PHP – PDO – Conexão com o BD

Categoria

No exercício referente a esta aula, vocês terão todos os arquivos utilizados - de criação do banco de dados, as instruções para isso, e os arquivos de front-end. Vamos começar criando uma pasta denominada "**classes**", em que incluiremos um novo arquivo chamado Categoria.php.

Basicamente, neste arquivo será criada uma classe para representar cada entidade, cujas necessidades estarão sob responsabilidade de cada uma delas em nosso sistema de estoque.

Começaremos com "Categoria", da seguinte maneira:

<?php

class Categoria

{

}

?>

Agora, vamos dar uma olhada em nosso front-end para entendermos o que precisaremos fazer nesta parte de categorias. Neste projeto, não usaremos Apache, IIS, ou qualquer outro servidor web. O PHP possui um servidor web embutido, que poderá ser utilizado para facilitar nosso trabalho.

Abriremos o Windows PowerShell, mas este pode ser substituído por qualquer outro terminal de sua preferência. Navegaremos até a pasta de nosso projeto, digitando os seguintes comandos intercalados com "Enter":

cd ..

cd ..

cd PDO

ls

O último comando nos lista todos os arquivos existentes no diretório para verficiarmos se estamos no local correto, ou seja, na raiz do nosso projeto. Para que o servidor web funcione, usaremos o comando php -S 127.0.0.1:8888 passando como parâmetros o IP e a porta para o qual queremos que o servidor suba, no caso, a mesma máquina que estamos utilizando, ou o IP local.

Com "Enter", o prompt fica bloqueado para uso, sendo que poderemos parar o processamento a qualquer momento com "Ctrl + C", e será impresso o log de tudo que está acontecendo no servidor web.

Vamos ver se deu certo?

Abriremos o navegador e digitaremos 127.0.0.1:8888, e nosso sistema é carregado conforme esperávamos:



Clicando na seção "Categorias", há um botão que serve para criarmos novas categorias, uma já existente ("Livros") sendo listada na tabela, um botão de "Editar" e outro de "Excluir". Entendemos que precisaremos fazer a lista de categorias, a ser apresentada ao usuário nesta página. Vamos voltar ao código de Categoria.php.

Precisaremos fazer uma busca no banco de dados para obtermos as categorias do nosso projeto, portanto usaremos uma variável $query, que será uma *string*, e nela usaremos o SELECT para a consulta no banco de dados.

Para executarmos a *query*, usaremos o PDO, com o qual precisaremos criar uma conexão a partir da variável $conexao, e fazendo uma nova instância da classe PDO passando o parâmetro que é a *string*de conexão, a única informação que variará de acordo com o banco de dados em uso.

Tecnicamente, esta é a única mudança no caso de trocarmos de banco de dados, desde que o novo siga a mesma estrutura de tabelas e não esteja utilizando nenhuma função específica.

No nosso caso, utilizaremos o MySQL, então a *string* de conexão será criada com base nele. Em seguida, informaremos o *host* onde se encontra o servidor MySQL, e o *dbname* do projeto.

O segundo parâmetro será o usuário de conexão, e o terceiro, a senha:

<?php

class Categoria

{

public function listar()

{

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

$conexao = new PDO('mysql:host=127.0.0.1;dbname=estoque', 'root', 'alura');

}

}

?>

Legal, desta forma fazemos a conexão com o banco de dados, e agora precisaremos justamente executar a *query*, e para isso usaremos o método query() do PDO, passando como parâmetro a *string* $query. Queremos que isso seja retornado para uma variável $resultado. O trecho do código ficará assim:

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

$conexao = new PDO('mysql:host=127.0.0.1;dbname=estoque', 'root', 'alura');

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

O método query() retorna um objeto chamado ***PDOStatement***, com várias informações da nossa execução, incluindo o retorno, as linhas retornadas. Para obtê-las, usaremos o método fetchAll(), que retornará um *array* com todas as linhas que queremos da consulta.

Colocaremos tudo isso em uma lista a ser retornada:

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

$conexao = new PDO('mysql:host=127.0.0.1;dbname=estoque', 'root', 'alura');

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

$lista = $resultado->fetchAll();

return $lista;

Vamos conferir seu funcionamento indo à página de Categorias e instanciando a nova classe para testarmos. Abriremos o arquivo categorias.php, em que consta puro HTML, exceto pelo require\_once do cabeçalho e do rodapé.

Começaremos incluindo outro require\_once para chamarmos a classe Categoria. Em seguida, abriremos um bloco de PHP para a chamada para o método que acabamos de criar. E queremos que esta informação vá para uma variável chamada lista:

<?php require\_once 'classes/Categoria.php' ?>

<?php

$categoria = new Categoria();

$lista = $categoria->listar();

?>

Para verificar se tudo funciona como esperado, voltaremos ao navegador para atualizá-la. Receberemos uma mensagem de erro, pois o PHP, por padrão, precisa que a extensão do PDO esteja ativada no servidor, o que poderemos fazer no arquivo de configuração do PHP, o php.ini.

Para abri-lo e habilitarmos sua extensão de PDO, acessaremos "C: > php > php.ini" (Windows), e abriremos o arquivo no próprio bloco de notas. Veremos que ele é cheio de configurações do PHP, e o que queremos é ativar a configuração do PDO.

Com "Ctrl + F", buscaremos "PDO", o que nos mostra diversas extensões, cada uma para um banco de dados diferente. No nosso caso, como estamos usando o MySQL, habilitaremos a extensão correspondente descomentando-a, isto é, retirando o ponto e vírgula (**;**) que a antecede, deixando-a assim:

extension=php\_pdo\_mysql.dll

Feito isso, salvaremos o arquivo e reiniciaremos o servidor que deixamos rodando no PowerShell, justamente para que se usem as configurações atualizadas.

Então, voltaremos ao PowerShell, em que usaremos "Ctrl + C" para pausar a execução. Em seguida, o executaremos novamente. Vamos ver se isto resolve nosso problema?

Sim! Agora ele está funcionando direitinho. No entanto, nossas categorias ainda não aparecem, então vamos verificar se realmente deu certo debugando a variável para ver se ela está sendo populada corretamente, por meio do método print\_r() do PHP. Em categorias.php, teremos:

<?php require\_once 'classes/Categoria.php' ?>

<?php

$categoria = new Categoria();

$lista = $categoria->listar();

print\_r($lista);

?>

O print\_r() é equivalente ao toString(), e tenta imprimir na tela qualquer tipo de variável, não importa se é um *array*, inteiro, ou booleano. Ele dará um jeito de convertê-la para uma *string* e imprimir.

Vamos executar o código e ver o que acontece. No navegador, é indicado que trata-se de um *array* e nos passa todas as informações... Mas está meio ruim de visualizar assim, não?

Para facilitar a visualização do retorno do print\_r(), utilizaremos o echo e a tag <pre> do HTML:

<?php require\_once 'classes/Categoria.php' ?>

<?php

$categoria = new Categoria();

$lista = $categoria->listar();

echo "<pre>";

print\_r($lista);

echo "</pre>";

?>

Isso faz com que se acertem as quebras de linha, ajudando na compreensão de que há quatro categorias com os respectivos *id*s e nomes. Para cada categoria, existe uma chave com o nome do campo de dados (o *id*), e uma chave para o mesmo valor, porém numérica (0), possibilitando que se use a que preferir.

Se forem usar o valor numérico, atentem-se para que não haja risco de mudar a ordem dos valores depois, pois esta informação irá variar ali.

Então, poderemos deletar o debug pois já sabemos que está tudo sendo rodado como esperado. Agora, vamos jogar estas informações à nossa lista com o foreach, navegando pelo *array* linha a linha e capturando as informações para mostrá-las na tela.

Antes da tag <tr> e após <tbody> em categorias.php, digitaremos:

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

<tr>

<!-- ... -->

</tr>

<?php endforeach ?>

Para melhorar a organização e legibilidade do nosso código quando misturamos o PHP com um arquivo HTML, costuma-se usar o padrão de escrita do PHP em uma mesma linha, usando dois pontos (**:**), como mostrado acima.

E uma vez que abrimos esta tag com dois pontos, ela será fechada com endforeach. Com isso, simplesmente imprimiremos os valores do id e nome da categoria:

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

<tr>

<td><a href="/categorias-detalhe.php" class="btn btn-link"><?php echo $linha['id'] ?></a><td>

<td><a href="/categorias-detalhe.php" class="btn btn-link"><?php echo $linha['nome'] ?></a><td>

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

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

</tr>

<?php endforeach ?>

Vamos rodar o código e verificar o que acontece.

Tudo está funcionando corretamente! Agora, temos o foreach com os nossos dados, ou seja, as categorias "Livros", "Jogos", "Filmes" e "Revistas".

Até agora, criamos a classe Categoria.php, com uma $query para fazermos o SELECT. Implementamos a primeira $conexao com o PDO, em que executamos a *query*, cujo resultado foi retornado para outra página.

Em categorias.php fizemos a instância da Categoria, a chamada no método listar(), a qual verificamos com o debug para nos averiguar sobre seu funcionamento, e fizemos um loop na tabela para imprimir todos os valores na tela!

**Aula 2**

Continuando nosso projeto, que já tem a listagem funcionando bem, agora precisaremos conseguir inserir categorias novas ao sistema. Na página aberta no navegador, vamos clicar no botão "Criar Nova Categoria" e ver como ele se comporta.

É um formulário bem simples contendo uma caixa de texto a ser preenchida com o nome da categoria que queremos adicionar ao sistema, e um botão de "Salvar".

Vamos criar um método em nossa classe Categoria.php para esta nova inserção:

public function listar()

{

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

$conexao = new PDO('mysql:host=127.0.0.1;dbname=estoque', 'root', 'alura');

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

$lista = $resultado->fetchAll();

return $lista;

}

public function inserir($nome)

{

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

}

Em seguida, precisaremos criar a conexão com o MySQL usando o PDO novamente, após o qual faremos a execução da query, só que desta vez não usaremos o método query(), e sim exec(), para usarmos em INSERT, no caso.

public function inserir($nome)

{

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

$conexao = new PDO('mysql:host=127.0.0.1;dbname=estoque', 'root', 'alura');

$conexao->exec($query);

}

Por enquanto, não precisaremos pegar seu retorno. Vamos testar para ver se funciona?

Abriremos categorias-criar.php, em que se encontra o formulário. A única coisa que teremos que fazer nele é indicar o name do input, a nossa caixa de texto. E precisaremos passar a action do formulário, indicando para qual página essas informações serão enviadas.

O código ficará da seguinte maneira:

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

<div class="row">

<div class="col-md-6 col-md-offset-3">

<div class="form-group">

<label for="nome">Nome da Categoria</lab.

.el>

<input name="nome" type="text" class="form-control" placeholder="Nome da Categoria">

</div>

<input type="submit" class="btn btn-success btn-block" value="Salvar">

</div>

</div>

</form>

Agora, criaremos o categorias-criar-post.php no projeto, abrindo o bloco PHP que não será fechado, isto é, não terá ?>, por conter apenas esta linguagem.

Precisaremos passar o nome da categoria, e para isso criaremos $nome, que pegará nome, lembrando que a chave da variável $\_POST equivale ao name do input no formulário HTML.

Após a inserção, redirecionaremos o usuário à tela de lista, por meio da função header().

<?php

require\_once 'classes/Categoria.php';

$categoria = new Categoria();

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

$categoria->inserir($nome);

header('Location: categorias.php');

Vamos testar para ver se funciona! No navegador, na página de criação de nova categoria, digitaremos "Eletrônicos" e clicaremos em "Salvar". Ela aparecerá na nossa lista corretamente. Da mesma forma, testaremos incluindo "Eletrodoméstico", o que funcionará bem também.

Então, criamos Categoria.php e o método inserir() passando a variável nome, criamos seu INSERT, fizemos a conexão com o banco de dados e o executamos.

Mas... Pensando que estamos tentando trabalhar com Orientação a Objetos, será que esta é a melhor forma de fazermos tudo isso? Por que não criar atributos para a nossa Categoria, justamente para deixar tudo mais claro e organizado?

Temos duas colunas em nossa tabela: uma para id e outra para nome, então poderemos criar dois atributos, um para cada coluna. E em vez de passarmos como parâmetro $nome em INSERT, usaremos o atributo recém criado.

Em Categoria.php, portanto, teremos:

class Categoria

{

public $id;

public $nome;

public function listar()

{

//...

}

public function inserir()

{

$query = "INSERT INTO categorias (nome) VALUES ('" . $this->nome . "')";

$conexao = new PDO('mysql:host=127.0.0.1;dbname=estoque', 'root', 'alura');

$conexao->exec($query);

}

}

A diferença é que, agora, antes de instanciarmos a classe Categoria e fazermos o INSERT, vamos popular o atributo com o valor de nome, que vem do POST. Sendo assim, categorias-criar-post.php ficará da seguinte forma:

$categoria = new Categoria();

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

$categoria->nome = $nome;

$categoria->inserir();

Para testarmos, criaremos a categoria "Quadrinhos", e veremos que ele foi adicionado à lista corretamente, e nosso código está bem mais organizado.

**Aula 3**

Agora que conseguimos inserir novas categorias, vamos dar uma olhada no código. Os métodos listar() e inserir() se diferenciam ao abrirmos a conexão com o PDO. Em listar(), usamos o método query() para a execução da query, e em inserir(), usamos exec().

Qual a diferença entre estes dois métodos?

Como falado anteriormente, basicamente, o método query() retorna um PHP Statement, que possui várias informações acerca do retorno da query, como o número de linhas, por exemplo, dados que podem acabar sendo úteis para nosso trabalho.

O método exec(), por sua vez, é mais puro e simples, e apenas executa, não trazendo resultados. Simplesmente sabemos se a execução foi bem sucedida ou não. E ele é usado em casos em que há inserts (inserções), updates (atualizações) ou deletes (exclusões).

Podemos reparar também que nestes dois métodos há uma informação duplicada, com a conexão sendo feita exatamente da mesma forma em ambos. Se continuarmos assim, teremos quatro, cinco, seis métodos contendo as mesmas informações.

Imaginando uma situação de troca de computador ou de termos que subir tudo isso para outro ambiente, cujas informações e dados são outros, cada método destes precisaria ser alterado, e isso não é legal.

Poderíamos, então, **centralizar estas informações em um único lugar**. Da mesma forma que criamos Categoria, responsável por tudo aquilo que se refere a categorias, é possível criar uma classe para as conexões, responsável pelo gerenciamento das mesmas.

Dentro da pasta "classes", então, criaremos um novo arquivo chamado Conexao.php, com a classe Conexao, que irá estabelecer a conexão e retorná-la para quem precisar. Por isso, implementaremos um método estático para este retorno:

<?php

class Conexao

{

public static function pegarConexao()

{

$conexao = new PDO('mysql:host=127.0.0.1;dbname=estoque', 'root', 'alura');

return $conexao;

}

}

Em seguida, voltaremos à classe Categoria.php e substituiremos o new PDO() tanto no método listar() quanto no inserir(), pelo método recém criado, usando dois pontos duplos (**::**), já que é um método estático:

public function listar()

{

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

$conexao = Conexao::pegarConexao();

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

$lista = $resultado->fetchAll();

return $lista;

}

public function inserir()

{

$query = "INSERT INTO categorias (nome) VALUES ('" . $this->nome . "')";

$conexao = Conexao::pegarConexao();

$conexao->exec($query);

}

Para verificarmos seu funcionamento, atualizaremos a lista no navegador novamente... Obtivemos um erro por esquecermos de chamar o arquivo da classe Conexao.php em Categoria.php.

Vamos corrigir digitando:

<?php

require\_once 'classes/Conexao.php';

class Categoria

{

//...

}

Dessa vez, ao atualizarmos a página no navegador, veremos a lista funcionando direitinho.

Neste vídeo, criamos a classe Conexao.php para centralizar as informações da nossa conexão e, agora, se por acaso alguma delas for alterada, teremos que mexer apenas neste arquivo, o que facilita muito o trabalho.

Os próximos passos consistem em trabalharmos com a edição das categorias existentes!

**Aula 4**

A partir de agora faremos a configuração da edição das nossas categorias implementando uma ação para quando o usuário clicar no botão "Editar": carregar um formulário preenchido cujas informações poderão ser modificadas e salvas posteriormente no banco de dados.

Para começar, entraremos na classe Categoria e criaremos um novo método para a atualização. O arquivo já possui os métodos listar() e inserir(), e incluiremos atualizar(), com estrutura similar ao que fizemos com os outros.

Criaremos uma query, que terá a instrução SQL que executaremos para a atualização, pegaremos a conexão e executaremos a query com o método exec(), como fizemos em inserir(). As únicas colunas que temos nas categorias são id e nome, sendo que o primeiro é um valor chave que nunca é alterado. Assim, o único dado que o usuário poderá editar é o nome.

Também criaremos a variável conexao que, como o nome indica, receberá a conexão da nossa classe, após o qual executaremos a query.

public function atualizar();

{

$query = "UPDATE categorias set nome = '" . this->nome . "' WHERE id = " . $this->id;

$conexao = Conexao::pegarConexao();

$conexao->exec($query);

}

A partir disso, precisaremos do formulário para a edição das informações. Vamos acessar categorias.php na raiz do nosso projeto, em que se encontra a lista, que já foi criada. Quando formos chamar o formulário de edição, diferentemente do formulário de inserção, que faz o acréscimo de um valor até então inexistente, estaremos mexendo com algo que já existe.

Então, precisaremos indicar a esta página o que será editado, e a melhor forma de fazê-lo é passando o id da categoria a ser editada.

Lembra-se do foreach, com que fizemos a lista contendo as colunas com id e nome, bem como o botão com "Editar" e outro com "Excluir"?

Poderemos aproveitar o id desta lista para passá-lo como parâmetro na url da página de edição, via query string. Em categorias.php já temos o arquivo que estamos chamando na linha do botão "Editar", isto é, categorias-editar.php. Então basta acrescentarmos um ponto de interrogação e o nome da variável, igual seu nome.

Como o valor é dinâmico e definido pelo PHP, colocaremos a tag correspondente, com echo, e imprimiremos linha com o id, fechando o PHP em seguida:

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

<tr>

<td><a href="/categorias-detalhe.php" class="btn btn-link"><?php echo $linha['id'] ?></a><td>

<td><a href="/categorias-detalhe.php" class="btn btn-link"><?php echo $linha['nome'] ?></a><td>

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

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

</tr>

<?php endforeach ?>

Vamos ver se isso funciona?

Atualizaremos a tela de listagem no navegador e passaremos o mouse sobre "Editar", e do lado inferior esquerdo é mostrado o link para o qual o botão aponta e, neste caso, o id apontado é 1, o que corresponde ao fato deste ser o primeiro botão.

Se fizermos o mesmo na linha seguinte, o id será 2, como esperado, e assim por diante. Ou seja, cada botão está passando como parâmetro o id de sua categoria.

Vamos testar clicando no botão referente à categoria "Livros". Aparentemente, tudo funcionando bem. Abriremos o arquivo categorias-editar.php para ver como está o código. No campo input do nome da categoria, o value está preenchido com Livros, isto é, este valor está "engessado" e não é real, tanto que se voltarmos à ferramenta e formos editar "Jogos", o placeholder continua como "Livros".

Então, precisaremos fazer com que este valor fique **dinâmico**, para que se apresente ao usuário o valor do nome real da categoria que foi clicada para alteração. Para isto, carregaremos nesta página o objeto categoria, que representa o id, mais uma vez.

Criaremos mais um método para a classe Categoria, o carregar(), similar ao método da listagem. No entanto, em vez de carregar uma lista com vários registros, carregaremos apenas um, com idespecífico.

A diferença, portanto, é que no SELECT teremos uma cláusula WHERE, e passaremos um id como parâmetro. Voltando então à Categoria.php, vamos criar um novo método logo abaixo de listar(), cuja query será bem similar, porém, agora adicionaremos WHERE para que se filtre pelo id, e usaremos o método query() para executar o script.

Não usaremos exec() pois ele não nos retorna o PHP Statement, como faz o query().

Ao fazermos uma execução de query(), o PDO não sabe quantas linhas serão retornadas, mesmo que nós saibamos. Sendo assim, ele criará um array com várias linhas, mesmo que saibamos que queremos apenas a primeira delas.

Então, neste caso criaremos a variável lista que receberá o resultado de fetchAll(), assim como em listar() mas, em vez de fazê-lo diretamente, ele será tratada no próprio método para que somente a primeira linha seja devolvida.

public function carregar()

{

$query = "SELECT id, nome FROM categorias WHERE id = " . $this->id;

$conexao = Conexao::pegarConexao();

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

$lista = $resultado->fetchAll();

foreach ($lista as $linha) {

return $linha;

}

}

Para verificarmos se está tudo certo, testaremos carregando uma categoria na página. Voltaremos ao arquivo categorias-editar.php e instanciaremos a classe Categoria e criaremos um bloco de PHP para instanciarmos a mesma classe.

Lembra que no método carregar() pegamos o valor de id? Precisaremos recuperá-lo neste momento! Acabamos de digitar a página de listagem passando o id como parâmetro de querystring, e precisaremos dele para popularmos o atributo id.

Então, invocaremos o método carregar() colocando seu resultado em uma variável resultado:

<?php require\_once 'classes/Categoria.php' ?>

<?php

$categoria = new Categoria();

$id = $\_GET['id'];

$categoria->id = $id;

$resultado = $categoria->carregar();

?>

Tecnicamente, esta variável receberá aquele array com a primeira linha do retorno da query. Então, pegaremos o valor de resultado e utilizaremos o echo com a chave do nome do campo, que se chama nome. Isto significa que, no value do input, onde está Livros, abriremos a tag <php>:

<form action="#" method="post">

<div class="row">

<div class="col-md-6 col-md-offset-3">

<div class="form-group">

<label for="nome">Nome da Categoria</label>

<input type="text" value="<?php echo $resultado['nome'] ?>" class="form-control" placeholder="Nome da Categoria">

</div>

<input type="submit" class="btn btn-success btn-block" value="Salvar">

</div>

</div>

</form>

Vamos ver se funciona? Atualizaremos a página no navegador e veremos que a categoria que queremos editar aparece como o value, como gostaríamos!

**Aula 5**

Legal, nossa edição está funcionando corretamente, pois o texto no botão em que clicamos e a categoria a ser editada são os mesmos. Mas será que não conseguimos deixar nosso código um pouco mais organizado?

Organizado no que diz respeito à economia de linhas de código, do processo que fizemos para carregar o valor do nome da categoria. Vamos dar uma olhada no código: em categorias-editar.php primeiro fazemos a instância da nossa Categoria, depois pegamos o valor do id da query string, do GET, e então passamos este valor do id ao atributo id da categoria, e chamamos o método carregar(), que nos retorna um resultado, um array com o nome da categoria.

Se estamos utilizando Orientação a Objetos, já temos atributos da categoria, o id, sendo utilizado, assim como o nome. Então, por que não o usamos para ser o nome da categoria, em vez de retornar no método carregar() um array com este nome?

Poderemos simplesmente chamar o método carregar(), o que de fato carregará o atributo nome da nossa categoria. Então, vamos abrir Categoria.php para mexermos no método carregar(), com o foreach, que por sua vez contém return $linha;.

A variável linha já possui o array com o valor que queremos, o nome da categoria, então, em vez de o retornarmos, passaremos este valor como atributo nome, e deletaremos a linha com return. O código ficará assim:

public function carregar()

{

//...

foreach ($lista as $linha) {

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

}

}

Agora, o importante é que não teremos mais um retorno, pois o método carregar() não retorna mais nada, já que o resultado é colocado no atributo categoria do objeto instanciado. Se não temos mais nenhum retorno, é preciso ajustarmos isto em categorias-editar.php, pois quando chamamos o método carregar(), estamos esperando o retorno, a ser colocado na variável resultado (agora inexistente).

O arquivo categorias-editar.php, portanto, ficará sem a variável resultado, e simplesmente chamando o método carregar(), da seguinte forma:

<?php require\_once 'classes/Categoria.php' ?>

<?php

$categoria = new Categoria();

$id = $\_GET['id'];

$categoria->id = $id;

$categoria->carregar();

?>

E quando ele for chamado, o atributo nome estará populado!

No echo, apagaremos onde antes imprimíamos o valor resultado com a chave nome, imprimindo-se o valor de categoria com a chave nome:

<form action="#" method="post">

<div class="row">

<div class="col-md-6 col-md-offset-3">

<div class="form-group">

<label for="nome">Nome da Categoria</label>

<input type="text" value="<?php echo $categoria->nome ?>" class="form-control" placeholder="Nome da Categoria">

</div>

<input type="submit" class="btn btn-success btn-block" value="Salvar">

</div>

</div>

</form>

Vamos ver se isto funciona?

Tudo parece funcionar bem quando clicamos no botão "Editar". O que mais poderá ser otimizado na classe Categoria, em relação ao carregamento de informações?

Novamente estamos instanciando a categoria, buscando o id e atribuindo-o à classe Categoria, fazendo o carregamento. Poderemos deixar este processo todo um pouco mais inteligente e, no momento em que instanciarmos a Categoria, já poderemos passar o id diretamente.

Assim, quando ela for instanciada com um valor, já entendemos que este objeto será carregado com o valor do id que está sendo passado. Ou seja, o método carregar() será chamado automaticamente, de forma que vamos conseguir economizar mais linhas de código.

Para isto, criaremos um construtor na classe Categoria, um método que é invocado sempre que a classe é instanciada. No PHP, todo construtor possui um padrão para ser instanciado.

Como dito anteriormente, queremos usar o construtor quando passamos algum valor a ele, porém seu método será chamado sempre que o objeto é instanciado, não importa se estamos ou não passando um valor a ele. E no caso, o valor que podemos ou não passar é o id.

Então, passaremos id na assinatura do construtor e daremos um valor default a ele, que será false. Isso porque caso nada seja passado, este será o valor assumido, e a validação poderá ser efetuada para sabermos se foi passado um valor, e então chamar o método carregar(), ou não.

Portanto, no método construct() teremos um if testando o valor id: se for true, ou seja, se houver algo positivo nele, ele entrará neste if. E uma vez que o id for populado, poderemos chamar carregar().

class Categoria

{

public $id;

public $nome;

public function \_\_construct($id = false)

{

if ($id) {

$this->id = $id;

$this->carregar();

}

}

Com isso, poderemos ajustar o que foi feito em categorias-editar.php para substituir aquilo que já consta no construtor que acabamos de criar. Vamos apagar as seguintes linhas deste arquivo:

$categoria->id = $id;

$categoria->carregar();

Porém, para que isso funcione, agora precisaremos passar o id no momento em que instanciamos a classe Categoria, e não depois, como estava acontecendo até então. Basta invertermos a ordem, deixando:

<?php

$id = $\_GET['id'];

$categoria = new Categoria($id);

?>

Feito isso, testaremos no navegador e veremos que tudo funciona como gostaríamos!

**Aula 6**

Nossa tela de edição carrega a categoria direitinho, no entanto, estamos trabalhando com a edição de categorias, que é basicamente a possibilidade de alterarmos seu nome. É para isto que serve a tela de "Editar Categoria".

Vamos testar modificando "Livros" para chamá-la de "Quadrinhos". Clicando no botão "Salvar", nada acontece, isso porque não estruturamos o comportamento desta página para que a edição seja feita. Então, em categorias-editar.php, há o formulário com o input passando o valor da categoria em seu value.

Como já temos uma categoria criada e, além do nome, o seu id (inclusive estamos instanciando a classe com ele), criaremos um campo novo no formulário, um input``hidden, um campo invisível que o usuário não vê, mas que consta em nosso HTML.

A informação que estiver em seu value também será processada. Portanto, teremos:

<div class="form-group">

<label for="nome">Nome da Categoria</label>

<input type="hidden" name="id" value="<?php echo $categoria->id ?>">

<input type="text" value="<?php echo $categoria->nome ?>" class="form-control" placeholder="Nome da Categoria">

</div>

Vamos ver se isso funciona?

No navegador clicaremos com o lado direito do mouse em "Exibir código fonte da página", e será exibido o código HTML processado da página. O input type="text" consta com valor Livros, enquanto input type="hidden" aparece com value 1, o que corresponde ao valor da nossa categoria. Isto indica que o formulário funciona corretamente!

Agora, precisaremos submeter estas informações ao servidor. Faremos isto da mesma maneira com que fizemos a inserção de categorias, passando um parâmetro usando o action do nosso form. No mesmo arquivo, substituiremos # por categorias-editar-post.php, deixando a linha assim:

<form action="categorias-editar-post.php" method="post">

Porém, este arquivo ainda não existe, então o criaremos na raiz do projeto. Para testarmos o fluxo de submissão do formulário, simplesmente deixaremos o arquivo com:

entrou

Em seguida, atualizaremos a página no navegador e vamos clicar no botão "Salvar". Seremos redirecionados a uma página em branco com o texto "entrou", o que indica que tudo funciona como esperado. Voltaremos ao arquivo categorias-editar-post.php, abriremos o bloco PHP e faremos o require\_once da classe Categoria.

Em categorias-editar.php, estamos passando as informações do formulário via method="post", criamos o input hidden com id e já temos o input com o nome da categoria, porém este não possui o atributo nome, sendo preciso criá-lo:

<input type="text" name="nome" value="<?php echo $categoria->nome ?>" class="form-control" placeholder="Nome da Categoria">

Como estamos submetendo o formulário via POST, resgataremos estas informações da mesma forma. Em categorias-editar-post.php, teremos:

<?php require\_once 'classes/Categoria.php' ?>

<?php

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

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

echo $id;

echo '<br>';

echo $nome;

Testaremos voltando ao navegador, editando a categoria "Filmes" para "Séries" e clicando em "Salvar". Na nova página, veremos:

3

Séries

Em que 3 é o id.

Então, apagaremos os echo que colocamos apenas para validação do código, e instanciaremos a classe Categoria, passando o id; toda aquela estrutura foi criada para que quando passarmos um valor de id no construtor da classe, esta automaticamente carregue o valor que queremos.

A partir deste momento, a categoria virá previamente com o nome original populado. No entanto, queremos mudá-lo para um novo, então chamaremos o método atualizar(), que criamos em Categoria.php.

E como a página não apresentará nada ao usuário, ele será redirecionado à lista novamente. Assim, ele terá um feedback visual de que a categoria editada foi de fato alterada.

O código de categorias-editar-post.php, portanto, ficará da seguinte maneira:

<?php require\_once 'classes/Categoria.php' ?>

<?php

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

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

$categoria = new Categoria($id);

$categoria->nome = $nome;

$categoria->atualizar();

header('Location: categorias.php');

Testando no navegador, voltaremos à página com todas as categorias, dentre as quais editaremos "Revistas", que alteraremos para "Quadrinhos", e clicaremos em "Salvar". Como esperávamos, somos redirecionados à lista de novo, com o nome da categoria alterado, mesmo tendo o mesmo id de antes!

**Aula 7**

Estamos conseguindo editar nossas categorias! A partir de agora, vamos focar no outro botão que por ora não faz nada ainda ("Excluir"), para quando o usuário quiser deletar categorias.

Para criarmos o método de exclusão, editaremos novamente a lista de categorias, com o foreach. Da mesma forma que quando passamos o mouse sobre o botão "Editar" é mostrado o id da categoria, faremos o mesmo para "Excluir".

Passaremos ao botão o id da categoria que queremos excluir. No entanto, não teremos uma segunda tela de interação com o usuário, pois trata-se de um processo mais simples, em comparação com "Editar".

Vamos abrir o arquivo categorias.php, que contém o foreach, com cada linha das nossas categorias: uma com id, outra com id, com o botão de editar, e a última com o botão para exclusão, em que substituiremos # por /categorias-excluir-post.php, deixando o código assim:

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

Feito isso, vamos atualizar o navegador! Desta vez, ao passarmos o mouse sobre o botão "Excluir", teremos o mesmo efeito visto em "Editar". Então, criaremos categorias-excluir-post.php, em que inicialmente faremos o require\_once da classe com que iremos trabalhar para podermos instanciá-la, assim como fizemos nos demais arquivos.

Neste caso, estamos recebendo o valor do id da categoria via query string, portanto usaremos a variável GET para pegá-lo. Por fim, utilizaremos o echo para verificar seu funcionamento:

<?php require\_once "classes/Categoria.php" ?>

<?php

$id = $\_GET['id'];

echo $id;

Feito isso, vamos testar excluindo a categoria "Jogos" da listagem no navegador. Isto fará com que sejamos redirecionados à página contendo 2, que é a impressão do id da categoria excluída.

Agora, voltaremos à classe Categoria para criarmos o método referente a esta ação. Assim como fizemos anteriormente com os outros métodos, criaremos uma string com a instrução SQL para o delete, instanciando a conexão e executando a query.

Então, adicionaremos ao arquivo:

public function excluir()

{

$query = "DELETE FROM categorias WHERE id = " . $this->id;

$conexao = Conexao::pegarConexao();

$conexao->exec($query);

}

Para testarmos, voltaremos ao arquivo que acabamos de criar (categorias-excluir-post.php), que deixaremos assim:

<?php require\_once "classes/Categoria.php" ?>

<?php

$id = $\_GET['id'];

$categoria = new Categoria($id);

$categoria->excluir();

header('Location: categorias.php');

Vamos ver se funciona voltando ao navegador e excluindo a categoria "Quadrinhos" e depois "Eletrodomésticos", o que conseguiremos fazer com sucesso!

**Aula 8**

Praticamente finalizamos as categorias; já conseguimos incluir novas, editá-las e excluí-las. Lembra quando refatoramos a parte de conexão? Criamos uma classe Conexao exclusiva para isto, com o método pegarConexao() para todas as informações.

Criamos uma conexao com PDO passando os valores do MySQL, como host name, o nome do database, usuário e senha. Isso já nos ajuda bastante, por **centralizarmos todas estas informações em um único arquivo**. Se isso porventura vier a mudar, como ambiente, ou no caso de uma instalação nova da aplicação, não será preciso fazer a alteração em vários lugares, e sim apenas em um.

No entanto, estamos na classe Conexao, em que deixamos informações acerca da conexão, como indica o nome. Então, para deixar tudo mais organizado ainda, poderemos criar um **arquivo de configuração** em nosso projeto, o qual não será uma classe, e sim simplesmente um arquivo com todas as informações de configuração do nosso ambiente, tais como banco de dados e afins.

Portanto, criaremos na raiz do projeto um arquivo denominado config.php, usando o define do PHP, que cria um objeto, com um valor fixo. Isso difere da variável, cujo valor pode ser modificado.

Por conveniência, no PHP, sempre que fazemos um define, ele terá o nome em maiúsculo, então, neste caso, o primeiro deles será o driver de conexão (DB\_DRIVE), e depois colocaremos o valor que queremos atribuir a ele (mysql). Faremos isso com as demais informações:

<?php

define('DB\_DRIVE', 'mysql');

define('DB\_HOSTNAME', '127.0.0.1');

define('DB\_DATABASE', 'estoque');

define('DB\_USERNAME', 'root');

define('DB\_PASSWORD', 'alura');

Precisaremos importar este arquivo de configuração no arquivo Conexao.php por meio do require\_once, e com isto teremos os valores instanciados em nosso arquivo!

Então, faremos as substituições nas strings de conexão do PDO, cujos valores fixos colocados anteriormente trocaremos por estes que acabamos de definir. O código ficará assim:

<?php

require\_once 'config.php';

class Conexao

{

public static function pegarConexao()

{

$conexao = new PDO(DB\_DRIVE .':host=' . DB\_HOSTNAME . ';dbname=' . DB\_DATABASE, DB\_USERNAME, DB\_PASSWORD);

return $conexao;

}

}

Vamos confirmar se tudo funciona da mesma forma como estava antes?

A lista está sendo carregada quando atualizamos a página no navegador, indicando que tudo funciona como esperado, havendo conexão com o banco de dados.

E se quisermos incluir mais parâmetros futuramente, simplesmente aumentaremos a lista de definições em config.php.

**Aula 9**

Antes de continuarmos, vamos melhorar um pouco o que foi feito até agora - um grande problema em nosso código após a criação do arquivo de configuração é que fazemos muitos requests de arquivos em nosso sistema.

Todos os arquivos estão chamando alguma classe com require\_once, o que torna o código um tanto bagunçado. Em um sistema pequeno como este, até que não é um problema tão grave, mas continua sendo um problema e, pensando que um sistema pode crescer exponencialmente, isto tende a gerar muita confusão.

No PHP, há um jeito de resolver este problema tranquilamente, retirando-se todos os require\_once, exceto por um único, que cuidará de tudo de forma inteligente. Isto é possível com o **autoloader**.

Sempre que uma classe é requisitada, instanciada, e ela não existir no contexto do arquivo em que se encontra, o PHP chamará uma função, que verificará se ela existe de fato e, caso positivo, o require\_once deste arquivo será feito.

Para isso, criaremos o arquivo global.php fora da nossa estrutura de classes, na raiz de nosso projeto. Este será o arquivo com a função global, para o qual todos os outros que não são classes farão uma requisição, e que fará a busca pela classe certa na hora certa.

Criaremos uma função com o nome de sua escolha, neste caso será carregarClasse(), com parâmetro nomeClasse. Sempre que a função for chamada, o valor que virá como parâmetro será o nome da classe que está sendo instanciada, isto é, aquele que o código está tentando instanciar.

Já seguimos um padrão em que o nome das classes é o mesmo de seus arquivos, como ocorre na classe Categoria, cujo arquivo se denomina Categoria.php. Portanto iremos aproveitá-la, juntamente com a função file\_exists(), sendo que o parâmetro a ser utilizado será o caminho do arquivo em questão.

Sabemos que todos os arquivos de classes estão contidos na pasta "classes", então passaremos o caminho completo do arquivo, lembrando que temos a extensão .php, que não existe na variável nomeClasse.

Mas quem se responsabilizará pela chamada desta função, no momento da instância de um objeto?

Trata-se de uma função do ṔHP chamada spl\_autoload\_register, e passaremos como parâmetro o nome do método de autoload, carregarClasse.

Para organizar ainda mais nosso código, o arquivo de configuração que criamos atualmente só está sendo usado na conexão, mas como é um arquivo de configuração global criado para o sistema, é importante que ele esteja em todas as páginas. E por não ser uma classe, o autoload que criamos não conseguirá carregá-lo.

Neste caso, será necessário um require exclusivo para ele, dentro deste arquivo. Ou seja, todos os arquivos chamarão o arquivo global, e este, por sua vez, chamará o que for necessário, e como sabemos que todas as páginas serão necessárias no arquivo de configuração, teremos que deixar o código assim:

<?php

require\_once 'classes/config.php';

spl\_autoload\_register('carregarClasse');

function carregarClasse($nomeClasse)

{

if (file\_exists('classes/' . $nomeClasse . '.php')) {

require\_once 'classes/' .$nomeClasse . '.php';

}

}

Vamos testar para ver se isso funciona, acessando inicialmente Conexao.php, deletando o require\_once feito ali e salvando. Faremos o mesmo em Categoria.php e em categorias.php, de listagem de categorias. Nele, há um require\_once chamando Categoria.php, que não é mais necessário, portanto, usaremos:

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

Agora sim, voltaremos ao navegador e atualizaremos a página de categorias. Testaremos a edição de uma delas, que resultará em erro por não termos colocado o arquivo global nos outros arquivos.

A partir de agora, então, vamos refatorar tudo que foi feito substituindo por <?php require\_once 'global.php' ?> quando houver <?php require\_once 'classes/Categoria.php' ?>, isto é, em categorias-criar-post.php, categorias-editar.php, categorias-editar-post.php, categorias-excluir-post.php.

Vamos conferir estas alterações no navegador, editando a categoria "Jogos", por exemplo, e criando uma categoria nova, "Roupas".

Assim, o autoload deixa tudo muito mais organizado, permitindo que façamos o require\_once do arquivo global, o qual cuidará de tudo se adicionarmos novas classes, desde que sigamos o mesmo padrão de estarem contidas no diretório correto e com o mesmo nome do arquivo que elas representam.

**Aula 10**

O código está mais organizado, mas ainda não há nenhum tipo de tratamento de erros nele, e isso é muito importante para um sistema, não importa seu tamanho, pois além de facilitar a identificação deles, com o tratamento de erros poderemos determinar caminhos para solucioná-los.

É interessante trabalharmos com duas formas para o tratamento de erros: uma para produção e outra para não produção. A diferença entre elas é a apresentação do erro ao usuário, porque quando estamos desenvolvendo, queremos facilidade na identificação do erro, mas não poderemos deixar que ele veja detalhes sobre o mesmo. Ele não saberá o que fazer com esta informação.

Vamos começar pelo arquivo de configuração (config.php), em que criaremos uma variável nova para diferenciarmos estes ambientes. Lembrem-se de que este arquivo existe justamente para as configurações de cada ambiente, inclusive deste.

Criaremos uma nova define denominada DEBUG, colocando como seu valor padrão true, isto é, o modo DEBUG estará ligado. Quando este sistema estiver em produção apenas alteraremos o valor para false.

<?php

define('DEBUG', true);

Feito isso, no arquivo categorias.php, em que fazemos a instância da classe Categoria e depois a listagem chamando o método listar(), criaremos o try.

Se ocorrer algum erro no código e nas funções dentro do try, ele cairá no catch()automaticamente, o qual recebe uma variável do tipo Exception, e, que é uma classe de erro do PHP. E quando ele chegar ao bloco do catch() estará todo carregado com informações úteis acerca do erro, como mensagem, código e afins.

Então, poderemos elaborar a forma como trataremos esta informação. No catch(), apresentaremos a mensagem de erro usando o echo:

<?php

try {

$categoria = new Categoria();

$lista = $categoria->listar();

} catch(Exception $e) {

echo $e->getMessage();

}

?>

Que mensagem será exibida? Vamos testar para ver se tudo funciona direitinho?

Vamos pegar o método listar() para forçar um erro nele, gerando uma exceção:

public function listar()

{

throw new Exception('Erro ao Listar Categorias')

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

$conexao = Conexao::pegarConexao();

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

$lista = $resultado->fetchAll();

return $lista;

}

Voltaremos ao navegador, testaremos, e receberemos a mensagem de erro como gostaríamos. No entanto, vários outros erros também apareceram, pois sempre que geramos um throw, tudo que vem abaixo dele não é executado, o processamento é pausado e acessa-se o primeiro catch() que é encontrado, no caso, no arquivo categorias.php.

Não sabemos exatamente quando irá acontecer uma exceção, então iremos incluir o exit, já que após o catch() ele continuará processando a página, e não queremos que isto aconteça. O código ficará assim:

<?php

try {

$categoria = new Categoria();

$lista = $categoria->listar();

} catch(Exception $e) {

echo $e->getMessage();

exit;

}

?>

Desta vez, ao recarregarmos a página no navegador, apenas obteremos a mensagem "Erro ao Listar Categorias".

No entanto, para o desenvolvedor, este erro não é muito bem apresentado desta forma. Queremos maiores informações para podermos tratá-lo adequadamente.

Lembram-se do print\_r, usado para imprimir diversas informações de uma variável? Vamos usá-lo, pois ele cabe perfeitamente neste caso:

<?php

try {

$categoria = new Categoria();

$lista = $categoria->listar();

} catch(Exception $e) {

echo '<pre>';

print\_r($e);

echo '</pre>';

exit;

}

?>

Retornando ao navegador e atualizando a página, teremos maiores informações, como a mensagem exibida, seu código, o arquivo que gerou o erro, a linha onde ele se encontra, informações sobre o arquivo que o chamou, ou seja, todo o caminho que foi traçado.

Deste modo, tudo fica bem mais claro para o desenvolvedor, mas não queremos que o usuário final tenha estas informações. E é por isto que usaremos o parâmetro DEBUG que criamos anteriormente.

<?php

try {

$categoria = new Categoria();

$lista = $categoria->listar();

} catch(Exception $e) {

if (DEBUG) {

echo '<pre>';

print\_r($e);

echo '</pre>';

} else {

echo $e->getMessage();

}

exit;

}

?>

Se atualizarmos o navegador, nada muda. Acessaremos o arquivo config.php, alteraremos true de DEBUG para false, o que fará com que apenas a mensagem seja exibida, tornando o trabalho mais organizado.

**Aula 11**

Nosso tratamento de erro está bem mais interessante: apresentamos o erro ao usuário com o modo DEBUG ativo ou não. Faremos o mesmo em categorias-editar.php:

<?php

try {

$id = $\_GET['id'];

$categoria = new Categoria($id);

} catch (Exception $e) {

}

?>

Para o tratamento, teríamos que reescrever o código assim como fizemos anteriormente, o que se tornaria complicado se quiséssemos mudar a forma como apresentamos a mensagem de erro, pois teríamos que mudar todos os arquivos, e não é nada prático ter o mesmo trecho de código em vários locais distintos do projeto.

Vamos, portanto, criar uma classe específica para o tratamento de erro, Erro.php, lembrando que os arquivos de classes seguem o padrão de terem o mesmo nome do arquivo correspondente.

O método estático trataErro() receberá uma Exception $e assim como o catch(), e é nele que iremos fazer o tratamento da classe, cujo if do DEBUG poderá ser copiado e colado de categorias.php:

<?php

class Erro

{

public static function trataErro(Exception $e)

{

if (DEBUG) {

echo '<pre>';

print\_r($e);

echo '</pre>';

} else {

echo $e->getMessage();

}

exit;

}

}

E no catch(), iremos simplesmente chamar o método. Assim, em categorias.php, teremos:

<?php

try {

$categoria = new Categoria();

$lista = $categoria->listar();

} catch(Exception $e) {

Erro::trataErro($e);

}

?>

Vamos ver se isso funciona? A nossa lista deverá ser executada da mesma forma que antes. Em seguida, também acrescentaremos a linha Erro::trataErro($e); no arquivo categorias-editar.php, assim:

<?php

try {

$id = $\_GET['id'];

$categoria = new Categoria($id);

} catch (Exception $e) {

Erro::trataErro($e);

}

?>

Feito isso, poderemos voltar à página do navegador e testar a edição de uma categoria, e depois deletar o throw da exceção da função listar() em Categoria.php, para que a lista seja carregada normalmente. E incluiremos a linha abaixo à função carregar():

throw new Exception('Erro ao Carregar a Categoria');

Atualizando-se a página, obteremos a mensagem de erro. Agora, vamos ligar o modo DEBUG, colocando true como seu valor em config.php, após o qual recarregaremos a página no navegador e teremos todos os detalhes do erro.

Neste caso, saberemos que o erro estava em categorias-editar.php, passando para a classe Categoria, terminando ali, sendo que o erro se deu na linha 28, e a mensagem de erro é exatamente aquela que definimos.

Vamos deletar a linha com a exceção de carregar() e aplicar o Try/Catch nos demais locais necessários, basicamente todos aqueles que não são classe, mas que estão sendo trabalhadas: categorias-excluir-post.php, categorias-editar-post.php, categorias-criar-post.php.

Daqui em diante, todos os nossos arquivos irão ter o tratamento de erros, com tudo funcionando direitinho.

Aula 12

Tendo finalizado o tratamento de erros do nosso projeto, o que será que acontecerá se houver um erro no banco de dados? Para testar, vamos simplesmente deletá-lo!

No PowerShell, abriremos uma conexão com o MySQL a partir do comando mysql -u root -p, ao que será solicitada a senha, que neste caso é alura e, com "Enter", nos conectamos ao banco de dados. O nome do nosso *database* é estoque, e para termos certeza disto, poderemos usar o comando show databases;, que mostrará todos os bancos de dados criados, dentre os quais o que queremos é o próprio estoque.

Com drop database estoque; "matamos" o banco de dados. Em seguida, abriremos outro PowerShell para subirmos o servidor do PHP a partir do comando php - S 127.0.0.1:8888.

Voltaremos ao Chrome, e ao atualizarmos a página veremos que ocorre uma exceção, como gostaríamos. Vamos, então, deixar o modo DEBUG como false em config.php para verificar o que acontece?

Atualizando a página no navegador, obteremos a mensagem SQLSTATE[HY000][1049]Unknown database estoque, que indica um erro do próprio SQL, isto é, do PDO. Vamos recriar o *database* estoque para ver se o erro persiste, por meio do comando create database estoque; no PowerShell.

Usaremos o show databases; para confirmar a criação do banco, e notaremos que não criamos tabelas ainda. Com use estoque; selecionaremos o banco, e em seguida usaremos show tables;, e veremos que não há nenhuma tabela. Ao atualizarmos a página no navegador, teremos um erro fatal.

O *Try/Catch* não pega erros fatais, os quais ocorrem por não termos configurado a conexão com o PDO para informar como queremos considerá-los. O PDO é interessante porque permite setarmos atributos, como a forma como ele irá tratar erros, de forma silenciosa, por padrão - que é mostrar um erro fatal como neste caso -, ou até gerar uma Exception.

Em relação às opções que temos, na [página de documentação do PHP](http://php.net/manual/pt_BR/pdo.setattribute.php) há o método PDO::setAttribute, dentro do qual se pode passar 'n' configurações distintas, entre elas a forma como o PHP irá tratar os erros da conexão PDO.

É legal vocês darem uma olhada nesta página para depois conferir os outros atributos que podem ser utilizados, pois eles podem não ser úteis para este caso específico, mas sim para outra situação.

Para o nosso caso, utilizaremos PDO::ATTR\_ERRMODE, que contém três opções de erro:

* PDO::ERRMODE\_SILENT
* PDO::ERRMODE\_WARNING
* PDO::ERRMODE\_EXCEPTION

Cujo primeiro não faz nada além de setar um código de erro, o segundo faz um aviso no código (porém, se o PHP estiver configurado para não exibir avisos, nada acontece também), e o terceiro gera um throw de Exception, que é o que queremos, já que estamos tratando tudo com *Try/Catch*.

Então, usaremos este método logo após abrirmos nossa conexão, na classe Conexao. Passaremos dois parâmetros no método: a chave que queremos setar e seu valor. Mas quais são estes? Eles são definidos pela própria classe PDO com atributos estáticos que ainda desconhecemos, pois precisaremos consultar a documentação novamente.

O código ficará assim:

<?php

class Conexao

{

public static function pegarConexao()

{

$conexao = new PDO(DB\_DRIVE .':host=' . DB\_HOSTNAME . ';dbname=' . DB\_DATABASE, DB\_USERNAME, DB\_PASSWORD);

$conexao->setAttribute(PDO::ATTR\_ERRMODE, PDO::ERRMODE\_EXCEPTION);

return $conexao;

}

}

Feito isso, atualizaremos a página no navegador e obteremos uma exceção, da forma como tínhamos definido. No arquivo de configuração (config.php), mudaremos o false do DEBUG para true de novo. Todos os detalhes dos erros são exibidos, vindos do PDO como esperado.

Testando com o PDO::ERRMODE\_SILENT no lugar de PDO::ERRMODE\_EXCEPTION para vermos o que acontece, mantém-se o erro fatal de antes, que é o padrão. Vamos experimentar o PDO::ERRMODE\_WARNING. O aviso de erro é exibido corretamente, porém depois ocorre um erro fatal, já que o que vem depois dele não é processado.

Para o nosso caso, o PDO::ERRMODE\_EXCEPTION é a melhor opção, pois a Exception pausa a transação em um ponto específico. Agora, só para não esquecermos, vamos rodar o arquivo de base com a criação das tabelas (banco\_de\_dados.sql) e os *inserts* padrões, para não perdermos tudo que foi feito.

Voltaremos com o banco como estava antes dando um "Ctrl + C" em todo código referente à criação da tabela, e voltando ao PowerShell para rodarmos tudo. Atualizaremos a página do navegador, e ela estará da mesma forma que antes, com tudo funcionando direitinho.