Tvorba webových aplikací

příklady

Michal Bubílek
Varnsdorf 10/2019

Anotace

Výukový materiál se zabývá tvorbou webových aplikací se zaměřením na serverovou část (Objektové programování v PHP na straně webového serveru). Student je seznamován s typickými modelovými situacemi, bezpečností, prací s databází a dalšími tématy. Materiál není koncipován jako kompletní referenční příručka jazyků, nýbrž čtenáře seznámí s problematikou a typickými modelovými příklady a nabídne a vysvětlí řešení.

Klíčová slova

OOP, web, PHP, praktické programování

Obsah

1 Úvod

V tomto materiálu budeme řešit několik typových problematik, se kterými se jistě při programování webových aplikací setkáte. Zabývat se budeme například automatickým načítáním souborů tříd, možnostmi konfigurace, základními návrhovými vzory, prací s databází, bezpečným přihlašováním, zpracováním datových souborů, a dalšími problematikami. U každé úlohy bude teoretické i praktické vysvětlení a mnohé bude názorně ukázáno na zdrojových kódech. Ukázky zdrojových kódů v dokumentaci nemohou být kvůli rozsáhlosti úloh zobrazeny kompletně, uvedeny budou tedy jen klíčové části. Na konci každé úlohy bude však k dispozici archiv s kompletním řešením úlohy. Tyto balíčky příkladů lze jen rozbalit do patřičné složky webového serveru a ihned použít. Jsou psány tak, aby posloužily ke studiu problematiky, a odpovídají uvedeným UML (Unified Modeling Language) diagramům u jednotlivých úloh.

Pro lepší orientaci v textu, zjednodušení, a tím i lepší pochopení problematiky budu dodržovat několik typografických prostředků:

- Třídy, metody, proměnné, funkce a další prvky kódu uvedené ve vysvětlujících odstavcích budou psány neproporcionálním písmem a modře.
- U metod budou vždy závorky.
- U proměnných v textu budu obvykle vynechávat dolar ($).
- Uvedené zdrojové kódy jsou psány neproporcionálním písmem, bez formátování a samozřejmě jsou bez výše uvedených omezení, vynechání dolaru u proměnných a názvu parametrů u metod...

V blocích zdrojových kódů budu obvykle mazat prázdné řádky, vynechávat dokumentační komentáře a značky počátku PHP kódu `<?php`, které se předpokládají na začátku každého php souboru. Jmenné prostory budu uvádět, jen když to pomůže pochopení problematiky. Zdrojové kódy příkladů uvedené v textu dokumentu slouží k pochopení nějaké problematiky, proto mohou být zkráceny či jinak upraveny, aby nebyl student rozptylován dalšími problematikami a příklad nezabíral několik stran dokumentace. V balíčkách s kompletní ukázkou je již vše výše uvedené řádně popsáno.

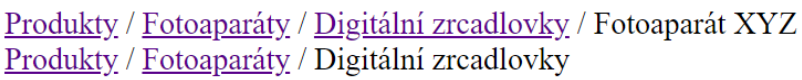
Dále upozorňuji, že uvedené příklady nejsou kompletní ve smyslu nasazení do ostrého provozu. Slouží pro demonstraci nějaké problematiky a nemusí tedy obsahovat další nutné ošetření stavů a správné generování výjimek.

2 Jednoduché úlohy

Při sestavování webové stránky si můžeme pomoci tím, že budeme mít jednotlivé prvky připravené jako objekty. V této kapitole si ukážeme velmi často používané prvky na webových stránkách, kterými je drobečková navigace a tabulka. Vytvořit si však můžeme i další prvky webu, jako jsou nadpis, obrázek, seznamy, formuláře a mnohé další.

2.1 Drobečková navigace

Drobečková navigace znázorňuje cestu strukturou webu k cíli. Díky ní tak jednak víme, kde se na webu nacházíme, a také se můžeme vrátit na libovolnou stránku zpět v cestě navigace. Cílem je tedy na webové stránce mít něco takového:



Obrázek 1 Snímek obrazovky s drobečkovou navigací

V programu bychom ji chtěli vytvářet, upravovat a vypisovat velmi jednoduše:

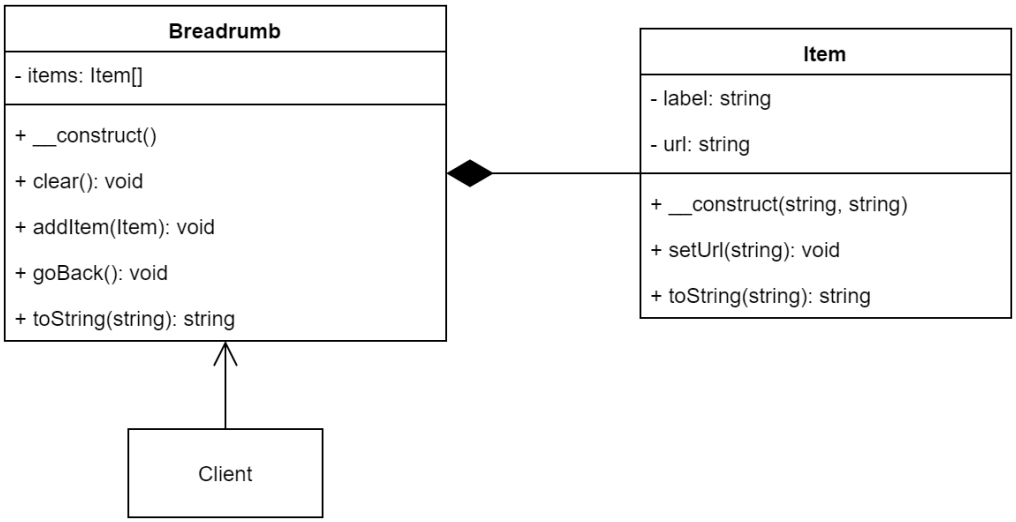
```
$bc = new Breadcrumb();
$bc->addItem(new Item('Produkty', 'index.php'));
$bc->addItem(new Item('Fotoaparáty', 'index.php'));
$bc->addItem(new Item('Digitální zrcadlovky', 'index.php'));
$bc->addItem(new Item('Fotoaparát XYZ'));
echo $bc;
```

Nejprve si tedy vytvoříme objekt z třídy `Breadcrumb` a poté jen voláme metodu `addItem()` s parametry zobrazovaného textu drobku a jeho cíle odkazu vygenerované jako instanci třídy `Item`. Postupně si tak sestavujeme onu cestu odkazů, které říkáme drobečková navigace. Nakonec referenci na objekt `bc` vypíšeme příkazem `echo`. Abychom zobrazili i onen druhý řádek z výše uvedeného obrázku, zavoláme užitečnou metodu `goBack()` a opět objekt vypíšeme.

```
$bc->goBack();
echo $bc;
```

Tohoto lze docílit jednoduše. Stačit nám k tomu budou dvě třídy. Jedna základní `Breadcrumb` a druhá `Item`. Instance z třídy `Item` je položkou drobečkového seznamu. Objekt drobečkové navigace je složen z jednotlivých položek. Využíváme zde skládání objektů. U každé jednotlivé položky vedeme její popisek a odkaz v podobě URL. Při vytváření objektu bude možné zadávat popisek i URL. Dále očekáváme setter URL, který řádně dovolený tvar adresy ošetří. Nakonec použijeme magickou metodu `__toString()`, která objekt převede korektně na řetězec, tedy HTML odkaz, je-li vyplněna URL, nebo jen text, není-li vyplněna URL. Třída `Breadcrumb` poté bude mít za úkol spravovat jednotlivé položky seznamu a nabízet vyčištění seznamu, přidání prvku do seznamu, vrátit se o krok zpět a samozřejmě také vykreslení celého seznamu v HTML jazyce. Jednotlivé třídy si pak mů-

žeme dále rozšiřovat o další funkce. Základní výše popsanou logiku znázorňuje následující digram tříd.



Obrázek 2 Diagram tříd drobečkové navigace

Protože se jedná o poměrně jednoduchý a krátký příklad, můžeme si zde uvést celý zdrojový kód tříd `Item` a `Breadrumb`:

```
class Item
{
   private $label;
   private $url;

    public function __construct(string $label, string $url = '')
    {
        $this->label = $label;
        $this->url = $this->setUrl($url);
    }

    public function setUrl(string $url)
    {
        if (empty($url)) {
            $this->url = '';
        } else {
            $this->url = preg_replace('~^-+|-+$~', '', strtolower(preg_replace('~[^a-zA-
Z0-9_/:#?=\.]+~', '-', $url)));
        }
    }

    public function __toString()
    {
        if (!empty($this->url)) {
            return '<a href="' . $this->url . '">' . $this->label . '</a>';
        } else {
            return $this->label;
        }
    }
}
```

```
class Breadcrumb
{
    private $items;

    public function __construct()
    {
        $this->clear();
    }

    public function clear()
    {
        $this->items = [];
    }

    public function addItem(Item $item)
    {
        array_push($this->items, $item);
    }

    public function goBack()
    {
        array_pop($this->items);
        $lastItem = end($this->items);
        $lastItem->setUrl('');
    }

    public function __toString()
    {
        return '<div class="breadcrumb">' . implode(' / ', $this->items) . '</div>';
    }
}
```

V kompletním příkladu na Githubu máte uveden i jmenný prostor a načítání tříd.

2.2 Tabulka

Vytváření tabulek je již komplikovanější než předchozí úloha, protože samotná tabulka je komplexnější prvek webové stránky. Chceme-li obdržet takovýto výsledek:

Tabulka: Produkty

Název	Barva	Cena	Email
A		123,50 CZK	a@b.cz
B		45 456 CZK	c@d.cz

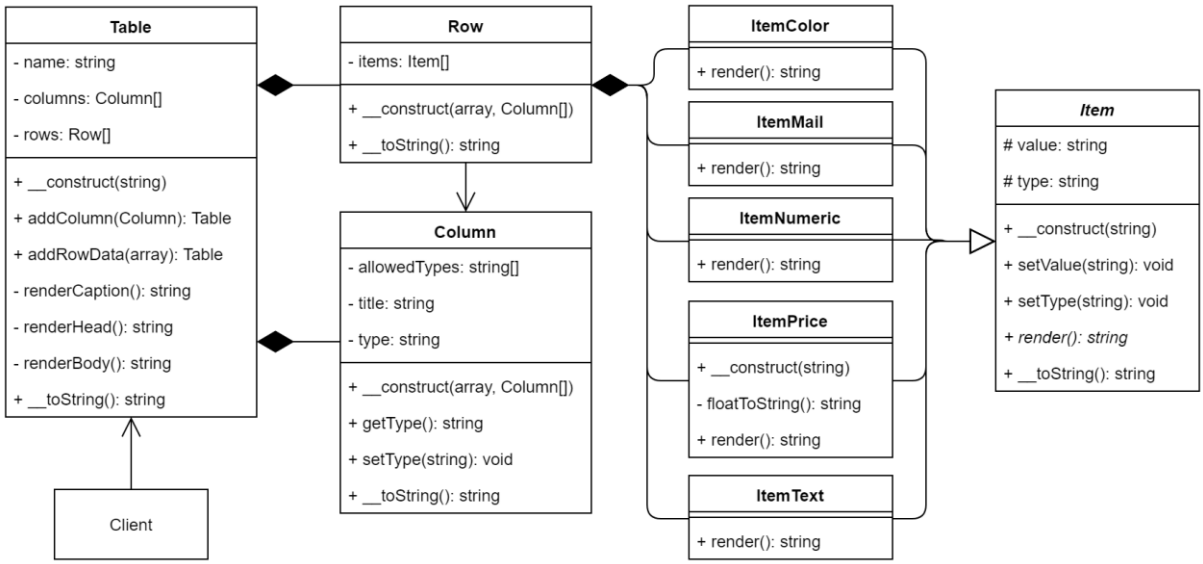
Obrázek 3 Snímek obrazovky HTML tabulky

V programu bychom chtěli takovouto tabulku vytvářet, upravovat a vypisovat jednoduše takto:

```
$table = new table\Table('Produkty');
```

```
$table->addColumn(new table\Column('Název'));
$table->addColumn(new table\Column('Barva', 'Color'));
$table->addColumn(new table\Column('Cena', 'Price'));
$table->addColumn(new table\Column('Email', 'Mail'));
$table->addRowData(array('A', '#0F0', 123.50, 'a@b.cz'));
$table->addRowData(array('B', '#00F', 45456, 'c@d.cz'));
echo $table;
```

Nejprve si vytvoříme objekt z třídy Table. Přidáme si všechny čtyři sloupce, u kterých si určíme jejich titulek a typ. Vložíme dva řádky s demonstrativními daty. Nakonec připravenou tabulku vypíšeme. Navenek potřebujeme jen třídu Table. Zbytek je od klienta odstíněn. Tabulka má sloupce, řádky a jednotlivé datové položky, které mohou být dle sloupce různého typu (text, barva, mail…). Struktura takovéhoto příkladu pak může vypadat například takto:



Obrázek 4 Diagram tříd HTML tabulky

Základní princip je podobný příkladu s drobečkovou navigací. Hlavní třída Table se stará o vytvoření tabulky, přidávání sloupců a dat řádků a samotné generování kompletní tabulky, které se skládá ze tří částí (titulek, hlavička, tělo). Metoda __toString() poté zajistí vygenerování kompletního HTML kódu tabulky do řetězce.

```
class Table
{
    private $name;
    private $columns;
    private $rows;

    public function __construct(string $name)
    {
        $this->name = $name;
        $this->columns = [];
        $this->rows = [];
    }
```

```
    public function addColumn(Column $column): Table
    {
        $this->columns[] = $column;
        return $this;
    }

    public function addRowData(array $rowData): Table
    {
        $this->rows[] = new Row($rowData, $this->columns);
        return $this;
    }

    private function renderCaption(): string
    {
        return '<caption>Tabulka: ' . $this->name . '</caption>';
    }

    private function renderHead(): string
    {
        return '<thead>' . implode('', $this->columns) . '<thead>';
    }

    private function renderBody(): string
    {
        return '<tbody>' . implode('', $this->rows) . '<tbody>';
    }

    public function __toString(): string
    {
        return '<table>' . $this->renderCaption() . $this->renderHead() . $this-
>renderBody() . '</table>';
    }
}
```

Třída `Column` se stará o sloupec tabulky. Každý sloupec má svůj titulek a typ. Kromě standardní práce s atributy je zde potřeba ověřit, zdali se do konstruktoru při vytváření sloupce zadává povolený existující typ. Zde je to vyřešeno velmi jednoduše statickým polem povolených typů, kterým se ověřuje zadaná hodnota.

```
class Column
{
    private static $allowedTypes = ['Text', 'Numeric', 'Price', 'Mail', 'Color'];
    private $title;
    private $type;

    public function __construct(string $title, string $type = '')
    {
        $this->title = $title;
        $this->setType($type);
    }

    public function getType(): string
    {
        return $this->type;
```

```
    }

    public function setType($type)
    {
        if (in_array($type, self::$allowedTypes)) {
            $this->type = $type;
        } else {
            $this->type = self::$allowedTypes[0];
        }
    }

    public function __toString()
    {
        return '<th>' . $this->title . '</th>';
    }
}
```

Třída `Row` obstarává celý řádek tabulky. Konstruktoru jsou předány všechny hodnoty řádku a definice sloupců, které mají podobu pole objektů z třídy `Column`. V konstruktoru se tak projede pole sloupců `columns` a pro každý sloupec se hledá jeho typ, dle kterého se vytvoří konkrétní datová položka řádku tabulky.

```
class Row
{
    private $items;

    public function __construct(array $row, array $columns)
    {
        foreach ($columns as $key => $column) {
            $data = isset($row[$key]) ? $row[$key] : NULL;
            $className = 'TWA\JednoduchePriklady\table\Item' . $column->getType();
            $this->items[] = new $className($data);
        }
    }

    public function __toString()
    {
        return '<tr>' . implode('', $this->items) . '</tr>';
    }
}
```

Ve výše uvedeném příkladu je zajímavé, jak lze generovat název třídy do proměnné `className` a objekt z této třídy dynamicky vytvořit a uložit na konec pole `items`. Protože tak pole `items` může obsahovat instance z různých tříd, potřebujeme zajistit správný převod prvků na HTML kód. Toto jsme v návrhu vyřešili třídou `Item` a jejími potomky, kteří vždy na řetězci „Item“ začínají. Třída `Item` je abstraktní, protože nepotřebujeme a ani nechceme, aby se z této třídy vytvářely instance. Slouží nám jako základní rodičovský předpis, ze kterého by měli již konkrétní potomci (`ItemMail`, `ItemText`, `ItemColor`…) vycházet. Také jsou zde definované všechny společné atributy a metody, které se již v potomcích implementovat nemusí. Každá instance z potomka třídy `Item` musí implementovat metodu `render()`. Pokud se tato instance převádí na řetězec, zavolá se metoda

`__toString()` u rodičovské třídy `Item`, protože v potomcích není implementována. V této metodě `__toString()` se zavolá metoda `render()` již z korektního potomka třídy `Item`, která vygeneruje HTML kód již specializované položky. Tomuto volání metody `render()` v metodě `__toString()`, kdy se dle kontextu zavolá ta správná metoda `render()` v potomcích třídy, se říká **polymorfismus**.

```
abstract class Item
{
    protected $value;
    protected $type;

    public function __construct($value)
    {
        $this->setValue($value);
    }

    public function setValue($value)
    {
        $this->value = $value;
    }

    public function setType($type)
    {
        $this->type = $type;
    }

    abstract public function render(): string;

    public function __toString()
    {
        return $this->render();
    }
}

class ItemText extends Item
{
    public function render(): string
    {
        return '<td>' . $this->value . '</td>';
    }
}
```

Ve výše uvedeném zdrojovém kódu je uvedena abstraktní třída `Item` a specializovaná třída `ItemText`, která ze třídy `Item` dědí. Další potomci třídy `Item` se liší jen v těle metody `render()`, tedy způsobem generování HTML kódu položky tabulky.

3 Návrhové vzory

V této kapitole poukážeme na různé zajímavosti, které nám mohou při objektovém programování zpříjemnit život. Návrhové vzory se obvykle dělí na vzory pro vytváření (Singleton, Builder, Prototype, Factory Method…), vzory struktury (Adapter, Decorator, Facade, Proxy…) a vzory chování (Interpreter, Observer, Strategy…). Všechny právě uvedené vzory patří do skupiny **GoF** (Gang Of Four – čtyři spisovatelé stojící za uznávanou knihou „Design Patterns: Elements of Reusable Object-Oriented Software“). S některými z nich se blíže seznámíme právě v této kapitole.

3.1 Lazy loading

Základní myšlenkou tohoto principu je načítat či zpracovávat data, až když je to opravdu potřeba. Ne vždy, když pracujeme s objektem, potřebujeme všechny jeho atributy. Některé atributy se získávají komplikovaněji než jiné. Někdy je potřeba data načíst z databáze, někdy se musí projít komplikovanými algoritmy, a to může něco stát. Tou cenou myslím čas na procesoru, čas při přenosu dat, čas zpracování na databázovém serveru, paměť atp. Mnohdy se tedy nevyplatí při vytváření objektu automaticky naplnit všechny jeho atributy. Základní primitivní příklad by mohl vypadat takto:

```
$product = new LazyLoading\ProductModel();
echo $product->getName();
```

Třída `ProductModel` volaná v předchozím úryvku:

```
class ProductModel
{
    private $name;

    public function __construct()
    {
        $this->name = NULL;
    }

    public function getName()
    {
        if ($this->name === NULL) {
            $this->load();
        }
        return $this->name;
    }

    public function load()
    {
        /* loading code - file parsing, database... */
        $this->name = 1;
    }
}
```

Z příkladu vidíme, že atribut `name` se naplní daty, až když je poprvé volán jeho vlastní getter `getName()`.

3.2 Plynulé rozhraní

Plynulé rozhraní (Fluent interface, Method chaining) je v tomto materiálu již několikrát použito. Šetří řádky kódů a přitom jej nijak nekomplikují. Představme si, že bychom chtěli vytvořit produkt a zadat jeho vlastnosti:

```
$product = new ProductModel();
$product->setName("A");
$product->setCode("1");
$product->setPrice(1.5);
```

To samé lze napsat snáze a bez stále opakovaného uvádění proměnné `product`. Jdeme tedy zároveň vstříc strategii DRY:

```
$product = new ProductModel();
$product->setName("A")->setCode("1")->setPrice(1.5);
```

Pokud nepotřebujeme dodržet konvenci názvu setterů, mohl by náš kód vypadat takto:

```
$product = new ProductModel();
$product->name("A")->code("1")->price(1.5);
```

Řešení tříd je jednoduché. Abychom mohli metody třídy `ProductModel` takto zřetězovat, stačí, aby metody vracely instanci objektu, ze kterého byly samy volány.

```
class ProductModel
{
    private $name;
    private $code;
    private $price;

    public static function create(): ProductModel
    {
        return new ProductModel();
    }

    public function name(string $value): ProductModel
    {
        $this->name = $value;
        return $this;
    }

    public function code(string $value): ProductModel
    {
        $this->code = $value;
        return $this;
    }

    public function price(float $value): ProductModel
    {
```

```
        $this->price = $value;
        return $this;
    }

    public function __toString()
    {
        return 'n:' . $this->name . ', c:' . $this->code . ', p:' . $this->price .
PHP_EOL;
    }
}
```

3.3 Návrhový vzor Singleton

Singleton znamená jedináček a míní to možnost vytvořit jen jedinou instanci ze třídy. Toho lze docílit v PHP velmi jednoduše. Privátním konstruktorem zakážeme vytváření instancí příkazem `new` z vnějšku třídy. Vracet instanci nám bude statická metoda `getInstance()`, která se postará i o vytvoření právě oné jedné instance, kterou si budeme udržovat ve statickém atributu vlastní třídy.

```
class Singleton
{
    private static $instance = NULL;

    private function __construct()
    { }

    public static function getInstance()
    {
        if (self::$instance === NULL) {
            self::$instance = new Singleton();
        }
        return self::$instance;
    }
}
```

Mimo třídu pak budeme pracovat následovně:

```
$singleton = Singleton::getInstance();
var_dump($singleton);
$nextSingleton = Singleton::getInstance();
var_dump($nextSingleton);
```

V obou proměnných bude reference na stejnou instanci:

```
object(TWA\NavrhoveVzory\Singleton\Singleton)[1]
object(TWA\NavrhoveVzory\Singleton\Singleton)[1]
```

Singleton se hodí v místech, kde nemáme potřebu vytvářet více instancí jedné třídy a kde tuto instanci potřebujeme jednoduše v různých částech kódu (různé metody různých tříd) volat a nechceme ji tedy předávat v parametrech konstruktoru nebo dalších metod. Toto však nemusí prospívat přehlednosti kódu a hůře se zjišťují závislosti. Singleton je právě z těchto důvodů někdy označován jako návrhový anti-vzor. Někdy nám

pro tyto potřeby, a v těchto materiálech se s tím mnohokrát setkáte, postačí statické třídy.

3.4 Návrhový vzor Factory Method

Návrhový vzor Factory má mnoho podob a provedení. Základním principem je vytváření instancí třídy v metodě buď té samé třídy, nebo i v jiné třídě. Pro jednoduchost a pochopení si uveďme jednoduchý příklad. Mějme třídu `Button` pro tlačítka:

```
class Button
{
    private $label;
    private $color;

    public function __construct(string $label)
    {
        $this->label = $label;
        $this->setColor('black');
    }

    public function setColor(string $color)
    {
        $this->color = $color;
    }

    public function __toString()
    {
        return $this->label . ' ' . $this->color . ' button.' . PHP_EOL;
    }
}
```

Naše tlačítko může mít popisek a barvu a umí se identifikovat v podobě textového řetězce. Popisek posíláme přímo do konstruktoru. Výchozí barva je černá. K dispozici máme i setter pro nastavení barvy. Toto nemá s opravdovým tlačítkem moc společného. Pro účely demonstrace nám to však stačí.

Vytvořme si nyní třídu `Factory`, která bude umět vytvářet metodou `createOK()` tlačítka OK, která generujeme v systému opakovaně. Konstrukci a nastavení necháme na této tovární metodě v této tovární třídě:

```
class Factory
{
    public function createOK(): Button
    {
        $btn = new Button('OK');
        $btn->setColor("green");
        return $btn;
    }
}
```

Tato třída může být dle potřeby statická:

```
class StaticFactory
```

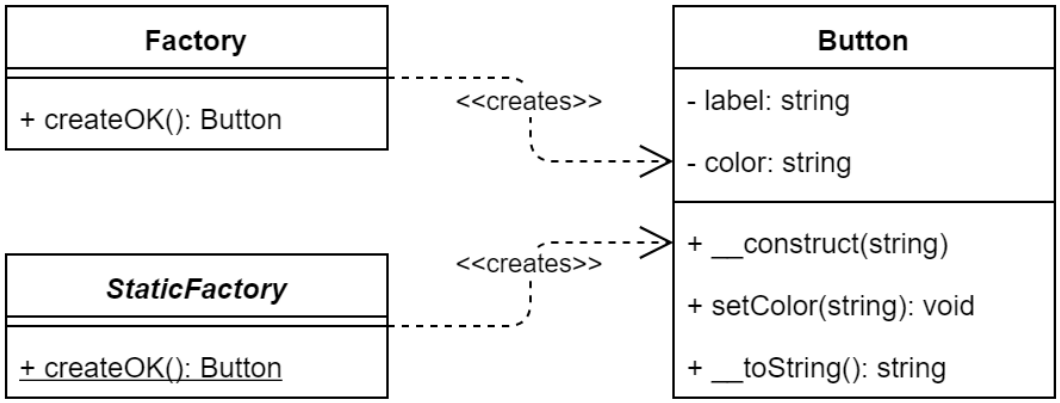
```
{

    public static function createOK(): Button
    {
        $btn = new Button('OK');
        $btn->setColor("green");
        return $btn;
    }
}
```

Nyní si ukážeme vytvoření tlačítka „OK“ klasicky pomocí příkazu `new`, továrnou `Factory` a statickou továrnou `StaticFactory`:

```
$btn = new Button("OK");
$btn->setColor("green");
echo $btn;

$factory = new Factory();
echo $factory->createOK();

echo StaticFactory::createOK();
```

Pro kontrolu je pak výstup:

```
OK green button.
OK green button.
OK green button.
```



Obrázek 5 Diagram tříd návrhového vzoru Factory Method

3.5 Návrhový vzor Adapter

Při snaze o znovupoužitelnost používáme své starší třídy, třídy svých kolegů, volně dostupná řešení jiných programátorů. Tyto třídy se mohou často měnit, jsou nestabilní. Jejich interface nemusí vyhovovat vašemu řešení. Pro tyto případy je obvyklé použití návrhového vzoru Adapter. Představme si, že máme od kolegy k dispozici třídu `SomeButton`:

```
class SomeButton
```

```
{

   private $title;
   private $decimalColor;

    public function __construct(string $title, string $decimalColor = '0,0,0')
    {
        $this->setTitle($title);
        $this->setColor($decimalColor);
    }

    public function setTitle(string $title)
    {
        $this->title = $title;
    }

    public function getTitle(): string
    {
        return $this->title;
    }

    public function setColor($decimalColor)
    {
        $this->decimalColor = $decimalColor;
    }

    public function getColor(): string
    {
        return $this->decimalColor;
    }
}
```

V našem systému však potřebujeme používat tlačítka implementující rozhraní `IButton`. Představme si například třídu `Buttons`, která má na starost tlačítka.

```
interface IButton
{
   public function setLabel(string $label);
   public function getLabel(): string;
   public function setColor(string $hexadecimalColor);
   public function getColor(): string;
   public function __toString(): string;
}

class Buttons
{
    private $buttons;

    public function __construct()
    {
        $this->buttons = [];
    }

    public function addButton(IButton $button)
```

```
    {
        $this->buttons[] = $button;
    }

    public function setLabel($label)
    {
        foreach ($this->buttons as $button) {
            $button->setLabel($label);
        }
    }

    public function __toString()
    {
        return implode('', $this->buttons);
    }
}
```

Ve třídě `Buttons` je metoda `addButton()`, která očekává instanci implementující rozhraní `IButton`. Toto je následně vidět v metodě `setLabel()`, kde se pracuje se setterem, který musí instance umět provést.

Rozdíly máme hned dva. V našem kódu pracujeme s atributem `label` a k tomu patřičným setterem a getterem. Kdežto ve třídě `SomeButton` se pracuje s atributem `title`. S barvou se pracuje sice stejnými metodami, ale v našem řešení se udává hodnota barvy v šestnáctkové soustavě začínající znakem `#`, kdežto ve třídě `SomeButton` se používá jako trojice desítkových čísel oddělených čárkou. Potřebujeme vytvořit adaptér `Button`, který přizpůsobí kolegovo řešení našemu:

```
class Button implements IButton
{
    private $button;

    public function __construct(External\SomeButton $button)
    {
        $this->button = $button;
    }

    public function setLabel(string $label)
    {
        $this->button->setTitle($label);
    }

    public function getLabel(): string
    {
        return $this->button->getTitle();
    }

    public function setColor(string $hexadecimalColor)
    {
        list($r, $g, $b) = sscanf($hexadecimalColor, "#%02x%02x%02x");
        $this->button->setColor("$r,$g,$b");
    }
```

```
    public function getColor(): string
    {
        $tmp = array_map(function ($value) {
            return str_pad(dechex($value), 2, '0', STR_PAD_LEFT);
        }, explode(',', $this->button->getColor()));
        return "#$tmp[0]$tmp[1]$tmp[2]";
    }

    public function __toString(): string
    {
        return $this->getLabel() . ' ' . $this->getColor() . ' button.' . PHP_EOL;
    }
}
```

Instance adaptované třídy je předána našemu adaptéru hned v konstruktoru. Adaptér si ji spravuje ve svém privátním atributu `button`. Zbylé gettery a settery implementující naše rozhraní `IButton` již provedou nezbytné operace k tomu, aby s adaptovanou třídou manipulovaly dle její implementace.

Práce s adaptérem může vypadat takto:

```
$saveBtn = new Adapter\External\SomeButton('Save');
$buttonAdapter = new Adapter\Button($saveBtn);
$buttonAdapter->setColor("#00FF00");
echo $buttonAdapter;
```

Výstup:

```
Save #00ff00 button.
```

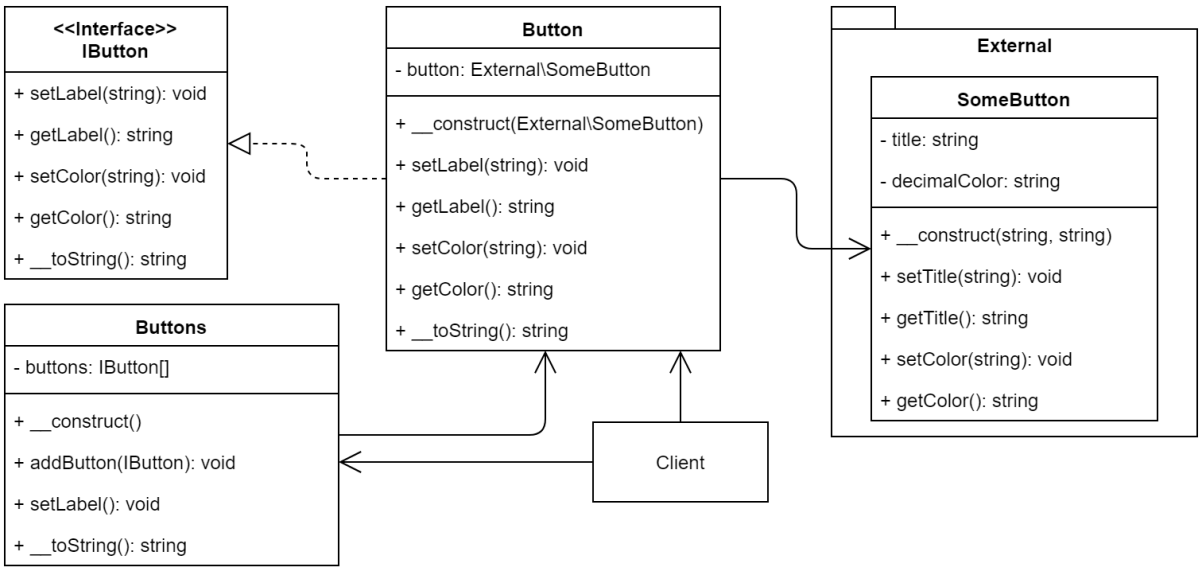
Práce s více tlačítky pomocí instance třídy `Buttons` pak může vypadat takto:

```
$saveBtn = new Adapter\External\SomeButton('Save');
$loadBtn = new Adapter\External\SomeButton('Load', '0,0,255');
$buttons = new Adapter\Buttons();
$buttons->addButton(new Adapter\Button($saveBtn));
$buttons->addButton(new Adapter\Button($loadBtn));
echo $buttons;
$buttons->setLabel('Reset');
echo $buttons;
```

Výstup:

```
Save #00ff00 button.
Load #0000ff button.
Reset #00ff00 button.
Reset #0000ff button.
```

Výhodu použití adaptéru můžeme nalézt zejména tedy v tom, že při úpravě adaptované třídy stačí upravit jen adaptér a zbytek našeho kódu může zůstat nezměněn. V následujícím obrázku je náš příklad vyjádřen diagramem tříd.



Obrázek 6 Diagram tříd návrhového vzoru Adapter

K Adapteru jsou podobné návrhové vzory Proxy (Zástupce) a Decorator (Dekorátor). Rozdíly jsou v tom, že Adapter implementuje jiné rozhraní, než jaké má původní objekt. Proxy implementuje stejné rozhraní. Dekorátor přidává k původnímu objektu nové funkce.

3.6 Návrhový vzor Facade

Představme si, že máme strukturu mnoha různých tříd, které slouží svým specifickým účelům. My však využijeme jen některé funkce nebo jejich kompozice. Navíc k nim chceme jednotný a přehledný přístup. Není pro nás důležité, abychom znali kompletně celou strukturu výše uvedených tříd. Pro tento účel se hodí návrhový vzor Facade (Fasáda), díky kterému si můžeme vytvořit „jednoduchou mezivrstvu“ mezi klientem a strukturou tříd, které klient nemusí znát. Pro demonstraci tohoto problému si představme dvě jednoduché třídy `Affix` a `Strings`:

```
class Affix
{
    public function addPrefix(string $string, string $prefix): string
    {
        return $prefix . $string;
    }

    public function addPostfix(string $string, string $postfix): string
    {
        return $string . $postfix;
    }
}

class Strings
{
    private $strings;

    public function __construct()
```

```
    {
        $this->strings = [];
    }

    public function add(string $value)
    {
        $this->strings[] = $value;
    }

    public function toString($separator): string
    {
        return implode($separator, $this->strings);
    }
}
```

Jak vidíme, třída `Affix` umí k řetězci přidávat prefix a postfix. Třída `Strings` vytváří pole řetězců prvek po prvku a umí toto pole řetězců metodou `toString()` převést na jeden textový řetězec oddělený volitelným oddělovačem `separator`. My však tyto třídy v našem systému využijeme jen pro dva účely. Jednak často používáme opatřování textu postfixem a také chceme všechny prvky zadaného pole opatřit zadaným prefixem a postfixem a vygenerovat řetězec takto upravených hodnot oddělených čárkou. K tomu si vytvoříme třídu `Facade`:

```
class Facade
{
    private $affix;
    private $strings;

    public function __construct()
    {
        $this->affix = new Affix();
        $this->strings = new Strings();
    }

    public function valuesToString(array $array, string $prefix, string $postfix): string
    {
        foreach ($array as $item) {
            $affixed = $this->affix->addPrefix($this->addPostfix($item, $postfix),
$prefix);
            $this->strings->add($affixed);
        }
        return $this->strings->toString(',');
    }

    public function addPostfix(string $string, string $prefix): string
    {
        return $this->affix->addPostfix($string, $prefix);
    }
}
```

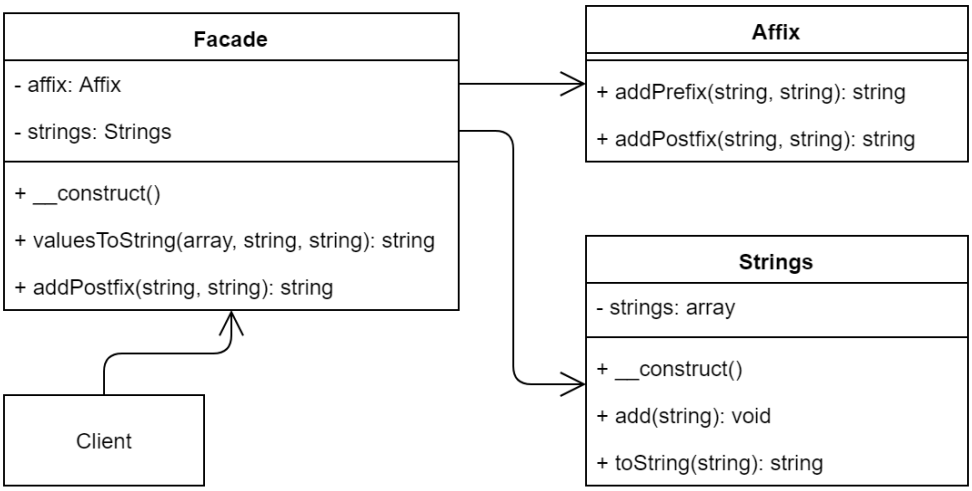
Nyní v našem kódu můžeme s fasádou pracovat samostatně a jednoduše:

```
$facade = new Facade\Facade();
```

```
$facade = new Facade\Facade();
echo $facade->valuesToString([24, 15, 7, 40], "'-", "'");
echo PHP_EOL;
echo $facade->addPostfix("19", ' °C');
```

Výstup:

```
'-24','-15','-7','-40'
19 °C
```



Obrázek 7 Diagram tříd návrhového vzoru Facade

Třídy za fasádou můžeme měnit se zachováním rozhraní fasády, a tedy i bez nutnosti úprav kódu, který fasádu využívá. Není také nutné znát implementace tříd za fasádou. K jednotlivým funkcím tříd za fasádou máme možnost přidat i řídící logiku.

3.7 Návrhový vzor Strategy

Mnohokrát máme více způsobů k dosažení jednoho výsledku. Algoritmy mají stejné rozhraní, ale jiné implementace. Chceme mít možnost si mezi těmito způsoby vybírat. Také chceme odstínit tyto implementace od klienta. To může být motivací pro použití návrhového vzoru Strategy (Strategie). Pro demonstraci si představme výpočet n. čísla Fibonacciho posloupnosti, které můžeme řešit sekvenčně nebo rekurzivně.

Fibonacciho řada je taková posloupnost čísel, kde každé číslo v řadě, kromě prvního a druhého, je hodnotou součtu svých dvou předchůdců. Příkladem takové řady je: 0 1 1 2 3 5 8 13 21…

Pro výpočet si připravíme třídy `RecursiveFibonacci` a `SequentialFibonacci` implementující rozhraní `IStrategy`:

```
interface IStrategy
{
    public function fib(int $n): int;
}

class RecursiveFibonacci implements IStrategy
```

```
{
    public function fib(int $n): int
    {
        if ($n < 2) {
            return $n;
        } else {
            return $this->fib($n - 1) + $this->fib($n - 2);
        }
    }
}

class SequentialFibonacci implements IStrategy
{
    public function fib(int $n): int
    {
        if ($n < 2) {
            return $n;
        } else {
            $beforePrevious = 0;
            $previous = 1;
            for ($i = 2; $i <= $n; $i++) {
                $fib = $previous + $beforePrevious;
                $beforePrevious = $previous;
                $previous = $fib;
            }
            return $fib;
        }
    }
}
```

Nyní si vytvoříme třídu `Context`, která bude uchovávat konkrétní instanci pro výpočet posloupnosti a bude na ni delegovat potřebné požadavky.

```
class Context
{
    private $strategy;

    public function __construct(IStrategy $strategy)
    {
        $this->strategy = $strategy;
    }

    public function fib(int $n): int
    {
        return $this->strategy->fib($n);
    }
}
```

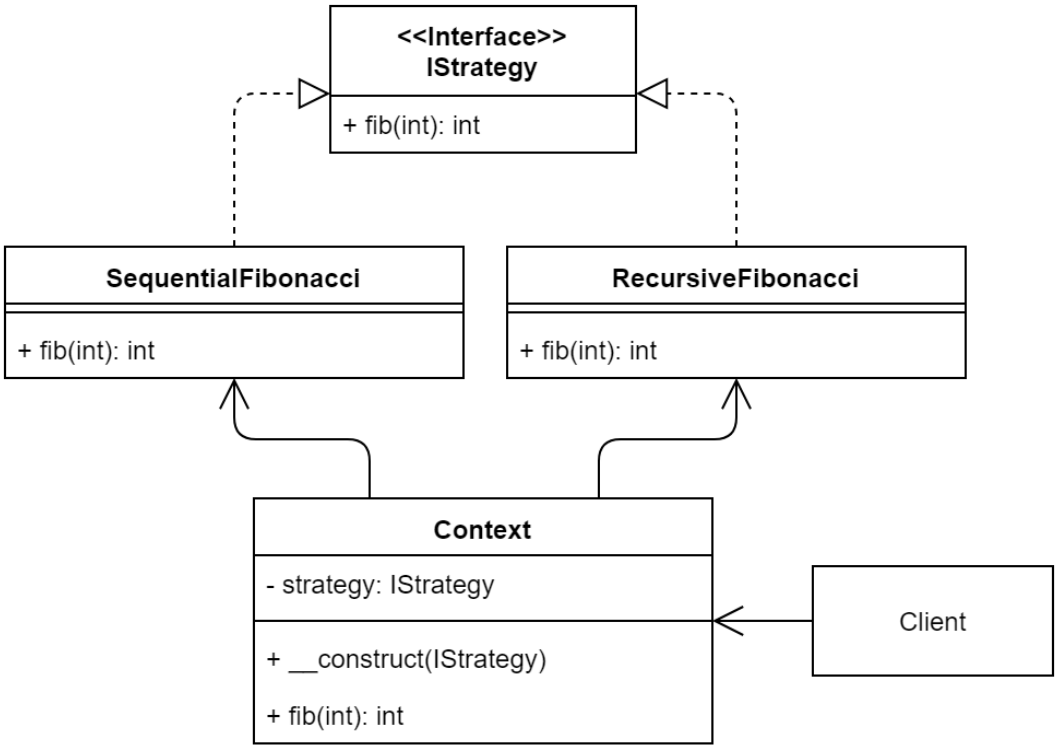
Příklad práce u klienta:

```
$recursiveContext = new Strategy\Context(new Strategy\RecursiveFibonacci());
echo $recursiveContext->fib(7);
echo PHP_EOL;
$sequentialContext = new Strategy\Context(new Strategy\SequentialFibonacci());
```

```
echo $sequentialContext->fib(7);
```

Výstup:

```
13
13
```



Obrázek 8 Diagram tříd návrhového vzoru Strategy

Existuje mnoho návrhových vzorů. Jejich znalost nám pomáhá lépe se rozhodovat při tvorbě architektury našeho řešení, která je obvykle syntézou více vzorů v jednom řešení.

4 Automatické nahrání tříd

Aby byla webová aplikace udržitelná, zdrojové kódy přehledné, efektivní a srozumitelné, neukládají se pouze do jednoho zdrojového souboru, ale obvykle do více souborů, u kterých platí nějaká autorem navržená struktura. Nevhodně strukturovaný kód je obvykle mezi programátory označován Špagetovým kódem, ve kterém se vše nepochopitelně míchá. OOP, MVC (Model View Controller) architektura, strukturované programování, rozdělení kódu do více částí a další strategie nám mohou k přehlednému a srozumitelnému kódu pomoci. Čím komplexnější webovou aplikaci vytváříme, tím více souborů se zdrojovými kódy máme a z nich sestavujeme výslednou aplikaci. U objektového programování bývá napříč programovacími jazyky zvykem, že každá vlastní třída nebo rozhraní mají svůj vlastní soubor. Podobně tomu může být i u jazyka PHP.

V PHP se soubory do kódu nejčastěji vkládají pomocí čtyř funkcí, respektive konstrukcí. Jde o `include`, `include_once`, `require` a `require_once`, za kterými se píše jako textový řetězec cesta souboru, který chceme vložit.

U všech čtyř konstrukcí můžeme použít i klasický zápis funkce s kulatými závorkami, kde se cesta předává jako argument parametru funkce. Podobně je tomu u konstrukce `echo`. Například:

```
include('inc.php');
echo('Ahoj');
```

Rozdíl mezi konstrukcemi `include` a `require` je v tom, že `include` při chybě tuto chybu pouze oznámí jako upozornění (`E_WARNING`) a skript dál pokračuje, kdežto `require` při chybě vypíše fatální chybu (`E_COMPILE_ERROR`) a skript se okamžitě ukončí. Postfix _once značí variantu, která zajistí, aby i při opětovném vkládání jednoho souboru, se tento soubor vložil jen jednou.

Nesmíme zapomenout, že vkládaný soubor se v místě volání i provede, jako by tam byl přímo celý napsán. Proměnné se tedy v tomto případě chovají jako globální. Pozor si musíme dát také na relativní cesty, kdy například ve vkládaném skriptu vkládáme další soubor nebo otevíráme ke čtení a zpracování další (jiný) soubor.

inc.php:

```
$x++;
```

index.php:

```
$x = 5;
include 'inc.php';
echo $x;
```

Výstup:

```
6
```

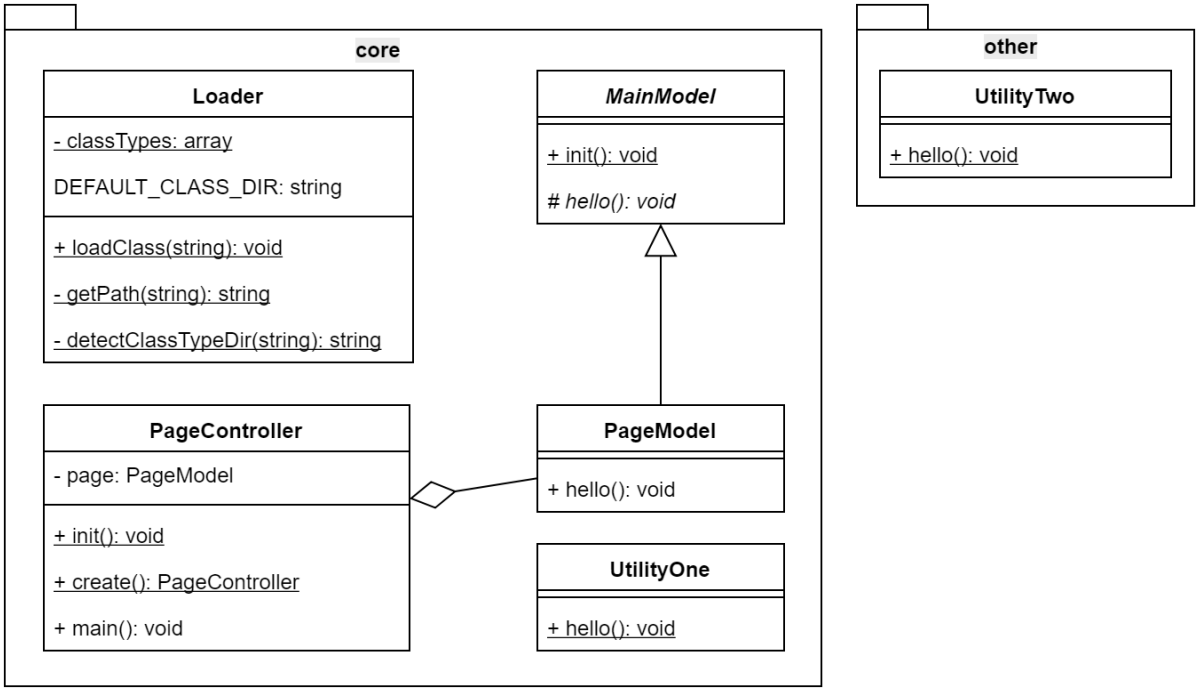
Protože chceme mít přehledný jasně strukturovaný kód a nechceme vkládat desítky souborů ručně, tak si ukážeme, jak lze vytvořit automatické načítání tříd ze souborů. K tomuto velmi dobře slouží funkce `spl_autoload_register()`, která je v PHP již od verze 5.1.0. Funkce vrací boolean, `true` při úspěchu a `false` při neúspěchu registrace nahrávací funkce. Není to ale jediný způsob, jak ošetřit případné neúspěšné provedení této funkce. Druhým parametrem funkce `spl_autoload_register()`, který je implicitně nastaven na `true`, lze zakázat – respektive explicitně povolit, vyhození výjimky, pokud se registrace nepovede. Toho využijeme i v našem případě. Prvním parametrem je název volatelné funkce určené k registraci.

4.1 Zavedení do problematiky úlohy

Nyní se vrhneme na naši praktickou úlohu. Architekturu našeho ukázkového příkladu zobrazuje digram tříd v následujícím obrázku.

Názvy mohou připomínat MVC architekturu. Kód je pro přehlednost a snadnou rozšiřitelnost rozdělen do tří logických částí (Model – View – Controller), které by mělo být možné upravovat tak, aby se při úpravě jedné části nemuselo zasahovat do jiné. Model reprezentuje daty a obvykle slouží jako konektor patřičných dat do databáze. View slouží pro zobrazení UI (User Interface). Controller řeší aplikační logiku. Controller a View má vazbu na Model. Model na Controller a View vazbu nemá.

Podobnost mého příkladu byla záměrná, ale neberte tuto strukturu jako typický příklad MVC architektury.

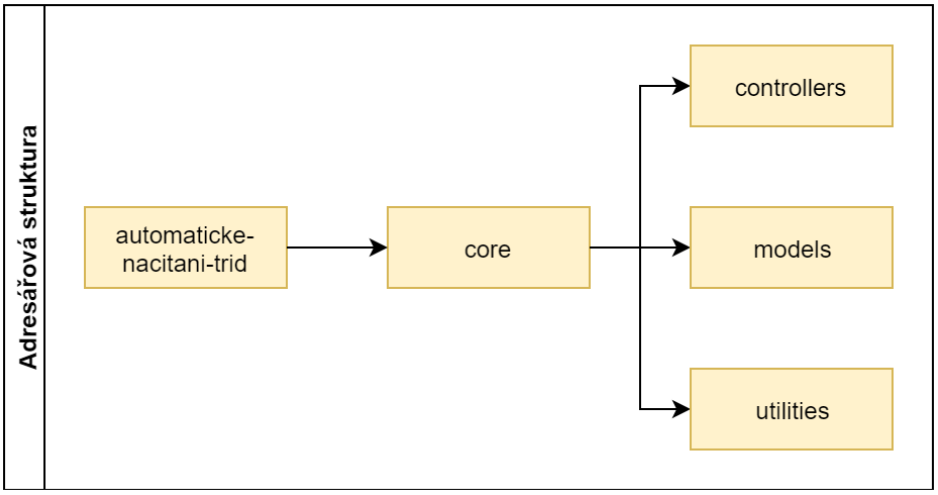


Obrázek 9 Diagram tříd pro nastínění architektury našeho ukázkového příkladu

Pro nás bude nejdůležitější pojednat o třídě `Loader`, která se stará o samotné automatické načítání souborů. Ostatní třídy tu slouží jen pro podporu a ukázku automatického

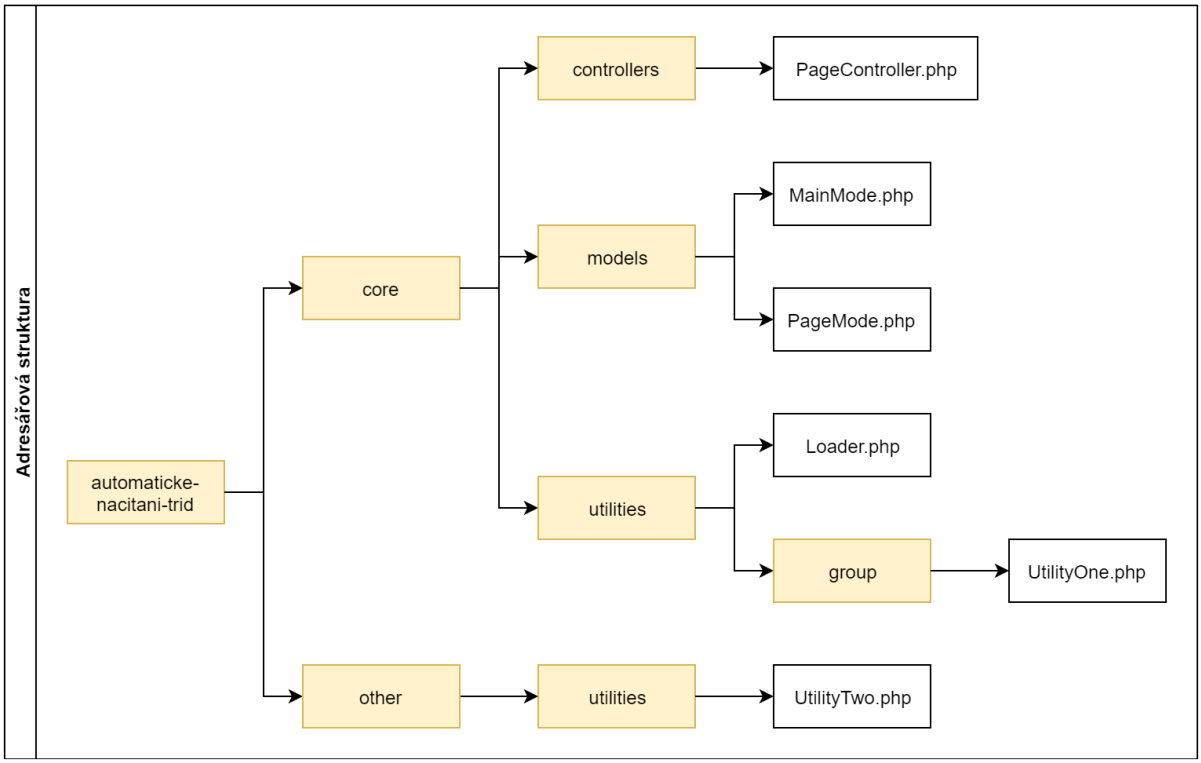
načítání a není třeba se tu o nich více rozvádět. Za zmínku však přesto stojí, že pro účely ukázky zde bylo zahrnuto dědění, abstraktní třída a například i jmenný prostor. Musíme si ujasnit pravidla našeho ukázkového příkladu. Každou třídu ukládáme do samostatného souboru, který se jmenuje stejně jako třída. Jen má příponu php. V našem příkladu máme tři typy tříd. Jsou to kontrolery, modely a ostatní. Kontroler respektive model poznáme tak, že má každá třída na konci svého názvu označení Controller respektive Model. Ostatní třídy se mohou jmenovat, jak chtějí. Kontrolery ukládáme do složky controllers, modely do složky models a ostatní třídy do složky utilities.

Pro jádro systému, které ukládáme do složky core, můžeme pak mít takovouto strukturu adresářů:



Obrázek 10 Základní adresářová struktura

V naší úloze používáme dva hlavní jmenné prostory, můžeme jich ale používat i více. Pokud si připravíme kromě jádra systému (jmenný prostor core) i něco dalšího (jmenný prostor other) a naplníme již složky ukázkovými třídami, pak získáme toto:



Obrázek 11 Kompletní adresářová struktura i se soubory

Našim úkolem bude automaticky načítat obsah souborů dle námi navržené architektury tak, jak jsou v průběhu vykonávání scriptu potřeba. Pokud tedy v průběhu vykonávání našeho skriptu nebudeme potřebovat nějakou třídu vůbec, nebude se ani načítat její kód.

Načítání datových prvků, až když jsou potřeba, je principem strategie Lazy loading. O tomto se budeme bavit v kapitole návrhových vzorů.

Dáme-li si dohromady navrženou strukturu adresářů a strukturu tříd, pak lze vypozorovat několik málo pravidel pro převod. Jmenný prostor (namespace) je vždy základním adresářem. Hlouběji pak následují adresáře dle typu tříd a v nich jsou již přímo soubory tříd nebo další podadresáře v případě komplikovanějších jmenných prostorů. Pro upřesnění si uveďme v tabulce jednoduché čtyři příklady převodu názvu tříd na požadovanou cestu k souboru.

Tabulka 1 Příklady převodu názvů tříd na cestu

Název třídy včetně jmenného prostoru	Kompletní cesta k souboru třídy
core\A\B	core/utilities/A/B.php
core\DemoController	core/controllers/DemoController.php
core\DemoModel	core/models/DemoModel.php
other\A\B\C	other/utilities/A/B/C.php

4.2 Postupné řešení úlohy

Nejprve pojednáme o obsahu souborů index.php, který slouží k „nastartování“ naší aplikace.

index.php:

```
mb_internal_encoding("UTF-8");
require_once('core/utilities/Loader.php');
try {
  spl_autoload_register('TWA\AutomatickeNacitaniTrid\core\Loader::loadClass', true);
} catch (Exception $exc) {
  echo 'Chyba při registraci automatické nahrávací funkce.';
  exit();
}
TWA\AutomatickeNacitaniTrid\core\PageController::create()->main();
```

Ve výše uvedeném scriptu ze souboru index.php si na začátku nastavíme pomocí funkce `mb_internal_encoding()` kódování UTF-8 (Unicode). Pokud pracujeme s řetězci, je dobré, aby PHP znalo naše kódování. Inicializovat zde můžeme mnoho další záležitostí, třeba Session, načtení konfiguračního souboru atp. Hned poté poprvé a naposledy ručně vložíme soubor Loader.php s třídou, která se stará o načítání souborů. V místě volání se tedy vloží celý obsah souboru Loader.php, tedy celá definice třídy `Loader`, jako bychom ji tam napsali. Nyní tedy můžeme třídu `Loader` používat, což hned v další části kódu uděláme. Pomocí metody `spl_autoload_register()` si zaregistrujeme statickou metodu `loadClass()` z třídy `Loader`. Druhým parametrem si explicitně nastavujeme vyhazování výjimek při neúspěchu registrace. Dělat to však nemusíme, protože `true` je implicitní hodnota. Zde tím chci na výjimky jen upozornit. V případě neúspěchu vypíšeme chybové hlášení a ukončíme skript. Nemá smysl pokračovat ve skriptu, který by hned v jeho další části skončil chybou. V této poslední části skriptu již jen vytvoříme instanci kontroleru a zavoláme metodu `main()`. Všimněme si, že zde již přímo voláme třídní (statickou) metodu `create()` třídy `PageController`, aniž bychom ručně vkládali soubor s definicí této třídy. Již zde se využije automatické načítání souborů.

Ve třídě `Loader` máme, jak vidíme ve výše uvedeném obrázku UML diagramu tříd, jedno statické pole a tři statické metody. Vše, krom metody `loadClass(string)`, která je veřejná, je privátní a slouží pro potřeby veřejné metody. Statické pole `classTypes` je mapovací pole typu volaných tříd na názvy jejich složek. Vypadá takto:

```
private static $classTypes = ['Controller' => 'controllers', 'Model' => 'models'];
```

Jako pole je to navrženo proto, že je velmi jednoduché přidávat další typy tříd (View, Component, Module…), aniž by se musel měnit zbytek kódu.

Třídní konstanta `DEFAULT_CLASS_DIR` slouží pro určení adresáře pro třídy, pro které nebyl nalezen adresář v mapovacím poli `classTypes`. Metody si začneme popisovat od spodu, abychom lépe pochopili jejich závislosti. První privátní statická metoda `detectClassTypeDir(string)` nám v jednoduchém cyklu prochází všechny položky mapovacího pole a pomocí regulárního výrazu hledá shodu s typem třídy uvedeným v argumentu `classFullName`. Jakmile například argument obsahuje na svém konci řetězec „Model“, v mapovacím poli se nejde v klíči dle regulárního výrazu shoda, funkce se ukončí a hodnota aktuálního prvku pole, což je adresář typu třídy (v našem případě „models“), je její návra-

tovou hodnotou. V případě, že by argument neobsahoval na svém konci řetězec „Controller“ ani „Model“, cyklus foreach se standardně ukončí. Následuje ukončení funkce s návratovou hodnotou dle výchozího nastavení třídní konstanty DEFAULT_CLASS_DIR, tedy hodnotou „utilities“.

```
private static function detectClassTypeDir(string $classFullName): string
{
    foreach (self::$classTypes as $classTypeName => $classTypeDirectory) {
        if (preg_match('~.*' . $classTypeName . '$~', $classFullName)) {
            return $classTypeDirectory;
        }
    }
    return self::DEFAULT_CLASS_DIR;
}
```

Metoda getPath(string) je klíčová pro funkci třídy Loader. Jejím argumentem je textový řetězec volané třídy a jejím výstupem je kompletní cesta souboru, ve kterém se volaná třída z argumentu nachází.

```
private static function getPath(string $classFullName): string
{
    $classFullName = substr($classFullName, strlen("TWA\\AutomatickeNacitaniTrid\\"));
    $classArray = explode('\\', $classFullName);
    if (!is_array($classArray) || count($classArray) < 2) {
        throw new \Exception("Nekompatibilní název třídy $classFullName.");
    }
    $path = array_shift($classArray) . '/' . self::detectClassTypeDir($classFullName) .
'/';
    for ($i = 0; $i < count($classArray); $i++) {
        $path .= $classArray[$i] . ($i < count($classArray) - 1 ? '/' : '.php');
    }
    if (!file_exists($path)) {
        throw new \Exception("Nemohu určit cestu $classFullName.");
    }
    return $path;
}
```

Na začátku si název třídy převedeme do pole pomocí funkce explode(). Pokud je pole v pořádku a obsahuje alespoň dva prvky (jmenný prostor a název třídy), tak pokračujeme, jinak vyhazujeme výjimku, která ukončí vykonávání zbytku kódu. Dalším krokem si již začínáme konstruovat samotnou cestu ke třídě do proměnné path. Nejprve si z výše uvedeného pole vezmeme první prvek pomocí funkce array_shift() a název adresáře dle typu třídy, což nám vrátí naše metoda detectClassTypeDir(string). Pak už jen cyklem for, pokud má pole více položek, generujeme podadresáře. U poslední položky pole vložíme nakonec příponu „.php“. Tím máme v proměnné path již kompletní cestu ke třídě. Nyní již jen vyzkoušíme, zdali soubor vůbec existuje pomocí funkce file_exists(). Pokud neexistuje, vyhodíme výjimku. V opačném případě ukončíme provádění funkce a návratová hodnota bude obsah proměnné path, tedy cestu ke třídě.

Poslední metodou je `loadClass(string)`, která je jako jediná veřejná, protože právě ona je registrována funkcí `spl_autoload_register()`.

```
public static function loadClass($classFullName)
{
    try {
        require_once self::getPath($classFullName);
    } catch (\Exception $exc) {
        echo $exc->getMessage();
        exit();
    }
    if (method_exists($classFullName, 'init')) {
        $classFullName::init();
    }
}
```

V metodě `loadClass(string)` si hned na začátku v bloku `try`, který zachycuje výjimky, „vyzkoušíme“ načíst konstrukcí `require_once` patřičný soubor, jehož cestu nám vrátí dříve vysvětlovaná privátní metoda `getPath(string)`. Pokud se to nepovede a zachytíme výjimku, vypíšeme hlášení a ukončíme skript. Pokud načtení souboru proběhne v pořádku, můžeme u vložené třídy hned například zavolat metodu `init()`, která může provést jakékoliv potřebné inicializační operace.

Ostatní pomocné třídy můžete zkoumat jak v UML diagramu, tak v kompletních zdrojových kódech uvedených na konci této kapitoly.

Nakonec si vysvětlíme základní kroky, které se dějí ve třídě `PageController`, která se spouští jako první, a ve které se používají zbylé třídy.

PageController.php:

```
namespace TWA\AutomatickeNacitaniTrid\core;

use TWA\AutomatickeNacitaniTrid\core\group;
use TWA\AutomatickeNacitaniTrid\other;

class PageController
{
    private $page;

    public static function init()
    {
        echo 'Inicializace ' . __METHOD__ . PHP_EOL;
    }

    public static function create(): PageController
    {
        return new PageController();
    }

    public function main()
    {
```

```
        $this->page = new PageModel();
        $this->page->hello();
        group\UtilityOne::hello();
        other\UtilityTwo::hello();
    }
}
```

Výstup:

```
Inicializace core\PageController::init
Inicializace core\MainModel::init
Inicializace core\MainModel::init
Pozdrav z core\PageModel::hello
Pozdrav z core\group\UtilityOne::hello
Pozdrav z other\UtilityTwo::hello
```

Na konci souboru index.php zavoláme statickou metodu `create()` třídy `PageController`. Při tomto pokusu se poprvé provede automatické načtení souboru třídy, a protože má třída definovanou veřejnou statickou metodu `init()`, tak se rovnou zavolá. To je na výstupu první řádek „`Inicializace core\PageController::init`“. Statická metoda `create()` vrací instanci této třídy `PageController`, takže na ni můžeme hned volat veřejnou metodu `main()`. V této metodě se hned na prvním řádku vytváří instance třídy `PageModel` do instanční proměnné `page`. Jen tento řádek může za dvojí vypsání „`Inicializace core\MainModel::init`“, protože třída `PageModel` dědí z abstraktní třídy `MainModel`, ve které je navíc definována metoda `init()`. První výpis „`Inicializace core\MainModel::init`“ se provede při nahrávání rodiče (`MainModel`) a druhý výpis pak při nahrávání potomka (`PageModel`). Na druhém řádku pak již jen zavoláme metodu `hello()` u instance třídy `PageModel`, která zajistí výpis „`Pozdrav z core\PageModel::hello`“. Poslední dva řádky již jen volají statické metody `hello()` u tříd zařazených do složek „utilities“ uložených v různých místech adresářové struktury našeho příkladu.

*Standardy PHP se zabývá **PSR** (PHP Standards Recommendations). Samotné PSR-4:Autoloader obsahuje doporučení pro způsoby pojmenování jmenných prostorů a vytváření adresářových struktur pro třídy za účelem jejich automatického nahrávání.*

5 Konfigurace

Tak jako každou aplikaci, tak i webové aplikace je potřeba konfigurovat. Je potřeba nastavit výchozí jazyk, email administrátora, přístupy do databáze, měnu a mnohé další položky. Vybrat si můžeme z mnoha souborových typů. V tomto příkladu si ukážeme čtyři běžně používané. V každém si připravíme pro ukázku přihlašovací údaje.

conf.csv:

```
USERNAME;csv
PASSWORD;1234
```

conf.ini:

```
[database login]
USERNAME = 'ini'
PASSWORD = '1234'
```

conf.php:

```
<?php
define('USERNAME','php');
define('PASSWORD','1234');
```

conf.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<conf>
    <item name="USERNAME" value="xml"></item>
    <item name="PASSWORD" value="1234"></item>
</conf>
```

Každý tento formát má své výhody a nevýhody. Rozdíly budou zejména ve složitosti načítání a zpracování údajů, kde vede jistě php, protože je konfigurace psána rovnou v jazyku PHP a nemusí se nic parsovat. V jednoduchosti správy položek je uživatelsky přívětivé ini. Dalším kritériem je bezpečnost. Kdybychom například odhadli název a umístění konfiguračního souboru s příponou jinou než php, tak nám jej webový server pošle do browseru. U těchto formátů nesmíme zapomenout nastavit zákaz zasílání těchto souborů. Pro webový server Apache bychom v souboru .htaccess nastavili něco takovéhoto:

```
<Files ~ "\.(csv|ini|xml)$">
   order deny,allow
   deny from all
</Files>
```

5.1 První „neobjektové“ řešení

Naše práce s konfigurací by měla být co nejjednodušší. Zde to pojmeme trochu jinak než obvykle. Protože v rámci systému budeme pracovat s jednou konfigurací, nebudeme potřebovat vytvářet více instancí. Stačila by nám jen jedna. Nám tuto instanci nahradí ale

samotná třída. Vystačíme si tak jen se statickým obsahem. Zároveň tím získáme možnost přístupu k public metodám této třídy odkudkoliv. Nemusíme řešit předávání objektů. Konfiguraci si „vyčarujeme“ kdekoliv a vždy, když ji potřebujeme.

Předávání objektů a přiznávání závislostí na objekty je velmi podstatné u OOP. V dobrém návrhu struktury tříd bychom na to neměli zapomenout. Ve výsledku to samotný kód zpřehledňuje.

Je-li potřeba, můžeme zakázat vytváření objektů ze třídy. Jednak to lze řešit abstraktní třídou:

```
abstract class Conf {}
```

Nebo privátním konstruktorem:

```
private function __construct() {}
```

Toto při vytváření nových instancí vede k chybám:

```
Fatal error:  Cannot instantiate abstract class…
Fatal error:  Call to private Conf::__construct() from invalid context…
```

V našem programu bychom chtěli v inicializační části konfiguraci načíst:

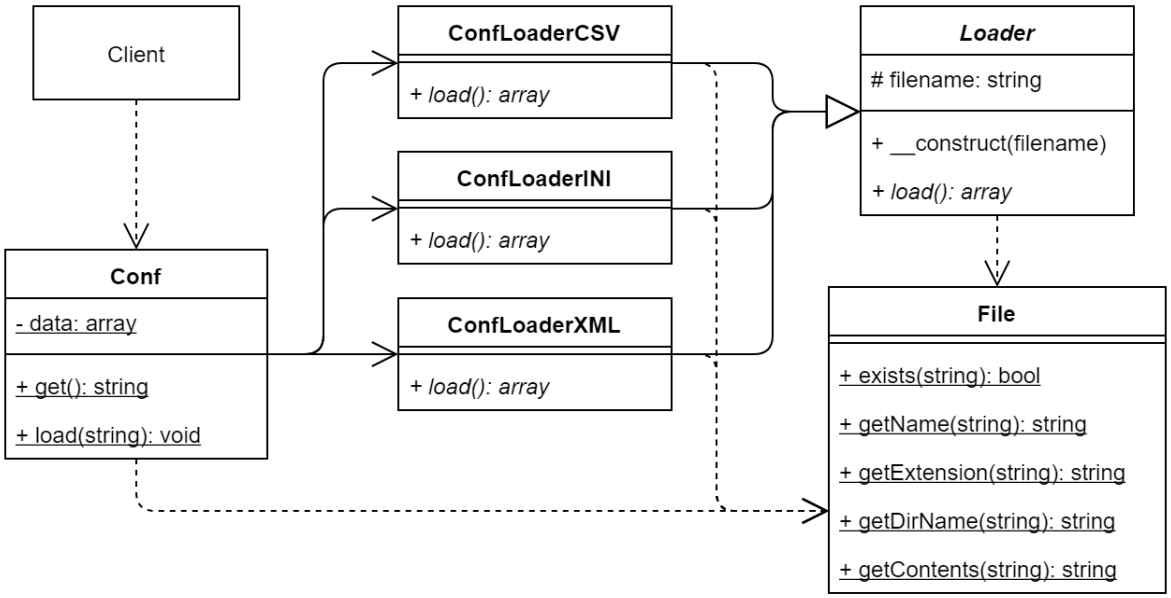
```
Conf::load("conf/conf.ini");
```

A poté kdekoliv pracovat ideálně takto:

```
echo Conf::get('USERNAME');
```

To berme jako zadání naší úlohy. V našem příkladu, jak výše vidíme, budeme potřebovat třídu `Conf`. V této třídě bude potřeba implementovat veřejné metody `load()` a `get()`. Pro načítání si vytvoříme novou abstraktní třídu `ConfLoader` sloužící jako rodič pro konkrétní potomky `ConfLoaderINI`, `ConfLoaderCSV` a `ConfLoaderXML`. `ConfLoaderPHP` není třeba, protože soubor PHP lze jednoduše vložit například funkcí `require_once()`.

Protože pracujeme se soubory, vytvoříme si pracovní třídu `File`, která bude umět načítat obsah souboru a pracovat s názvy a příponami. Celé schéma našeho příkladu je znázorněné na následujícím obrázku:



Obrázek 12 Diagram tříd pro příklad práce s konfigurací webové aplikace – první příklad

Conf.php:

```
class Conf
{
    private static $data = [];

    public static function get(string $name, string $default = ''): string
    {
        if (isset(self::$data[$name])) {
            return self::$data[$name];
        } else if (defined($name)) {
            return constant($name);
        } else {
            return $default;
        }
    }

    public static function load(string $filename)
    {
        $loader = null;
        try {
            switch (File::getExtension($filename)) {
                case 'ini':
                    $loader = new ConfLoaderINI($filename);
                    break;
                case 'csv':
                    $loader = new ConfLoaderCSV($filename);
                    break;
                case 'xml':
                    $loader = new ConfLoaderXML($filename);
                    break;
            }
            if (is_object($loader)) {
                self::$data = array_merge(self::$data, $loader->load($filename));
```

```
            }
        } catch (\Exception $e) {
            echo $e->getMessage() . '<br>';
        }
    }
}
```

Metoda `get()` je jednoduchá. Hledanou položku předávanou parametrem `name` postupně hledáme nejprve v poli `data` a poté, není-li konstantou. Když není ani jedno, vrátíme druhý parametr `default`, který je implicitně `null`. Druhý parametr nám tedy slouží k možnosti určení výchozí hodnoty položky, když není položka konfigurace nalezena:

```
echo Conf::get('SERVER', 'localhost');
```

Zajímavější je metoda `load()`, která dle přípony konfiguračního souboru rozhodne o tom, z jaké třídy bude instance sloužící k načtení položek konfiguračního souboru do statické proměnné data.

Abstraktní třída `ConfLoader` má implementovaný konstruktor, který však prakticky využijí jen její potomci. Postará se o určení názvu a existence konfiguračního souboru. Také má definovanou abstraktní metodu `load()`, potomci ji musí implementovat:

ConfLoader.php:

```
abstract class ConfLoader
{

    protected $filename;

    public function __construct($filename)
    {
        if (File::exists($filename)) {
            $this->filename = $filename;
        } else {
            throw new \Exception("File $filename not found.");
        }
    }

    abstract public function load();
}
```

ConfLoaderINI.php:

```
class ConfLoaderINI extends ConfLoader
{
    public function load()
    {
        $fileData = parse_ini_file($this->filename);
        if (is_array($fileData)) {
            return $fileData;
        } else {
            throw new \Exception("Loading $this->filename problem.");
        }
```

```
    }
}
```

Zbylé dva loadery a pomocnou třídu `File` naleznete v balíčku celé úlohy. Toto řešení není návrhově optimální a využít jej můžete v jednoduchých malých projektech. Objekty se zde nevytváří a třídy slouží vlastně jen pro účely zapouzdření funkcí.

5.2 Druhé „objektové“ řešení

Nyní si ukážeme řešení, které má čitelnější schéma, umožňuje vytvářet několik instancí různých konfigurací, zabývá se univerzálněji datovým úložištěm, nabízí programování vůči rozhraní a například i řeší **DI** (Dependency Injection). Již nemáme jen zapouzdřenou obálku funkcí pro práci se soubory obsahující základní funkce z PHP. Nyní už máme korektní objekt souboru, který předáváme konfiguraci na zpracování. Samotný objekt souboru nabídne svůj obsah a konfigurační parser jej zpracuje. Výsledné použití naší konfigurace by mohlo vypadat nějak takto:

```
/* INI config file */
$confINI = new Util\Conf(new Util\File("conf/conf.ini"));
echo $confINI->get('USERNAME', 'default') . PHP_EOL;
/* CSV config file */
$confCSV = new Util\Conf(new Util\File("conf/conf.csv"));
echo $confCSV->get('USERNAME', 'default') . PHP_EOL;
/* XML config file */
$confXML = new Util\Conf(new Util\File("conf/conf.xml"));
echo $confXML->get('USERNAME', 'default') . PHP_EOL;
```

Výsledek:

```
ini
csv
xml
```

Parametrem konstruktoru třídy `Conf` je v našem příkladu vždy instance třídy `File`. Nemusí to ale být pravidlem. Argument parametru konstruktoru může být jakákoliv instance třídy, která implementuje rozhraní `IData`. Takový objekt musí implementovat metodu, která vrátí data z datového úložiště metodou `getContents()` a také typ dat metodou `getType()`. To může být zde implementovanou třídou `File`.

IData.php:

```
interface IData
{
   public function getType(): string;
   public function getContents(): string;
}
```

File.php:

```
class File implements IData
{
    private $path;
```

```
    public function __construct($path)
    {
        $this->path = $path;
        if (!$this->exists()) {
            throw new \Exception("File $this->path not found.");
        }
    }

    public function exists(): bool
    {
        return file_exists($this->path);
    }

    public function getType(): string
    {
        return strval(strtolower(pathinfo($this->path, PATHINFO_EXTENSION)));
    }

    public function getContents(): string
    {
        if ($this->exists()) {
            $contents = file_get_contents($this->path);
        } else {
            throw new \Exception("File $this->path not found.");
        }
        if ($contents !== FALSE) {
            return $contents;
        } else {
            throw new \Exception("File::getContents() problem.");
        }
    }
}
```

ConfDataParser.php:

```
abstract class ConfDataParser
{
    protected $data;

    public function __construct()
    {
        $this->data = [];
    }

    abstract public function parse(IData $dataStorage);

    public function get(): array
    {
        return $this->data;
    }
}
```

ConfDataParserINI.php:

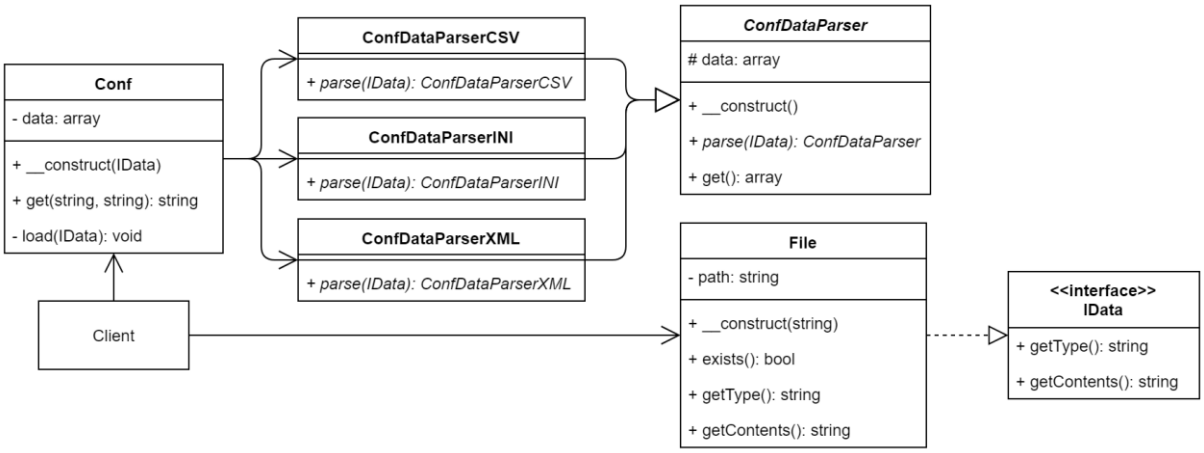
```
class ConfDataParserINI extends ConfDataParser
{
    public function parse(IData $dataStorage)
    {
        $this->data = parse_ini_string($dataStorage->getContents());
        if (!is_array($this->data)) {
            throw new \Exception("Parsing INI problem.");
        }
        return $this;
    }
}
```

Conf.php:

```
class Conf
{
    private $data;

    public function __construct(IData $dataStorage)
    {
        $this->data = array();
        $this->load($dataStorage);
    }

    public function get(string $name, $default = null): string
    {
        if (isset($this->data[$name])) {
            return $this->data[$name];
        } else {
            return $default;
        }
    }

    private function load(IData $dataStorage)
    {
        $parser = null;
        try {
            switch ($dataStorage->getType()) {
                case 'ini':
                    $parser = new ConfDataParserINI();
                    break;
                case 'csv':
                    $parser = new ConfDataParserCSV();
                    break;
                case 'xml':
                    $parser = new ConfDataParserXML();
                    break;
            }
            if (is_object($parser)) {
                $this->data = $parser->parse($dataStorage)->get();
            }
        } catch (\Exception $e) {
            echo $e->getMessage() . PHP_EOL;
        }
```

```
    }
}
```

Nyní, a to je vhodnější, samotné načítání dat ze souboru provádí třída `File`. Obsah souboru je pak dle typu předán korektnímu parseru, který data připraví a předá objektu třídy `Conf`. Zjednodušily se i některé metody.



Obrázek 13 Diagram tříd pro příklad práce s konfigurací webové aplikace – druhý příklad

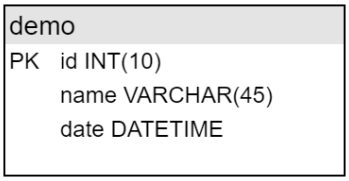
Nechám nyní na studentovi, aby si obě řešení vyzkoušel, porovnal a zhodnotil. Jistě jej napadne třetí, čtvrté a další řešení.

Ve druhém příkladu se vytratil způsob načítání konfigurace pomocí konstant PHP. Nechtěl jsem příklad zbytečně komplikovat dalšími třídami. Načtení souboru by zůstalo stejné pomocí příkazu `require`. Konstruktoru v `Conf` však musíme předat instanci implementující `IData`. Tak bychom jednoduše vytvořili novou třídu, například `EmptyDataStorage`, která by `IData` implementovala a metody `getContent()` a `getType()` by nic nevracely. Do konstruktoru v `Conf` bychom pak předávali instanci této třídy. Poté by vše proběhlo v pořádku a mohli bychom po lehké úpravě opět volat metodu `get()` z třídy `Conf` i na PHP konstanty. Otázka je však, zdali to vůbec takto chceme.

6 Práce s databází

Práce s databází je pro webové aplikace více než obvyklá. Programátoři se s ní setkávají tak často, že si vytváří vlastní, nebo využívají již hotové objektové nástavby na standardní nabízené databázové rozšíření (MySQLi, PDO…) do běžných systémů řízení báze dat (MySQL, MariaDB…). My si v tomto příkladu sami jednoduché objektové rozšíření vytvoříme.

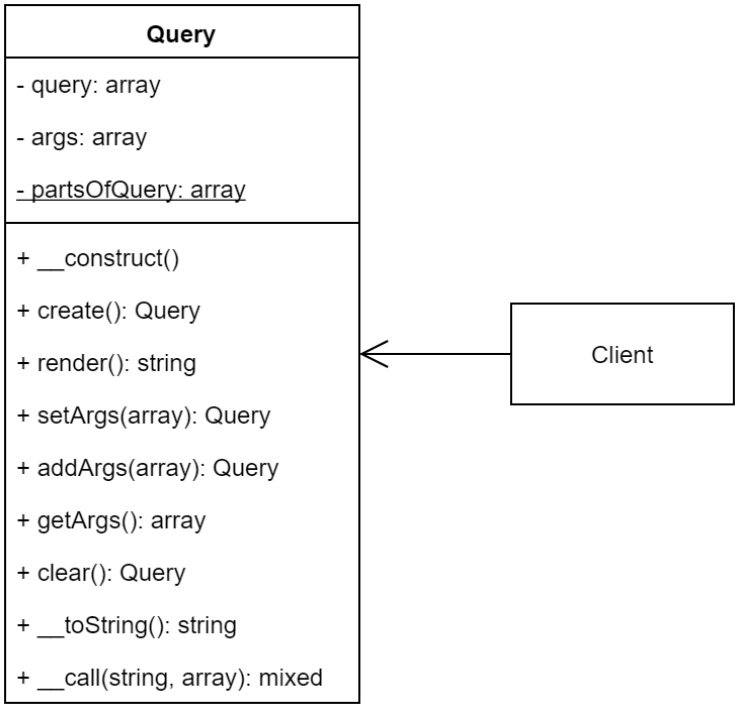
Pro příklady v této kapitole budeme používat jednoduchou jednotabulkovou databázi:



Obrázek 14 Schéma demo databáze o jedné tabulce

6.1 Dotaz

Nejdříve se však zamysleme nad SQL. Nejpoužívanějším SQL dotazem je SELECT. Takovýto dotaz může obsahovat mnoho částí (`SELECT`, `FROM`, `WHERE`, `GROUP BY`, `HAVING`, `ORDER BY`, `LIMIT`). Bylo by dobré, abychom s dotazem mohli zacházet jako s objektem. Vytvořit jej, postupně upravovat a v případě potřeby rychle vygenerovat řetězec jeho kompletního znění a pole argumentů.



Obrázek 15 Diagram tříd pro SQL dotaz

Příklad užití:

```
echo Database\Query::create()->setSelect("*")->setFrom("demo")->setWhere("id > 1");
```

Výstup:

```
SELECT * FROM demo WHERE id > 1
```

Nebo trochu komplexněji:

```
$query = new Database\Query();
$query->setSelect("id")->setFrom("demo")->setWhere("id > :val");
$query->addSelect("name")->addWhere("name LIKE 'I%'");
$query->setArgs(array(':val' => 1));
var_dump($query->render(), $query->getArgs());
```

Výstup:

```
string(60) "SELECT id, name FROM demo WHERE id > :val AND name LIKE 'I%'"
array(1) { [":val"]=> int(1) }
```

K tomuto nám bude stačit jedna třída Query. V této třídě budou dva atributy. Pole query pro jednotlivé části SQL dotazu a pole args pro seznam jednotlivých argumentů. Dále pak třídní (statické) pole partsOfQuery, ve kterém bude seznam jednotlivých částí dotazu SELECT. Jistě si vytvoříme i statickou metodu create(), která nám bude vytvářet instance třídy Query. Její využití je ukázáno hned u první ukázky této kapitoly.

```
<?php

class Query
{
    private $query;
    private $args;
    private static $partsOfQuery = [
        'select' => 'SELECT',
        'from' => 'FROM',
        'where' => 'WHERE',
        'group' => 'GROUP BY',
        'having' => 'HAVING',
        'order' => 'ORDER BY',
        'limit' => 'LIMIT'
    ];

    public function __construct()
    {
        $this->query = $this->args = [];
    }

    public static function create(): Query
    {
        return new Query();
    }
}
```

Dále pak už jen potřebujeme metody pro manipulaci s částmi dotazu `query` a parametry dotazu `args`. Pro `args` si vytvoříme metody `setArgs()`, `addArgs()` a `getArgs()`. Pro každou část dotazu není nutné vytvářet všechny settery, gettery a další obslužné operace. Je v tom systém a ten využijeme. Každou část dotazu můžeme nastavit (`set`), získávat (`get`), přidávat (`add`) a vyčistit (`clr`). Seznam povolených částí dotaz je uveden ve statickém poli `partsOfQuery`. V úvahu tedy přichází jen volání metod: `setSelect()`, `getWhere()`, `addFrom()`, `clrLimit()` atp. Můžeme tedy využít magické metody `__call()` a vše si řádně ošetřit tam:

```
public function __call($name, $arguments)
{
    $typeOfMethod = strtolower(substr($name, 0, 3));
    if (in_array($typeOfMethod, array('set', 'add', 'get', 'clr'))) {
        $name = strtolower(substr($name, 3));
        if (array_key_exists($name, self::$partsOfQuery)) {
            switch ($name) {
                case 'select':
                case 'order':
                case 'group':
                    $separator = ', ';
                    break;
                case 'where':
                case 'having':
                    $separator = ' AND ';
                    break;
                case 'from':
                default:
                    $separator = ' ';
                    break;
            }
            switch ($typeOfMethod) {
                case 'set':
                    $value = implode($separator, $arguments);
                    if (!empty($value)) {
                        $this->query[$name] = $value;
                    }
                    break;
                case 'add':
                    $value = implode($separator, $arguments);
                    if (!empty($value)) {
                        if (isset($this->query[$name]) && !empty($this->query[$name])) {
                            $value = $this->query[$name] . $separator . $value;
                        }
                        $this->query[$name] = $value;
                    }
                    break;
                case 'get':
                    if (isset($this->query[$name])) {
                       return $this->query[$name];
                    } else {
                        return NULL;
                    }
```

```
                    break;
                case 'clr':
                    if (isset($this->query[$name])) {
                        unset($this->query[$name]);
                    }
                    break;
            }
        }
    }
    return $this;
}
```

Metody `clear()`, `render()` a `__toString()` nejsou již ničím speciálně zajímavé. Kompletní řešení včetně jejich obsahu najdete samozřejmě v archívu celého příkladu.

6.2 Univerzální práce s databází

Při psaní kódu webové aplikace nechceme řešit detaily ohledně typu systému řízení báze dat, jeho ovladačů a podobně. Chceme mít nějakou nástavbu, která bude řešit zadávání dotazů. Která bude flexibilní. Tedy pokud se změní systém řízení báze dat například z MySQL na MS SQL Server, tak abychom nemuseli kvůli tomu měnit stovky řádků kódu, ale jen jednoduše změnili jednu třídu. Užití by mohlo vypadat takto:

```
try {
    $db = new Database\MySQL\DBDriver('127.0.0.1', 'demo', 'root', '');
    $db->connect();
    try {
        //$db->insert('demo', array('name' => 'Item', 'date' => date('Y-m-d H:i:s')));
    } catch (Database\DatabaseException $e) {
        echo $e->getMessage();
    }
    $query = new Database\Query();
    $query->setSelect("id")->setFrom("demo")->setWhere("id > :val");
    $query->addSelect("name")->addWhere("name LIKE 'I%'");
    $query->setArgs(array(':val' => 1));
    $result = $db->query($query->render(), $query->getArgs())->toArray();
    var_dump($result);
    $db->close();
} catch (Database\DatabaseException $e) {
    echo $e->getMessage();
}
```

Výsledek může být podobný tomuto:

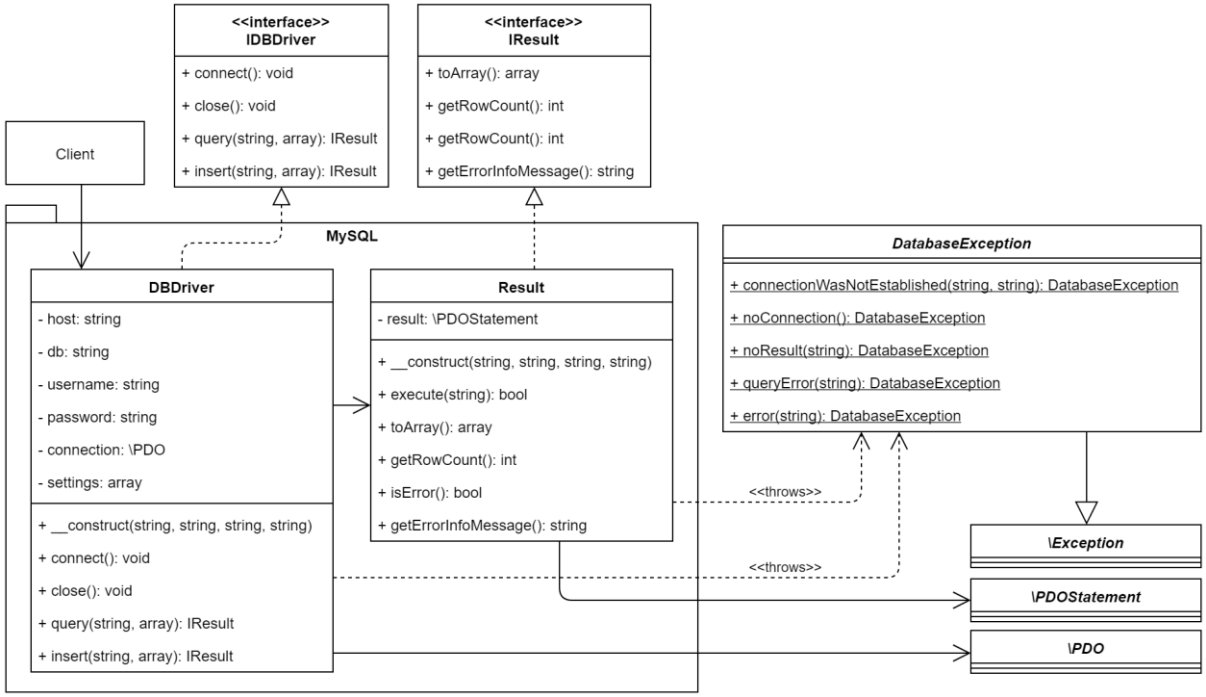
```
array (size=2)
  0 =>
    array (size=2)
      'id' => string '2' (length=1)
      'name' => string 'Item' (length=4)
  1 =>
    array (size=2)
      'id' => string '3' (length=1)
      'name' => string 'Item' (length=4)
```

Na začátku se jen přihlásíme do MySQL a vybereme databázi. Veškeré další operace jsou již na typu systému řízení báze dat zcela nezávislé. Připojíme se. Připravíme si dotaz. Pošleme jej do databáze a dostaneme výsledek v podobě asociativního pole. Budeme-li chtít přidat záznam do databáze, použijeme tento zápis:

```
$db->insert('demo', array('name' => 'Item', 'date' => date('Y-m-d H:i:s')));
```

Abychom přidali záznam do databáze, potřebujeme znát jen tabulku a hodnoty záznamu, které předáme jako pole. Ani nepotřebujeme znalost SQL dotazu `INSERT`. Programování na takovéto míře abstrakce nám ušetří mnoho práce a kód je navíc dobře čitelný a univerzální. Pro další projekty můžeme jednoduše naši nástavbu znovu použít.

Řešení tohoto problému nám poskytnou třídy implementující základní rozhraní `IDBDriver` a `IResult`. Příkladem je ovladač `MySQL\DBDriver` a třída pro zpracování SQL dotazu `MySQL\Result`. Ovladače implementují klíčové výkonné operace (query, insert a případně i update a delete) pro konkrétní ovladač do DBMS. Při změně systému řízení báze dat pak stačí doprogramovat jen tyto dvě třídy, které splňují patřičná rozhraní. Celý zbylý kód vaší aplikace může zůstat beze změny. Nesmíme zapomenout ani na základní ošetření chyb pomocí zachytávání výjimek. K tomu jsme si vytvořili vlastní třídu `DatabaseException`.



Obrázek 16 Diagram tříd pro objektové databázové rozšíření

Ukážeme si nyní rozhraní `IDBDriver` a `MySql\DBDriver`, který slouží jako ovladač do MySQL využívající PDO.

IDBDriver.php:

```
namespace TWA\PraceSDatabazi\Database;
```

```
interface IDBDriver
{
    public function connect();
    public function close();
    public function query(string $queryString, array $queryArgs = []): IResult;
    public function insert(string $table, array $values): IResult;}
```

MySQL/DBDriver.php:

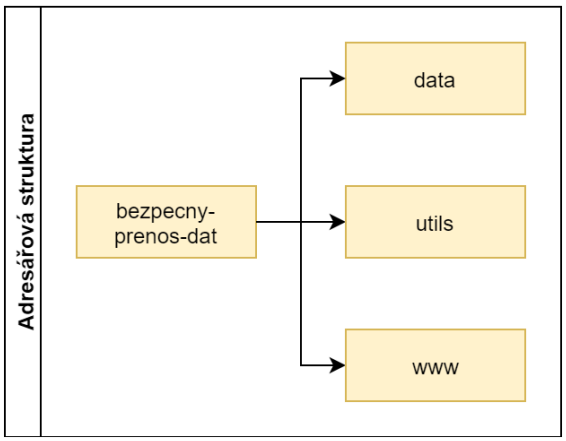
```
namespace TWA\PraceSDatabazi\Database\MySQL;

use TWA\PraceSDatabazi\Database\IResult;
use TWA\PraceSDatabazi\Database\IDBDriver;

/**
 * MySQL PDO driver
 */
class DBDriver implements IDBDriver
{

    private $host;
    private $db;
    private $username;
    private $password;
    private $connection = null;

    private static $settings = [
        \PDO::ATTR_ERRMODE => \PDO::ERRMODE_EXCEPTION,
        \PDO::MYSQL_ATTR_INIT_COMMAND => "SET NAMES utf8"
    ];

    public function __construct($host, $db, $username, $password)
    {
        $this->host = $host;
        $this->db = $db;
        $this->username = $username;
        $this->password = $password;
    }

    public function connect()
    {
        try {
            $this->connection = new \PDO("mysql:host=$this->host;dbname=$this->db",
$this->username, $this->password, self::$settings);
        } catch (\PDOException $e) {
            throw DatabaseException::connectionWasNotEstablished('MySQL', $e-
>getMessage());
        }
    }

    public function close()
    {
        $this->connection = NULL;
    }
```

```
    public function query(string $queryString, array $queryArgs = []): IResult
    {
        if (!$this->connection instanceof \PDO) {
            throw DatabaseException::noConnection();
        }
        try {
            if (is_array($queryArgs) && !empty($queryArgs)) {
                $result = new Result($this->connection->prepare($queryString));
                $result->execute($queryArgs);
            } else {
                $result = new Result($this->connection->query($queryString));
            }
        } catch (\PDOException $e) {
            throw DatabaseException::queryError($e->getMessage());
        }
        if (!$result->isError()) {
            return $result;
        } else {
            throw DatabaseException::queryError($result->getErrorInfoMessage());
        }
    }

    public function insert(string $table, array $values): IResult
    {
        if (is_array($values) && !empty($values)) {
            $colNames = implode(',', array_map(function ($v, $k) {
                return '`' . $k . '`';
            }, $values, array_keys($values)));
            $colValKeys = implode(',', array_map(function ($v, $k) {
                return ':' . $k;
            }, $values, array_keys($values)));
            return $this->query('INSERT INTO `' . $table . '` (' . $colNames . ') VALUES
(' . $colValKeys . ')', $values);
        } else {
            throw DatabaseException::error("Bad args");
        }
    }
}
```

Kompletní řešení včetně dokumentačních komentářů naleznete v archívu této úlohy.

7 Bezpečné zasílání dat a přihlašování

V této kapitole si ukážeme, jak ověřit validitu přijatých dat. Také si ukážeme, jak se bezpečně přihlásíme na nebezpečném spojení. Pro oba naše příklady předpokládejme tuto adresářovou strukturu:



Obrázek 17 Adresářová struktura ukázky bezpečného zasílání dat a přihlašování

Ve složce data předpokládejme soubory uvedené v následující tabulce.

Tabulka 2 Pracovní soubory ze složky data

Název souboru	Význam
_email	demonstrativní přihlašovací jméno
_password	demonstrativní přihlašovací heslo
password_hash	hash hesla
salt	náhodně vygenerovaná sůl
time	datum a čas požadavku na přihlášení

Ve složce www předpokládejme soubory pro kaskádové styly a sha256.js, který, jak název napovídá, umí JavaScriptem na webovém klientovi hašovat funkcí sha256.

Protože budeme pracovat se soubory, generovat HTML kód, generovat náhodné řetězce a haše, vytvoříme si rychle jednoduché pomocné třídy do složky utils. Budou sloužit jen jako zapouzdření pracovních funkcí. Důvodem je přehlednější zdrojový kód.

File.php:

```
class File
{
    public static $path = 'data/';

    public static function read(string $filename)
    {
        if (file_exists(self::$path . $filename)) {
            return trim(file_get_contents(self::$path . $filename));
        } else {
```

```
            return false;
        }
    }

    public static function write(string $filename, string $data)
    {
        $fd = @fopen(self::$path . $filename, "w");
        if ($fd) {
            fwrite($fd, trim($data));
            fclose($fd);
        }
    }
}
```

Html.php:

```
class Html
{
    public static function showTable(string $caption, array $data)
    {
        echo "<table><caption>$caption</caption>";
        foreach ($data as $key => $val) {
            echo "<tr><th>$key</th><td>$val</td></tr>";
        }
        echo "</table>";
    }

    public static function message(string $text)
    {
        echo "<p>$text</p>";
    }
}
```

Security.php:

```
class Security
{
    public static function generateToken(int $length = 32): string
    {
        $characters = 'abcdefghijklmnopqrstuvwxyz0123456789';
        $token = '';
        for ($i = 0; $i < intval($length); $i++) {
            $token .= $characters[mt_rand(0, strlen($characters) - 1)];
        }
        return $token;
    }

    public static function hash($data): string
    {
        return hash('sha256', is_array($data) ? implode('', $data) : $data);
    }

    public static function checkHash(string $hash, string $data): bool
    {
```

```
        return self::hash($data) == $hash;
    }
}
```

`File` nám tak umí jednoduše číst a psát ze souboru. `Html` umí jednorozměrné pole zobrazit jako HTML tabulku a textový řetězec jako zprávu v odstavci. Třída `Security` je v těchto příkladech nejdůležitější. Jsou v ní metody, které umí vygenerovat náhodný token, hašovat data, ověřit, zdali jsou data hašována patřičným hašem.

Hašovací funkce převede „neomezená" vstupní data obvykle do čísla v šestnáctkové soustavě. Vstupní data jsou omezená jen omezeními technologií. Výstupu se říká haš (hash). Tato funkce je tedy již z podstaty jednosměrná. Z malého haše neexistuje způsob získání původních „neomezených" dat. Zranitelnost hašování spočívá v tom, aby se lehce nenašel jiný vstup, který má stejný hash. Vzhledem k „neomezenosti" velikosti vstupu a „omezenosti" velikosti výstupu, bude jistě mnoho vstupů mít společný výstup. Bezpečnost můžeme tedy zvýšit velikostí výstupu. Dobrá hašovací funkce reaguje na minimální změny vstupu maximálními změnami výstupu. Malými změnami vstupu se tedy nedá postupně dostat k výsledku.

7.1 Zasílání dat

Představme si, že potřebujeme metodou GET, tedy přímo v URL, poslat informaci o akci a emailu. Tato data nám může v URL kdokoliv změnit. Jak tedy ověřit, že data, která jsme do odkazu vygenerovali, nejsou změněna? Princip je jednoduchý. K datům přidáme haš (hash), který vytvoříme z originálních dat a hesla. Tím se jednak ověří nepozměnění dat a jednak původní autorství. Mějme tedy například soubor button.php, který obsahuje stránku s tlačítkem s těmito demo daty:

```
$data = ['action' => 'send', 'email' => Util\File::read('_email')];
Util\Html::showTable('Data', $data);
```

Nyní si postupně vytvoříme jen data pro GET, heslo, data pro haš, haš a výsledný GET data s podpisem:

```
$work = [];
$work['GET bez podpisu'] = http_build_query($data);
$work['Heslo'] = Util\File::read('_password');
$work['Data pro haš'] = implode(':', $data) . ":" . $work['Heslo'];
$work['Haš'] = Util\Security::hash($work['Data pro haš']);
$work['GET s podpisem'] = http_build_query($data + array('hash' => $work['Haš']));
```

Nyní si vytvoříme odkaz s daty:

```
echo '<a href="button-check.php?' . $work['GET s podpisem'] . '">Zpracuj</a>';
```

Pro kontrolu si můžeme vypsat pracovní pole:

```
Util\Html::showTable('Pracovní pole s mezivýsledky', $work);
```

Obdržet bychom měli něco podobného tomuto:

Obrázek 18 Snímek obrazovky se zasíláním dat pomocí odkazu

Nyní se podíváme na skript v button-check.php, který přijatá data ověří. Nejprve si načteme a zobrazíme získaná data:

```
$data = [
    'action' => filter_input(INPUT_GET, 'action', FILTER_SANITIZE_STRING),
    'email' => filter_input(INPUT_GET, 'email', FILTER_SANITIZE_EMAIL),
    'haš' => filter_input(INPUT_GET, 'hash', FILTER_SANITIZE_STRING)
];
Util\Html::showTable('Získaná GET data', $data);
```

Všimněte si, že data nezískáváme přímo z pole `_GET`, ale metodou `filter_input()`, která nám umožní dle potřeby data validovat nebo sanitizovat. Nyní si připravíme pole data tak, abychom si z něj mohli vypočítat nový haš. Odstraníme načtený haš a přidáme heslo:

```
unset($data['haš']);
$data['password'] = Util\File::read('_password');
```

Vytvoříme si pracovní pole work a do něj postupně uložíme data vedoucí až k novému haši:

```
$work = [
    'Heslo ze souboru' => Util\File::read('_password'),
    'Data pro haš' => implode(':', $data),
    'Nový haš' => Util\Security::hash(implode(':', $data))
];
Util\Html::showTable('Vypočtená data', $work);
```

Nyní si již ověříme shodu nového vypočteného haše s původním hašem v GET datech:

```
if ($work['Nový haš'] == filter_input(INPUT_GET, 'hash', FILTER_SANITIZE_STRING)) {
    echo '<p>Data jsou ověřena.</p>';
} else {
    echo '<p>Data nejsou ověřena.</p>';
}
```

Zpracování tlačítka

Získaná GET data

action	send
email	john.doe@example.com
haš	191a405a227fc20e6d9a15f725fd12a9e5eaf7238c11274b517259a91d064158

Vypočtená data

Heslo ze souboru	1234
Data pro haš	send:john.doe@example.com:1234
Nový haš	191a405a227fc20e6d9a15f725fd12a9e5eaf7238c11274b517259a91d064158

Data jsou ověřena.

Obrázek 19 Snímek obrazovky s přijatými daty z tlačítka

Podobný princip se používá například při důvěryhodné komunikaci se servery platebních bran. Podstatný rozdíl je však v tom, že heslo není jen jedno a nemusí se tedy sdílet. Používá se privátních a veřejných klíčů.

7.2 Přihlašování

Začneme nejhorším možným způsobem přihlašování a postupně jej budeme zlepšovat. Pro začátek mějme tedy komunikaci na nezabezpečeném protokolu HTTP. Na straně klientské bude jednoduchý přihlašovací formulář:

```
<form action="bad-login-go.php" method="GET">
    Email: <input type="text" name="email">
    Heslo: <input type="text" name="password">
    <input type="submit" name="submit" value="Přihlásit">
</form>
```

Na straně serveru bude uživatelské heslo v databázi nešifrované. Přihlášení se bude ověřovat například takto:

```
if ($serverData["Uživatelský email"] == $_GET['email']) {
    if ($serverData["Uživatelské heslo"] == $_GET['password']) {
        echo "Uživatel přihlášen.";
    } else {
        echo "Špatně zadané heslo.";
    }
} else {
    echo "Špatně zadaný email.";
}
```

Tento způsob je nešťastný z mnoha důvodů, a přesto se s podobnými přístupy setkávám. Sami jistě najdete spousty chyb a bezpečnostních rizik. My si mnohé ukážeme, vysvětlíme a naznačíme řešení.

Zkuste si před pokračováním ve čtení udělat vlastní seznam hrozeb výše uvedeného příkladu.

Jaké chyby vidíme přímo v uvedeném příkladu:

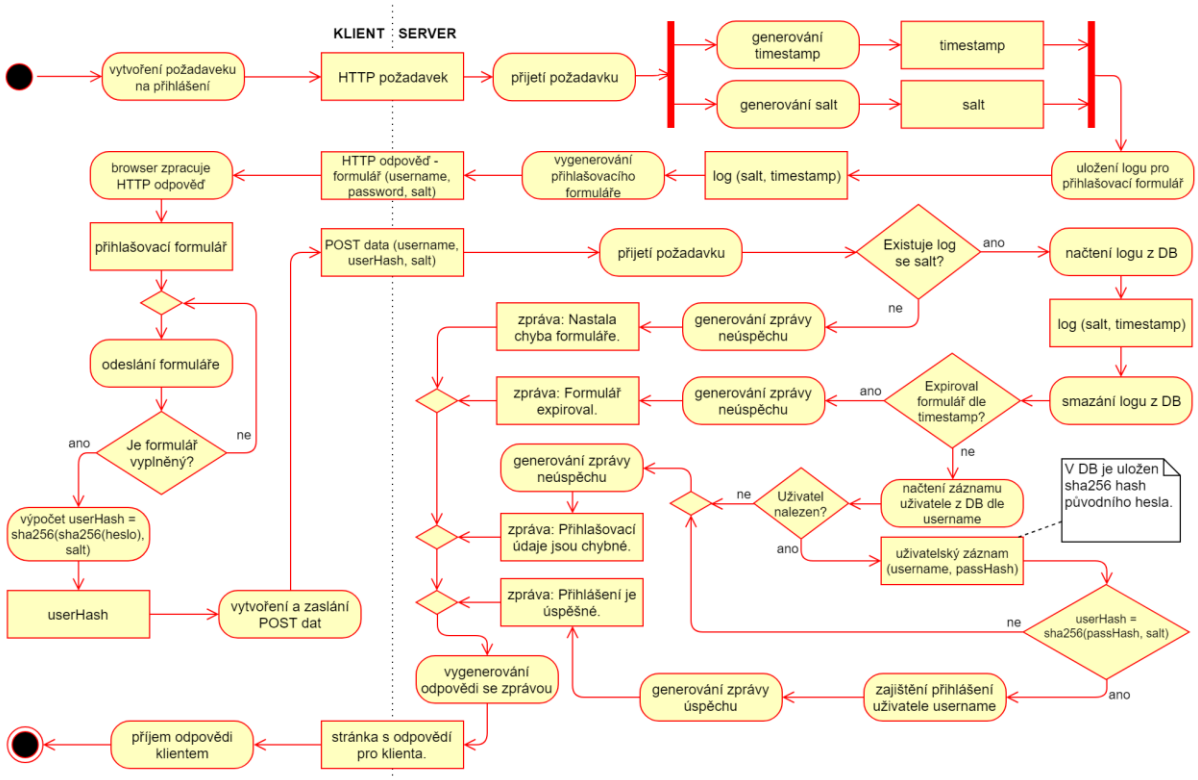
1. Data z přihlašovacího formuláře neposílejme metodou GET, protože poslané hodnoty jsou vidět v URL adrese a je velmi jednoduché je změnit. POST data lze sice také podvrhnout, ale už je to o trochu pracnější, a navíc se alespoň nezobrazuje například heslo po přihlášení v URL adrese.
2. Heslo by se mělo zadávat do formulářového pole typu „password“ a ne „text“. Tím se při zadávání hesla nezobrazují zadávané znaky.
3. Využijme plně jazyka HTML a nadefinujme již zde co nejlépe jednotlivá pole. I email by se měl posílat polem typu „email“. Tím se před odesláním ověří, má-li zadaný text podobu emailu. Určit můžeme i další mnohé parametry polí, které ověří již na straně klienta minimální nebo maximální délku zadaného řetězce, zdali bylo pole vůbec zadané a podobně. Návěští pro input je vhodné implementovat jako label, který s inputem řádně provážeme. Myslíme i na přístupnost webového obsahu.
4. Nikdy nepracujme přímo s daty pole `_GET` a `_POST`. Mějme na mysli, že tato data mohou být jakkoliv podvržená. Viděl jsem i vložení těchto dat přímo do SQL dotazu `SELECT`, který ověřoval uživatele v databázi. To si koleduje o SQL Injection. Vždy je tedy před použitím potřeba GET a POST data řádně sanitizovat. Dobré je použít funkci `filter-input()`. Pokud z nějakého důvodu `filter-input()` nechceme použít, můžeme například při očekávání celého kladného čísla ze vstupu použít kombinaci `intval()` a `abs()`.
5. Špatně je zde navržena i soustava podmínek, a to z důvodu nevhodného hlášení úspěchu či neúspěchu přihlášení. Není vhodné informovat návštěvníka separátně o špatném emailu a heslu. V tomto případě platí, že méně je více. Při špatném emailu nebo heslo stačí hlásit něco univerzálního, například „Špatné přihlašovací údaje“. Útočník si tak nemůže jednoduše ověřovat existenci nějakého účtu v systému.

Jak koncepčně změnit způsob přihlašování a jaké otázky si dále klást:

1. Heslo na serveru by mělo být šifrované. Obvykle se k tomu používá silná hašovací funkce, aby bylo téměř nemožné získat původní heslo zpět. Tím se chráníme před odhalením hesla při úniku dat ze serveru. Dejte si pozor na služby, které vám heslo při jeho ztrátě pošlou. Vaše heslo byste měli znát jen vy a ani na službě, kde máte váš účet, by neměl být možný způsob získání vašeho původního hesla. Někdo se může ptát: „Když tedy není možné získat heslo zpět a v databázi je jen jeho haš, jak je možné provést ověření hesla při přihlašování?“ Na to je jednoduchá odpověď. Neporovnávají se původní hesla, ale jejich haše.
2. Sítí by nemělo putovat otevřené heslo. Je tedy vhodné toto heslo ještě před odesláním na server šifrovat. K tomu se může použít opět hašovací funkce. Tímto si pomůžeme od odhalení našeho hesla, ale pokud někdo odchytí náš haš, tak se může přihlásit právě pomocí něho. Je tedy vhodné, aby byl haš

stejného hesla pokaždé jiný. To zajistíme tak, že k heslu přidáme sůl (salt). Tu může představovat například nějaké pseudonáhodné číslo. Na serveru tedy pro patřičný přihlašovací formulář vygenerujeme sůl, uložíme si ji a pošleme s formulářem. Na klientovi se při odeslání vyplněného přihlašovacího formuláře vytvoří haš z hesla a soli, která je pokaždé jiná, tedy i haš bude pokaždé jiný. Nyní je opět na místě otázka: „Jak ověříme správnost hesla při přihlašování, když jeden haš je pokaždé jinak osolen?“. Dejme tomu, že používáme hašovací funkci sha256(). Na serveru je tedy uložen hašHesla = sha256("mojeTajnéHeslo"). Na serveru je také uložena sůl = "123abc". Tato sůl je pro každý přihlašovací formulář jiná. Pro přihlašování potřebujeme serverLoginHaš = sha256(hašHesla + sůl). Na straně klienta známe sůl, protože jsme si ji poslali například jako hodnotu atributu nějakého elementu na stránce. Heslo uživatel zadá do přihlašovacího formuláře. Můžeme tedy na straně klienta před odesláním vygenerovat klientLoginHaš = sha256(sha256("hesloVeFormuláři") + sůl). Matematické postupy jsou na obou stranách identické, a pokud klientLoginHaš = serverLoginHaš pak i mojeTajnéHeslo = hesloVeFormuláři.

3. Na jeden přihlašovací formulář by mělo být možné odpovědět pouze jednou. Je tedy dobré při prvním zpracování na serveru sůl mazat.
4. Přihlašovací formulář by měl mít časově omezenou platnost. Je tedy dobré si k soli uložit i časové razítko a před ověřováním uživatele testovat platnost formuláře. Podobně i samotná doba nečinnosti uživatele po přihlášení by měla být také omezena.
5. Používejme zabezpečený protokol HTTPS, který využívá bezpečnostní protokoly SSL (Secure Sockets Layer), TLS (Transport Layer Security) a standard pro digitální certifikáty X.509. Firmy nabízející hostingové služby jej běžně a obvykle bezplatně podporují. HTTPS je již standard, který mnohé třetí strany, které můžete na webu používat (sociální sítě, cloudové služby, bankovnictví…), vyžadují.



Obrázek 20 Schéma logiky přihlašování

Před generování přihlašovacího formuláře si nejprve připravíme data:

```
Util\File::write('password_hash', Util\Security::hash(Util\File::read('_password')));
Util\File::write('salt', Util\Security::generateToken());
Util\File::write('time', time());
```

Do souboru password_hash si uložíme haš obsahu souboru _password, ve kterém je pracovně naše otevřené heslo. Vygenerujeme si náhodnou sůl a uložíme ji do souboru salt. Podobně tak i časové razítko do souboru time. Přihlašovací formulář by mohl vypadat již takto:

```
<form id="loginForm" action="login-go.php" method="POST">
    <label for="user">Email:</label>
    <input id="user" type="email" name="email" value="" maxlength="100" autofocus
required>
    <label for="pass">Heslo:</label>
    <input id="pass" type="password" name="password" value="" maxlength="10" required>
    <input id="salt" type="hidden" name="salt" value="<?php echo Util\File::read('salt');
?>">
   <input type="submit" name="submit" value="Přihlásit">
</form>
<script src="www/sha256.js"></script>
<script>
    document.getElementById("loginForm").onsubmit = function () {
        var pass = document.getElementById('pass').value;
        var salt = document.getElementById('salt').value;
        document.getElementById('pass').value = CryptoJS.SHA256(CryptoJS.SHA256(pass) +
salt);
```

```
    };
</script>
```

HTML formulář si již hlídá povinná pole, podobu zadaného emailu, maximální počty znaků a také před odesláním se provede haš hesla, který se solí a dohromady ještě jednou hešují.

Na straně serveru si připravíme data z formuláře a serverová data:

```
$postData = [
    'email' => filter_input(INPUT_POST, 'email', FILTER_SANITIZE_EMAIL),
    'password' => filter_input(INPUT_POST, 'password', FILTER_SANITIZE_STRING),
    'salt' => filter_input(INPUT_POST, 'salt', FILTER_SANITIZE_STRING)
];

$serverData = [
    'Uživatelský email' => Util\File::read('_email'),
    'Uživatelské heslo' => Util\File::read('_password'),
    'Heslo v databázi' => Util\File::read('password_hash'),
    'Sůl' => Util\File::read('salt'),
    'Časové razítko požadavku' => Util\File::read('time')
];
```

Jednotlivá ověření již provedeme jednoduše:

```
if ($serverData["Sůl"] == $postData['salt']) {
    …
} else {
    …
}
if (time() >= $serverData['Časové razítko'] &&
                                time() <= $serverData['Časové razítko'] + 120) {
    …
} else {
    …
}
if ($serverData['Uživatelský email'] == $postData['email']) {
    …
} else {
    …
}
if (Util\Security::hash($serverData['Heslo v databázi'] . $serverData['Sůl']) ==
$postData['password']) {
    …
} else {
    …
}
```

V balíčku zdrojových kódů k této úloze najdete kompletní příklad, ve kterém je vše postupně ověřováno a výsledek přihlašování může vypadat jako na následujícím obrázku. Pro ostrý provoz bychom samozřejmě podmínky přeorganizovali a šetřili bychom hlášeními. Také přístup k datům by byl již přes relační databázi, a nikoliv jednoduše ze souboru.

Proces přihlášení

Data na serveru

Uživatelský email	john.doe@example.com
Uživatelské heslo	1234
Heslo v databázi	03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4
Sůl	6kcrbt2ay7wwrpfzy8r7vvib14n6gg8s
Časové razítko požadavku	1550852707

Data přicházející z formuláře metodou POST (To co jde po síti.)

email	john.doe@example.com
password	9b500672f598406a54d47365c15bb8ed86950d72a86ad60b8d7ca37f48bc8a69
salt	6kcrbt2ay7wwrpfzy8r7vvib14n6gg8s

Postup ověření

1. Nalezení požadavku na zobrazení formuláře
   - V databázi se nalezne požadavek třeba dle soli.
   - U takovéhoto řešení je však potřeba zajistit, aby sůl byla unikátní. Toho lze docílit třeba pomocí unikátního prefixu, který bude id požadavku.
   - Pokud se požadavek nenajde, přihlášení bude zamítnuto.
   - Nezpracované požadavky po expiraci se smazají.
   - **Sůl poslána formulářem se shoduje se solí na serveru.**
2. Ověření platnosti požadavku
   - Časové razítko požadavku je nyní: 1550852707
   - Formulář platný například na 2 minuty (120 sekund) musí být načten mezi: 1550852707 a 1550852827
   - Aktuální časové razítko: 1550852713
   - **Platnost formuláře ještě nevypršela.**
3. U nalezeného uživatele provedeme autentizaci (autentifikaci, ověření identity uživatele)
   1. Nalezení uživatele dle uživatelského emailu
      - Uživatelský email v databázi: john.doe@example.com
      - Zadané uživatelské jméno do formuláře: john.doe@example.com
      - **Uživatel byl nalezen.**
   2. Ověříme heslo
      - Uživatelské heslo v databázi (jeho hash): 03ac674216f3e15c761ee1a5e255f067953623c8b388b4459e13f978d7c846f4
      - Zadané původní uživatelské heslo do formuláře: Neznámé!
      - Příchozí uživatelské heslo z formuláře: 9b500672f598406a54d47365c15bb8ed86950d72a86ad60b8d7ca37f48bc8a69
      - sha253(heslo z DB + sůl požadavku z DB): 9b500672f598406a54d47365c15bb8ed86950d72a86ad60b8d7ca37f48bc8a69
      - **Heslo bylo zadáno v pořádku.**

Obrázek 21 Snímek obrazovky po přihlašování

8 Seznam obrázků

9 Seznam tabulek