# PHP – objektově orientovaný přístup

PHP může být používáno jak procedurálním způsobem, tak objektovým způsobem. Datové struktury se mohou tedy používat jako objekty s vlastnostmi a metodami. Teorie k OOP je popsána zevrubně v části skript věnované přímo problematice Objektově orientovaného programování. Na úvod si tedy jen shrneme, jak to chodí přímo u PHP, abyste se mohli rychle zorientovat a mít na papíře všechny důležité postupy, které ve cvičeních využijeme.

Otázka, která vás napadne – Jaký je přínos použití OOP v PHP? Stejný jako platí obecně pro užití OOP v programování, tedy psaní DRY (*don't repeat yourself* – přístup neopakovat stále stejné kusy kódu) code, přehlednost, variabilita.

Navíc, v drtivé většině PHP frameworky využívají OOP principy.

## Class a objekt

Třídu si představte jako předpis nějakého objektu. Objekt můžeme nazvat i instancí třídy.

class Person

{

    // vlastnosti - properties

    public $name;

    public $surname;

    // explicitní konstruktor

    public function \_\_construct($n, $s) {

        $this->name = $n;

        $this->surname = $s;

    }

// metody – methods

    public function getFullName() {

        return "{$this->name} {$this->surname}";

    }

}

Instanci třídy Person vytvoříme pomocí konstruktoru, který jsme si definovali ve třídě Person s názvem *\_\_construct* (pozor – dvě podtržítka). Každá třída má automaticky implicitní bezparametrický konstruktor, která vytvoří instanci bez definovaných vlastností. Platnost tohoto konstruktoru zaniká, pokud definujeme jakýkoliv vlastní konstruktor – jako v našem případě. Konstruktor se volá pomocí slůvka *new* a názvu třídy s kulatými závorkami, které případně obsahují parametry konstruktoru.

Klíčové slovo *this* je odkaz, který ukazuje na aktuální objekt, jehož metoda byla volána. Když vytvoříte instanci třídy a voláte její metody, *this* v těchto metodách odkazuje zpět na tuto konkrétní instanci.

Pokud bude objekt třídy Person, víme, že má vlastnosti *name* a *surname* a pomocí metody *getFullname()* získáme celé jméno ve formě řetězce.

$osoba = new Person("Henry", "Vocet"); // voláme konstruktor, nová instance

echo $osoba->getFullName(); // vypíše: Henry Vocet

Perfektní. První krůček ve vodách PHP OOP máme za sebou, jdeme plavat.

## Dědičnost

Jedním ze základních konceptů OOP je dědičnost, neboli možnost převzít vlastnosti a metody z jiné třídy. Můžeme si to ukázat na nové třídě *Employee*, která bude dědit ze třídy *Person*.

class Employee extends Person {

// přidání nové vlastnosti

public $position;

// rozšíření konstruktoru

public function \_\_construct($n, $s, $p) {

// volání konstruktoru rodičovské třídy Person

parent::\_\_construct($n, $s);

$this->position = $p;

}

// nová metoda specifická pro třídu Employee

public function getInfo() {

return "{$this->getFullName()} - {$this->position}";

}

}

Třída *Employee* zdědí všechny vlastnosti a metody třídy *Person*. Pokud bychom zde neaplikovali dědičnost, pak bychom duplikovali zbytečně kód. Vypisovali bychom znovu vlastnosti *name* a  *surname* stejně jako metodu *getFullname()*, jelikož chceme, aby toto vše měla i třída *Employee.*

## Abstraktní třída a interface

Ze třídy *Person* si uděláme třídu abstraktní, neboť pro naše účely budeme potřebovat už jen třídy s různými rozšiřujícími vlastnostmi, ale v základu budeme chtít, aby všechny měly vlastnosti a metody třídy *Person*. Z ní samotné tedy již instance vytvářet nebudeme moci.

abstract class Person {

public $name;

public $surname;

public function \_\_construct($n, $s) {

$this->name = $n;

$this->surname = $s;

}

// abstraktní metoda – musí být implementována v odvozených třídách

abstract public function getFullName();

}

Dále si vytvoříme interface, neboli rozhraní – budeme chtít totiž metodu, která vypíše všechny informace. A tuto metodu budeme následně chtít použít i pro nějakou třídu, která nedědí z *Person.*

interface Information {

public function getInfo();

}

No, a na třídu *Employee* aplikujeme dědičnost z abstraktní třídy a interface.

class Employee extends Person implements Information {

public $position;

public function \_\_construct($n, $s, $p) {

parent::\_\_construct($n, $s);

$this->position = $p;

}

// implementace abstraktní metody getFullName

public function getFullName() {

return "{$this->name} {$this->surname}";

}

// implementace metody getInfo z rozhraní Information

public function getInfo() {

return "{$this->getFullName()} - {$this->position}";

}

}

// vytvoření instance Employee

$employee = new Employee("Henry", "Vocet", "Manager");

echo $employee->getInfo(); // vypíše: Henry Vocet - Manager

## Zapouzdření – encapsulation

Dalším důležitým aspektem OOP je zapouzdření/enkapsulace. Jde o to, aby nebyly vlastnosti či metody přístupné bez explicitně definovaných metod. A to jak pro nastavení nové hodnoty (setter), tak pro čtení hodnoty (getter). Proč tomu tak je? Například nechceme, aby byl přístup k některým údajům vůbec. Jindy zase, aby nastavování nové hodnoty vlastnosti splňovalo daná kritéria.

abstract class Person {

// změna viditelnosti na private pro zvýšení enkapsulace

private $name;

private $surname;

public function \_\_construct($n, $s) {

$this->setName($n);

$this->setSurname($s);

}

abstract public function getFullName();

// getter a setter pro name a surname

protected function getName() {

return $this->name;

}

protected function setName($n) {

$this->name = $n;

}

protected function getSurname() {

return $this->surname;

}

protected function setSurname($s) {

$this->surname = $s;

}

}

class Employee extends Person implements Information {

private $position;

public function \_\_construct($n, $s, $p) {

parent::\_\_construct($n, $s);

$this->setPosition($p);

}

public function getFullName() {

return "{$this->getName()} {$this->getSurname()}";

}

public function getInfo() {

return "{$this->getFullName()} - {$this->getPosition()}";

}

// getter a setter pro position

private function getPosition() {

return $this->position;

}

private function setPosition($p) {

$this->position = $p;

}

}

### Přístupnosti:

* **Public** – metody a vlastnosti s touto přístupností můžete použít odkudkoliv
* **Protected** – k těmto metodám a vlastnostem máte přístup z metod stejné třídy a tříd od ní odvozených
* **Private** – přístup je povolen pouze metodám stejné třídy

## Polymorfismus

Řekněme, že bychom si vytvořili třídu, která by implementovala rozhraní *Information*.

class InventoryItem implements Information {

private $itemName;

private $itemId;

public function \_\_construct($i, $id) {

$this->itemName = $i;

$this->itemId = $id;

}

public function getInfo() {

return "Item: {$this->itemName}, Id: {$this->itemId}";

}

}

A nyní si vytvořím pole, které bude obsahovat instance tříd *Employee* i *InventoryItem*. V případě PHP není polymorfismus až tak zřejmý, jelikož do pole můžeme vlastně nastrkat prvky stylem *každý pes jiná ves.* Ovšem všimněte si posléze, že jsme všechny prvky prohnali smyčkou a použili jsme jedinou metodu, kterou mají všechny společnou – *getInfo()* z interface *Information*, které obě třídy implementují*.* Zpracováváme je tedy jednotně, byť jsou to objekty odlišných tříd pouze se společným interface.

$employee1 = new Employee("Mylene", "Dubois", "Manager");

$employee2 = new Employee("Renny", "Huffman", "Assistant");

$inventory1 = new InventoryItem("Laptop", 1);

$inventory2 = new InventoryItem("Car", 2);

$items = [$employee1, $inventory1, $employee2, $inventory2];

foreach ($items as $item) {

echo $item->getInfo() . "\n";

}

## Výjimky

Výjimky jsou objekty, které reprezentují chybové stavy nebo neočekávané situace v programu. V PHP jsou výjimky instance třídy Exception nebo jejích odvozenin a umožňují oddělit kód pro zpracování chyb od hlavního logického toku programu, což vede k čistšímu a lépe udržovatelnému kódu. Výjimky jsou generovány (házeny) pomocí klíčového slova throw a zachyceny pomocí bloků try a catch. To umožňuje programu pokračovat v provozu i po setkání s chybou.

try {

// code, který může vygenerovat výjimku

throw new Exception("Došlo k chybě");

} catch (Exception $e) {

// code pro zpracování výjimky

echo $e->getMessage();

}

S výjimkami se setkáte v dalších kapitolách, tak ať víte, o co go.

# PHP kontaktní formulář

Receptů, jak vyrobit funkční kontaktní formulář, který Vám bude z vašich webovek odesílat zprávu od návštěvníka, najdete spoustu. Leč právě v té spoustě se člověk může jednoduše ztratit. My si ukážeme způsob, který využívá knihovnu PHPMailer. Je to vcelku spolehlivé řešení, které má ne až tak náročný proces nastavení. Na konci kapitoly 2 budete mít odkaz na hotový code. Zkuste si jej ale udělat krok po kroku a dělejte si komentáře, abyste si kód osahali a byli si vědomi, co dělá a hlavně, co děláte vy.

Ještě, než začneme, spusťte si na Xampp kontrolním panelu lokální server Apache a vytvořte si pracovní adresář ve složce C:\xampp\htdocs.

## Composer

Jedná se o nástroj, jenž je určen pro správu závislostí/knihoven pro PHP. V podstatě jde o sourozence balíčkovacího systému npm, který je zase určen pro JavaScript (Node.js).

[getcomposer.org/](https://getcomposer.org/)

Po instalaci můžete používat příkazy pro Composer v příkazové řádce, balíčky pro PHP projekt se budou instalovat automaticky.

## PHPMailer

Jedná se o PHP knihovnu, usnadňující odesílání emailů pomocí SMTP serveru, jenž je zodpovědný za přenos emailů z jednoho emailového serveru na druhý – přijímá, přenáší a doručuje emaily.

Protokol SMTP je zkratkou pro *Simple Mail Transfer Protocol.*

Na GitHub stránce PHPMaileru (dále už jen PM) se dočtete, že je několik způsobů, jak si dotáhnout PM do svého projektu:

1. buď si stáhnete do kořenového adresáře zazipovanou složku se zdrojovými soubory – *download PHPMailer as a zip file.* Složku si rozbalte.
2. nebo použijete příkaz pro Composer v terminálu

*composer require phpmailer/phpmailer*

[github.com/PHPMailer/PHPMailer](https://github.com/PHPMailer/PHPMailer)

## Vytvoření základních souborů

Vytvořte si v kořeni projektu soubor **index.html**, kam si do body umístíte tento jednoduchý formulář.

<form action="./send-mail.php" method="post">

    <div>

       <label for="email" class="form-label">Email address</label>

       <input type="email" id="email" name="email" required>

    </div>

    <div>

       <label for="name" class="form-label">Your name</label>

       <input type="text" id="name" name="name" required>

    </div>

    <div>

      <label for="text" class="form-label">Your message</label>

       <textarea id="text" name="text" rows="3" required></textarea>

    </div>

     <input type="submit" value="Send">

</form>

Pak si jej můžete dostylovat.

Dále si vytvořte v kořenové složce soubor **send-mail.php** a podle toho, jak jste si do projektu stáhnuli PM:

1. pokud jste si jej stáhnuli **manuálně**, nataháme potřebné třídy PM z rozbalené složky. Cestu si k souborům upravte podle vlastní skutečnosti.

<?php

require "./PHPMailer-master/PHPMailer-master/src/Exception.php";

require "./PHPMailer-master/PHPMailer-master/src/PHPMailer.php";

require "./PHPMailer-master/PHPMailer-master/src/SMTP.php";

use PHPMailer\PHPMailer\PHPMailer;

use PHPMailer\PHPMailer\SMTP;

use PHPMailer\PHPMailer\Exception;

1. pokud jste šli cestou **Composeru**, tak prosím takto

<?php

require "./vendor/autoload.php";

use PHPMailer\PHPMailer\PHPMailer;

use PHPMailer\PHPMailer\SMTP;

use PHPMailer\PHPMailer\Exception;

* třída Exception spravuje výjimky vyvolané knihovnou PHPMailer
* hlavní třída PHPMailer obsahuje hlavní funkcionality pro odeslání emailů
* třída SMTP zaštiťuje komunikaci s SMTP serverem

## Příjem dat z formuláře

Zatím si nahrubo nastrouháme kód, učešeme si jej později i s bezpečnostními prvky.

$email = $\_POST["email"];

$name = $\_POST["name"];

$message = $\_POST["text"];

echo $email . ", " . $name . ", " . $message;

Echem byste si měli ověřit, zda vám skript funguje, ale tady zatím žádný chyták není, mělo by to být v cajku. Pamatujte, že lokální server Apache musí být spuštěn a stránku index.html si spouštíte přes localhost (např. místo my-project si napište název své projektové složky: *localhost/my-project/*).

Echo si po úspěšném testu můžete zakomentovat nebo smazat.

## Nastavení SMTP serveru – Gmail

Pro tento krok doporučuji zřídit testovací účet u Googlu. Jakmile si jej vytvoříte, můžeme zkusit napsat tento kus kódu a uvidíme, jak se nám povede. Pro ukázkové účely náš testovací Google účet bude *example@gmail.com* s heslem *examplepassword*. Vy si samozřejmě zřiďte podle vlastního uvážení.

$mail = new *PHPMailer*(true);

$mail->SMTPDebug = *SMTP*::DEBUG\_SERVER;

$mail->isSMTP();

$mail->SMTPAuth = true;

$mail->Host = "smtp.gmail.com";

$mail->SMTPSecure = *PHPMailer*::ENCRYPTION\_STARTTLS;

$mail->Port = 587;

$mail->Username = "example@gmail.com";

$mail->Password = "examplepassword";

$mail->CharSet = 'UTF-8';

$mail->Encoding = 'base64';

$mail->setFrom($email, $name);

$mail->addAddress("recipient@mail.com");

$mail->Subject = "Email from your website";

$mail->Body = $message;

$mail->send();

echo "Message has been sent";

A nyní si některé kroky popíšeme:

* $mail = new *PHPMailer*(true);

Vytvoří novou instanci třídy PHPMailer. Parametr true zapíná výjimky, což znamená, že chyby budou vyvolávat výjimky, které lze zpracovat.

* $mail->SMTPDebug = *SMTP*::DEBUG\_SERVER;

Toto nastavení určuje úroveň ladění. SMTP::DEBUG\_SERVER poskytuje podrobné informace o komunikaci s poštovním serverem, což je užitečné pro diagnostiku problémů. Jak zjistíme v dalším kroku, velice užitečné.

* $mail->isSMTP();

$mail->SMTPAuth = true;

isSMTP(): Nastaví PHPMailer, aby používal SMTP protokol pro odesílání e-mailů.

SMTPAuth = true: Povolí autentizaci SMTP. To znamená, že musíte zadat uživatelské jméno a heslo.

* $mail->Host = "smtp.gmail.com";

$mail->SMTPSecure = *PHPMailer*::ENCRYPTION\_STARTTLS;

$mail->Port = 587;

Host: Nastaví adresu SMTP serveru.

SMTPSecure: Nastaví typ šifrování. ENCRYPTION\_STARTTLS znamená, že se použije TLS šifrování.

Port: Nastaví port pro komunikaci se SMTP serverem. Port 587 je běžně používán pro TLS.

* $mail->CharSet = 'UTF-8';

$mail->Encoding = 'base64';

Nastavení znakové sady a kódování.

* $mail->setFrom($email, $name);

$mail->addAddress("recipient@mail.com");

$mail->Subject = "Email from your website";

$mail->Body = $message;

Nastavení samotné zprávy – od koho, komu, předmět zprávy a samotná zpráva. Komu: nastavíte emailovou adresu, kam chcete, aby zprávy z vaší stránky chodily.

* $mail->send();

echo "Message has been sent";

send(): Odešle e-mail. Pokud dojde k chybě, bude vyvolána výjimka.

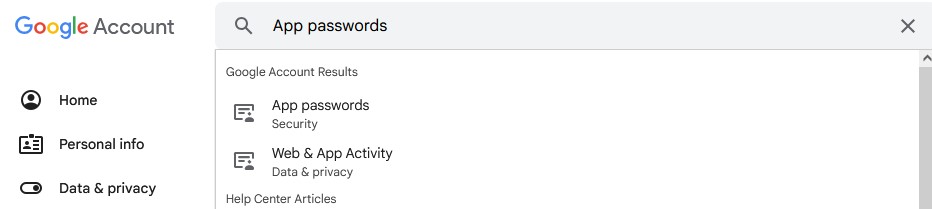
echo "email sent": Vypíše potvrzení, že e-mail byl odeslán.

Pokud si zkusíte odeslat první mail přes formulář:

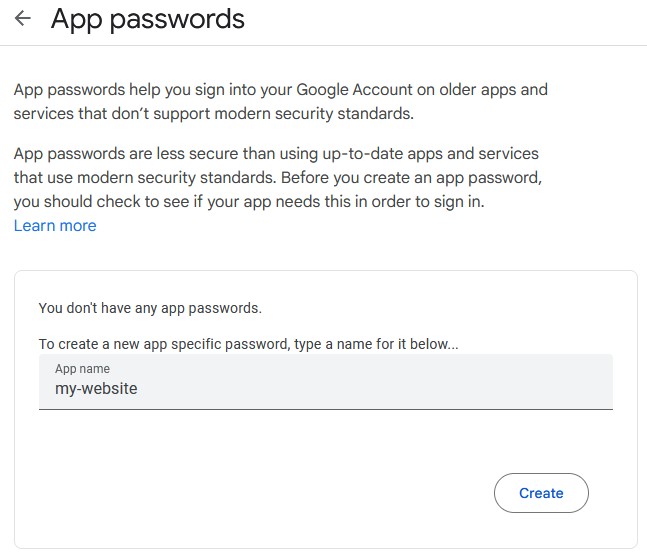
* jednak na vás vyskočí přehled komunikace s poštovním serverem a už zde vidíme, že je problém s autentikací
* také na vás pod komunikací se serverem vyskočí chyba Fatal error

Je třeba udělat ještě jeden krok. Vraťte se do nastavení vašeho nového testovacího účtu na Google a nastavte si dvoufázové ověření.

Poté je třeba si nastavit jednu skrytou funkci, ke které se neproklikáte z nabídky. Napište do vyhledávacího pole klíčová slova **App passwords**. Mělo by vám to nabídnout nastavení přímo s tímto pojmenováním.



Klikněte a vyskočí na vás průvodce, viz další obr.



Pojmenujte si generované heslo třeba *my-website.* A vytvořte jej.

Vygenerované heslo si zkopírujte a bez mezer jím nahraďte původní heslo pro přihlášení.

$mail->Password = "newapppassword";

Zkuste teď z formuláře odeslat email a již by vám měl přijít, jelikož komunikace s poštovním serverem nevykazuje žádné chyby a pod ní se zobrazilo *email sent*. Přišel? Pecka!

## Validace a bezpečnost vstupních dat

Doposud jsme si nelámali hlavu s validací a bezpečností vstupních dat. Kód si tedy upravíme, abychom splnili tyto požadavky.

$email = filter\_var($\_POST["email"], FILTER\_SANITIZE\_EMAIL);

**filter\_var**: Tato funkce se používá k filtrování a validaci vstupních dat.

**FILTER\_SANITIZE\_EMAIL**: Tento filtr odstraní všechny neplatné znaky z e-mailové adresy, které by mohly být použity k útoku nebo narušení aplikace. Zajistí, že vstup obsahuje pouze platné znaky pro e-mailovou adresu.

$name = htmlspecialchars(trim($\_POST["name"]), ENT\_QUOTES, 'UTF-8');

$message = htmlspecialchars(trim($\_POST["text"]), ENT\_QUOTES, 'UTF-8');

**trim**: Tato funkce odstraní bílé znaky (mezery, tabulátory, nové řádky) z obou stran vstupu.

**htmlspecialchars**: Tato funkce převádí speciální znaky na HTML entity, aby se zabránilo XSS (Cross-Site Scripting) útokům. Například znak < se převede na &lt;, což zabrání tomu, aby byl interpretován jako HTML značka.

**ENT\_QUOTES**: Tento parametr zajišťuje, že také budou převedeny uvozovky (jednoduché a dvojité) na HTML entity.

A nakonec specifikujeme použití znakové sady.

Přidejte si do skriptu pod deklaraci proměnných tyto části kódu.

if (!filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {

die("Invalid email address.");

}

Proženeme email odesílatele dalším filtrem **FILTER\_VALIDATE\_EMAIL**, tentokrát pro validaci. Pokud emailová adresa není validní, vykonávání skriptu se přeruší.

if (!preg\_match("/^[a-zA-Zá-žÁ-Ž\s]+$/", $name)) {

die("Invalid name. Only letters and spaces are allowed.");

}

**preg\_match**: Tato funkce provádí kontrolu podle regulárního výrazu na daném vstupu. Vrací true, pokud vstup odpovídá vzoru, a false, pokud ne.

**"/^[a-zA-Zá-žÁ-Ž\s]+$/"**: Toto je regulární výraz, který definuje povolené znaky ve jménu. Pojďme si ho rozebrat:

* **^**: Značí začátek řetězce.
* **[a-zA-Zá-žÁ-Ž\s]**: Značí množinu povolených znaků:
  + a-z: Malá písmena anglické abecedy.
  + A-Z: Velká písmena anglické abecedy.
  + á-ž: Malá písmena české abecedy s diakritikou.
  + Á-Ž: Velká písmena české abecedy s diakritikou.
  + \s: Bílý znak (mezera).
* **+**: Značí, že se předchozí skupina znaků musí vyskytovat jednou nebo vícekrát.
* **$**: Značí konec řetězce.

**$name**: Toto je vstupní hodnota, která se kontroluje proti regulárnímu výrazu.

if (strlen($message) < 10) {

die("Message is too short. Please enter at least 10 characters.");

}

Zde kontrolujeme, zda délka řetězce zprávy není kratší než 10 znaků.

## Úprava těla emailu

Chceme-li, aby byl email přehledný pro nás, můžeme si jeho tělo strukturovat pomocí html tagů. Plus si musíme přidat před deklaraci těla ještě jeden řádek kódu, kterým říkáme, že má text těla interpretovat jako html. Přihodíme si i deklaraci body v prostém textu pro klienty, kteří si s html neporadí.

$mail->isHTML(true);

$mail->Body = "<p>You have received a new message from your website contact

form.

</p>

               <p><strong>Name:</strong> $name</p>

               <p><strong>Email:</strong> $email</p>

               <p><strong>Message:</strong><br>$message</p>";

$mail->AltBody = "You have received a new message from your website contact

form.\n\n" .

"Name: $name\n" .

"Email: $email\n\n" .

"Message:\n$message";

pzn.: Debugovat komunikaci s poštovním serverem již potřebovat nebudeme. Zakomentujte si tedy tyto dva řádky kódu:

// use PHPMailer\PHPMailer\SMTP;

// $mail->SMTPDebug = SMTP::DEBUG\_SERVER;

## Výjimky a přesměrování ze skriptu

Kód si nyní upravíme, aby zachytával a vypsal výjimky. Použijeme blok try-catch.

try {

    $mail = new *PHPMailer*(true);

    // $mail->SMTPDebug = SMTP::DEBUG\_SERVER;

    $mail->isSMTP();

    $mail->SMTPAuth = true;

    $mail->Host = "smtp.gmail.com";

    $mail->SMTPSecure = *PHPMailer*::ENCRYPTION\_STARTTLS;

    $mail->Port = 587;

  $mail->Username = "example@gmail.com";

$mail->Password = "newapppassword";

    $mail->CharSet = 'UTF-8';

    $mail->Encoding = 'base64';

    $mail->setFrom($email, $name);

    $mail->addAddress("recipient@mail.com");

    $mail->Subject = "Email from your website";

    $mail->isHTML(true);

    $mail->Body = "<p>You have received a new message from your website

contact form.

</p>

                   <p><strong>Name:</strong> $name</p>

                   <p><strong>Email:</strong> $email</p>

                   <p><strong>Message:</strong><br> $message</p>";

$mail->AltBody = "You have received a new message from your website

contact form.\n\n" .

"Name: $name\n" .

"Email: $email\n\n" .

"Message:\n$message";

    $mail->send();

    echo "Message has been sent";

} catch (*Exception* $e) {

    echo "Message could not be sent. Mailer Error: {$mail->ErrorInfo}";

}

K dokonalosti nám už chybí jen jedna věc – nechceme zůstat ve skriptu, ale po vykonání skriptu bychom rádi zpátky na formulář. Přeci jen, pokud budete mít formulář na své osobní stránce, návštěvník nechtěl po odeslání zprávy stránku opustit.

Řádky s echo a die si zakomentujte. Teď si určitě ťukáte na hlavy, proč jsme si ty hlášky vlastně psali, když je nevyužijeme. Využít je můžete. Třeba tím, že si všechna použití skriptu budete zalogovávat do databáze. To už nechám na vás. Místo echo a die si vložte tento code.

header("Location: ./index.html");

exit();

Funkcí *header()* s takto definovaným parametrem řekneme prohlížeči, aby načetl stránku *index.html.* Pomocí funkce *exit()* zastavíme vykonávání skriptu.

Takže nyní se po odeslání i neodeslání emailu vracíme zpět na formulář. Co nám chybí ještě k dokonalému štěstí? Uživatel by asi rád věděl, zda se jeho zpráva odeslala nebo ne. Stránku máme ve formátu .html a nechceme ji jen kvůli hlášce z PHP skriptu měnit na .php soubor. Pomůžeme si tedy našim dobrým FE kamarádem JavaScriptem.

Začneme tím, že pošleme stránce s formulářem parametr v URL.

v případě úspěchu:

header("Location: ./index.html?status=success");

v případě neúspěchu:

header("Location: ./index.html?status=error");

Upravte si tedy všechna přesměrování.

Toto si již JS na straně frontendu může přebrat.

Nad formulář si v body vepište tento code.

<div id="alert-placeholder"></div>

A na konec body tento skript.

<script>

  document.addEventListener('DOMContentLoaded', () *=>* {

*const* params = new *URLSearchParams*(window.location.search);

*const* alertPlaceholder = document.querySelector('#alert-placeholder');

     if (params.has('status')) {

*let* message = '';

*let* alertType = '';

        if (params.get('status') === 'success') {

            message = 'Email has been sent successfully.';

      } else if (params.get('status') === 'error') {

            message = 'There was an error sending the email. Try again.';

        }

        if (message) {

*const* alertDiv = document.createElement('div');

            alertDiv.innerText = message;

            alertPlaceholder.appendChild(alertDiv);

        }

      }

   });

</script>

## Přesun přihlašovacích údajů

Přihlašovací údaje k SMTP serveru si umístíme mimo hlavní skript, aby byl recyklovatelný a abychom neměli tyto citlivé údaje utopené někde v kódu.

Vytvořte si v kořenu projektu novou složku **config.php** – tento skript bude vracet asociativní pole s aktuálními SMTP přihlašovacími údaji**.**

<?php

return [

    'smtp\_host' => 'smtp.gmail.com',

    'smtp\_port' => 587,

    'smtp\_username' => 'example@gmail.com',

    'smtp\_password' => 'newapppassword'

];

V hlavním skriptu **send-mail.php** načteme toto asociativní pole do proměnné $config.

$config = require './config.php';

A nezapomeneme nahradit příslušné řádky kódu.

$mail->Host = $config['smtp\_host'];

$mail->Port = $config['smtp\_port'];

$mail->Username = $config['smtp\_username'];

$mail->Password = $config['smtp\_password'];

Hotovo! Funguje-li lokálně, bude vám fungovat i na webhostingu. Zde je link na code na GitHub.

[github.com/veetecheck/PHPMailer\_contact\_form](https://github.com/veetecheck/PHPMailer_contact_form)

# PHP API a frontend v jiné technologii (React)

Prerekvizity – VS Code, Xampp se spuštěným Apache, MySQL. Plus dobrou náladu a chuť programovat.

Kód FE React aplikace, který se bude pro potřeby propojení s backendem dále upravovat:

<https://github.com/veetecheck/cars>

## Tabulka v db MySQL

Ve vámi vybrané databázi na lokálním MySQL si vytvořte tabulku **cars.**

## Obsah obrázku snímek obrazovky, text, číslo, Paralelní Popis byl vytvořen automaticky

struktura:

* id – int(5), not null, auto increment, primární klíč
* brand – varchar(20)
* model – varchar(20)
* reg – varchar(7)
* km – int(6)
* year – int(4)

A vložte si tam nějaká data.

INSERT INTO `cars` (`brand`, `model`, `reg`, `km`, `year`) VALUES

('Škoda', 'Scala', '2AP0809', 159212, 2017),

('Honda', 'Civic', '9T64589', 101515, 2005),

('Škoda', 'Fabia', '6MA8915', 230555, 2012),

('Seat', 'Ibiza', '8TR9634', 159453, 2008),

('Škoda', 'Felicia', '1T15825', 256996, 1999),

('Škoda', 'Felicia', '1L26828', 228249, 2000),

('Seat', 'Leon', '1AN5586', 152023, 2016),

('Kia', 'Schuma', '2T55896', 299021, 1999),

('Škoda', 'Octavia', '2A18526', 251666, 1999);

## Zprovoznění PHP backendu pomocí PDO a výpis aut do JSON

V kurzu jsme doposud používali funkcionální přístup pro práci s databází MySQL pomocí mysqli funkcí. My si nyní ukážeme objektově orientovaný přístup, který je v dnešní době preferovaný.

Pro náš případ použijeme objektový ovladač PDO – zkratka pro *PHP data object*. Jeho nespornou výhodou je, že poskytuje jednotný přístup k různým databázovým systémům. Čili, pokud se rozhodnete pro jinou databázi, než je MySQL, nebo pokud budete v rámci jedné aplikace používat více db systémů, můžete stále používat PDO.

PDO je jinými slovy, pro větší představu, objektově orientované rozhraní pro vykonávání dotazů, manipulaci s výsledky a chybový management.

Oficiální manuál k PDO (anglicky): <https://www.php.net/manual/en/book.pdo.php>

Vytvoříme si soubor **DbConnect.php**, kde si definujeme třídu DbConnect, která slouží k navázání spojení s databází. Soubor si vytvořte ve vámi zvolené složce v htdocs.

<?php

class DbConnect

{

    private $server = 'localhost';

    private $dbname = 'cars';

    private $user = 'root';

    private $pass = '';

    private $options = array(

        PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION,

        PDO::ATTR\_EMULATE\_PREPARES => false,

    );

    public function connect()

    {

        try {

$conn = new PDO('mysql:host=' . $this->server .

';dbname=' . $this->dbname . ';charset=utf8',

$this->user, $this->pass, $this->options );

            return $conn;

        } catch (PDOException $e) {

            echo "Database Error: " . $e->getMessage();

        }

    }

}

Jako první si definujeme vlastnosti třídy, které jsou nutné pro připojení k databázi – server (**$server**), název databáze (**$dbname**), uživatelské jméno (**$user**), heslo (**$pass**) a pole možností (**$options**). Všechny tyto vlastnosti jsou typu private, což znamená, že získání hodnoty vlastností, nebo jejich změna, je možná pouze v případě existence speciálních metod třídy, které by toto umožnily. My tyto údaje nechceme ani měnit, ani se s nimi chlubit. Proto zůstanou natvrdo napsány v rámci této třídy a budou private.

Vlastnost **$options** je polem možností:

* PDO::ATTR\_ERRMODE nastaven na PDO::ERRMODE\_EXCEPTION pro vyvolání výjimky v případě chyby
* PDO::ATTR\_EMULATE\_PREPARES nastaven na *false* zvyšuje bezpečnost tím, že vynucuje použití nativních připravených dotazů databázovým enginem (což může pomoci chránit proti SQL injection útokům)

Dále si definujeme metodu **connect()**, která zajišťuje spojení s databází. Abychom zajistili správné zpracování výjimek, použijeme blok try – catch, což umožňuje PHP kódu "zkusit" vykonat určité operace, které mohou vyvolat výjimku (zde pokus o připojení k databázi pomocí PDO). Pokud během těchto operací dojde k výjimce, provádění kódu se okamžitě přesune do nejbližšího odpovídajícího bloku catch, kde můžete chybu zpracovat (např. vypsat chybovou zprávu).

Metoda vytvoří novou instanci PDO objektu s použitím přihlašovacích údajů a možností uložených ve vlastnostech třídy DbConnect. Řetězec DSN (Data Source Name), který je prvním parametrem konstruktoru PDO, je dynamicky sestaven z vlastností $server a $dbname, a je specifikováno kódování utf8.

Pokud je spojení úspěšně navázáno, metoda connect vrátí objekt PDO reprezentující připojení k databázi.

Pokud dojde k chybě při pokusu o připojení k databázi, chyba je zachycena pomocí bloku catch, který zachytí výjimku typu PDOException. Chybová zpráva je poté vypsána pomocí echo.

Nyní si vytvořte ve stejné složce soubor **index.php**.

<?php

header("Access-Control-Allow-Origin: \*");

header("Access-Control-Allow-Headers: \*");

header("Access-Control-Allow-Methods: GET, POST, PUT, DELETE");

header('Content-Type: application/json; charset=utf-8');

include('./DbConnect.php');

$connection = new DbConnect();

$database = $connection->connect();

$sql = "SELECT \* FROM cars";

$stmt = $database->prepare($sql);

$stmt->execute();

$cars = $stmt->fetchAll(PDO::FETCH\_ASSOC);

echo json\_encode($cars, JSON\_UNESCAPED\_UNICODE);

Hlavičky HTTP (HTTP headers) jsou klíčovou součástí HTTP požadavků a odpovědí. Jsou to jednoduše páry klíč-hodnota, které předávají důležité informace mezi klientem (např. webovým prohlížečem nebo jinou aplikací) a serverem. Hlavičky mohou být použity jak v požadavcích odesílaných na server, tak v odpovědích zasílaných zpět klientovi.

* header("Access-Control-Allow-Origin: \*");

Tento řádek umožňuje CORS (Cross-Origin Resource Sharing) pro všechny domény. To znamená, že váš skript může být volán z jakékoli jiné domény ve webovém prohlížeči.

* header("Access-Control-Allow-Headers: \*");

Tímto se povoluje, aby jakékoli hlavičky mohly být posílány v requestu na váš server.

* header("Access-Control-Allow-Methods: GET, POST, PUT, DELETE");

Specifikuje, které HTTP metody jsou povoleny při přístupu k prostředku. Tady jsou povoleny metody GET, POST, PUT a DELETE. Tímto se připravujeme na další kroky v kódu.

* header('Content-Type: application/json; charset=utf-8');

Nastavuje Content-Type response na application/json s kódováním UTF-8.

$connection = new DbConnect();

Vytvoří novou instanci třídy DbConnect. Třídu máme k dispozici pomocí include z DbConnect.php.

$database = $connection->connect();

Volá metodu connect() na instanci DbConnect, která by měla vrátit objekt PDO reprezentující připojení k databázi.

$sql = "SELECT \* FROM cars";

SQL dotaz pro výběr všech záznamů z tabulky cars.

$stmt = $database->prepare($sql);

Připraví SQL dotaz pro bezpečné spuštění, čímž se chrání před SQL injections.

$stmt->execute();

Metoda execute spustí připravený SQL dotaz.

$cars = $stmt->fetchAll(PDO::FETCH\_ASSOC);

Načte všechny záznamy vrácené dotazem jako asociativní pole, kde klíče pole odpovídají názvům sloupců v databázi.

echo json\_encode($cars, JSON\_UNESCAPED\_UNICODE);

Převede data z asociativního pole do formátu JSON a vypíše je. Použití JSON\_UNESCAPED\_UNICODE zajistí, že unicode znaky budou správně zakódovány. Bez tohoto to dělalo na hostingu paseku a české znaky se vypisovaly chybně.

Pokud si nyní spustíte script na localhostu, měli byste vidět JSON v podobě pole aut.

Čili, teď máme přesně to, co jsme si importovali ve formě JSON souboru do React aplikace, když jsme vytvářeli čistý frontend. Jen s tím rozdílem, že teď dostáváme data přímo z databáze a máme je vypsané na konkrétní url.

## Rozšíření skriptu pomocí metod a akcí

My se samozřejmě neomezíme jen na výpis všech aut – budeme chtít výpis konkrétních aut na základě čísel id z frontendu, budeme chtít záznamy do db vkládat, upravovat, mazat.

Toho docílíme na základě metod a akcí, které použijeme v rámci http requestů.

$method = $\_SERVER['REQUEST\_METHOD'];

Hodnota je získána ze superglobálního pole $\_SERVER, které obsahuje informace o serveru a prostředí spuštění. Klíč 'REQUEST\_METHOD' v tomto poli obsahuje metodu použitou pro aktuální požadavek, například GET, POST, PUT, DELETE atd.

### Metoda GET a akce getAll a getSpec

Můžeme si index.php přepsat takto:

<?php

header("Access-Control-Allow-Origin: \*");

header("Access-Control-Allow-Headers: \*");

header("Access-Control-Allow-Methods: GET, POST, PUT, DELETE");

header('Content-Type: application/json; charset=utf-8');

include('./DbConnect.php');

$connection = new DbConnect();

$database = $connection->connect();

$method = $\_SERVER['REQUEST\_METHOD'];

switch ($method) {

case 'GET':

$action = $\_GET['action'];

if ($action == 'getAll'){

$sql = "SELECT \* FROM cars";

$stmt = $database->prepare($sql);

$stmt->execute();

$cars = $stmt->fetchAll(PDO::FETCH\_ASSOC);

echo json\_encode($cars, JSON\_UNESCAPED\_UNICODE);

} else if ($action == 'getSpec') {

$idsParam = isset($\_GET['ids']) ? $\_GET['ids'] : '';

$ids = explode(',', $idsParam);

$ids = array\_filter($ids, function ($value) {

return $value !== '';

});

$ids = implode(',', array\_map('intval', $ids));

if (!empty($ids)) {

$sql = "SELECT \* FROM cars WHERE id IN ($ids)";

$stmt = $database->prepare($sql);

$stmt->execute();

$cars = $stmt->fetchAll(PDO::FETCH\_ASSOC);

echo json\_encode($cars, JSON\_UNESCAPED\_UNICODE);

} else {

echo json\_encode([]);

}

}

break;

case 'POST':

break;

case 'DELETE':

break;

case 'PUT':

break;

default:

break;

}

Na základě akce rozlišujeme typy GET metody. Přeci jen chceme jednak získat data o všech autech a pak budeme potřebovat jen nějaká na základě filtru.

$action = $\_GET['action'];

Takto si uložíme hodnotu atributu action z query stringu URL. Za předpokladu, že mám index.php v htdocs uložen ve složce cars (vy můžete mít jinak), pak URL mohou vypadat takto:

'http://localhost/cars/?action=getAll'

'http://localhost/cars/?action=getSpec&ids=1,2,3'

Co se týče getSpec metody – předpoklad je, že nám přijde parametrem string čísel id aut oddělené čárkou. Tento string si pomocí explode nalijeme do pole, zjistíme, zda jsou položky validní a zpátky z něj uděláme string pro přidání do sql dotazu za operátor IN.

Pojďme se na to kouknout z druhé strany barikády. V React aplikaci si otevřte App.jsx pro editaci. Vyměníme čtení dat z JSON file na čtení z db pomocí http requestu.

Otevřete si terminál a nainstalujte si npm balíček Axios (<https://axios-http.com/docs/intro>).

npm install axios

Axios je JavaScriptová knihovna používaná v aplikacích pro interakci s backendovými API pomocí http requestů. Funguje asynchronně na základě JS promises. A nezapomeňte na import do App.jsx.

import axios from "axios";

Přepišme si iniciační hodnoty cars a carsToShow na prázdná pole.

const [cars, setCars] = useState([]);

const [carsToShow, setCarsToShow] = useState([]);

Můžeme si napsat rovnou funkci http requestu pomocí Axios.

const getCars = () => {

    axios.get('http://localhost/cars/?action=getAll').then((response) => {

      if (Array.isArray(response.data)) {

        setCars(response.data);

        setCarsToShow([]);

      } else {

        console.error("Odpověď serveru není pole.");

      }

    }).catch((error) => {

      console.error("There was an error!", error);

      alert(`Chyba: ${error}`);

    });

}

Funkce getCars asynchronně načítá seznam aut ze serveru pomocí Axios, ověřuje, že odpověď je ve správném formátu (pole), a aktualizuje stav komponenty na základě těchto dat. Pokud dojde k chybě, informuje uživatele a zároveň vypíše chybu do konzole.

useEffect(() => { getCars(); }, []);

Pro zavolání funkce při zavedení komponenty použijeme useEffect hook.

const filterCars = (ids) => {

    const param = ids.join();

    console.log(param);

    axios.get(`http://localhost/cars/?action=getSpec&ids=${param}`)

.then((response) => {

      if (Array.isArray(response.data)) {

        setCarsToShow(response.data);

      } else {

        console.error("Odpověď serveru není pole.");

      }

    }).catch((error) => {

      console.error("There was an error!", error);

      alert(`Chyba: ${error}`);

    });

  }

Funkce filterCars je mutací na fci getCars, ale s tím rozdílem, že posílá v těle url i parametr ids, což je string čísel id aut oddělěných čárkou.

const handleFilterData = (filteredCars) => {

const ids = filteredCars.map((car) => car.id);

filterCars(ids);

};

Nyní si upravíme fci handleFilterData, aby filtrovaná data šla rovnou z databáze za použití nové fce filterCars.

### Metoda DELETE

Obdobou předchozích funkcí v App.jsx je deleteCar. Všimněte si ovšem změny – URL odesílá id auta pro výmaz z db jinak než v případě filterCars. Ne jako parametr, ale již jako součást URL.

const deleteCar = (id) => {

    axios.delete(`http://localhost/cars/${id}`).then((response) => {

      console.log(response.data);

      getCars();

      alert("Auto úspěšně smazáno.");

    }).catch((error) => {

      console.error("There was an error!", error);

    });

}

URL může vypadat takto, pokud budeme mazat auto s id = 5:

'http://localhost/cars/5'

To nám ale bude dělat paseku, jelikož server bude hledat v cars složce položku 5, která tam fyzicky není. U parametrů nám to nevadilo. V tomto případě již to problém je. Musíme si tedy pomoci nastavením na straně serveru.

Pomůžeme si novým souborem **.htaccess**¸ který umístíme do stejné složky jako index.php. V něm bude následující code:

RewriteEngine On

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteRule ^ index.php [L]

Pro webový server Apache tento soubor zajišťuje, aby všechny http requesty byly přesměrovány na konkrétní vstupní bod.

RewriteEngine On

zapíná modul přepisování URL

RewriteCond %{REQUEST\_FILENAME} !-d

kontroluje, zda zadaná cesta není existující složka

RewriteCond %{REQUEST\_FILENAME} !-f

kontroluje, zda zadaná cesta nevede k existujícímu souboru

RewriteRule ^ index.php [L]

no a pokud ani jedna z výše zmíněných podmínek není splněna – url bude přesměrována na index.php

Nyní tedy URL s id auta ke smazání povede rovnou do index.php. A jelikož axios říká, jaká metoda se používá, tedy DELETE. Můžeme napsat v index.php pravidlo pro smazání auta z databáze.

case 'DELETE':

$requestPath = $\_SERVER['REQUEST\_URI'];

$pathSegments = explode('/', trim($requestPath, '/'));

$carId = (int) $pathSegments[count($pathSegments) - 1];

if ($carId > 0) {

$sql = "DELETE FROM cars WHERE id= :id";

$stmt = $database->prepare($sql);

$stmt->bindParam(':id', $carId, PDO::PARAM\_INT);

if ($stmt->execute()) {

$data = ['status' => 1, 'message' => "Car deleted"];

} else {

$data = ['status' => 0, 'message' => "Failed to delete car."];

}

} else {

$data = ['status' => 0, 'message' => "Id not numeric."];

}

echo json\_encode($data);

break;

Něco málo na vysvětlenou k začátku kódu, pomůžeme si var\_dump pro výpis hodnot:

$requestPath = $\_SERVER['REQUEST\_URI'];

var\_dump($requestPath);

vrací string requestu z url

string(7) "/cars/5"

$pathSegments = explode('/', trim($requestPath, '/'));

var\_dump($pathSegments);

vrací pole segmentů ořezané o lomítka, aby tam nebyly prázdné prvky

array(2) {

[0]=>

string(4) "cars"

[1]=>

string(1) "5"

}

No a nás zajímá poslední prvek z pole. Vyzkoušíme, zda je int a větší než nula. A můžeme mazat.

V App.jsx si upravíme už jen fci handleDelete a hotovo.

const handleDelete = (idToDel) => {

    deleteCar(idToDel);

};

### Metoda POST

V případě metody POST, kdy chceme do databáze vložit nový záznam auta, předáváme údaje o autě formou parametru funkce.

const insertCar = (car) => {

    axios.post('http://localhost/cars/', car).then((response) => {

      console.log(response.data);

      getCars();

      alert("Auto úspěšně přidáno.");

    }).catch((error) => {

      console.error("There was an error!", error);

      alert(`Chyba: ${error}`);

    });

}

Nutno podotknout, že je třeba změnit defaultní hodnoty proměnné newCar, jelikož už nezadáváme id na frontendu, to si teď řeší sama databáze.

const [newCar, setNewCar] = useState({

    brand: "",

    model: "",

    reg: "",

    km: "",

    year: "",

});

V index.php si přidáme možnost POST do switche.

case 'POST':

    $car = json\_decode(file\_get\_contents('php://input'));

$sql = "INSERT INTO cars(brand, model, reg, km, year) values(:brand, :model, :reg, :km, :year)";

      $stmt = $database->prepare($sql);

      $stmt->bindParam(':brand', $car->brand);

      $stmt->bindParam(':model', $car->model);

      $stmt->bindParam(':reg', $car->reg);

      $stmt->bindParam(':km', $car->km);

      $stmt->bindParam(':year', $car->year);

      if ($stmt->execute()) {

          $data = ['status' => 1, 'message' => "Car successfully created"];

      } else {

          $data = ['status' => 0, 'message' => "Failed to create car."];

      }

      echo json\_encode($data);

      break;

Pro vysvětlení, jak dostat z requestu údaje o autě:

$car = json\_decode(file\_get\_contents('php://input'));

vytvoří PHP objekt z JSON dat, zaslaných v requestu

### Metoda PUT

A na závěr si přidáme metodu PUT, která se stará o aktualizaci existujícího záznamu. Funguje obdobně jako metoda POST.

const [carToChange, setCarToChange] = useState({

    id: 0,

    brand: "",

    model: "",

    reg: "",

    km: "",

    year: "",

});

Zde nám v rámci předávaného objektu jako parametru zůstává i vlastnost id, jelikož databáze potřebuje vědět nejlépe podle id, který záznam vlastně mění.

const updateCar = (car) => {

    axios.put('http://localhost/test/', car).then((response) => {

      console.log(response.data);

      getCars();

      alert("Auto úspěšně aktualizováno.");

    }).catch((error) => {

      console.error("There was an error!", error);

      alert(`Chyba: ${error}`);

    });

}

V rámci index.php je oproti POST metodě změna jen příkazu do databáze.

case "PUT":

    $car = json\_decode(file\_get\_contents('php://input'));

$sql = "UPDATE cars SET brand= :brand, model= :model, reg= :reg, km= :km, year= :year WHERE id= :id";

      $stmt = $database->prepare($sql);

      $stmt->bindParam(':id', $car->id);

      $stmt->bindParam(':brand', $car->brand);

      $stmt->bindParam(':model', $car->model);

      $stmt->bindParam(':reg', $car->reg);

      $stmt->bindParam(':km', $car->km);

      $stmt->bindParam(':year', $car->year);

      if ($stmt->execute()) {

            $data = ['status' => 1, 'message' => 'Car updated.'];

        } else {

            $data = ['status' => 0, 'message' => 'Failed to update car.'];

        }

      echo json\_encode($data);

      break;

Nyní už jen zbývá upravit fci handleUpdate v App.jsx:

const handleUpdate = (source) => {

    let temp;

    switch (source) {

      case "add-car-form": {

        temp = fillEmptyInfos(newCar);

        if (confirmCar(temp)) {

          insertCar(temp);

          setNewCar({

            brand: "",

            model: "",

            reg: "",

            km: "",

            year: "",

          });

          alert("Data byla úspěšně odeslána");

        } else {

          alert("Odeslání dat bylo zrušeno");

        }

        break;

      }

      case "change-car-form": {

        temp = fillEmptyInfos(carToChange);

        if (confirmCar(temp)) {

          const index = cars.findIndex((car) => car.id === temp.id);

          if (index !== -1) {

            updateCar(temp);

            setCarToChange({

              id: 0,

              brand: "",

              model: "",

              reg: "",

              km: "",

              year: "",

            });

            alert("Aktualizace dat úspěšná");

          } else {

            alert("Auto s daným id nebylo nalezeno");

            setCarToChange({

              id: 0,

              brand: "",

              model: "",

              reg: "",

              km: "",

              year: "",

            });

          }

        } else {

          alert("Aktualizace neproběhla");

        }

        break;

      }

      default:

        break;

    }

};

## Závěrem

A to jest vše. Před buildem aplikace nezapomeňte nastavit:

* homepage v package.json
* změnit cesty v Axios URL parametru na složku, kde máte index.php
* v DbConnect.php nastavit správně přihlašovací údaje k databázi

Odkaz na GitHub hotové aplikace: <https://github.com/veetecheck/cars-with-backend>

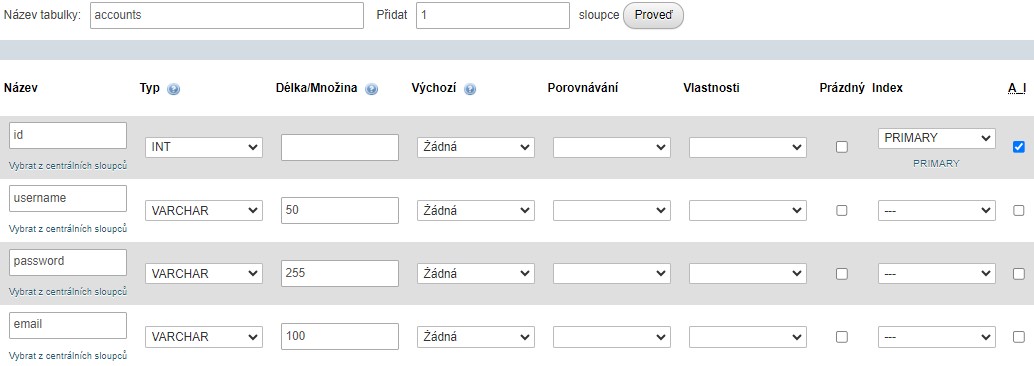
Odkaz na nasazenou appku: <https://veetektest.g6.cz/projects/cars/client/index.html>

# Registrace a přihlašování uživatele – PHP/MySQL

V této sekci se podíváme, jak využít zejména práci se session v případě velice jednoduchého registračního/přihlašovacího systému.

## Příprava databáze

V databázi si vytvořte tabulku s názvem *accounts*.



struktura tabulky:

* id – int(11), not null, auto increment, primární klíč
* username – varchar(50)
* password – varchar(255)
* email – varchar(100)

## Registrace nového uživatele

### dbConnect.php

Pro připojování k databázi si recyklujte/zkopírujte si soubor **dbConnect.php** ze strany 15, nezapomeňte však změnit případně název databáze, ve které je tabulka *accounts* umístěna.

### register.html

A můžeme psát náš první soubor **register.html**. Do těla stránky si vytvořte tento formulář a poznámky pro uživatele v odstavcích.

<h1>Registration</h1>

<form action="register.php" method="post">

    <input type="text" name="username" placeholder="Username" required>

    <input type="password" name="password" placeholder="Password" required>

    <input type="email" name="email" placeholder="Email" required>

    <input type="submit" value="Register">

</form>

<p>

Username must be 3-20 characters long and can contain letters, numbers, underscores, dots and dashes.

</p>

<p>

    Password must be between 8 and 20 characters long!

</p>

Máme tedy formulář, ve kterém uživatel zadá svoje uživatelské jméno, heslo a email, jakožto údaje, pod kterými se registruje do našeho mini systému.

### validation.php

Uděláme si speciální skriptík **validation.php**, kde si definujeme funkce validace uživatelských vstupů, které budeme následně používat v jiných skriptech.

<?php

*function* validateInput($data) {

    return isset($data) && !empty(trim($data));

}

*function* validateEmail($email) {

    if (!filter\_var($email, FILTER\_VALIDATE\_EMAIL)) {

        return 'Email is not valid!';

    }

    return true;

}

*function* validateUsername($username) {

    if (preg\_match('/^[a-zA-Z0-9.\_-]{3,20}$/', $username) == 0) {

        return 'Username is not valid! It must be 3-20 characters long and can contain letters, numbers, underscores, dashes, and dots.';

    }

    return true;

}

*function* validatePassword($password) {

    if (strlen($password) > 20 || strlen($password) < 8) {

        return 'Password must be between 8 and 20 characters long!';

    }

    return true;

}

* **validateInput** – fce vrátí boolean, zda jsou vůbec data zadána a zároveň nejsou prázdná. Takto můžeme kontrolovat každý input.
* **validateEmail** – fce kontroluje správnost formátu zadaného emailu, vrací true nebo hlášku, že email není validní.
* **validateUsername** – fce kontroluje délku řetězce a povolené znaky – jsou to písmena, číslice, tečka, pomlčka a podržítko. Vrací true nebo hlášku o špatném zadání.
* **validatePassword** – fce kontroluje délku řetězce. Vrací true nebo hlášku o špatném zadání.

### register.php

Pro zpracování formuláře si vytvoříme skript **register.php**.

<?php

include('DbConnect.php');

include('validation.php');  // Zahrnutí souboru s validacemi

if (!validateInput($\_POST['username']) || !validateInput($\_POST['password']) || !validateInput($\_POST['email'])) {

    exit('Please fill the form correctly.');

}

if (($error = validateEmail($\_POST['email'])) !== true) {

    exit($error);

}

if (($error = validateUsername($\_POST['username'])) !== true) {

    exit($error);

}

if (($error = validatePassword($\_POST['password'])) !== true) {

    exit($error);

}

$username = $\_POST['username'];

$password = $\_POST['password'];

$email = $\_POST['email'];

?>

Jak vidíte, první část je ryze validační, kde používáme právě ty vytvořené funkce ze skriptu *validation.php*. Pokud bude něco v uživatelském vstupu špatně, vyhodí vás to ze skriptu s příslušnou hláškou. V kódu, který následuje, již řešíme část skriptu po validaci inputů.

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Register</title>

</head>

<body>

    <?php

    $db = new *DbConnect*();

    $conn = $db->connect();

    // kontrola, zda už username neexistuje

    $sql = 'SELECT COUNT(\*) FROM accounts WHERE username = :username';

    if ($stmt = $conn->prepare($sql)) {

        $stmt->bindParam(':username', $\_POST['username'], *PDO*::PARAM\_STR);

        $stmt->execute();

        $userExists = $stmt->fetchColumn();

        if ($userExists > 0) {

exit('<p>Username already exists. Please choose another one.</p></body></html>');

        } else {

            // funguje

$sql = 'INSERT INTO accounts (username, password, email) VALUES (:username, :password, :email)';

            if ($stmt = $conn->prepare($sql)) {

$hashedPassword = password\_hash($password, PASSWORD\_DEFAULT);

                $stmt->bindParam(':username', $username, *PDO*::PARAM\_STR);

$stmt->bindParam(':password', $hashedPassword, *PDO*::PARAM\_STR);

                $stmt->bindParam(':email', $email, *PDO*::PARAM\_STR);

                $stmt->execute();

                echo '<p>You have successfully registered!</p>';

                echo '<a href="index.php">Go to login page</a>';

            } else {

exit("<p>Error: Could not prepare statement</p></body></html>");

            }

        }

    } else {

exit("<p>Error: Could not prepare statement</p></body></html>");

    }

    ?>

</body>

</html>

Jako první se připojíme k databázi a zkontrolujeme dotazem, zda náhodou již zvolené username v databázi již existuje. Toho docílíme tím, že pomocí COUNT(\*) v dotazu spočítáme, kolik řádků pro daného uživatele nám vrátí.

Funkce *fetchColumn()*, dáme-li ji bez parametru, nám vezme hodnotu z prvního sloupce dotazu. My máme výsledek právě jeden sloupec i řádek. Pokud číslo bude 0, pak uživatel v db vedený není. Pokud to bude nějaké jiné číslo (nemělo by být jiné než 1), pak uživatel bohužel již v db existuje.

Pokud uživatel ještě neexistuje, pokračuje skript v další databázové operaci – vložení údajů o novém userovi. A máme jej tam.

Povšimněte si, jakým způsobem pracujeme s heslem. Předtím, než jej jako řetězec zadáme do databáze, proženeme jej přes funkci *password\_hash()*. To nám zahashuje heslo a máme tím pádem splněn bezpečnostní prvek pro práci s hesly. V databázi je heslo uloženo v hashované formě a není možné jej zpětně zjistit. Nebojte, pro přihlášení si budete moci pamatovat původní zadanou verzi hesla, na to jsou fígle.

## Přihlášení uživatele

### index.php a auth.php

Vytvořte si v pracovní složce nový soubor **index.php**, nyní zatím s klasickým html tělem. V body umístěte tento code.

<h1>Login</h1>

<form action="auth.php" method="post">

    <input type="text" name="username" placeholder="Username" required>

    <input type="password" name="password" placeholder="Password" required>

    <input type="submit" value="Login">

</form>

<p>Not registered yet? Then register here...</p>

<a href="register.html">Register</a>

Takže klasický formulář se dvěma inputy – vyplnění username a hesla. A ještě jeden input pro odeslání formuláře. Formulář zpracujeme ve skriptu **auth.php**, který si teď napíšeme. Všimněte si, že pracujeme krom emailu s validací podobně jako v *register.php*. Jedním rozdílem, který vás udeří do očí patrně hned – vytváříme na začátku skriptu i **session**. Proč? Stránky budeme chtít samozřejmě, aby se chovaly jinak, když je někdo přihlášen. A informace o přihlášeném uživateli chceme a budeme uchovávat na straně serveru.

<?php

include('DbConnect.php');

include('validation.php');  // Zahrnutí souboru s validacemi

session\_start();

if (!validateInput($\_POST['username']) || !validateInput($\_POST['password'])) {

    exit('Please fill both the username and password fields!');

}

$username = $\_POST['username'];

$password = $\_POST['password'];

$db = new *DbConnect*();

$conn = $db->connect();

$sql = "SELECT email, password FROM accounts WHERE username = :username";

if ($stmt = $conn->prepare($sql)) {

    $stmt->bindParam(':username', $\_POST['username']);

    $stmt->execute();

    $user = $stmt->fetch(*PDO*::FETCH\_ASSOC);

    if ($user) {

        if (password\_verify($\_POST['password'], $user['password'])) {

            session\_regenerate\_id();

            $\_SESSION['loggedin'] = true;

            $\_SESSION['username'] = $\_POST['username'];

            $\_SESSION['email'] = $user['email'];

            header('Location: home.php');

            exit;

        } else {

            echo 'Nesprávné uživatelské jméno nebo heslo!';

        }

    } else {

        echo 'Nesprávné uživatelské jméno nebo heslo!';

    }

} else {

    exit("Could not prepare statement");

}

Nejdřív si zjistíme, zda existuje záznam o uživateli. Pokud ano, můžeme verifikovat heslo pomocí funkce **password\_verify()**, která zadané heslo porovná se zahashovaným heslem v databázi. Je-li heslo verifikováno, uložíme si údaje o uživateli do session a pomocí **session\_regenerate\_id()** přegenerujeme id dané session, aby se zabránilo útokům, jako je session fixation.

Pokud vše proběhne hladce, header nás přesměruje do skriptu **home.php**, který si záhy napíšeme. Ještě však uděláme malou změnu ve skriptu *index.php*. Na úplný vrch sriptu si přidáme podmínku. Pokud je nějaký uživatel přihlášen, dojde k automatickému přesměrování na *home.php*, protože je hloupost, aby přihlášený uživatel opět viděl přihlašovací formulář. Bylo by to pro něj matoucí.

<?php

session\_start();

// Pokud je uživatel přihlášen, přesměrujte ho na hlavní stránku

if (isset($\_SESSION['loggedin']) && $\_SESSION['loggedin'] === true) {

    header('Location: home.php');

    exit;

}

?>

### home.php

Předposledním krokem je profilová stránka přihlášeného uživatele, kde jsou vypsány jeho podrobnosti.

<?php

session\_start();

if (!isset($\_SESSION['loggedin'])) {

    header('Location: index.php');

    exit;

}

?>

<!DOCTYPE html>

<html lang="en">

<head>

    <meta charset="UTF-8">

    <meta name="viewport" content="width=device-width, initial-scale=1.0">

    <title>Home Page</title>

</head>

<body>

      <h1>Home Page</h1>

<p>Welcome back, <strong><?php echo htmlspecialchars($\_SESSION['username'], ENT\_QUOTES) ?></strong></p>

      <h2>Profile card</h2>

<p>username: <strong><?php echo htmlspecialchars($\_SESSION['username'], ENT\_QUOTES) ?></strong></p>

<p>email: <strong><?php echo htmlspecialchars($\_SESSION['email'], ENT\_QUOTES) ?></strong></p>

      <a href="logout.php">Logout</a>

</body>

</html>

Pokud se ze session zjistí, že není přihlášený žádný uživatel, přesměruje jej header na přihlašovací stránku. Pro odhlášení odkaz spustí skript **logout.php**.

### logout.php

Tento malý skript zavře session a tím pádem odstraní údaje o přihlášení uživatele na straně serveru. Pak se může přihlásit uživatel nový.

<?php

session\_start();

session\_destroy();

header('Location: index.php');

exit;

## Závěrem

Máte nejnutnější minimum pro registraci a přihlašování uživatele. Již můžete dál rozvíjet. Pamatujte, že pro složitější aplikace se hodí si kód uspořádat do tříd a využít naplno OOP. Dále si můžete vyhrát s typy účtů – admin, správce, řadový uživatel. Tam bude prostor pro různě dostupné funkcionality a viditelnost obsahu.

code: [github.com/veetecheck/PHP\_basic\_registration\_login](https://github.com/veetecheck/PHP_basic_registration_login)

# MVC pattern a projekt Message board

## Princip MVC

Jak vlastně PHP funguje v rámci web developmentu? Čím je aplikace složitější a robustnější, je třeba code zpřehlednit a rozdělit jej na 3 různé části.

1. připojení k databázi a zasílání dotazů/příkazů do databáze – Model
2. zobrazení dat uživateli webu – View
3. zpracování dat od uživatele webu – Controller

Oproti procedurálnímu přístupu PHP, kdy se tyto 3 využití mohou psát takřka v jednom souboru, MVC přístup je odděluje a pro každý takový úkon existuje vlastní soubor. Soubory jsou mezi sebou prolinkovány. Mohou být tedy použity vícekrát v rámci projektu a vyvarujeme se DRY kódu.

Tento design pattern se nazývá MVC, což je zkratkou pro Model – View – Controller, a je základem mnoha moderních PHP frameworků. Pokud se s nějakým PHP frameworkem v budoucnu setkáte, určitě oceníte pohled pod pokličku fungování MVC. Nebudete tak vyplesklí a budete mít šajnu, jak jejich princip funguje.

## Projekt Message board – základ

V tomto projektu si vytvoříme pomocí PHP MVC jednoduchou aplikaci na zobrazení a správu textových příspěvků různých uživatelů na stránce, jež funguje jako nástěnka. Databázová část se bude odehrávat na MySQL. Základ této aplikace jsem vzal z kurzu lektora *traversymedia* alias Brad Traversy z Udemy, za což velice děkuji – MVC projekt od píky je docela horor, stejně jako debugging PHP aplikace. Níže pro zvídavé odkaz na kurz a zároveň uvádím jako hlavní zdroj.

[www.udemy.com/course/object-oriented-php-mvc](http://www.udemy.com/course/object-oriented-php-mvc)

Jelikož jste už v minulých kapitolách s mnoha věcmi pracovali, nebudu se vracet k teorii, berte tento text jako kuchařku, kde zmíním zajímavosti k věcem, které jste si ještě neosahali.

Zde také dávám k dispozici hotový code, který jsem s pomocí tutoriálu, chatGPT a vlastního umu, stvořil:

<https://github.com/veetecheck/PHP_MVC_messageboard>

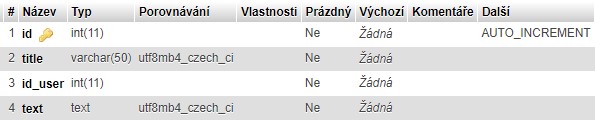
Doporučuji, abyste pracovali s hotovým kódem, abyste si udělali obrázek o souvislostech. A silně doporučuji video z výuky. Tento text není plnohodnotným návodem. Jakmile kód bude delší, než je pro skripta únosné, budu se odkazovat na GitHub repozitář.

### Struktura databáze

Vytvořte si databázi **messageboard** se dvěma tabulkami **users** a **posts**, jejichž strukturu máte níže.

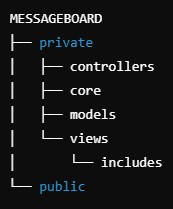


*obr. tabulka users*



*obr. tabulka posts*

### Struktura projektu

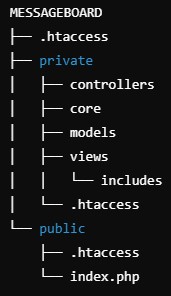
V root složce lokálního serveru Apache *(C:\xampp\htdocs)* si vytvořte pracovní složku. Já tu svoji pojmenoval **messageboard**.

Pomocí adresářového schématu si vytvořte strukturu složek. Nejdříve jako podsložky adresáře **messageboard** vytvořte **private** a **public**.

V adresáři **private** vytvořte podsložky **controllers**, **core**, **models** a **views**.

Na závěr jako podsložku adr. **views** s názvem **includes**.

### Soubory .htaccess a index.php

Pro začátek si vytvoříme 4 soubory:

1. v root složce *messageboard* soubor **.htaccess**

<IfModule mod\_rewrite.c>

RewriteEngine On

RewriteRule ^$ public/ [L]

RewriteRule (.\*) public/$1 [L]

</IfModule>

*RewriteEngine On*: Zapíná přepisování URL pomocí modulu mod\_rewrite.

*RewriteRule ^$ public/ [L]*: Pokud je URL prázdná (žádná cesta není specifikována), přesměruje na složku public/.

např.: <http://localhost/messageboard/> přesměruje na

<http://localhost/messageboard/public/>

*RewriteRule (.\*) public/$1 [L]*: Přesměrovává všechny ostatní požadavky na odpovídající cestu v rámci složky public.

např.: <http://localhost/messageboard/neco/neco> přesměruje na <http://localhost/messageboard/public/neco/neco>

1. ve složce *private* soubor **.htaccess**

<IfModule mod\_rewrite.c>

Options -Multiviews

RewriteEngine On

RewriteCond %{REQUEST\_FILENAME} !-f

RewriteCond %{REQUEST\_FILENAME} !-d

RewriteRule ^(.\*)$ index.php?url=$1 [L,QSA]

</IfModule>

Tento .htaccess soubor směruje všechny příchozí požadavky na index.php soubor, pokud požadavek není na konkrétní existující soubor nebo složku.

*Options -Multiviews*: Vypíná Multiviews, což zabraňuje automatickému mapování URL na soubory se stejným názvem (bez ohledu na příponu).

*RewriteEngine On*: Zapíná přepisování URL.

*RewriteCond %{REQUEST\_FILENAME} !-f*: Podmínka, která říká, že pravidlo se použije pouze v případě, že požadavek není na existující soubor.

*RewriteCond %{REQUEST\_FILENAME} !-d*: Podmínka, která říká, že pravidlo se použije pouze v případě, že požadavek není na existující adresář.

*RewriteRule ^(.\*)$ index.php?url=$1 [L,QSA]*: Přesměruje všechny ostatní požadavky na index.php, přičemž předává požadovanou cestu jako parametr url.

*[L]*: Určuje, že se jedná o poslední pravidlo (neprovádí se další).

*[QSA]*: Zajišťuje, že query string (např. ?key=value) je připojen k URL.

1. ve složce *public* soubor **.htaccess**

Options -Indexes

Zabraňuje indexování souborů a adresářů ve složce private.

*Options -Indexes*: Zakáže automatické zobrazování obsahu adresáře, pokud neexistuje index.php nebo jiný výchozí soubor. Pokud by někdo zadal přímou cestu k adresáři private, obdržel by chybu 403 Forbidden, místo zobrazení seznamu souborů. My máme ale z ostatních souborů pravidla nastavená tak, že skočí opět na *index.php*.

1. ve složce *public* soubor **index.php**

Tato konfigurace .htaccess souborů pomáhá chránit složku private, směrovat veškerý provoz přes *public/index.php*, a správně směrovat URL na správné soubory v rámci aplikace.

<?php

echo 'hello';

Zkuste si teď všechny zmíněné případy URL a uvidíte, že vás to vždy spláchne k *index.php.* Neboli v tuto chvíli na vás bude na stránce koukat nápis *hello.*

Upravte si code na následující znění:

<?php

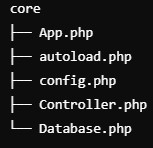
require '../private/core/autoload.php';

$app = new App();

Toto bude celý vnitřek souboru *index.php.* Jednak budeme vyžadovat soubor *autoload.php*, který bude sloužit jako styčný bod mezi *public* a *private* částmi aplikace.

A dále zde budeme tvořit novou instanci třídy *App*. To bude na straně *public* vše. Výše zmíněné soubory a třídy nemáme ještě vytvořeny, dáme se tedy do toho.

### Složka core

1. soubor **config.php**

<?php

define('PRIVATEROOT', dirname(dirname(\_\_FILE\_\_)));

define('URLROOT', 'http://localhost/messageboard');

V tomto souboru si definuji dvě konstanty – jednak cestu ke složce *private* a za druhé kořen URL (ten se samozřejmě mění v závislosti, zda je projekt na lokálním počítači nebo na hostingu).

*dirname(dirname(\_\_FILE\_\_))*: získáme cestu nadřízené složky aktuální složky, kde se nachází aktuální soubor. To bude platit pro všechny soubory v *core*, protože jsou všechny na stejné úrovni jako *config.php*. Máme tedy cestu ke složce *private.*

1. soubor **autoload.php**

<?php

session\_start();

require 'config.php';

require 'Controller.php';

require 'App.php';

require 'Database.php';

V tomto souboru budeme vyžadovat všechny soubory z *core* složky, jelikož funguje jako propojovací uzel do *public/index.php.* Nachází se zde i start session, bude se hodit později při přihlašování a odhlašování uživatele, s čímž se budou vázat další funkcionality aplikace.

1. soubor **App.php**

Jak jste si všimli, název souboru je velkým písmem, takže bude obsahovat definici třídy App, kde bude definovaná logika pro fungování URL, jakožto spouštěče controllerů a jejich metod.

<?php

class App

{

    private $controller = "Home";

    private $method = "index";

    private $params = [];

    public function \_\_construct()

    {

        $url = $this->getUrl();

        // controller

        if (file\_exists(PRIVATEROOT . "/controllers/" . ucfirst($url[0]) . ".php")){

            $this->controller = ucfirst($url[0]);

            unset($url[0]);

        }

        require PRIVATEROOT . "/controllers/" . $this->controller . ".php";

        $this->controller = new $this->controller();

        // metoda

        if (isset($url[1]) && method\_exists($this->controller, $url[1])){

            $this->method = $url[1];

            unset($url[1]);

        }

        // parametry

        $url = array\_values($url);

        $this->params = $url;

        call\_user\_func\_array([$this->controller, $this->method], $this->params);

    }

    private function getUrl()

    {

        if (isset($\_GET['url']) && $\_GET['url'] !== '') {

            return explode("/", filter\_var(trim($\_GET['url'], "/"), FILTER\_SANITIZE\_URL));

        }

        return ['home'];

    }

}

Jak vlastně bude aplikace pracovat s URL jako s nástrojem, který určuje, co se bude dít? Je to vcelku jednoduché – rozdělí si URL na části, ze kterých vyčte, co použít za controller, jakou jeho metodu a případně jaké parametry té metodě předat.

např.: voláme controller *posts*, jeho metodu *delete*, jíž předáváme parametr 5 (což reprezentuje id postu, jak si později doprogramujeme): <http://localhost/messageboard/posts/delete/5>

Takže vždy, když se spustí skript *index.php*, vytvoří se instance třídy *App*, jež si vyzobe z URL info, kam se vydat.

Všimněte si, že máme defaultně nastavené vlastnosti *$controller*, *$method* a *$params.* Takže, když bude URL nějaká blbost, použije se controller *home* a spustí jeho metoda *index*, které se nepředává žádný parametr.

<http://localhost/messageboard/home/index/>

* funkce **getUrl()** nám rozdělí URL do pole, respektive v našem případě vše po *messageboard*, takže dostaneme z příkladů výše ['posts', 'delete', '5'] nebo ['home', 'index']. Pokud je URL prázdná, vrátí nám pole o jednom prvku ['home'].
* konstruktor **\_\_construct()**  nejdřív pomocí fce *getUrl()* získá potřebné informace a pak zjišťuje, zda daný kontroller fyzicky existuje. Pokud ano, nastaví jej do vlastnosti *$controller.*

Funkce **ucfirst** zajišťuje uppercase prvního písmene řetězce.

Podobně tomu je i u metody, kterou, pokud v daném controlleru existuje, nastaví jako aktuální hodnotu *$method.* Prvky postupně umazáváme z pole.

Pomocí fce **array\_values** zbyvší prvky, které jsou parametry metody, naindexujeme od 0. Pokud je pole prázdné – parametry neexistují, zůstane prázdné.

call\_user\_func\_array([$this->controller, $this->method], $this->params);

Tato kouzelná funkce zavolá z vybraného controlleru danou metodu a předá jí i parametry. Nebo prázdné pole, pokud nejsou.

1. soubor **Controller.php**

Bázová třída controlleru, ze které budou všechny specifické controllery dědit.

<?php

*class* Controller {

    public *function* view($view, $data = array()){

        extract($data);

        if(file\_exists(PRIVATEROOT . "/views/" . $view . ".view.php")){

            require(PRIVATEROOT . "/views/" . $view . ".view.php");

        } else {

            require(PRIVATEROOT . "/views/404.view.php");

        }

    }

    public *function* model($model){

        require(PRIVATEROOT . '/models/' . $model . '.php');

        return new $model();

    }

}

Bázovka v sobě nese dvě esenciální metody, a těmi jsou **view** a **model**. Pojďme se na ně podívat:

* bázová metoda **view($view, $data = array())** přijímá dva parametry – jednak název příslušného view, které chceme zobrazit, a zadruhé pole dat, které mu chceme předat. Defaultně je nastaven na prázdné pole. Nemusíme tedy předat data žádné. Následně zjistí, zda daný view existuje. Pokud ano, načte jej. Pokud ne, načte view chybové stránky.
* Druhá bázová metoda **model($model)** načte soubor se třídou modelu a vytvoří jeho instanci.

Proč obě metody nefungují podobně? View jen načítá HTML/PHP kód, který předvádí uživateli, nejedná se o žádnou třídu. Model naproti tomu inicializuje připojení do databáze a vykonává dotazy/příkazy v rámci třídy modelu.

1. soubor **Database.php**

<?php

class Database

{

private $host = 'localhost';

private $user = 'root';

private $pass = '';

private $dbname = 'messageboard';

private $dbh;

private $stmt;

private $error;

public function \_\_construct()

{

$dsn = 'mysql:host=' . $this->host . ';dbname=' . $this->dbname;

$options = array(

PDO::ATTR\_PERSISTENT => true,

PDO::ATTR\_ERRMODE => PDO::ERRMODE\_EXCEPTION

);

try {

$this->dbh = new PDO($dsn, $this->user, $this->pass, $options);

} catch (PDOException $e) {

$this->error = $e->getMessage();

echo $this->error;

}

}

// preparing statement

public function query($sql)

{

$this->stmt = $this->dbh->prepare($sql);

}

// bind values

public function bind($param, $value, $type = null)

{

if (is\_null($type)) {

switch (true) {

case is\_int($value):

$type = PDO::PARAM\_INT;

break;

case is\_bool($value):

$type = PDO::PARAM\_BOOL;

break;

case is\_null($value):

$type = PDO::PARAM\_NULL;

break;

default:

$type = PDO::PARAM\_STR;

break;

}

}

$this->stmt->bindValue($param, $value, $type);

}

// execute statement

public function execute()

{

return $this->stmt->execute();

}

// get result set as array of objects

public function resultSet()

{

$this->execute();

return $this->stmt->fetchAll(PDO::FETCH\_OBJ);

}

// get single record as array of objects

public function single()

{

$this->execute();

return $this->stmt->fetch(PDO::FETCH\_OBJ);

}

// get row count

public function rowCount(){

return $this->stmt->rowCount();

}

}

Zde jsme do třídy **Database** schovali proces připojení k databázi a potřebné PDO operace s databází jsme si pro lepší přehlednost implementovali ve vlastních metodách.

Při vytvoření instance třídy *Database* se vytvoří připojení ze zadaných údajů, jak jsme už dělali v předchozích tutoriálech, tentokrát jsme použili v konstruktoru třídy *Database.* Co se týče dalších metod, jen rychlý přehled, neboť využívají známé funkcionality PDO:

* metoda **query($sql)** převezme parametrem SQL dotaz/příkaz a připraví jej
* metoda **bind($param, $value, $type = null)** naváže na parametr v dotazu hodnotu, typ hodnoty si zjistí sama
* metoda **execute()** vykoná dotaz/příkaz
* metoda **resultSet()** vrátí výsledek dotazu jako pole objektů
* metoda **single()** vrátí jeden řádek jako pole objektů
* metoda **rowCount()** vrátí počet řádků výsledku dotazu

Tím jsme vyčerpali základ aplikace, nyní můžeme vytvářet Controllery, Modely a Views.

## Message board – MVC

### Home controller

Ve složce *controllers* si vytvořte soubor **Home.php**.

<?php

class Home extends Controller

{

public function index() {

$data = [

'title' => 'Welcome to the Messageboard',

'description' => 'Lorem ipsum dolor sit amet consectetur, adipisicing elit. '

];

$this->view('home', $data);

}

}

Jak vidno, *Home* dědí ze třídy *Controller.* A proto může v metodě index vytvořit pomocí metody *view* třídy *Controller* příslušný view *'home'.* Ten si teď vytvoříme. Ve složce *views* si vytvořte soubor **home.view.php.**

<?php

echo $data['title'];

echo $data['description'];

Sice ne zrovna hezky, ale pokud si spustíte aplikaci <http://localhost/messageboard/>, pak vás to samozřejmě automaticky hodí na *home* controller s metodou *index* (říkali jsme si, že je to defaultní nastavení) a vykreslí se nám pomocí té metody i view, kam jsme předali parametry ve formě pole **$data**, respektive dvě hodnoty atributů *title* a *description*.

### Home view, header a footer

Soubor *home.view.php* jsme si už vytvořili, nyní jej změníme na skutečnou html stránku. Než se k tomu ale dostaneme, vytvořte si ve složce *views/includes* dva nové views – **header.view.php** a **footer.view.php**. Jelikož tyto části kódu stránky budou v každém view, nebudeme je psát pokaždé zas a znova.

* **header.view.php** – tento „vrch“ stránky bude mít každý view stejný. Zmiňuji, že jsem již nalinkoval stylesheet pro Bootstrap – pro tentokrát stačí web link, nic stahovat nebudeme.

<!DOCTYPE html>

<html lang="en">

<head>

<meta charset="UTF-8">

<meta name="viewport" content="width=device-width, initial-scale=1.0">

<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/css/bootstrap.min.css" rel="stylesheet" integrity="sha384-QWTKZyjpPEjISv5WaRU9OFeRpok6YctnYmDr5pNlyT2bRjXh0JMhjY6hW+ALEwIH" crossorigin="anonymous">

<title>Message board</title>

</head>

<body class="bg-secondary-subtle">

* **footer.view.php** – spodní pokračování stánky se skriptem pro BS a hlavně ukončení tagů z headeru.

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.3/dist/js/bootstrap.bundle.min.js" integrity="sha384-YvpcrYf0tY3lHB60NNkmXc5s9fDVZLESaAA55NDzOxhy9GkcIdslK1eN7N6jIeHz" crossorigin="anonymous"></script>

</body>

</html>

* **home.view.php** – jak header, tak footer teď budeme na každém view jen takto volat.

<?php $this->view('includes/header'); ?>

<div class="container">

<h1 class="display-5 text-center my-5"><?php echo $data['title'] ?></h1>

<p><?php echo $data['description'] ?></p>

</div>

<?php $this->view('includes/footer') ?>

### User model

Než začneme pracovat s uživateli webu, připravíme si příslušný model pro machinace s databází. Ve složce *models* si utvořte soubor **User.php**. Při vytváření instance modelu *User* se inicializuje připojení k databázi pomocí nové instance třídy *Database.* Vizte konstruktor.

<https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/models/User.php>

V případě metod a jejich dotazů/příkazů, předpokládám, není nutno co dodávat. Použity jsou metody třídy *Database*, s jejíž instancí model pracuje. Jen si povšimněte, že všechny metody modelové třídy *User* vracejí bool hodnoty, jen metoda *login* v případě úspěchu vrací celý řádek záznamu. Ty hodnoty se nám budou později v controlleru *Users* hodit při vytváření session, abychom věděli, zda je někdo přihlášen a kdo konkrétně.

### Users controller

Vytvořte si ve složce *controllers* soubor **Users.php**. Jednak si připravíme vlastnost *$userModel*, kterou inicializujeme v konstruktoru.

<https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/controllers/Users.php>

Chceme-li registrovat nového uživatele, jednak musíme zjistit, zda z formuláře na view *register.view.php*, který si za malou chvíli vytvoříme, byla metoda **register** zavolána spolu se serverovou metodou POST.

Kratší varianta – pokud metoda POST na serveru není přítomna, načte se jen *register* view s prázdnými iniciačními daty (naleznete na konci metody *register* za else).

Taktéž si všimněte, že proměnná *$data* obsahuje i chybové položky ke každé regulární položce, které jsou nutné pro informace do registrace (např. username <-> username\_err). Tím budeme posílat informaci o chybách do view *register* aby byl uživatel korigován pro správné vyplnění formuláře.

Pokud tedy uživatel odešle z view *register* data z formuláře, proběhne sáhodlouhá validace vstupních dat. Jednak na prázdné vstupy, na délku hesla, na nesedící potvrzení hesla, na již existujícího registrovaného uživatele podle username nebo emailu.

Pokud všemi validacemi skript projde a zjistí, že v poli jsou chybové hodnoty prázdné, uživatele zaregistruje a přesměruje na *login* view. Pokud jsou stále nějaké chyby, bude to uživatele vracet stále na formulář a informovat jej, co vyplnil špatně.

Metoda **login** funguje na stejném principu, jen při úspěšném zadání údajů zavolá metodu **createUserSession**, která vytvoří session s údaji o přihlášeném uživateli a přesměruje jej rovnou na view s příspěvky (vytvoříme si následně).

Metoda **logout** zničí session a přesměruje uživatele na formulář přihlášení – view *login.*

### Navbar, register a login view

Ve složce *views/includes* si vytvořte **navbar.view.php**.

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/views/includes/navbar.view.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/views/includes/navbar.view.php)

Všimněte si, že pokud je v session info o uživateli, což se rovná infu, že je někdo přihlášený, zobrazují se pod podmínkou odkazy login/register nebo posts/logout.

Navbar si stejným způsobem jako *header* a *footer* šoupněte do každého view. Hned pod *header*. Nezapomeňte i na *home.view.php.*

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/views/register.view.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/views/register.view.php)

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/views/login.view.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/views/login.view.php)

Formuláře pro registraci a login mají jednu zajímavost – pracují s třídami Bootstrapu. Ano, i v PHP to jde. Pokud je přítomna v poli *$data* některá z chybových hlášek, přidá se třída, která zvýrazní inkriminovaný input a zároveň se zobrazí hláška se zprávou té chyby.

### Post model

Metody modelu **Post.php**, který si vytvoříte ve složce models, jsou opět jasnými příkazy/dotazy do databáze. Ale rád bych vyjasnil, proč v metodě **getPosts** používáme JOIN. Je to proto, že u každého postu zobrazujeme i jméno uživatele, který jej vytvořil, což jsou info ze dvou tabulek.

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/models/Post.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/models/Post.php)

### Posts controller

Controller **Posts** používá stejné principy jako *Users*. Při vytváření instance se však kontroluje, zda v session je někdo přihlášený. Pokud ne, přesměruje to uživatele na přihlašovací formulář. Tím se zabrání, aby si posty přečetl někdo nepřihlášený.

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/controllers/Posts.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/controllers/Posts.php)

Metoda **index** je obdobná té z *Home* controlleru. V parametru *$data* posílá do view *posts* informace o všech příspěvcích, kde se zobrazí.

Metody **add** a **edit** pracují s příslušnými formuláři ve views *addPost* a *editPost*. Přidání příspěvku si vezme informaci o id přihlášeného uživatele ze session. Editace příspěvku se podaří pouze tomu přihlášenému uživateli, kteréhož id se shoduje s id uživatele, který příspěvek vytvořil.

Metoda **delete** stejně tak dovolí smazat přihlášenému uživateli jen ty příspěvky, které sám vytvořil.

### Views posts, addPost, editPost

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/views/posts.view.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/views/posts.view.php)

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/views/addPost.view.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/views/addPost.view.php)

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/views/editPost.view.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/views/editPost.view.php)

### Chybová stránka – view 404

Pokud půjde o neexistující view, vyskočí chybová stránka.

[github.com/veetecheck/PHP\_MVC\_messageboard/blob/main/private/views/404.view.php](https://github.com/veetecheck/PHP_MVC_messageboard/blob/main/private/views/404.view.php)