**MVC e conexão com banco de dados**

**MVC e conexão com banco de dados**

**Model-View-Controller**

Repare novamente na forma como fizemos para registrar nossa primeira lógica:

Route::get('/', function()

{

return '<h1>Primeira lógica com Laravel</h1>';

});

O problema de definir as rotas dessa forma, com o código de resposta implementado diretamente em uma **função anônima**, é que não estamos seguindo nem um pouco as boas práticas da orientação a objetos. O código fica todo espalhado, difícil de manter e reutilizar.

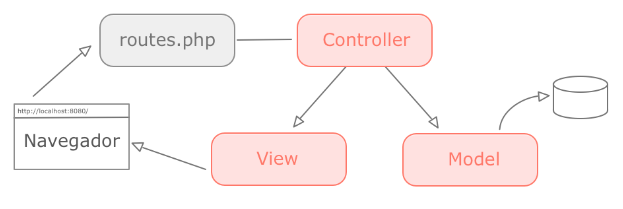
Conforme nossa aplicação for crescendo, vamos precisar fazer acesso ao banco de dados, executar lógicas com nossas regras de negócio etc. Tudo isso ficará dentro do mesmo arquivo route.php? Seria uma bagunça! Além disso, vamos misturar nossa lógica de negócio com nossa lógica de apresentação, o HTML de resposta.

Precisamos de uma forma melhor de dividir as responsabilidades de nossa aplicação. É aí que entra o **MVC**, ou**M**odel **V**iew **C**ontroller.

A grande ideia desse padrão arquitetural é que você separe suas regras de negócio em 3 camadas, cada uma com sua responsabilidade muito bem definida:

* **Model** é a camada onde ficam nossas regras de negócio, nossas entidades e classes de acesso ao banco de dados.
* **View** é a responsável por apresentar as páginas e outros tipos de resultado para o usuário (ou mesmo para outros sistemas, que se comunicam). É a resposta que o framework envia para o navegador, que normalmente é um HTML.
* **Controller** é quem cuida de receber as requisições web e decidir o que fazer com elas. Nessa camada definimos quais **model**os devem ser executados para determinada ação e para qual **view** vamos encaminhar a resposta. Em outras palavras, essa camada quem faz o link entre todas as outras.

Diversos frameworks, das mais diferentes linguagens, seguem esse padrão do MVC. Com Laravel não é diferente, o fluxo fica assim:



Repare que, quando nosso cliente envia uma requisição pelo navegador, primeiramente temos um arquivo PHP, que é o routes.php, que está frente de todos. Ele cuida de atender as requisições e enviá-las para o local correto, no caso os nossos controllers. Os controllers, por sua vez, decidem o que fazer com as requisições, passando pela camada de model (que fazem acesso ao banco, executam as regras de negócio etc.) e logo em seguida delegam pra ::view:: que será exibida como resposta no navegador do cliente.

Agora que já sabemos um pouco da teoria, vamos colocar o MVC em prática?

**Colocando MVC em prática**

No lugar de definir todas as lógicas do nosso sistema nesse arquivo único, o routes.php, vamos organizá-las desde o início em ::Controller::s distintos. Nosso sistema de estoques vai ter uma página principal, com a listagem de produtos. Podemos começar por ela.

Vamos criar um novo arquivo chamado ProdutoController, dentro da pasta app/Http/Controllers, que é o diretório padrão para esse tipo de classe. Dentro do controller, crie um método chamado lista. O arquivoProdutoController.php ficará assim:

<?php namespace estoque\Http\Controllers;

class ProdutoController {

public function lista(){

// nosso código vai aqui

}

}

Note que, como estamos trabalhando com uma estrutura de diretórios, tivemos que definir o ::namespace:: no cabeçalho do arquivo. Outra regra importante é que todo controller do Laravel deve herdar de uma classe chamada Controller, que foi criada automaticamente junto com nosso projeto. Para isso, só precisamos adicionarextends Controller na declaração da nossa classe, que ficará assim:

<?php

class ProdutoController extends Controller {

public function lista(){

// nosso código vai aqui

}

}

A classe Controller existe exclusivamente para que seja possível definirmos lógicas em comum que podem ser compartilhadas entre todos os controller de nossa aplicação. Além disso, ela já traz alguns ::imports:: (uses) essenciais para os controllers, que serão detalhados conforme formos precisando.

Inicialmente, vamos fazer o método lista retornar um HTML puro, com o cabeçalho da listagem de produtos. O código ficará assim:

<?php

class ProdutoController extends Controller {

public function lista(){

return '<h1>Listagem de produtos com Laravel</h1>';

}

}

E agora que temos esse comportamento definido no controller, vamos criar uma rota para ele no arquivoroutes.php, que continua responsável pelo registro de nossas rotas e outras configurações, mas a diferença é que agora ele apenas apontará para o método do controller que deve ser executado.

O arquivo de rotas deve ficar assim:

<?php

Route::get('/produtos', 'ProdutoController@lista');

// demais rotas omitidas

Note que o padrão é o nome do controller, seguido de um @ e o nome do método. Agora quando uma requisição for feita para a URL /produtos, o método lista do ProdutoController será executado. Bem simples, não acha?

Vamos testar? Basta acessar <http://localhost:8000/produtos> em seu navegador.



Sucesso, o HTML de resposta foi exibido conforme esperado.

**Trabalhando com banco de dados**

A listagem ainda está muito simples, ainda estamos mostrando apenas um texto onde queremos mostrar todos os produtos. Nosso objetivo agora será buscar essas informações de um banco de dados, como é natural nas aplicações do dia a dia.

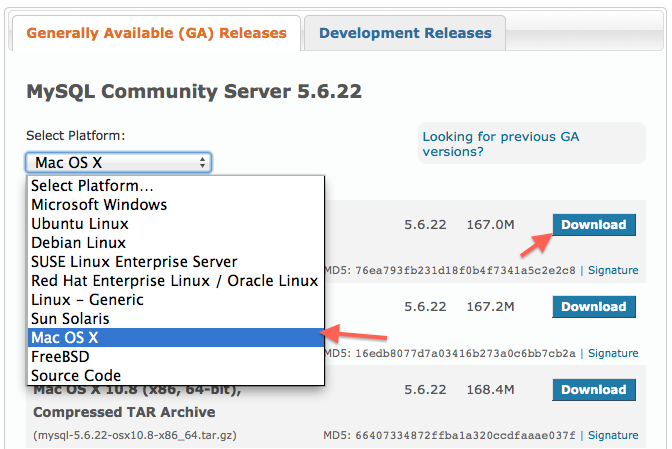
Iniciando com MySQL

Usaremos o banco de dados MySQL em nosso projeto, já que é um dos bancos de dados relacionais mais utilizados no mercado, é gratuito e bastante simples de instalar. Se você ainda não tem o MySQL instalado, pode fazer download pelo link:

<http://dev.mysql.com/downloads/mysql/>



Ao final da página, você encontrará um campo para escolher seu sistema operacional. Selecione e depois clique em download, na opção que preferir:



No mesmo site, você encontra um tutorial de instalação de acordo com o seu sistema operacional. Ou você pode acessar o tutorial diretamente em:

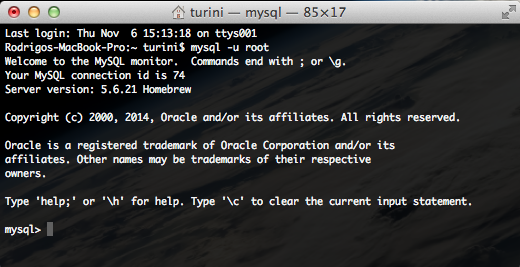
<http://dev.mysql.com/doc/refman/5.7/en/installing.html>

Após baixar e instalar o MySQL, vamos nos logar no MySQL e criar a base de dados (::database::) de nosso projeto. O primeiro passo é bem simples, tudo que você precisa fazer para se logar é abrir seu terminal e digitar:

mysql -u SEU\_USUARIO -p SUA\_SENHA

Como em meu caso o usuário é root e eu não tenho uma senha, só preciso fazer:

mysql -u root



MySQL Command Line ou Workbench?

Há quem prefira utilizar o ::MySQL Workbench::, que é uma ferramenta visual. Pela simplicidade, nos exemplos do curso vamos utilizar o ::MySQL Command Line::. Nele, nós executamos as instruções diretamente pela linha de comando, no terminal.

Se tiver qualquer dificuldade para instalar ou executar os comandos, não pense duas vezes antes de abrir uma dúvida no fórum.

Agora que já estamos logados, podemos criar uma nova ::database:: chamada estoque\_laravel. Isso pode ser feito com o comando:

create database estoque\_laravel;

Após executá-lo, a saída deve ser parecida com:

Query OK, 1 row affected (0.00 sec)

Perfeito, já temos o banco. Para evitar que você tenha que criar todas as tabelas e cadastrar algumas informações manualmente, vamos importar o arquivo dump.sql com alguns produtos já cadastrados em uma tabela de produtos. Um arquivo de ::dumb:: é um arquivo de texto com instruções SQL, com os ::inserts:: de dados ou até mesmo com as instruções de criação de tabelas.

Criando seu próprio dump

Se quiser saber mais sobre esse arquivo de ::dump::, ou mesmo criar o seu próprio, você pode dar uma olhada no link:

<http://dev.mysql.com/doc/refman/5.0/en/mysqldump-sql-format.html>

Isso pode ser bem útil quando queremos fazer ::backups:: de segurança de nossas bases de dados.

Faça o download do dump.sql nesse link:

<http://bit.ly/1BKZMgo>

Feito isso, tudo que precisamos fazer para importar esse ::dump:: em nosso banco de dados é executar a seguinte instrução no terminal:

mysql -uroot < CAMINHO\_COMPLETO\_PARA\_O\_DUMP

Para facilitar o processo, você pode copiar o arquivo dump.sql para a pasta raiz de seu usuário, e executar apenas:

mysql -uroot < dump.sql

Ótimo! Já podemos dar o próximo passo.

**Configuração e conexão com MySQL**

Agora que já temos o banco instalado e configurado, nosso objetivo será estabelecer uma conexão com o MySQL. O Laravel torna essa tarefa bem fácil, basta adicionar as informações do nosso banco no arquivoconfig/database.php. Ele já vem pré-configurado no projeto, só precisamos mudar o trecho de código do MySQL, que deve estar assim:

'mysql' => [

'driver' => 'mysql',

'host' => env('DB\_HOST', 'localhost'),

'database' => env('DB\_DATABASE', 'forge'),

'username' => env('DB\_USERNAME', 'forge'),

'password' => env('DB\_PASSWORD', ''),

'charset' => 'utf8',

'collation' => 'utf8\_unicode\_ci',

'prefix' => '',

'strict' => false,

],

No campo database usaremos estoque\_laravel, usuário root e senha vazia. Caso seu MySQL tenha um usuário ou senha diferente, basta substituir o valor desses campos. O trecho de configuração do MySQL no arquivodatabase.php ficará assim:

'mysql' => [

'driver' => 'mysql',

'host' => env('DB\_HOST', 'localhost'),

'database' => env('DB\_DATABASE', 'estoque\_laravel'),

'username' => env('DB\_USERNAME', 'root'),

'password' => env('DB\_PASSWORD', ''),

'charset' => 'utf8',

'collation' => 'utf8\_unicode\_ci',

'prefix' => '',

'strict' => false,

],

Pronto, isso é tudo que precisamos configurar para acesso ao banco de dados. Vamos testar? Uma forma simples, porém bem manual, de rodar SQLs pelo Laravel é utilizando a classe DB, presente emIlluminate\Support\Facades\DB. Ela tem uma série de métodos para nos ajudar a realizar operações no banco de dados, como select, insert etc.

Quer ver como é simples? Para buscar todos os registros da tabela produtos, basta utilizar o método select com a SQL de consulta:

$produtos = DB::select('select \* from produtos');

Nosso controller ficará assim:

<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;

class ProdutoController extends Controller {

public function lista(){

$produtos = DB::select('select \* from produtos');

return '<h1>Listagem de produtos com Laravel</h1>';

}

}

Todos os valores dos produtos do banco de dados serão retornados em um ::array::, que chamamos de $produtos. Vamos garantir que isso está funcionando? Podemos mudar o método lista para ::concatenar:: o nome e descrição de cada um dos produtos na string de resposta. O código ficará assim:

<?php namespace estoque\Http\Controllers;

use Illuminate\Support\Facades\DB;

class ProdutoController extends Controller {

public function lista(){

$html = '<h1>Listagem de produtos com Laravel</h1>';

$html .= '<ul>';

$produtos = DB::select('select \* from produtos');

foreach ($produtos as $p) {

$html .= '<li> Nome: '. $p->nome .', Descrição: '. $p->descricao .'</li>';

}

$html .= '</ul>';

return $html;

}

}

Achou o código feio? Então somos dois! Mas não se preocupe com isso, logo ele será melhorado, o nosso objetivo agora é garantir que a conexão com o banco de dados foi estabelecida corretamente. Como já temos alguns produtos cadastrados pelo dump, ao acessar <http://localhost:8000/produtos> o resultado deverá ser parecido com:



Excelente, com quase nenhuma configuração já conseguimos nos conectar e executar consultas no banco de dados!

**Aonde chegamos e para onde queremos ir**

Com um pouco de teoria e prática, o significado do padrão MVC começa a fazer sentido. Como vimos, no lugar de deixar todas as responsabilidades no arquivo de rotas, cada comportamento pode ficar muito bem ::encapsulado:: em seu devido controller. As vantagens são inúmeras, mas pra mim as principais são:

* **Legibilidade:** pois no lugar de várias funções anônimas e inúmeras linhas de código em um mesmo arquivo, tudo fica muito bem distribuído. Cada comportamento em seu devido lugar, em classes e métodos com nomes bem definidos.
* **Manutenibilidade:** pois se tudo está bem organizado e encapsulado, não precisaremos mexer em um arquivo com 1500 linhas sempre que um problema aparecer. A listagem de produtos parou de funcionar? Sei que tenho que ir a um método do ProdutoController. Não consigo adicionar usuários? Sei que tenho que verificar na classe UsuarioController.

Viu só? Isso nada mais é do que bom uso da orientação a objetos, que cada vez mais faz parte e influencia o PHP e seus frameworks.

Mas espera, nosso código ainda não obedece o princípio das 3 camadas do MVC, por enquanto todo o trabalho está sendo feito na camada controller! Isso não é bom, vamos resolver?