PHP – PDO II – Conexão com o BD

Produto

Olá, meu nome é Mario, sejam bem vindos à segunda parte do curso de PHP com PDO. Neste curso, falaremos sobre o que envolve os produtos de nosso sistema, além de fazermos algumas melhorias na parte de categorias, desenvolvida na [primeira parte](https://cursos.alura.com.br/course/php-pdo-parte-1).

Então, se você ainda não fez a primeira parte do curso, sugiro fortemente que o faça, para que você tenha toda a base necessária, com dicas sobre as conexões, as formas com as quais poderemos fazer consultas nos bancos de dados, e a manipulação de informações do mesmo.

Com isso, seguiremos trabalharemos com a preparação de queries, funcionalidade muito interessante trazida pelo PDO, que implica em basicamente passarmos um parâmetro em vez de concatenar valores diretamente em uma query.

Isto traz muito mais segurança, além de deixar o código muito mais organizado.

Então, vamos começar?

**Aula 1**

No curso anterior, conseguimos fazer a inclusão, exclusão e edição de categorias, além de visualizá-las e carregá-las em uma listagem. Porém, nosso sistema também é composto de produtos, e precisamos desenvolver todo o fluxo para lidarmos com eles também, isto é, para cadastramentos, exclusões e listagem.

Vamos começar a trabalhar com isso?

Acessando o menu "Produtos" na página do navegador, veremos "O Senhor dos Anéis" cadastrado, de *id* 1, com preço R$ 80,55, quantidade 2 e categoria Livros. Esta tela é um pouco mais complexa que a de categorias, por trazer mais informações.

Vamos abrir produtos.php, localizado na raiz do projeto. Ele possui basicamente a mesma estrutura dos demais arquivos com que trabalhamos anteriormente, com tabelas mocadas, isto é, fixas ao HTML, não dinâmico, servindo apenas para termos uma visão do front end.

Começaremos repetindo o que fizemos com as categorias, criando uma classe para representar a entidade Produto no sistema. Ou seja, em classes, criaremos Produto.php, em que representaremos todos os campos da parte de produtos com atributos:

<?php

class Produto

{

public $id;

public $nome;

public $preco;

public $quantidade;

public $categoria;

}

Porém, como estas informações estão representadas no banco de dados? Para conferir, vamos abrir o *script* de criação do banco de dados, banco\_de\_dados.sql. Na parte de produtos, temos o id, nome, preco, quantidade e categoria\_id, pois logo em seguida criamos uma FOREIGN KEY, fk\_categorias, que funciona como chave estrangeira no banco de dados, cuja categoria\_idrepresenta o id da tabela categorias.

Para trabalharmos da mesma forma como fazemos no banco de dados na nossa classe, o atributo categoria, previamente criado, será convertido para categoria\_id.

Começaremos pela listagem de produtos, que será apresentada ao usuário, e trabalharemos com o método listar() que a retornará para nós. Vamos manter o padrão estabelecido na parte de categorias - sabemos que todas as classes que representam as entidades terão o método listar()para os registros da entidade.

Para a *query* de consulta, você poderá estar se perguntando se não teríamos o mesmo efeito se escrevessemos "\*$ FROM nome da tabela". Sim, teríamos. No entanto, isto não é nem um pouco recomendado, pois não sabemos como esta estrutura ficará futuramente. É legal sermos fiéis ao que precisamos no SELECT. Neste caso, como todos estes campos serão utilizados, deixaremos isso bem explícito em nosso código.

Faremos também a requisição de conexão, lembrando que agora não precisaremos de nenhum require\_once na classe para pegarmos o método de conexão que temos, pois já temos o ***autoloader***, então, quando este ponto for alcançado, ele tentará buscar por esta classe, cairá no *autoloader*, o qual fará o *require*, e aí, sim, o método funcionará.

Executaremos a *query* com o método query() para obtermos retorno a partir da função fetchAll(), da classe PDOStatement.

O código ficará assim:

<?php

class Produto

{

public $id;

public $nome;

public $preco;

public $quantidade;

public $categoria\_id;

public static function listar()

{

$query = "SELECT id, nome, preco, quantidade, categoria\_id FROM produtos";

$conexao = Conexao::pegarConexao();

$resultado = $conexao->query($query);

$lista = $resultado->fetchAll();

return $lista;

}

}

Vamos testar para ver se funciona, utilizando a tag <pre> e print\_r no próprio arquivo produtos.php, em que serão listadas estas informações. Antes de mais nada, faremos o require\_once para o arquivo global.php, acrescentando a linha <?php require\_once 'global.php' ?> no topo do código.

Agora temos o arquivo global.php, que carrega todas as informações de que nossa página precisará. Tudo que chamamos de compilador, que é este bloco de código e não é uma classe, precisa estar contido em um *Try/Catch*.

<?php

try {

$lista = Produto::listar();

echo '<pre>';

print\_r($lista);

echo '</pre>';

exit;

} catch (Exception $e) {

Erro::trataErro($e);

}

?>

Feito isso, testaremos novamente, e será exibida uma página com uma matriz de vários índices, sendo que cada um é um *array* que representa uma linha do nosso banco de dados.

Temos os livros cadastrados, jogos, e assim por diante. Vimos, portanto, que nossa lista funciona direitinho. Podemos deletar o debug no try, mantendo apenas o método listar():

<?php

try {

$lista = Produto::listar();

} catch (Exception $e) {

Erro::trataErro($e);

}

?>

Agora, precisaremos ir à parte do HTML em produtos.php, em que há uma tabela e, entre as tags <tbody> e <tr> faremos o foreach para o laço que irá passar por cada linha do *array* e imprimir os valores de que precisamos.

Usaremos a forma alternativa de controle do PHP neste momento, pois estamos misturando-o com HTML, para fins de organização do código.

<?php foreach ($lista as $linha): ?>

<tr>

<!-- informações referentes a um dos livros cadastrados -->

</tr>

<?php endforeach ?>

Ao abrirmos um PHP usando um comando e fechando-o logo em seguida, não é preciso utilizar o ponto e vírgula (**;**), pois o fechamento do PHP já é o suficiente.

Vamos testar o que foi feito até aqui! Serão impressos os mesmos valores em cinco linhas, que é o que temos no nosso banco de dados. Desta vez, queremos que apareçam os valores de cada linha.

Em relação ao preço, só estamos armazenando o valor, não o R$. Sendo assim, manteremos isto no HTML, colocando o PHP depois dele. Voltaremos a produtos.php e o deixaremos assim:

<?php foreach ($lista as $linha): ?>

<tr>

<td><?php echo $linha['id'] ?></td>

<td><?php echo $linha['nome'] ?></td>

<td>R$ <?php echo $linha['preco'] ?></td>

<td><?php echo $linha['quantidade'] ?></td>

<td><?php echo $linha['categoria\_id'] ?></td>

<td><a href="/produtos-editar.php" class="btn btn-info">Editar</a></td>

<td><a href="#" class="btn btn-danger">Excluir</a></td>

</tr>

<?php endforeach ?>

Testaremos novamente e veremos que temos o id de cada registro, bem como os nomes, preços e quantidades. Porém são exibidos os números das categorias, e não os nomes, e isto não é o que queremos por não ser nem um pouco prático.

Vamos trabalhar de maneira que tenhamos um carregamento completo do que precisamos desta listagem, alterando nossa *query*. No arquivo Produto.php, quebraremos a linha referente à *query*para que ela não fique muito extensa, o que não trará nenhum problema no PHP desde que se coloque o ponto e vírgula no final.

public static function listar()

{

$query = "SELECT p.id, p.nome, preco, quantidade, categoria\_id, c.nome as categoria\_nome

FROM produtos p

INNER JOIN categorias c ON p.categoria\_id = c.id";

$conexao = Conexao::pegarConexao();

$resultado = $conexao->query($query);

$lista = $resultado->fetchAll();

return $lista;

}

Ao usarmos categoria\_nome conseguimos diferenciar o *array* retornado pelo fetchAll(), que seguirá exatamente o que colocamos no SELECT. Por meio dos *alias*, com as, damos um apelido para o campo, que em vez de aparecer como nome, será c.nome.

Não alteraremos nada e veremos se tudo continua funcionando como antes. As categorias ainda aparecem com o número, e não o nome. Assim, no campo em que as apresentamos, usaremoscategoria\_nome no foreach do HTML, em vez de imprimirmos categoria\_id. Em produtos.php, portanto, teremos:

<td><?php echo $linha['categoria\_nome'] ?></td>

Salvaremos e atualizaremos a página, e as categorias são apresentadas como gostaríamos, com seus nomes, e não números!

**Aula 2**

Nossa listagem de produtos está funcionando direitinho, com um pequeno porém: vocês lembram que na listagem de categorias acrescentamos um if para verificar quantos registros são retornados, e assim exibirmos uma mensagem amigável ao usuário caso não exista nada cadastrado no banco de dados?

Precisaremos fazer o mesmo nesta parte de produtos. Vamos abrir produtos.php e incluir na tabela um if com o método count():

<div class="col-md-12">

<?php if (count($lista) > 0): ?>

<table class="table">

<thead>

<tr>

<!-- código omitido -->

</tr>

</thead>

</table>

<?php else: ?>

<p>Nenhum produto cadastrado</p>

<?php endif ?>

</div>

Testando no navegador, veremos que isso não afeta nossa página, isto é, tudo continua funcionando bem. Para confirmarmos, pode-se forçar a regra inversa, substituindo > por <= na linha abaixo:

<?php if (count($lista) <= 0): ?>

Assim, quando atualizamos a página, será exibida a mensagem "Nenhum produto cadastrado". Deixaremos o código da maneira como estava anteriormente.

Outro ponto feito nas categorias que ainda não implementamos são os ids dos nossos botões. Ao passarmos o mouse sobre eles, seus endereços não mostram nenhum id ou ação a ser tomada.

Em produtos.php, passaremos então o valor do id do produto via query string, simplesmente acrescentando o ponto de interrogação, e a variável igual a valor, que no caso é dinâmico.

O botão **Excluir** ainda não possui um link, porém poderemos estipular que ele seguirá com o mesmo padrão da página de categoria. O código ficará da seguinte maneira:

<td><a href="/produtos-editar.php?id=<?php echo $linha['id'] ?>" class="btn btn-info">Editar</a></td>

<td><a href="/produtos-excluir-post.php?id=<?php echo $linha['id'] ?>" class="btn btn-danger">Excluir</a></td>

Feito isso, vamos salvar o arquivo e atualizar a página. Ao passarmos o mouse novamente sobre os botões, desta vez veremos nos links no canto inferior esquerdo o id correspondente aos produtos.

Parece um tanto desorganizado, porque a listagem está seguindo a ordem das categorias, e não dos nomes dos produtos. Vamos alterar isso indo à classe Produto.php, em que acrescentaremos um ORDER BY na $query:

$query = "SELECT p.id, p.nome, preco, quantidade, categoria\_id, c.nome as categoria\_nome

FROM produtos p

INNER JOIN categorias c ON p.categoria\_id = c.id

ORDER BY p.nome ";

Tudo certo! E você poderá organizar a listagem da maneira como achar melhor.

Nessa aula fizemos nossa classe **Produto** e nosso primeiro método Produto::listar(), no qual usamos um INNER JOIN na tabela de **categorias** para trazer o nome da categoria na listagem dos produtos. Sem isso só conseguíamos mostrar o **ID** da categoria, que é um campo da tabela de produtos.

Existe uma outra forma.

O que poderíamos ter feito para não usar o INNER JOIN em nosso **SELECT** é, dentro do foreach, da listagem de produtos, ter usado o método carrega da Classe Categoria para carregar a entidade de cada categoria e, assim, exibido o nome dela a partir da própria entidade. Isso é o que chamamos de Lasy Loading.

Embora para essa situação usar Lazy Loading não seja a melhor escolha, pois vai gerar um excesso grande de processamento, além de deixar nosso código um pouco desorganizado, vamos alterar nosso código de listagem, para ver essa técnica funcionando.

**OBS**: Mais pra frente em nosso curso, vamos utilizar a técnica de Lazy Loading de uma forma parecida, onde tem sua aplicabilidade muito mais válida. =)

Precisamos alterar o método listar() da classe classes/Produto.php:

public static function listar()

{

$query = "SELECT p.id, p.nome, preco, quantidade, categoria\_id, c.nome as categoria\_nome

FROM produtos p

INNER JOIN categorias c ON p.categoria\_id = c.id

ORDER BY p.nome ";

$conexao = Conexao::pegarConexao();

$resultado = $conexao->query($query);

$lista = $resultado->fetchAll();

return $lista;

}

E o bloco do foreach() do arquivo produtos.php:

<?php foreach ($lista as $linha): ?>

<tr>

<td><?php echo $linha['id'] ?></td>

<td><?php echo $linha['nome'] ?></td>

<td>R$ <?php echo $linha['preco'] ?></td>

<td><?php echo $linha['quantidade'] ?></td>

<td><?php echo $linha['categoria\_nome'] ?></td>

<td><a href="/produtos-editar.php?id=<?php echo $linha['id'] ?>" class="btn btn-info">Editar</a></td>

<td><a href="/produtos-excluir-post.php?id=<?php echo $linha['id'] ?>" class="btn btn-danger">Excluir</a></td>

</tr>

<?php endforeach ?

**Opinião do instrutor**

Para fazer isso, vamos começar **tirando** todo o bloco do INNER JOIN e suas referências de nosso SELECT dessa forma:

$query = "SELECT p.id, p.nome, preco, quantidade, categoria\_id

FROM produtos p

ORDER BY p.nome ";

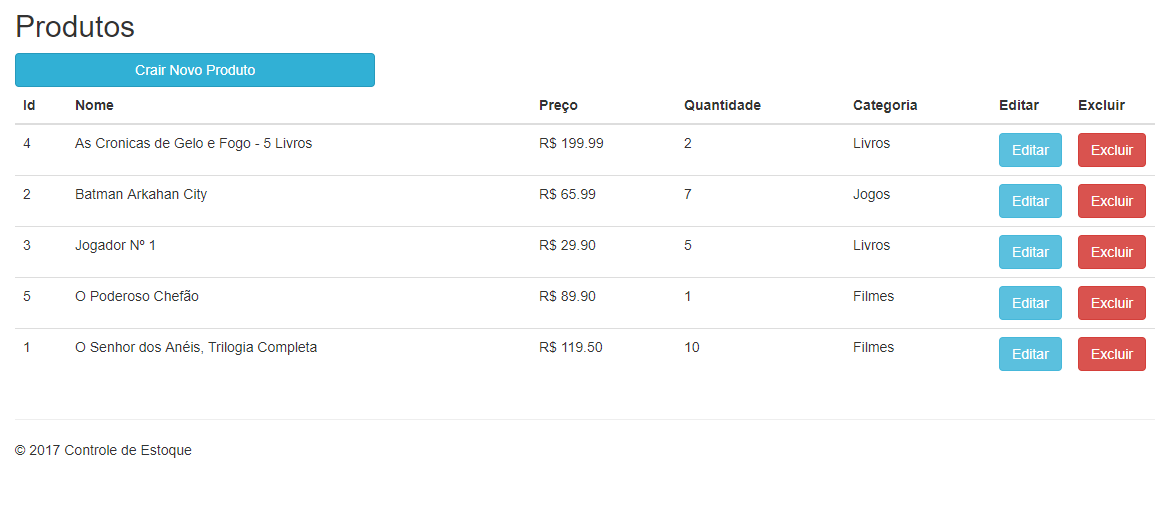
Agora, vamos alterar, dentro do bloco do foreach() do arquivo produtos.php, para tirar a refeência ao campo do nome da categoria que vinha em nossa query antes da alteração. Alterando a linha: <td><?php echo $linha['categoria\_nome'] ?></td> para <td><?php echo $linha['categoria\_id'] ?></td>.

Agora, vamos criar um novo bloco de php logo abaixo da linha do foreach() e colocar o seguinte código:

<?php $categoria = new Categoria($linha['categoria\_id']); ?>

E por fim, vamos usar o comando echo com o atributo nome da classe Categoria em vez de mostrar o ID da categoria na coluna. Alterando a linha <td><?php echo $linha['categoria\_id'] ?></td> para <td><?php echo $categoria->nome ?></td>

Teremos o mesmo resultado de antes, mas com uma forma diferente de obter os dados!



**Aula 3**

Neste momento, tudo o que fizemos em relação à listagem de produtos funciona bem. Em seguida, queremos conseguir incluir novos produtos no sistema. Para isso, vamos clicar no botão "Criar novo produto" na página do navegador.

Somos redirecionados a outra página com um formulário contendo nome do produto, preço, quantidade e categoria - que não está sendo listado corretamente -, além do botão "Salvar". Começaremos ajustando esta página, produtos-criar.php, para trabalharmos em cima dela depois.

De imediato pode-se notar que estão faltando os names nos inputs, para podermos recuperar as informações quando elas forem submetidas ao servidor.

Assim, em cada um destes campos incluiremos o name correspondente no código: name="nome"para "Nome do Produto", name="preco" para "Preço do Produto", name="quantidade" para "Quantidade de Produto". E no select referente à "Categoria do Produto", acrescentaremos name="categoria\_id".

Agora, precisaremos fazer com que esta lista de categorias do formulário funcione, ou seja, é necessário um método para listarmos isso. Ou poderemos usar o método que já criamos na classe Categoria.php, pois **listar** é uma responsabilidade delas. Vamos relembrá-lo?

Em Produto.php criamos um método estático, o que não ocorre em Categoria.php. A diferença entre eles é que um método estático não se relaciona com nada além do que existe na própria classe, no objeto propriamente dito. Significa que ele não faz referência a outros métodos ou atributos do objeto.

O método estático se encaixa bem no nosso caso, então, aproveitando, vamos regularizar isso, convertendo o método, a ser utilizado na listagem de categorias:

public static function listar()

{

$query = "SELECT id, nome FROM categorias";

$conexao = Conexao::pegarConexao();

$resultado = $conexao->query($query);

$lista = $resultado->fetchAll();

return $lista;

}

No try de categorias.php, fazemos a instância e depois chamamos o método listar(). Como ele é estático agora, não precisaremos instanciá-lo, e o código ficará assim:

<?php

try {

$lista = Categoria::listar();

} catch(Exception $e) {

Erro::trataErro($e);

}

Vamos conferir se a página de listagem de categorias continua funcionando como esperado.

Tendo confirmado isto, voltaremos a produtos-criar.php para fazermos o require\_once do arquivo global.php, e criarmos um bloco de PHP para o *Try/Catch*.

Salvaremos o código e retornaremos à página de inserção de novo produto no navegador. Aparentemente, tudo funciona corretamente, mas para confirmar incluiremos o *debug* para popularmos o valor:

<?php require\_once 'global.php' ?>

<?php

try {

$listaCategoria = Categoria::listar();

echo '<pre>';

print\_r($listaCategoria);

echo '</pre>';

exit;

} catch (Exception $e) {

Erro::trataErro($e);

}

?>

A listagem de categorias aparece da maneira como esperávamos! Vamos deletar o *debug* que fizemos, e incluir um foreach no select para termos várias opções.

No select do arquivo produtos-criar.php há dois valores, correspondentes a "Livros" e "Revistas". Vamos manter apenas um deles, e incluir o foreach:

<div class="form-group">

<label for="nome">Categoria do Produto</label>

<select class="form-control" name="categoria\_id">

<?php foreach ($listaCategoria as $linha): ?>

<option value="<?php echo $linha['id'] ?>"><?php echo $linha['nome'] ?></option>

<?php endforeach ?>

</select>

</div>

Com isso, temos mais opções de seleção na parte de "Categoria do Produto", quando um novo produto for inserido na tabela. Para sabermos se o value está funcionando bem, podemos inspecionar o elemento no navegador e confirmar que cada categoria possui um valor diferente:

<option value="1">Livros</option>

<option value="2">Jogos</option>

<option value="3">Filmes</option>

<option value="4">Roupas</option>

<option value="5">Revistas</option>

E já que organizamos os produtos em ordem alfabética a partir dos seus nomes, vamos fazer o mesmo em relação às categorias. Na classe Categoria.php, deixaremos o query da seguinte forma:

$query = "SELECT id, nome FROM categorias ORDER BY nome";

Agora temos a listagem das categorias da tabela de produtos! Mas e se não houver nenhuma categoria cadastrada no sistema? Imaginem a situação em que um sistema acabou de ser instalado, e não existir nada além do banco de dados, e o usuário quer cadastrar um produto. O que acontecerá?

Não poderemos inserir um novo produto sem que haja uma categoria vinculada a ele.

Portanto, precisaremos fazer uma verificação que contará o número de categorias que temos. E se não existir nenhuma, não deixaremos que um usuário cadastre um novo produto.

Sendo assim, vamos retornar a produtos-criar.php e colocar o if com count antes da exibição do formulário ao usuário, entre as <div>s do título e a abertura do <form>:

<div class="row">

<!-- código omitido -->

</div>

<?php if (count($listaCategoria) > 0): ?>

<form action="produtos-criar-post.php" method="post">

<!-- código omitido -->

</form>

<?php else: ?>

<p>Nenhuma categoria cadastrada no sistema. Por favor, crie uma categoria antes de cadastrar um produto.</p>

<?php endif ?>

Visualmente, nada foi afetado em nossa página. Então, para testarmos, usaremos a técnica de inverter o if, desta vez acrescentando **!**, o operador de negação, para todo o argumento:

<?php if (!(count($listaCategoria) > 0)): ?>

A mensagem será exibida corretamente. Vamos tirar a negação para voltarmos ao estado anterior.

**Aula 4**

Nossa tela de inserção de novos produtos está praticamente finalizada, falta apenas acrescentarmos o método para o cadastro em si. Faremos isto seguindo o padrão definido na parte de categorias. Então, abriremos Produto.php, em que teremos:

public function inserir()

{

$query = "INSERT INTO produtos (nome, preco, quantidade, categoria\_id)

VALUES ('" . $this->nome . "', " . $this->preco . ", " . $this->quantidade . ", " . $this->categoria\_id . ")";

$conexao = Conexao::pegarConexao();

$conexao->exec($query);

}

Para verificar se o código funciona, precisaremos criar uma página na raiz do projeto, que instanciará a classe Produto.php, irá popular os atributos que queremos inserir, e que depois chamará este método, da mesma forma que fizemos com as categorias.

Também precisamos fazer com que o formulário envie os dados para este arquivo. Portanto, vamos voltar a produtos-criar.php, em que veremos que a tag <form> já está com method="post", corretamente.

Criaremos produtos-criar-post.php, em que chamaremos o require\_once do arquivo global, e acrescentaremos o *Try/Catch*. Precisamos resgatar os valores obtidos por meio do formulário, instanciando uma classe Produto em branco, e populando os atributos desta classe.

<?php

require\_once 'global.php';

try {

$nome = $\_POST['nome'];

$quantidade = $\_POST['quantidade'];

$preco = $\_POST['preco'];

$categoria\_id = $\_POST['categoria\_id'];

$produto = new Produto();

$produto->nome = $nome;

$produto->quantidade = $quantidade;

$produto->preco = $preco;

$produto->categoria\_id = $categoria\_id;

$produto->inserir();

header('Location: produtos.php');

} catch (Exception $e) {

Erro::trataErro($e);

}

Vamos testar e ver o que acontece? Voltaremos à página de cadastro de novos produtos, cujo nome é "Camiseta Azul", o preço é "40", quantidade "10" e categoria "Roupas". Clicaremos no botão "Salvar", e veremos que o produto foi cadastrado com sucesso, constando na listagem junto aos demais.

**OBS:**

O método PDO::prepare(), assim como o PDO::query(), retorna um objeto do tipo PDOStatement, mas com a diferença que a query não é executada automaticamente.

O objeto PDOStatement possui diversos métodos que podemos usar, em diferentes situações que podem ajudar. Vamos agora dar uma listada nas possibilidade que temos com esse objeto!

PDOStatement::fetch

Retorna a próxima linha de um resultset. Podemos usar o método fetch, combinado por exemplo com um while() para buscar cada linha do resultado. Ele sempre começa na primeira linha do resultado de sua query e a cada vez que o método é chamado, ele vai para a próxima linha. Quando chega na ultima linha, o método retorna um valor false.

Veja um exemplo de um SELECT em nossa tabela de produtos com isso:

<?php

require\_once 'global.php';

$conexao = Conexao::pegarConexao();

$query = "SELECT nome FROM produtos";

$stmt = $conexao->prepare($query);

$stmt->execute();

$row = $stmt->fetch();

while ($row) {

echo 'Produto:' . $row['nome'] . '<br>';

$row = $stmt->fetch();

}

PDOStatement::fetchObject()

O fetchObject() funciona de forma muito parecida com o método fetch(), mas com a diferença simples de que, o retorno não vai ser um array onde, cada indice representa uma coluna na entidade e sim, um objeto, onde, cada atributo representa a entidade.

Veja o mesmo exemplo com a aplicação de PDOStatement::fetchObject():

<?php

require\_once 'global.php';

$conexao = Conexao::pegarConexao();

$query = "SELECT nome FROM produtos";

$stmt = $conexao->prepare($query);

$stmt->execute();

$row = $stmt->fetchObject();

while ($row) {

echo 'Produto:' . $row->nome . '<br>';

$row = $stmt->fetchObject();

}

PDOStatement::rowCount

Sempre que usamos uma instrução SQL do tipo INSERT, UPDATE ou DELETE, não teremos um resultset, Mas, temos um retorno do Banco de Dados com a quantidades de linhas que foram afetadas pelo nosso script. E esse método justamente retorna isso para nós. Veja o exemplo abaixo:

<?php

require\_once 'global.php';

$novo\_preco = 11;

$conexao = Conexao::pegarConexao();

$query = "UPDATE produtos SET preco = :preco";

$stmt = $conexao->prepare($query);

$stmt->bindValue(':preco', $novo\_preco);

$stmt->execute();

echo $stmt->rowCount() . ' Linhas atualizadas!';

E temos o seguinte resultado: 13 Linhas atualizadas!.

<http://php.net/manual/pt_BR/class.pdostatement.php>

**Aula 5**

Após configurarmos a parte de cadastro de novos produtos, e antes de continuarmos com a edição e exclusão dos mesmos, faremos uma reformulação em nosso código para **reforçarmos sua segurança**, no sentido de que, da maneira como ele se encontra, principalmente em relação às *queries* de bancos de dados feitos com PDO, acabamos gerando risco de vulnerabilidade no sistema.

Para demonstrar isso, vamos abrir Produto.php, com o método que acabamos de criar, inserir(). Nele, estamos concatenando os valores inseridos diretamente à *query*, e isto gera grande risco de segurança por não fazermos nenhum tratamento no valor que o usuário coloca no formulário, após o qual simplesmente fazemos a chamada do método de INSERT.

Existem várias maneiras de prevenirmos *SQL Injections*, e as mais claras e simples implicam em pegarmos o valor e tratarmos o método inserir() antes de chamá-lo, procurando por padrões que normalmente são utilizados em uma tentativa de *SQL Injection* - como caracteres especiais como ponto e vírgula, traço duplo (um comentário do SQL), aspas simples, or 1 = 1. Estas normalmente são as técnicas mais usadas.

O PDO, no entanto, traz uma funcionalidade para o tratamento de SQL de uma forma muito mais fácil e que, além disso, deixará as *queries* do nosso código mais organizadas. Até então, já tínhamos usado os métodos query() e exec(). Desta vez, faremos diferente, preparando esta execução com o método prepare(), e depois usando o execute().

No código de Produto.php, precisaremos alterar basicamente todas as concatenações na parte de values do INSERT, colocando chaves para indicar que um valor será passado de forma dinâmica. Não usaremos aspas simples, e sim dois pontos (**:**) antes do valor, ainda como *string*.

O método prepare() é bem parecido com o query(), e ele nos retornará um *PHP Statement* mas, diferentemente do query(), que devolve o *Statement* (stmt) já executado, o prepare(), como o próprio nome indica, irá nos devolvê-lo preparado, mas ainda não executado.

É preciso indicar à preparação da *query* os valores que queremos adicionar em nome, preco, quantidade e categoria\_id, por meio do bindValue, sendo que o primeiro será a chave que usamos no formato *string*, e o segundo será o valor que queremos inserir efetivamente no campo. Este processo faz a validação caso haja algo de *SQL Injection*, além de deixar o código mais organizado.

Para executar este código, usaremos o método execute(), que não é do objeto de conexão PDO, e sim do *PDO Statement* da variável stmt. Tecnicamente, isso trará o mesmo resultado que tínhamos antes, porém de forma mais legível e muito mais segura.

public function inserir()

{

$query = "INSERT INTO produtos (nome, preco, quantidade, categoria\_id)

VALUES (:nome, :preco, :quantidade, :categoria\_id)";

$conexao = Conexao::pegarConexao();

$stmt = $conexao->prepare($query);

$stmt->bindValue(':nome', $this->nome);

$stmt->bindValue(':preco', $this->preco);

$stmt->bindValue(':quantidade', $this->quantidade);

$stmt->bindValue(':categoria\_id', $this->categoria\_id);

$stmt->execute();

}

Vamos fazer um teste inserindo um novo produto à tabela: "Bermuda", que custará "25" reais, cuja quantidade será "3" e a categoria, "Roupas". Salvaremos e seremos redirecionados à listagem, que terá a nova categoria adicionada, como esperado. Aproveitando que aprendemos a fazer isso, vamos fazer o mesmo nos métodos que já temos e que fazem consulta ao banco de dados.

Não é estritamente necessário fazê-lo com todos eles, o ideal é usar a preparação de *queries* quando há **dados dinâmicos**. Ou seja, quando passamos algum *id* ou valor passado pelo usuário. Sendo assim, em Produto.php não há nada em que ele possa interferir, portanto não precisaremos realizar alterações.

Na classe Categoria.php, o método listar() não precisa deste tratamento pois seu valor é fixo, e em carregar() concatenamos o valor do id na nossa *query*. Mas de onde ele vem?

Na página de edição de categoria no navegador, é exibido o valor do id na barra de endereço. Logo, este é um bom lugar para alterarmos. Então, o código ficará da seguinte forma:

public function carregar()

{

$query = "SELECT id, nome FROM categorias WHERE id = :id";

$conexao = Conexao::pegarConexao();

$stmt = $conexao->prepare($query);

$stmt->bindValue(':id', $this->id);

$stmt->execute();

$lista = $stmt->fetchAll();

foreach ($lista as $linha) {

$this->nome = $linha['nome'];

}

}

Testaremos o código apertando a tecla "F5" no navegador, e veremos que tudo funciona corretamente. Uma dica para deixarmos o código ainda melhor é que neste caso sabemos que a *query* retornará apenas uma única linha, e criamos um foreach mesmo sabendo que ele passará por esse *loop* apenas uma vez.

Por que, mesmo assim, usamos o foreach neste caso?

Por conta do fetchAll(). E para não termos que fazer isso, em vez de lista, usaremos linha, e no lugar de fetchAll() usaremos simplesmente fetch(), que pegará a primeira linha. Daí, não precisaremos mais do foreach, e teremos:

public function carregar()

{

$query = "SELECT id, nome FROM categorias WHERE id = :id";

$conexao = Conexao::pegarConexao();

$stmt = $conexao->prepare($query);

$stmt->bindValue(':id', $this->id);

$stmt->execute();

$linha = $stmt->fetch();

$this->nome = $linha['nome'];

}

Vamos forçar algo de *SQL Injection* digitando na barra de endereço 127.0.0.1:8888/categorias-editar.php?id=2or1=1. Não obteremos nenhum resultado inesperado, porém o usuário também ficaria "sem saída". É bem difícil conseguirmos simular isso, mas existem programas que o fazem automaticamente, não importa se você tentar fazer manualmente e não conseguir fazer nenhuma vulnerabilidade. **É importante sempre usarmos o prepare() quando formos passar algum valor**.

Portanto, da mesma forma, voltaremos ao arquivo Categoria.php para alterar o método inserir(), em que temos um valor engessado:

public function inserir()

{

$query = "INSERT INTO categorias (nome) VALUES (:nome)";

$conexao = Conexao::pegarConexao();

$stmt = $conexao->prepare($query);

$stmt->bindValue(':nome', $this->nome);

$stmt->execute();

}

Para ver se isto funciona como gostaríamos, vamos adicionar uma nova categoria ao nosso sistema, "Brinquedos", que aparecerá na listagem com *id* 6. Depois, faremos o mesmo em atualizar():

public function atualizar()

{

$query = "UPDATE categorias set nome = :nome WHERE id = :id";

$conexao = Conexao::pegarConexao();

$stmt = $conexao->prepare($query);

$stmt->bindValue(':nome', $this->nome);

$stmt->bindValue(':id', $this->id);

$stmt->execute();

}

Para testarmos, editaremos a categoria recém criada "Brinquedos", alterando-a para "Brinquedo". Salvaremos, e tudo funciona bem. Continuando, excluir() é outro método que poderemos alterar:

public function excluir()

{

$query = "DELETE FROM categorias WHERE id = :id";

$conexao = Conexao::pegarConexao();

$stmt = $conexao->prepare($query);

$stmt->bindValue(':id', $this->id);

$stmt->execute();

}

Feito isso, conseguiremos excluir qualquer uma das categorias com sucesso!

Nosso código está bem mais organizado e seguro para se trabalhar com PDO. Lembrando que o uso do bindValue() para poder usar o prepare() e depois o execute() não é obrigatório.

No entanto, sempre que um parâmetro for passado, para usarmos o bindValue(), é preciso **fazer a preparação da *query***, e isto é muito importante.

Exercicios:

Nessa aula, utilizamos o método PDO::prepare() para preparar uma conexão.

Qual das formas abaixo, é a forma correta de se usar o PDO::prepare()?

Parte superior do formulário

* Alternativa correta



* + Criar um script SQL em uma string: $query = "SELECT ....."
  + Efetuar a conexão: $conexao = Conexao::pegarConexao();
  + Preparar a query em um Statement: $stmt = $conexao->prepare($query);
  + Executar o script com o método execute() da conexão: $conexao->execute();
* Alternativa correta



* + Criar um script SQL em uma string: $query = "SELECT ....."
  + Efetuar a conexão: $conexao = Conexao::pegarConexao();
  + Preparar a query em um Statement: $stmt = $conexao->prepare($query);
  + Executar o script com o método execute() do Statement: $stmt->execute();

Quando usamos o método prepare() do PDO, nos é retornado uma variavel do tipo PDOStatement. Mas, esse statement ainda não foi rodado no banco de dados e podemos trabalhar na query para passar parâmetros por exemplo. Alem disso, se fomos executar mais de uma vez a mesma query (mudando ou não os parâmetros), ela acaba sendo otimizada pelo banco de dados que já sabe a estrutura do seu script.

* Alternativa correta



* + Criar um script SQL em uma string: $query = "SELECT ....."
  + Efetuar a conexão: $conexao = Conexao::pegarConexao();
  + Usar o método query() que retorna um Statement: $stmt = $conexao->query($query);
  + Executar o script com o método execute() do Statement: $stmt->execute();

Parte inferior do formulário

**Melhorias**

Nessa aula, utilizamos o método PDO::prepare() para preparar uma conexão.

Qual das formas abaixo, é a forma correta de se usar o PDO::prepare()?

Parte superior do formulário

* Alternativa correta



* + Criar um script SQL em uma string: $query = "SELECT ....."
  + Efetuar a conexão: $conexao = Conexao::pegarConexao();
  + Preparar a query em um Statement: $stmt = $conexao->prepare($query);
  + Executar o script com o método execute() da conexão: $conexao->execute();
* Alternativa correta



* + Criar um script SQL em uma string: $query = "SELECT ....."
  + Efetuar a conexão: $conexao = Conexao::pegarConexao();
  + Preparar a query em um Statement: $stmt = $conexao->prepare($query);
  + Executar o script com o método execute() do Statement: $stmt->execute();

Quando usamos o método prepare() do PDO, nos é retornado uma variavel do tipo PDOStatement. Mas, esse statement ainda não foi rodado no banco de dados e podemos trabalhar na query para passar parâmetros por exemplo. Alem disso, se fomos executar mais de uma vez a mesma query (mudando ou não os parâmetros), ela acaba sendo otimizada pelo banco de dados que já sabe a estrutura do seu script.

* Alternativa correta



* + Criar um script SQL em uma string: $query = "SELECT ....."
  + Efetuar a conexão: $conexao = Conexao::pegarConexao();
  + Usar o método query() que retorna um Statement: $stmt = $conexao->query($query);
  + Executar o script com o método execute() do Statement: $stmt->execute();

Parte inferior do formulário

Criação de um Robo.php

Nessa aula, aprendemos a utilizar o PDO::prepare(), que é um dos pontos **mais importantes** do php-pdo. Com ele, podemos passar valores para nossos scripts SQL de forma dinâmica. Isso, alem de trazer muito mais segurança para nosso código, deixa as interações com o Banco de Dados mais eficientes.

Vamos elaborar agora, um simples script php que podemos usar, para criar registros em nossa base de dados. Esse tipo de script é bem útil, quando por exemplo, querermos criar *"massa de dados"* em nosso sistema, para fins de testes por exemplo.

Nosso script, vai ter como objetivo, criar registros de diversos produtos e para isso, vamos seguir os passos a seguir:

* Para deixar elegante nosso script, vamos gerar nomes aleatórios de roupas, então, antes de começar, certifique de que você tenha uma categoria **Roupas** cadastrado no seu banco de dados, e já pegue o **ID** dessa categoria. No meu caso, criei uma nova categoria Roupas que recebeu o ID **7**.
* Na raiz do diretório de nosso projeto, vamos criar um novo arquivo em branco chamado robo.php. Ele será responsável pelo nosso script. Não se esqueça de incluir nele o require\_once para nosso arquivo global.
* Para gerar o nome da roupa, vamos usar o script abaixo. Basicamente, criamos nele, 3 arrays. O primeiro, com tipos de roupas, o segundo, com o sexo a qual é indicado a roupa e o terceiro com a cor da roupa. E depois, usamos a função rand() do php para randomizar um número aleatório de 0 até o total de índices de cada array, assim, cada vez que rodamos esse script, temos um resultado diferente com uma combinação aleatória de cada opção do Array.

O Script ficou assim:

<?php

$tipo\_roupa = ['Blusa', 'Camisa', 'Camiseta', 'Bermuda', 'Calça', 'Jaqueta'];

$sexo\_roupa = ['Masculina', 'Feminina'];

$cor\_roupa = ['Preta', 'Vermelha', 'Azul', 'Amarela', 'Verde', 'Branca', 'Marrom', 'Rosa'];

$tipo\_index = rand(0, 5);

$sexo\_index = rand(0, 1);

$cor\_index = rand(0, 7);

$roupa = $tipo\_roupa[$tipo\_index] . ' ' . $sexo\_roupa[$sexo\_index] . ' ' . $cor\_roupa[$cor\_index];

Para testar, vamos só colocar ele em nosso arquivo robo.php. no final, coloque um echo $roupa; e acesse esse arquivo pelo navegador:

Teremos o resultado como: "Bermuda Feminina Azul" ou "Jaqueta Masculina Rosa" ou "Camiseta Feminina Vermelha" e assim por diante.

* Agora, vamos continuar, fazendo um **laço for** que vai terminar quantos registros vamos incluir cada vez que rodamos nosso arquivo robo.php.
* Depois, precisamos fazer nossa string com o INSERT do produto, pegar a conexão e passar como parâmetro do bindValue() no nome da roupa, o valor que geramos aleatoriamente no código acima.
* Para os outros campos da tabela Produtos, como **quantidade** e **preço**, você também pode usar a função rand(); para gerar números aleatórios ;)

**VER OPINIÃO DO INSTRUTOR**

**Opinião do instrutor**

A primeira coisa que precisamos fazer, é criar nosso arquivo robo.php na raiz do nosso projeto. Nele, vamos abrir um bloco de php com <?php e fazer o require\_once 'global.php';

Em seguida, vamos definir todos os valores gerais de nosso robô que são:

* Número de roupas que queremos criar a cada vez que o robô for executado
* Colocar uma variável para guardar o valor da categoria roupas, que é a categoria que vamos inserir registros.
* Colocar os 3 arrays que foi mostrado acima. Já com seus respectivos valores: $tipo\_roupa, $sexo\_roupa e $cor\_roupa.
* Vamos agora, criar um laço for() que vai ser rodado exatamente o número de vezes na qual, especificamos no segundo passo.
* Vamos criar todos os valores aleatórios que precisamos:
  + $tipo\_index = rand(0, 5);
  + $sexo\_index = rand(0, 1);
  + $cor\_index = rand(0, 7);
  + $preco = rand(1, 100);
  + $quantidade = rand(1, 50);
* Vamos criar a variável com nome de nossa roupa: $roupa = $tipo\_roupa[$tipo\_index] . ' ' . $sexo\_roupa[$sexo\_index] . ' ' . $cor\_roupa[$cor\_index];.
* Agora, vamos fazer a query de **INSERT** na tabela de produtos
  + Vamos criar a variável $conexao com o método Conexao::pegarConexao().
  + Vamos criar a variável $stmt que recebe o retorno da $conexao->prepare($query);.
  + Vamos passar todos os valores aleatórios que criamos com o método bindValue().
    - $stmt->bindValue(':nome', $roupa);
    - $stmt->bindValue(':preco', $preco);
    - $stmt->bindValue(':quantidade', $quantidade);
    - $stmt->bindValue(':categoria\_id', $categoria\_id);
  + Vamos executar nossa query com o método $stmt->execute();;
* Por fim, vamos aproveitar apenas para imprimir o nome da roupa que acabamos de cadastrar com um echo, ainda dentro do nosso laço for(): echo $roupa . ' Cadastrada com sucesso!<br>';

Nosso código vai ficar desse jeito:

<?php

require\_once 'global.php';

$numero\_de\_roupas = 10;

$categoria\_id = 7;

$tipo\_roupa = ['Blusa', 'Camisa', 'Camiseta', 'Bermuda', 'Calça', 'Jaqueta'];

$sexo\_roupa = ['Mascilona', 'Feminina'];

$cor\_roupa = ['Preta', 'Vermelha', 'Azul', 'Amarela', 'Verde', 'Branca', 'Marrom', 'Rosa'];

for ($x = 1; $x <= $numero\_de\_roupas; $x++) {

$tipo\_index = rand(0, 5);

$sexo\_index = rand(0, 1);

$cor\_index = rand(0, 7);

$preco = rand(1, 100);

$quantidade = rand(1, 50);

$roupa = $tipo\_roupa[$tipo\_index] . ' ' . $sexo\_roupa[$sexo\_index] . ' ' . $cor\_roupa[$cor\_index];

$query = "INSERT INTO produtos (nome, preco, quantidade, categoria\_id)

VALUES (:nome, :preco, :quantidade, :categoria\_id)";

$conexao = Conexao::pegarConexao();

$stmt = $conexao->prepare($query);

$stmt->bindValue(':nome', $roupa);

$stmt->bindValue(':preco', $preco);

$stmt->bindValue(':quantidade', $quantidade);

$stmt->bindValue(':categoria\_id', $categoria\_id);

$stmt->execute();

echo $roupa . ' Cadastrada com sucesso!<br>';

}

Quando executarmos nossa página no navegador, acessando a URL: http://127.0.0.1:8888/robo.php, teremos o seguinte resultado:

Calça Feminina Amarela Cadastrada com sucesso!

Bermuda Feminina Rosa Cadastrada com sucesso!

Camiseta Mascilona Branca Cadastrada com sucesso!

Jaqueta Mascilona Azul Cadastrada com sucesso!

Blusa Mascilona Azul Cadastrada com sucesso!

Camiseta Mascilona Branca Cadastrada com sucesso!

Camiseta Mascilona Amarela Cadastrada com sucesso!

Calça Feminina Verde Cadastrada com sucesso!

Bermuda Feminina Marrom Cadastrada com sucesso!

Jaqueta Feminina Branca Cadastrada com sucesso!

**OBS** - O resultado final pode variar para você pois os valores são aleatórios. O importante é que todos os valores tenham sido cadastrados com sucesso! ;)