

Curso: Análise e Desenvolvimento de Sistemas – ADS

Ano: 2023/1

Orientação Técnica (OT) – 21

Consulta REST - Programação

Introdução

Nesta OT, criaremos os códigos necessários para que sejam mostrados os produtos registrados no BD: no lado cliente, os códigos de requisição ao servidor e a criação da tabela para exibição dos dados e, no lado servidor, os códigos necessários para a consulta desses dados e retorno para o lado cliente.

Porém, apesar de parecer um processo comum, temos um detalhe muito relevante e específico das consultas de dados: não é apenas uma ação que deve acionar sua execução!

Para esclarecer isso, vamos exemplificar. Quando falamos do registro de algo no banco, a única ação do usuário que deve acionar esse processo deve ser o clique no botão de envio do formulário, certo? Mas quando a consulta dos dados deve acontecer?

Não seria coerente carregarmos os dados apenas uma vez, pois nossas ações posteriores na página deixariam a lista desatualizada... Imagine só um produto ser excluído com sucesso, mas continuar aparecendo na consulta! O usuário ficaria extremamente confuso, provavelmente tentaria excluir novamente esse produto, e como ele não existe mais, um erro seria gerado, instaurando o caos, não é, Rômulo Mendonça?



E como esse caos não é bem vindo, precisamos fazer com que cada ação do usuário que resulte em alguma alteração no BD atualize a tabela!

Além disso, há ainda o campo de pesquisa, que obviamente deve funcionar... Então, observe o seguinte: como ao final desta OT não teremos completado o CRUD, faremos aqui o que já puder ser feito, mas ainda falaremos mais sobre isso nas OTs seguintes.

Contextualização realizada, vamos em frente!

Criando a função JS de busca

Justamente por essa situação de termos vários momentos onde esse processo será necessário, vamos começar criando a função em si, e depois a chamaremos onde necessário. Assim, no final do arquivo **product.js**, mas ainda dentro do **\$(document).ready**, insira a função como abaixo:

```
//Busca no BD e exibe na página os produtos que atendam à solicitação do usuário  
COLDIGO.produto.buscar = function(){  
  
  
};
```

Como já explicado, pode ser que o usuário realize o filtro ou não, e a consulta deve aparecer de qualquer jeito. *Ah, mas os dados não vão ser diferentes?* [Independente, irmão!](#)

Teremos apenas uma função de pesquisa, que servirá aos dois propósitos.

Afinal, se tivéssemos 2 consultas separadas, uma para quando usamos o filtro e outra para quando não usamos, teríamos duas coisas para alterar caso alguma mudança de projeto fosse necessária... Inviável, né?

Considerando isso, vamos começar recebendo o valor digitado no filtro:

```
//Busca no BD e exibe na página os produtos que atendam à solicitação do usuário  
COLDIGO.produto.buscar = function(){  
    var valorBusca = $("#campoBuscaProduto").val();  
  
};
```

Assim, se o campo estiver vazio, a variável **valorBusca** também estará, mas se ele for preenchido, seu valor será o que foi digitado.

Agora, vamos fazer o Ajax que solicitará ao servidor os dados dos produtos registrados. para isso, copie o código novo (entre os destaques em amarelo):

```
//Busca no BD e exibe na página os produtos que atendam à solicitação do usuário
COLDIGO.produto.buscar = function(){
    var valorBusca = $("#campoBuscaProduto").val();

    $.ajax({
        type: "GET",
        url: COLDIGO.PATH + "produto/buscar",
        data: "valorBusca="+valorBusca,
        success: function(dados){

        },
        error: function(info){
            COLDIGO.exibirAviso("Erro ao consultar os contatos: " + info.status + " - " + info.statusText);
        }
    });
};
```

Primeiro, para deixar claro, deixaremos o conteúdo interno do **success** para depois, já que não vamos exibir os dados em uma modal como no caso anterior, e sim em uma tabela na própria página. Assim, nos concentramos no que realmente podemos identificar agora:

- Usamos o tipo **GET** de requisição. **Por quê?**
- Nossa **URL** é o **caminho padrão** + **"produto/buscar"**;
- enviaremos no **data** uma **chave chamada valorBusca** tendo como **valor** a **variável valorBusca**;
- a função de erro também já está pronta, com a nossa mensagem padrão a ser exibida em uma modal.

Com isso, podemos iniciar a programação do lado servidor.

Programação backend

Primeiro, vamos ao arquivo que será encontrado pelo Jersey para responder à solicitação criada acima: nossa classe **ProdutoRest**. Nela, crie abaixo do método existente, mas obviamente ainda dentro da classe, o código abaixo:

```
@GET
@Path("/buscar")
@Consumes("application/*")
@Produces(MediaType.APPLICATION_JSON)
public Response buscarPorNome(@QueryParam("valorBusca") String nome){
}
```

Nesse código, indicamos que:

- O **método** de requisição é **GET**;
- O **caminho** onde ela se encontra é **“/buscar”**, resultando em **“/ProjetoTrilhaRest/rest/produto/buscar”** (por que mesmo?);
- Ele **consome** alguma informação (Qual? Enviada como lá no Ajax?);
- Ele **produz** como resultado informações no formato **Json**.
- O nome do método é **buscarPorNome**, ele deve retornar uma **Response**, e tem um parâmetro...

Vamos explicar um pouco melhor esse parâmetro. Lá no lado cliente, no Ajax, colocamos a configuração do **data** uma **chave chamada valorBusca** com o **valor** digitado pelo usuário no campo de busca, certo? O **@QueryParam** é uma **anotação** com o poder de **buscar o valor de alguma chave enviada pelo lado cliente** e **passá-lo a alguma variável**. Nesse caso, passamos o valor da **chave valorBusca** para a **variável** do tipo **String chamada nome**. Traduzindo, o que for digitado no campo de busca do formulário será recebido aqui, e se o campo estiver vazio, essa variável também estará.

Para tudo isso funcionar, faça os *imports* necessários logo acima...

```
import javax.ws.rs.Consumes;
import javax.ws.rs.GET;
import javax.ws.rs.POST;
import javax.ws.rs.Path;
import javax.ws.rs.Produces;
import javax.ws.rs.QueryParam;
import javax.ws.rs.core.MediaType;
import javax.ws.rs.core.Response;
```

Como vai ver aí em seus códigos, ainda ficaremos com um erro na função nova após esses *imports*.

Agora, vamos à parte interna desse método. Insira dentro dele o try-catch, como na imagem a seguir:

```
public Response buscarPorNome(@QueryParam("valorBusca") String nome){
    try{
    }catch(Exception e){
        e.printStackTrace();
        return this.buildErrorResponse(e.getMessage());
    }
}
```

Nada de novo, certo? Veja se você lembra de tudo isso e siga em frente...

Como já aconteceu antes na consulta de marcas para carregarmos o campo select do formulário, temos aqui a possibilidade de **vários registros** atenderem à nossa solicitação. Assim, se faz necessária a **criação de uma lista** de produtos para retornarmos ao lado cliente. Faça, assim, a inserção do código destacado dentro do `try`:

```
try{  
    List<JsonObject> listaProdutos = new ArrayList<JsonObject>();  
}catch(Exception e){
```

Percebeu algo diferente do caso anterior? Observe que **não vamos usar a classe modelo Produto** como identificador do tipo de objeto aceito pela lista, e sim a **JsonObject**. Isso será explicado mais à frente.

Agora vamos aos novos *imports*:

```
import java.sql.Connection;  
import java.util.ArrayList;  
import java.util.List;  
  
import javax.ws.rs.Consumes;  
import javax.ws.rs.GET;  
import javax.ws.rs.POST;  
import javax.ws.rs.Path;  
import javax.ws.rs.Produces;  
import javax.ws.rs.QueryParam;  
import javax.ws.rs.core.MediaType;  
import javax.ws.rs.core.Response;  
  
import com.google.gson.Gson;  
import com.google.gson.JsonObject;
```

Continuando a programação do método de busca, dentro do `try`, prepare-o para poder se conectar ao BD para podermos fazer a busca:

```
try{  
    List<JsonObject> listaProdutos = new ArrayList<JsonObject>();  
  
    Conexao conec = new Conexao();  
    Connection conexao = conec.abrirConexao();  
    JDBCProdutoDAO jdbcProduto = new JDBCProdutoDAO(conexao);  
}catch(Exception e){
```

Mais uma vez, **já conhece o código**, por outra OT. Tente analisá-lo e explicar para si. Caso tenha dificuldade de se lembrar de algo, revise a primeira OT em que interagimos com o BD em nosso software.

Na última linha, referenciamos a **classe JDBCProdutoDAO**. Se bem lembra, ela **implementa a interface ProdutoDAO**, onde devem ser **indicados os métodos que ela deve implementar**. Assim, antes de fazermos o método de busca no BD, vamos adicionar esse método na interface supracitada, adicionando as linhas destacadas na imagem abaixo:

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

import java.util.List;

import com.google.gson.JsonObject;

import br.com.coldigogeladeiras.modelo.Produto;

public interface ProdutoDAO {

    public boolean inserir(Produto produto);
    public List<JsonObject> buscarPorNome(String nome);

}
```

Agora sim, abra a classe JDBCProdutoDAO, e abaixo do último método dela, crie o código da imagem abaixo:

```
public List<JsonObject> buscarPorNome(String nome) {

    //Inicia criação do comando SQL de busca
    String comando = "SELECT produtos.*, marcas.nome as marca FROM produtos "
        + "INNER JOIN marcas ON produtos.marcas_id = marcas.id ";
    //Se o nome não estiver vazio...
    if (!nome.equals("")) {
        //concatena no comando o WHERE buscando no nome do produto
        //o texto da variável nome
        comando += "WHERE modelo LIKE '%" + nome + "%' ";
    }
    //Finaliza o comando ordenando alfabeticamente por
    //categoria, marca e depois modelo.
    comando += "ORDER BY categoria ASC, marcas.nome ASC, modelo ASC";

}
```

Veja que, além da declaração do método em si, já fizemos uma manipulação da variável String comando, para que criemos o comando SQL de um modo mais dinâmico. Analise os comentários do código (lembrando que eles também devem ser copiados por você). A lógica é bem simples, mas em caso de dúvida discutiremos na validação. Fique atento apenas ao comando SQL, pois tem duas novidades que ainda não conheceu nas trilhas: **INNER JOIN** e **ORDER BY**.

Pesquise-os e nos explique o comando como um todo na validação.

Agora, continue a implementação do método, inserindo o código abaixo entre as partes destacadas:

```
comando += "ORDER BY categoria ASC, marcas.nome ASC, modelo ASC";

List<JsonObject> listaProdutos = new ArrayList<JsonObject>();
JsonObject produto = null;

try {

} catch (Exception e) {
    e.printStackTrace();
}

return listaProdutos;
}
```

Mais uma vez, tirando o JsonObject, nada de novo (já fizemos algo semelhante na consulta de marcas para o formulário de inserir produtos), mas em caso de dúvida, dê uma recapitulada...

Faça os devidos **import** para corrigir os erros dos últimos códigos:

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

import com.google.gson.JsonObject;
```

Vamos agora para a codificação do try, onde realizaremos a consulta no BD:

```
try {
    Statement stmt = conexao.createStatement();
    ResultSet rs = stmt.executeQuery(comando);

    while (rs.next()) {

    }

} catch (Exception e) {
```

Mais uma vez, idêntico ao que fizemos na OT de carregar marcas, assim caso você não se lembre das explicações de algo visto aqui, **recorra àquela OT. Se ainda for necessário, conversamos novamente na validação.** Faça os devidos **import** para corrigir os erros:

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.sql.Statement;
import java.util.ArrayList;
import java.util.List;
```

Agora, vamos implementar a parte dentro da estrutura de repetição, que será feita para cada linha do resultado de nossa consulta no banco:

```
while (rs.next()) {

    int id = rs.getInt("id");
    String categoria = rs.getString("categoria");
    String modelo = rs.getString("modelo");
    int capacidade = rs.getInt("capacidade");
    float valor = rs.getFloat("valor");
    String marcaNome = rs.getString("marca");

    if (categoria.equals("1")) {
        categoria = "Geladeira";
    } else if (categoria.equals("2")) {
        categoria = "Freezer";
    }

}
```

Repare que **pegamos cada campo** da linha atual de nossa consulta e **colocamos em variáveis**, e ainda **fazemos a transformação** dos valores da **categoria** que guardamos no BD **para seu “nome real”**.

Agora, finalizando o try, abaixo desse código insira:

```
} else if (categoria.equals("2")) {
    categoria = "Freezer";
}

produto = new JsonObject();
produto.addProperty("id", id);
produto.addProperty("categoria", categoria);
produto.addProperty("modelo", modelo);
produto.addProperty("capacidade", capacidade);
produto.addProperty("valor", valor);
produto.addProperty("marcaNome", marcaNome);

listaProdutos.add(produto);

}
```

No código aplicado da classe `JsonObject`, primeiro criamos uma nova instância dela, depois adicionamos uma nova propriedade para cada informação que queremos mostrar na tabela lá do frontend com os valores das variáveis que receberam os dados do BD, e adicionamos na lista `listaProdutos`. Podemos

continuar?



O que foi, João Kléber?

Ah, sim! Agora é o momento ideal para falarmos da tal **classe JsonObject!!!**

A função dela é **criar códigos no formato JSON, mas como um objeto Java**. Assim, temos o **benefício de criar um objeto dinâmico, sem clareza de quais são seus atributos**.

Aqui, a necessidade de fazermos uso dela vem do fato de que o **nosso modelo criado com a classe Produto não atende às nossas necessidade de exibição**. **Observe** a classe Produto e veja seus atributos:

```
private static final long serialVersionUID = 1L;  
  
private int id;  
private String categoria;  
private int marcaId;  
private String modelo;  
private int capacidade;  
private float valor;
```

Você acredita que o usuário do sistema conseguiria **identificar uma marca através de seu id**? Não é uma tarefa fácil, seria muito melhor se fosse o **nome da marca**, mas aí **não tem nenhum atributo para armazená-lo nessa classe modelo...** Também não seria coerente **mudar a classe modelo para isso, nem para cada consulta diferente** que fizermos, principalmente **pensando em possíveis INNER JOINs**, concorda?

Pois é, justamente por isso usaremos em consultas como essa, que **possuírem INNER JOIN e devam ser retornadas ao usuário de forma compreensível**, a classe JsonObject! Esperamos que você e o [JK](#) estejam satisfeitos...

Assunto encerrado, voltando ao contexto: só para esclarecer, terminamos a codificação da classe JDBCProdutoDAO para essa OT. Por conseguinte, continuaremos por onde esse método que criamos nela será chamado:

a **ProdutoRest**. Insira, então, o novo código dela (entre as partes destacadas na imagem abaixo):

```
try{  
    List<JsonObject> listaProdutos = new ArrayList<JsonObject>();  
  
    Conexao conec = new Conexao();  
    Connection conexao = conec.abrirConexao();  
    JDBCProdutoDAO jdbcProduto = new JDBCProdutoDAO(conexao);  
    listaProdutos = jdbcProduto.buscarPorNome(nome);  
    conec.fecharConexao();  
  
    String json = new Gson().toJson(listaProdutos);  
    return this.buildResponse(json);  
}catch(Exception e){
```

Novamente, sem novidades em relação ao que fizemos na consulta de marcas, então caso haja dúvidas, consulte-a... Finalizamos, assim, a programação backend da consulta de produtos!

Primeiro teste de funcionamento

Para fazermos nosso primeiro teste, verificando se os dados foram recebidos com sucesso no lado cliente (e, conseqüentemente, se tudo até esta etapa está OK), faça o seguinte código no success do Ajax que realiza a busca dos produtos:

```
$.ajax({  
    type: "GET",  
    url: COLDIGO.PATH + "produto/buscar",  
    data: "valorBusca="+valorBusca,  
    success: function(dados){  
        dados = JSON.parse(dados);  
        console.log(dados);  
    },  
    error: function(info){  
        COLDIGO.exibirAviso("Erro ao consu  
    }  
});
```

Com ele, como recebemos os produtos no parâmetro dados como uma String, usamos o JSON.parse para identificarmos que esses dados estão no formato JSON e sobrescrevemos a variável dados dessa maneira, e posteriormente usamos o console.log para exibirmos os dados recebidos na console do navegador (pois se estamos fazendo isso no JavaScript, linguagem executada no lado cliente, é óbvio que não vai aparecer na console do Eclipse...).

Agora só falta chamar a função! Primeiro, vamos fazer a chamada que fará com que a função seja executada ao se acessar a página do CRUD de produtos. Para isso, repare bem na imagem, e entre os locais destacados insira o chamado dessa função:

```
var valorBusca = $("#campoBuscaProduto").val();

$.ajax({
  type: "GET",
  url: COLDIGO.PATH + "produto/buscar",
  data: "valorBusca="+valorBusca,
  success: function(dados){

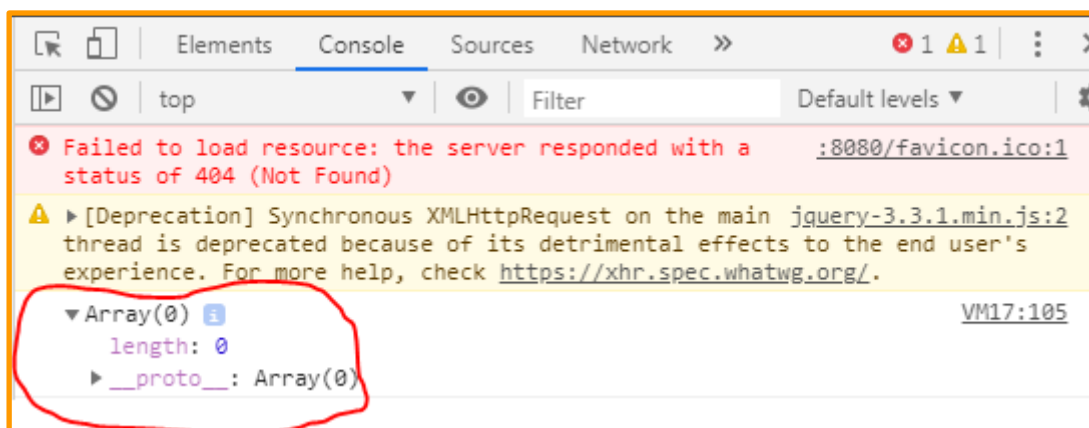
    dados = JSON.parse(dados);
    console.log(dados);

  },
  error: function(info){
    COLDIGO.exibirAviso("Erro ao consultar");
  }
});

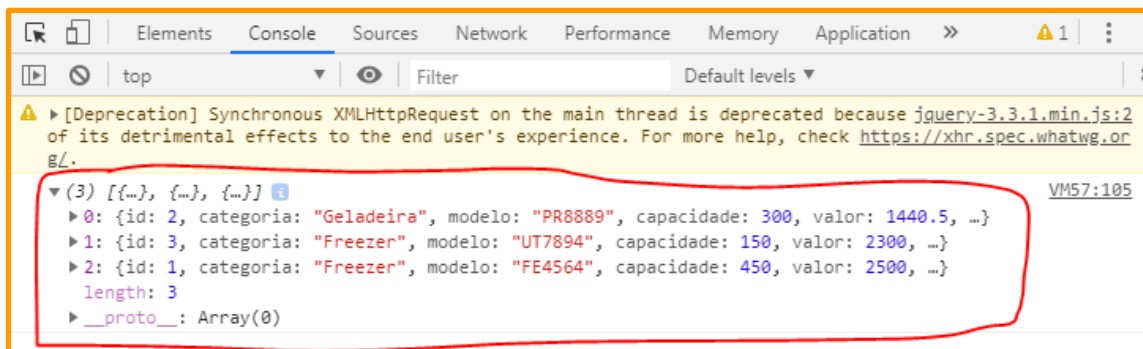
//Executa a função de busca ao carregar a página
COLDIGO.produto.buscar();
});
```

Repare que o chamado foi feito logo **após o fechamento da função e antes do fechamento do \$(document).ready**.

Agora, atualize a página e verifique a aba **console do inspetor do navegador**. Há **duas possibilidades** de resultado... A primeira, exibida na imagem abaixo, representa a situação de **não termos nenhum produto registrado**:



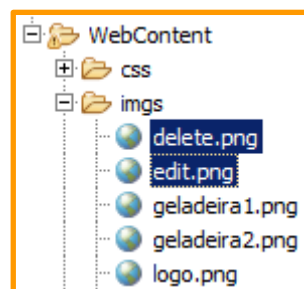
A segunda, vista abaixo, representa o caso de **termos um ou mais produtos registrados no BD** (na imagem, vemos 3 produtos).



ATENÇÃO: Teste as duas possibilidades! Caso alguma delas mostre algo de estranho, nos avise ou tente resolver o(s) erro(s) por conta própria, mas não prossiga!

Exibindo na página os produtos retornados pelo servidor

Agora que já sabemos que os dados estão chegando ao lado cliente, podemos fazer a exibição dos dados como realmente desejamos. Primeiro, faça o download das imagens **delete.png** e **edit.png**, disponibilizadas juntamente à OT, e as coloque na pasta **WebContent/imgs**, conforme abaixo:



Agora, vamos EXCLUIR aquele console.log que utilizamos e inserir o código que indica onde os dados devem ser carregados:

```
success: function(dados){
    dados = JSON.parse(dados);

    $("#listaProdutos").html(COLDIGO.produto.exibir(dados));
},
error: function(info){
```

Veja que indicamos que no elemento com **id listaProdutos** (onde está esse elemento?) inserimos um código HTML obtido na função **COLDIGO.produto.exibir**, que recebe como **parâmetro a variável dados** com os produtos registrados.

Mas pera aí... **Ainda não temos esse método...** Vamos criá-lo então? Localize os locais destacados em amarelo na imagem a seguir, para criá-lo no local certo:

```
});  
  
//Transforma os dados dos produtos recebidos do servidor em uma tabela HTML  
COLDIGO.produto.exibir = function(listaDeProdutos) {  
  
};  
  
//Executa a função de busca ao carregar a página  
COLDIGO.produto.buscar();  
});
```

Esse método, como pode observar no comentário do código, vai gerar uma tabela HTML com os dados vindos do lado servidor. Diferentemente de como fizemos com os campos **select** que criamos pelo JS anteriormente, vamos criar o código HTML da tabela de uma maneira menos Nutella e mais raiz. Insira, assim, o código abaixo no corpo do método:

```
COLDIGO.produto.exibir = function(listaDeProdutos) {  
  
    var tabela = "<table>" +  
        "<tr>" +  
        "<th>Categoria</th>" +  
        "<th>Marca</th>" +  
        "<th>Modelo</th>" +  
        "<th>Cap.(l)</th>" +  
        "<th>Valor</th>" +  
        "<th class='acoes'>Ações</th>" +  
        "</tr>";  
  
};
```

Veja que **recebemos os dados enviados como parâmetro** em uma variável local chamada **listaDeProdutos**, criamos **em forma de texto** as **tags de criação da tabela** e a **primeira linha**, com os **títulos** de cada coluna.

Como já explicado, temos 2 possíveis resultados: não há produtos ou há pelo menos um produto registrado... Primeiro, vamos fazer o necessário para quando não houverem produtos no BD, através do código abaixo, que você deve inserir entre os destaques da imagem:

```
"<th class='acoes'>Ações</th>" +
"</tr>";

if (listaDeProdutos != undefined && listaDeProdutos.length > 0){
} else if (listaDeProdutos == ""){
    tabela += "<tr><td colspan='6'>Nenhum registro encontrado</td></tr>";
}
tabela += "</table>";

return tabela;
};
```

Veja que a lógica não é complexa:

- Se a lista de produtos não estiver indefinida e seu comprimento for maior do que 0, é porque há produtos no BD (será feito futuramente o código sobre isso);
- Senão se a lista de produtos estiver vazia, concatenamos na variável tabela uma linha (tr) indicando que nenhum registro foi encontrado;
- Encerramos o HTML da tabela;
- Retornamos o valor contido na variável tabela (ou seja uma tabela em formato HTML) para quem chamou esse método.

RESPONDA: para que serve o colspan mesmo?

Se quiser, **pode testar:** exclua todos os produtos, atualize a página, e veja se o resultado obtido é o mesmo da imagem abaixo:

Categoria	Marca	Modelo	Cap.(l)	Valor	Ações
Nenhum registro encontrado					

ATENÇÃO: Se não ficar assim, já sabe, né?

Agora, vamos fazer a parte de **dentro do if**, onde devemos realmente criar os conteúdos necessários para **exibir cada produto registrado no BD**. Siga a imagem:


```
if (listaDeProdutos != undefined && listaDeProdutos.length > 0){



    for (var i=0; i<listaDeProdutos.length; i++){
        tabela += "<tr>" +
            "<td>" + listaDeProdutos[i].categoria + "</td>" +
            "<td>" + listaDeProdutos[i].marcaNome + "</td>" +
            "<td>" + listaDeProdutos[i].modelo + "</td>" +
            "<td>" + listaDeProdutos[i].capacidade + "</td>" +
            "<td>R$ " + listaDeProdutos[i].valor + "</td>" +
            "<td>" +
                "<a><img src='../img/edit.png' alt='Editar registro'></a> " +
                "<a><img src='../img/delete.png' alt='Excluir registro'></a>" +
            "</td>" +
            "</tr>"
    }

} else if (listaDeProdutos == ""){
    tabela += "<tr><td colspan='6'>Nenhum registro encontrado</td></tr>";
}
```

Veja que:

- Criamos um **for**, com uma **variável i** que começa de 0, indicamos que o código deve ser repetido enquanto **i** for menor que o tamanho da lista, e o incremento de **i** é 1.
- Dentro do **for**, concatenamos na variável **tabela** uma linha, com os dados advindos do BD nas 5 primeiras colunas (usando **i** para indicar a posição na lista) e com os botões de alterar e excluir na última coluna.

Agora, pode testar mais uma vez, mas com registros de produtos no BD. Se tudo estiver OK, vai ficar semelhante a isso:

Produtos registrados					
<div>Filtrar</div> <div> <input type="text" value="Pesquise pelo modelo"/> <input type="button" value="Buscar"/> </div>					
Categoria	Marca	Modelo	Cap.(l)	Valor	Ações
Geladeira	Prosdócimo	PR8889	300	R\$ 1440.5	 
Freezer	Consul	UT7894	150	R\$ 2300	 
Freezer	Eletrólux	FE4564	450	R\$ 2500	 
Copyright © Escola Sistêmica 2019					

ATENÇÃO: precisamos falar algo?

Tratando a exibição do valor do produto

Observe rapidamente os valores financeiros na tabela:

Valor
R\$ 1440.5
R\$ 2300
R\$ 2500

Apesar do R\$, os dados não estão com a formatação correta da nossa moeda. Para isso, vamos criar uma função que trate esses valores e os apresente corretamente. Como essa função pode nos servir em outros momentos (como na venda, ou em relatórios), a faremos no arquivo **admin.js**, para que seja aproveitada em todo o software! Assim, abra o arquivo, e abaixo da função de carregamento de página, insira o código da imagem a seguir.







```
//Exibe os valores financeiros no formato da moeda Real
COLDIGO.formatarDinheiro = function(valor){
    return valor.toFixed(2).replace('.', ',').replace(/(\d)(?=(\d{3})+\.)/g, "$1.");
}
```

Tente entender por conta o que se passa aqui, conversaremos com mais detalhes na validação.

Com a função criada, vamos voltar ao local onde a usaremos lá na criação da tabela de produtos, e altere a linha que exibe o valor do produto para o código a seguir:

```
"<td>" + listaDeProdutos[i].capacidade + "</td>" +
"<td>R$ " + COLDIGO.formatarDinheiro(listaDeProdutos[i].valor) + "</td>" +
"<td>" +
```

Assim, a formatação dos valores vai ficar como destacado abaixo:

Categoria	Marca	Modelo	Cap.(l)	Valor	Ações
Geladeira	Prosdócimo	PR8889	300	R\$ 1.440,50	 
Freezer	Consul	UT7894	150	R\$ 2.300,00	 
Freezer	Eletrolux	FE4564	450	R\$ 2.500,00	 

ATENÇÃO: precisamos falar algo?

Realizando a busca através do filtro

Como dissemos anteriormente, não é só no carregamento da página que o procedimento de busca dos produtos deve ser realizado. Por hora, vamos

também fazer com que o filtro funcione. Para isso, vá até o formulário de filtro no **arquivo product/index.html** e insira o código destacado no **botão Buscar** já existente:





```
<form id="filtraProduto" class="frmFiltrar">
  <fieldset>
    <legend>Filtrar</legend>
    <input type="text" name="txtbusca" id="campoBuscaProduto" placeholder="Pesquise pelo modelo">
    <button type="button" onclick="COLDIGO.produto.buscar()">Buscar</button>
  </fieldset>
</form>
```

Agora, teste o filtro, de acordo com os produtos que você mesmo inseriu:

Exemplo 1:

Filtrar

Buscar

Categoria	Marca	Modelo	Cap.(l)	Valor	Ações
Geladeira	Prosdócimo	PR8889	300	R\$ 1.440,50	 
Freezer	Consul	UT7894	150	R\$ 2.300,00	 

Exemplo 2:

Filtrar







Buscar

Categoria	Marca	Modelo	Cap.(l)	Valor	Ações
Nenhum registro encontrado					

Exemplo 3:

Filtrar

Buscar

Categoria	Marca	Modelo	Cap.(l)	Valor	Ações
Geladeira	Prosdócimo	PR8889	300	R\$ 1.440,50	 
Freezer	Consul	UT7894	150	R\$ 2.300,00	 
Freezer	Elerolux	FE4564	450	R\$ 2.500,00	 

Com isso, finalizamos a parte explicativa desta OT!

Reforçando o conhecimento adquirido

Quanta novidade, né? Porém, é tudo interconectado, então não tem como separar mais se quisermos manter sentido em cada parte do código... Assim, 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 carregamos a página até o momento em que os dados são mostrados na tabela. 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.**
- Sobre os dois modos de criar HTML com JS (através de funções da linguagem ou de codificação dentro de Strings), **qual achou mais interessante? Traga sua opinião para discutirmos...**
- Além do momento em que carregamos a página e a partir do clique no botão “Buscar”, há **mais um momento onde seria interessante chamarmos o método JS de busca que criamos nesta OT.** Pense bem no que já fizemos, e se chegar a uma conclusão de que momento é esse, **faça esse novo chamado!** Se não entendeu ou não souber como fazer, **converse com um orientador...**

Conclusão

Com essa OT, 50% do CRUD foi realizado! Ainda nos faltam as ações relativas justamente aos dois botões que aparecem em cada produto consultado: alterar e excluir um produto.

Caso não tenha notado, um dos dados que enviamos do lado servidor para o lado cliente não foi usado! Sabe dizer qual? Imagina o motivo? Essa e outras perguntas você terá a resposta no Globo-Repórter nas próximas OTs...

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