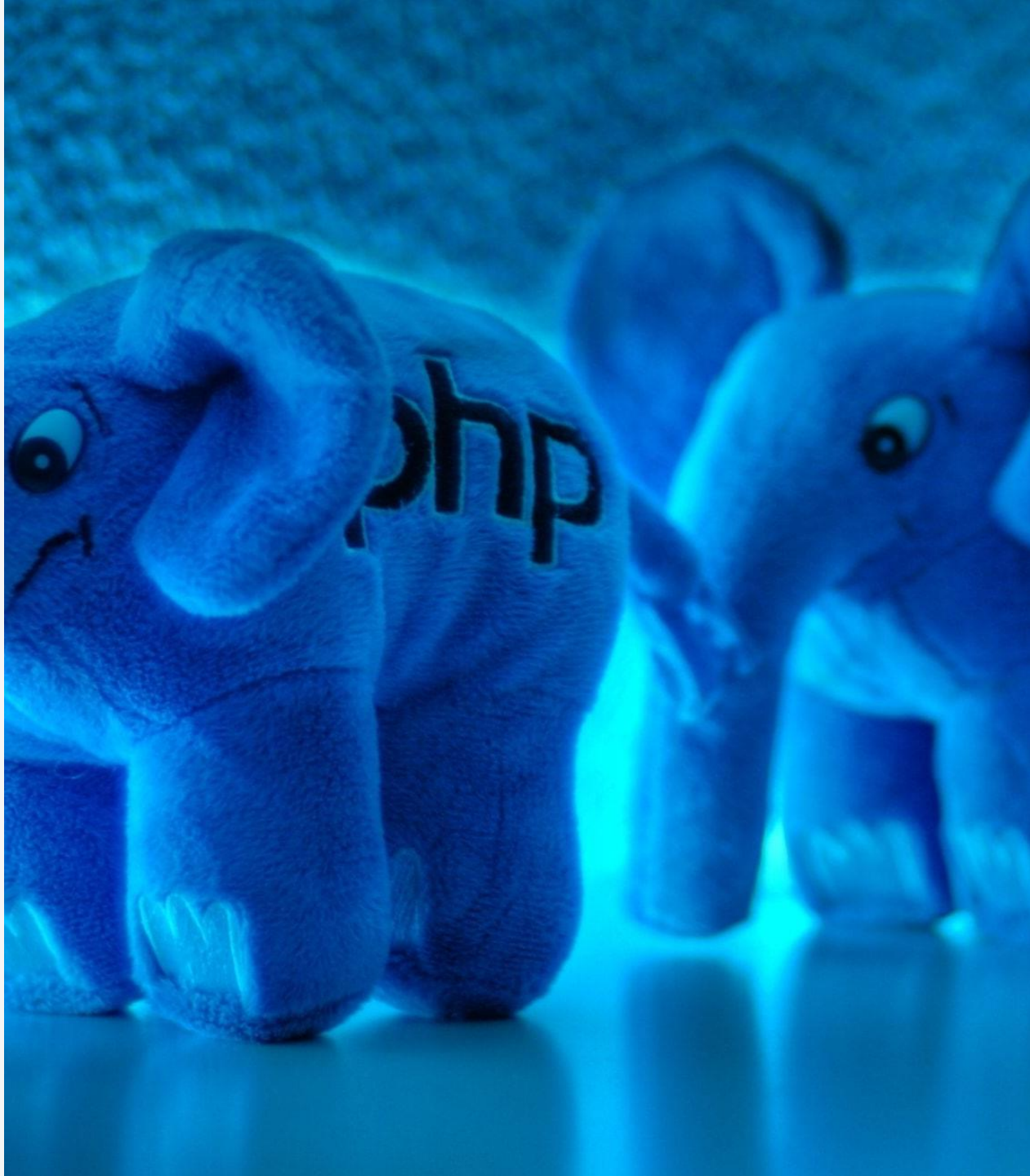
Rozkvět PHP

Lokální vývoj s použitím Dockeru

Antonín Neumann



PeoplePath

Co je PHP?

PHP je skriptovací programovací jazyk navržený především pro vývoj webových aplikací.
*Zkratka "**PHP**" původně stála za názvem "**Personal Home Page**", ale nyní se oficiálně označuje jako "**PHP Hypertext Preprocessor**". PHP je široce používaný pro vytváření dynamických webových stránek a aplikací.*

ChatGPT

A popular general-purpose scripting language that is especially suited to web development.
Fast, flexible and pragmatic, PHP powers everything from your blog to the most popular websites in the world.

php.net

Stručná historie PHP

- PHP 3 (1998)
    - základní podpora OOP, podpora pro **MySQL**, podpora pro **formuláře**, základní podpora **sessions**
- PHP 4 (2000)
    - nový **Zend Engine**, podpora extensions, zavedení **referencí a garbage collection,** podpora pro **XML**, příkaz **foreach**,
      nový datový typ **boolean**
- PHP 5 (2004)
    - Zend Engine II, **OOP**, podpora SOAP protokolu a SimpleXML, podpora PDO, Exception Handling (try-catch), magické metody (`__get()`), přidány iterátory
- PHP 5.3 (2009)
    - Jmenné prostory, anonymní funkce, late static binding (`static::`), ternární operátor (`$if ? $true : $false`), příkaz `goto`, nový garbage collector (rozpozná cykly)

Stručná historie PHP

- PHP 5.4 (2012)
    - Traits, built-in server (`php -S localhost:8000`), zkracený zápis polí,
- PHP 5.5 (2013)
    - Generátory (operátor `yield`), blok `finally`, OpCode Cache, konstanta `::class`, práce s hesly (`password_hash`), `array_column`
- PHP 5.6 (2014)
    - Variabilní funkce (...), skalární výrazy v konstantách, exponenciální operátor (**), příkaz `use` pro funkce

Stručná historie PHP

PHP 7.0 - 7.4 (2015 - 2019)

- Scalar Type Declarations, Return type declarations, Typed properties, Anonymous Classes, Null Coalesce Operator (??, ??=), Spaceship Operator (<=>), Anonymous Classes, Short Closures (arrow functions), Numeric Literal Separator (10_000)

PHP 8.0 - 8.3 (2020 – 2023)

- Named arguments, Attributes, Constructor property promotion, Union types (float|int), Match expression, nový datový typ mixed, funkce str_contains() str_starts_with() a str_ends_with()
- Enumerations, Readonly Properties, Pure Intersection Types (Iterator&Countable), return type never, Fibers
- Readonly classes, DNF Types (dysjunktní normální forma, kombinace Union types a Intersection types: (A&B)|null), nové datové typy null, false a true, Constants in traits, #[\SensitiveParameter] attribute (není vidět ve vyjímkách)
- Nová funkce json_validate, volání konstant dynamicky (echo MyClass::{$constName};),

Popularita PHP

- Facebook
- Wikipedia
- Wordpress
- Mailchimp
- Tumblr
- Slack

Popularita PHP

Jazyk	12/2022	12/2023
PHP	77,5%	76,6%
ASP.NET	7,4%	6,7%
Ruby	5,5%	5,6%
Java	4,6%	4,7%
JavaScript	2,2%	3,1%

Zdroj: https://w3techs.com/technologies/history_overview/programming_language

Popularita PHP

Proč je PHP tak populární?

- Dobrá křivka učení
  - na začátku lze používat jen drobné funkce
  - třídy a komplexní frameworky až později
- Fungující komunita
  - PSR doporučení
  - Správce balíčků Composer
  - Autoloading (PSR + Composer)
- Levná a dostupná infrastruktura (hosting za pár korun, web může běžet do pár minut)

PHP 101

```
<html>
 <head>…</head>
 <body>
  …layout
  …navigation

<?php
$path = 'pages/';
$page = $_GET['page'] ?? 'home';
if (file_exist($path . $page)) {
    require_once $path . $page;
} else {
    require_once $path . 'error-404';
}
?>

  …footer
 </body>
</html>
```

Rozběhání PHP

- Typicky tzv. LAMP/WAMP
  - Linux (Windows)
  - Apache (NginX, Lighttpd)
  - MySQL (MariaDB, PostreSQL)
  - PHP
- Lokální vývoj
  - Ve Windows Xampp (WampServer, EasyPHP, …)
  - PHP built-in web server (`php -S localhost:8000`)
  - PHPStorm built-in web server (View > Open in Browser)
  - **Docker**

Co je Docker

Docker je platforma pro kontejnerizaci aplikací

- umožňuje zapouzdřit a spustit aplikace a jejich závislosti v izolovaném prostředí nazývaném kontejner
- kontejnery umožňují snadné přenášení aplikací mezi různými prostředími
- Docker nabízí předdefinované images (šablony kontejnerů)

Dockerfile

- Soubor pro definici vlastního image

Docker compose

- Nástroj na propojení více kontejnerů
- Alternativně Kubernetes, Docker Swarm a další

Proč zvolit Docker?

1. Všichni mají všude stejnou verzi!
2. Pro nováčky a vývojáře obecně velmi jednoduché (`docker compose up -d`).
3. Snažší údržba (vše na jednom místě, všichni mají okamžitě update k dispozici)
4. Dockerfile i Compose file jsou současní kódu (dostupnost, verzování, …)
5. Všechny cloudy docker podporují => snažší migrace (teoreticky)
6. Lze dobře škálovat (některé kontejnery mohou běžet v cloudu a jiné lokálně)

! Na co nezapomínat:

Vše ohledně Vaší aplikace (composer, npm, checkstyle, phpstan, phpunit, …) je nutné pouštět uvnitř kontejneru.

Základ pro lokální vývoj v PHP

- Server (naše aplikace)
  - oficiální image `php:8.3.0-apache`
  - obsahuje webový server **Apache2** a **PHP 8.3**
  - poslouží jako základ k vytvoření vlastního aplikačního kontejneru
- Databáze
  - oficiální **MySQL** image
  - **Adminer** - webové rozhraní pro přístup k databázi
  - Případně kombinace **Mongo** a **MongoExpress**

Aplikační Dockerfile

```
FROM php:8.3.0-apache

#install necessary PHP extensions
RUN docker-php-ext-install pdo pdo_mysql
#install needed software
RUN DEBIAN_FRONTEND=noninteractive apt-get -y update && apt-get -y install npm
#allow mod rewrite for Apache
RUN a2enmod rewrite

#install composer
RUN php -r "copy('https://getcomposer.org/installer', 'composer-setup.php');" \
    && php composer-setup.php \
    && rm composer-setup.php \
    && mv composer.phar /usr/local/bin/composer

CMD /usr/sbin/apache2ctl -D FOREGROUND
```

Docker compose file

```
version: '3.1'
services:
  app:
    build: .
    ports:
      - 80:80
    volumes:
      - .:/var/www/html
  db:
    image: mysql
    environment:
      MYSQL_ALLOW_EMPTY_PASSWORD: 'yes'
      MYSQL_ROOT_PASSWORD: ''
      MYSQL_DATABASE: test
    volumes:
      - ./data/db.sql:/docker-entrypoint-
initdb.d/1.sql
  adminer:
    image: adminer
    ports:
      - 8080:8080
    volumes:
      - ./data/adminer/login-pass-
less.php:/var/www/html/plugins-enabled/login-pass-
less.php
```

Aktualní struktura aplikace

```
aplikace/
├─ Dockerfile
├─ docker-compose.yml
├─ index.php
```

Soubor index.php

```
<?php

echo 'Hello World!';
```

Jak to spustit?

1. Je nutné mít naistalovaný **Docker** a **Docker Compose**.
2. V adresáři s projektem provedeme tzv. build kontejneru
   `docker-compose build`
3. Nakonec spustíme všechny kontejnery
   `docker-compose up -d`
4. A pokud nám nic neblokuje zvolené porty a všechno proběhlo správně tak nám pojede web na adrese http://localhost

Základní příkazy pro Docker

- `docker-compose down`
  - Ukončí všechny kontejnery
- `docker-compose logs -f <serviceName>`
  - Zobrazí logy z daného kontejneru
- `docker-compose exec -u root <serviceName> sh`
  - Připojí se k danému kontejneru
- `docker-compose run --rm --no-deps app composer "$@"`
  - Spustí příkaz **composer** v kontextu **app** kontejneru a zbytek předá jako parametry
  - *Pro tip: vytvořte si na často používané příkazy bash scripty*

Proč někdo nemá PHP rád...?

```
<?php

if ($_POST['send']) { //zpracuj formulář }

if (isset($_GET['page'] && file_exist('cesta/' . $_GET['page'])) {
  include 'cesta/' . $_GET['page'];
}

include 'function.php';
$db = get_connection();

$user = null;
if (isset($_SESSION['user_id']) {
  $statement = $db->prepare('SELECT * form user where user_id = ?');
  $statement->execute([$_SESSION['user_id']);
  $user = $statement->fetch();
}
```

Jde to i lépe - díky komunitě :-)

PSR (PHP Standards Recommendations)

- Obdoba RFC, PHP Framework Interop Group
- Např. PSR-3 Logger Interface, PSR-7 HTTP Message Interface
- PSR-4 Autoloading Standard

Composer

- Správa závislostí (knihovny, frameworky, …)
- Podpora pro PSR-4 autoloading
- Informace a nastavení uchovává v souboru *composer.json*

Autoloading

- Standardně v PHP se soubory/scripty propojují pomocí `include/require`
- U velkých projektů nedáva smysl (velké množství souborů, většina nevyužitá)
- `spl_autoload_register(callable)`
  - Umožnuje definici automatického nahrání souboru pokud PHP nenajde definici funkce nebo třídy
- PSR-4 k tomu dodalo užitečná a jednotná pravidla
  - 1 class = 1 soubor
  - File path = namespace
  - Vendor prefix namespace

Fully qualified class name	Namespace prefix	Base directory	Resulting file path
\MojeApp\Modul\Trida	\MojeApp	/path/to/src/	/path/to/src/Modul/Trida.php

Composer - autoloading

- Composer umožňuje použít jejich PSR-4 autoloader
- Stačí přidat informaci do souboru *composer.json*

```
{
    "autoload": {
        "psr-4": {"MojeApp\\": "src/"}
    }
}
```

- Důležitý soubor **`vendor/autoload.php`**

Fully qualified class name	Namespace prefix	Base directory	Resulting file path
\MojeApp\Modul\Trida	\MojeApp	./src/	./src/Modul/Trida.php

Composer – init a přidání závislostí

- `composer init`
  - Pokud soubor composer.json neexistuje -> vytvořím ho
- `composer require`
  - Dovolí vyhledat a stáhnout balíčky
- Všechny závislosti jsou potom ve složce `vendor/`
- Zajímavá je též složka `vendor/bin/`
  - Obsahuje spustitelné soubory pro některé knihovny
  - PHPUnit, Checkstyle, PHPStan
  - Nutné pouštět uvnitř kontejneru
    `docker-compose run --rm --no-deps app vendor/bin/phpcs <filePath>`

composer.json

```
{
    "name": "tonda/workshop",
    "require-dev": {
        "phpstan/phpstan": "^1.10",
        "phpunit/phpunit": "^10.5",
        "squizlabs/php_codesniffer": "^3.7"
    },
    "require": {
        "slim/slim": "^4.12",
        "guzzlehttp/psr7": "^2.6"
    },
    "autoload": {
        "psr-4": {
            "Tonda\\Workshop\\": "src/"
        }
    }
```

Evergreen knihovny

- **phpstan/phpstan**
  - statická analýza soborů
- **phpunit/phpunit**
  - unitové testování
- **squizlabs/php_codesniffer**
  - checkstyle (formátování kódu)

Moderní aplikace

```
<?php
use Psr\Http\Message\ResponseInterface as Response, Psr\Http\Message\ServerRequestInterface
as Request, Slim\Factory\AppFactory;

require __DIR__ . '/vendor/autoload.php';

$app = AppFactory::create();

$app->get('/hello/{name}', function (Request $request, Response $response, $args) {
    $name = $args['name'];
    $response->getBody()->write("Hello, $name");
    return $response;
});

$app->run();
```

Šablony

Stáhneme jednoduchý view renderer přímo od Slim

- `composer require slim/php-view`

Změníme definici routy

```
$app->get('/hello/{name}', function ($request, $response, $args) {
    $renderer = new PhpRenderer('src/templates');
    $renderer->setLayout("layout.html");
    $args['title'] = 'Hello...';
    return $renderer->render($response, "hello.html", $args);
});
```

Šablony

src/layout.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title><?=$title?></title>
</head>
<body>
    <?=$content?>
</body>
</html>
```

src/hello.html

```
Hello <?= $name ?>!
```

Větší projekt => přehlednější struktura

- OK pro web pro strejdu, co když ale chceme víc?
- Přidáme Controllery (pojem z MVC)
  - src/Controller/HelloController.php
- `Upravíme volání routy`
  - `$app->get('/hello/{name}', [HelloController::class, 'hello']);`

Controller

```php
<?php
namespace Tonda\Workshop\Controller;

use Psr\Http\Message\ResponseInterface as Response;
use Psr\Http\Message\ServerRequestInterface as Request;
use Slim\Views\PhpRenderer;

class HelloController
{
    public function hello(Request $request, Response $response, array $args): Response
    {
        $renderer = new PhpRenderer('src/templates');
        $renderer->setLayout("layout.html");
        $args['title'] = 'Hello...';
        return $renderer->render($response, "hello.html", $args);
    }
}
```

A teď už jedu...

- Nová routa

```
$app->get('/hello-rev/{name}', [HelloController::class, 'reverse']);
```

- Přidání metody do controlleru (nebo jiný controller)

```
public function reverse(Request $request, Response $response, array $args): Response
{
    $renderer = new PhpRenderer('src/templates');
    $renderer->setLayout("layout.html");
    $args['title'] = 'Hello...';

    // reverse the name
    $args['name'] = strrev($args['name']);

    return $renderer->render($response, "hello.html", $args);
}
```

Formuláře

- **Nová routa**

```
$app->post('/hello-post', [HelloController::class, 'form']);
```

- **HTML**

```
<form method="post" action="hello-post">
  <input type="text" name="name">
  <button type="submit">Odeslat</button>
</form>
```

- **Přidání metody do controlleru (nebo jiný controller)**

```
public function reverse(Request $request, Response $response, array $args): Response
{
  $renderer = new PhpRenderer('src/templates');
  $renderer->setLayout("layout.html");
  $body = $request->getParsedBody();
  $args['title'] = 'Form Hello';
  $args['name'] = $body['name'];
  return $renderer->render($response, "hello.html", $args);
}
```

Dependency Injection

- **Stáhneme např. PHP-DI (na doporučení Slimu)**
  - `composer require php-di/slim-bridge`
- **index.php**
  - `$app = \DI\Bridge\Slim\Bridge::create();`
- **Controller**

```
class HelloController
{
    public function __construct(private readonly PhpRenderer $renderer) {
        $this->renderer->setTemplatePath('src/templates');
        $this->renderer->setLayout('layout.html');
    }

    public function hello(Request $request, Response $response): Response {
        $args['title'] = 'Hello...';
        $args['name'] = $request->getAttribute('name');
        return $this->renderer->render($response, "hello.html", $args);
    }
```

Zdrojový kód

https://github.com/tonda13/pph-workshop-php

Díky za pozornost