



ictPRO

16

ictPRO
International Corporate Training

KOMPETENČNÍ STROM

Konkrétní profesní kompetence
Projekty a procesy Obchod
Management Marketing
Výroba, inovace Personalistika

Obecné pracovní kompetence
Osobnostní kompetence

pozitivní IQ stres vztahy sebepojetí
typologie kreativita emoce paměť motivace

Již bezmála 20 let rozvíjíme,
inspirujeme, motivujeme,
jsme tu s Vámi a pro Vás...

www.ictpro.cz

Základy Laravel frameworku

Filip Majerík

ictPRO
International Corporate Training

Průběh kurzu

- Kam kráčí PHP8+
- Základní přehled OOP v PHP
- PSR-4 & Composer
- Představení frameworku Laravel
- Základní debug Laravel aplikací

(Laravel Debugbar & Laravel Telescope)

Průběh kurzu

- Routing & Controllery
- Základy Eloquent & ORM
- Blade - šablonovací systém
- Autorizace v UI (Laravel Breeze)
- API a autorizace (Laravel Sanctum)

Kam kráčí PHP8+.





PHP 8.0 - 11/2020

● JIT (Just-In-Time Compiler)

- ⊙ Rozšíření pro podporu long-running systémů.
- ⊙ Doplněk k existující OPcache.

● Union Types

- ⊙ Definice více typů pro vstupní parametr / návratovou hodnotu.
- ⊙ Není nutné psát PHPDoc.

```
function foo(int|string $value): int|string {  
    return $value;  
}
```

PHP 8.0 - 11/2020

● Nullsafe Operator

- ⦿ Nástroj pro zjednodušené řetězové volání s null výsledkem.

// Starý styl

```
$image = $user ? $user->getProfile() ? $user->getProfile()->getImage() : null : null;
```

// Nový styl

```
$image = $user?->getProfile()?->getImage();
```

● Named Arguments

- ⦿ Předávání parametrů dle názvu nikoliv dle pořadí.

```
htmlspecialchars($string, double_encode: false);
```

PHP 8.0 - 11/2020

● Attributy (anotace)

- ⦿ Nativní alternativa k PHPDoc anotacím.
- ⦿ Lze je číst pomocí ReflectionClass, ReflectionMethod apod.

```
#[Route("/home")]  
function index() {}
```


PHP 8.0 - 11/2020

● Constructor Property

⊙ Nástroj k zjednodušení tříd - méně kódu.

// Dříve:

```
class Point {  
    public int $x;  
    public int $y;  
    public function __construct(int $x, int $y) {  
        $this->x = $x;  
        $this->y = $y;  
    }  
}
```

// Nově:

```
class Point {  
    public function __construct(  
        public int $x,  
        public int $y,  
    ) {}  
}
```

PHP 8.0 - 11/2020

● Funkce `match()`

- ⊙ Alternativní funkce k `switch()`.
- ⊙ Přímo vrací odpovídající hodnotu.
- ⊙ Může volat funkci apod.

```
echo match($status) {  
    'draft' => 'Čeká na schválení',  
    'published' => 'Zveřejněno',  
    default => 'Neznámý stav',  
};
```

PHP 8.0 - 11/2020

● throw jako výraz

- ⦿ Nově lze throw použít například v ternárním operátoru.
- ⦿ Není nutné dělat speciální "if(...) { throw new ... }"

```
$fn = fn($x) => $x > 0 ? $x : throw new Exception("Chyba");
```

PHP 8.1 - 11/2021

Enums - výčtový typ

- ⦿ Pro definování pevně dané sady hodnoty s vybraným typem.
- ⦿ Alternativa (náhrada) "constant class".

```
enum Status {  
    case Draft;  
    case Published;  
    case Archived;  
}  
  
function canEdit(Status $status): bool {  
    return $status === Status::Draft;  
}
```


PHP 8.1 - 11/2021

● Readonly Properties

- ⦿ Pro vytvoření třídní proměnné s nezměnitelnou hodnotou.
- ⦿ Pokus o změnu končí na PHP Fatal error.

```
class User {  
    public function __construct(  
        public readonly string $name  
    ) {}  
}
```

PHP 8.1 - 11/2021

○ Fibers

- ⊙ Přináší do PHP kooperativní multitasking - základ pro async PHP.
- ⊙ Využívají se knihovny ReactPHP/AMP pro neblokující IO operace.

```
$fiber = new Fiber(function(): void {  
    $value = Fiber::suspend('pause');  
    echo "Resumed with: $value\n";  
});  
  
echo $fiber->start(); // pause  
$fiber->resume('resume'); // Resumed with: resume
```

PHP 8.1 - 11/2021

● Intersection Types

- ⊙ Definice typu, který musí kombinovat více typů současně.

```
function handle(A&B $value): void {  
    $value->doSomething();  
}
```

● Návratová hodnota **never**

```
function fail(): never {  
    throw new Exception("Fatal");  
}
```

PHP 8.2 - 12/2022

● @deprecated ~~Dynamické vlastnosti~~

- ⦿ PHP upozorňuje na práci s nedeklarovanými property třídy.
- ⦿ V budoucích PHP bude kompletně tato funkce odebrána.

```
class User {}
```

```
$user = new User();
```

```
$user->name = 'Anna'; // ⚠ Warning
```


PHP 8.2 - 12/2022

● Readonly class

- ⦿ Označení, že celá instance třídy je po vytvoření neměnná.
- ⦿ Snadné vytváření Immutable objektů v kombinaci s public properties.
- ⦿ Vhodné např. pro DTO.

```
readonly class Point {  
    public function __construct(  
        public int $x,  
        public int $y,  
    ) {}  
}
```

PHP 8.2 - 12/2022

- Disjunktní typy - **true, false, null**

- ⦿ Tyto typy lze použít jako návratové hodnoty u metod a funkcí.
- ⦿ Pokud výstup neodpovídá -> PHP Fatal Error.

```
function isReady(): true {  
    return true;  
}
```

```
function foo(): int|false {}
```

PHP 8.2 - 12/2022

● Konstanty v Traits

- ⦿ Nově lze definovat konstanty na úrovni Traitů, stejným způsobem jako u tříd.

```
trait Identifiable {  
    public const TABLE = 'users';  
}
```

PHP 8.3 - 11/2023

● Typování třídních konstant

- ⊙ U konstant lze nově uvádět datový typ.

```
class Config {  
    public const string ENV = 'production';  
}
```

● Přidána funkce `json_validate()`

- ⊙ Výkonově efektivnější než `json_decode($input)` a test na `json_last_error()`.

```
$isValid = json_validate($input); // true / false
```


PHP 8.3 - 11/2023

● Random Extension

- ⦿ Objektové rozhraní pro práci s náhodnými čísly. Náhrada za `rand()` a `mt_rand()` funkce.
- ⦿ Podpora využívání různých engine (Mt19937, Xoshiro256**,...).
- ⦿ Možnost vytvoření vlastního engine - **Random\Engine**.

```
use Random\Randomizer;
```

```
$randomizer = new Randomizer();
```

```
$value = $randomizer->getInt(1, 10);
```

PHP 8.3 - 11/2023

● Initializer pro deklaraci třídních atributů

- ⦿ Nově lze používat `new` direktivu přímo při deklaraci konstant a defaultních hodnot v rámci deklarace třídy.
- ⦿ Není tak nutné vytvářet `__constructor()`.

```
class Controller {  
    private Logger $logger = new ConsoleLogger();  
}
```

PHP 8.4 - 11/2024

Property Hooks

- Definice oddělené logiky pro čtení/zápis atributů (C# style).
- Není nutné vytvářet gettery a settery.

```
class User {  
    public string $name {  
        get => $this->firstName . ' ' . $this->lastName;  
        set(string $value) {  
            [$this->firstName, $this->lastName] = explode(' ', $value, 2);  
        }  
    }  
}
```

- Propertities mohou mít asymetrickou viditelnost.

```
class Config {  
    public private(set) string $env = 'production';  
}
```

PHP 8.4 - 11/2024

● Nové array funkce

- ⦿ `array_find()` - vrátí první prvek odpovídající predikátu
- ⦿ `array_find_key()` - vrátí klíč prvku odpovídajícího predikátu
- ⦿ `array_any()` - vrátí *true*, pokud některý z prvků odpovídá predikátu
- ⦿ `array_all()` - vrátí *true*, pokud všechny prvky odpovídají predikátu

PHP 8.4 - 11/2024

- **@deprecated** ~~Implicitní nullable typy~~

- ⦿ Vždy je nutné uvádět "?" před nullable typem.

// Dříve

```
function example(Type $param = null) {}
```

// Nyní

```
function example(?Type $param = null) {}
```

PHP 8.5 - (11/2025)

- Podpora Closures v konstantách.

```
class Config {  
    const DEFAULT_FILTER = fn($value) => trim(strip_tags($value));  
}
```

- Rozšířený error tracking.
- Převod dalších funkcí na objektový přístup
(např. Directory).

PHP 9+ - Plánované změny

- TypeError při inkrementaci/dekrementaci na stringu.
- Obecně přísnější typové kontroly (`$a = false; $a[] = 2`).
 - `strlen(null)` => `TypeError`
- Odebrání interpolace řetězců skrz zápis `${}`.
- Některé warn budou nově fatal errors (`echo $undefined`).
- Odebrání *Serializable* interface.
- Počátky odchodu od "do-it-all" funkcí.
 - Např.: rozdělení `array_keys` na funkce `array_keys` a `array_keys_filter`.

PHP 9+ - Plánované změny

- TypeError při inkrementaci/dekrementaci na stringu.
- Obecně přísnější typové kontroly (`$a = false; $a[] = 2`).
 - ⊙ `strlen(null) => TypeError`
- Odebrání interpolace řetězců skrz zápis `${}`.
- Některé warn budou nově fatal errors (`echo $undefined`).
- Odebrání *Serializable* interface.
- Počátky odchodu od "do-it-all" funkcí.
 - ⊙ Např.: rozdělení `array_keys` na funkce `array_keys` a `array_keys_filter`.

Objektové PHP



OOP - Třída vs. Objekt

- **Třída** - definuje strukturu a chování.
- **Objekt** - je konkrétní instance třídy s vlastním stavem.

```
class User {  
    public string $name;  
  
    public function sayHello(): void {  
        echo "Hello, " . $this->name;  
    }  
}
```

```
$user = new User();  
$user->name = "Anna";  
$user->sayHello(); // Hello, Anna
```

OOP - Vlastnosti a metody

- **Vlastnost** - proměnná v objektu.
- **Metoda** - funkce v objektu.

```
class Product {  
    public float $price;  
  
    public function getPriceWithVat(): float {  
        return $this->price * 1.21;  
    }  
}
```

OOP - Viditelnost

- Určuje přístupnost prvků (atributy a metody) třídy.
 - ⊙ **public** - prvek je viditelný pro všechny.
 - ⊙ **protected** - prvek je viditelný pro třídu a její potomky.
 - ⊙ **private** - prvek je viditelný pouze v konkrétní třídě.

OOP - Dědičnost

- Umožňuje rozšiřování tříd o rodičovské chování.
- Pro dědičnost se obecně používá direktiva **extends**.

```
class Animal {  
    public function speak() {  
        echo "Some sound";  
    }  
}
```

```
class Dog extends Animal {  
    public function speak() {  
        echo "Woof!";  
    }  
}
```

```
$dog = new Dog();  
$dog->speak(); // Woof!
```

OOP - Abstraktní třídy

- **Třída**, která nemůže být instancována přímo.
- Z pravidla slouží jako rodič konkrétních implementací.
- Může obsahovat **abstract** funkce, které je nutné v potomcích implementovat.

```
abstract class Shape {  
    abstract public function getArea(): float;  
}
```

OOP - Interface

- Slouží pro definici rozhraní, které třída musí implementovat.
- Často se využívá jako "zástupný" typ, který nevyžaduje žádnou další implementaci.

```
interface Loggable {  
    public function log(string $message): void;  
}
```

OOP - Polymorfismus

- Různé funkce se mohou chovat různě v různých potomcích.
- Lze využívat pro předávání objektů za pomoci rodičovských tříd, abstraktních tříd, nebo interface.

```
function notify(Loggable $logger) {  
    $logger->log("Done");  
}
```

OOP - Konstruktor a destruktork

- Umožňuje definovat kroky, které se mají provést při vytváření a zániku instance třídy.
- CTOR()** se využívá např. pro nastavení readonly vlastností, init akcí...
- DTOR()** se využívá např. pro uvolnění alokované paměti, zavření FD...

```
class FileHandler {  
    public function __construct() {  
        echo "Opened";  
    }  
  
    public function __destruct() {  
        echo "Closed";  
    }  
}
```

OOP - Statické metody a vlastnosti

- Tyto prvky jsou shodné pro všechny instance dané třídy a zároveň se dají použít i bez existující instance.
- Běžně se užívají např. u helper/utils tříd, kdy není potřeba přímo vytvářet instanci.

```
class Math {  
    public static function add($a, $b) {  
        return $a + $b;  
    }  
}
```

```
echo Math::add(3, 4); // 7
```

OOP - Trait

- Nástroj pro sdílení kusů kódu (v JS mixiny).
- Nelze použít v rámci *instanceof* ani jako type-hint.
- Lze využívat `class_uses()` pro ověření implementace traitu.
- Problém mezi traity, které mají shodné metody/atributy.
- Problematická práce s *\$this*.
- **Nejedná se o náhradu za dědičnost!**

OOP - Trait

```
trait LoggerTrait {  
    public function log(string $message): void {  
        echo "[LOG] " . $message . PHP_EOL;  
    }  
}
```

```
class FileUploader {  
    use LoggerTrait;  
  
    public function upload() {  
        // ...  
        $this->log("Soubor nahrán.");  
    }  
}
```

```
class EmailSender {  
    use LoggerTrait;  
  
    public function send() {  
        // ...  
        $this->log("E-mail odeslán.");  
    }  
}
```


Znalosti pro Laravel



Magické metody tříd

- `__get()` - pro vytváření dynamických getterů
- `__set()` - pro vytváření dynamických setterů
- `__call()` - pro volání dynamických metod (např. QueryBuilder - whereEmail)
- `__callStatic()` - pro volání dynamických statických metod
- `__toString()` - pro převod objektu na string
- `__serialize()` - pro serializaci objektu
- `__invoke()` - pro vytváření "funkčních" objektů - často užívané pro predikáty
- `__isset()` - pro ověření, že existuje dynamický atribut
- `__unset()` - pro dynamické odebrání dat

Magické metody tříd

```
class SmartBox
{
    private array $data = [];

    public function __set(string $name, mixed $value): void
    {
        $this->data[$name] = $value;
    }

    public function __get(string $name): mixed
    {
        return $this->data[$name] ?? null;
    }

    public function __toString(): string
    {
        return json_encode($this->data, JSON_PRETTY_PRINT);
    }

    public function __invoke(): string
    {
        $count = count($this->data);
        $keys = implode(', ', array_keys($this->data));
        return "SmartBox holds $count item(s): [$keys]";
    }
}
```

```
$box = new SmartBox();

$box->name = 'Testovací uživatel';
$box->email = 'test@example.com';
$box->active = true;

echo $box->name . PHP_EOL;           // Testovací uživatel
echo $box . PHP_EOL;                 // JSON reprezentace
echo $box() . PHP_EOL;               // SmartBox holds 3 item(s): [name, email, active]
```

Strukturalizace a destrukturalizace

- **compact()** - strukturalizace proměnných do asociativního pole.
- **extract()** - destrukturalizace asociativního pole do proměnných.
 - ⊙ Běžně se nepoužívá - na rozdíl od compact hrozí přepsání proměnných při neznalosti kompletní destrukturalizované struktury pole.
 - ⊙ Bezpečnostní riziko - např. při použití v kombinaci s ***\$_GET***, ***\$_POST***...
 - ⊙ Ignoruje číselné klíče a klíče začínající číslem.
 - Vývojář se o tom nijak nedozví.
 - Proměnné ***\$0*** nebo ***\$123hodnota*** atd. v PHP nejsou povolené.

Strukturalizace a destrukturalizace

Strukturalizace

```
$name = 'Anna';  
$age = 30;  
  
$data = compact('name', 'age');  
  
print_r($data);  
// Array ( [name] => Anna [age] => 30 )
```

Destrukturalizace

```
$data = ['name' => 'John', 'age' => 42];  
  
extract($data);  
  
echo "$name is $age"; // John is 42
```

Namespace

- Základní prostředek pro organizaci kódu v moderním PHP.
- Nutnost u frameworků postavených na PSR-4.
- Nástroj pro řešení kolizních názvů mezi třídami - např. více tříd
Logger v různých namespaces.
- Pokud soubor obsahuje `namespace { ... }`, musí být všechny kód
uveden pod namespace!

Namespace - základní použití

- globální namespace - `new \Exception();`
- deklarace - `namespace App\Services;`
- definování třídy - `use App\Services\MyClass;`
- definování funkce - `use function App\foo;`
- definování konstanty - `use const App\VERSION;`

Základy PSR-4 a nástroj Composer



Co je PSR?

- PSR = PHP Standards Recommendation
 - ⊙ Sada doporučení (standardů) vydávaných skupinou PHP-FIG (<https://www.php-fig.org>).
 - ⊙ PHP-FIG sdružuje vývojáře z PHP projektů - Symfony, Laravel, Zend, Doctrine, atd.
 - ⊙ Než se něco přidá do standardu je o tom hlasováno uvnitř PHP-FIG.
- Cílem PSR je zavést společné konvence a pravidla, aby:
 - ⊙ bylo možné snadněji používat frameworky v kombinaci s knihovnami.
 - ⊙ vývojáři psali čitelnější a konzistentnější kód.
- PSR nejsou závazné (ale pak může bolet hlava ... 😊).
- Některé PSR již nejsou platné, nebo byly nahrazeny novějšími.
- Mýtus - PSR **NEURČUJE kvalitu kódu**, ale jednotnost implementace.

Nejčastěji používané PSR standardy

PSR	Název	Využití
PSR-1	Basic Coding Standard	Základní pravidla vývoje v PHP (např. pojmenovávání)
PSR-3	Logger Interface	Standard pro logovací rozhraní (LoggerInterface)
PSR-4	Autoloading via namespace	Mapování namespace na adresářovou strukturu (Composer)
PSR-7	HTTP Message Interface	Objektové rozhraní pro HTTP request a response
PSR-11	Container Interface	Rozhraní pro DI kontejnery (get(), has())
PSR-12	Extended Coding Style Guide	Rozšířená pravidla pro psaní PHP kódu.

Co je Composer?

- Nástroj pro správu balíčků a závislostí v PHP.
- Umí stahovat knihovny/balíčky z packagist.org .
- Automaticky řeší verze a závislosti mezi knihovnami.
- Při inicializaci projektu sestavuje základní autoloader, který:
 - ⦿ automaticky načítá všechny PSR-4 třídy ze složky `src/`.
 - ⦿ automaticky sestavuje mapy tříd třetích stran.
- Definuje 2 základní soubory:
 - ⦿ `composer.json`
 - ⦿ `autoload.php`



Základní příkazy

Příkaz	Akce
composer init	Inicializace projektu užívajícího composer (composer.json)
composer install	Instalace závislostí z composer.lock
composer update	Aktualizace závislostí podle composer.json, aktualizuje lock
composer require vendor/pkg	Přidá závislost do projektu a aktualizuje composer.json
composer require --dev vendor/pkg	Přidá vývojovou závislost.
composer remove vendor/pkg	Odebere balíček a aktualizuje composer.json a lock
composer dump-autoload	Vygeneruje nový soubor autoloader.php
composer show	Zobrazí všechny nainstalované balíčky a jejich verze
composer outdated	Ukáže balíčky, které mohou být aktualizovány
composer validate	Zkontroluje syntaxi souboru composer.json
composer install --no-dev	Instalace bez dev závislostí (např. pro produkční použití).

Základní composer.json

Nový projekt

composer init

Přidání balíčku

composer require nesbot/carbon

```
{  
  "name": "moje/aplikace",  
  "require": {  
    "nesbot/carbon": "^2.0"  
  },  
  "autoload": {  
    "psr-4": {  
      "App\\": "src/"  
    }  
  }  
}
```

Soubor - composer.json

● Identita balíčku

- ⦿ **name** - název balíčku (moje/aplikace)
- ⦿ **description** - popis balíčku/projektu
- ⦿ **version** - číslo aktuální verze
- ⦿ **type** - typ balíčku (library, project, plugin)
- ⦿ **keywords** - klíčová slova pro vyhledávání
- ⦿ **license** - Typ licence (MIT, GPL-3.0, ...)
- ⦿ **homepage** - odkaz na domovskou stránku projektu
- ⦿ **support** - odkaz na issues, tracker, wiki, atd.
- ⦿ **authors** - seznam autorů



Soubor - composer.json

○ Závislosti

- ⦿ **require** - nutné závislosti pro běh aplikace
- ⦿ **require-dev** - vývojové závislosti (testy, debug, profiler...)
- ⦿ **conflict** - seznam balíčků, se kterými se nesmí instalovat
- ⦿ **replace** - mapa balíčků, které tento balíček nahrazuje
- ⦿ **provide** - balíčky, které jsou skrz tento balíček poskytovány
- ⦿ **suggest** - doporučené doplňky (např. ext-gd knihovna pro obrázky...)



Soubor - composer.json

Autoloading

- ⦿ **autoload** - definice PSR-4/PSR-0/výčet souborů atd.
- ⦿ **autoload-dev** - shodné s **autoload**, ale pouze pro dev prostředí
- ⦿ **scripts** - definice vlastních hooků a příkazů - např. post-install

Konfigurace & meta

- ⦿ **config** - nastavení composeru - <https://getcomposer.org/doc/06-config.md>
- ⦿ **bin** - sada vlastních spustitelných souborů
- ⦿ **archive** - nastavení exportu balíčku (např. co ignorovat, jak balíček pojmenovat...)
- ⦿ **scripts-descriptions** - popisy vlastních scriptů (uvedených v scripts)
- ⦿ **scripts-aliases** - aliasování scriptů - např. pro lepší zapamatovatelnost



Zjednodušená autoload funkce dle PSR-4

```
spl_autoload_register(function (string $class) {  
    $prefix = 'App\\';  
    $baseDir = __DIR__ . '/src/';  
  
    $len = strlen($prefix);  
    if (strncmp($prefix, $class, $len) !== 0) {  
        return;  
    }  
  
    $relativeClass = substr($class, $len);  
    $file = $baseDir . str_replace('\\\\', '/', $relativeClass) . '.php';  
  
    if (file_exists($file)) {  
        require $file;  
    }  
});
```

Laravel Framework



Co je Laravel Framework?

- Moderní MVC PHP framework pro vývoj webových aplikací.
- Elegantní syntaxe, čitelný kód a přehledná dokumentace.
- Hotové nástroje pro běžné potřeby - routing, validace, práce s databází, autentizace...
- Komplexní ekosystém nástrojů - Blade, Sanctum, Artisan CLI, Breeze a další.
- Staví na moderním PHP stacku - Composer, PSR-4, DI, eventy, testing.
- Aktivní vývoj a silná komunita vývojářů.

Porovnání s konkurencí



Vlastnost	Laravel	Symfony	Nette
Architektura	MVC, convention-based	MVC, config-first	MVC-ish, DI-heavy
Syntaxe	Čistá, přehledná	Explicitní, robustní	Vlastní "freestyle"
Komunita	Obrovská - globální přesah	Velká (Evropa, enterprise řešení)	Malá - převážně CZ/SK
Popularita (dle github)	#1	#2	"jednička" v ČR (?)...
Dokumentace	Laracast, robustní web dokumentace	Obsáhlá web dokumentace - málo příkladů	Existuje
Založení projektu	Starter kit	Starter Kit / DIY přístup	DIY přístup
Testování	PHPUnit, Pest	PHPUnit, Symfony Test Tools	Tester (vlastní)

- MVC - Model - View - Controller
- DIY - "Do it yourself"
- DI-heavy - vše se řeší skrz DI...

Nette má takový český přístup:

"Já vím, jak to dělat nejlépe a ty se podříd!"

Shrnutí filozofie porovnávaných Frameworků

● Laravel

- ⊙ Nabízí hotové řešení s důrazem na jednoduchost a produktivitu.

● Symfony

- ⊙ Dává vývojářům plnou kontrolu skrze konfiguraci a rozšiřitelnost.

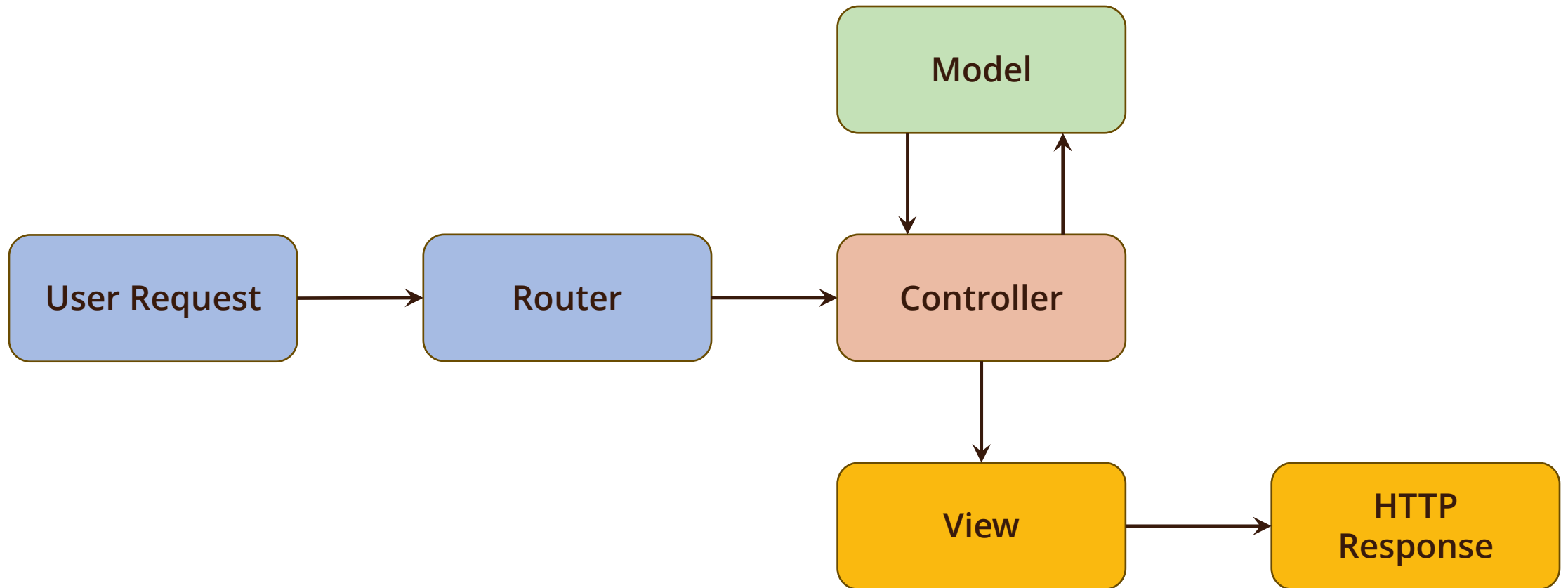
● nette FRAMEWORK

- ⊙ Prosazuje názorový přístup s důrazem na čistý návrh a strukturu aplikace.

Architektura MVC

- Moderní architektura, která odděluje logiku, zobrazení a vstup uživatele.
- Rozdělení zodpovědností:
 - ⊙ **Model** - Pracuje s daty (např. databáze, validace, business logika).
 - ⊙ **View** - Zodpovídá za vzhled - šablony, HTML, prezentace pro uživatele, API.
 - ⊙ **Controller** - Řídí tok aplikace - přijímá vstupy, volá modely a vrací view (response).
- Výhody MVC:
 - ⊙ Oddělení zodpovědností = přehledný kód.
 - ⊙ Snažší údržba a lehčí testování.
 - ⊙ Týmy mohou pracovat paralelně (např. frontend vs. backend).

Model architektury MVC v Laravelu



Inversion of Control (IoC)

- Běžně používaný model vytváření závislostí.
- Představuje "obrácené řízení" - třída si závislosti nevytváří, ale dostává je zvenčí.
- Laravel si sestavuje vlastní IoC kontejner (Service Container) a pomocí tohoto kontajneru předává vhodné závislosti.
 - ⦿ Kontejner ví, jak jednotlivé třídy vytvořit - vč. jejich závislostí a v jakém pořadí.
 - ⦿ Dovoluje manuální registraci dalších služeb.
 - ⦿ Zvládá binding konkrétních tříd, interface, singletonů, nebo Closures.

Dependency Injection (DI)

- Způsob, jakým se závislosti třídám dodávají.
- Laravel umožňuje pouze 2 metody injektování závislostí:

- ⊙ **Constructor Injection**

```
public function __construct(ReportService $service) {  
    $this->service = $service;  
}
```

- ⊙ **Method Injection** - pouze pro kontrolery, middleware, service apod.

```
public function show(Request $request, ReportService $service) {  
    // Laravel tohle všechno injectne  
}
```

Laravel Service Container

- Centrální registr pro třídy, instance tříd a služby - mozek DI.
- Laravel zde spravuje - závislosti, singletony, interface a instance, vytvořené na míru (manuálně registrované).
- Základní metody
 - ⦿ `bind()` - vytvoření factory na sestavení instance konkrétní třídy na požádání
 - ⦿ `singleton()` - vytvoření instance třídy dle factory pro sdílenou instanci
 - ⦿ `instance()` - zaregistruje vytvořenou instanci, která je vždy vrácena
 - ⦿ `__invoke()` - volání pro získání instance konkrétní třídy
 - ⦿ `make()` - alias k `__invoke()`

Laravel Service Container

Nové instance

```
$app->bind(Foo::class, function () {  
    return new Foo(Str::random());  
});  
  
$one = app(Foo::class);  
$two = app(Foo::class);  
  
var_dump($one === $two); // false ✅ nová instance
```

Singletony

```
$app->singleton(Foo::class, function () {  
    return new Foo(Str::random());  
});  
  
$one = app(Foo::class);  
$two = app(Foo::class);  
  
var_dump($one === $two); // true ✅ sdílená instance
```

Instalace Laravel Frameworku



Příprava prostředí

- GIT: <https://git.koduj.dev> - projekt PHP Sandbox
- Stažení pomocí `$ git clone https://github.com/koduj-dev/php-sandbox.git`
- Prostředí obsahuje:
 - ⦿ **PHP 8.3** s předinstalovaným composerem
 - ⦿ **Nginx** s připravenou konfigurací pro běh serveru na portu 8080
 - ⦿ **MySQL** 8.0.40 se základním nastavením
- Spuštění kontainerů
 - ⦿ `$./docker/bin/up.sh`

Instalace Laravel Frameworku

- Připojení k docker containeru (php_sandbox_php_1)
 - ⦿ `$ docker exec -it php_sandbox_php_1 bash`
- Instalace Laravel instalátoru
 - ⦿ `$ composer global require laravel/installer`
- Vytvoření projektu skrz instalátor
 - ⦿ `$ laravel new project` // lze uvést volitelný název, dockery mají ale konfiguraci pouze pro "project"
- Fix oprávnění v dockeru
 - ⦿ `$ chown -R www-data:www-data /php_sandbox`
 - ⦿ `$ chown -R mysql:mysql /var/lib/mysql` // kontainer php_sandbox_mysql_1

Základní struktura projektu

project/

— app/	Vlastní aplikační kódy (Commandy, Model, Kontrolery, Middleware...)
— bootstrap/	Spouštěcí kód aplikace a autoloader
— config/	Konfigurace aplikace
— database/	Migrace, factory, seedery
— public/	Webroot - index.php, assety, obrázky, atd.
— resources/	View - Layouty, Blade šablony
— storage/	Logy, cache, nahrané soubory
— tests/	Testy (PHPUnit / Pest)
— vendor/	Composer balíčky a závislosti
— .env	Lokální konfigurační soubor
— artisan	CLI nástroj Laravel Artisan
— composer.json	Konfigurace pro Composer

Laravel Artisan CLI

- Vestavěný příkazový řádek Laravelu
- Spuštění - `$ php artisan`
- Základní ovládání & hlavní příkazy

Příkaz	Popis akce
<code>php artisan</code>	Zobrazení všech dostupných příkazů
<code>php artisan help make:model</code>	Zobrazení nápovědy pro konkrétní příkaz
<code>php artisan route:list</code>	Zobrazení seznamu registrovaných rout
<code>php artisan tinker</code>	Interaktivní PHP konzole s Laravel aplikací
<code>php artisan migrate</code>	Spuštění databázových migrací
<code>php artisan make:</code>	Zobrazení všech dostupných maker příkazů

Laravel Artisan CLI - Maker extension

- Uvedené příkazy vždy začínají `$ php artisan`

Příkaz	Vytvoří...
<code>make:model Todo</code>	Model třídy "Todo"
<code>make:controller TodoController</code>	Controller "TodoController"
<code>make:migration create_todos_table</code>	Vytvoří soubor pro migraci s vytvořením tabulky "todos"
<code>make:seeder TodoSeeder</code>	Seeder pro model Todo
<code>make:factory TodoFactory</code>	Factory pro vytváření Todo
<code>make:request StoreTodoRequest</code>	Form request (validační komponenta) pro Todo

- Některé make příkazy podporují přepínače - např. `make:model`

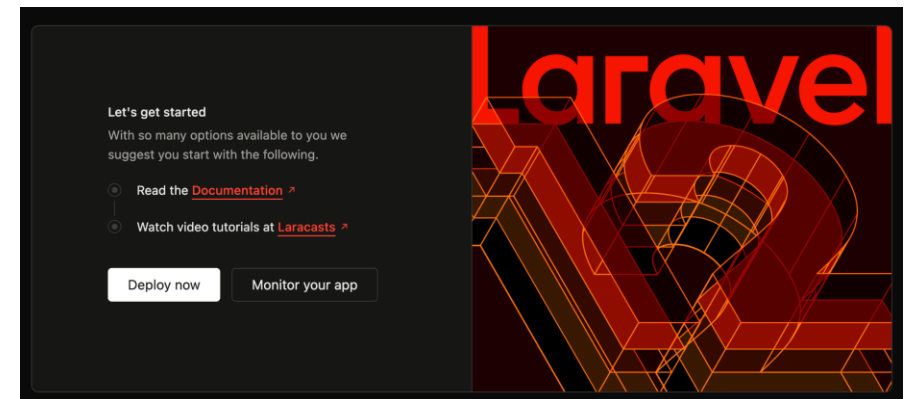
⊙ `$ php artisan make:model Todo -m -f -s -c -r` // vytvoří Model, Factory, Seeder, Controller, ResourceController

Laravel Artisan CLI - Další příkazy

Příkaz	Popis akce
php artisan about	Informační shrnutí informací o projektu
php artisan schedule:	Příkazy týkající se plánovače (CRONY)
php artisan db:	Příkazy pro práci s databází
php artisan event:	Příkazy pro práci s eventy
php artisan route:	Příkazy pro práci s routami
php artisan [down up]	Přepínání aplikace do/z maintenance režimu

Instalace Laravel Frameworku

- Inicializace DB po instalaci:
 - ⦿ `$ php artisan session:table`
 - ⦿ `$ php artisan migrate`
- V prohlížeči: <http://localhost:8080>
- Kontrola funkčnosti: <http://localhost:8080/up>



Debugging Laravel aplikace



Základní debug funkce

- `dump()` - vypsání proměnné/instance na obrazovku.
- `dd()` - kombinace `dump()` a `die()`.
 - ⊙ Většina fluent příkazů Laravelu podporuje volání `route()->dd();`
 - ⊙ Lze použít i v Blade šablonách pomocí direktivy `@dd($var)`
- Obě funkce lze používat kdekoliv - controller, view, command, seeder...

Laravel Debugbar

- Instalace:
 - ⦿ `$ composer require barryvdh/laravel-debugbar --dev`
 - ⦿ Instaluje se jako `--dev` závislost, nelze ho používat v produkci.
- Zero-config - aktivuje se automaticky s requestem, nevyžaduje DB.
- Podmínka pro zobrazení je pouze ta, že musí být vráceno View.
- Co umí:
 - ⦿ SQL dotazy a časování
 - ⦿ Requesty, session, routy
 - ⦿ Přihlášení uživatelé, eventy, service container, view data, atd.
- Lze ho vypnout v `.env` pomocí "`DEBUGBAR_ENABLED=false`"

Laravel Telescope

- Oficiální nástroj Laravelu pro detailní sledování chování aplikace.
- Instalace:
 - ⦿ `$ composer require laravel/telescope --dev`
 - ⦿ `$ php artisan telescope:install` // instalace routingu + views
 - ⦿ `$ php artisan migrate` // vytvoření DB
- Dostupnost:
 - ⦿ <http://localhost:8080/telescope>
- Umožňuje sledovat:
 - ⦿ Requesty, Exceptiony, Logy
 - ⦿ SQL dotazy
 - ⦿ Eventy, joby, notifikace, Maily, cache, queue a další...

Routing & Contollery



Routing - co to je?

- Mechanismus, který určuje jak se odpoví na konkrétní HTTP požadavek.
 - ⊙ Router převádí URL na konkrétní logiku aplikace - např. metodu Controlleru.
- Umožňuje definovat HTTP metody GET, POST, PUT, DELETE, atd.
- V Laravelu pak lze definovat routy s:
 - ⊙ parametry
 - ⊙ obslužnými middleware
 - ⊙ jmény
 - ⊙ validací
 - ⊙ zabezpečením
- Definice pro router jsou ve složce **routes/**/***

```
Route::get('/hello', function () {  
    return 'Ahoj světe!';  
});
```

route() - základní metody

- `route()` - samo o sobě vrací Facade Router
- `::get()` - vytvoření HTTP GET routy
- `::post()` - vytvoření HTTP POST routy
- `::put()` - vytvoření HTTP PUT routy
- `::patch()` - vytvoření HTTP PATCH routy
- `::delete()` - vytvoření HTTP DELETE routy
- `::options()` - vytvoření HTTP OPTIONS routy
- `::match()` - vytvoření routy s více metodami
- `::any()` - vytvoření obecné routy, která akceptuje vše

Základní metody - užití

- Route s Closure

```
Route::get('/hello', function () {  
    return 'Ahoj světe!';  
});
```

- Implicitní routy (`__invoke`)

```
Route::get('/about', AboutController::class);
```

- Explicitní routy

```
Route::get('/contact', [ContactController::class, 'show']);
```

- Matching routy (více metod)

```
Route::match(['get', 'post'], '/form', [FormController::class, 'handle']);
```

route() - pokročilé metody

- ◉ `::redirect()` - přesměruje z routy A -> B s HTTP 302 Found
- ◉ `::permanentRedirect()` - přesměruje z routy A -> B s HTTP 301 Moved Permanently
- ◉ `::view()` - definice routy pro konkrétní View
- ◉ `::controller()` - definice skupiny rout pro konkrétní Controller
- ◉ `::prefix()` - definice skupiny rout se společným URL prefixem (např. /todos/...)
- ◉ `::name()` - definice skupiny rout se společným name prefixem
- ◉ `::fallback()` - definice "404" routy pro router/skupinu rout
- ◉ `::group()` - vytvoření skupiny rout - pro skupinové metody

Pokročilé metody - užití

- Definice s `middleware()`

```
Route::middleware(['auth'])->group(function () {  
    Route::get('/profile', [ProfileController::class, 'show']);  
    Route::post('/profile', [ProfileController::class, 'update']);  
});
```

- Definice s `prefix()`

```
Route::prefix('admin')->group(function () {  
    Route::get('/users', [AdminUserController::class, 'index']);  
    Route::get('/settings', [AdminController::class, 'settings']);  
});
```

- Definice s `controller()`

```
Route::controller(DashboardController::class)->group(function () {  
    Route::get('/dashboard', 'index');  
    Route::get('/dashboard/settings', 'settings');  
});
```

- Definice s `name()`

```
Route::name('admin.')->group(function () {  
    Route::get('/users', [AdminController::class, 'users'])->name('users');  
    Route::get('/logs', [AdminController::class, 'logs'])->name('logs');  
});
```

Pokročilé metody - užití

- Pokročilé metody definice rout lze snadno kombinovat

```
Route::middleware('auth')
->prefix('account')
->name('account.')
->controller(AccountController::class)
->group(function () {
    Route::get('/', 'dashboard')->name('dashboard');    // /account → account.dashboard
    Route::get('/settings', 'settings')->name('settings'); // /account/settings → account.settings
});
```

Pokročilý routing - parametrizace

- Pro předávání proměnných v rámci routy, lze definovat povinné a nepovinné parametry přímo v url routy.
 - ⊙ Nepovinné parametry se označují `{..?}`, musí být vždy na posledním místě!
- Eloquent rozšíření pak pomocí typů umožňuje přímí binding modelu do funkce routy.

```
Route::get('/users/{id}', [UserController::class, 'show']);  
Route::get('/category/{slug?}', [CategoryController::class, 'show']);
```

Pokročilý routing - parameter expressions

- `::where()` - definice parametru pomocí regulárního výrazu
 - `::whereNumber()` - definice parametru jako čísla
 - `::whereAlphaNumeric()` - definice parametru jako alfa numerického výrazu
 - `::whereIn()` - omezení parametru na konkrétní hodnoty
 - `::whereUuid()` - omezení parametru na formát UUID
 - `::whereUlid()` - omezení parametru na formát ULID (časově řaditelný identifikátor)
 - `::pattern()` - definice globální regex pattern platný pro celou aplikaci
- Ideální definovat v `RouteServiceProvider::boot()`

Pokročilý routing - parameter expressions

// Klasický where() s regulárním výrazem

```
Route::get('/products/{slug}', [ProductController::class, 'show'])
    ->where('slug', '[a-z0-9-]+');
```

// Číselný parametr (jen čísla)

```
Route::get('/orders/{id}', [ProductController::class, 'order'])
    ->whereNumber('id');
```

// Alfnumerický parametr

```
Route::get('/users/{username}', [ProductController::class, 'user'])
    ->whereAlphaNumeric('username');
```

// Parametr omezený na konkrétní hodnoty

```
Route::get('/status/{state}', [ProductController::class, 'status'])
    ->whereIn('state', ['active', 'archived', 'deleted']);
```

Eloquent - ORM



ORM - proč ho (ne)chtít?

- ORM = Objektově relační mapování - most mezi relační databází a OOP světem.
- Proč ho chtít?
 - ⊙ Není potřeba znát SQL - pracuje se pouze s PHP objekty.
 - ⊙ Integruje automatickou validaci dat, kontrolu typů a řeší relace.
 - ⊙ DRY princip - vztahy a logika se definuje pouze jednou v modelech.
- Proč ho někdy nechtít?
 - ⊙ ORM je "těžký" black-box - často vytváří neefektivní dotazy.
 - ⊙ Způsobuje problém N+1.
 - ⊙ Na komplexní joiny a pokročilou optimalizaci se vždy nehodí.

Základní relační vazby - 1:N

- 1:N = jedna instance entity **A** je spojena s N instancemi entity **B**.
 - ⊙ Spojení je v 3.NF uváděno na straně entity **B**, která drží odkaz na rodiče.
- N:M = N instancí entity **A** je spojeno s M instancemi entity **B**.
 - ⊙ Spojení je realizováno pomocí spojovací tabulky (v Laravelu Pivot Table).
 - ⊙ Relace může být v rámci spojovací tabulky doplněna o další extra data.
- 1:1 = jedna instance entity **A** je spojena s jednou instancí entity **B**.
 - ⊙ Speciální 1:N relace.
 - ⊙ Nedoporučuje se používat - může způsobovat deadlocky v databázi (např. při mazání).
 - ⊙ Pokud není relace NULL, tak ji nelze bez vypnutí cizích klíčů ani realizovat.

Eloquent

- Oficiální ORM implementace v Laravelu.
- Data reprezentuje pomocí modelových souborů.
- Používá databázovou konvenci *snake_case*.
- Přístup k datům je realizován pomocí magic metod `__get()` a `__set()`.
- Entita a její model jsou postaveny na architektuře *Active Record*.
 - ⦿ Třída entity reprezentuje konkrétní tabulku.
 - ⦿ Instance entity pak reprezentuje kompletní model s veškerou CRUD logikou.

Eloquent - Metody modelové třídy (::)

Metoda třídy	Popis
::all()	Vrátí všechny záznamy
::find(<i>\$id</i>)	Najde záznam podle primárního klíče
::where(...)	Vrátí záznamy odpovídající podmínce
::first()	Vrátí první záznam
::create([...])	Vytvoří a uloží novou instanci
::make([...])	Vytvoří novou instanci třídy (neuloženou)
::query()	Vytvoří query builder
::with(...)	Načte vztahy pomocí eager loadingu
::pluck('col')	Vrátí hodnoty jednoho sloupce
::count()	Vrátí počet výsledků

Eloquent - Metody instance objektu (->)

Metoda třídy	Popis
<code>save()</code>	Uloží model (insert nebo update)
<code>delete()</code>	Smaže záznam
<code>update([...])</code>	Hromadně aktualizuje data
<code>refresh()</code>	Přenačte data z DB
<code>replicate()</code>	Vytvoří kopii instance (bez ID)
<code>wasChanged()</code>	Zjistí, jestli se něco změnilo
<code>isDirty('field')</code>	Zjistí, jestli se změnilo konkrétní pole
<code>getAttribute('name')</code>	Získá hodnotu atributu
<code>setAttribute('name', \$val)</code>	Nastaví hodnotu atributu
<code>"relation"</code>	Přístup k relaci (<code>\$book->author</code>)

Eloquent - Model

○ Základní prvky

- ⦿ `$table` = manuální určení tabulky
- ⦿ `$fillable` = pole atributů modelu
- ⦿ `$casts` = seznam konverzí atributů
- ⦿ `$attributes` = výchozí hodnoty
- ⦿ `$primaryKey` = název primárního klíče
- ⦿ `$keyType` = typ klíče
- ⦿ `$incrementing` = flag jestli je PK autoincrement

○ Vytvoření modelu:

- ⦿ `$ php artisan make:model Book`

```
class Book extends Model
{
    use HasFactory;

    protected $fillable = [
        'title',
        'author_id',
        'published_at',
    ];

    protected $casts = [
        'published_at' => 'date',
    ];

    public function author()
    {
        return $this->belongsTo(Author::class);
    }

    public function getPublishedYearAttribute(): ?int
    {
        return $this->published_at->year;
    }
}
```


Eloquent - Model

○ Relační metody

- ⊙ **hasOne()** - mám právě jeden druhý model
- ⊙ **belongsTo()** - patřím jinému modelu
- ⊙ **hasMany()** - mám více druhých modelů
- ⊙ **belongsToMany()** - mám mnoho jiných modelů skrz pivot tabulku

```
// Author.php
class Author extends Model
{
    public function books()
    {
        return $this->hasMany(Book::class);
    }
}

// Book.php
class Book extends Model
{
    public function author()
    {
        return $this->belongsTo(Author::class);
    }
}
```

Eloquent - Polymofrní vztahy

- Řešení, kdy jedna entita může mít vazby na různé entity.
 - ⊙ Např. "Komentář" + "Video" a "Komentář" + "Obrázek".
- Na straně DB je to řešeno 2 sloupci - ID modelu a typ, kterému vztah patří.
- Polymorfní metody
 - ⊙ `morphTo()` - univerzální zpětný vztah
 - ⊙ `morphMany()` - model má více polymorfních sub modelů
 - ⊙ `morphToMany()` - model má více polymorfních vztahů přes pivot tabulku (M:N)
 - ⊙ `morphedToMany()` - druhá strana polymorfního M:N.

Eloquent - Polymofrní vztahy

```
class Comment extends Model
{
    public function commentable()
    {
        return $this->morphTo();
    }
}
```

id	body	commentable_id	commentable_type
1	Super článek!	5	App\Models\Post
2	Líbí se mi to!	3	App\Models\Video

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

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

```
$comment->commentable;    // může vrátit Post nebo Video
$post->comments;           // vrátí kolekci komentářů
$video->comments;          // taky
```

Eloquent - Factory (továrničky)

- Nástroj pro vytváření "fakes" (fixtures) instancí modelu.
- Lze definovat různé Factory pro různá prostředí - např. testovací data, load testy atd.
- Používají PHP Faker (<https://fakerphp.org>).
 - ⦿ Lze tak snadno generovat jména, texty, datumy...
 - ⦿ Vhodné i např. pro prezentační ukázky.
- Vytvoření faktory:
 - ⦿ `$ php artisan make:factory BookFactory --model Book`

Eloquent - Factory (továrničky)

```
// database/factories/BookFactory.php

use Illuminate\Database\Eloquent\Factories\Factory;

class BookFactory extends Factory
{
    public function definition(): array
    {
        return [
            'title' => $this->faker->sentence(),
            'published_at' => $this->faker->date(),
            'author_id' => \App\Models\Author::factory(), // vytvoří autora automaticky
        ];
    }
}
```

Eloquent - Seedery

- Nástroj pro plnění databáze testovacími, nebo výchozími daty.
- Seedery mohou být definovány pro různá prostředí.
- Pro testování a vývoj se z pravidla používají *Factory* modelů.
- Seedery lze spouštět opakovaně, vč. vyčištění databáze.
- Dle dělení je lze spouštět po jednom, nebo si připravit seeder, který bude plnit konkrétní sadu dat.
- Vytvoření seederu:
 - ⦿ `$ php artisan make:seeder BookSeeder`

Eloquent - Seedery

- Možnosti spouštění seederů:
 - \$ php artisan db:seed // spuštění všech seederů
 - \$ php artisan db:seed --class=BookSeeder // spuštění konkrétního seederu
 - \$ php artisan migrate:fresh --seed // seed s vyprázdněním databáze

```
use Illuminate\Database\Seeder;  
use App\Models\Book;  
  
class BookSeeder extends Seeder  
{  
    public function run(): void  
    {  
        Book::factory()->count(10)->create();  
    }  
}
```

Eloquent - Migrate

- Migrate slouží ke správě databázové struktury pomocí PHP kódu.
- Definují tabulky, sloupce, indexy, cizí klíče...
- Umožňují verzovat databázi a poskytují případný rollback.
- Migrate by měli být shodné napříč *dev*, *test* a *prod* prostředím.
- Název migrate vždy obsahuje časové razítko, aby bylo jasné pořadí migrací.
- Provedené migrate jsou ukládány do DB, aby nebyly spouštěny znovu.
- Migrate jsou řešeny pomocí anonymních tříd.

Eloquent - Migrate

- Pro cizí klíče Eloquent automaticky vytváří indexy.
- Vytvoření migrace
 - ⦿ `$ php artisan make:migration create_book_table`
- Spuštění migrace
 - ⦿ `$ php artisan migrate // spuštění migrací`
 - ⦿ `$ php artisan migrate:fresh --seed // spuštění migrací, vyčištění databáze a seed dat`

Eloquent - Migrace

```
return new class extends Migration
{
    public function up(): void
    {
        Schema::create('books', function (Blueprint $table) {
            $table->id();
            $table->string('title');
            $table->foreignId('author_id')->constrained()->cascadeOnDelete();
            $table->date('published_at')->nullable();
            $table->timestamps();
        });
    }

    public function down(): void
    {
        Schema::dropIfExists('books');
    }
};
```

Eloquent - Implicitní binding (Model binding)

- DI funkce Laravelu, která dle ID (nebo jiného sloupce) vyhledá entitu a injectne ji do Controlleru, nebo Closure jako instanci.
- Nejběžněji se tato metoda používá v kombinaci s routerem.
- Základní použití - interně se volá metoda `findOrFail($id)`:

```
Route::get('/books/{book}', function (Book $book) {  
    return view('books.show', ['book' => $book]);  
});
```

- Pokud není model nalezen - vrátí Laravel 404.
 - ⦿ Pomocí exception handlingu lze vracet i vlastní 404, nebo reagovat jinak.

Eloquent - Implicitní binding (Model binding)

- Vyhledávání podle sloupce:

```
Route::get('/books/{book:slug}', function (Book $book) {  
    return $book;  
});
```

- Interně je voláno: `Book::where('slug', $value)->firstOrFail()`
- Předávání více Entit s kontrolou vlastnictví:

```
Route::scopeBindings()->group(function () {  
    Route::get('/authors/{author}/books/{book}', function (Author $author, Book $book) {  
        return $book;  
    });  
});
```

Eloquent - Error handling

- Základní error handling pro routu pomocí `missing()`:

```
Route::get('/books/{book}', function (Book $book) {  
    return $book;  
})->missing(function () {  
    return redirect('/books')->with('error', 'Kniha nebyla nalezena.');
```

- Alternativní error handling:

- ⦿ Handler založený na `Illuminate\Foundation\Exceptions\Handler` v `app\Exceptions\Handler`.
- ⦿ Registrace v `app.php` metodou `withBindings([...])`.

Šablonovací systém Blade



Blade - šablonovací systém

- Oficiální šablonovací systém Laravelu.
- Napsaný v čistém PHP - nevyžaduje žádný speciální preprocesor.
- HTML rozšířené o jednoduchou a čitelnou syntaxi.
- Proč ho používat?
 - ⊙ Čistý zápis direktiv `@if`, `@foreach`, `{{ $var }}`
 - ⊙ Bezpečný výstup - automaticky escapuje proměnné.
 - ⊙ Dědičnost a komponenty - snadno lze opakovat části kódu, komponenty zjednodušují organizaci.
 - ⊙ Přímá integrace s Laravelem - routing, assety, CSRF tokeny, formuláře...
 - ⊙ Rychlost - Blade šablony se předem kompilují do nativního PHP a cachují.

Blade - základní šablona 😊

```
<!DOCTYPE html>
<html>
<head>
  <title>Example</title>
</head>
<body>
  <p>This is an example of a simple HTML page with one paragraph.</p>
</body>
</html>
```


Blade - základní šablona

```
<!DOCTYPE html>
<html>
<head>
    <title>Blade Template</title>
</head>
<body>
    <h1>Server info</h1>

    <p><strong>Prohlížeč:</strong> {{ request()->header('User-Agent') }}</p>
    <p><strong>PHP verze:</strong> {{ PHP_VERSION }}</p>
    <p><strong>Datum a čas:</strong> {{ now()->toDayDateTimeString() }}</p>
</body>
</html>
```

Blade - direktivy - Echo statements

- Interpolace
 - ⊙ `{{ $var }}` - výpis proměnné do HTML s interním voláním `htmlspecialchars()`.
 - ⊙ `{{ fn() }}` - výpis výstupu funkce do HTML s interním voláním `htmlspecialchars()`.
 - ⊙ `@{{ ... }}` - označení literálního výstupu - např. pro vyhodnocování v JS (VueJS).
- `{!! $var !!}` - vypsání proměnné (nebo výstupu z funkce) **bez escapingu - nebezpečné (XSS)!**

Blade - direktivy - podmínky

- `@verbatim`, `@endverbatim` - označení části šablony, kterou generuje JS.
- `@if`, `@elseif`, `@else`, `@endif` - klasické rozhodování s podmínkami.
- `@unless`, `@endunless` - negace podmínky pro zjednodušený `@if(!...)`.
- `@isset`, `@endisset` - zkrácený if pro `isset` call.
- `@empty`, `@endempty` - zkrácený if pro `empty` call.
- `@auth`, `@endauth` - zkrácený if pro `@if(Auth::check())`
- `@guest`, `@endguest` - zkrácený if pro `@unlessauth`.
- `@session`, `@endsession` - zkrácený if pro ověření existence hodnoty v session.

Blade - direktivy - switch

- `@switch($var)` - označení začátku switche pro proměnnou `$var`.
- `@case('value')` - konkrétní switch case s hodnotou/podmínkou.
- `@break` - ukončení switch case.
- `@default` - defaultní switch branch.
- `@endswitch` - ukončující direktiva switche.

Blade - direktivy - loops

- `@for`, `@forendfor` - jednoduchý for - např. `@for($i = 0; $i < 10; $i++)`.
- `@foreach`, `@foreach` - foreach pro všechny prvky v poli `@foreach($array as $value)`.
- `@forelse`, `@endforelse` - rozšíření foreach o direktivu `@empty`.
 - ◉ Část `@empty` pak umožňuje definovat šablonu, pokud je vstupní pole prázdné.
- `@while`, `@endwhile` - cyklus while, jehož délku definuje podmínka.
- `@continue` - skipnutí aktuální iterace.
- `@break` - přerušování probíhajícího cyklu.

Blade - direktivy - loops - \$loop variable

- Speciální proměnná uvnitř cyklů, která poskytuje informace o aktuální iteraci.
- Atributy \$loop->:
 - ⊙ **index** - Index aktuální iterace (od 0).
 - ⊙ **iteration** - Počítadlo iterací (od 1).
 - ⊙ **remaining** - Počet zbývajících iterací.
 - ⊙ **count** - Celkový počet iterovaných prvků.
 - ⊙ **first** - Příznak zda se jedná o první iteraci.
 - ⊙ **last** - Příznak zda se jedná o poslední iteraci.
 - ⊙ **even** - Příznak zda jde o "sudou" iteraci.
 - ⊙ **odd** - Příznak zda jde o "lichou" iteraci.
 - ⊙ **depth** - Hloubka aktuálního zanoření (nested loops).
 - ⊙ **parent** - Zpřístupňuje \$loop proměnnou rodičovské smyčky.

Blade - podmíněné třídy a styly

- **@class** - direktiva, která umožňuje vygenerovat dynamický seznam tříd pro html tag.
- **@style** - direktiva, která umožňuje vygenerovat dynamické styly pro html tag.

```
<div @class([
    'alert',
    'alert-success' => $success,
    'alert-danger' => !$success,
])>
    Výsledek akce
</div>
```

```
<span @style([
    'color: red' => $error,
    'font-weight: bold' => $highlighted,
])>
    Text s podmíněným stylem
</span>
```

Blade - rozšiřující direktivy pro formuláře

- **@checked** - přidává k checkboxu "checked" - pokud je uvedená podmínka *true*.
- **@selected** - přidává k select > option tagu "selected" - pokud je uvedená podmínka *true*.
- **@disabled** - přidává k inputu/buttonu "disabled" - pokud je uvedená podmínka *true*.
- **@readonly** - přidává k inputu "readonly" - pokud je uvedená podmínka *true*.
- **@required** - přidává k inputu "required" - pokud je uvedená podmínka *true*.

```
<input type="checkbox" name="agree" @checked($agree)>
<select name="option">
    <option value="1" @selected($selectedId === 1)>Možnost 1</option>
</select>
<input type="text" name="email" @disabled($readonlyMode)>
<input type="text" name="name" @readonly($readonlyMode)>
<input type="text" name="username" @required($isFormRequired)>
```


Blade - subviews

- **@include** - vloží obsah požadovaného view.
- **@includeIf** - vloží obsah požadovaného view, pokud view file existuje.
- **@includeWhen** - vloží obsah požadovaného view, pokud je podmínka *true*.
- **@includeUnless** - vloží obsah požadovaného view, pokud je podmínka *false*.
- **@includeFirst** - vloží první existující view.
- **@once** - označí oblast, která se má vykreslit pouze jednou (např. v komponentách).
- **@each** - zkrácený foreach s @include požadovaného view.

Blade - subviews - @foreach vs. @each

```
<body>
```

```
<!-- Klasický foreach -->
```

```
@foreach ($users as $user)
```

```
    @include('partials.user', ['user' => $user])
```

```
@endforeach
```

```
<!-- Zkrácená verze -->
```

```
@each('partials.user', $users, 'user')
```

```
</body>
```

Blade - layouting

- **@yield** - označení místa v rodičovské šabloně, které očekává obsah z potomka.
- **@section** - označení části v potomkovi, které se předá rodiči do **@yield**.
- **@extends** - určuje, že šablona bude potomkem vybrané šablony (často layoutu).
- **@parent** - získání původního obsahu sekce v potomkovi (např. pro spojování obsahu).
- **@push** - přidání dat do pojmenovaného stacku.
- **@stack** - výpis pojmenovaného stacku (např. pro skripty, breadcrumb atd.).
- **@prepend** - přidání dat na začátek pojmenovaného stacku.

Blade - layouting

layout.blade.php

```
<html>
<head>
    <title>@yield('title', 'Výchozí titul')</title>
</head>
<body>
    <header>
        <h1>Moje aplikace</h1>
    </header>

    <main>
        @yield('content', 'Zatím žádný obsah')
    </main>

    <footer>
        <p>&copy; 2025</p>
    </footer>
</body>
</html>
```

homepage.blade.php

```
@extends('layout')

@section('title', 'Domů')

@section('content')
    <h2>Vítej zpět, {{ $user->name }}!</h2>
    <p>Toto je tvoje domovská stránka.</p>
@endsection
```

Blade - komponenty

- Opakovaně použitelné "šablony" - jednotná definice vzhledu skrz projekt.
- Komponenta je obyčejná Blade šablona, která obsahuje pouze vlastní HTML + obslužný kód.
- Pro lepší plnění komponent lze využívat sloty.
- Vytvoření komponenty:
 - ⦿ `$ php artisan make:component Card`
 - ⦿ Vytvoří 2 základní soubory:
 - `views/components/card.blade.php` - samotná šablona komponenty.
 - `app/View/Components/Card.php` - třída pro implementaci logiky komponenty.
 - ⦿ komponentu pak lze použít jako `<x-card>...</x-card>`

Blade - komponenty - sloty

- Sloty umožňují definovat části v komponentách.
- Často se využívají v kombinaci s CSS frameworky pro předpřipravené designové prvky.
- `<x-slot>` - nepojmenovaný defaultní slot vykresluje se skrz `{{ $slot }}`.
- `<x-slot:name>` - pojmenovaný slot, vykresluje se skrz `{{ $name }}`.
- `@props` - definuje proměnné pro atributy komponenty.

```
@props(['type' => 'info'])
```

```
<div class="card card-{{ $type }}" <← <x-card type="warning">
  <h2>{{ $title }}</h2> <← <x-slot:title>Upozornění</x-slot:title>

  <div class="card-body">
    {{ $slot }} <← <p>Uživatel {{ $user->name }} má neuložené změny.</p>
  </div>
</div> </x-card>
```

Validace a formuláře



Formuláře

- Laravel přímo nenabízí, jako ostatní frameworky, typy pro práci s formuláři.
- Poskytuje pouze nástroje, pro zjednodušení obsluhy formulářů.
- Dodržuje striktně principy HTTP!
 - ⦿ Pro mazání instance entity je nutné udělat formulář s metodou DELETE.
 - ⦿ Pro odhlašování je nutné udělat formulář s metodou POST.
- Poskytuje nástroj pro snadnou validaci příchozích dat - skrz "Requesty".
- Při neúspěšné validaci provádí autoredirect zpět do stránky s formulářem a předává naplněné chyby pomocí *\$errors* proměnné a původní vstupní data.

Formuláře - Validací requesty

- Samostatná třída pro definici validace requestu formuláře.
- Lze použít shodnou Request třídu i na validaci dat v API.
- Vytvoření requestu:
 - ⦿ `$ artisan make:request StoreBookRequest`
- V rámci requestu je pak možné definovat sadu pravidel, která jsou validována a sadu zpráv, které jsou vráceny při chybě.
- Sada pravidel pak může být definována jako pole polí, nebo jako pipe notace.

Formuláře - Validací requesty - Array in Array

```
class AuthorRequest extends FormRequest
{
    public function rules(): array
    {
        return [
            'name' => ['required', 'string', 'min:3', 'max:100'],
            'email' => ['required', 'email', 'unique:authors,email', ".$this->route('author')?->id],
            'date_of_birth' => ['required', 'date', 'before:today'],
            'started_writing_at' => ['nullable', 'date', 'after:date_of_birth'],
            'biography' => ['nullable', 'string', 'max:5000'],
            'website' => ['nullable', 'url'],
            'tags' => ['nullable', 'array'],
            'tags.*' => ['string', 'min:2'],
            'country_id' => ['required', 'exists:countries,id'],
        ];
    }

    public function messages(): array
    {
        return [
            'email.unique' => 'Tento e-mail už je zaregistrován pro jiného autora.',
            'date_of_birth.before' => 'Datum narození musí být v minulosti.',
            'started_writing_at.after' => 'Začátek psaní musí být po narození.',
        ];
    }
}
```

Formuláře - Validační requesty - Pipe notation

```
class StoreAuthorRequest extends FormRequest
{
    public function rules(): array
    {
        return [
            'name' => 'required|string|min:3|max:100',
            'email' => 'required|email|unique:authors,email,' . $this->route('author')?->id,
            'date_of_birth' => 'required|date|before:today',
            'started_writing_at' => 'nullable|date|after:date_of_birth',
            'biography' => 'nullable|string|max:5000',
            'website' => 'nullable|url',
            'tags' => 'nullable|array',
            'tags.*' => 'string|min:2',
            'country_id' => 'required|exists:countries,id',
        ];
    }
}
```

Nejčastější pravidla validace

Pravidlo	Popis
required	Hodnota musí být přítomná a nesmí být prázdná.
string,numeric,date,array	Hodnota musí být string, číselná hodnota vč. float, platné datum, nebo pole.
min: <i>x</i> max: <i>y</i>	Hodnota musí být menší než <i>x</i> a větší než <i>y</i> . Lze použít odděleně.
email	Hodnota musí mít platný formát emailu.
unique:table,column	Hodnota musí být unikátní v tabulce a jejím sloupci.
exists:table,column	Hodnota musí existovat v tabulce a jejím sloupci.
confirmed	Hodnota musí být shodná s konfirmačním polem *_confirmation.
nullable	Hodnota může být prázdná.
sometimes	Hodnota se validuje pouze v případě, že je v requestu dostupná (PATCH).
in:foo,bar	Hodnota musí být jedna z uvedených.

Nástroje v Blade

- `$errors` - proměnná, která nese pole chyb ve validaci.
- `@error('name')` - direktiva, která ověří přítomnost chyby na konkrétním poli.
- `old()` - objekt, který nese původní vstup.
- `@if (session('status'))` - nástroj pro ověření přítomnosti flash message v session.

Příklad formuláře

```
@if (session('status'))  
    <div class="alert alert-success">  
        {{ session('status') }}  
    </div>  
@endif  
  
<form method="POST" action="{{ route('books.store') }}">  
    @csrf  
  
    <input type="text" name="title" value="{{ old('title') }}">  
    @error('title')  
        <div class="error">{{ $message }}</div>  
    @enderror  
  
    <button type="submit">Uložit</button>  
</form>
```

Příklad formuláře - obslužný kód

```
public function store(StoreBookRequest $request)
{
    Book::create($request->validated());

    return redirect()
        ->route('books.index')
        ->with('status', 'Kniha byla úspěšně uložena.');
```

CSRF - Cross-Site Request Forgery

- Technika, která zabraňuje provádění neoprávněných requestů na server.
- Neochrání při XSS - pokud je prohlížeč napadený - využívá same-origin policy.
- CSRF využívá většina moderních frameworků.
- Laravel ho pro web POST metody vynucuje - zabezpečení frameworku automaticky request vrátí jako chybný.
- Základní princip:
 - ⦿ Při renderingu formuláře se vygeneruje klíč - ten se uloží do hidden atributu.
 - ⦿ Server si tento klíč uloží do uživatelské session.
 - ⦿ Při vyhodnocování POST se pak podívá do session a ověří, že je klíč platný.

CSRF v Laravelu

- Token vývojáře nemusí vůbec zajímat - postará se o to `VerifyCsrfToken` middleware.
- Do Blade šablony se vloží pomocí direktivy `@csrf` (jako hidden pole).
- Formulář v Blade:

```
<form method="POST" action="{{ route('register') }}">
    @csrf

    <input type="text" name="name" placeholder="Jméno" value="{{ old('name') }}">
    @error('name')
        <div class="error">{{ $message }}</div>
    @enderror

    <button type="submit">Registrovat</button>
</form>
```

CSRF v Laravelu



- Výstupní formulář v HTML:

```
<form method="POST" action="/register">  
  <input type="hidden" name="_token" value="yDjQrcU2aD9xiUhXZtEJGe9WzNdKb4AaA2AYHZrS">  
  
  <input type="text" name="name" placeholder="Jméno" value="">  
  
  <div class="error">Jméno je povinné.</div>  
  
  <button type="submit">Registrovat</button>  
</form>
```

Autentizace a ochrana rout



Autentizace vs. Autorizace

	Autentizace 	Autorizace 
Otázka	"Kdo jsi?"	"Na co máš právo?"
Příklad Laravel	Auth::check(), Auth::user()	Gate::allows(), @can, politiky
Používá se	při přihlašování uživatele	při kontrole oprávnění pro "akci"
Prostředky	email, heslo, session, token	role, práva, politiky
Middleware	auth, guest	can, role, vlastní middleware
Chyba při selhání	401 Unauthorized	403 Forbidden

- **Autentizace** - ověření, že uživatel existuje (vstupenka).
- **Autorizace** - kontrola, zda má uživatel právo něco udělat (kontrola ochrankou).

Laravel Breeze

- Minimalistický autentizační starter-kit.
- Po instalaci do aplikace přidá:
 - ⦿ přihlašování
 - ⦿ registraci uživatelů
 - ⦿ emailové ověření
 - ⦿ základní layout v Blade & Tailwind
 - ⦿ jednoduché kontrollery pro přihlášení, registraci, atd.
- Plná integraci s Blade, VueJS, React a Livewire.
- Vhodný pro střední projekty.

Laravel Breeze - kdy nestačí?

- Pokud je potřeba 2FA ověřování.
- Pokud je potřeba implementovat OAuth/Socialite.
- Pokud chceme vytvářet autentizační skupiny.
- Nepodporuje rozšířené metody autorizace - např. role a politiky.
- Laravel obecně nemá přímou podporu Impersonate (přepínání uživatelů).

Laravel Breeze - co umí?

- Emailové ověřování účtů - vč. časového omezení platnosti linků.
- Integruje změnu zapomenutého hesla.
- Lze ho kombinovat snadno s RateLimiterem/Throttlingem = ochrana proti brute-force.
- Podporuje udržování přihlášení v Session / Remember login.

```
RateLimiter::for('login', function (Request $request) {  
    return Limit::perMinute(5)->by($request->email.$request->ip());  
});
```

```
Route::post('/login', ...)  
    ->middleware(['guest', 'throttle:login']);
```

Laravel Breeze - jak funguje uvnitř?

- Přihlašování je realizováno pomocí session-based techniky:
 1. Uživatelská data se ověřují pomocí `Auth::attempt()`.
 2. Pokud jsou data úspěšně ověřena, vytvoří se Session cookies (Laravel web guard).
 3. Uživatel je následně dostupný skrz `Auth::user()`.
 4. Pro ověřování pak lze používat middleware auth, nebo `Auth::check()`.
- Blade pak nabízí direktivy:
 - ⦿ `@auth` - zjednoduše `if` pro `Auth::check()`
 - ⦿ `@guest` - zjednodušený `if` pro `!Auth::check()`

Laravel Breeze - instalace

- `$ composer require laravel/breeze`
- `$ php artisan breeze:install blade`
- `$ php artisan migrate`

- **!! Není vhodné instalovat Breeze do projektu dodatečně !!**

Laravel Breeze - **auth** middleware

- Integrovaný middleware Laravelu.
- Pokud není uživatel přihlášen - je automaticky zajištěno přesměrování na login stránku.
 - ⦿ Defaultní url pro login je: **/login**
 - ⦿ Vlastní ruta pro login musí mít název **'login'**.
- Pokud se Breeze používá v kombinaci s API - vrací HTTP 401.

```
Route::get('/dashboard', function () {  
    return view('dashboard');  
})->middleware('auth');
```

```
Route::middleware(['auth'])->group(function () {  
    Route::get('/books', [BookController::class, 'index']);  
    Route::get('/profile', [UserController::class, 'edit']);  
});
```

Laravel Breeze - **guest** middleware

- Integrovaný middleware Laravelu.
- Pokud je uživatel přihlášen, nemá k routám chráněným **guest** přístup.
- Věřejné routy je nutné držet ideálně bez middleware.

```
Route::get('/prihlaseni', [AuthenticatedSessionController::class, 'create'])  
    ->middleware('guest')  
    ->name('login');
```

Laravel Breeze - **verified** middleware

- Rozšiřující middleware Breeze - nutno si ho vyžádat při instalaci.
 - ⦿ `$ php artisan breeze:install blade --verification`
- Po registraci nabízí možnost ověření emailu => vyžadován SMTP server.
- Model `User` musí implementovat interface `MustVerifyEmail`.
- V routách pak lze definovat, pro které akce musí být email ověřen.

```
Route::get('/dashboard', function () {  
    return view('dashboard');  
})->middleware(['auth', 'verified']);
```

Laravel Breeze - password.confirm middleware

- Integrovaný middleware Laravelu.
- Cesta jak vynutit nové ověření uživatele heslem (např. pro remember me a long-session).
- Pokud uživatel vstoupí na routu, která je chráněna tímto middleware a nejsou splněny časové podmínky platnosti ověření hesla - dojde k přesměrování na routu `password.confirm`.
- Defaultní délka platnosti ověření hesla je 3 hodiny - lze přenastavit v `config/auth.php`.
 - ⊙ `'password_timeout' => 10800 // platnost hesla 3 hodiny.`

```
Route::get('/account/delete', [AccountController::class, 'confirmDeletion'])
    ->middleware(['auth', 'password.confirm']);
```

Rozšíření o vlastní Role autorizaci (middleware)

- Přidání **roles** atributu do modelu User (**\$fillable** a **\$hidden**).
- Vytvoření enumu pro role **App\Enums\UserRole** : **string**.

- `$ php artisan make:enum UserRole`

- Vytvoření migrace pro nový atribut roles:

- `$ php artisan make:migration add_role_to_users_table`

- `$ php artisan migrate // po doplnění migrace!!`

```
Schema::table('users', function (Blueprint $table) {  
    $table->json('roles')->nullable();  
});
```

```
DB::table('users')->whereNull('roles')->update([  
    'roles' => json_encode([UserRole::User])  
]);
```

Rozšíření o vlastní Role autorizaci

- Rozšíření modelu User o metodu `hasRole(UserRole $role)` a přidání attribute castu:

```
protected function casts(): array
{
    return [
        'email_verified_at' => 'datetime',
        'password' => 'hashed',
        'roles' => 'array',
    ];
}

public function hasRole(UserRole $role): bool {
    return in_array($role->value, $this->roles);
}
```

Rozšíření o vlastní Role autorizaci

- Vytvoření vlastního middleware.
- ⦿ `$ php artisan make:middleware UserRoleAccessMiddleware`

```
public function handle(Request $request, Closure $next, string $role): Response
{
    $role = UserRole::tryFrom($role);
    $user = $request->user();

    if (!$role) {
        abort(400, 'Supplied role is not valid UserRole enum value.');
```

```
    }

    if (!$user->hasRole($role)) {
        abort(403, 'Access denied');
```

```
    }

    return $next($request);
}
```


Rozšíření o vlastní Role autorizaci

- Registrace vlastního middleware v app.php.
- Nutno registrovat skrz `alias()` - jinak nelze předávat parametry!

```
return Application::configure(basePath: dirname(__DIR__))
    ->withRouting(
        web: __DIR__.'/../routes/web.php',
        commands: __DIR__.'/../routes/console.php',
        health: '/up',
    )
    ->withBindings([
        ExceptionHandler::class => App\Exceptions\Handler::class,
    ])
    ->withMiddleware(function (Middleware $middleware) {
        $middleware->alias(['role-access-control' => UserRoleAccessMiddleware::class]);
    })
    ->withExceptions(function (Exceptions $exceptions) {
    })
    ->create();
```

Rozšíření o vlastní Role autorizaci

- Použití middleware ('role-access-control' registrovaný alias) a předání požadované role:

```
Route::prefix('todos')
    ->middleware('web')
    ->middleware('auth')
    ->middleware('role-access-control:' . UserRole::Todo->value)
    ->group(function () {
        /**
         * Route list for todo-items
         */
        Route::get('/', [TodoController::class, 'index'])->name('todo.index');
        Route::get('/{todoItem}/complete', [TodoController::class, 'complete'])->name('todo.complete');
        Route::get('/{todoItem}/delete', [TodoController::class, 'delete'])->name('todo.delete');
        Route::post('/', [TodoController::class, 'store'])->name('todo.store');
    });
```

Rozšíření o vlastní Role autorizaci

- Vlastní Blade direktiva (založena na *if*) - `App\Providers\AppServiceProvider` - `@role()`:

```
public function boot(): void
{
    Blade::if('role', fn($role) => auth()->check() && auth()->user()->hasRole(
        UserRole::tryFrom($role)
    ));
}
```

- Použití v rámci Blade šablon:

```
@role('BOOK_ADMIN')
<div class="mb-4">
    <a href="{{ route('book.create') }}" class="btn btn-outline-warning">Vytvořit knihu</a>
</div>
@endrole
```

REST API



REST API

- **REST** = Representational State Transfer - architektonický styl pro webová API.
- Je postaveno na práci se zdroji (např. /todos, /books, /authors).
- Veškerá komunikace je prováděna skrz HTTP metody.
- Body requestu a odpovědi jsou striktně ve formátu **application/json** (případně json+ld).
- **Komunikace je bez stavová** (state-less) = server nic neví o předchozí komunikaci.
- Autorizaci je nutné řešit v každém requestu.
 - ⦿ V každém requestu se musí posílat ověřovací token v hlavičce (např. Authorization: Bearer <token>).
 - ⦿ Pro bezpečnost je nutná HTTPS komunikace!
 - ⦿ Bearer token vychází ze specifikace [RFC 6750 - OAuth 2.0 - Bearer Token Usage](#).

REST API - HTTP Metody

Metoda	Účel	Příklad
GET	Získávání dat	GET /todos GET /todos/5
POST	Vytváření nového záznamu	POST /todos
PUT	Úplná aktualizace záznamu	PUT /todos/5
PATCH	Částečná aktualizace záznamu	PATCH /todos/5
DELETE	Smazání záznamu	
OPTIONS	Dotaz na dostupné metody (např. pro CORS)	OPTIONS /todos

REST API - HTTP stavové kódy odpovědi

Kód	Význam	Příklad použití v API
200 OK	Úspěšný požadavek	GET /todos PUT /todos/1
201 Created	Záznam vytvořen	POST /todos
204 No Content	Úspěšný požadavek bez odpovědi	DELETE /todos/1
400 Bad Request	Neplatný požadavek	Chybí data, neplatný JSON
401 Unauthorized	Nepřihlášený uživatel	Chybí session token, přihlášení
403 Forbidden	Přístup zakázán	Není oprávnění
404 Not Found	Záznam nenalezen / ruta neexistuje	GET /todos/infinity
409 Conflict	Logický konflikt s existujícím stavem	Neunikátní záznam, nelogický obsah.
422 Unprocessable Entity	Chybná validace	POST /todos
500 Internal Server Error	Neočekávaná chyba	Špatné SQL, bug atd.

REST API v Laravelu

- Laravel poskytuje:
 - ⊙ `routes/api.php` - oddělené místo pro API routy
 - ⊙ `Route::apiResource()` - nástroj pro rychlé definování CRUD endpointů
 - ⊙ Requesty - pro rychlou validaci vstupů
 - ⊙ Resource - pro snadno definovatelnou strukturu výstupu
 - ⊙ Sanctum - oficiální Laravel rozšíření pro autentizaci přes API Tokeny

REST API v Laravelu - Route::**apiResource()**

- Silný nástroj pro rychlou definici routes pro Resource API kontrolery.

- ⦿ ! Nepoužívá HTTP Patch => nutno přes POST posílat vždy kompletní Resource !

```
Route::apiResource('todos', TodoController::class);
```

- Routy jsou svázány s API controllerem, který lze vytvořit přes:

- ⦿ `$ php artisan make:controller Api/TodoController --api`

- Kontroler pak musí plnit interface:

```
interface ApiResourceControllerInterface
{
    public function index();
    public function store(Request $request);
    public function show(Todo $todo);
    public function update(Request $request, Todo $todo);
    public function destroy(Todo $todo);
}
```

REST API v Laravelu - Requesty

- API Requesty jsou shodné s FormRequesty = lze je snadno recyklovat.
 - ⦿ `$ php artisan make:request StoreTodoRequest`
- V případě chybné validace si Laravel podle Accept hlavičky odpověď sám.
- Pokud chceme používat PATCH - musí se request upravit ('sometimes').
 - ⦿ Validace je vhodné upravovat na míru pouze pro metodu patch!

```
public function rules(): array
{
    $isPatch = $this->isMethod('patch');

    return [
        'title' => ($isPatch ? 'sometimes|' : '') . 'required|string|max:255',
        'completed' => ($isPatch ? 'sometimes|' : '') . 'boolean',
    ];
}
```

REST API v Laravelu - Resource

- Nástroj pro custom definici výstupu z API.
- Není nutno ho používat - Laravel v rámci API provádí automatickou serializaci.
- Lze vytvářet skrz
 - ⦿ `$ php artisan make:resource TodoResource`
- Vhodný především ve chvíli, kdy je třeba přesně kontrolovat výstup - nebo ho definovat v konkrétním formátu.
- Vhodné pro verzování výstupů nebo při přizpůsobování oprávněním.
- Doporučený pro velké projekty, kde hrozí nekonzistence v `Model::$hidden`.
- Doporučeno používat pro veřejná API - zamezuje nechtěnému zveřejnění dat.

REST API v Laravelu - JsonResponse

```
class Todo extends Model
{
    protected $fillable = ['title', 'completed', 'user_id'];

    // Skryjeme user_id
    protected $hidden = ['user_id'];

    // Přidáme virtuální atribut
    protected $appends = ['is_owner'];

    public function getIsOwnerAttribute(): bool
    {
        return auth()->id() === $this->user_id;
    }
}
```

```
class TodoResource extends JsonResponse
{
    public function toArray($request): array
    {
        return [
            'id' => $this->id,
            'title' => $this->title,
            'completed' => $this->completed,
            'is_owner' => $this->is_owner,
            'created_at' => $this->created_at->toIso8601String(),
        ];
    }
}
```

REST API v Laravelu - bez Resource

```
class TodoController extends Controller
{
    public function show(Todo $todo)
    {
        return $todo;
    }
}
```

```
{
    "id": 1,
    "title": "Vynést koš",
    "completed": false,
    "is_owner": true,
    "created_at": "2024-05-27T12:00:00Z",
    "updated_at": "2024-05-27T12:30:00Z"
}
```

REST API v Laravelu - S Resource

- Výstup je navíc obalen klíčem "data".

```
use App\Http\Resources\TodoResource;  
  
public function show(Todo $todo)  
{  
    return new TodoResource($todo);  
}
```

```
{  
  "data": {  
    "id": 1,  
    "title": "Vynést koš",  
    "completed": false,  
    "is_owner": true,  
    "created_at": "2024-05-27T12:00:00Z"  
  }  
}
```

API - základní stránkování

- Laravel poskytuje naprosto TRIVIÁLNÍ cestu, pro stránkování dat.
- Od vývojáře, krom jednoho řádku, se neočekává VŮBEC NIC!
- Lze používat i bez Resource objektu - ALE NEKOMBINOVAT v projektu = různé výstupy!!
- `::paginate()` - vrací objekt [LengthAwarePaginator](#).

```
public function index()
{
    return TodoResource::collection(
        Todo::paginate(10)
    );
}
```

```
public function index()
{
    return Todo::paginate(10);
}
```

API - základní stránkování

S resource

```
{
  "data": [
    {
      "id": 1,
      "title": "Koupit kafe",
      "completed": false,
      "is_owner": true,
      "created_at": "2024-05-27T11:00:00Z"
    },
    ...
  ],
  "links": {
    "first": "http://localhost/api/todos?page=1",
    "last": "http://localhost/api/todos?page=5",
    "prev": null,
    "next": "http://localhost/api/todos?page=2"
  },
  "meta": {
    "current_page": 1,
    "from": 1,
    "last_page": 5,
    "path": "http://localhost/api/todos",
    "per_page": 10,
    "to": 10,
    "total": 50
  }
}
```

Bez resource

```
{
  "current_page": 1,
  "data": [ /* pole modelů */ ],
  "first_page_url": "http://localhost/api/todos?page=1",
  "from": 1,
  "last_page": 5,
  "last_page_url": "http://localhost/api/todos?page=5",
  "links": [
    {
      "url": null,
      "label": "&laquo; Previous",
      "active": false
    },
    {
      "url": "http://localhost/api/todos?page=1",
      "label": "1",
      "active": true
    },
    ...
  ],
  "next_page_url": "http://localhost/api/todos?page=2",
  "path": "http://localhost/api/todos",
  "per_page": 10,
  "prev_page_url": null,
  "to": 10,
  "total": 47
}
```


UI - základní stránkování

- Identický přístup lze použít i v rámci UI.

```
class TodoWebController extends Controller
{
    public function index()
    {
        $todos = Todo::orderByDesc('created_at')->paginate(10);

        return view('todos.index', compact('todos'));
    }
}
```

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

@section('content')
    <h1 class="text-xl font-bold mb-4">Seznam úkolů</h1>

    <ul class="space-y-2">
        @foreach ($todos as $todo)
            <li class="border-b pb-2">
                <strong>{{ $todo->title }}</strong>
                @if ($todo->completed)
                    <span class="text-green-600">[hotovo]</span>
                @endif
            </li>
        @endforeach
    </ul>

    <div class="mt-6">
        {{ $todos->links() }}
    </div>
@endsection
```

UI - základní stránkování

- Pro UI jsou pak nabízeny další nástroje [paginátoru](#):
 - ⊙ [cursorPaginate\(\)](#) - používá princip "Oracle Cursor" - do stránek si šifruje row number posledního řádku.
 - Vhodné pro nekonečné scrollování.
 - Vhodné pro extrémně dlouhé seznamy dat.
 - ⊙ [withQueryString\(\)](#) - pro stránkovací URL je zachován query string - např. pro filtrování.
 - ⊙ [appends\(\)](#) - umožňuje přidat další části query stringu.
 - ⊙ [simplePaginate\(\)](#) - vytvoří paginátor pouze s další a předchozí stránkou.
 - ⊙ [onEachSide\(\)](#) - určí kolik stránek má být zobrazeno od aktuální stránky.
 - ⊙ [useBootstrap\(\)](#) - nastaví interní vykreslovací šablonu pro Bootstrap místo Tailwind.

API - Laravel Sanctum

- Oficiální Laravel rozšíření pro autentifikaci tokenem.
- Vhodný pro implementaci SPA a REST API.
- Funguje na principu:
 - ⦿ cookies-based - autentifikace pro SPA (CSRF-protected)
 - ⦿ token-based - autentifikace pro API, mobilní aplikace, CLI a další služby
- Proč ho používat?
 - ⦿ Extrémně rychlá integrace - middleware `auth:sanctum`.
 - ⦿ Tokeny jsou uloženy v DB - lze je snadno spravovat, případně invalidovat.
 - ⦿ Automatická CSRF ochrana pro webové SPA.

Laravel Sanctum - Instalace

- Instalace závislosti
 - ⦿ `$ composer require laravel/sanctum`
- Publikace konfiguračního souboru - do config/sanctum.php
 - ⦿ `$ php artisan vendor:publish --tag=sanctum-config`
- Publikace migrací a konfiguračního souboru
 - ⦿ `$ php artisan vendor:publish --provider="Laravel\Sanctum\SanctumServiceProvider"`
- Spuštění migrací nutných pro běh Laravel Sanctum
 - ⦿ `$php artisan migrate`

Laravel Sanctum - AuthController - HTTP POST

```
public function login(): JsonResponse
{
    $request = request();

    $request->validate([
        'email' => 'required|email',
        'password' => 'required|string',
    ]);

    $user = User::where('email', $request->email)->first();

    if (!$user || !Hash::check($request->password, $user->password)) {
        throw new AuthenticationException("Invalid credentials.");
    }

    $token = $user->createToken('api-token', ['*'], now()->addWeek());

    return response()->json([
        'token' => $token->plainTextToken,
        'token_expiration' => $token->accessToken->expires_at,
        'user' => $user,
    ]);
}
```

Laravel Sanctum - AuthController - příklad

● Odhlášení - HTTP metoda DELETE

```
public function logout()
{
    request()->user()->currentAccessToken()->delete();

    return response()->noContent();
}
```

● Odhlášení ze všech zařízení - HTTP metoda DELETE

```
public function revokeAllTokens()
{
    request()->user()->tokens()->delete();

    return response()->noContent();
}
```

Laravel Sanctum - Routing

○ Základní definice routes pro práci s Laravel Sanctum.

- ⦿ Ruta **/logout-all** je volitelná - odhlášení všech tokenů.
- ⦿ Route **/me** je volitelná - slouží pro vrácení aktuálního uživatele.

```
Route::post('/login', [AuthController::class, 'login']);
```

```
Route::middleware('auth:sanctum')->group(function () {  
    Route::delete('/logout', [AuthController::class, 'logout']);  
    Route::delete('/logout-all', [AuthController::class, 'logoutAll']);  
    Route::get('/me', fn () => request()->user());  
});
```

Laravel Sanctum - Příklad API se zabezpečením

- Todo API se zabezpečením **auth:sanctum**:

```
Route::middleware(['api', 'auth:sanctum'])
->prefix('api/todo')
->group(function () {
    Route::get('/', [TodoController::class, 'index']);
    Route::get('/{id}', [TodoController::class, 'show']);
    Route::post('/', [TodoController::class, 'store']);
    Route::put('/{id}', [TodoController::class, 'update']);
    Route::delete('/{id}', [TodoController::class, 'destroy']);
});
```


Laravel Sanctum - Příklad volání s tokenem

- Token se do hlavičky přidává pod názvem Authorization.
- Hodnota hlavičky vždy začíná "Bearer ".

```
GET /api/todos HTTP/1.1
```

```
Host: your-api.test
```

```
Authorization: Bearer 1|eyJ0eXAiOiJKV1QiLCJhfsfajowqQsa
```

```
Accept: application/json
```

Rozšíření



Laravel Sanctum - Limitování počtu tokenů

- Sanctum bohužel nenabízí vlastní řešení.
- Lze ale snadno rozšířit **AuthController** o vlastní logiku.

```
public function login()
{
    $user = \App\Models\User::where('email', request('email'))->first();

    if (! $user || ! \Hash::check(request('password'), $user->password)) {
        throw new \Illuminate\Auth\AuthenticationException("Invalid credentials.");
    }

    if ($user->tokens()->count() >= 5) {
        $user->tokens()->oldest()->first()->delete();
    }

    $token = $user->createToken('api-token')->plainTextToken;

    return response()->json(['token' => $token]);
}
```

API - Rate limiting

- Pokud chceme omezovat počet dotazů za určité období na jednotlivé routy.

- ⊙ Po vyčerpání se vrací HTTP 429 Too Many Requests

- Lze definovat vlastní podmínku, defaultně IP.

- Definice Limiteru např. v `RouteServiceProvider::boot()`:

```
RateLimiter::for('todo-api', function (Request $request) {  
    return Limit::perMinute(30)->by(optional($request->user())->id ?: $request->ip());  
});
```

- Následné použití v routingu (**throttle** middleware):

```
Route::middleware(['auth:sanctum', 'throttle:todo-api'])  
->prefix('api/todo')  
->group(function () {  
    Route::get('/', [TodoController::class, 'index']);  
    // ostatní routy...  
});
```

API - Rate limiting - pro login

- Stejný princip jako v předchozí ukázce - pouze rozšířené o zadaný email.
- Definice Rate Limiteru např. v `ApiServiceProvider::boot()`:

```
RateLimiter::for('login', function (Request $request) {  
    $email = (string) $request->email;  
    return Limit::perMinute(5)->by($email . '|' . $request->ip());  
});
```

- Následné použití v routingu (**throttle** middleware):

```
Route::post('/login', [AuthController::class, 'login'])->middleware('throttle:login');
```

Rate limiting - pro login (LoginRequest)

```
public function authenticate()
{
    $key = 'login:' . strtolower($this->email) . '|' . $this->ip();
    $lock = $key . ':lock';

    if (cache()->has($lock)) {
        abort(429, 'Too many attempts.');
```

```
    }

    $user = \App\Models\User::where('email', $this->email)->first();

    if (! $user || ! \Hash::check($this->password, $user->password)) {
        if (cache()->increment($key) >= 5) {
            cache()->put($lock, true, now()->addMinutes(5));
            cache()->forget($key);
        }

        throw new \Illuminate\Auth\AuthenticationException();
    }

    cache()->forget($key);
    cache()->forget($lock);

    return $user;
}
```

```
public function login(LoginRequest $request)
{
    $user = $request->authenticate();
    $token = $user->createToken('api-token')->plainTextToken;

    return response()->json(['token' => $token]);
}
```

- **\$key** = pro počítadlo pokusů o přihlášení
- **\$lock** = klíč pro vložený zámek
- **addMinutes(5)** = zámek na 5 minut

Reference Zákazníci

Profesionální
vzdělávací služby
poskytujeme
firmám a
institucím různé
velikosti a
zaměření již více
než 25 let.



Kvalitu našich
vzdělávacích
služeb podtrhuje
množství ohlasů
významných
českých i
zahraničních firem
a institucí.

Partnerství Certifikace

Hodnotu vzdělávání a renomé našich kurzů podporujeme prostřednictvím autorizací a partnerství s řadou světových společností.

Jako autorizované testovací centrum zajišťujeme klientům zkoušky pro většinu mezinárodně uznávaných kompetencí. A rádi vás na tyto zkoušky i připravíme.



Nabízíme vám přes pět set praktických workshopů a seminářů, které vám pomáhají rozvíjet měkké dovednosti a profesní kompetence.

Komplexní, přístup, odborné přesahy a osobní zkušenosti našich školitelů jsou zárukou skvělého výsledku.

ICT Pro v Brně



Adresa sídla
Sochorova 38
616 00 Brno



Telefon
+420 775 86 44 00



Web & Email
www.ictpro.cz
skoleni@ictpro.cz



ICT Pro v Praze



Adresa pobočky
Líbalova 1
149 00 Praha 11



Telefon
+420 277 77 11 99

Školení ICT Pro probíhala již ve 30 zemích světa na 4 kontinentech

Už více než 25 let pomáháme firmám a jednotlivcům růst. Po odborné i lidské stránce.

Rozvíjíme, inspirujeme a motivujeme: Pro efektivnější a kvalitnější práci.

Pro naplněný a spokojený život. A snad i pro lepší svět.



Na cestě k pozitivním změnám vás provází lektoři, co skutečně naučí a nadchnou k další práci na vašem rozvoji.

A to tak, abyste dokázali zase o něco více využít svůj potenciál.