Laravel /2

Základy webových technológií

Eduard Kuric

STU FIIT

03. 04. 2025

- Základné koncepty Laravel architektúry
- Blade
- Eloquent ORM
- Vzťahy medzi modelmi

Základné koncepty Laravel architektúry

Service Container

- aparát na **manažment závislostí vyžadovaných službou**
- uvažujme, máme nejakú triedu (službu) `MyClass.php` a pri vytvorení inštancie je potrebné podsunúť nejakú závislosť priamo jej konštuktoru
  - `$dependency = new DepClass( config('some.value') );`
  - `new MyClass($dependency);`
- závislostí môže byť niekoľko

Service Container /2

- chceme, aby boli všetky závislosti vytvorené naraz, na jednom mieste
  - v čase vytvorenia inštancie danej služby sú jej k dispozícii
- vytvoríme Service Container:

```
$this->app->bind(MyClass::class, function()
{
  $dependency = new DepClass
              ( config('some.value') );
  return new MyClass( $dependency );
});
// app uroven, napr controller
$instance = $this->app->bind(MyClass::class);
```

Service Provider

- pre aplikáciu zabezpečuje registráciu služieb, komponenty laravelu … databáza, validácia
- v `config/app.php` je pole `providers`, v ktorom je zoznam registrovaných služieb
  - triedy (služby), ktoré sú načítané pre aplikáciu
  - viaceré sú v režime „**deferred**“, sú načítané, keď sú potrebné
- v čistej inštalácii je vytvorený a zaregistrovaný `AppServiceProvider`
  - `App/Providers/AppServiceProvider`

Service Provider /2

- vytvorenie z príkazového riadku
  - `php artisan make:provider MyProvider`
- má metódy `register` a `boot`

- `register`
  - registrácia služieb, napr. Service Container
    ```
    $this->app->bind(MyClass::class, function()
    {}
    ```
- `boot`
  - zavolaná, keď sú zaregistrovaní všetci Service Providers
  - obsahuje napr. View Composer (o tom neskôr)

Deferred Service Provider

```
class DeferredServiceProvider extends
ServiceProvider {
  protected $defer = true;
  public function register()
  {
    $this->app->singleton(Connection::class,
                          function () {});
  }

   public function provides()
   {
     return [Connection::class];
   }
}
```

Middleware

- aparát, ktorý sa zaoberá HTTP požiadavkami pre aplikáciu
- globálny middleware
  - prejde ním každá HTTP požiadavka, napr. sessions
- route middleware
  - middleware priradený ku konkrétnej route
  - napr. overuje, či je používateľ autentifikovaný
  - ak nie je, middleware zabezpečí jeho presmerovanie na stránku prihlasovania

```
Route::get('admin/profile',
    'Dahboard@index')->middleware('auth');
```

Middleware /2

- vytvorenie middleware cez CLI
  - `php artisan make:middleware` **`CheckAge`**
- ak chceme, použiť konkrétny middleware pri smerovaní, musíme ho pridať v `app/Http/Kernel.php`

```
use App\Http\Middleware\CheckAge;
Route::get('admin/profile', function () {
…
})->middleware(CheckAge::class);
```

Middleware /3

```
class CheckAge
{
      public function handle($request,
                Closure $next)
      {
         if ($request->age <= 20) {
            return redirect('home');
         }
         return $next($request);
      }
}
```

Middleware parametre

```
Route::get('post/{id}', function ($id) {
    //
})->middleware(CheckRole::class, 'role:editor');


public function handle($request, Closure $next, $role) {
  if (! $request->user()->hasRole($role)) {
   // Redirect...
  }
}
```

Blade

Blade

- **jednoduchý jazyk na vytváranie šablón** (views) v Laraveli, definuje vlastnú syntax a direktívy
- **šablóny sú kompilované** do natívneho PHP a **cacheované**, dokedy nie sú pozmenené
  - preto prakticky nulová záťaž navyše
- priamo v šablónach je možné použiť aj natívny PHP kód
- extenzia šablón `.blade.php`
- koreňový priečinok `resources/views`

Blade - Layout

- **weby**/webové aplikácie často **zdieľajú rovnaký** hlavný **layout naprieč stránkami**
- preto je dobré, aby (pod)stránky využívali layout a nepoužívali sa rovnaké časti zbytočne opakovane
- **layout** je dobré definovať **ako samostatnú** Blade **šablónu**
- hlavným prínosom sú **dedenie** a **sekcie**

Blade – Layout `@section`

```
<!-- /resources/views/layouts/app.blade.php -->

<!doctype html>
<html>
  <head>
    <title>Page title - @yield('title-subpage')</title>
  </head>
  <body>
    @section('sidebar')
       This is the master sidebar.
       <div class="container">
          @yield('content')
       </div>
    @endsection
  </body>
</html>
```

definícia sekcie

Blade – Layout `@yield`

```
<!-- /resources/views/layouts/app.blade.php -->

<!doctype html>
<html>
  <head>
    <title>Page title - @yield('title-subpage')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master sidebar.
    @show
      <div class="container">
         @yield('content')
      </div>
</body>
</html>
```

vloží/očakáva obsah sekcie `content`

Blade – Layout dedenie `@extends`

- chceme vytvoriť **šablónu pre konkrétnu stránku** webu, ktorá bude **založená na** vytvorenom **layoute**
  - podstránka bude do neho začlenená
- direktívou `@extends` určíme, **ktorý layout má** predmetná **stránka** (jej šablóna) použiť – **zdediť**
- **šablóna stránky môže vložiť do layoutu obsah** prostredníctvom direktívy @section
  - tento obsah sa vloží na miesto, na ktorom je prislúchajúca direktíva @yield

Blade – šablóna konkrétnej stránky

```
@extends('layouts.app')

@section('title-subpage', 'Hage Title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

Blade – šablóna konkrétnej stránky

```
@extends('layouts.app')

@section('title-subpage', 'Subpage title')
```

v layoute

```
<title>Page title - @yield('title-subpage')</title>
```

```
    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

Blade – šablóna konkrétnej stránky

```
@extends('layouts.app')

@section('title-subpage', 'Subpage title')

@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection

@section('content')
    <p>This is my body content.</p>
@endsection
```

v layoute

```
<div class="container">@yield('content')</div>
```

Blade – Layout `@section`

```
<!-- /resources/views/layouts/app.blade.php -->

<!doctype html>
<html>
  <head>
    <title>Page title - @yield('title-subpage')</title>
  </head>
  <body>
    @section('sidebar')
      This is the master side
    @show
      <div class="container">
         @yield('content')
      </div>
  </body>
</html>
```

môže byť vložený aj iný obsah

Blade – @endsection vs. @show

v layoute

```
@section('sidebar')
    This is the master sidebar.
@show
```

```
@section('sidebar')
    @parent

    <p>This is appended to the master sidebar.</p>
@endsection
```

@endsection – ukončuje definíciu sekcie

@show – je akýsi @yield, ktorý umožňuje spojiť pôvodný obsah sekcie s vkladaným, direktívou @parent určíme miesto pôvodného obsahu

Blade – konkretna stránka výsledok

```
<!doctype html>
<html>
  <head>
    <title>Page title - Subpage title</title>
  </head>
  <body>
    This is the master sidebar.
    <p>This is appended to the master sidebar.</p>
    <div class="container">
       <p>This is my body content.</p>
    </div>
  </body>
</html>
```

Blade – Components, slots

- **komponenty** – **znovupoužiteľné** časti rozhrania

```
<!--
/resources/views/components/alert.blade.php
-->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

Blade – Components, slots

- **komponenty** – **znovupoužiteľné** časti rozhrania

```
<!--
/resources/views/components/alert.blade.php
-->

<div class="alert alert-danger">
    {{ $slot }}
</div>
```

`$slot` reprezentuje obsah, ktorý bude vložený do komponentu

Blade – `@component`

- komponent (jeho šablónu) vložíme do stránky cez direktívu `@component`

```
@component('components.alert')
    <strong>Whoops!</strong> Something went wrong!
@endcomponent
```

do stránky sa vloží komponent alert a premenná `$slot` v šablóne komponentu sa nahradí obsahom:
`<strong>Whoops!</strong> Something went wrong!`

Blade – viacero slotov

- **často je potreba zahrnúť viacero slotov v šablóne komponentu**
- slotu môžeme priradiť **identifikátor**

```
<!-- /resources/views/components/alert.blade.php -->

<div class="alert alert-danger">
    <div class="alert-title">{{ $title }}</div>
    {{ $slot }}
</div>
```

Blade – viacero slotov

- pri konštruovaní komponentu použijeme direktívu `@slot`

```
@component('alert')
    @slot('title')
        Forbidden
    @endslot

    You are not allowed to access this resource!
@endcomponent
```

premenná `$title` v šablóne komponentu sa nahradí obsahom `Forbidden`
premenná `$slot` v šablóne komponentu sa nahradí obsahom `You are not allowed to access this resource!`

Blade – Aliasing components

- koreňovým priečinkom šablón je `resource/views`
- nie je špeciálny priečinok na komponenty,
  tie sa očakávajú v priečinku
  `resource/views/components`
- **prehľadnosť v šablónach je dobré podporiť** vhodnou štruktúrou priečinkov
  - podobne ako je priečinok `layout`, pre komponenty napr. priečinok `components`
- pri konštruovaní (používaní) komponentov
  v šablónach je ale potrebné použiť celú cestu
  ku komponentu, napr. `components.alert`

Blade – Aliasing components /2

- pri **zložitejšej hierarchii šablón môže byť ťažkopádne** zakaždým **zahrnúť** k názvu komponentu **celú** jeho **cestu**
- je možné vytvoriť alias, a teda určíme, že keď použijeme `alert`, máme v skutočnosti namysli `components.alert`
- v `App/Providers/AppServiceProvider` v metóde `boot` pridáme:

```
use Illuminate\Support\Facades\Blade;
Blade::component('components.alert',
'alert');
```

Blade – Aliasing components /3

- ak je definovaný alias, je možné pri vkladaní komponentu do stránky priamo použiť direktívu **identifikátor aliasu**:

```
@alert
    You are not allowed to access this resource!
@endalert
```

miesto:

```
@component('alert')
@endcomponent
```

Blade - Control structures `@if`

```
@if (count($records) === 1)
    I have one record!
@elseif (count($records) > 1)
    I have multiple records!
@else
    I don't have any records!
@endif
```

Blade - @unless

```
@unless (Auth::check())
    You are not signed in.
@endunless
```

Blade - @isset, @empty

```
@isset($records)
   // $records je definované a nie je null
@endisset

@empty($records)
    // $records je "empty"...
@endempty
```

- rovnako ako PHP funkcie isset, empty

Blade - `@empty`

`empty` vracia `true` ak:

- premenná nie je definovaná
- `""` (prázdny reťazec)
- `0` (0 ako integer)
- `0.0` (0 ako float)
- `"0"` (0 ako reťazec string)
- `NULL`
- `FALSE`
- `array()` (prázdne pole)

Blade - authentication

```
@auth
// používateľ je autentifikovaný
@endauth

@guest
// používateľ NIE JE autentifikovaný
@endguest
```

Blade - `@hasSection`

- kontrola, či má sekcia nejaký obsah
- ak sekcia `navigation` nemá obsah, nebude táto časť súčasťou výstupu (sekcia nie je použitá):

```
@hasSection('navigation')
    <div class="pull-right">
        @yield('navigation')
    </div>

    <div class="clearfix"></div>
@endif
```

Blade - @switch

```
@switch($i)
    @case(1)
        First case...
        @break

    @case(2)
        Second case...
        @break

    @default
        Default case...
@endswitch
```

Blade - loops

```
@for ($i = 0; $i < 10; $i++)
    The current value is {{ $i }}
@endfor

@foreach ($users as $user)
    <p>This is user {{ $user->id }}</p>
@endforeach

@while (true)
    <p>I'm looping forever.</p>
@endwhile
```

Blade – loops `@forelse`

```
@forelse ($users as $user)
    <tr>{{ $user->name }}</tr>
@empty
// v prípade, že kolekcia $users je prázdna
    <p>No users</p>
@endforelse
```

Blade – loops `@forelse /2`

- `@forelse` inak, použitím `@unless` a `@else`

```
@unless($users->count())
  // Content if there are no users
@else
  // We have users
<table class="table table-striped">
  <thead><th>Full name</th></thead>
  <tbody>
    @foreach($users as $user)
       <tr><td>{{{ $user->full_name }}}</td></tr>
    @endforeach
   </tbody>
</table>
@endunless
```

Blade – loops `@forelse /2`

- `@forelse` inak, cez `@unless` a `@else`

```
@unless($users->count())
  // Content if there are no users
@
<
    </tbody>
</table>
@endunless
```

1. „nevýhoda“ `@forelse`
   - to čo sa má zobraziť keď nemám dáta je zvyčajne kratšie ako to, keď mám dáta
   - konštrukcia – kombinácia `@unless @else` môže byť prehľadnejšia
   - najskôr vidím ako kóder, čo sa má zobraziť, keď nie sú dáta a až potom rieším „ten relatívne dlhý obsah“ keď dáta sú

Blade – loops `@forelse /2`

- `@forelse` inak, cez `@unless` a `@else`

```
@unless($users->count())
  // Content if there are no users
@
<
    @endforeach
   </tbody>
</table>
@endunless
```

2. „nevýhoda“ `@forelse`
   - uvažujme, že potrebujem použiť tabuľku, v ktorej je zoznam používateľov
   - ak použijem `@forelse`, musím ho zaobaliť `<table>`
   - ak použijem `@unless` s `@else`, môžem zobraziť table, iba ak nie sú dáta

Blade – @continue, @break

```
@foreach ($users as $user)
    @if ($user->type == 1)
        @continue
    @endif

    <li>{{ $user->name }}</li>

    @if ($user->number == 5)
        @break
    @endif
@endforeach
```

Blade – loop variable

- index aktuálnej iterácie, prvej, poslednej...

```
$loop->index      // od 0
$loop->iteration // od 1
$loop->remaining
$loop->count
$loop->first
$loop->last
$loop->depth      // úroveň vnorenia aktuálnej slučky
$loop->parent
```

Blade – loop variable /2

```
@foreach ($users as $user)
  @foreach ($user->posts as $post)
    @if ($loop->parent->first)
      This is first iteration of the parent loop.
    @endif
  @endforeach
@endforeach
```

Blade – Comments --

```
{{ -- This comment will not be
  present in the rendered HTML -- }}
```

Blade - PHP

- cez direktívu `@php` je možné v šablóne zahrnúť priamo PHP kód

```
@php
    // PHP kód
@endphp
```

Blade - Subviews

- zahrnutie inej šablóny v cieľovej šablóne

```
<div>
    @include('shared.errors')

    <form>
        <!-- Form Contents -->
    </form>
</div
```

Blade - Subviews

- **zahrnutá šablóna má k dispozícii („dedí“) dáta cieľovej šablóny**, ale je možné jej poskytnúť „extra“ dáta
  - `@include('view.name', ['some' => 'data'])`
- ak šablóna neexistuje, je vyhodená exception, šablónu je možné zahrnúť iba ak existuje
  - `@includeIf('view.name', ['some' => 'data'])`
- alebo na základe podmienky
  - `@includeWhen($bool, 'view.name', ['some' => 'data'])`
- alebo použiť prvú dostupnú/existujúcu šablónu:
  - `@includeFirst(['custom.admin', 'admin'], ['some' => 'data'])`

Blade – Šablóny pre kolekcie

- direktívou @each môžeme povedať, že „čiastková“ šablóna sa má použiť pre každý prvok z kolekcie

```
@each('view.name', $jobs, 'job')
```

- `jobs` je kolekcia, `job` je názov premennej (prvku z kolekcie), ktorá sa očakáva v čiastkovej šablóne

```
@each('view.name', $jobs, 'job',
'view.empty')
```

- 4. voliteľný argument, určuje šablónu, ktorá sa má použiť, **ak je kolekcia prázdna**

Blade – Šablóny pre kolekcie

- direktívou @each môžeme povedať, že „čiastková“ šablóna sa má použiť pre každý prvok z kolekcie

- čiastkové šablóny **nemajú k dispozícii** („nededia“) **dáta z cieľovej šablóny**
- ak by mali mať, treba miesto `@each` použiť `@foreach` a `@include`

```
@each('view.name', $jobs, 'job',
'view.empty')
```

- 4. voliteľný argument, určuje šablónu, ktorá sa má použiť, ak je kolekcia prázdna

Blade - Stacks

- uvažujme, **nejaká šablóna vyžaduje javascript/css, ktorý okrem nej nepotrebuje „nikto iný“**
  - preto sme ho nezahrnuli v hlavičke layoutu
- chceme, aby vo výslednom HTML dokumente bol v hlavičke, ak sa vykresluje predmetná šablóna
- použijeme direktívu `@push`, s identifikátorom, ktorý určuje miesto, kde sa daný obsah vloží

```
@push('scripts')
 <script src="/example.js"></script>
@endpush
```

Blade – Stacks /2

- direktíva `@stack` určuje miesto, na ktoré sa má vložiť „pushnutý“ obsah
- identifikátor určuje konkrétny obsah

```
<head>
    <!-- Head Contents -->

    @stack('scripts')
</head>
```

Blade – vlastné direktívy

- napr. chceli by sme zadefinovať vlastný formát dátumu
  - použiť direktívu pri každom výskyte dátumu v šablóne
  - argumentom direktívy bude dátum z databázy
  - a teda `@datetime($var)`
- v `App/Providers/AppServiceProvider` v metóde `boot` pridáme:

```
Blade::directive('datetime', function ($expression) {
        return "<?= $expression->format('m/d/Y H:i'); ?>";
       });
```

- **pozor** značky `<?= ?>`, resp. `<?php echo ?>` **- musia byť**

View composer

- uvažujme, máme dáta, ktoré chceme mať k dispozícii zakaždým, keď je nejaká šablóna (napr. **`profile`**) zostavovaná
- vytvoríme view composer a v metóde **`boot`** nejakého Service Providera ho zaregistrujeme: (napr. `ComposerServiceProvider`)

```
View::composer(
    'profile',
    'App\Http\ViewComposers\ProfileComposer'
);
```

View composer /2

- view composer má metódu `compose`, ktorá je zavolaná predtým, ako sa začne daná (`profile`) šabloná zostavovať

```
class ProfileComposer
{
  public function compose(View $view) {
     $view->with('some', 'data');
  }
}
```

Eloquent ORM

Eloquent ORM

- objektovo-relačný mapovač
- model ~= záznam/riadok v tabuľke
- ak chceme pristupovať k dátam tabuľky, musíme vytvoriť model, napr. cez CLI:
  - `php artisan make:model Product`

ORM - konvencie

- názov tabuľky
  - model: `Product`, tabuľka: **`products`**
- primárny kľúč
  - názov stĺpca **`id`**, možné zmeniť v modeli
    `protected $primaryKey='nazov_stlpca';`
  - očakáva sa inkrementujúci **`int`**, možné zmeniť v modeli
    `$incrementing=false;`
    `protected $keyType='string'`

ORM – konvencie/2

- očakáva, že existujú stĺpce **`created_at`** a **`updated_at`**
- ak ich nechceme v modeli nastavíme
  - `protected $timestamps = false;`
- ak chceme vlastné názvy stĺpcov:
  - `const CREATED_AT = 'creation_date';`
  - `const UPDATED_AT = 'last_update';`

ORM dopyt - všetky záznamy

```
use App\Models\Product;

$products = App\Product::all();

// iterovanie cez produkty
foreach ($products as $product) {
    echo $product->name;
}
```

ORM dopyt – podmienky, usporiadanie, obmedzenia

```
$products =
App\Product::where('status', 1)
            ->orderBy('name', 'desc')
            ->take(10)
            ->get();
```

ORM – `get/all` kolekcia

- metódy get a all vracajú kolekciu - inštanciu: `Illuminate\Database\Eloquent\Collection`
- kolekcia má definovaných veľa užitočných metód, napr. `reject`:

```
// vráti iba aktívne produkty
$products = $products->reject(
  function ($product) {
    return $product->active == false;
});
```

ORM – dávkovanie `chunk()`

- ak potrebujeme spracovať tisícky záznamov,
  je vhodné použiť dávkovanie (tzv. chunking results)
  niečo ako stránkovanie

```
Product::chunk(100, function ($products) {
   foreach ($products as $product) {
      //
   }
   // ak vratime return false; dalsi
   // chunk nebude
});
select * from `products` limit 100 offset 0;
select * from `products` limit 100 offset 100;
…
```

ORM – `cursor()`

- pri spracovaní množstva dát je možné pomocou cursora iterovať cez záznamy v DB
- vykoná sa síce 1 dopyt, ale driver nevyberie výsledný súbor naraz (`PDOStatement::fetch`)

```
foreach (Product::where('col1', 'val1')
->cursor() as $product)
{
    //
}
```

Chunk vs cursor

- Test: dáta tabuľka users – default Laravel app
- PHP 7.0.12, MySQL 5.7.16, Laravel 5.3.22

10,000 records

	Time(sec)	Memory(MB)
get()	0.17	22
chunk(100)	0.38	10
chunk(1000)	0.17	12
cursor()	0.16	14

100,000 records

	Time(sec)	Memory(MB)
get()	0.8	132
chunk(100)	19.9	10
chunk(1000)	2.3	12
chunk(10000)	1.1	34
cursor()	0.5	45

`cursor` – rýchlosť ; `chunk` – menej pamäte

ORM – konkrétne záznamy

```
// produkt s id 1
$product = App\Product::find(1);

// prvy aktivny produkt
$product = App\Product::where('active',
1)->first();

// produkty, ktorych id su 1,2,3
$products = App\Product::find([1, 2, 3]);
```

ORM – výnimky záznam neexistuje

```
// vyhodi vynimku, ak produkt
// s id 1 nie je v DB
$product = App\Product::findOrFail(1);

// vynimka, ak neexistuje produkt
// splnajuci podmienku
$product = App\Product::where('price',
'>', 100)->firstOrFail();
```

- ak nie je výnimka odchytená, vratí sa 404 - not found

ORM – agregačné funkcie

- `count, sum, max, ...`

```
$count = App\Product::where('active', 1)
->count();
```

```
$max = App\Product::where('active', 1)
->max('price');
```

ORM – vloženie záznamu

- **metóda** `save`

```
// novy produkt
$product = new Product;
$product->name = 'Superbike 300';
// insert do DB
$product->save();
```

ORM – zmena záznamu //update

- **metóda** `save`

```
// načítanie exist. produktu s id 1
$product = App\Product::find(1);
$product->name = 'Novy nazov';
$product->save();
```

ORM – hromadné zmeny

- **Mass assignment**
- predvolene sú modely chránené pred (neželanými) zmenami
  - v modeli je potrebné nastaviť `fillable` alebo `guarded` atribút

```
App\Product::where('active', 1)
  ->where('price', '>', '200')
  ->update(['price' => 300]);
```

ORM – `fillable/guarded`

- `fillable` určuje, ktoré atribúty môžu byť hromadne menené - „white list“

```
protected $fillable = ['name'];
```

- `guarded` naopak, ktoré nesmú byť hromadné menené – „black list“
- nastavuje sa buď `fillable` alebo `guarded`, nie súčasne

ORM - `create`

- keď je nastavený `fillable`, môžeme použiť metódu `create` na vytvorenie záznamu

```
$product =
App\Product::create(['name' => 'GPR
DAS 500']);
```

```
$product->fill(['name' => 'GPR DAS
600']);
```

ORM – `firstOrNew/firstOrCreate`

- pokúsia sa nájsť záznam v tabuľke, ak nenájdu vytvoria nový záznam s atribútmi v argumente

```
$product =
App\Product::firstOrNew(['name' =>
'GPR DAS 500']);
```

- `firstOrNew` – vytvorí záznam, ale neuloží do DB, je potrebné ešte zavolať metódu `save`
- `firstOrCreate` – vytvorí perzistentný záznam

ORM – `updateOrCreate`

- ak existuje záznam z Bratislavy do Bruselu, zmení sa cena na 120, inak sa vytvorí nový záznam

```
$flight = App\Flight::updateOrCreate(
    ['departure' => 'Bratislava',
     'destination' => 'Brussels'],
    ['price' => 120]
);
```

- vytvorí perzistentný záznam, nie je potrebné volať `save`

ORM – vymazanie záznamu

- podľa `id`

```
App\Product::destroy(1);
```

- pole `id`čiek

```
App\Product::destroy([1, 2, 3]);
```

```
$deletedRows =
App\Product::where('active', 0)
->delete();
```

ORM – „soft“ vymazanie

- v skutočnosti sa záznam z DB nevymaže, nastaví sa hodnota stĺpca `deleted_at`
- v tabuľke je potrebné vytvoriť stĺpec `deleted_at`
- v modeli:

```
class Product extends Model
{
  use SoftDeletes;
  protected $dates = ['deleted_at'];
}
```

- po zavolaní metódy `delete` bude nastavený v `deleted_at` aktuálny dátum a čas

ORM – „soft“ vymazanie /2

- kontrola, či je záznam vymazaný „soft“

```
if ($product->trashed()) {}
```

- „undelete“ – obnovenie vymazaných záznamov

```
$product->restore();
```

- niekedy je potreba skutočne vymazať aj „soft“ záznam

```
$product->forceDelete();
```

ORM – „soft“ záznamy

- záznamy vymazané „soft“ sú automaticky ignorované pri dopytovaní sa, v určitých prípadoch s nimi môžeme chcieť počítať

```
// zahrnieme aj vymazané záznamy soft
$products = App\Product::withTrashed()
->where('id', 1)->get();


// iba v záznamoch vymazaných soft
$products = App\Product::onlyTrashed()
->where('id', 1)->get();
```

ORM – globálne obmedzenia

- v modeloch môžeme definovať obmedzenia, ktoré budú uplatňované pri všetkých dopytoch
  - napr. vždy chceme uvažovať iba používateľov, ktorých vek je väčší ako 20
- **príkladom takéhoto „obmedzenia“ je aj vymazanie „soft“**
  - pri každej požiadavke na vymazanie nejakého záznamu v modeli, nastavujeme atribút `deleted_at`

ORM – globálne obmedzenia /2

```
namespace App\Scopes;

use Illuminate\Database\Eloquent\Scope;
use Illuminate\Database\Eloquent\Model;
use Illuminate\Database\Eloquent\Builder;

class AgeScope implements Scope
{
    public function apply(Builder
                $builder, Model $model)
    {
        $builder->where('age', '>', 20);
    }
}
```

ORM – globálne obmedzenia /3

```
namespace App;

use App\Scopes\AgeScope;
use Illuminate\Database\Eloquent\Model;

class User extends Model
{
    protected static function boot()
    {
        parent::boot();

        static::addGlobalScope(new AgeScope);
    }
}
```

ORM – globálne obmedzenia /4

- s použitím anonymnej funkcie

```
class User extends Model
{
    protected static function boot() {
        parent::boot();

        static::addGlobalScope('age',
        function (Builder $builder)
        {
            $builder->where('age', '>', 20);
        });
    }
}
```

ORM – globálne obmedzenia /5

- dočasné zrušenie obmedzenia/obmedzení

```
User::withoutGlobalScope(AgeScope::class)
->get();

User::withoutGlobalScopes()->get();

User::withoutGlobalScopes([
    FirstScope::class, SecondScope::class
])->get();
```

- ak bola použitá anonymná funkcia

```
User::withoutGlobalScope('age')->get();
```

ORM – lokálne obmedzenia

- priamo v modeli definujeme metódy – obmedzenia, kľúčový prefix `scope`
- použijeme ich pri dopytovani sa

```
class User extends Model {
    public function scopePopular($query) {
        return $query->where('votes', '>', 100);
    }

    public function scopeActive($query) {
        return $query->where('active', 1);
    }
}

$users = App\User::popular()->active()
->orderBy('created_at')->get();
```

ORM – dynamické obmedzenia

- pri obmedzení môžeme požadovať odovzdať aj parameter

```
class User extends Model {
 public function scopeOfType($query, $type)
 {
   return $query->where('type', $type);
 }
}

$users = App\User::ofType('admin')->get();
```

Vzťahy medzi modelmi

One-to-one

- používateľ má telefón

```
class User extends Model
{
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

One-to-one /2

Phone model musí mať cudzí kľúč, očakáva sa konvencia:
`user_id`

```
$phone = User::find(1)->phone;
```

konvenciu môžeme zmeniť:

```
return $this->hasOne('App\Phone', 'foreign_key');
```

```
    {
        return $this->hasOne('App\Phone');
    }
}
```

One-to-one /3

ak nie je **v nadradenej** tabuľke **dodržaná konvencia** `id` pre primárny kľúč, môžeme ho určiť (`local_key`):

```
return $this->hasOne('App\Phone', 'foreign_key', 'local_key');
```

```
    public function phone()
    {
        return $this->hasOne('App\Phone');
    }
}
```

One-to-one inverzný vzťah

- používateľ má telefón

```
class Phone extends Model
{
    public function user()
    {
      return $this->belongsTo('App\User');
    }
}
```

One-to-one predvolený model

- ak nie je k telefónu priradený používateľ, vráť predvolený model namiesto prázdneho modelu
- Null Object pattern

```
// vrati prazdny model
return $this->belongsTo('App\User')->withDefault();

// model s preddefinovanym atributom, cez pole
return $this->belongsTo('App\User')->withDefault([
        'name' => 'Guest Author',
    ]);

// cez anonymnu funkciu
return $this->belongsTo('App\User')->withDefault(function ($user) {
        $user->name = 'Guest Author';
    });
```

One-to-many

- príspevok blogu má niekoľko komentárov

```
class Post extends Model
{
    public function comments()
    {
        return $this->hasMany('App\Comment');
    }
}
$comments = App\Post::find(1)->comments()
->where('title', 'foo')->first();
```

- konvencie rovnako ako pri one-one, možné zmeniť
- inverzne `belongsTo`

Many-to-many

- používateľ môže mať niekoľko rolí,
  tú istú rolu môže mať niekoľko používateľov

```
class User extends Model
{
 public function roles()
 {
  return $this->belongsToMany('App\Role');
 }
}
```

Many-to-many

- očakáva sa väzobná tabuľka, máme tabuľky **u**sers, **r**oles
- je očakávná väzobná tabuľka s názvom role_user
- abecedné poradie - **r**ole, potom **u**ser, oddeľovač _,

```
$roles = App\User::find(1)->roles()->orderBy('name')->get();
```

- očakávanú konvenciu je možné zmeniť:

```
return $this->belongsToMany('App\Role', 'role_user', 'user_id', 'role_id');
```

Stĺpce vo väzobnej tabuľke

- k stĺpcom vo väzobnej tabuľke sa pristupuje použitím tzv. pivotu

```
$user = App\User::find(1);

foreach ($user->roles as $role) {
  // created_at je stĺpec
  // vo väzobnej tabuľke
  echo $role->pivot->created_at;
}
```

Filtrovanie vzťahov

- **vo väzobnej tabuľke môžu byť stĺpce, na základe ktorých chceme filtrovať**
- napr., iba tie role, ktoré boli schválené

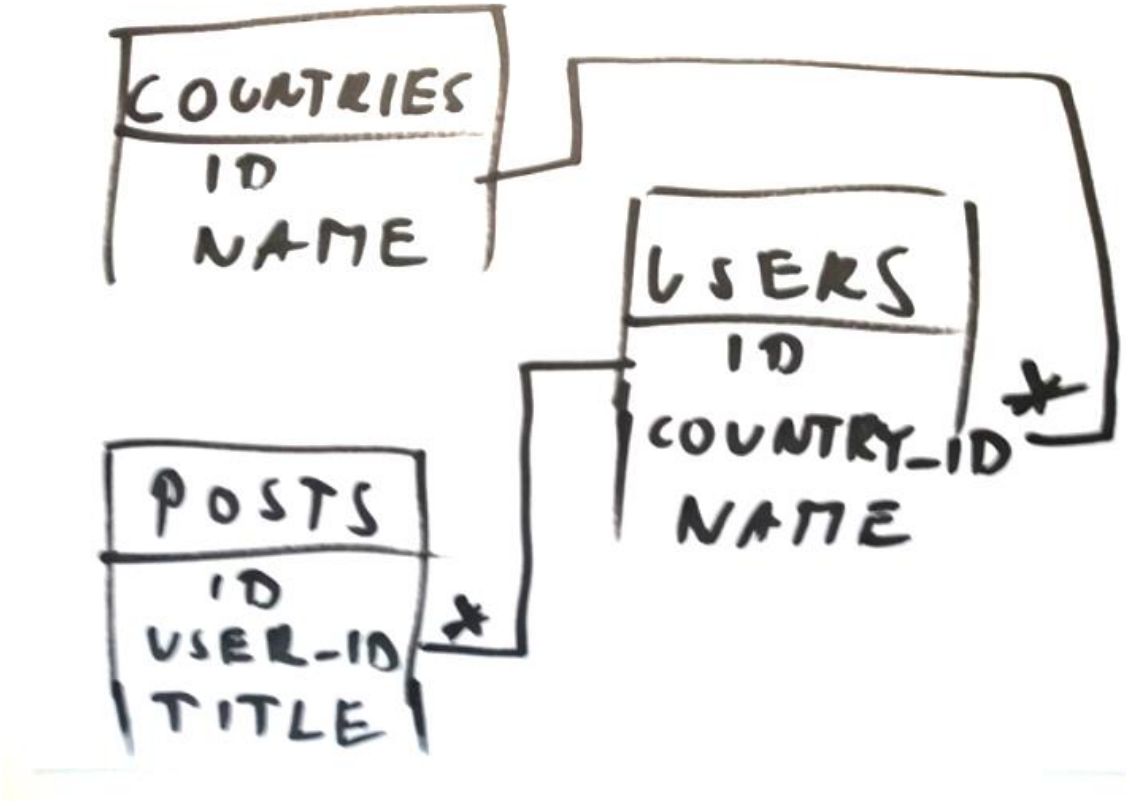
```
return $this->belongsToMany('App\Role')
->wherePivot('approved', 1);
```

- alebo tie, ktorých priorita je 1 a 2

```
return $this->belongsToMany('App\Role‘)
->wherePivotIn('priority', [1, 2]);
```

Has many through

- uvažujme, chceme všetky príspevky, ktoré sú z nejakej krajiny



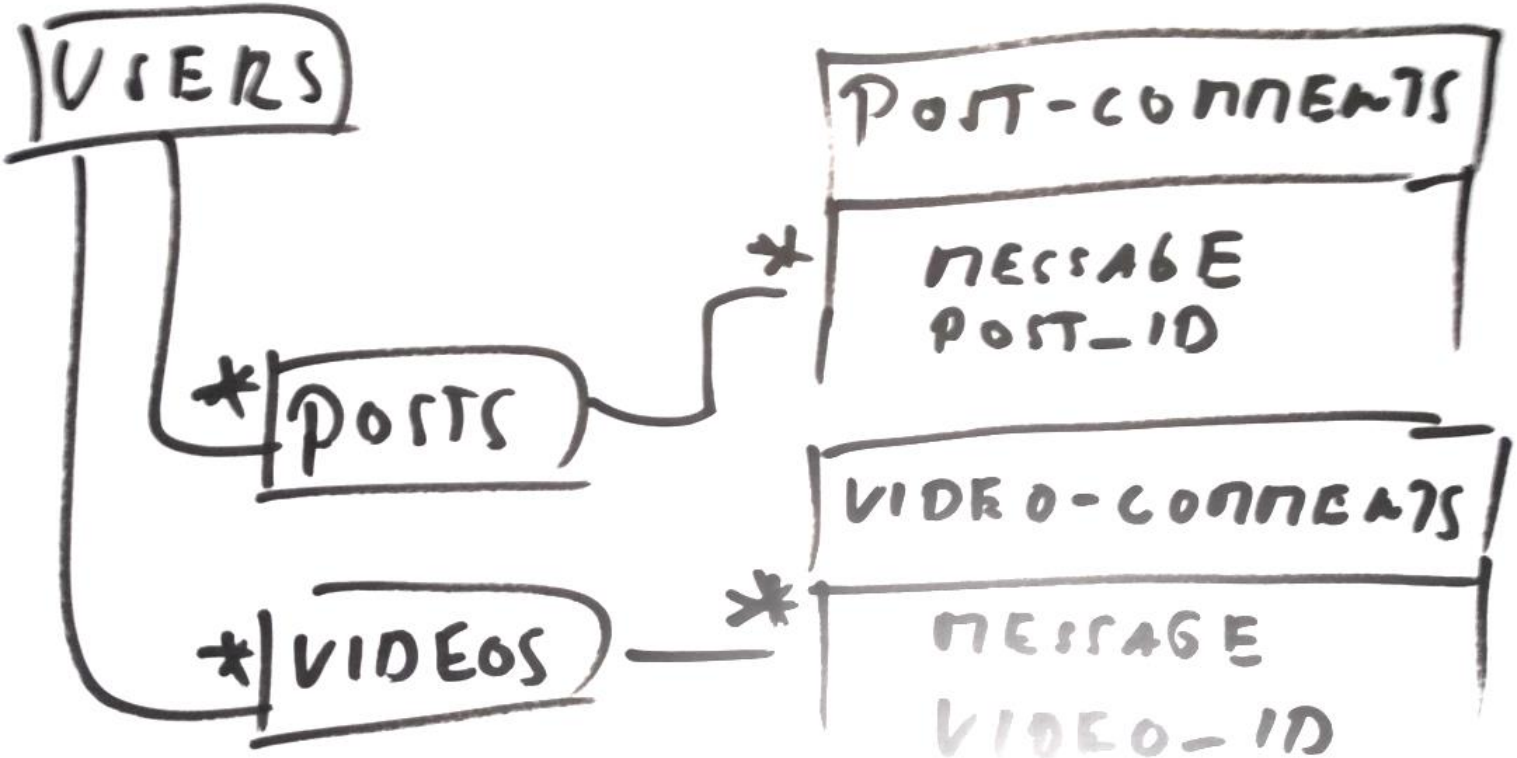
Has many through /2

```
class Country extends Model
{
 public function posts()
 {
  return $this->hasManyThrough('App\Post',
                               'App\User');
 }
}
```

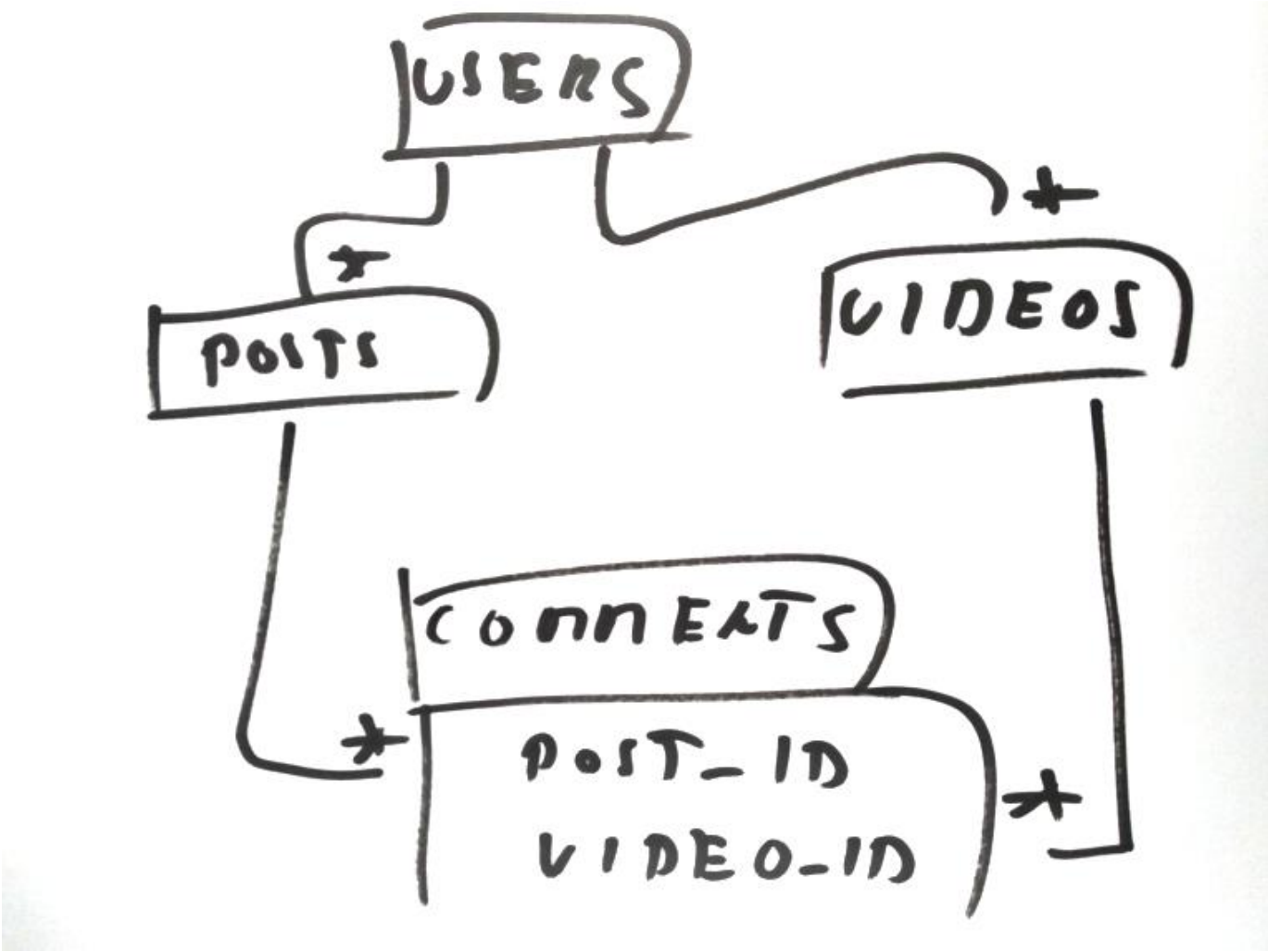
- **1. argument je model, ktorý chceme získať**
- **2. argument je „väzobný model“**

Polymorfné vzťahy

- uvažujme **používateľov, ktorí môžu komentovať príspevky aj videá**



Polymorfné vzťahy /2



Polymorfné vzťahy /3

- Laravel - `comments`
  - `id – integer`
  - `body – text`

  `// type: post alebo video`
  - **`commentable_type`** `- string`

  `// id: konkrétny post alebo video`
  - **`commentable_id`** `– integer`

Polymorfné vzťahy /4

```
class Comment extends Model {
 public function commentable() {
  return $this->morphTo();
 }
}
class Post extends Model {
 public function comments() {
  return $this->morphMany('App\Comment', 'commentable');
 }
}
class Video extends Model{
 public function comments() {
  return $this->morphMany('App\Comment', 'commentable');
 }
}
```

Načítanie záznamov

- prostredníctvom **metód**, alebo **dynamických atribútov**

```
class User extends Model
{
  public function posts()
  {
   return $this->hasMany('App\Post');
  }
}
```

Načítanie záznamov /2

```
// použijeme metódu posts()
$user = App\User::find(1);
$user->posts()->where('active', 1)
->get();

// ak nepotrebujeme ďalšie podmienky
// použijeme dynamický atribút
$user = App\User::find(1);
foreach ($user->posts as $post) {
    //
}
```

Lazy loading, eager loading

- dynamické atribúty – **dáta sú načítané, keď sú požadované – lazy loading**
- ak vieme, že budeme k dátam pristupovať, je dobré ich „prednačítať“ – **eager loading**
  - dá sa tým významne zredukovať množstvo SQL dopytov

Lazy loading /2

- problém N+1 dopytov
- uvažujme príklad, vzťah kniha -[má]- autora

```
class Book extends Model
{
  public function author()
  {
    return $this->belongsTo('App\Author');
  }
}
```

Lazy loading /3

```
$books = App\Book::all(); // 25 knih
// lazy loading
foreach ($books as $book) {
    echo $book->author->name;
}
```

- vykoná sa prvý dopyt na získanie všetkých kníh
- v slučke, sa pre každú knihu vykoná dopyt na získanie informácií o autorovi k danej knihe
- ak máme 25 kníh, vykoná sa 26 dopytov (N+1)

Eager loading

- použitím metódy `with` dokážeme túto operáciu zredukovať na 2 dopyty // eager loading

```
$books = App\Book::with('author')->get();
foreach ($books as $book) {
    echo $book->author->name;
}
select * from books
select * from authors where id in (1, 2, 3, 4, 5, ...)
```

Pridávanie záznamov

- uvažujme, chceme pridať komentár k príspevku
- namiesto nastavovania `post_id` komentáru, vložíme komentár priamo k príspevku s využitím vzťahu

```
$comment = new App\Comment(
['message' => 'A new comment.']);
$post = App\Post::find(1);
$post->comments()->save($comment)
```

Pridávanie záznamov - saveMany

```
$post = App\Post::find(1);

$post->comments()->saveMany([
 new App\Comment(['message' => 'A new comment.']),
 new App\Comment(['message' => 'Another comment.']),
]);
```

Pridávanie záznamov - `create`

- rozdiel - `create` očakáva pole

```
$comment = $post->comments()->create([
 'message' => 'A new comment.',
]);

$post->comments()->createMany([
 ['message' => 'A new comment.',],
 ['message' => 'Another new comment.',],
]);
```

Metóda `attach`

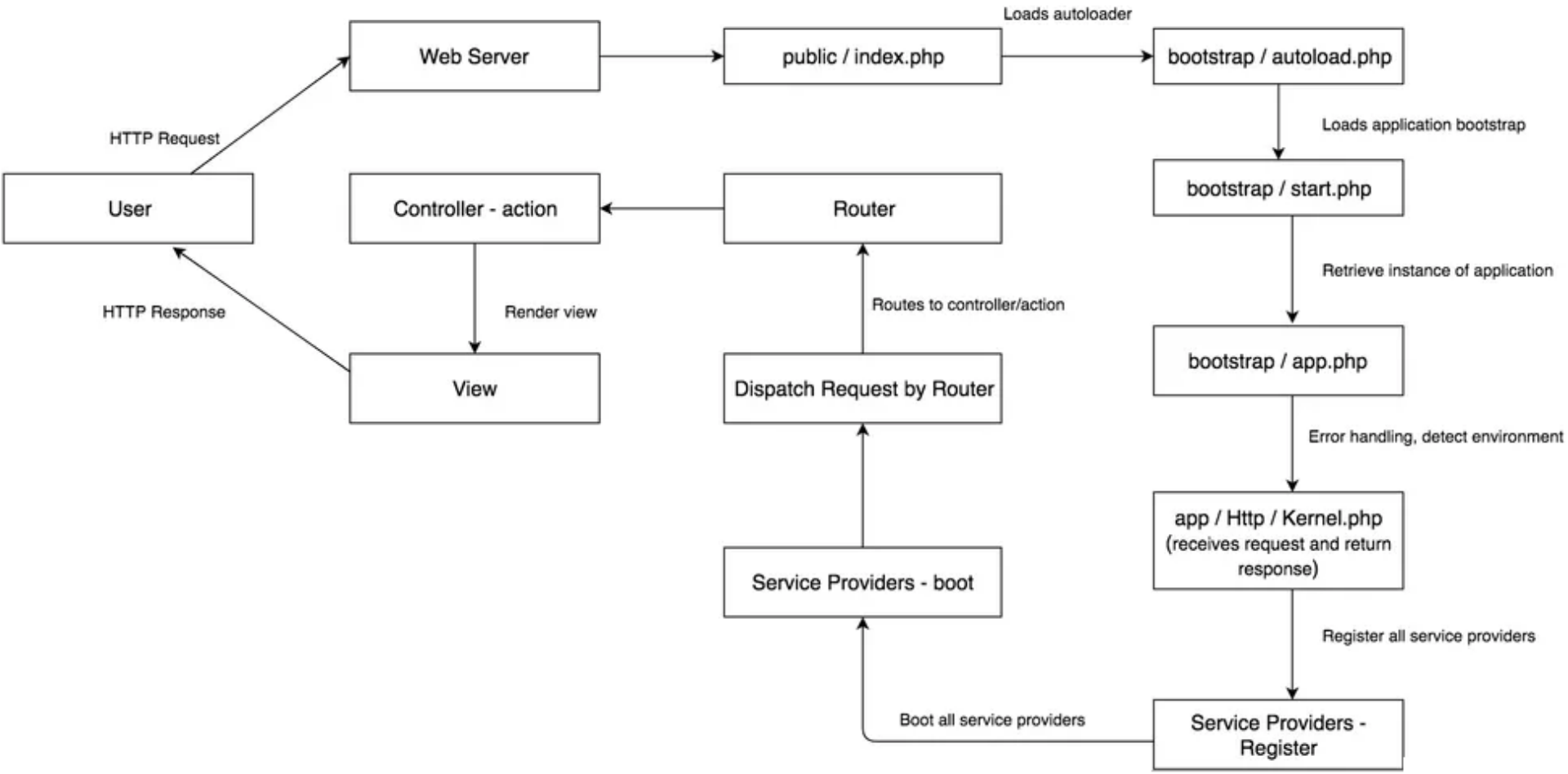
- uvažujme vzťah M:N, users – roles
- chceme prepojiť používateľa s rolou,
  potrebujeme vytvoriť záznam vo väzobnej tabuľke

```
$user = App\User::find(1);
$user->roles()->attach($roleId);

// následne môžeme nastaviť ďalšie stĺpce
$user->roles()->attach($roleId,
['expires' => $expires]);
```

Metóda `detach`

```
$user->roles()->detach($roleId);

// všetky role
$user->roles()->detach();
```

Životný cyklus požiadavky



Štúdium

- za domácu úlohu naštudovať
  - Raw SQL, transactions
  - Query builder
  - Migrácie
- prečítajte si tutoriál ako vyvíjať a debugovať Laravel aplikácie pomocou PHPStorm
- príklady a tutoriály
- študujte Laravel dokumentáciu