value;
produto.modelo = document.frmAddProduto.modelo.value;
produto.capacidade = document.frmAddProduto.capacidade.value;
produto.valor = document.frmAddProduto.valor.value;
```

No lado servidor, criamos a classe modelo com os seguintes atributos:

```
private int id;
private String categoria;
private int marcaId;
private String modelo;
private int capacidade;
private float valor;
```

Percebeu alguma semelhança?

Precisamos usar os mesmos nomes no lado cliente e no lado servidor, pois assim, através do **fromJson**, o objeto **produto** que criamos em **ProdutoRest** no Java conterá em seus atributos exatamente o que enviamos do lado cliente para o lado servidor, **exceto** no atributo **id**, pois **não enviamos** nada do lado cliente com essa chave.

Agora, vamos fazer algo que já conhece: a conexão com o BD! Para isso, insira os códigos abaixo no local indicado:

```
try{
    Produto produto = new Gson().fromJson(produtoParam, Produto.class);
    Conexao conec = new Conexao();
    Connection conexao = conec.abrirConexao();

}catch(Exception e){
```

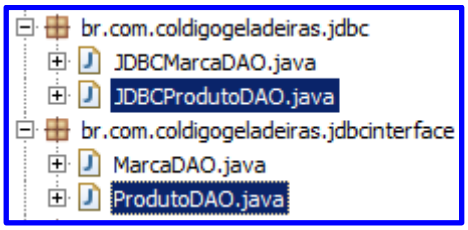
Se precisar de explicações, volte a OT anterior. Ah, não esqueça de importar as classes necessárias para que esse código funcione!
A seguir, também um processo familiar. Insira a seguinte linha:

```
try{
    Produto produto = new Gson().fromJson(produtoParam, Produto.class);
    Conexao conec = new Conexao();
    Connection conexao = conec.abrirConexao();

    JDBCProdutoDAO jdbcProduto = new JDBCProdutoDAO(conexao);


}catch(Exception e){
```

Com ela, criamos uma instância de JDBCProdutoDAO, atribuindo também a conexão para que possa usá-la, pois é nessa classe que teremos os métodos de interação com o BD relativas aos produtos. Porém, essa classe ainda não existe,

então vamos criá-la. Aliás, vamos criar também a interface que esse arquivo utilizará!

Crie em seu projeto **os arquivos selecionados** na imagem abaixo, cuidando para não errar os pacotes, e lembrando que **ProdutoDAO é uma interface**...



Na **interface ProdutoDAO**, apenas apontaremos a assinatura do método inserir, para que possamos criá-lo na classe JDBCProdutoDAO, então adicione o código destacado abaixo nela:

```
package br.com.coldigogeladeiras.jdbcinterface;

import br.com.coldigogeladeiras.modelo.Produto;

public interface ProdutoDAO {

    public boolean inserir(Produto produto);

}
```

Agora, vamos à **JDBCProdutoDAO**. Primeiro, vamos fazer sua programação inicial, conforme a imagem a seguir:

```
package br.com.coldigogeladeiras.jdbc;

import java.sql.Connection;

import br.com.coldigogeladeiras.jdbcinterface.ProdutoDAO;
import br.com.coldigogeladeiras.modelo.Produto;

public class JDBCProdutoDAO implements ProdutoDAO {

    private Connection conexao;

    public JDBCProdutoDAO(Connection conexao) {
        this.conexao = conexao;
    }

    public boolean inserir(Produto produto) {

    }

}
```

Como é bem parecida com a JDBCMarcaDAO, dispensamos apresentações. Vamos então trabalhar no **método inserir**, para gravarmos o produto informado

no BD. Mas **antes, vamos já chamá-lo na classe ProdutoRest**, conforme imagem a seguir:

```
try{
    Produto produto = new Gson().fromJson(produtoParam, Produto.class);
    Conexao conec = new Conexao();
    Connection conexao = conec.abrirConexao();

    JDBCProdutoDAO jdbcProduto = new JDBCProdutoDAO(conexao);
    boolean retorno = jdbcProduto.inserir(produto);

}catch(Exception e){
```

Veja que ao chamarmos o método, já criamos uma variável local para armazenar o retorno da execução dele...

Agora voltemos ao **método inserir em JDBCProdutoDAO.** Para começar sua implementação, vamos criar o comando SQL de inserção:

```
public boolean inserir(Produto produto) {

    String comando = "INSERT INTO produtos "
            + "(id, categoria, modelo, capacidade, valor, marcas_id) "
            + "VALUES (?,?,?,?,?,?)";

}
```

Mas o que são essas interrogações mesmo?

Calma, já chegaremos lá...
Continuando a programação do método, adicione o novo código em destaque...

```
public boolean inserir(Produto produto) {

    String comando = "INSERT INTO produtos "
            + "(id, categoria, modelo, capacidade, valor, marcas_id) "
            + "VALUES (?,?,?,?,?,?)";
    PreparedStatement p;

    try {

    } catch (SQLException e) {
        e.printStackTrace();
        return false;
    }
    return true;
}
```

Veja que, **diferentemente do que fizemos na busca de marcas,** vamos usar o PreparedStatement. Depois, criamos o **try-catch** no qual tentaremos executar o INSERT SQL no BD.

Faça então os **imports** das novas classes que estão apontando erro, como abaixo:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.SQLException;
```

Agora, vamos fazer a programação de dentro do **try**. Como ela é simples, vamos fazê-la **de uma vez só**, então siga a imagem abaixo e adicione os novos códigos:

```
try {

    //Prepara o comando para execução no BD em que nos conectamos
    p = this.conexao.prepareStatement(comando);

    //Substitui no comando os "?" pelos valores do produto
    p.setInt(1, produto.getId());
    p.setString(2, produto.getCategoria());
    p.setString(3, produto.getModelo());
    p.setInt(4, produto.getCapacidade());
    p.setFloat(5, produto.getValor());
    p.setInt(6, produto.getMarcaId());

    //Executa o comando no BD
    p.execute();

} catch (SQLException e) {
```

Explicando um pouco mais do que nos comentários do código:

- Primeiro, usamos a conexão pelo **atributo conexao** para **prepararmos o comando INSERT** que criamos para execução **e salvamos o comando preparado no objeto p** do tipo PreparedStatement.
- Depois, **com os métodos set**, **substituímos as interrogações** do comando **pelos valores do objeto produto**, indicando a **posição da interrogação** que deve ser substituída e qual é **o valor** que a substituirá (assim, na primeira linha, dizemos que a primeira interrogação terá o id do produto, na próxima linha trocamos a segunda interrogação pela categoria, e assim por diante). Veja também que os **métodos set** são de um **tipo de dados equivalente** ao que estamos indicando como segundo parâmetro em cada linha.
- Por fim, **o comando preparado,** já com as interrogações trocadas pelos valores desejados**, é executado**.

ATENÇÃO: em seu fichamento na OT anterior, pesquisou sobre a diferença entre as interfaces Statement e PreparedStatement. Por isso, talvez já tenhamos conversado sobre SQLInjection, mas se não o fizemos, por favor pesquise sobre o assunto e sobre como essas interfaces lidam com ela para discutirmos na validação.

Com isso, finalizamos a implementação da classe JDBCProdutoDAO para o momento.

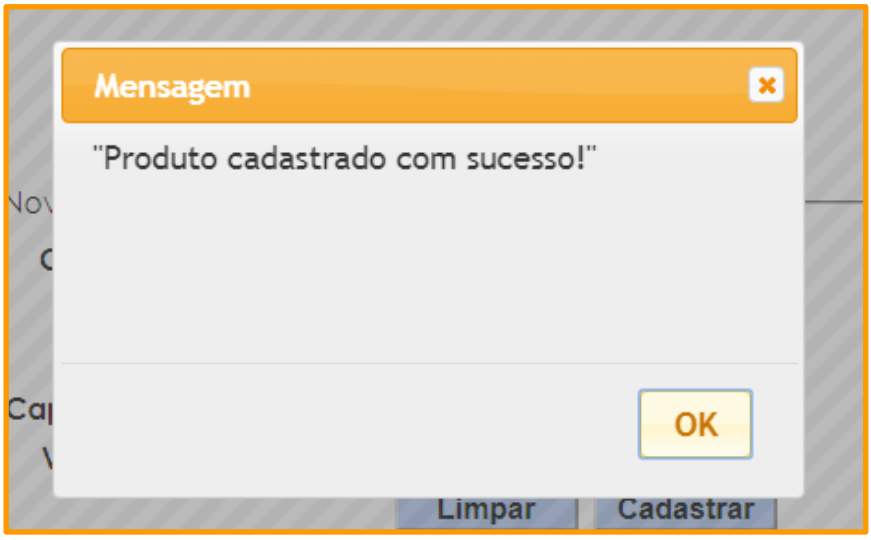
Agora, voltemos ao **método inserir de ProdutoRest**, para finalizar a programação do lado servidor, adicionando no local indicado o seguinte código:

```
public Response inserir(String produtoParam){

    try{
        Produto produto = new Gson().fromJson(produtoParam, Produto.class);
        Conexao conec = new Conexao();
        Connection conexao = conec.abrirConexao();

        JDBCProdutoDAO jdbcProduto = new JDBCProdutoDAO(conexao);
        boolean retorno = jdbcProduto.inserir(produto);
        String msg = "";

        if(retorno){
            msg = "Produto cadastrado com sucesso!";
        }else{
            msg = "Erro ao cadastrar produto.";
        }

        conec.fecharConexao();

        return this.buildResponse(msg);

    }catch(Exception e){
        e.printStackTrace();
        return this.buildErrorResponse(e.getMessage());
    }

}
```

Poucas diferenças em relação ao que fizemos na consulta de marcas nessa parte, mas vamos dar uma geral... Criamos uma **String *msg*** que, dependendo do retorno da tentativa de inserção no BD, terá uma mensagem de sucesso ou erro; **fechamos a conexão com o BD** e **retornamos uma Response de sucesso** para o lado cliente **com a frase de *msg***.

Testando...

Com o código finalizado, podemos testar... Preencha todos os dados de um produto no formulário, e veja se obtém este resultado:



Além disso, **confira também no Workbench** se o produto foi realmente cadastrado no BD, afinal de contas, pode ser que por alguma falha na programação a mensagem acima apareça mesmo quando os dados não forem inseridos no BD...

Deu tudo certo? Se sim, show! *Se não*... Bem, vamos a alguns erros possíveis além do código em si e como resolvê-los:

- Você não criou o banco no Workbench - nesse caso, bem, é só criá-lo;
- Você digitou o valor separando as casas decimais com vírgula - apesar de o separador de casas decimais em nosso país ser a vírgula, lembre-se que no BD e nas linguagens de programação, *o separador de casas decimais é o ponto*. Para resolver isso, *faça uma expressão regular para validar o valor digitado aceitando a vírgula e apenas 2 casas decimais*, e logo antes de chamar o Ajax, use o método JS replace para *trocar a vírgula digitada pelo usuário pelo ponto no objeto produto*.

Se não for nada disso, aí é provável que seja algo no código mesmo... Verifique com carinho o(s) erro(s) apontado(s) e corrija-os. Se precisar, chame o coleguinha do lado ou um professor :)

Reforçando o conhecimento adquirido

Para que reforce sua compreensão de como realizamos o processo de inserção dos produtos, pedimos com muito carinho que execute as seguintes ações:

- **Releia o documento com calma!**
- **Comente o código!**
- **Através de um fluxograma ou outra forma que não seja um texto corrido**, **esquematize o funcionamento do processo criado nesta OT**, desde o momento em que clicamos no botão de enviar, até quando a modal é exibida com o aviso de sucesso/erro. Represente os arquivos, quando um chama o outro, como o servidor encontra a Rest, enfim, **desenhe** o fluxo criado nesta OT. **O esquema criado será usado na validação.**

Conclusão

Com esta OT, encerramos os passos de cadastro de um produto. Veja que no processo, usamos 2 classes diferentes relativas ao BD: Statement e PreparedStatement. Esperamos que, após o fichamento realizado sobre as 2 mais as práticas, tenha ficado claro o funcionamento de ambas e que você consiga pensar em qual usar em cada caso...
Seguindo a trilha, nossa próxima etapa é exibir os dados dos produtos registrados, e iniciaremos esse processo na próxima OT. Até lá!

Após finalizar a OT, crie um novo commit no Git com o nome da OT e comunique um orientador para novas instruções.