Veronika Pavlíková & Martin Trnečka

Webové aplikace

Katedra informatiky, Přírodovědecká fakulta, Univerzita Palackého v Olomouci

Tento text slouží jako doplňkový učební materiál pro předmět *Webové aplikace*, jenž je vyučován v rámci bakalářského studia na Katedře informatiky Přírodovědecké fakulty Univerzity Palackého v Olomouci. Kurz je zaměřen na vybrané technologie umožňující tvorbu webový aplikací a předpokládá základní znalosti programování, tvorby webových stránek a sítě Internet. Kurz volně navazuje na kurzy *Úvod do informačních technologií*, *Tvorba webových stránek* a *Počítačové sítě 1*. Přestože jejich dřívější absolvování není podmiňující pro absolvování tohoto kurzu, je velmi doporučené. Některé aspekty, které byly podrobně adresovány v uvedených kurzech, budou v tomto textu zestručněny na potřebnou úroveň. Text je průběžně rozšiřován a doplňován, ale vzhledem k rychlému vývoji webových technologií mohou být některé části textu zastaralé. Poslední změna proběhla 27. listopadu 2023. Pokud v textu naleznete chyby, prosím, kontaktujte nás prostřednictvím e-mailové adresy martin.trnecka@upol.cz.

Veronika Pavlíková & Martin Trnečka

Obsah

8 Webové aplikace

Webové aplikace

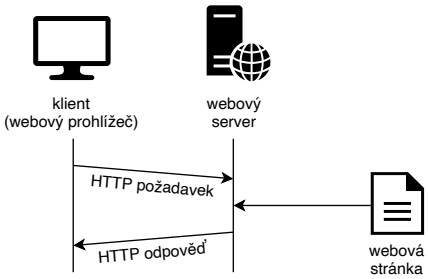
„People tend to think of the web as a way to get information or perhaps as a place to carry out ecommerce. But really, the web is about accessing applications. Think of each website as an application, and every single click, every single interaction with that site, is an opportunity to be on the very latest version of that application."

Marc Andreessen

Tento kurz se věnuje tvorbě *webových aplikací*. Pojem webová aplikace je často vysvětlován velmi neurčitě. Důvodů je hned několik. Jednak je v dnešní době poměrně obtížné vymezit co je to webová aplikace a samotná podoba a prostředky pro tvorbu webových aplikací se v průběhu vývoje sítě Internet, respektive služby WWW, značně měnily.

Historicky se pojem webová aplikace objevil již v první etapě vývoje služby WWW (označované jako Web 1.0), kdy byly webové stránky pouze statické. V této době webová aplikace označovala webové stránky, které navíc obsahovaly programový kód (přesněji řečeno skript) na straně klienta.[1]

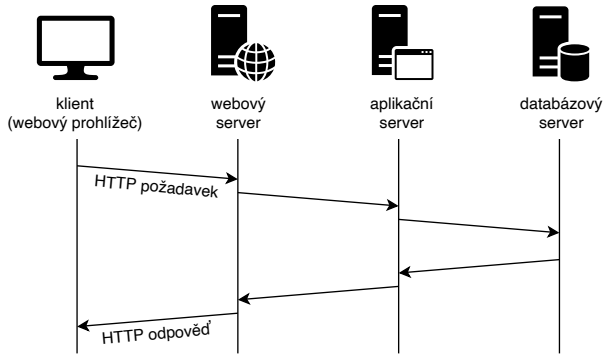
Statická webová stránka je tvořena statickým obsahem, jehož součástí je i programový kód, který je vždy vrácen webovému prohlížeči, jenž jej interpretuje.[2] Schématicky je možné statickou stránku zobrazit následovně.



Později, s nástupem etapy Web 2.0, se webová aplikace stala synonymem pro dynamickou webovou stránku, jejíž ilustrační schéma následuje.

[1] První webové aplikace se objevily v roce 1995, kdy dnes již neexistující společnost Netscape představila JavaScript.

[2] Tvorbě statických webových stránek se věnuje kurz *Tvorba webových stránek.*



O vytvoření stránky se stará *aplikační server*, který ji vytváří dle požadavků *klienta*.[3] K vytvoření stránky může aplikační server využit informace uložené na *databázovém serveru*.[4] Veškerá komunikace mezi klientem a serverem je zprostředkovávána skrze *webový server*, který přijímá požadavky od klienta (v podobě HTTP požadavků) a vrací klientovi výslednou webovou stránku (jako odpověď na HTTP požadavek). Stejně jako v případě statické webové stránky, může být součástí výsledné dynamické webové stránky programový kód, který je dále interpretován webovým prohlížečem.

Obecně můžeme říct, že webová aplikace označuje webové stránky, kde je ve větší míře používán programový kód na straně klienta nebo serveru.[5] V dnešní době nemají webové aplikace podobu pouze dynamické webové stránky, ale mohou být realizovány jako mobilní nebo desktopové aplikace. Poněkud přesněji můžeme webovou aplikaci vymezit jako aplikaci, která potřebuje pro neomezené fungování přístup k Internetu nebo jádro prohlížeče.

V tomto kurzu budeme pro jednoduchost nahlížet na webovou aplikaci jako na dynamickou webovou stránku a postupně se budeme věnovat technologiím na klientovi, webovém, aplikačním a databázovém serveru,[6] přičemž se budeme věnovat pouze vybraným technologiím a především základním principům uplatňovaným při tvorbě webových aplikací.

[3] Aplikační server není nic jiného než program, jehož úkolem je sestavit stránku dle zadaných parametrů.

[4] Dodejme, že databázový server nemusí nutně být ve výše uvedeném schématu přítomen.

[5] Později uvidíme, že to, zda se nachází podstatná část programového kódu na klientu nebo serveru se v průběhu vývoje webových aplikací opakovaně měnilo.

[6] V textu je pojem webová aplikace a (dynamická) webová stránka volně zaměňován.

Webové versus klasické aplikace

Pro úplnost se podívejme na srovnání klasických[7] a webových aplikací. Významnou výhodou webových aplikací je jejich multiplatformnost. Ke svému fungování jim stačí pouze jádro webového prohlížeče a přístup k Internetu, přičemž druhé zmíněné nemusí být vyžadováno vždy.[8] Díky tomu, že jsou webové aplikace dostupné skrze síť Internet, je možné k nim přistupovat vzdáleně. S tím je spojena i možnost vynucení aktualizace aplikace. Obojí je u klasických aplikacích, v porovnání s webovými, výrazně obtížnější. V neposlední řadě je velkou předností webových aplikací *škálovatelnost*.

[7] Klasickou aplikací máme na mysli desktopovou nebo mobilní aplikaci, která je umístěna na jednom zařízení.

[8] Jádro webového prohlížeče je v dnešní době přístupné na všech zařízeních, které mají nějaký „rozumný“ operační systém.

Tím je myšleno, že je možné jejich vhodnou architekturou snadno navyšovat výpočetní výkon, dostupnou paměť a také dostupnost. Klasické aplikace v tomto ohledu narážejí na limity zařízení, na kterém jsou provozovány.[9]

Na druhou stranu v případě webových aplikací je náročnější zajištění bezpečnosti, a to zejména kvůli možnosti vzdáleného přístupu, distribuci a komunikaci skrze síť Internet. Webové aplikace jsou rovněž po technologické stránce výrazně náročnější. Zatímco při tvorbě klasické aplikace je dostačující ovládnout poměrně kompaktní ekosystém technologií, v případě tvorby webových aplikací je třeba ovládat mnoho rozličných technologií.

[9] Dodejme, že ve své podstatě se jedná pouze o rozdíl mezi centralizovanou a decentralizovanou architekturou. Uvedené výhody pramení z toho, že webové aplikace je možné poměrně snadno realizovat decentralizovaně.

Závěr

V úvodní kapitole jsme orientačně vymezili pojem webová aplikace a diskutovali jejich výhody a nevýhodo oproti klasickým aplikacím.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to webová aplikace?
2. Jaké jsou výhody a nevýhody webových aplikací oproti klasickým aplikacím?
3. Uveďte několik příkladů webových aplikací.

12 Webové aplikace

Uživatelské rozhraní

„If you think math is hard, try web design."

Trish Parr

V následující části textu se budeme věnovat tvorbě uživatelského rozhraní (UI) webové aplikace. Pokud se vrátíme ke schématu webové aplikace, které jsme představili v předešlé kapitole, jedná se o tu část aplikace, která je zaslána zpět klientovi, jako odpověď na HTTP požadavek. V případě, že webová aplikace disponuje uživatelským rozhraním,[10] je toto rozhraní realizováno pomocí standardních (klientských) webových technologií: HTML, CSS a JavaScript. Ve své podstatě je tedy tvorba uživatelského rozhraní webové aplikace tvorba webové stránky.

Uživatelské rozhraní webové aplikace se ale od typické webové stránky liší. Zatímco webová stránka je určena spíše k pasivnímu používání,[11] u webové aplikace se předpokládá její aktivní ovládání uživatelem. Běžně se používají standardní ovládací prvky jako například inputboxy, checboxy, tlačítka nebo dropdown menu. Také je kladen důraz na přehlednost a celkovou snadnost ovládání.

Typickým rysem je, že se jednotlivé části uživatelského rozhraní neustále opakují.[12] Uveďme si jednoduchý příklad: administrační rozhraní pro e-shop. Takové rozhraní bude zcela určitě obsahovat jako jeden z ovládacích prvků tlačítko a to hned v několik variantách, jako například „smazat" nebo „přidat". Dále bude obsahovat výpis položek opět v několik variantách. Například výpis produktů nebo výpis objednávek.

Jednotlivé části uživatelského rozhraní se označují jako *komponenty*. Komponentou může být například tlačítko, ale i větší část uživatelského rozhraní (například dříve zmíněný výpis položek) nebo dokonce celá stránka. Komponenty mohou být složeny z dalších komponent. Příklad komponent je ilustrován na obrázku 1.

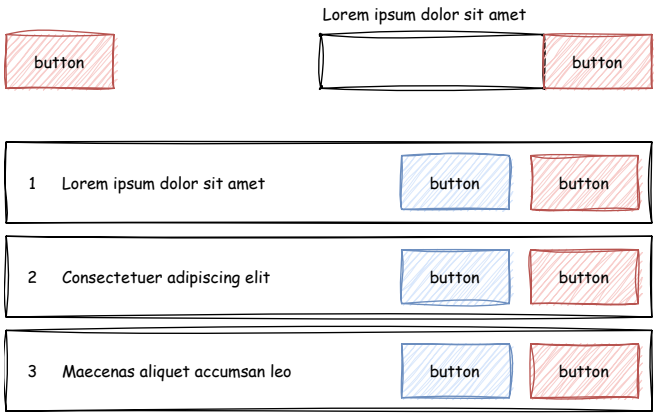
Přestože nám nic nebrání si uživatelské rozhraní vytvořit od samotného začátku, v tomto kurzu půjdeme jinou cestou a použijeme existující (univerzální) řešení. Ve skutečnosti je použití existujícího řešení pro tvorbu uživatelského rozhraní webové aplikace naprosto běžné a ušetříme si jím značné množství času a práce. Dodejme ale, že dobré uživatelské rozhraní vytvořené na míru, je pro uživatele

[10] Některé webové aplikace nemusí mít uživatelské rozhraní, případně jejich UI je triviální. Typicky se jedná o webové služby.

[11] Typická webová stránka nebo také prezentace je určena především ke čtení. Tomu se přizpůsobuje její design a celkové možnosti ovládání.

[12] Uživatelské rozhraní webových aplikací je obvykle založeno na design systému.

Průvodce studiem
Při torbě uživatelského rozhraní webové aplikace se často uplatňuje komponentní přístup. Ten je diskutován v kurzu *Tvorba webových stránek*.



Obrázek 1: Příklad několika komponent: tlačítko, inputbox s tlačítkem (obsahující jako podkomponentu tlačítko) a výpis položek (obsahující jako podkomponenty tlačítka).

vždy lepší než univerzální řešení.
V dnešní době máme na výběr z celé řady technologií, které nám usnadní tvorbu uživatelského rozhraní. Jednou z nejvíce používaných technologií je knihovna Bootstrap, kterou si představíme v následujíc části textu.

Průvodce studiem
Vytvořit si vlastní řešení uživatelského rozhraní webové aplikace (vytvořit design systém) je lepší volba, ale pouze pokud dobře víme co děláme. V takovémto případě se neobejdeme bez grafických dovedností. Pokud jimi nedisponujeme, je vždy lepší (z pohledu uživatele) sáhnout po existujícím řešení. Nevýhodou existujících řešení je univerzálnost. Zjednodušen řečeno, vždy je tam toho víc než potřebujeme.

Knihovna Bootstrap

Knihovna *Bootstrap*[13] slouží pro snadné a rychlé vytváření responsivního webového front-endu založeném na jednoduchém design systému. Hodí se zejména pro tvorbu uživatelského rozhraní s rozsáhlejší funkcionalitou (například administrační rozhraní nebo jednoduché aplikace), ale je možné ji využít i pro tvorbu běžné webové stránky. Univerzálnost knihovny je vykoupena její velikostí.[14]

[13] Volně dostupná pod MIT licencí na stránce `getbootstrap.com`.

[14] Rychlost načítání stránky, která je přímo ovlivněna její velikostí, je klíčovým faktorem udávající kvalitu webové stránky. Jednoduše řečeno, čím menší je velikost zdrojových kódů, tím lépe.

Struktura knihovny

Knihovna obsahuje CSS kód definující vizualizaci základních komponent a JavaScriptový kód, který zabezpečuje běžnou funkcionalitu. Součástí je i externí knihovna *Popper*,[15] která je využívána v případě pokročilejších UI prvků (například dropdown) a knihovna *Bootstrap Icons*,[16] která obsahuje základní ikonografiku. Všechny části je možné používat samostatně. CSS a JavaScriptový kód se dále dělí na menší celky. Podrobně je struktura knihovny ukázána v oficiální dokumentaci.[17]
Obecně je výhodné používat pouze potřebné části. Proto je knihovnu možné sestavit podle potřeby. Pro jednoduchost budeme používat výchozí (celou) knihovnu.

[15] `popper.js.org`

[16] `icons.getbootstrap.com`

[17] `getbootstrap.com/docs/5.3/getting-started/contents`

Průvodce studiem
Knihovnu Bootstrap je možné upravit tak, aby obsahovala jen potřebné části.

Použití knihovny

Knihovnu je možné nainstalovat pomocí balíčkovacích nástrojů. Například.

```
npm install bootstrap@5.3.1
```

Následně je možné ji vložit do webové stránky z lokálního umístění nebo z CDN. Následující kód ukazuje vložení z CDN.[18]

[18] Pro jednoduchost jsme z ukázky odstranily atributy `integrity` a `crossorigin`.

```
<link href="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1
    /dist/css/bootstrap.min.css" rel="stylesheet">

<script src="https://cdn.jsdelivr.net/npm/bootstrap@5.3.1
    /dist/js/bootstrap.bundle.min.js"></script>
```

Obecně je lepší používat knihovnu z lokálního umístění.[19]

[19] Důvodem je opět rychlost a absence závislosti na třetí straně.

Práce s knihovnou

Knihovna obsahuje již definované třídy upravující vzhled a chování mnoha běžných komponent a také prostředky pro snadnou tvorbu layoutu stránky.[20] Knihovna Bootstrap využívá klasických 12-ti sloupcový layout. Příklad použití následuje.

[20] `getbootstrap.com/docs/5.3/layout/grid`

```
<!-- kontejner přes celou šířku -->
<div class="container-fluid">
  <!-- řádek mřížky -->
  <div class="row">
    <!-- první buňka mřížky o velikosti 4/12 -->
    <div class="col-4">
      A
    </div>
    <!-- druhá buňka mřížky o velikosti 8/12 -->
    <div class="col-8">
      B
    </div>
  </div>
</div>
```

Šířky sloupců se upravují automaticky dle velikosti viewportu. Konkrétní breakpointy je možné adresovat pomocí definovaných tříd. Přehled výchozích breakpointů a jim odpovídajících tříd je shrnut v tabulce 1.

zařízení	prefix	velikost
Extra small	`.col-`	$\leq$ 576px
Small	`.col-sm-`	$\geq$ 576px
Medium	`.col-md-`	$\geq$ 768px
Large	`.col-lg-`	$\geq$ 992px
Extra large	`.col-xl-`	$\geq$ 1200px
Extra extra large	`.col-xxl-`	$\geq$ 1400px

Tabulka 1: Přehled výchozích breakpointů a jim odpovídajících tříd knihovny `Bootstrap`.

Jedním z novějších vylepšení knihovny Boostrap jsou utility,[21] které umožňují snadněji vytvářet vlastní komponenty bez nutnosti psaní CSS kódu. Uveďme si jednoduchý příklad.

[21] Utility, neboli CSS pravidla obsahující pouze jednu deklaraci, byly přidáno do knihovny v reakci na velkou popularitu frameworku Tailwind, který je založen na utility třídách.

```
<div class="d-flex p-3 border">
  <div class="border p-2 m-2 flex-fill">Lorem
  <div class="border p-2 m-2 flex-fill">ipsum
  <div class="border p-2 m-2 flex-fill">dolor
</div>
```

Výše uvedený kód vytvoří tři boxy pozicované pomocí flexboxu. Každý box má nastaven `padding`, `margin` a `border`.

Sestavení knihovny

Častou začátečnickou chybou je stažení (výchozí) knihovny a následné přepisování existujícího CSS kódu. Tento způsob práce je velmi neefektivní. Knihovnu je možné snadno přizpůsobit pomocí preprocesoru SASS[22] a CSS proměnných.
Pro sestavení knihovny budeme potřebovat její zdrojové kódy a nástroj `npm`.[23] Poté co stáhneme zdrojové kódy, ve složce, kde je umístěn soubor `package.json`, provedeme instalaci potřebných nástrojů.

[22] Základy práce s preprocesorem SASS jsou shrnuty v příloze tohoto textu.

[23] Zdrojové kódy jsou dostupné na stránce `getbootstrap.com/docs/5.3/getting-started/contribute`, případně je možné je nainstalovat pomocí nástroje `npm`.

```
npm install
```

Nejprve provedeme modifikace knihovny. Předpokládejme, že budeme chtít změnit výchozí kulaté rohy na ostré rohy. Základní konfigurace se nachází v souboru `scss/_variables.scss`. Vypnutí kulatých rohů je možné provést změnou proměnné `$enable-rounded`. V souboru `scss/bootstrap.scss` je možné upřesnit, které části knihovny budou ve finálním sestavení zahrnuty. Následně provedeme samotné sestavení.

```
npm run dist
```

Sestavená knihovna se nachází ve složce `dist/css`.

Výše uvedeným postupem je možné si přizpůsobit celou knihovnu a to včetně definice konkrétních breakpointů, které bývají velmi často nesprávně považovány za jediné platné.[24]

[24] Design určuje breakpointy, ne naopak.

Framework Tailwind

Pro úplnost zmíníme ještě velmi populární framework *Tailwind*.[25] Ten je, stejně jako knihovna Bootstrap, určen pro tvorbu uživatelského rozhraní. Jeho celková filozofie a také použití je ale od knihovny Bootstrap značně odlišné.
Tailwind je založen na utility třídách, neobsahuje žádné komponenty a je určen vývojářům, kteří mají již zkušenosti s tvorbou webového front-endu.[26]
S frameworkem se pracuje následovně. Nejprve je určen design systém. Například vybereme barvy, písmo, mezery a další aspekty, které budou v uživatelském rozhraní. Poté pro určený design systém pomocí Tailwind vygenerujeme sadu utility tříd. Uživatelské rozhraní je následně vytvářeno pouze pomocí těchto tříd.

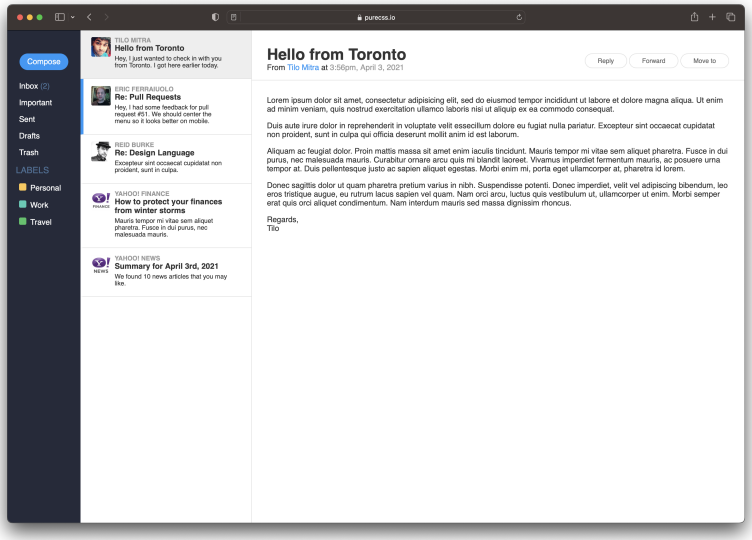
[25] `tailwindcss.com`

[26] To je velký kontrast s knihovnou Bootstrap, u které nemusí vývojář o tvorbě webového front-endu moc vědět. Jednoduše používá připravené komponenty, případně je mírně upravuje. V případě framworku Tailwind si musí komponenty vytvořit sám, nebo použít některou z mnoha existujících, obvykle placených, knihoven.

Závěr

Stručně jsme představili knihovnu Bootstrap a ukázali její základní použití. Knihovna Bootstrap není jedinou knihovnou. Existuje celá řada dalších podobných řešení, například knihovna *Pure*,[27] která se vyznačuje velmi malou velikostí a čistým jednoduchým designem. Předností této knihovny, je její velká modularita, díky které je možné využívat pouze skutečně nezbytné části. Příklad jednoduchého designu pomocí knihovny Pure je ukázán na obrázku 2.

[27] Volně dostupná na stránce `purecss.io`.

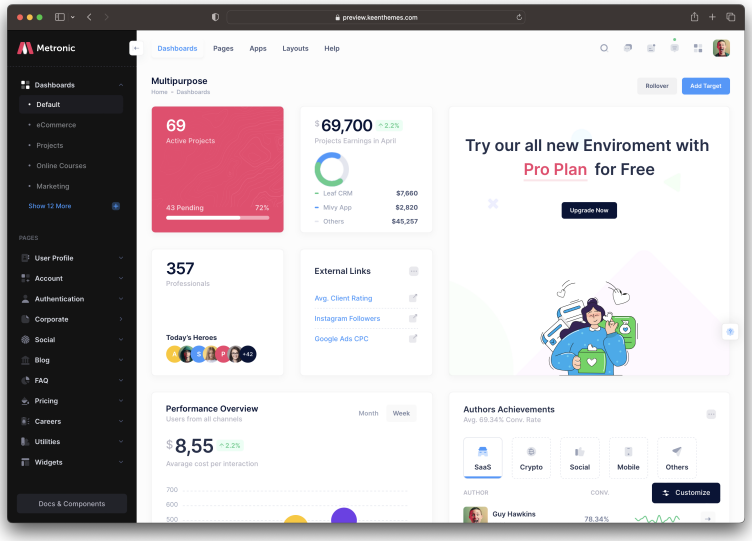


Obrázek 2: Emailová aplikace vizualizovaná pomocí knihovny `Pure`.

Dalším velmi populárním zástupcem je knihovna *Materialize*,[28] která implementuje dnes velmi populární Material desing.
Další možností je implementace vlastního či zakoupení existujícího design systému. Ultimátní je v tomto ohledu placená knihovna *Metronic*, [29] která je určena především pro různá administrační rozhraní. Knihovna obsahuje mnoho variací layoutů komponent a je sestavena pro řadu webových technologií. Příklad jednoduchého designu pomocí knihovny Metronic je ukázán na obrázku 3.

[28] Volně dostupná na stránce `materializecss.com`.

[29] Dostupná na stránce `keenthemes.com/metronic`.



Obrázek 3: Dashboard jedné z mnoha variací knihovny `Metronic`.

Dodejme, že na knihovně Bootstrap je vystavěna celá řada existujících (obvykle placených) řešení. Příkladem je již zmíněná knihovna Materialize nebo *AdminLTE*.[30]

[30] Volně dostupný na stránce `adminlte.io`.

Úkoly

Kontrolní otázky
Odpovězte na následující otázky:

1. K čemu slouží knihovna `Bootstrap`?
2. Jaké jsou alternativy ke knihovně `Bootstrap`?

Úkol 1
Sestavte knihovnu `Bootstrap` tak, aby neobsahovala kulaté rohy a jako primární barvu používala oranžovou (na místo výchozí modré).

Úkol 2
Upravte předdefinovanou třídu pro průhlednost (opacity), aby bylo možné použít třídu `op-33`. Třída `op-33` by měla nastavit průhlednost 33%.

Úkol 3
Z `trnecka.inf.upol.cz/teaching/weba/bootstrap-minimal.zip` si stáhněte jednoduché administrační rozhraní vytvořené pomocí knihovny `Bootstrap` (nepoužívá `Popper`) a otestujte knihovnu z předchozího úkolu.

Úkol 4
Do administračního rozhraní doplňte komponentu pro stránkování (pagination). Na předchozí stránku bude možné se dostat pomocí odkazu s nápisem „Previous“ a na následující stránku pomocí odkazu s nápisem „Next“.

Úkol 5
Upravte komponentu na stránkování tak, aby místo nápisů „Previous“ a „Next“ používala ikony šipek.

Úkol 6
Pomocí knihovny `Bootstrap` vytvořte jednoduchý layout (například formulář) a vyzkoušejte si práci s gridem.

Úkol 7
Podobný layout vytvořte i pomocí jiných knihoven vlastního výběru (například pomocí `PureCSS`, `Materialize` nebo `Tailwind`).

Úkol 8
Projděte si dokumentaci knihovny `Bootstrap`.

Webový server

„I don't care if it works on your machine! We are not shipping your machine!"

Vidiu Platon

Webový server je software, který zprostředkovává webovou stránku klientům, jenž o ni požádají pomocí HTTP požadavku. Existuje celá řada webových serverů. Mezi ty nejznámější patří: *Apache 2*, *IIS*, *nginx* nebo *Node.js*.[31]

[31] Později ukážeme, že Node.js je víc než jen webový server.

Zprovoznit webový server je poměrně snadné, stačí jej nainstalovat a spustit. Pokud chceme na webovém serveru provozovat statické webové stránky, jednoduše umístíme statické soubory (HTML, CSS a JavaScript kód) na určené místo a máme hotovo. V případě, že chceme provozovat dynamické webové stránky, musíme k webovému serveru přidat aplikační server a případně databázový server. V další kapitole tohoto textu si mimo jiné ukážeme tvorbu dynamických webových stránek s využitím jazyka PHP a databáze MySQL. Nyní si ukážeme, jak připravit vše potřebné. Ve spojení s výše uvedenými technologiemi se obvykle využívá webový server Apache 2. Standardně je tato trojice součásti *LAMP balíčků*.[32] Mezi nejznámější LAMP balíčky patří například *WAMP*, určený pro operační systém Windows, *XAMPP*[33] nebo *MAMP* určený pro operační systém macOS. Jednotlivé části LAMP balíčku je možné nainstalovat i samostatně.

[32] Zkratka LAMP je odvozena od zahrnutých technologií: Linux, Apache, MySQL a Python. Dnes se běžně používá pro software, který obsahuje libovolný webový, aplikační a databázový server.

[33] XAMP je multiplatformní a existuje i v portabilní verzi.

Nevýhodou použití LAMP balíčku je, že konfigurace jednotlivých částí je společná pro všechny vyvíjené projekty. Navíc přenesení této konfigurace, nemluvě o problémech s různými verzemi software (či knihoven), na jiný stroj může být velmi obtížné. Lepším způsobem je využít při nasazení webové stránky kontejnerizaci, kterou si stručně ukážeme v následující části.[34]

[34] Připomeňme, že kontejnerizace je odlehčený způsob virtualizace, kdy nevirtualizujeme celý operační systém a v něm běžící aplikace, ale pouze aplikaci samotnou.

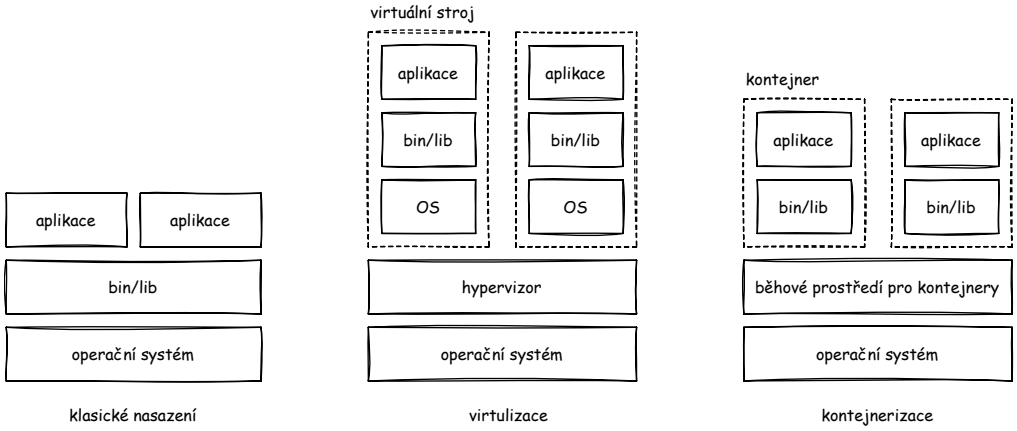
Docker

Docker je dnes běžně používaná (de facto standard) technologie pro kontejnerizaci. *Kontejner* představuje izolované prostředí, ve kterém běží aplikace.[35] Přesněji řečeno v kontejneru běží *(docker) image*,

[35] Ne nutně webová.

který aplikaci obsahuje.[36] Rozdíl mezi klasickým nasazením, virtualizací a kontejnerizací je ilustrován na obrázku 4.

[36] Docker image (nebo také Docker obraz) kromě aplikace samotné obsahuje i další náležitosti potřebné pro její běh.



Obrázek 4: Porovnání klasického nasazení, virtualizace a kontejnerizace. `bin/lib` označuje instalovaný software, knihovny a jejich konfiguraci. Příkladem hypervizoru je třeba nástroj *VirtualBox*, běhové prostředí pro kontejnery zastupuje například zde zmíněný Docker.

V případě klasického nasazení sdílejí aplikace instalovaný software, knihovny a jejich konfiguraci. Přenesení aplikace na jiný hardware by v tomto případě znamenalo opětovnou instalaci a konfiguraci všeho potřebného pro běh aplikace.
V případě virtualizace je aplikace a vše potřebné pro její běh uloženo ve virtuálním stroji. Přesněji řečeno vše je uloženo v image (obrazu) virtuálního stroje, který je následně spuštěn hypervizorem, jenž zprostředkovává komunikaci s operačním systémem.[37] Součástí virtuálního stroje je i operační systém. Jelikož jsou jednotlivé virtuální stroje nezávislé,[38] jsou na sobě nezávislé i aplikace, instalovaný software, knihovny a jejich konfigurace. Problémem je ale velikost. K naší aplikaci je navíc připojen celý operační systém.
V případě kontejnerizace je v kontejneru uloženo vše potřebné pro běh aplikace. Přesněji řečeno vše je uloženo v image a běhové prostředí, kterým je například již zmíněný Docker, zprostředkovává komunikace s operačním systémem.[39] Přenesení aplikace na jiný hardware je v tomto případě pouze přenesení image.[40]

[37] Dodejme, že virtuální stroj je ve skutečnosti běžící image virtuálního stroje.

[38] Velkou výhodou tohoto přístupu je, že aplikace mohou běžet na různých operačních systémech.

[39] Opět dodejme, že kontejner není nic jiného něž běžící image.

[40] Kontejnerizace odstraňuje nevýhody klasického nasazení a virtualizace. Na druhou stranu není možné provozovat aplikace na různých operačních systémech, jelikož ten je v případě kontejnerizace mezi aplikacemi sdílený.

Instalace

Instalace `Docker` je popsána na stránkách `docs.docker.com`. Aplikace *Docker desktop* obsahuje vše potřebné pro práci s kontejnery. Kontrolu instalace je možné provést následujícím příkazem.[41]

[41] V ukázkách se předpokládá verze docker klienta a serveru vetší nebo rovna verzi 20.10.17.

```
docker version
```

Vytvoření image a spuštění kontejneru

Abychom mohli aplikaci v kontejneru spustit, musíme nejprve vytvořit docker image, který bude aplikaci obsahovat. Sestavení image je popsáno v konfiguračním souboru `Dockerfile.yml`, který je zapsán ve formátu YAML.[42] Nejprve si ukážeme ukázkový příklad. Vytvoříme složku `hello-world` a v ní složku `public-html` a soubor `Dockerfile.yml` obsahující následující kód.

[42] Dodejme, že přípona `.yaml` bude fungovat také. Při práci je ale třeba dávat pozor na přípony. Na různých operačních systémech se chování mírně liší. V některých případech je třeba přípony i vynechat.

```
FROM httpd:2.4
```

Příkaz `FROM` určuje základ, označovaný jako *base image*, ze kterého bude vytvořen náš image. Využijeme existující image `httpd`, který je volně dostupný na *Docker hubu*.[43] Za znak dvojtečky zapíšeme verzi, kterou chceme využít. Následně vytvoříme image příkazem.

[43] `Docker hub` je volně dostupný repozitář obsahující předpřipravené image. Repozitář je dostupný na stránce `hub.docker.com`.

```
docker build -t hello-world .
```

Přehled již vytvořených image je možné zobrazit příkazem

```
docker image ls
```

Následně spustíme image v kontejneru příkazem `docker run`.

```
docker run -p 8080:80 -v ./hello-world/public-html/:/usr/
    local/apache2/htdocs/ hello-world
```

Parametr `-p` udává mapování portů. `-p 8080:80` zajistí mapování lokálního portu (portu na našem počítači) `8080` na port kontejneru `80`. Parametr `-v` udává mapování souborů. Ve výše uvedeném příkazu je mapována lokální složka `./hello-world/public-html/` do složky kontejneru `/usr/local/apache2/htdocs/`.[44]
Do lokální složky `public-html` umístíme soubor `index.html`. Webový server, který běží v kontejneru, je nyní dostupný na lokální adrese `http://localhost:8080`. Ukončíme jej pomocí ctrl + c.
Případně je možné spustit kontejner na pozadí pomocí parametru `-d`. Přehled kontejnerů je možné zobrazit pomocí následujícího příkazu.

[44] Mapování se může na různých operačních systémech lišit. Typicky je třeba udávat absolutní cestu. V případě operačního systému Windows je možné psát cestu ve tvaru `C:\path\to\host\folder\`, případně `//c/path/to/host/folder/`.

```
docker container ls
```

Ukončení běhu kontejneru se provádí příkazem `docker stop jméno`, kde `jméno` je název kontejneru z výše uvedeného výpisu.
Pokud bychom chtěli webový server distribuovat i s obsahem složky `public-html`, stačí upravit `Dockerfile.yml` následovně.[45]

[45] Příkaz `COPY` provede překopírování obsahu z lokálního adresáře do adresáře v kontejneru.

```
FROM httpd:2.4
COPY ./public-html/ /usr/local/apache2/htdocs/
```

Poté již stačí vytvořit image a spustit jej následujícím příkazem.[46]

[46] Mapování je uloženo v souboru `Dockerfile.yml`, není třeba jej znovu zadávat parametrem `-v`.

```
docker run -p 8080:80 hello-world
```

Jednou z možností, jak vytvořit image obsahující webový, aplikační a databázový server, je využít existující image, například `ubuntu` obsahující jádro operačního systému a do něj doinstalovat potřebné náležitosti. Následující konfigurační soubor ukazuje instalaci a spuštění webového serveru Apache.

```
FROM ubuntu:latest
RUN apt update
RUN apt -y install apache2

RUN echo "ServerName localhost" >> /etc/apache2/apache2.conf
CMD ["apachectl", "-D", "FOREGROUND"]
```

Nevýhodou uvedeného způsobu je *monolitická architektura*, ve které je vše v jednom kontejneru. Zajímavější alternativou je vytvořit několik vzájemně spolupracujících kontejnerů.

`docker-compose`

Nástroj `docker-compose` umožňuje vytvořit několik vzájemně spolupracujících kontejnerů. Opět je nejprve potřeba vytvořit a nakonfigurovat image, případně využít existující. Postup sestavení všech image je specifikován v souboru `docker-compose.yml`. Výhodou `docker-compose` je možnost specifikovat všechny parametry přímo v konfiguračním souboru a tím zjednodušit spouštění kontejnerů.
Nyní si vytvoříme komplexní prostředí obsahující webový server Apache, podporu jazyka PHP, MySQL databázi a administrační rozhraní pro databázi. Pro databázi a její administraci nám postačí existující nezměněné image. Pro webový server, který bude podporovat PHP a MySQL databázi, musíme modifikovat výchozí image. Následuje obsah `docker-compose.yml`.

```
version: "3.9"
services:

  # Apache a PHP
  www:
    build: ./docker/php-apache # cesta k Dockerfile
    volumes:
```

```
      - ./src:/var/www/html/
    ports:
      - 8080:80


  # MySQL databáze
  db:
    image: mysql:latest
    restart: always
    environment:
      MYSQL_ROOT_PASSWORD: toor # nastavení hesla pro už
          ivatele root

  # administrace databáze
  adminer:
    image: adminer:latest
    restart: always
    ports:
      - 8081:8080
```

`Dockerfile.yml` umístěný ve složce `./docker/php-apache` vypadá následovně.

```
FROM php:8.1-apache
RUN docker-php-ext-install mysqli && docker-php-ext-enable
    mysqli
```

Image sestavíme a spustíme příkazy.

```
#sestavení
docker-compose build

#spuštění
docker-compose up
```

Případně je možné obojí spustit současně příkazem.

```
docker-compose up --build
```

Po spuštění běží webový server na portu `8080` a administrační rozhraní na portu `8081`. Výše popsané konfigurační soubory a adresářová struktura je předpřipravena v souboru `trnecka.inf.upol.cz/teaching/weba/docker-web-server.zip`.

Závěr

Možnosti kontejnerizace jsou daleko nad rámec tohoto kurzu. Ukázali jsme pouze základní spuštění běhového prostředí pro klasickou PHP/MySQL aplikaci, kterým se budeme v příštích seminářích věnovat. Kontejnerizace mimo jiné usnadňuje škálovatelnost a průběžnou integraci (continuous integration). V nadcházející kapitole se podíváme na PHP a MySQL a ukážeme jak pomocí nich vytvořit webovou aplikaci.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jak je možné nasadit webovou aplikaci a jaké jsou výhody a nevýhody jednotlivých přístupů?
2. Co je to `Docker`?

Úkoly

Úkol 9
Spusťte nebo nainstalujte některý z LAMP balíčků. Vytvořte HTML soubor a spusťte jej na webové serveru.

Úkol 10
Zprovozněte webový server `Apache`, s podporou PHP a MySQL v Docker kontejnerech.

Úkol 11
Podívejte se na docker hub a projděte si images, které zde jsou k dispozici.

Úkol 12
Projděte si dokumentaci k `Docker`.

Úkol 13
Podívejte se na alternativy k `Dockeru`, například `Podman`, `LXD`.

PHP a MySQL

„PHP is about as exciting as your toothbrush. You use it every day, it does the job, it is a simple tool, so what? Who would want to read about toothbrushes?“

Rasmus Lerdorf

V následujících kapitolách si postupně ukážeme tvorbu jednoduchého webového back-endu. Ten je v dnešní době možné vytvořit pomocí mnoha různorodých technologií jako například PHP, JavaScript, C#, Python nebo Go. V této kapitole se budeme věnovat technologii PHP. Budeme předpokládat základní znalost programování, stručně si představíme syntaxi jazyka PHP a zaměříme se především na záležitosti specifické pro PHP a vývoj webového back-endu.

Průvodce studiem
Nabízí se otázka: Proč vybíráme právě PHP? Volba technologie není v tento okamžik až tak zásadní. Půjde nám především o princip tvorby, který je ve většině případů stejný. Dodejme, že některé ze zmíněných technologií jsou diskutovány v jiných kurzech.

Jazyk PHP

Jazyk PHP[47] je zástupce skriptovacích jazyků, který se využívá zejména na straně serveru. Obecně je přirozeným požadavkem na tento typ technologie snadná práce s textem, jelikož jejím primárním úkolem je vygenerovat webovou stránku, která není nic jiného než textový soubor obsahující HTML, CSS a JavaScriptový kód.
Aktuálně se jazyk PHP nachází ve verzi 8.[48] PHP vytvořil v roce 1995 Rasmus Lerdorf pro tvorbu dynamických webových stránek jako náhradu za komplikované CGI skripty. Jazyk se brzy stal velmi populární a dnes patří k nejběžnější technologii pro tvorbu webového back-endu.[49]
Zdrojový kód jazyka PHP je obvykle zapsán v textových souborech s příponou `.php`.[50] Jednotlivé příkazy se ukončují středníkem a vždy se zapisují mezi `<?php` a `?>`. Pro jednoduchost budeme tuto část kódu v některých příkladech vypouštět. PHP je možné používat i jako součást HTML kódu. Následující dva kódy vygenerují stejný výstup.

[47] PHP je rekurzivní zkratka pro PHP: Hypertext Preprocessor.

[48] V době psaní tohoto textu je již k dispozici verze 8.3.

[49] Toto je běžně známá informace, ale jako taková je dosti zavádějící. Vysoké procento zastoupení PHP na webovém back-endu je dáno zejména tím, že v jazyce PHP jsou naprogramovány nejběžnější CMS systémy jako WordPress, Joomla! nebo Drupal.

[50] Konkrétní přípona závisí na konfiguraci webového serveru.

```
<div>
  <?php echo "Hello world!"; ?>
```

```
</div>
```

```
<?php
  echo "<div>";
  echo "Hello world";
  echo "</div>";
?>
```

V prvním kódu je PHP použito v HTML kódu. V druhém, je HTML kód součástí PHP kódu. Ve výše uvedených ukázkách jsme použili základní příkaz `echo`.[51] Ten slouží pro textový výpis. Příkaz `echo` akceptuje více textových řetězců. Můžeme tedy psát.

```
echo "<div>", "Hello world", "</div>";
```

Příkaz `echo` je možné zapisovat i jako volání funkce.[52]

```
echo("<div>Hello world</div>");
```

Výše uvedený zápis umožňuje vypsat pouze jeden textový řetězec. Kromě příkazu `echo` je k dispozici ještě příkaz `print`, který akceptuje pouze jeden textový řetězec a vždy vrací hodnotu `1`.[53] V některých situacích je užitečný víceřádkový výpis.[54]

```
echo <<<KONEC
<div>
Hello world
</div>
KONEC;
```

Soubory obsahující zdrojový kód je možné vkládat do jiných pomocí příkazu `include`. Tím lze vygenerovat, v závislosti na podmínkách, různé stránky a nevytvářet zbytečně redundantní kód. Předpokládejme, že webová stránka obsahuje hlavičku a menu, které jsou na všech stránkách stejné a mění se pouze obsah. Následující kód ukazuje nástin řešení.

```
<?php
  // jedná se pouze o ilustraci řešení, kód není funkční
  include "header.php"; // hlavička stránky
  include "menu.php"; // menu stránky

  if ($URL == 'podstranka-1') {
    include "podstranka-1";
  }
```

Průvodce studiem
Jazyk PHP je velmi často programátory zatracován a to zejména kvůli bolestivému vývoji. Ten byl poznamenán zejména komplikovaným přechodem z verze 4 na verzi 5, absencí verze 6, která nebyla kvůli technickým komplikacím dokončena a nepříliš inovativní verzi 7. Verze 8 je ale v mnoha ohledech revoluční a jazyku PHP doslova vdechla nový život.

[51] Úkolem PHP kódu není nic jiného než vypsat požadovaný výstup.

[52] `echo` není funkce, ale konstrukt jazyka. Jedná se pouze o syntaktický cukr.

[53] Příkaz `echo` nevrací žádnou hodnotu.

[54] Jedná se o *heredoc* syntaxi. Výskyt proměnných je nahrazen jejich hodnotou. Pokud chceme chování jako v případě použití apostrofů (viz dále) využívá se *nowdoc* syntaxe, kdy je `KONEC` nahrazen `'KONEC'`. Dodejme, že `KONEC` může být libovolný řetězec bez mezer, čísel a speciálních znaků.

Průvodce studiem
Idea zachycená v ukázce je velmi důležitá. PHP nám umožňuje konstruovat stránku dle zadaných podmínek. Díky tomu je možné vyhnout se opakujícímu kódu. Uvažme například výpis produktů v e-shopu. Stačí naprogramovat pouze jeden výpis a ten opakovaně použít pro různé kategorie produktů.

```
  elseif ($URL == 'podstranka-2') {
    include "podstranka-2";
  }
?>
```

Rychlokurz jazyka PHP

V následující části textu v rychlosti představíme syntaxi jazyka PHP. Jelikož předpokládáme znalost programování, je tato část velmi stručná.[55]

[55] Podrobně je jazyk PHP dokumentován na stránce `www.php.net`.

Proměnné

Jazyk PHP je dynamicky typovaný jazyk. Umožňuje ale vynutit typovou kontrolu parametrů funkcí a metod tříd, což je velmi užitečné na větších projektech.[56]
Proměnné se zapisují s prefixem `$`. Informaci o proměnné můžeme získat pomocí funkce `var_dump()`.

[56] Typová kontrola je podrobněji popsána v dokumentaci jazyka. Z časových důvodů se tomuto tématu věnovat nebudeme, je ale velmi užitečné jej znát.

```
$x; // deklarace proměnné
var_dump($x); // warning, nedefinovaná proměnná

$x = 42; // definice proměnné, číslo
var_dump($x); // vypíše int(42)

$x = "Hello"; // definice proměnné, řetězec
var_dump($x); // vypíše string(5) "Hello"
```

Řetězce

V jazyce PHP je možné řetězce zapisovat pomocí uvozovek a apostrofů. V případě, že jsou použity uvozovky, dochází k detekci proměnných v daném textovém řetězci a k jejich automatické náhradě za hodnotu detekované proměnné.[57] Například následující kód vypíše textový řetězec: `Obsah proměnné x je 42.`.

[57] To mimo jiné znamená, že textové řetězce zapsané uvozovkami se zpracovávají pomaleji. Interpret jazyka PHP musí nejprve vyhledat výskyt proměnných a to i v případě, že se zde žádné nenacházejí. Z tohoto důvodu je lepší používat uvozovky pouze v případě, že má být proměnná nahrazena za svoji hodnotu.

```
$x = 42;
echo "Obsah proměnné x je $x.\n";
```

Pokud použijeme apostrofy na místo uvozovek,

```
$x = 42;
echo 'Obsah proměnné x je $x.\n';
```

bude výstupem textový řetězec: `Obsah proměnné x je $x.`.[58]
Spojování řetězců se provádí pomocí operátoru `.` (tečka).

[58] V obou případech představuje \n standardní escape sekvenci.

```
// spojení řetězců
echo "Hello" . " " . "world\n";
```

Pole

Pole se v jazyce PHP vytváří pomocí funkce `array()`, případně pomocí symbolů `[` a `]`.

```
// pole
$y = array(1, 2, 3, 4, 5);

// jiný zápis pole
$y = [1, 2, 3, 4, 5];
```

Výše uvedené vytváří asociativní pole indexované čísly od 0. Asociativní pole s vlastním indexování se vytváří následovně.

```
// asociativní pole
$y = array(5=>1, 4=>2, 3=>3, 2=>4, 1=>5);

// jiný zápis asociativního pole
$y = [5=>1, 4=>2, 3=>3, 2=>4, 1=>5];
```

Práce s polem a jeho prvky je v PHP shodná s jinými programovacími jazyky. Užitečnou konstrukcí je automatické přidání na konec pole.

```
// přidání na konec pole
$z[] = 42; // vytvoří pole a přidá hodnotu
$z[] = "Hello";
$z[] = array(1, 2, 3);
```

Přístup k prvkům pole se provádí pomocí indexačního operátoru `[]`.

```
echo $z[0]; // 42
echo $z[2][2]; // 3
```

Pro přehledný výpis složitějších datových struktur, jako jsou například pole, vícerozměrné pole nebo objekty, se v PHP používá funkce `print_r()`. Pro výše uvedený kód volání funkce `print_r($z)` vypíše následující výstup.[59]

```
Array
(
    [0] => 42
    [1] => Hello
```

[59] Dodejme, že aby byl výstup formátovaný jako v ukázce, je třeba zobrazit zdrojový kód stránky. Pro formátování jsou použity znaky, které webový prohlížeč při renderování stránky ignoruje.

```
    [2] => Array
        (
            [0] => 1
            [1] => 2
            [2] => 3
        )
)
```

Pro úplnost upozorněme na poněkud nelogické použití indexačního operátoru v textovém řetězci zapsaném pomocí uvozovek.[60]

[60] Nelogičnost je pouze při prvním pohledu. Při druhém by již vše mělo být jasné.

```
$p = ["foo"=>1, 'bar'=>2];
echo $p['foo'];
echo $p["foo"];

// echo $p[foo] nefunguje
//echo "$p['foo']"; // nefunguje

echo "$p[foo]"; // zde pozor
```

Objektově orientované programování

Jazyk PHP umožňuje zapisovat třídy pomocí klíčového slova `class`. Následuje příklad třídy reprezentující bod, vytvoření instance této třídy a volání metody.

```
// definice třídy
class Bod {
  private $x = null;
  private $y= null;

  // konstruktor
  function __construct($x, $y) {
    $this->x = $x;
    $this->y = $y;
  }

  public function getX() {
    return $this->x;
  }

  public function getY() {
    return $this->y;
  }
}

// vytvoření instance třídy
```

```
$x1 = new Bod(1,2);
print_r($x1);

//volání metody objektu
echo $x1->getX();
```

Dodejme, že při tvorbě back-endu webové aplikace se objektově orientovaný přístup uplatňuje velmi často. Smyl má zejména u větších aplikací a informačních systémů, které pracují s mnoha entitami.[61]

[61] Typicky do této kategorie spadají e-shopy nebo systémy evidující osoby.

Konstrukce pro řízení běhu programu

Jazyk PHP disponuje většinou běžně známých konstrukcí pro řízení běhu programu.

Větvení programu

Základní příkazy pro větvení programu jsou `if`, `elseif` a `else`. Jejich použití je stejné jako v jiných programovacích jazycích.[62]

[62] Analogicky jako v jiných jazycích je možné `elseif` a `else` vynechat. Stejně tak je možné vynechat `{` a `}` pokud daná větev obsahuje pouze jediný příkaz.

```
$a = 42;
$b = 2;

if ($a > $b) {
  echo "a is bigger than b";
} elseif ($a == $b) {
  echo "a is equal to b";
} else {
  echo "a is smaller than b";
}
```

Dalším příkazem pro větvení programu je příkaz `switch`.

```
$i = 0;

switch ($i) {
  case 0:
    echo "i equals 0";
    break;
  case 1:
    echo "i equals 1";
    break;
  case 2:
    echo "i equals 2";
    break;
  default:
    echo "i not equals 0, 1, 2";
}
```

Jazyk PHP disponuje i příkazem `match`.[63]

[63] Jedná se o poměrně nový příkaz pro pattern matching, který se začíná objevovat v celé řadě programovacích jazyků.

```
$food = 'cake';

$return_value = match($food) {
  'apple' => 'This food is an apple',
  'bar' => 'This food is a bar',
  'cake' => 'This food is a cake',
  1 => 'This food is a number one',
};

echo $return_value;
```

Cykly

Základní příkazy umožňující opakovat část programu jsou `for`, `while` a `do-while`. Jejich použití je stejné jako v jiných programovacích jazycích.[64]

[64] Všechny uvedené cykly vypíší hodnoty 1 až 10.

```
// cyklus for
for ($i = 1; $i <= 10; $i++) {
  echo $i;
}

// cyklus while
$i = 1;
while ($i <= 10) {
  echo $i++;
}

// cyklus do-while
$i = 1;
do {
  echo $i;
} while ($i <= 10);
```

Velmi často potřebujeme iterovat přes prvky pole či jiné iterovatelné struktury. V PHP je možné využít, kromě výše uvedených, i cyklus `foreach`.

```
// foreach cyklus
$arr = array(1, 2, 3, 4, 5, 6, 7, 8, 9, 10);

foreach ($arr as $item) {
  echo $item;
}
```

Dodejme, že v PHP je možné použít klasická klíčová slova `break` (pro zastavení cyklu) a `continue` (pro pokračování další iterací cyklu).

Funkce

Funkce v jazyce PHP kombinují řadu přístupů, které můžeme znát z různých programovacích jazyků. Základním příkazy jsou `function` pro definici funkce a `return` pro vrácení návratové hodnoty.

```
// funkce f1 se dvěma argumenty
function f1($arg_1, $arg_2) {
  return $arg_1 + $arg_2;
}

f1(42, 1); // volání funkce
```

Parametrům funkce je možné zadat výchozí hodnotu tak, jak je ukázáno v kódu níže.

```
// funkce s výchozí hodnotou parametru arg_2
function f2($arg_1, $arg_2 = 100) {
  return $arg_1 + $arg_2;
}

echo f2(42, 1); // vypíše 43
echo f2(42); // vypíše 143
```

Samozřejmostí jsou funkce vyšší řádů a anonymní funkce.

```
$arr = [1, 2, 3, 4, 5];

// funkce f3 akceptuje jako argument funkci
function f3($array, $g) {
  foreach($array as $item) {
    echo $g($item);
  }
}

// použití, jako parametr je použita anonymní funkce
f3($arr, function ($x) {return $x * $x;});
```

Anonymní funkce je možné zapisovat pomocí a͡rrow operátoru.

```
f3($arr, fn ($x) => $x * $x);
```

Pro úplnost dodejme, že následující běžná konstrukce je možná i v PHP.

```
$g = function ($x) {return $x * $x;};
echo $g(42);
```

Funkce mohou být i vnořené. Zde je třeba si dávat pozor, jelikož při volání funkce dochází k její automatické definici v globálním prostředí. Uvažme následující funkci.

```
function f4() {
  function f5() {
    echo "Hello from f5()";
  }

  f5();
}
```

Funkce `f5()` je vnořená funkce ve funkci `f4()`. Pokud zavoláme funkci `f5()` v globálním rozsahu, funkce není definována. Pokud ale nejprve zavoláme funkci `f4()`, dojde k definici `f5()` a tu již nyní můžeme zavolat.
Jazyk PHP disponuje variabilními funkcemi.[65]

[65] Variabilní funkce jsou velmi silný koncept, který umožňuje spouštět funkci, jejíž jméno je uloženo v hodnotě proměnné.

```
$variable = "f1";
echo $variable(42, 1); // zde dochází k volání funkce f1()
```

V jazyce PHP je možné pojmenovávat argumenty.[66] Tento způsob je velmi výhodný, jelikož vytváří samodokumentující kód.

[66] Analogicky jako například v jazyce Python.

```
// pojmenované argumenty
function print_name($name = "", $surname = "", $title = "") {
  echo "$title $name $surname";
}

print_name(title: "Dr.", surname: "Carter");
```

Kromě uživatelských funkcí disponuje jazyk PHP řadou vestavěných funkcí. Velmi užitečné jsou funkce `explode()` a `implode()`. Funkce `explode()` rozděluje řetězec do prvků pole dle zadaného oddělovače. Funkce `implode()` spojuje řetězec z prvků pole, přičemž přidává zadaný oddělovač. Příklad použití těchto funkcí následuje.

Průvodce studiem
Standardní knihovna jazyka PHP je velmi bohatá a obsahuje i poměrně pokročilé funkce. Je výhodné se s ní seznámit.

```
$URL = "USER/EDIT/42";

// rozdělí řetězec v proměnné URL a uloží do pole
$URL_explode = explode('/', $URL);
```

```
print_r($URL_explode);
```

Výše uvedený kód vypíše.

```
Array
(
    [0] => USER
    [1] => EDIT
    [2] => 42
)
```

Analogicky následující příkaz vypíše textový řetězec `USER, EDIT, 42`.

```
// spojí prvky pole do řetězce a vypíše jej
echo implode(', ', $URL_explode);
```

Napojení na webový server

Data (informace od uživatele) jsou mezi interpretem jazyka PHP a webovým serverem předávány automaticky pomocí *superglobálních polí* jejichž přehled a popis je uveden v tabulce 2.

Tabulka 2: Přehled superglobálních polí v jazyce PHP.

pole	popis
`$_GET`	předává data poslaná `GET` metodou HTTP
`$_POST`	předává data poslaná `POST` metodou HTTP
`$_SESSION`	předávaná data v rámci session[67]
`$_FILES`	předává datové soubory předané webovému serveru přes `POST` metodu HTTP
`$_COOKIE`	předává data poslaná pomocí `COOKIES` HTTP
`$GLOBALS`	globální proměnné (dostupné ve všech skriptech)
`$_SERVER`	informace o skriptu, který je spouštěn a parametry se kterými byl volán (parametry předané webovému serveru)
`$_REQUEST`	obsah `$_GET`, `$_POST` and `$_COOKIE`
`$_ENV`	informace předené operačním systém, který spouští interpret jazyka PHP

[67] Připomeňme, že session je zahájena připojením k webovému serveru a ukončena zavřením prohlížeče nebo vypršením časového intervalu.

V následující části se blíže podíváme na práci s dvěma nejdůležitějšími poli `$_GET` a `$_POST`, které obsahují informace zaslané uživatelem.

Pole `$_GET` a `$_POST`

Superglobální pole můžeme vypsat pomocí funkce `print_r()`, kterou jsme již několikrát použili. Například.

```
// uvažujme volání skriptu s parametry
// ?name=Samantha&surname=Carter
print_r($_GET);
```

Výše uvedený kód vypíše.

```
Array
(
    [name] => Samantha
    [surname] => Carter
)
```

Velmi častou chybou je čtení nedefinovaných proměnný. Například následující kód nefunguje, pokud mu nejsou předány dané proměnné.[68]

[68] Přesněji řečeno je zobrazen warning, ale běh skriptu není zastaven.

```
echo "Hello " . $_GET['name'] . " " . $_GET['surname'];
```

Uvedený příklad je navíc velmi nepřehledný. Obecně se nedoporučuje používat superglobální pole přímo v kódu, ale nejprve si je zpracovat do pomocných proměnný. Následující kód je přehlednější, ale stále neošetřuje zmíněné chyby.

```
// přehlednější, ale chyby stále nejsou ošetřeny
$name = $_GET['name'];
$surname = $_GET['surname'];

echo "Hello $name $surname";
```

Ošetření čtení nedefinované proměnné je možné provést starším a dnes již nepoužívaným způsobem pomocí funkce `isset()`.[69]

[69] Tento způsob si uvádíme pouze proto, že se s nim stále můžeme setkat a to především ve starší literatuře

```
// starý způsob pomocí isset (dnes je lepší již nepoužívat)
$name = isset($_GET['name']) ? $_GET['name'] : '';
$surname = isset($_GET['surname']) ? $_GET['surname'] : '';

echo "Hello $name $surname";
```

Od PHP verze 8 je možné k ošetření čtení nedefinované proměnné použít operátor `??`, který celý zápis výrazně zpřehledňuje. Příklad následuje.[70]

[70] Proměnná `$name` bude nabývat hodnoty `$_GET['name']`, pokud bude tato proměnná inicializována, jinak bude obsahovat prázdný řetězec. Analogické platí pro proměnnou `$surname`.

```php
<?php
// modernější a preferovaný způsob
$name = $_GET['name'] ?? '';
$surname = $_GET['surname'] ?? '';

echo "Hello $name $surname";
?>
```

Analogicky můžeme pracovat s polem `$_POST`. V tomto případě můžeme data poslat pomocí formuláře, nebo jednodušeji pomocí nástroje *Postman*.[71] Jednoduchý formulář odesílající data pomocí metody `POST` je ukázán níže.

[71] Nástroj je volně dostupný na stránce `www.postman.com`. Dodejme, že existuje celá řada alternativy například *Testfully* nebo *Hoppscotch*.

```html
<form action="skript.php" method="post">
  <input type="text" name="name">
  <button type="submit">submit</button>
</form>
```

Klíčový je atribut `action` udávající adresu skriptu, kterému mají být data z formuláře předána. Atribut `method` určuje metodu HTTP. Výchozí je metoda `GET`.

Session

Data v session jsou uchovávána webovým serverem po dobu, po kterou je otevřeno okno prohlížeče. Data tedy přežijí opětovné načtení stránky. Aby bylo možné s polem `$_SESSION` začít pracovat, je nejprve nutné jej zpřístupnit pomocí funkce `session_start()`.[72] Postup práce se `$_SESSION` je ukázán v komentovaném kódu níže.

[72] Funkci `session_start()` je třeba volat před jakýmkoliv výstupem (například výpis pomocí `echo`).

```php
// zpřístupnění session
session_start();

// uložení do session
$_SESSION["variable"] = 42;

// smazání proměnné v session
unset($_SESSION["variable"]);

// smazání celé session
session_destroy();
```

`modrewrite`

Skripty jsme doposud spouštěli zadáním jejich URL v okně prohlížeče, případně voláním skrze jinou aplikaci. Webový server umožňuje automatické přepsaní zadané URL adresy do požadovaného

formátů. Tuto funkcionalitu zajišťuje modul `modrewrite` a slouží pro vytvářené pěkných URL.[73] Například URL

`produkty.php?kategorie = notebooky`

je možné zpřístupnit pomocí moderewrite přes URL

`/produkty/notebooky`

Dodejme, že bez `modrewrite` by `/produkty/notebooky` vedla do složky na webovém serveru se stejnou adresou.
V závislosti na konfiguraci webového serveru je třeba `modrewrite` zapnout. Postup se na různých webových serverech liší. V případě webového serveru `Apache2` je třeba upravit soubor `httpd.conf`. Konkrétně je nutné odkomentovat řádek

```
LoadModule rewrite\_module modules/mod\_rewrite.so
```

a změnit `AllowOverride None` na `AllowOverride All`.
Samotná přepisovací pravidla, která řídí přepis jedné URL na druhou jsou zapsána v souboru `.htaccess`.[74] Například pravidla

```
RewriteEngine On
RewriteRule ^([^.]*)$ index.php?pozadavek=$1 [L,QSA]
```

přepíší vše za prvním lomítkem URL do parametru `pozadavek`. Fungování snadno vyzkoušíme pomocí následujícího kódu.

```
echo "index.php\n";
echo $_SERVER['HTTP_HOST'] . "\n";
echo $_SERVER['REQUEST_URI'] . "\n";

print_r($_GET);
```

[73] Můžeme se také setkat s označením SEO URL.

[74] Pozor na některých operačních systémem jsou soubory s prefixem . (tečka) ve výchozím stavu skryty.

MySQL

Jazyk PHP podporuje práci s celou řadou různých databází. Nejběžněji se, zejména v případě menších projektů, využívají MySQL nebo MariaDB databáze. Přestože jsou mezi MySQL a MariaDB rozdíly, mají mnoho společného. Pro jednoduchost se omezíme pouze na MySQL databázi.

Průvodce studiem
Historie stojící za MySQL a MariaDB databází je poměrně bohatá. MariaDB vznikla jako reakce na odkoupení společnosti vyvíjející MySQL společností Oracle. Přestože se obavy, že jedna z nejvíce používaných open-source databází ve webovém prostředí bude zpoplatněna či uzavřena nevyplnili, MariaDB si našla mnoho příznivců a je dnes velmi populární alternativou k MySQL.

Správa databáze

Databázi je možné spravovat přímo prostřednictvím příkazové řádky nebo klienta (pokud je k dispozici). Při použití ve webovém pro-

středí se často využívají jednoduché administrační nástroje jako například *phpMyAdmin* nebo velmi populární český projekt *Adminer*, které jsou oba napsány v jazyce PHP.[75] Rovněž si správu databáze můžeme naprogramovat sami, ale typicky se struktura databáze vytváří při vývoji a po nasazení se příliš nemění, tedy strávíme neúměrné množství času implementací funkcionality, která se provede pouze jednou.

[75] Nástroje jsou dostupné na stránce `www.phpmyadmin.net` a `www.adminer.org`. Samozřejmě existují i mnohé další.

Propojení PHP a MySQL databáze

Pokud využíváme LAMP balíček, máme již MySQL databázi nainstalovanou a její podpora je v PHP aktivní. V jiných případech je obvykle nutné databázi nainstalovat a aktivovat `mysqli` rozšíření.[76] Rozšíření se spravují v souboru `php.ini`.
Následuje komentovaný příklad připojení k existující MySQL databázi v jazyce PHP.[77]

[76] Případně je možné využít i jiné rozšíření, například `PDO`.

[77] V příkladu je ukázáno v komentářích i připojení k databázi, která neběží lokálně. Konkrétně se jedná o řádky 4, 8, a 16.

```
1  <?php
2    // připojení k lokální databázi (UNIX socket)
3    $db_host = 'localhost';
4    // $db_host = '127.0.0.1'; // pro TCP/IP spojení
5    $db_user = 'root';
6    $db_password = 'root';
7    $db_db = 'test';
8    //$db_port = 8889; // pro TCP/IP spojení
9
10   // vytvoření objektu zodpovědného za přístup k DB
11   $mysqli = new mysqli(
12     $db_host,
13     $db_user,
14     $db_password,
15     $db_db,
16     //$db_port // pro TCP/IP spojení
17   );
18
19   // test zda bylo spojení úspěšné
20   if ($mysqli->connect_error) {
21     echo 'Error: '.$mysqli->connect_error;
22     exit();
23   }
24
25   echo 'Success: A proper connection to MySQL was made.';
26   echo 'Host information: '.$mysqli->host_info;
27   echo 'Protocol version: '.$mysqli->protocol_version;
28
29   $mysqli->close();
30 ?>
```

Výše uvedený kód ukazuje objektový přístup, kdy je pro práci s da-

tabází vytvořen objekt třídy `mysqli` (řádky 11–17). `mysqli` rozšíření umožňuje pracovat s databází i čistě procedurálně, tento způsob je ale dnes značně na ústupu.[78]
K ukončení spojení s databází dochází automaticky po skončení běhu skriptu. Obecně je ale lepší spojení ukončit v okamžiku kdy jej již nadále nepotřebujeme.[79] K ukončení spojení slouží metoda `close()`.

[78] Rozšíření PDO umožňuje pouze objektový přístup.

[79] V případě chyby by mohlo zůstat spojení otevřeno.

Provedení SQL dotazu

K provedení SQL dotazu nad databází slouží metoda `query()`. Následující kód ukazuje provedení dotazu a zpracování výsledků dotazu.[80]

[80] V ukázce předpokládáme, že jsme již k databázi připojeni a databáze obsahuje tabulku `users`, která má atributy `id`, `name` a `surname`.

```
// provedení dotazu
$query_result = $mysqli->query("SELECT * FROM users;");

// výpis výsledků
while ($row = $query_result->fetch_assoc()) {
  echo $row['id'], $row['name'], $row['surname'];
}
```

Po provedení dotazu obsahuje proměnná `$query_result` instanci třídy `mysqli_result`, obsahující vrácená data (řádky tabulky), přes které je možné iterovat. K iterování slouží různé `fetch-*` metody. V ukázce je použita metoda `fetch_assoc()`, která vždy přečte jeden řádek z výsledných dat a převede jej na asociativní pole.[81] Analogicky je možné použít metodu `fetch_row()`, která výsledky převádí na asociativní pole indexované čísly nebo `fetch_object()`, která výsledky převádí na objekty. Rovněž je možné použít metodu `fetch_all()`, která vrací všechny výsledky v jednom poli.

[81] Klíče odpovídají atributům.

Průvodce studiem
Metod pro získání dat z třídy `mysqli_result` je celá řada. Je dobré se s nimi seznámit.

Ošetření vstupů

Při konstrukci SQL dotazu by nikdy neměly být používány neošetřené vstupy. Následující kód obsahuje závažnou bezpečnostní chybu.

```
$name = $_POST['name'] ?? null;
$surname = $_POST['surname'] ?? null;

if($name && $surname) {
  $mysqli->query("INSERT INTO users (name, surname) VALUES ('$name', '$surname');");
}
```

Pokud by potencionální útočník předal v proměnné `$_POST['name']` řetězec `1' OR '1=1)`, změnil by příjmení všem uživatelům v tabulce `users`.[82]
Ošetření vstupů lze provést pomocí metody `real_escape_string()`, která nahrazuje speciální znaky jejich entitami.

[82] Tento typ útoku je znám jako SQL injection.

```
// bezpečné řešení
$name = $mysqli->real_escape_string($_POST['name']) ?? null;
$surname = $mysqli->real_escape_string($_POST['surname']) ?? null;
```

Případně je možné využít metodu `prepare()`.

```
$name = $_POST['name'] ?? null;
$surname = $_POST['surname'] ?? null;

if($name && $surname) {
  $query = $mysqli->prepare("INSERT INTO users (name,
      surname, date) VALUES ('?', '?', now());");
  $query->bind_param("ss", $name, $surname);
  $query->execute();
}
```

Metoda `prepare()` vytváří šablonu dotazu, kterou je pak možné opakovaně provést s různými parametry.[83]

[83] Tento postup přípravy a provedení dotazu, je k dispozici v mnoha databázových systémech. Typicky databázový systém může provést různé optimalizace, aby byl dotaz proveden rychleji.

Shrnutí

V této kapitole jsme představili jazyk PHP, ukázali jeho napojení na webový server a propojení s MySQL databází. Nyní máme vše potřebné pro tvorbu naší první webové aplikace.
Pro úplnost dodejme, že v případě webových aplikací se můžeme velmi často setkat i s nerelačními databázemi. Volba vhodné databáze je součástí vývoje webové aplikace.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jak jsou předávány hodnoty mezi klientem a interpretem jazyka PHP?
2. Co je to `moderewrite`?
3. Jak je prováděn dotaz nad MySQL databází pomocí jazyka PHP?

Úkoly

Úkol 14
Upravte administrační rozhraní z předchozí kapitoly tak, aby bylo dynamicky generované. Vytvořte jednotlivé podstránky: `dashboard` (výchozí), `items`, `others` a `users`. URL odpovídá názvům podstránek. V navigační liště zvýrazněte, která stránka je aktivní (dle zadaného URL). Nezapomeňte ošetřit vložení nee-

xistující stránky.

Úkol 15
Implementujte přihlášení uživatele (bez autentizace). Vytvořte přihlašovací obrazovku, která se zobrazí nepřihlášenému uživateli. Po odeslání (jakýchkoliv dat) dojte k přihlášení uživatele a zobrazení administrace. Informaci o tom, že je uživatel přihlášen a vyplněné uživatelské jméno uložte v `$_SESSION`. Na stránce `users` vypište uživatelské jméno. Implementujte také odhlášení uživatele.

Úkol 16
Rozšiřte administrační rozhraní o správu uživatelů. Je tedy třeba: navrhnout vhodnou strukturu databáze; v sekci `users` implementovat: výpis existujících uživatelů, přidání nového uživatele, editaci a smazání stávajících uživatelů;[84] přehled (výpis) posledních 10 přihlášených uživatelů v sekci dashboard. Uživatelé budou mít: jméno, příjmení, e-mail (slouží jako login), telefon, pracovnu (krátký text), popisek (delší text), heslo a příznak, zda jsou či nejsou správci. Pouze správci budou smět libovolně měnit uživatele

[84] Výpis implementujte pomocí tabulky. Na konci každého řádku, budou tlačítka s příslušnými operacemi.

Úkol 17
Rozšiřte administrační rozhraní o autentizaci. Je tedy třeba upravit přihlášení tak, aby ověřovalo přihlašovací údaje oproti databázi uživatelů.

Úkol 18
K výpisu všech uživatelů přidejte stránkování.

Úkol 19
Rozšiřte funkcionalitu tak, že přihlášený uživatel, který není správce, bude moci editovat své údaje.

Úkol 20
Seznamte se s jazykem PHP a jeho vestavěnými funkcemi, jejichž dokumentace je dostupná na stránce `www.php.net/manual/en/funcref.php`.

Úkol 21
Vyzkoušejte administrační rozhraní phpMyAdmin a Adminer.

Architektury webových aplikací

V následující stručné kapitole se zaměříme na architekturu webové aplikace. V dnešní době se můžeme setkat s řadou různých architektur používaných pro vývoj webových aplikací. My se zaměříme na *monolitickou architekturu* a *microservices architekturu*.
Většina webových aplikací se skládá ze tří částí: uživatelského rozhraní, logiky aplikace a datového rozhraní. Význam uživatelského rozhraní je jasný.[85] Logika aplikace reprezentuje implementovanou funkcionalitu. Datové rozhraní představuje rozhraní pro připojení k databázi, kde jsou uložena data, se kterými aplikace pracuje.

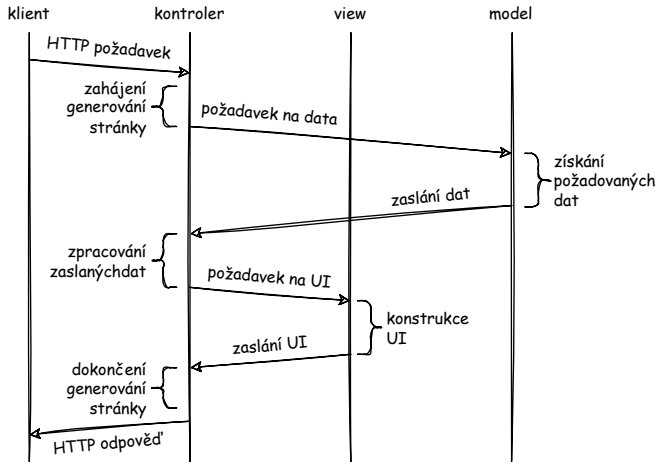
[85] Stručně řečeno, je to to co uživatel vidí a používá.

Průvodce studiem
V této kapitole budeme implementovat architektury pomocí dříve zmíněného jazyka PHP. Pohled zde prezentovaný je ale obecný.

Monolitická architektura

Monolitická architektura, jak její název napovídá, je architektura, kde uživatelské rozhraní, logika aplikace a datové rozhraní tvoří jeden celek. Vzhledem k tomu, že každá z těchto částí je typicky realizována pomocí jiné technologie, je velmi výhodné aplikaci rozdělit na několik částí.[86] Běžně se využívá *Model-View-Controller (MVC)* architektura, která je ilustrována na obrázku 5.

[86] Tím v ní udržíme pořádek a navíc zavedeme abstrakci.



Obrázek 5: MVC architektura webové aplikace. Klient zašle HTTP požadavek, který je webovým serverem předán kontroleru. Ten je zodpovědný za vytvoření odpovědi. Nejprve požádá model o získání potřebných dat, která zpracuje. Následně požádá view o vytvoření uživatelského rozhraní, které předá zpět klientovi.

Jednotlivé části MVC architektury (model, view a kontroler) implementují datové rozhraní, uživatelské rozhraní a logiku aplikace. Model je entita zodpovědná za výhradní přístup k databázi a práci s daty. S nejjednodušším příkladem modelu jsme se již setkali, byl jim objekt `$mysqli`.[87] View je entita, která vytváří vizualizaci (UI). Typicky je realizována jako šablona, ve které jsou nahrazeny výskyty proměnných předanými hodnotami. Triviální šablona můžeme být jednoduchý PHP skript, který generuje HTML a CSS kód.[88] Kontroler je entita implementující část funkcionality aplikace a je realizován jako PHP skript. Kontroler je obvykle rozdělen na nezávislé metody, kterým je možné předávat parametry.

[87] Běžně tuto roli plní různé mezivrstvy mezi aplikací a databází. Například ORM systémy.

[88] Běžně je view implementován pomocí šablonovacího systému.

V následující části si ukážeme nástin implementace monolitické aplikace s MVC architekturou.[89]

[89] Kompletní zdrojové kódy jsou k dispozici na stránce `trnecka.inf.upol.cz/teaching/weba/mvc.zip`.

Naše webová aplikace bude využívat následující schéma URL.

```
domena/kontroler
domena/kontroler/metoda
domena/kontroler/metoda/parametr1
domena/kontroler/metoda/parametr1/parametr2
```

Například URL `domena/users/edit/42` znamená, že voláme metodu `edit` kontroleru `users`, který implementuje funkcionalitu spojenou s uživateli, a předáváme ji parametr `42`.

Následující funkce zpracovává data v URL.[90] Tato část aplikace se označuje jako *router*. Jejím úkolem je určit jaký kontroler má být spuštěn a předat mu parametry zadané v URL. V ukázce se předpokládá, že volaná URL byla pomocí `moderewrite` přepsána do tvaru `domena/index.php?url=kontroler/metoda/parametr1`.

[90] Jedná se o velmi triviální implementaci. Pro jednoduchost vynecháme ošetření chyb.

```php
// zpracování URL - jednoduchý router
function process_URL()
{
  // parsování URL
  $URL = explode('/', $_GET['url'] ?? '');

  $controller = $URL[0] ?? null;
  $method = $URL[1] ?? null;
  $num_of_params = count($URL);

  $params = [];
  for ($i = 2; $i < $num_of_params; $i++) {
    $params[] = $URL[$i] ?? null;
  }

  return ['controller' => $controller, 'method' => $method,
      'params' => $params];
}
```

Použití funkce následuje.

```
$router = process_URL();

if ($router['controller']) {
  include "Controllers/$router[controller].php";
} else {
  // výchozí stránka
}
```

Pokud byl volán nějaký kontroler, je tento kontroler spuštěn. V opačném případě dojde k zobrazení výchozí stránky (případně spuštění kontroleru výchozí stránky). Dodejme, že plnohodnotné řešení by mělo ověřovat existenci volaného kontroleru. Jednotlivé kontrolery se nacházení ve složce `Controllers`. Následuje kostra kontroleru `users.php`.[91]

[91] V kontroleru chybí připojení k databázi. To v principu může být realizováno mimo kontroler, ale pokud by daný kontroler připojení k databázi nepotřeboval, bylo by realizováno zbytečně. Je tedy lepší jej ponechat v kontroleru. Také zde chybí ošetření toho, že je kontroler volán oprávněným uživatelem. Teoreticky, pokud by někdo věděl o existenci kontroleru, mohl by jej volat přímo. Tento typ útoku se označuje jako spuštění skriptu mimo kontext. Každý kontroler by měl ověřovat, že jej má daný uživatel oprávnění spustit.

```
1  // SHOW
2  if ($router['method'] == 'show') {
3    $users = ['Jack', 'Samantha', 'Daniel', 'Teal\'c'];
4
5    // zobrazení příslušné šablony (view)
6    include "Views/$router[controller]/$router[method].php";
7  }
8
9  // ADD
10 elseif ($router['method']== 'add') {
11   print_r($_POST);
12
13   // přesměrování na URL
14   header("Location: $GLOBALS[domena]/users/");
15   exit();
16 }
17
18 // EDIT
19 elseif ($router['method'] == 'edit') {
20   print_r($router['params']);
21 }
22
23 // DELETE
24 elseif ($router['method'] == 'delete') {
25   print_r($router['params']);
26 }
27
28 else {
29   echo "požadovaná metoda není podporována";
30 }
```

Kontroler má čtyři metody. V metodě `show` je na řádku 6 ukázáno volání view. Jak jsme již uvedli, view jsou opět PHP skripty, které

jsou pro pořádek umístěny ve složce `Views`. Kód view pro metodu `show` následuje.

```
1  <h1>List of users</h1>
2
3  <ul>
4    <?php foreach($users as $user) echo "<li>$user</li>"; ?>
5  </ul>
```

Samozřejmě daná metoda nemusí mít implementovaný view. V takovém případě se typicky předává řízení odpovídající části aplikace. Nástin předání řízení je vidět na řádku 14, kdy je použita funkce `header()`, která způsobí předání (přesměrování) na jinou URL.
Dodejme, že není nutné, aby jednotlivé metody měly oddělené view. Pokud to má smysl, je možné vytvořit jeden view, který dle zvolené metody určí, co má být zobrazeno.[92]
Architektura, kterou jsme nastínili má celou řadu výhod, ale i nevýhod. Výhodou je, že přidání nové funkcionality je vlastně jen přidání kontroleru a příslušných view. Nevýhodou je omezené škálování aplikace. Monolitická architektura neumožňuje snadné rozdělení aplikace na více uzlů (počítačů). Dnes se můžeme setkat s aplikacemi, které musí zvládat velké počty uživatelů a být neustále dostupné a to jak geograficky, tak v čase. To je typicky mimo možnosti jednoho uzlu. Nabízí se monolitickou architekturu rozdělit na nezávislé části. Jednou z množností je oddělit od sebe model, view a kontroler.[93] Toto oddělení je ale nedostatečné. Další možností je rozdělit samotnou funkcionalitu aplikace na menší nezávislé části.

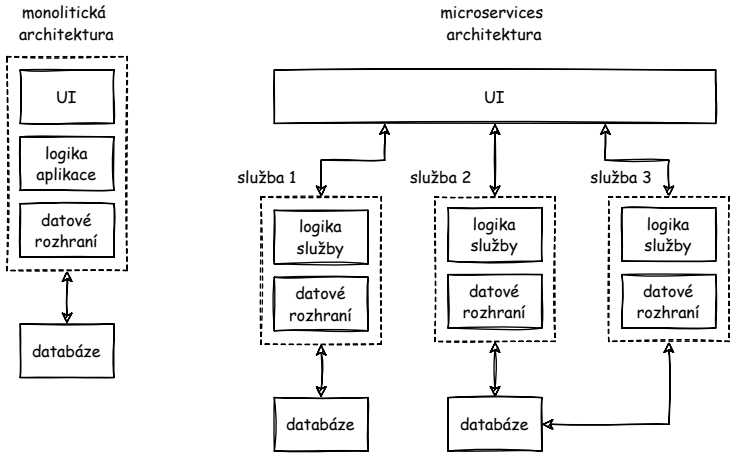
[92] Kód takového view by byl velmi podobný uvedenému kódu kontroleru. Typicky vytvoření a editace nějaké entity mají velmi podobné view. Například formuláře pro vytvoření nového uživatele a editaci existujícího uživatele jsou si velmi podobné. Nabízí se mít jedno view, které v případě přidání zobrazí prázdný formulář, v případě editace doplní data z databáze (které view získá od kontroleru). Takové view je lepší mít v jednom souboru, aby nevznikala zbytečná redundance kódu.

[93] Částečné rozdělení jsme provedli v kapitole Webový server, kde jsme pomocí `docker-compose` vytvořili několik nezávislých kontejnerů. Aplikace ale stále má monolitickou architekturu.

Microservices architektura

Microservices architektura je v určitém smyslu protikladem monolitické architektury, ve které je aplikace realizována jako jeden celek. V případě microservices architektury jsou jednotlivé části aplikace rozděleny na malé nezávislé *služby*, jejímž pospojováním (vzájemnou komunikací) získáme výslednou aplikaci. Rozdíl mezi monolitickou a microservices architekturou je ilustrován na obrázku 6.
Ve skutečnosti je problematika microservice aplikací výrazně složitější a můžeme se setkat s různými variantami této architektury. Základní myšlenka, tedy rozdělení na nezávislé části, je ale stejná. Jednotlivé služby mohou být realizovány tak, jak jsme to již nastínili v části věnující se monolitické architektuře.[94] Komunikace uživatele s jednotlivými službami probíhá pomocí webového API (Application Programming Interface).
Dodejme, že webové API je možné (a výhodné) použít i v případě monolitické architektury. Tím docílíme výraznějšího oddělení uži-

[94] Služby jsou ve své podstatě velmi malé monolitické aplikace.



Obrázek 6: Rozdíl mezi monolitickou a microserveces architekturou. V případě monolitické architektury je obsahuje logika aplikace jednotlivé služby (funkcionality). V případě microservices architektury tvoří služby samostatné malé monolitické aplikace.

vatelského rozhraní a logiky aplikace. Výhodou je, že uživatelské rozhraní může být realizováno i několika různými způsoby současně. Například jako mobilní aplikace, desktopová aplikace nebo webová stránka.

Průvodce studiem
Velmi často opomíjenou architekturou je modulární monolitická architektura. Zjednodušeně se jedná o kompromis mezi monolitickou a microservices architekturou. Přesněji se jedná o monolitickou architekturu se striktním oddělením jednotlivých podčástí. V zásadě se ale nejedná o speciální typ architektury, ale o monolitickou architekturu, ve které jsou dodrženy zásady dobrého programového kódu.

Webové API

Webové API je API, které využívá pro komunikaci protokol HTTP (či HTTPS).[95] Typicky je realizováno jako back-end s žádným nebo jednoduchým front-endem a veřejně dostupnými *endpointy*. Endpoint představuje rozhraní, které je využito pro spuštění požadované funkcionality.[96]
Nejběžněji se můžeme setkat s *REST API* a *GraphQL API* architekturou.[97]

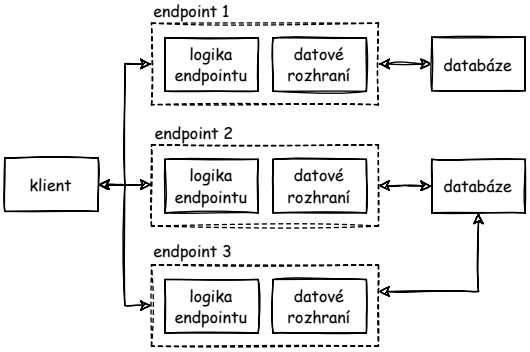
[95] Jedná se tedy o request-response systém.

[96] Ve své podstatě se jedná o URL.

[97] Architektur API je více, stejně tak REST API a GraphQL API architektury mají několik variant.

REST

Architektura REST API je zobrazena na obrázku 7.



Obrázek 7: REST API architektura.

Klient komunikuje s endpointy, které implementují požadovanou

funkcionalitu, pomocí HTTP. Běžně se používají HTTP metody `GET` (pro získání dat), `POST` (pro vytvoření nových dat), `PUT` (pro změnu dat) a `DELETE` (pro smazání dat).[98] Může se také setkat s implementací, která využívá pouze `GET` a `POST`.

Data jsou z REST API předávána v podobě strukturovaných dat. Nejčastěji se můžeme setkat s formáty JSON, XML nebo CSV. Dodejme, že data jsou vždy klientovi předávána tak, jak to implementuje daný endpoint. Klient tedy nemůže zažádat pouze o část dat.

Implementace jednoduchého REST API pomocí PHP by mohla vypadat následovně. Aplikaci si rozdělíme na dvě části `index.php` a soubory implementující jednotlivé endpointy, které budou uloženy ve složce `Endpoints`. Následuje část kódu v souboru `index.php`.[99]

```
1  function process_URL()
2  {
3    $URL = explode('/', $_GET['pozadavek'] ?? '');
4
5    $version = $URL[0] ?? null;
6    $endpoint = $URL[1] ?? null;
7    $method = $URL[2] ?? null;
8    $num_of_params = count($URL);
9
10   $params = [];
11   for ($i = 3; $i < $num_of_params; $i++) {
12     $params[] = $URL[$i] ?? null;
13   }
14
15   return [
16     'version' => $version,
17     'endpoint' => $endpoint,
18     'method' => $method,
19     'params' => $params
20   ];
21 }
22
23 // zpracování URL
24 $api = process_URL();
25
26 if ($api['version'] != "v1") {
27   die("This version of API is not supported!");
28 }
29
30 else {
31   include "endpoints/$api[endpoint].php";
32 }
```

Výše uvedený kód plní roli routeru.[100] Dle zadané URL rozhodne o volání konkrétního endpointu (řádek 31). Dodejme, že z prak-

[98] `POST` je možné využit i pro změnu dat, běžně ale slouží pouze pro vytváření nových dat.

Průvodce studiem
Identifikátor endpointu (URL) může obsahovat název metody. Například `user/delete/42`. Jedná se ale pouze o název, který neříká nic o použité HTTP metodě. V uvedeném případě je možné použít i metodu `GET`.

[99] Pro jednoduchost vypustíme ověření, že uživatel má možnost volat daný endpoint a metodu. Rovněž vynecháme ověření existence daného endpointu. Upozorněme ale, že obojí je z pohledu bezpečnosti zásadní.

[100] Je to pouze mírná modifikace funkce `process_URL()`, kterou jsme již viděli dříve.

tických důvodů je výhodné API verzovat. API se může v průběhu vývoje aplikace měnit, což může způsobit nekompatibilitu. Běžně se číslo verze API udává v URL. V ukázce výše je provedena triviální detekce a kontrola verze API.
Kostra kódu implementující konkrétní endpoint by mohla vypadat například následovně.

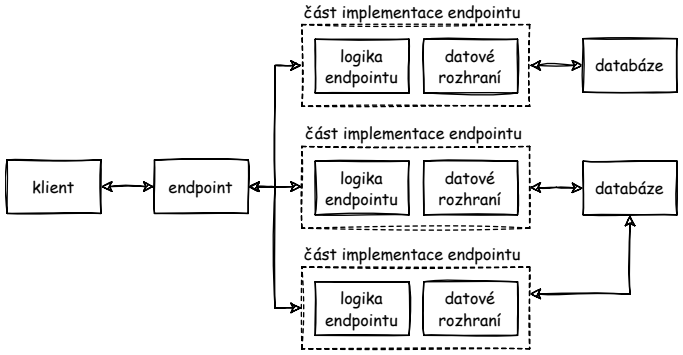
```
// GET
if ($api['method'] == "get") {
  print_r($api['params']);
}

// POST
elseif ($api['method'] == "post") {
  print_r($_POST);
}

// UPDATE
elseif ($api['method'] == "update") {
  print_r($_POST);
}

// DELETE
elseif ($api['method'] == "delete") {
  print_r($api['params']);
}

else {
  die("This method is not supported!");
}
```

Pokud výše uvedený kód uložíme do souboru `users.php` ve složce `Endpoints`. Je možné konkrétní metodu volat pomocí URL. Například metodu `get` pomocí URL `domena/v1/users/get/` nebo případně s parametry `domena/v1/users/get/42`.

GraphQL

Architektura GraphQL API je zobrazena na obrázku 8. Na rozdíl od REST API architektury je v GraphQL API architektuře pouze jeden endpoint, se kterým klient komunikuje pomocí metody `POST`. V těle zaslané zprávy je uložen, ve formátu JSON, požadavek na data nebo zasílaná data samotná. Požadavky na data se zapisují pomocí dotazovacího jazyka. Pomocí něj je možné například specifikovat podobu vrácených dat a tím snížit celkovou datovou zátěž.[101] Endpoint je zodpovědný za vyhodnocení dotazu a jeho předání příslušné části implementace.
Dodejme, že GraphQL je velmi pokročilá API architektura, která

[101] Klient dostane jen data o která žádá.



Obrázek 8: GraphQL API architektura.

umožňuje provádět i například typovou kontrolu dat. Její podrobné vysvětlení je nad rámec tohoto textu.

Shrnutí

V této kapitole jsme stručně představili různé architektury používané ve webových aplikací, přičemž jsme se zaměřili výhradně na back-end aplikace. Později s k této problematice vrátíme a popíšeme architektury webových aplikací z pohledu front-endu.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jakou strukturu má monolitická webová aplikace?
2. Jakou strukturu má microservices webová aplikace?
3. Co je to MVC architektura?
4. Co je to webové API, k čemu slouží, a jak je možné jej realizovat?

Úkoly

Úkol 22
Upravte administrační z předchozích kapitol tak, aby používalo MVC architekturu.

Úkol 23
Napište jednoduchou REST API službu uchovávající seznam uživatelů v session.[102] Endpointy API a jejich popis jsou shrnuty v následující tabulce.

[102] Uživatelé budou mít tři položky `id`, `name`, `surname`.

endpoint	popis
`get`	vrátí seznam všech uživatelů v CSV formátu
`get/id`	vrátí uživatele s daným `id` v CSV formát
`post`	vytvoří nového uživatele, data jsou předána skrze HTTP POST požadavek
`update/id`	změní uživatele s daným `id`, nová data jsou předána skrze HTTP POST požadavek
`delete`	smaže uživatele se zadaným `id`

Úkol 24
Upravte API z předchozího úkolu tak, aby se seznam uživatelů ukládal do souboru.

JavaScript

„JavaScript is the duct tape of the Internet“

Charlie Campbell

JavaScript je plnohodnotný a v dnešní době velmi populární programovací jazyk. Jeho autorem je Eich Brendan, který jej v roce 1995 vytvořil pro dnes již neexistující společnost Netscape.[103] Původně byl určen pro jednoduché skriptování na straně klienta, tedy vytváření programového kódu, jenž je vykonáván webovým prohlížečem. Běžně se pro tuto část JavaScriptu používá označení *klientský JavaScript*. S rostoucími možnostmi jazyka a webových prohlížečů,[104] bylo možné pomocí JavaScriptu vytvářet na straně klienta i plnohodnotné webové aplikace. Později přibyla možnost spouštět JavaScriptový kód i na straně serveru. V nadcházející kapitole se budeme věnovat základům samotného jazyka a klientskému JavaScriptu. Další možnosti ukážeme později.[105]
Základy klientského JavaScriptu byly představeny v kurzu *Tvorba webových stránek*. Pro zachování nezávislosti tohoto textu je následující kapitola přepracovanou a rozšířenou verzí kapitoly *JavaScript* z textu *Trnecka, M.: Tvorba webových stránek*.[106] Stejně jako v předchozích kapitolách, budeme i nyní předpokládat základní znalost programování.

[103] JavaScript nesl původně jméno LiveScript, ale díky velmi velké popularitě jazyka Java, byl z čistě marketingových důvodů přejmenován na JavaScript.

[104] Dodejme, že možnosti webových prohlížečů se stále rozšiřují, zatímco jazyk JavaScript se v současnosti mění spíše minoritně.

[105] JavaScript se běžně používá všemi uvedenými způsoby.

[106] Původní text se věnuje tvorbě webového front-endu. Z tohoto důvodu jsou v tomto přepracovaném textu některé méně podstatné části vynechány, jiné jsou rozšířeny o koncepty potřebné pro pokročilejší programování, které budou třeba v nadcházejících kapitolách, a tvorbu webových aplikací.

Jazyk JavaScript

JavaScript je konkrétní implementací standardu ECMAscript. Standard vychází jednou ročně a čísluje se aktuálním rokem. Tento způsob číslování se používá od verze 6, která byla největším updatem ECMAscriptu v jeho historii.[107] Dodnes se můžeme setkat s pojmem ECMAscript nebo JavaScript verze 6 vlastnosti. Ty odkazují právě na možnosti jazyka přidané v této verzi. Zejména v poslední době nové verze ECMAscriptu přinášejí pouze drobné změny.

[107] ECMAscript 6 byl vydán v roce 2015.

Klientský JavaScript

Možnosti a omezení klientského JavaScriptu pramení z faktu, že se jedná o programový kód, který je vykonáván, obvykle automa-

ticky, ve webovém prohlížeči na straně klienta. JavaScript umí řídit vzhled a obsah webové stránky tím, že umožňuje manipulaci s HTML a CSS na stránce. Kromě toho dokáže manipulovat přímo s oknem prohlížeče, ovládat formuláře, zapisovat a číst cookies a interagovat s uživatelem formou různých událostí.[108]
Omezení klientského JavaScriptu mají především chránit uživatele, který si webovou stránku prohlíží. Je třeba si uvědomit, že možnost automaticky spustit programový kód je velmi nebezpečná. Zjednodušeně můžeme říct, že klientský JavaScript nedokáže ovládat nic mimo webový prohlížeč.[109] Omezeních je samozřejmě více, ale pro účely tohoto kurzu jsou nezajímavé.
JavaScriptový kód je možné do webové stránky přidat pomocí elementu `script`. Ten umožňuje vložit externí soubor s příponou `.js`, obsahující kód, který se po načtení začne automaticky vykonávat. `script` se obvykle zapisuje do elementu `head`. Obecně může být i kdekoliv v elementu `body`, ale jeho pozice má vliv na vykonávání skriptu, jelikož se JavaScriptový kód vykonává ihned po jeho načtení. Později ukážeme, že JavaScript umožňuje manipulovat s webovou stránkou, například přidávat nebo odebírat elementy. Pokud například umístíme kód měnící část webové stránky před tuto část, může dojít k jeho vykonání ještě před renderováním dotyčné části stránky. V tento okamžik tedy prohlížeč nemůže kód vykonat, jelikož dotyčná část pro něj ještě neexistuje. Další možností začlenění JavaScriptového kódu do webové stránky je jeho zapsání přímo do elementu `script`.

[108] Například reagovat na událost odeslání formuláře.

[109] To není zcele pravda, jelikož pokročilý klientský JavaScript umožňuje například čtení a ukládání dat na klientském počítači.

Rychlokurz jazyka JavaScript

V následující části stručně představíme základní syntaxi jazyka JavaScript.[110] JavaScript je case-sensitive a běžně se pro zápis používá camel case konvence. Příkazy jazyka musí být ukončeny středníkem.[111] Komentáře se zapisují pomocí `//` (jednořádkový komentář) nebo `/* */` (víceřádkový komentář).

[110] Plnohodnotné představení jazyka je nad rámec tohoto kurzu. Výborným zdrojem informací je kniha *Flanagan, D.: JavaScript: The Definitive Guide, 7th Edition, 2020.*

[111] V některých situacích je možné středník vypustit. Z důvodů přehlednosti se to ale moc nedoporučuje.

Proměnné

JavaScript je dynamicky typovaný programovací jazyk. Proměnné se deklarují pomocí klíčového slova `var`.

```
var x = 42;
console.log(x); // zápis do terminálu

var y = 1;
console.log(x + y);

var z;
console.log(z); // nedefinován proměnná "undefined"
```

Ve výše uvedeném `console.log()` provede zápis do terminálu.[112] Neměnné proměnné (konstanty) se vytvářejí pomocí klíčového slova `const`.[113]
Proměnné definované pomocí `var` mají lexikální rozsah platnosti na úrovni funkcí. Pomocí operátoru `let` je možné definovat proměnné s rozsahem platnosti na úrovni bloku (vymezeného symboly `{` a `}`).[114] Uvažme následující příklad.

[112] Ten je dostupný v prohlížeči ve vývojářských nástrojích.

[113] Pokus o změnu konstanty skončí chybou.

[114] Tento způsob definice je velmi populární. Na klíčové slovo `var` se v moderním JavaScriptu téměř zapomnělo.

```
1 var x = 42;
2
3 {
4   var x = 0;
5 }
6
7 console.log(x);
```

Na řádku 4 je vytvořena deklarována `x`. Výpis na řádku 7 vypíše hodnotu `0`. V případě, že řádek 4 nahradíme příkazem `let x = 0`, výpis na 7 řádku vypíše hodnotu `42`. Pro úplnost dodejme, že konstanty mají rovněž rozsah platnosti na úrovni bloku.
Základní datové typy jsou číslo, řetězec, logický typ, `undefined`, `null`. S proměnnými je možné provádět všechny běžné operace, které známe z jiných jazyků.
Pro úplnost dodejme, že proměnné v jazyce JavaScript mohou začínat symbolem `_` a `$`.[115]

[115] Tyto proměnné se typicky využívají pro označení privátních vlastností objektů, či pro jiné speciální účely.

Řetězce

V jazyce JavaScript se řetězce zapisují pomocí uvozovek a apostrofů. Oba způsoby jsou ekvivalentní. Pro spojení řetězců se využívá operátor +. Například.

```
console.log(fisrtName + ", " + lastName);
```

Výše uvedený kód je možné přepsat pomocí modernější syntaxe využívající *template string*.[116]

[116] Ty se zapisují pomocí znaku ‘ (obrácený apostrof).

```
console.log(‘${fisrtName}, ${lastName}‘);
```

Pole

Pole se vytváří pomocí hranatých závorek, případně pomocí funkce `Array()`. Přístup k prvkům pole se provádí pomocí indexačního operátoru. Příklad následuje.[117]

[117] Jednotlivé prvky pole mohou být různého typu. Netřeba dodávat, že prvky pole mohou být opět pole.

```
var pole = [1, 2, 3, "42"];
console.log(pole[3]);

pole[3] = 4
console.log(pole); // [1, 2, 3, 4]
```

Objekty

JavaScript neumožňuje vytvářet asociativní pole s vlastním indexování. Na místo toho se využívají *objekty*.[118] Ty se zapisují pomocí `{ a }` tak, jak je ukázáno v následujícím kódu, případně pomocí konstruktoru `Object`.

[118] Z terminologického hlediska je pojmenování objekt poněkud matoucí. Zde objektem myslíme datový typ jazyka, nikoliv instanci třídy v OOP.

```
var objekt = {a: 1, b: 2, c: 3};
```

Výše uvedený kód vytvoří objekt s vlastnostmi `a`, `b` a `c`. K těm je možné přistoupit pomocí operátoru `.` (tečka).

```
objekt.a
```

Objekty v jazyce JavaScript lze chápat jako asociativní pole. Jednotlivým vlastnostem je možné přistupovat pomocí indexu stejného jména.[119] Můžeme tedy psát.

[119] Indexy jsou textové řetězce obsahující název vlastnosti.

```
objekt["a"]
```

Pro úplnost uveďme, že pole, objekty a řetezce jsou *iterovatelné* entity, které mají své (různé) vlastnosti a metody. K zjištění délky pole slouží vlastnost `length`.

```
// délka pole
console.log(pole.length);
```

Přehled vybraných metod je uveden v tabulce 3.

Větvení programu

Následující příklad ukazuje základní konstrukci pro větvení programu.

```
var x = 42

if (x < 42) {
  console.log("x je menší než 42");
}
```

metoda	popis
`.concat()`	spojí dvě nebo více polí do jednoho
`.copyWithin()`	kopíruje prvky pole v rámci pole ze zadaných pozic na zadané pozice
`.entries()`	vrátí iterátor pole, který obsahuje páry klíč-hodnota pro každý index v poli
`.every()`	ověří, zda každý prvek pole splňuje test
`.fill()`	naplní všechny prvky pole zadanou hodnotou
`.filter()`	vytvoří nové pole s prvky, které splňují zadaný test
`.find()`	vrátí hodnotu prvního prvku v poli, který splňuje zadaný test
`.findIndex()`	vrátí index prvního prvku v poli, který splňuje zadaný test
`.findLast()`	vrátí hodnotu posledního prvku v poli, který splňuje zadaný test
`.findLastIndex()`	vrátí index posledního prvku v poli, který splňuje zadaný test
`.forEach()`	zavolá zadanou funkci pro každý prvek pole
`.includes()`	zjistí, zda pole obsahuje určitý prvek (vrácí `true` nebo `false`
`.indexOf()`	vyhledá prvek v poli a vrátí jeho pozici
`.join()`	spojí všechny prvky pole do řetězce
`.keys()`	vrátí iterátor pole, který obsahuje klíče každého indexu v poli
`.lastIndexOf()`	vyhledá prvek v poli od konce a vrátí jeho pozici
`.map()`	mapuje zadanou funkci na každý prvek pole, vrací nové pole
`.push()`	uloží na konec pole zadaný prvek
`.pop()`	odstraní z pole poslední prvek
`.reduce()`	redukuje hodnoty pole na jedinou hodnotu (zleva doprava)
`.reverse()`	obrátí pořadí prvků v poli
`.slice()`	vybere část pole a vrátí ji jako nové pole
`.some()`	zkontroluje, zda některý z prvků pole splňuje zadaný test
`.sort()`	seřadí prvky pole
`.splice()`	přidá/odstraní prvky z pole
`.values()`	vrátí iterátor pole, který obsahuje hodnoty pro každý prvek v poli

Tabulka 3: Přehled vybraných metod JavaScriptových polí.

```
else if (x > 42) {
  console.log("x je větší než 42");
}

else {
  console.log("x je rovno 42");
}
```

Analogicky jako v jiných jazycích je povinná pouze část `if`. Kromě konstrukce `if` je možné využít i příkaz `switch`. JavaScript rovněž disponuje ternárním operátorem.

Cykly

Jako ve většině programovacích jazyků je možné i v JavaScriptu opakovaně vykovat programový kód pomocí cyklů. Mezi základní cykly patří `for` a `while`. Jednoduchý příklad vypisující prvky pole následuje.

```
var pole = [1, 2, 3, 4, 5];

// for cyklus
for (var i = 0; i < pole.length; i++) {
  console.log(pole[i]);
}

// cyklus while
var i = 0
while (i < pole.length) {
  console.log(pole[i]);
    i = i + 1;
}
```

V případě, že je iterujeme přes iterovatelný objekt, je možné využít i zjednodušenou syntaxi uvedenou níže.

```
// for cyklus přes iterovatelný objekt
for (i in pole) {
  console.log(pole[i]);
}

// for cyklus přes prvky iterovatelného objektu
for (i of pole) {
  console.log(i);
}
```

V cyklech je možné používat `break` a `continue`.
Pole navíc disponují metodou `.forEach()`, umožňující iteraci přes všechny jejich položky. Argumentem této metody je funkce, která je aplikována na každou položku pole.

Funkce

Funkce se zapisují pomocí klíčového slova `function`. Návrat z funkce se provádí pomocí `return`. Příklad následuje.

```
var pole = [1, 2, 3, 4, 5]

function suma(pole) {
  let soucet = 0;
```

```
  for (let i = 0; i < pole.length; i++) {
    soucet += pole[i];
  }

  return soucet;
}

console.log(suma(pole)); // 15
```

Parametrům funkce je možné zadat výchozí hodnotu tak, jak je ukázáno v kódu níže.

```
function show(value = 42) {
  console.log(value);
}
```

Samozřejmostí jsou funkce vyšší řádů a anonymní funkce. Anonymní funkce se zapisují následovně.

```
(x) => {console.log(x);}
```

V případě, že anonymní funkce má právě jeden vstupní parametr, je možné zapsat anonymní funkci následovně.

```
x => {console.log(x);}
```

Analogicky je možné vynechat `{` a `}`, pokud tělo funkce obsahuje pouze jeden příkaz.[120]
Upozorněme, že při vracení objektů z anonymní funkce je třeba použít `()`.

[120] V tomto případě je možné vynechat i `;` (středník).

```
const user = (firstName, lastName) => ({givenName: firstName, surnNme: lastName})
console.log(user("Samantha", "Carter"));
```

Objektově orientované programování

Jazyk JavaScript umožňuje zapisovat třídy pomocí klíčového slova `class`. Následuje příklad třídy reprezentující osobu, vytvoření instance této třídy a volání metody.[121]

[121] Tento způsob zápisu přišel s ES6. Kvůli filozofii JavaScriptu, která upřednostňuje funkcionální přístup, není tento zápis příliš populární.

```
class Person {
  name;

  constructor(name) {
    this.name = name;
  }
```

```
  introduceSelf() {
    console.log(`${this.name}`);
  }
}

p = new Person("Samantha Carter");
p.introduceSelf();
```

Níže je uveden alternativní zápis nevyužívající klíčové slovo `class`.

```
Person = {
  name,

  introduceSelf: function() {
    console.log(`${this.name}`);
  }
}

p = Object.create(Person);
p.name = "Smantha Carter";

p.introduceSelf();
```

Pro úplnost ještě uveďme použití *getterů* a *seterů*.[122]

[122] K jejich zápisu se v jazyce JavaScript používají klíčová slova `get` a `set`. V příkladu je také uvedeno typické konvence (použití symbolu _) pro zápis privátních vlastností.

```
let Bod = {
  _x: null,
  _y: null,

  get x() {
    return this._x;
  },

  set x(val) {
    this._x = val;
  },

  get y() {
    return this._y;
  },

  set y(val) {
    this._y = val;
  }
}

bod = Object.create(Bod);
bod.x = 10;
```

```
bod.y = 10;
console.log(bod);
```

Pokročilejší konstrukce

Od verze ES6 je možné objekty vytvářet následovně.[123]

[123] Tento způsob se označuje jako *object literal enhancement*.

```
var givenname = "Samantha";
var surname = "Carter";

const user = {
  givenname,
  surname,
  log() {
    console.log(`${this.givenname}, ${this.surname}`);
  }
};

user.log(); // vypíše "Samantha, Carter"
```

Analogicky je možné následující.

```
var name = "SG1";

const teams = {
  [name]: "Jack O'neill, ..."
}

console.log(teams.SG1);
```

Další užitečnou konstrukcí je *destrukturalizace* objektu.

```
const objekt = {a: 42, b: 43};

var {a} = objekt;
console.log(a);
```

Jazyk JavaScript umožňuje použití *spread syntaxe,* která s zapisuje podobně jako v jiných jazycích pomocí . . . (tři tečky).

```
function sum(x, y, z) {
  return x + y + z;
}

const pole = [1, 2, 3];

console.log(sum(...pole)); // 6
```

Spread syntaxi je možné používat jako argument funkce a v iterovatelných entitách jako jsou pole, objekty a řetězce. Upozorněme na potencionální záměnu s *rest syntaxí*, které se zapisuje stejně, ale slouží pro vytvoření variadické funkce (funkce s proměnlivým počtem parametrů).

```
function sum(...args) {
  let total = 0;
  for (let arg of args) {
    total += arg;
  }
  return total;
}

console.log(sum(1, 2, 3));
```

Prototypy

Jazyk JavaScript je prototypový programovací jazyk.[124] Jednotlivé entity vznikají kopírováním existujících prototypů.[125] Pokud vytvoříme například pole, je ve skutečnosti toto pole konkrétní instancí prototypu pole. JavaScript umožňuje k prototypu přistoupit přes vlastnost `prototype`. Pokud bychom chtěli například vytvořit novou metodu pro pole, můžeme to udělat následovně.

[124] Jedná se o významný rys tohoto jazyka.

[125] To je velmi nepřesné, ale pro účely tohoto kurzu dostačující.

```
// rozšíříme prototyp pole
Array.prototype.metoda = function() {
  console.log('Hello');
}

var pole = [];

// volání nové metody
pole.metoda();
```

Manipulace s webovou stránkou

Klientský JavaScript disponuje objektem `window`, který propojuje JavaScript s webovým prohlížečem. Změnou tohoto objektu je možné provádět manipulaci s oknem prohlížeče. Například změnit velikost, zavřít jej a mnoho další. V dnešní době je přímá manipulace s oknem prohlížeče historický přežitek, přesto jsou některé metody stále užitečné. Několik z nich je shrnuto v tabulce 4.
Nejdůležitější částí objektu `window` je objekt `window.document`, který obsahuje reprezentaci webové stránky. Pro jednoduchost se `window`

metoda	popis
`window.alert()`	zobrazení hlášení, které musí být potvrzeno
`window.confirm()`	zobrazení hlášení, které musí být potvrzeno, nebo odmítnuto
`window.scrollTo()`	rolování stránky na danou pozici
`window.history.back()`	posun v historii prohlížeče zpět
`window.history.forward()`	posun v historii prohlížeče vpřed
`window.location`	aktuální URL

Tabulka 4: Základní metody pro manipulaci s objektem `window`

ze zápisu vynechává. Lze tedy psát pouze `document`.

Reprezentace webové stránky

Webová stránka, která je dostupná přes objekt `document`, je v prohlížeči reprezentována jako Document Object Model (DOM) struktura. Podoba této struktury je dána standardem.[126] Ten kromě samotné struktury popisuje i operace, které lze s touto strukturou provádět. V současnosti je většina běžné funkcionality podporována napříč prohlížeči, ale analogicky jako u HTML a CSS i u JavaScriptu je nutné ověřovat podporu pomocí `caniuse.com`.
DOM je stromová struktura, přičemž jednotlivé HTML elementy, které jsou obsaženy na stránce, jsou podmnožinou uzlů DOM. Podívejme se na jednoduchý příklad. Uvažme následující HTML kód.

[126] Standard je dostupný na stránce `dom.spec.whatwg.org`.

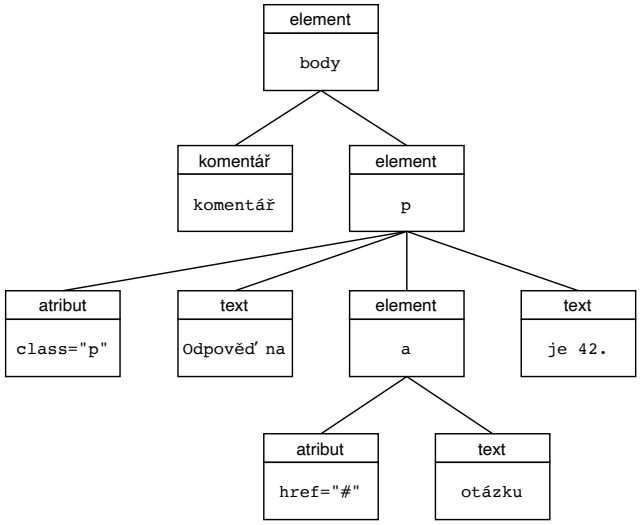
```
<body>
  <!-- komentář -->
  <p class="p">Odpověď na <a href="#">otázku</a>je 42.</p>
</body>
```

Průvodce studiem
HTML elementy jsou skutečně podmnožinou uzlů DOM. Toto je velmi důležité, jelikož JavaScript poskytuje nejen operace nad všemi uzly, ale i operace omezené pouze na HTML elementy, které ostatní uzly ignorují. Operace pouze nad elementy jsou z pohledu tvorby webového frontendu zajímavější.

DOM struktura pro výše uvedený HTML kód je ilustrována na obrázku 9.
Můžeme vidět, že DOM uchovává veškeré informace o HTML kódu. Kromě samotných elementů a zachycení vztahu potomek-rodič je v DOM uchováván i obsah elementů, atributy elementů a také komentáře. JavaScript umožňuje s touto strukturou manipulovat. Pokud chceme na stránce provést nějakou změnu, musíme nejprve identifikovat místo v DOM, kde bude změna provedena a následně v tomto místě změnu provést.

Výběr elementů v DOM

Tabulka 5 shrnuje základní metody pro výběr elementů. Všechny uvedené metody vrací element nebo pole elementů. Pokud není nalezen žádný odpovídající element, je vrácena hodnota `null`. Pokud máme k dispozici konkrétní element, můžeme vybírat jeho po-



Obrázek 9: Příklad DOM struktury.

metoda	popis
`document.getElementById()`	výběr elementu dle atributu `id`
`document.getElementsByClassName()`	výběr elementů dle atributu `class`
`document.getElementsByTagName()`	výběr elementů dle jejich jména
`document.querySelector()`	výběr elementu dle CSS selektoru, vrací první odpovídající element
`document.querySelectorAll()`	výběr elementů dle CSS selektoru

Tabulka 5: Základní metody pro výběr elementů v DOM.

tomky, sourozence a rodiče. Tomuto se běžně říká navigace v DOM. Základní metody jsou shrnuty v tabulce 6.

Změna DOM

Manipulace s webovou stránkou je prováděna skrze manipulaci DOM. Uvažme následující část HTML kódu

```
<ul class="list">
  <li class="list__item">A</li>
  <li class="list__item">B</li>
  <li class="list__item">C</li>
</ul>
```

Budeme chtít přidat čtvrtou položku seznamu, která bude obsahovat písmeno `D`. Nejprve nalezneme v DOM odpovídající element.

metoda	popis
`.firstElementChild`	vrací element, který je prvním potomkem daného elementu
`.lastElementChild`	vrací element, který je posledním potomkem daného elementu
`.parentElement`	vrací element, který je rodičem daného elementu
`.nextElementSibling`	vrací element, který je následujícím sourozencem daného elementu
`.previousElementSibling`	vrací element, který je předchozím sourozencem daného elementu
`.children[cislo_uzlu]`	vrací uzel DOM, který je potomkem daného elementu

Tabulka 6: Základní metody pro navigaci v DOM.

```
// výběr seznamu
var element = document.querySelector(".list");
```

Následně vytvoříme uzly DOM.

```
// vytvoříme nový element li
var newElement = document.createElement("li");

// vytvoříme nový uzel DOM, který bude obsahovat text
var newTextNode = document.createTextNode("D");

// uzel obsahující text nastavíme jako potomka elementu
newElement.appendChild(newTextNode);
```

Nakonec připojíme nový element (a jeho potomka) na odpovídající místo v DOM.

```
// přidání další položky seznamu
element.appendChild(newElement);
```

Tímto způsobem je možné do DOM přidávat libovolné elementy. Kromě uvedených je možné použít i metodu `.createAttribute()` pro vytvoření uzlu reprezentující atribut elementu. Tabulka 7 shrnuje základní metody pro manipulaci s DOM.

Vlastnosti elementů

Každý element má několik vlastností. Například `.innerHTML` obsahuje HTML kód obsažený v daném elementu, `.innerText` obsahuje obsah elementu v textové podobě (tedy bez HTML kódu), `.style` obsahuje CSS vlastnosti a jejich nastavení a `.classList` seznam tříd. Tyto vlastnosti je možné číst a nastavovat.

```
// nastavení vlastnosti color
newElement.style.color = "gold";
```

metoda	popis
`.appendChild()`	přidání nového potomka
`.removeChild()`	odebrání existujícího potomka
`.replaceChild()`	nahrazení potomka novým potomkem
`.insertAdjacentElement()`	vložení před, za a do elementu, lze řídit pomocí parametrů
`.insertAdjacentHTML()`	analogické jako předchozí, elementy jsou vytvořeny z HTML kódu zadaného jako textový řetězec
`.getAtributte()`	přečtení hodnoty atributu
`.setAtributte()`	nastavení hodnoty atributu
`.removeAtributte()`	odstranění atributu

Tabulka 7: Základní metody pro manipulaci s DOM. Podrobnější informace lze nalézt na stránce `developer.mozilla.org/en-US/docs/Web/API/Element`.

```
// přidání třídy k elementu
newElement.classList.add("list__item")
```

Kompletní ukázka je dostupná na stránce `codepen.io/trnecka/pen/XWpYoJp`.

Události

JavaScriptový kód je možné vykonávat při výskytu určité události, například při kliknutí na tlačítko myši (událost `onclick`) nebo při načtení stránky (událost `onload`).[127]
Aby se kód vykonal po vzniku události, je třeba jej s konkrétní událostí provázat. To lze provést pomocí atributu, který nese stejné jméno jako událost. Uvažme následující jednoduchou funkci `show()`.

[127] Událostí existuje velké množství. Přehled je dostupný na stránce `developer.mozilla.org/en-US/docs/Web/Events`.

```
function show() {
  console.log("kliknuto");
}
```

Propojení s událostí `onclick` provedeme následovně.

```
<div class="tlacitko--A">tlačitko A</div>
<div class="tlacitko--B" onclick="show();">tlačitko B</div>
```

Další možností propojení je použít funkci `addEventListener()`. Příklad analogického propojení, které bylo uvedeno výše následuje.[128]

[128] Upozorněme, že uvedení `show` bez závorek je opravdu správné.

```
document.querySelector(".tlacitko--A").addEventListener("
    click", show);
```

Časovače

Jednou z možných událostí je vypršení zadaného časového intervalu. Tato událost je řízena pomocí časovače. Časovač je možné vytvořit pomocí pomocí funkce `setInterval(funkce, cas)`. Tento časovač umožňuje opakovaně spouštět kód. Jakmile dojde k vypršení zadaného času `cas`, je vykonána funkce `funkce` a časovač je opět spuštěn. Pro jeho zastavení je třeba použít funkci `clearInterval()`. Čas se zadává v milisekundách. Příklad následuje.

```
// periodické opakování
var timer = setInterval(function(){console.log("čas vypršel");}, 3000);

// zastavení opakování
clearInterval(timer);
```

Dále je možné časovač vytvořit pomocí funkce `setTimeout(funkce, cas)`. Tento časovač umožňuje jednou spustit kód. Jakmile dojde k vypršení zadaného času `cas`, je vykonána funkce `funkce` a časovač zaniká.

```
// jedno opakování
setTimeout(function(){console.log("čas vypršel");}, 3000);
```

Chyby a ladění

Pokud obsahuje JavaScripový kód chybu, může být obtížné ji na první pohled najít, jelikož jakákoliv chyba způsobí zastavení vykonávání veškerého JavaSriptového kódu na stránce. Pro účely ladění je možné využít logování do konzole, případně ladící nástroje, které jsou součástí vývojářských nástrojů. Příklad logování následuje.

```
var promenna = "konec se již blíží";

console.log(promenna);
console.info(promenna);
console.warn(promenna);
console.debug(promenna);
console.error(promenna);
```

Praktické ukázky

V následující části si ukážeme několik typických použití klientského JavaScriptu.

Změna vzhledu stránky

Změna vzhledu stránky (CSS vlastností) je nejběžnější akce, kterou klientský JavaScript provádí. Následuje komentovaná ukázka.

```
// změna CSS vlastností u jednoho elementu
// vybereme element, vrací první nalezený
const element = document.querySelector(".box");

// změníme vlasnost
element.style.color = "red";

// vzhled je možné měnit záměnou tříd
// přidání a odebrání třídy
element.classList.add("box--marked");
element.classList.remove("box--marked");

// změna u více elementů
var color = "blue";

// vybereme elementy
const elements = document.querySelectorAll(".box");

// aplikujeme změnu na všechny vybrané elementy
elements.forEach(e => e.style.color = color);
```

Validace formuláře

Základní validace formuláře je poskytována webovým prohlížečem.[129] Tato validace je poskytována formou API a JavaScript se na toto API dokáže napojit. Uvažme následující vstupní pole.

[129] Například HTML atribut `required`.

```
<form method="post" action="">
  <label for="mail">E-mail address:</label>
  <input type="email" id="mail" name="mail">

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

Napojení na validaci je ukázáno v kódu níže.

```
const email = document.getElementById("mail");
const form = document.querySelector("form");

// validace při psaní
email.addEventListener("input", e => {
  if (email.validity.typeMismatch) {
```

```
    email.setCustomValidity("I am expecting an e-mail address!");
    email.reportValidity();
  } else {
    email.setCustomValidity("");
  }
});

// validace při odeslání
form.addEventListener("submit", e => {
  if (!email.value) {
    showError(email);
    event.preventDefault(); // zabránění odeslání formuláře
  }
});

// pomocná funkce
function showError(e) {
  e.style.backgroundColor = "#ffcccb"; // vestavěné hlášky samy zmizí
  e.setCustomValidity("E-mail have to be entered!");
  e.reportValidity();
}

// poznámka: lze snadno rozšířit, třeba přidáním elementu span s {display: none;} za input
// a (například): e.nextElementSibling.style.display="inline"; do showError()
```

Zcela analogicky je možné vytvářet validaci vlastní.

Modální okno s potvrzením

Následující kódy ukazují možné řešení vyvolání modálního okna, které obsahuje žádost o potvrzení akce. Uvedené řešení je jedno z mnoha možných. Zcela univerzální řešení neexistuje.

```
<!-- tlačítka -->
<button class="button button--delete" data-action="users/delete/1">Delete forever</button>
<button class="button button--delete" data-action="users/delete/2">Delete forever</button>
<button class="button button--delete" data-action="users/delete/3">Delete forever</button>

<!-- modální dialogové okno -->
<dialog id="dialog">
  <p id="dialog__text">Do you really want to delete item
  <span id="dialog__item-to-delete">X</span>? This action cannot be undone.</p>
  <a href="" id="dialog__confirm-link">yes</a>
  <a href="#" onclick="closeDeleteDialog()">no</a>
</dialog>
```

Následuje JavaScriptový kód.

```
// najdeme všechny delete tlačítka
const deleteButtons = document.querySelectorAll(".button--delete");

// navázání události
deleteButtons.forEach(b => b.addEventListener("click", e => {
  const dialog = document.getElementById("dialog");
  const action = b.dataset.action; // data z data-* atributu

  // parsování id z data atributu
  const item = document.getElementById("dialog__item-to-delete");
  item.innerHTML = action.split("/").pop();;

  // změna linku
  const link = document.getElementById("dialog__confirm-link");
  link.setAttribute("href", action);

  dialog.showModal();
}));

// zavření dialogového okna
function closeDeleteDialog() {
  const dialog = document.getElementById("dialog");
  dialog.close();
}
```

Mizející notifikace

Velmi často používaným UI prvkem jsou notifikace. Následující kód ukazuje použití časovače pro zmizení notifikace.

```
// najdeme všechny notifikace
const notifications = document.querySelectorAll(".
    notification");

// po 3s je odstraníme
setTimeout(() => {notifications.forEach(n => n.style.display
    ="none")}, 3000);
```

Výše uvedený kód způsobí okamžité zmizení notifikace. Hezčí řešení by zahrnovalo postupné zmizení (fade-out efekt).

```
var opacity = 1;

var fadeOutInterval = setInterval(() => {
  opacity = opacity - 0.01;
  notifications.forEach(n => {
```

```
    n.style.opacity = opacity;
    if (opacity <= 0) {
      n.style.display = "none";
      clearInterval(fadeOutInterval);
    }
  })}, 30);
```

Případně čistější řešení pomocí rekurze.

```
const fadeOutEffect = (opacity = 1, delay = 30, decrement = 0.01) => {
  const notifications = document.querySelectorAll(".notification");

  if (opacity > 0) {
    notifications.forEach(n => n.style.opacity = opacity);
    setTimeout(() => fadeOutEffect(opacity - decrement, delay, decrement), delay);
  }
  else {
    notifications.forEach(n => n.style.display = "none");
  }
};

fadeOutEffect();
```

Závěr

V této kapitole jsme představili základní možnosti klientského JavaScriptu. Další představíme v následujících kapitolách. Ukázky v této kapitole se zaměřovali především na uživatelské rozhraní. Pro úplnost dodejme, že se v UI běžně vyskytují prvky jako například rozbalovací menu, modální okna, fotogalerie a slideshow. Existuje řada knihoven, které usnadňují jejich vytváření nebo přímo obsahují uvedené prvky. Jedna z nejuniverzálnějších knihoven je knihovna *jQuery*. Ta již není tak populární, jako před několika lety, ale stále se hojně používá. Současným trendem je využívání menších jednoúčelových knihoven a celkové odlehčení klientského JavaSriptu, pokud je to možné. Dalším příkladem je knihovna Bootstrap, které implementuje mnoho užitečné funkcionality.

Úkol 25
V administračním rozhraní z minulých kapitol přidejte: validaci na straně klienta; dotaz při mazaní; a notifikaci o úspěšném vložení, která automaticky zmizí.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co umožňuje klientský JavaScript?
2. Co je to DOM?
3. Jak lze pomocí JavaScriptu měnit DOM?
4. Co jsou to události?

AJAX

Následující krátké kapitole si stručně představíme AJAX (Asynchronous JavaScript and XML). Jedná se o revoluční použití asynchronního JavaScriptu pro změnu části webové stránky, bez nutnosti jejího opětovného načtení.[130] Přestože se to dnes může zdát jako triviální záležitost, právě AJAX stojí za současnou podobou webových stránek a aplikací.

Princip na kterém je AJAX založen, je poměrně jednoduchý. Klient pomocí JavaScriptu pošle zprávu webovému serveru a asynchronně čeká až mu server odpoví.[131] Jakmile obdrží odpověď (například nová data), vykoná příslušnou operaci (například změnu v DOM). Jelikož je vše řízené JavaScriptem, není nutné provádět opětovné načtení stránky.

X v názvu AJAX představuje XML, které je využíváno pro přenos dat ze serveru na klienta v asynchronní komunikaci. Přestože je možné XML použít, dnes se běžněji využívá formát JSON případně prostý text.

[130] AJAX byl představen na konci 90. let minulého století společností Microsoft.

[131] Asynchronní čekání neblokuje vykonávání dalšího JavaSriptového kódu. Dodejme, že JavaScript ve výchozím stavu běží v jednom vlákně. Vykonávání instrukcí je tedy synchronní. K problematice asynchronního programování se později ještě vrátíme.

`XMLHttpRequest` objekt

Asynchronní požadavek je možné poslat dvěma způsoby. Starší způsob využívá `XMLHttpRequest` objekt. Tento způsob je velmi univerzální, široce podporovaný a umožňuje sledovat postup zpracování asynchronního dotazu na server. Komentovaný příklad následuje.

```
1  // vytvoření XMLHttpRequest objektu
2  const xhr = new XMLHttpRequest();  // xhr.readyState === xhr.UNSENT
3  xhr.open("GET", "http://localhost:8888/ajax/server.php");  // xhr.readyState === xhr.OPENED
4
5  // očekávaný typ odpovědi, nyní text, pro JSON: "json"
6  xhr.responseType = "text";
7
8  // obsluha zpracování odpovědi, // xhr.readyState === xhr.LOADING (stahování dat)
9  xhr.onload = () => {
10   // všechna data stažena, můžeme je zpracovat
```

```
11    if (xhr.readyState === xhr.DONE) {
12      if (xhr.status === 200) {
13        console.log(xhr.response);
14      }
15    }
16  };
17
18  // poslání požadavku
19  xhr.send(); // xhr.readyState === xhr.HEADERS_RECEIVED
```

Stav požadavku je možné sledovat pomocí vlastnosti `readyState`. Příslušné hodnoty jsou uvedeny v komentářích výše. Vlastnost `status` uchovává status HTTP požadavku.[132]

[132] Vždy je dobré testovat nejen dokončení AJAX requestu (řádek 11), ale i zda byla odpověď v pořádku (řádek 12).

Funkce `fetch()`

Novější, ale mírně omezenější způsob, využívá funkci `fetch()`, která je založena na *příslibech*. Zjevnou výhodou je snadnost použití.

```
// fetch
// vytvoří promis (příslib)
// pomocí, then() zpracujeme výsledek až je k dospozici
fetch("http://localhost:8888/ajax/server.php")
.then(r => r.text())
.then(console.log);
```

Následující kód rozšiřuje předchozí příklad o ošetření chyb.[133]

[133] Při asynchronní zpracování je třeba vždy ošetřovat výjimky.

```
fetch("http://localhost:8888/ajax/server.php")
.then(r => r.text())
.then(console.log)
.catch(console.error);
```

Pro úplnost uveďme ještě zpracování požadavku, která vrací data ve formátu JSON.

```
fetch("http://localhost:8888/ajax/server.php")
.then(r => r.json())
.then(j => j.results)
.then(console.log)
.catch(console.error);
```

Příslib můžeme vytvořit i explicitně pomocí `async()` a `await()`. Jedná se o alternativní zápis výše uvedeného.

```
1  const ajax = async() => {
2    try {
```

```
3     let res = await fetch("http://localhost:8888/ajax/
          server.php");
4     let results = await res.text();
5     console.log(results);
6   } catch (error) {
7     console.error(error);
8   }
9 }
10
11 ajax();
```

Přirozeně se nabízí možnost spojit přísliby a použití `XMLHttpRequest` objektu a docílit robustního asynchronního volání s možností sledování stavu.

```
1  const ajax = new Promise((resolve, reject) => {
2    const api = "http://localhost:8888/ajax/server.php";
3    const xhr = new XMLHttpRequest();
4    xhr.open("GET", api);
5    xhr.onload = () =>
6      xhr.status === 200
7        ? resolve(xhr.response)
8        : reject(xhr.status);
9    xhr.onerror = err => reject(err);
10   xhr.send();
11 });
12
13 ajax.then(r => console.log(r))
14 .catch(error => console.error(error));
```

V případě zpracování odpovědi ve formátu JSON stačí změnit řádek 7 na

```
resolve(JSON.parse(xhr.response).results)
```

Co přenášet v odpovědi?

Přirozeně se nabízí otázka co přenášet v HTTP odpovědi klientovi. Zde velmi záleží na konkrétní aplikaci. Uvažme následující situaci. Budeme chtít zobrazit seznam uživatelů, který získáme pomocí AJAX requestu. Jednou možností je, že součástí odpovědi bude i vizualizace tohoto seznamu. Například.

```
<table>
  <thead>
    <tr><th>Name</th><th>Surname</th></tr>
  </thead>
```

```
  <tbody>
    <tr><td>Jack</td><td>O'Neill</td></tr>
    <tr><td>Samantha</td><td>Carter</td></tr>
    <tr><td>Daniel</td><td>Jackson</td></tr>
  </tbody>
</table>
```

Tu potom můžeme snadno zobrazit. Uvedené řešení má ale mnoho nevýhod. Odpověď je zbytečně velká, jelikož se navíc přenáší vizualizace. Kromě toho jsou data spojena s vizualizací. Pokud bychom je chtěli použit v jiné podobě, bylo by to problematické. V neposlední řadě, pokud se změní vizualizace aplikace, tedy front-end, musí se (poněkud nelogicky) měnit i back-end. Z uvedených důvodů se běžně přenáší pouze potřebná data a jejich vizualizace je zajištěna na front-endu. Zjednodušený příklad zpracování a vizualizace následuje.[134]

[134] Jedná se pouze o princip řešení. Konkrétních implementací je celá řada.

```
//data ve formátu JSON
const data = '[{"name": "Jack", "surname": "O\Neill"}, {"
    name": "Samantha", "surname": "Carter"}, {"name": "
    Daniel", "surname": "Jackson"}]';

// parsování dat
const users = JSON.parse(json);

// vytvoření vizualizace (zjednodušené)
let result = "<table>";

for(user of users) {
  result += `<tr><td>${user.name}</td><td>${user.surname}</
      td></tr>`;
}

result += "</table>";

// zobrazení výsledné vizualizace
document.write(result);
```

CORS

Z bezpečnostních důvodů je ve většině webových prohlížečů implementován mechanizmus *Cross-Origin Resource Sharing* (CORS), který brání přístupu ke zdrojům z míst (domén), které se neshodují s místem (doménou), kde se daný zdroj nachází. Toto výchozí chování je možné upravit pomocí HTTP hlavičky. Příklad hlavičky povolující neomezený přístup ze všech domén je následující.[135]

[135] V mnoha případech představuje takovéto nastavení bezpečnostní riziko. Běžně se přístup omezuje na jednu či více konkrétních domén. V HTTP hlavičce je možné uvést pouze jednu doménu. Pokud jich potřebujeme uvést více, musíme server uzpůsobit tak, aby vracel odpovídající hlavičku.

```
Access-Control-Allow-Origin: *
```

V jazyce PHP je možné výše uvedenou hlavičku poslat pomocí funkce `header()`.

```
<?php
  header("Access-Control-Allow-Origin: *");
  // hlavička musí být uvedena před jakýmkoliv výstupem
?>
```

Dodejme, že standardně většina veřejných služeb povoluje přístup z jiných domén. Pro výměnu dat je ale zapotřebí HTTP protokolu. To znamená, že obvykle nelze testovat asynchronní dotaz na jiné servery bez použití lokálního či jiného webového serveru.

Příklad: Odeslání formuláře

Odeslání formuláře pomocí AJAX je velmi jednoduché. Uvažme následující jednoduchý formulář.

```
<form id="form" action="http://localhost/weba/ajax/skript.php" method="post">
  <label>Text:</label>
  <input type="text" name="message">
  <button>Submit</button>
</form>
```

Následuje JavaScriptový kód, který formulář odešle pomocí AJAX.

```
async function ajaxFormSubmit(event) {
  event.preventDefault();
  const data = new FormData(event.target);

  try {
    let request = await fetch(event.target.action, {method:
        form.method, body: data});

    let results = await request.text();
    console.log(results);

  } catch (error) {
    console.log(error);
  }
}

// navázání obsluhy událostí
var form = document.getElementById("form");
```

```
18 form.addEventListener("submit", ajaxFormSubmit)
```

Na řádku 2 dochází k zabránění odeslání formuláře prohlížečem. Na řádku 3 jsou data zabalena do speciálního objektu `FormData`, který slouží pro přenos formulářových dat.[136] Na řádku 6 je vytvořen a poslán AJAX požadavek.
Pro vyzkoušení stačí, aby volaný PHP skript obsahoval následující kód.

[136] Existuje více způsobů, jak data do AJAX požadavku uložit. Použití `FormData` je jeden z nich.

```
print_r($_POST);
```

Po úspěšném vykonání odeslání, by měl být v konzoli vidět následující výpis.

```
Array
(
    [message] => Data zadaná v input elementu
)
```

Analogicky je možné poslat i soubory. Nejprve upravíme formulář.

```
<form id="form" action="http://localhost/weba/ajax/skript.php" method="post">
  <label>File:</label>
  <input type="file" name="file">
  <button>Submit</button>
</form>
```

Pro vyzkoušení stačí, aby volaný PHP skript obsahoval následující kód.

```
print_r($_FILES);
```

Po úspěšném vykonání odeslání, by měl být v konzoli vidět následující výpis.[137]

[137] Výpis předpokládá, že byla poslána tento text ve formátu PDF.

```
Array
(
    [file] => Array
        (
            [name] => skripta-weba.pdf
            [full_path] => skripta-weba.pdf
            [type] => application/pdf
            [tmp_name] => /Applications/MAMP/tmp/php/
                phpE5lh22
            [error] => 0
            [size] => 3081902
        )
```

```
)
```

Na straně serveru by mělo dojít ke zpracování souboru. Soubor je uložen v dočasném úložišti, které zaniká po ukončení běhu skriptu. Pokud chceme soubor zachovat, musíme jej překopírovat na jiné místo.

Závěr

V této kapitole jsme představili AJAX a nastínili několik příkladů jeho použití. Pro úplnost uveďme ještě dvě zajímavé pokročilé technologie, které s AJAX spolupracují (vyžadují asynchronní zpracování): *Server sent event* a *WebSocket*.
První uvedená, umožňuje serveru, bez vyžádání klienta, posílat data klientovi. Toto je velmi zajímavé, jelikož to odbourává základní premisu v klient-server komunikaci (klient žádá server). Takto je možné například řešit live-feed, ve kterém se klient nemusí neustále dotazovat na nová data. Ty jsou mu poslána v o okamžiku, kdy jsou dostupná.
Druhá technologie implementuje komunikaci pomocí síťových soketů. Ve své podstatě se jedná o napojení na běžný síťový soket. Pomocí této technologie je možné zprovoznit rychlou obousměrnou komunikaci mezi klientem a serverem. Takto je možné řešit například chatovací místnosti nebo real-time přenos dat.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to AJAX?
2. Jak AJAX funguje?
3. Jak lze pomocí JavaScriptu měnit DOM?
4. Co jsou to události?

Úkol 26
Napište PHP skript povolující přístup ze dvou různých domén.

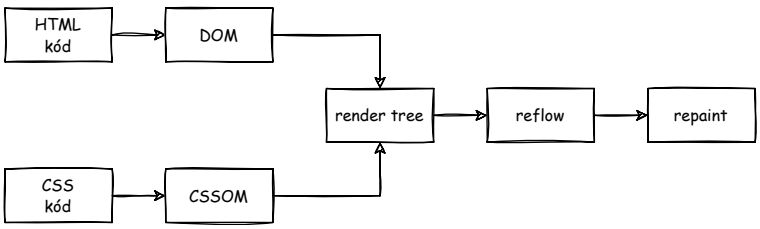
Úkol 27
V administračním rozhraní z předchozích kapitol upravte mazání uživatelů tak, aby bylo provedeno asynchronně (při mazání nedojde k načtení stránky stránky).

Úkol 28
Upravte REST API z předchozích kapitol tak, aby vracelo data ve formátu JSON. Pomocí AJAX načtěte tato data do administračního rozhraní.

React

V předchozích kapitolách jsme ukázali, jak vytvořit serverovou část aplikace (back-end) pomocí jazyka PHP. Vzhledem k možnostem JavaScriptu a webových prohlížečů se nabízí funkcionalitu z back-endu přesunout na klienta (front-end) a tím snížit celkové nároky na back-end. Samozřejmě ne vše je možné přesunout na klienta. Zejména ukládání a poskytování dat ze své podstaty musí být realizováno na serveru.
Pokud bychom začali psát aplikaci v JavaScriptu, narazíme na několik problémů, které jsou spojeny s DOM. Všechny pramení z neefektivity s jakou JavaScript s DOM pracuje. Obrázek 10 ukazuje postup renderování webové stránky.

Průvodce studiem
Přesun funkcionality ze serveru na klienta adresuje otázku škálovatelnosti aplikace. Jednoduše řečeno server bude mít méně práce, jelikož se vykonávání kódu přesune na klienta. Prakticky takto můžeme vytvářet aplikace, které zvládnout výrazně větší počtu uživatelů. Dodejme, že tento způsob škálování je nezávislý na použité architektuře.



Obrázek 10: Renderování webové stránky.

Z HTML kódu je vytvořen DOM, který již známe. Z CSS kódu je vytvořen *CSSOM* (CSS Object Model) což je analogie DOM pro CSS styly. Jedná se o stromovou reprezentaci elementů a k nim příslušných CSS pravidel, se kterou je možné pracovat pomocí JavaScriptu. Z DOM a CSSOM je sestaven *render tree*, který je kombinace těchto dvou stromů, jež slouží jako podklad pro renderování webové stránky. Následně se provede *reflow* (výpočet pozice každého elementu) a *repaint* (vykreslení elementů).
Jakákoliv změna na stránce ovlivní DOM nebo CSSOM nebo obojí. Obecně jsou změny dvojího typu: *reflow* a *repaint*. Reflow jsou změny, které ovlivňují layout stránky. Například změna pozice elementu nebo přidání nového elementu. Repaint jsou změny vizualizace. Například změna barvy. Obě změny jsou často vykonávány po sobě v pořadí reflow a repaint.[138]

[138] Při změnách typu repaint je možné přeskočit reflow fázi renderování webové stránky. Tyto změny jsou časově nejméně náročné.

Při změně musí prohlížeč přepočítat render tree, opět provést rozvržení stránky (pokud je třeba) a stránku vykreslit. To je ale časově náročné.[139]
V malých projektech se můžeme pokusit tento problém vyřešit tak, že se budeme snažit minimalizovat počet změn. Uvažme následující kód.

[139] Časté změny v DOM způsobují celkové zpomalení stránky, narušení plynulosti animací nebo nadměrné vybíjení baterie zařízení.

```
// neefektivní přístup
// dvakrát se provede reflow
element.style.width = '200px';
element.style.height = '200px';
```

Výše uvedený kód vyvolá dvě změny typu reflow, přičemž se provede nejprve jedna a následně druhá.[140] Pokud kód upravíme následovně

[140] Zde je nutné si uvědomit, že dojde dvakrát ke změně CSSOM, dvakrát se změní render tree a dvakrát se provede reflow a repaint.

```
// efektivnější přístup
// reflow se provede pouze jednou
element.style.cssText = 'width: 200px; height: 200px;';
```

dojte pouze k jedné změně typu reflow.
Při psaní komplexnějších aplikací, které budou často měnit svůj vzhled a interagovat s uživatelem si ale s výše uvedeným přístupem nevystačíme. Zde přichází na řadu koncepce *virtuální DOM* a *shadow DOM*.

Virtuální DOM

Virtuální DOM představuje vnitřní reprezentaci skutečného DOM.[141] Veškerá práce se odehrává právě s touto reprezentací. Při jakékoliv změně na stránce se děje následující: Vytvoří se nový virtuální DOM. Ten se porovná s původním virtuálním DOM a určí se rozdíly mezi nimi. Následně se vypočítají minimální změny, které bude nutné zanést do skutečného DOM. Poté jsou změny promítnuty do skutečného DOM.[142]
Shadow DOM umožňuje zapouzdřit část DOM do samostatně stojícího celku. Jeho hlavním účelem je vytvořit izolované prostředí pro HTML, CSS a JavaScriptový kód,[143] což může být užitečné především při psaní větších aplikací.
Další potíže při psaní JavaScriptové aplikace nám způsobuje nízkoúrovňový přístup JavaScriptu. I poměrně jednoduché úkony s DOM se zapisují velmi komplikovaně. Při zavedení virtuálního DOM se situace ještě více komplikuje. Zásadním konceptem je přechod z imperativního stylu programování na deklarativní, přesněji řečeno na reaktivní programování, které je variantou deklarativního paradigmatu.

[141] To je běžná nepřesnost, kterou budeme také používat. Ve skutečnosti je virtuální DOM reprezentací render tree.

[142] Pro větší efektivitu dochází ke kumulaci změn, které jsou do reálného DOM zaneseny současně.

[143] Umožňuje tedy vytvoření samostatně stojících komponent.

Viruální a shadow DOM a stejně tak reaktivní paradigma je adaptováno v mnoha JavaScriptových frameworcích, které jsou určeny pro tvorbu UI komponent nebo webových aplikací. V principu lze na webovou aplikaci nahlížet jako na sadu UI komponent. Příklady takovýchto frameworků jsou *React*, *Vue*, *Angular*, *Svelte*, *SolidJS* nebo *Lit*.
Přiblížit všechny hlavní frameworky a jejich filozofii je poměrně složitý a časově náročný úkol, proto se zaměříme na v současné době jednu z nejrozšířenějších knihoven, knihovnu React a to konkrétně ve verzi 18. Knihovnu React vytvořil v letech 2011–2013 Jordan Walke pod záštitou společnosti Facebook (dnes Meta).[144] Dnes je v Reactu napsána celá řada známých aplikací, mezi které kromě Facebooku patří i Instagram, Netlix nebo GitHub.

Průvodce studiem
Přestože vybíráme jednu konkrétní knihovnu, značná část zde prezentovaných principů je přenositelná a použitelná i v případě ostatních. Je ale nutné si uvědomit, že mezi uvedenými knihovnami jsou rozdíly a to zejména na technické úrovni.

[144] Původně měla knihovna řešit problémy, které jsme nastínili výše.

Použití knihovny

Knihovnu je možné používat dvěma způsoby. Prvním způsobem je integrace knihovny do již existující webové aplikace či stránky. V principu se takto React běžně používá, ale toto použití má své limitace. Druhým způsobem je vytvoření kompletní React aplikace. Nyní se zaměříme na první způsob. Druhý způsob si ukážeme později.
React má dvě základní komponenty, a to `React` a `ReactDOM`. Ty můžeme považovat za jádro knihovny. Obě mají klíčovou roli ve vytváření a manipulaci s uživatelským rozhraním. Následující kód ukazuje vložení vývojové verze knihovny z CDN.

```
<script src="https://unpkg.com/react@18/umd/react.development.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js"></script>
```

Pro finální produkci je nutné použít `react.production.min.js` a `react-dom.production.min.js`.[145]
Pro ladění React knihovny existuje řada vývojářských nástrojů, ty ale vyžadují komunikaci skrze HTTP. Je tedy výhodnější (a v některých případech nezbytné) při vývoji používat webový server.

[145] Rozdíly mezi uvedenými soubory se projevují především ve velikosti, výkonu a přítomnosti kontrol chyb. Vývojová verze obsahuje čitelný kód a dodatečné chybové kontroly vhodné pro vývoj, zatímco produkční verze je optimalizovaná, minifikovaná a minimalizuje chybové kontroly pro lepší výkon a menší velikost souboru v produkčním prostředí.

Nízkoúrovňový přístup

Základní metodou knihovny React je `ReactDOM.createRoot()`. Tato metoda zabezpečuje propojení s virtuálním a shadow DOM. Ve své podstatě vytvoří z vybraného elementu *React root element*, do kterého se budou vykreslovat jednotlivé komponenty. Je ustálené, že root element má atribut `id` nastavený na hodnotu `root`. Následuje příklad části HTML kódu, ve kterém je kořenový element.

```
<body>
  <h1>This title is pernament</h1>
  <div id="root"></div>
</body>
```

Nové elementy můžeme vytvořit pomocí metody `createElement()` a následně je zobrazit pomocí metody `render()`.

```
// nalezení React root elementu
const root = ReactDOM.createRoot(document.getElementById('root'));

// vytvoření nového elementu
const element = React.createElement("h2", {className: 'title'}, "This title is rendered");

// renderování nového elementu
root.render(element);
```

V porovnání s běžným JavaScriptem, je výše uvedený přístup výrazně méně nízkoúrovňový.

React komponenta

Základní stavební jednotkou aplikací vytvořených pomocí knihovny React je *React komponenta*. Ta představuje ucelený kus HTML, CSS a JavaScriptového kódu, používaný k vytváření uživatelského rozhraní. Zjevnou výhodou komponent je, že umožňují strukturovat aplikaci do samostatných částí. Další výhodou je znovupoužitelnost těchto částí. Celkově komponentní přístup výrazně zvyšuje čitelnost a přehlednost zdrojového kódu.
Z programátorského hlediska jsou komponenty *čisté funkce*,[146] které přijímají vlastnosti (*props*) a vrací výstup závislý na těchto vlastnostech. Vlastnosti jsou v tomto kontextu pouze pro čtení.
Následuje příklad React komponenty, která implementuje základ pro navigaci (seznam položek).[147]

```
1  //obsah navigace
2  const nav = ["Home", "Users", "Items", "Others"];
3
4  // komponenta
5  function Navigation(props) {
6    return React.createElement("ul",
7    {className: "navigation"},
8    props.map((item, i) =>
9      React.createElement("li", {key: i}, item)));
10 }
11
```

[146] Čisté funkce, anglicky *pure functions*, jsou funkce, které nemají vedlejší efekt a pro stejný vstup vždy vrací stejný výstup, který závisí pouze na vstupních parametrech dané funkce.

[147] V mapování na řádcích 8–9 je použitý povinný atribut `key`, který se v Reactu používá k identifikaci jedinečných prvků v seznamu a je důležitý pro aktualizaci a zobrazování komponent.

```
12 root.render(Navigation(nav));
```

Komponentu je možné vytvořit i pomocí arrow (anonymních) funkcí.

```
const Navigation = (props) => {
  return React.createElement("ul",
  {className: "navigation"},
  props.map((item, i) =>
    React.createElement("li", {key: i}, item)));
}
```

Komponenty vytvářené pomocí funkcí se označují jako *funkční komponenty*.
Komponenty lze v Reactu vytvářet také prostřednictvím tříd. Takto vytvořené komponenty se označují jako *třídní komponenty*. Od Reactu verze 18 se ale tento způsob vytváření komponent nedoporučuje. Navíc převést třídní komponentu na funkční je jednoduché. Vzhledem k tomu, že se v existujících projektech můžeme s třídními komponentami stále setkat, uvedeme si ukázku.

```
class Navigation extends Component {
    constructor(props) {
    super(props);
  }

  render() {
    return React.createElement(
      "ul",
      { className: "navigation" },
      this.props.items.map((item, i) =>
        React.createElement("li", { key: i }, item)
      )
    );
  }
}

root.render(React.createElement(Navigation, {items: nav}),
    null);
```

JSX

Výše představený nízkoúrovňový přístup názorně ukazuje práci Reactu s DOM. Pro řešení komplexnějších situací se ale, vzhledem ke stále velké nízkoúrovňovosti, tento přístup moc nehodí. V těchto případech React využívá JSX (Javascript and XML) syntaxi, který výrazně zjednodušuje popis React elementů.

Webové prohlížeče samy o sobě nerozumí syntaxi JSX, proto je nutné provést transpilaci do JavaScriptu.[148] Tento překlad pro nás zajišťuje knihovna *Babel*.[149] My si ji nyní pro jednoduchost integrujeme přímo v prohlížeči, ale takové řešení by nemělo být nikdy použito ve finální produkci. Pro prvotní seznámení je ale tento způsob dostačující. Dodejme, že vhodnější je použít nástroje, které umožní transpilaci kódu předtím, než je poslán webovému prohlížeče.

[148] Ve skutečnosti dochází k transpilaci na kód, který jsme uvedli výše, tedy do „řeči“ `React.createElement()`.

[149] `babeljs.io`

```
<script src="https://unpkg.com/@babel/standalone/babel.min.js"></script>
```

Skript s React kódem musí mít nastavený atribut `type` na hodotu `text/babel`, aby prohlížeč rozpoznal použití JSX syntaxe a byl schopen automaticky přeložit kód pomocí knihovny Babel.

Vytváření komponent pomocí JSX

S použitím knihovny `Babel` můžeme dříve definovanou komponentu přepsat následujícím způsobem.

```
1  function Navigation(props) {
2    return (
3      <ul className="navigation">
4        {props.map((item, i) => (
5          <li key={i}>{item}</li>
6        ))}
7      </ul>
8    );
9  }
10
11 root.render(Navigation(nav));
```

Uživatelsky definované komponenty je možné volat přímo v JSX. Mírnou úpravou předešlého získáme čisté řešení z pohledu Reactu. Uživatelsky definované komponenty mohou být volány přímo v JSX. Mírnou úpravou předchozího přístupu, kdy se komponenta vytvářela pomocí metody `root.render()` na řádku 11, přejdeme na moderní přístup s JSX, kde je komponenta volána přímo ve značce s atributy, což je výrazně přehlednější zápis.

```
function Navigation(props) {
  return (
    <ul className="navigation">
      {props.items.map((item, i) => (
        <li key={i}>{item}</li>
      ))}
    </ul>
  );
```

```
}

root.render(<Navigation items={nav} />);
```

V JSX jsou komponenty běžně zapisovány jako nepárové (prázdné) elementy s lomítkem na konci (`<komponenta />`). Rovněž je možné zapisovat komponenty jako párové elementy (`<komponenta> </komponenta>`), což se často využívá při vytváření nadřazených komponent.
Při přiřazování třídy je v JSX nutné použít atribut `className`, jelikož atribut `class` je vyhrazen pouze pro JavaScript. Výrazy JavaScriptu lze do JSX vložit pomocí složených závorek.

```
const flexClass = "flex";
const myElement = <div className={`${flexClass}`}>Flex
    content</div>;
```

Složené komponenty

Doposud jsme si ukázali práci s jednoduchými komponentami. V Reactu lze vytvořit i komponenty, které jsou složeny z několika dalších. Tímto způsobem lze velmi dobře strukturovat kód, což vede nejen k větší přehlednosti, ale hlavně k jejich případné znovupoužitelnosti. Ukázka složené komponenty následuje.

```
const root = ReactDOM.createRoot(
  document.getElementById('root')
);

// pomocná funkce
function formatDate(date) {
  return date.toLocaleDateString();
}

function Avatar(props) {
  return (
    <img className="avatar"
      src={props.user.avatarUrl}
      alt={props.user.name} />
  );
}

function UserInfo(props) {
  return (
    <div className="user-info">
      <Avatar user={props.user} />
      <div className="user-info__name">
```

```
        {props.user.name}
      </div>
    </div>
  );
}

// hlavni komponenta
function Comment(props) {
  return (
    <div className="comment">
      <UserInfo user={props.author} />
      <time className="comment__date">
        {formatDate(props.date)}
      </time>
      <p className="comment__text">
        {props.text}
      </p>
    </div>
  );
}

// data
const comment = {
  author: {
    name: 'Samantha Carter',
    avatarUrl: 'http://placekitten.com/g/42/42'
  },
  date: new Date(),
  text: 'Chevron Seven Locked!'
};

// renderování komponenty
root.render(
  <Comment
  author={comment.author}
  date={comment.date}
  text={comment.text} />
);
```

V případě, že chceme, aby byl výstup tvořený z více komponent, je nutné jej zabalit do fragmentu, který se v JSX zapisuje mezi <> a </>. Například.

```
function MyComponent() {
  return (
    <>
      <Header />
      <Content />
      <Footer />
```

```
    </>
  );
}
```

Bez použití fragmentu bychom museli použít nový, nadřazený element.

Stav komponenty

Komponenty si v Reactu mohou udržovat svůj vlastní stav. Zatímco dříve měly tuto funkcionalitu pouze třídní komponenty, od Reactu verze 16.8 si svůj stav mohou udržovat i funkční komponenty.[150]

[150] Vnitřní stav funkčních komponent se realizuje pomocí *react hooks*, které podrobněji představíme později.

```
1  import { useState } from 'react';
2
3  function MyFunctionalComponent() {
4    // vytvoření stavu a nastavení počáteční hodnoty
5    const [counter, setCounter] = useState(0);
6
7    return (
8      <div>
9        <p>Counter: {counter}</p>
10       <button onClick={() => setCounter(counter + 1)}>Increment</button>
11     </div>
12   );
13 }
```

Na řádku 5 je pomocí `useState` vytvořena proměnná `coutner`, která bude uchovávat stav komponenty. Proměnná je nastavena na hodnotu 0. Zároveň je vytvořena funkce `setCounter` umožňující tuto proměnou měnit.[151] Na řádku 9 je ukázáno čtení stavu a na řádku 10 změna stavu komponenty.

[151] A tím zároveň měnit stav komponenty.

Zpracování událostí

Zpracování událostí v Reactu se provádí pomocí inline funkcí v JSX. Každá komponenta může mít obslužné funkce pro události, jako je například kliknutí. Pro potlačení výchozího chování při vzniku dané události se používá metoda `preventDefault()`.

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
```

```
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

Vizualizace komponent

Komponenty je možné vizualizovat pomocí CSS. Příklad, kdy jsme komponent přiřazovali CSS třídu jsme již ukázali. Dále můžeme použít inline CSS. Zde je nutné si uvědomit, že CSS vlastnosti daného elementu jsou v JavaScriptu reprezentovány jako objekt. Vlastnosti tudíž nemohou mít v názvu pomlčku. Na místo toho se používá camelCase notace. Navíc je nutné použít `a` . Například pokud budeme chtít změnit vlastnost `background-color`, musíme nastavit vlastnost `backgroundColor` následovně.

```
<h1 style={{ backgroundColor: 'red' }}>Title</h1>
```

React aplikace

Doposud jsme knihovnu React používali poměrně omezeným způsobem. Vytvářeli jsme modulární aplikaci, přičemž těmito moduly byly právě React komponenty. JavaScript je ale ze své podstaty monolitický. To znamená, že je vždy načítán celý skript. To může být značně neefektivní, zejména u větších projektů.

JS moduly

S příchodem ES6 byly do JavaScriptu přidány *JS moduly*, které tvorbu modulárního kódu značně usnadňují.[152]
Při práci s moduly se používají klíčová slova `export` a `import`. Příkaz `export` označuje část kódu, kterou bude možné importovat do jiné části kódu, pomocí příkazu `import`. Uvažujme následující modul zapsáný v souboru `App.js`. Modul obsahuje jednu proměnou a jednu funkci.

[152] Dnes jsou JS moduly standardně podporovány všemi prohlížeči.

```
export const PI = Math.PI;

export function degreesToRadians(d) {return d * PI / 180;}
```

Uvedený modul načteme a použijeme následovně.[153]

[153] Jedná se o příklad použití na klientovi. Element `script` musí mít nastavený atribut `type` na hodnotu `module`.

```
<script type="module">
```

```
  import {PI} from './App.js';
  console.log(PI);
</script>
```

Výhodou uvedeného kódu je, že se nenačítá celý soubor `App.js`, ale pouze požadovaná část.[154]
Ruční práce s moduly je poměrně obtížná a typicky se pro správu modulů využívají různé nástroje jako například *Webpack*,[155] které umožňují sestavení aplikace programátorem.[156]
Abychom mohli využít všech možností, které knihovna React nabízí, musíme správně nakonfigurovat projekt, což zahrnuje nastavení `Babel` knihovny, `WebPack` a řady dalších nástrojů, které jsou pro vývoj React aplikací nezbytné. Provést správnou konfiguraci může být časově poměrně náročné, proto se často využívá předpřipraveného skriptu `create-react-app`.[157] Případně pro základní vyzkoušení je možné použít sandbox `codesandbox.io`.
Skript `create-react-app` usnadňuje přípravu vývojového prostředí pro React. Automaticky konfiguruje sestavování modulů, umožňuje snadnou integraci externích modulů z `npm` a provádí automatickou kontrolu kódu, včetně syntaxe a dodržování „best practices“ pomocí nástroje *ESLint*.[158] Dále se stará o automatické sestavení a spuštění celého projektu.
Pomocí následujících příkazů si vytvoříme aplikaci `hello-world`.

```
# instalace create-react-app
npm install -g create-react-app

# vytvoření a spuštění aplikace
create-react-app hello-world
cd hello-world
npm start
```

Ve výchozím stavu jsou všechny zdrojové kódy uloženy ve složce `src`. Nyní máme vše připravené na pokročilejší práci s knihovnou React.

React hooks

React hooks jsou významným rozšířením knihovny React. V následující části se podíváme na několik nejvýznamnějších.

`useState` hook

V předchozí části jsme si ukázali použití `useState` hooku, který se používá pro uchovávání stavu ve funkčních komponentách.[159]

[154] V našem příkladu není funkce `degreesToRadians()` načtena.

[155] `webpack.js.org`

[156] V naší ukázce je sestavení ponecháno na klientech.

[157] Obecně není doporučované ponechávat výchozí konfiguraci provedenou tímto skriptem, jelikož výsledné projekty obsahují příliš mnoho závislostí.

[158] ESLint je nástroj pro statickou analýzu JavaScriptového kódu.

[159] Často se v tomto stavu ukládají vstupní hodnoty, viditelnost prvků či jazykové mutace.

`useEffect` hook

Dalším často používaným hookem je `useEffect` hook. Tento hook umožňuje provádět vedlejší efekt. Nejčastěji se používá při načítání dat, odběru událostí, změně DOM nebo změně stavu komponenty. Kromě toho je v něm možné zavolat i takzvanou *cleanup funkci*, která je provedena před dalším spuštěním hooku. Následuje příklad použití.

```
// načtení hooks
import { useState, useEffect } from 'react';

// komponenta s hooks
export function Example() {
  // vnitřní stav komponenty (useState hook)
  const [count, setCount] = useState(0);

  // useEfect
  useEffect(() => {
    console.log (`You clicked ${count} times`);

    //  cleanup funkce
    return () => {console.log('Cleanup');};
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
      Click me
      </button>
    </div>
  );
}
```

`useEffect` hook akceptuje jako druhý volitelný parametr *dependency array*. Ten určuje proměnné při jejichž změně se hook spustí. Pokud není specifikován, bude hook spuštěn při každém renderování včetně toho prvního.[160] Pokud výše uvedený kód upravíme přidáním dependenci array `[count]` následovně

```
useEffect(() => {
  console.log(`You clicked times`);

    // cleanup funkce
  return () => {console.log("Cleanup");};
}, [count]); // dependeci array
```

bude `useEffect` hook spouštěn pouze při změně proměnné `count`.

[160] V aplikaci vytvořené pomocí skriptu `create-react-app` může docházet k volání `useEffect` hooku dvakrát za sebou. To je způsobeno použitím komponenty `React.StrictMode` v souboru `index.js`, která přidává několik kontrolních opatření.

`useContext` hook

Velmi užitečným hookem je `useContext` hook, který umožňuje komponentám získávat stav z *kontextu*. Kontext v Reactu je implicitní mechanismus sdílení hodnoty mezi komponentami bez nutnosti explicitního předávání pomocí `props`. Velmi zjednodušeně tento hook umožňuje vytvářet globální stav.[161] Následuje jednoduchý příklad použití `useContext` hooku.

[161] To je poněkud nepřesné, jelikož React komponenty jsou uspořádány ve stromu a jednotlivé vlastnosti mohou být přejímány pouze od rodičů v tomto stromu. `useContext` umožňuje vytvářet `props`, které jsou následně sdíleny v daném podstromu.

```
1  import React, { useContext } from 'react';
2
3  // barevné schéma
4  const themes = {
5    light: {
6      foreground: "#000000",
7      background: "#eeeeee"
8    },
9    dark: {
10     foreground: "#ffffff",
11     background: "#222222"
12   }
13 };
14
15 // vytvoření kontextu, vytváří komponentu
16 const ThemeContext = React.createContext(themes.light);
17
18 export function App() {
19   return (
20     <ThemeContext.Provider value={themes.dark}>
21       <Toolbar />
22     </ThemeContext.Provider>
23   );
24 }
25
26 function Toolbar(props) {
27   return (
28     <div>
29       <ThemedButton />
30     </div>
31   );
32 }
33
34 function ThemedButton() {
35   // získání kontextu
36   const theme = useContext(ThemeContext);
37   return (
38     <button style={{ background: theme.background, color:
         theme.foreground }}>
39     blah
40     </button>
```

```
41    );
42  }
```

V příkladu je vytvořeno barevné schéma (proměnná `themes`). Následně je vytvořen kontext (řádek 16), který které je v komponentě `ThemedButton` získán pomocí hooku `useContext`. Propagace kontextu do jednotlivých komponent se děje na řádcích 20–22.

Další hooky

Kromě výše uvedených hooků, které jsou nejdůležitější, se můžeme setkat ještě s hooky `useReducer` a `useMemo`. `useReducer` hook umožňuje vytvoření nového stavu z aktuálního. Například přepínání `true` a `false`, které je ukázáno níže.

```
const [checked, toggle] = useReducer(checked => !checked, false);
```

Hook `useMemo` umožňuje zapamatování hodnot vypočítaných při renderování. Ty následně nemusí být opět přepočítávány (pouze pokud dojde ke změně hodnot, ze který byly vypočítány). Tento hook slouží pro optimalizaci výkonu.

```
const result = useMemo(() => {
  console.log('Expensive calculation executed');
  // simulace náročného výpočtu
  return data.reduce((acc, val) => acc + val, 0);
}, [data]);
```

Hooky je možné volat pouze na top-level úrovni. Nesmí být v cyklech a podmínkách, mohou je ale obsahovat.[162] Hook by měl být volán pouze uvnitř komponenty. `ESlint` v `create-react-app` všechna tato pravidla hlídá. Kromě vestavěných hooks je možné vytvářet za pomoci existujících hooků, hooky vlastní.[163]

[162] Pokud by byly obsaženy v podmínce, mohli bychom kromě problémů s výkonem dostat i nekonzistentní stav komponenty.

[163] Tato problematika je ale nad rámec tohoto kurzu.

Získání dat

Z pohledu aplikace je každá komponenta samostatný celek, který zabezpečuje danou funkcionalitu. Například seznam uživatelů by mohl být komponenta. Tato komponenta by měla data nejen vizualizovat, ale také si je získat od serveru. Získání dat jsme již ukázali. Následuje příklad integrace `fetch()` do React komponenty.

```
export function GitUser({login}) { // destukturalizace props
  const [data, setData] = useState();

  useEffect(() => {
```

```
    if(!login) return;
    fetch(`https://api.github.com/users/${login}`)
      .then(r => r.json())
      .then(setData)
      .catch(console.error);
  }, [login]); // [] je dependency array

  if(!data) return loading...;

  if(data) {
    return {JSON.stringify(data, null, 2)}
  }
}

// vyzkoušet lze například <GitUser login="martin-trnecka"
    />
```

Dodejme, že hooků stejného typu může být i více a mohou se lišit v dependency array. Vždy je dobré složitější funkcionalitu rozložit do více hooků.
Při ladění výše uvedené komponenty si můžeme všimnout, že `fetch()` je volán dvakrát. To se děje pouze ve vývojovém módu. V produkční verzi je volání pouze jednou. Spuštění produkční verze lze provést například následovně.

```
npm install -g serve
npm run build # sestavení aplikace
serve -s build
```

Zaslání dat

Pro úplnost dodejme, že pomocí `fetch()` je možné data i poslat.[164]

[164] Jeden příklad jsme již uvedli.

```
const formData = new FormData();

formData.append("name", "Samantha");
formData.append("surname", "Carter");

fetch("/USER/ADD", {method: "POST", body: formData}).catch(console.error);
```

Uložení dat

Doposud jsme předpokládali, že odpovědnost za ukládání dat leží na back-endu webové aplikace. Nicméně, posunutím značné části funkcionality směrem k webovému front-endu může nastat situace,

kdy je výhodné uchovávat určitá data přímo na straně klienta. Je ale třeba mít na paměti, že tento způsob uložení dat přináší i svá úskalí. Hlavně je nutné zdůraznit, že na straně klienta nemůžeme zaručit integritu a perzistenci dat, což vyžaduje pečlivou manipulaci s daty. V JavaScriptu existují mechanismy pro ukládání dat na straně klienta. Běžně je možné použít lokální úložiště (storage) nebo webovou databázi (Indexed Database API) přímo v prohlížeči.

Průvodce studiem
Lokální úložiště i webová databáze jsou dostupné v rámci prostředí prohlížeče a poskytují možnost ukládat data přímo na zařízení uživatele. Je ale důležité si uvědomit, že takto uložená data jsou snadno přístupná uživateli. Nikdy by takto neměla být uchovávána citlivá data.

Local a session storage

Na straně klienta je možné data ukládat do dvou úložišť: *local strage* a *session storage*. Obě umožňují ukládat data ve tvaru klíč-hodnota. Local storage je omezné na 2–5 MB (v závislosti na prohlížeči) a uchovává data i po ukončení prohlížeče. Tato data mohou být uživatelem vymazána.[165] Session storage uchovává data pouze po dobu existence session,[166] velikost je omezena maximální pamětí, kterou lze přidělit procesu (tedy prohlížeči). Dodejme, že session storage nefunguje v anonymním režimu prohlížeče. Obsah storage je možné prohlížet ve webovém prohlížeči v nástrojích pro vývojáře. Ukázka práce s local storage následuje.

[165] Jedná se o data prohlížeče, může je tedy vymazat i operační systém, typicky se tak ale děje na přání uživatele.

[166] Uzavření prohlížeče data vymaže.

```
const d = new Date();
const user = {name: "Samantha", surname: "Carter"};

localStorage.setItem("user", JSON.stringify(user));
localStorage.setItem("time", d.getTime());

const savedUser = JSON.parse(localStorage.getItem("user"));
console.log(`${savedUser.name} ${savedUser.surname}`);
```

Jaká data je tedy vhodné ukládat na straně klienta? V praxi se jedná zejména o stav UI, nastavení uživatele, přihlašovací informace,[167] jazykové mutace či stav formulářů.

[167] Tím nemyslíme heslo, ale informace o tom, že je uživatel přihlášen.

Závěr

Ukázali jsme pouze základní koncepty a práci s knihovnou React. Velkým tématem je server-side rendering. Doposud jsme nechali generování webové aplikace na klientovi. V principu je možné nechat vygenerovat aplikaci webový server, který klientovi pošle pouze to co má být zobrazeno. Výhodou je, že kód na serveru může být totožný s kódem, který jsme doposud psali na straně klienta. Hlavní výhodou je ale rychlost. K této problematice se ještě později vrátíme.
Na knihovně React je vystavěna technologie *React native*, která umožňuje generovat nativní (přesněji řečeno hybridní) aplikaci pro různé

Kontrolní otázky
Odpovězte na následující otázky:

1. K čemu slouží knihovna React?
2. Co je to Viruální DOM?
3. Co je to React komponenta a jak ji lze vytvořit?
4. Co jsou JS moduly?
5. Co je to React hook?

platformy. Například iOS nebo Android. Obzvláště ve spojení s `Expo Go` nabízí velmi zajímavé možnosti vývoje.

Úkol 29
Vytvořte komponentu `Panel`, která bude obsahovat tlačítka. Příklad použití a vizualizace následuje. Snažte se o co nejuniverzálnější řešení.

```
// příklad použití
const buttons = [
  {text: "add", color: "#eee"},
  {text: "edit", color: "#eee"},
  {text: "delete", color: "#FF5733", inverse: true}
]

const buttons2 = [
  {text: "a", color: "#34eb8f"},
  {text: "b", color: "#1f8753", inverse: true},
  {text: "c", color: "#34eb8f"},
  {text: "d", color: "#1f8753", inverse: true},
  {text: "e", color: "#34eb8f"}
]

const buttons3 = [
  {text: "info", color: "#349beb", inverse: true},
]

root.render(
  <>
  <Panel shadow buttons={buttons} />
  <Panel buttons={buttons} />
  <Panel buttons={buttons2} />
  <Panel buttons={buttons3} />
  <Panel shadow buttons={buttons3} />
  </>
);

// vizualizace
```

Úkol 30
V administračním rozhraní z předchozích kapitol realizujte výpis logů v sekci `dashboard` jako React komponentu. Přidejte možnost smazat, konkrétní log a možnost vypsat pouze zadaný počet posledních logů (tuto informaci uchovávejte pouze

na straně klienta).[168]

[168] V local storage.

Node.js

Doposud jsme spouštěli JavaScriptový kód výhradně ve webovém prohlížeči. Ty poskytují běhové prostředí pro jazyk JavaScript. Asi nejznámějším běhovým prostředím je *V8 engine* používaný v Google Chrome.
V současné době existují i běhová prostředí, která nejsou spojena s webovým prohlížečem. Příkladem je dnes velmi populární open-source technologie Node.js, kterou si v této kapitole představíme. Ta poskytuje (alternativní) běhové prostředí založené na enginu V8. Node.js možňuje spouštět JavaScriptový kód i mimo webový prohlížeč, například na straně server.
V následující kapitole si postupně ukážeme vybrané klíčové vlastnosti technologie Node.js.[169]

Průvodce studiem
Popularita Node.js není příliš překvapivá. Vytvářet back-end pomocí stejné technologie, kterou je vytvořen front-end, zejména pokud front-end představuje větší část funkcionality, je velmi výhodné.

[169] Dodejme, že plnohodnotné představení a adaptace asynchronního paradigmatu, na kterém je Node.js založen, je nad rámec tohoto kurzu.

Práce s Node.js

Nejprve si vytvoříme jednoduchou Node.js aplikaci.[170] Uvažme kód uložený v souboru `Apj.js`.

```
console.log("Hello world");
```

Aplikaci spustíme příkazem níže.

```
node App.js
```

[170] Nejprve je nutné Node.js nainstalovat. Instalátor je dostupný na webových stránkách projektu `nodejs.org`.

Oproti klasickému vykonávání JavaScriptu ve webovém prohlížeči, které jsme představili v předchozích kapitolách, je zde několik rozdílů. Hlavním rozdílem je, že nemáme k dispozici objekt `window` a tudíž ani objekt `document`. V Node.js nemůžeme použít žádný kód, který adresuje funkcionalitu webového prohlížeče.[171] Dalším rozdílem je, že Node.js kód neinterpretuje, ale provádí JIT kompilaci. Je také dobré upozornit, že ne vše z ECMAScript je v Node.js podporováno.
Součástí Node.js je rozsáhlé API, které značně usnadňuje vývoj aplikací. Kromě API je možné využívat i moduly z `npm`.[172]

[171] Nejsou dostupná ani webová API.

[172] Ty je třeba vždy doinstalovat pomocí nástroje `npm`.

Dodejme, že ve výchozím stavu je Node.js vždy spouštěn v development módu. Pro finální produkci je třeba tento mód vypnout.[173]

[173] To se provádí nastavením proměnné `NODE_ENV` na hodnotu `production`.

Jedno vlákno

Node.js běží v jednom neblokujícím vlákně a používá asynchronní událostní řízení.[174] Tím je do značné míry vynuceno adaptování funkcionálního paradigmatu. Následuje příklad vytvoření triviálního webového serveru.[175]

[174] S asynchronním řízením jsme se již setkali v kapitole AJAX.

[175] Na prvním řádku je načten modul `http`, který obsahuje implementaci webového serveru.

```
1 // triviální webový server s počitadlem
2 const http = require('http');
3
4 // sdílená proměnná
5 var pocet = 0;
6
7 // vytvoření serveru a spuštění
8 http.createServer((req, res) => {res.end(`${++pocet}`)})
9     .listen(3000);
```

Po spuštění uvedeného kódu bude vytvořen webový server, který bude přijímat požadavky na portu 3000. Při vytvoření spojení je vykonána funkce, která inkrementuje proměnou `pocet`. Chování výše uvedeného serveru je odlišné něž chování, které jsme mohli pozorovat u serveru Apache. Jelikož server běží v jednom vlákně, bude proměnná `pocet` postupně narůstat. Server Apache (a jemu další podobné) pro každé spojení vytváří nové vlákno. To znamená, že každé nové spojení uvidí proměnou `pocet` nastavenu na 0.[176]

[176] Tento rozdíl je zcela zásadní z pohledu funkcionality. Běžné servery mají problémy zpracovat větší počty souběžných spojení. To je způsobené tím, že režie na udržování většího počtu souběžně běžících vláken se od určitého počtu vláken stává neúnosná. Běžné servery zvládnou kolem 10 tis. souběžných spojení. Asynchronní řízení, které je použito v Node.js tímto problémem netrpí.

Asynchronní událostní řízení

Při asynchronním programování se kód dělí na funkce, které se vykonávají asynchronně. Funkce jsou mezi sebou provázány pomocí volání. Nejprve si přiblížíme, jakým způsobem je vykonáván JavaScriptový kód. Běhová prostředí pro JavaScript používají událostní řízení. To funguje tak, že při vzniku události, je tato událost zařazena do *fronty událostí* a následně je z fronty vyzvednuta a obsloužena. Uvažme následující kód.

```
setTimeout(() => console.log("Hello!"), 0);
```

Jak již víme, tento kód vytvoří časovač, který je následně spuštěn. Vypršení časovače je událost a její obsluha je předaná funkce. Po vypršení časovače je událost přidána do fronty událostí. Následně je tato událost z fronty vyzvednuta a je spuštěna odpovídající funkce. Události mohou být obslouženy pouze pokud má běhové prostředí

čas spustit vykonání obsluhy. Následující kód demonstruje blokování vykonání obsluhy.[177]

[177] Obsluha je vykonána až po dokončení kódu na řádcích 3–5.

```
1 setTimeout(() => console.log("Hello"), 0)
2
3 let items = [1, 2, 3, 4, 5];
4 items.forEach(
5   (item) => {console.log(item)}
6 );
```

Pokud nahradíme řádek 5 následovně

```
(item) => {setTimeout(() => console.log(item), 0)}
```

nebude docházek k blokování.[178]

[178] Do fronty událostí jsou postupně zařazeny a obslouženy všechny události.

Při asynchrooním programování se snažíme vyhnout blokování obsluhy událostí. Běžně se používají tři základní vzory: *sekvenční*, *paralelní* a *omezený paralelní* (kombinace sekvenčního a paralelního). Sekvenční vzor je ukázán v kódu níže.

```
1  function f(callback, arg) {
2    setTimeout(() => {callback(Math.pow(2, arg));}, 0);
3  }
4
5  var items = [1, 2, 3, 4, 5];
6  var results = [];
7
8  function serial(item) {
9    if(item) {
10     f((result) => {
11       results.push(result);
12       return serial(items.shift());
13     }, item);
14   } else {
15     console.log(results);
16   }
17 }
18
19 serial(items.shift());
```

Funkce `f()` vytvoří časovač (řádek 2), po jehož vypršení dojde k vyvolání události a následnému spuštění obsluhy, předané jako parametr `callback` s argumentem `arg`. Funkci `f()` je na řádcích 10–12 předána funkce, která uloží výsledek `callback` funkce (řádek 11) a zavolá funkci `series()` (řádek 12). Celý kód je vykonáván sekvenčně, přičemž při každém volání funkce `serial()` se zpracuje výsledek z události a opět se zavolá `serial()` s dalším prvkem v poli `items`.[179]

[179] Přestože je kód vykonáván sériově, nedochází k blokování obsluhy událostí.

Paralelní vzor je zachycen v následujícím kódu.

```
items.forEach((item) => {
  f((result) => {
    results.push(result);
    if(results.length == items.length) {
      console.log(results);
    }
  }, item)
});
```

Jednotlivá volání funkce `f()` se vykonávají paralelně, přičemž není možné zajistit pořadí vykonávání.
Výše uvedené vzory jsou nízkoúrovňové. Node.js disponuje modulem `events`, který zápis znatelně usnadňuje. Příklad následuje.

```
const EventEmitter = require('events');

const eventEmitter = new EventEmitter();

// registrace události
eventEmitter.on('start', number => {
  console.log(`started ${number}`);
});

// vyvolání události
eventEmitter.emit('start', 42);
```

Pro úplnost dodejme, že metoda `.off()` odstraní obsluhu události.

Neblokující I/O

Veškeré vstupně výstupní operace jsou v Node.js neblokující.

```
const fs = require('fs');

fs.readFile('soubor.txt', (err, data) => {
  console.log('A');
});

console.log('B');
```

Výše uvedený kód nejprve vypíše `B` a poté `A`. Toto chování je odlišné od chování, které jsme mohli pozorovat u serveru Apache.[180]

[180] V případě Apache serveru (a jemu podobných) dojde k blokování výpočtu, dokud nebude dokončena vstupně výstupní operace. Nejprve se vypíše `A` a poté `B`.

Webový server

Hlavní výhoda technologie Node.js spočívá ve spojení webového a aplikačního serveru. Tím je možné dosáhnout velkého výkonu.

Typicky se Node.js využívá pro tvorbu (škálovatelného) webového back-endu, přičemž je možné snadno vytvořit monolitickou i mikro architekturu.
Následující kód vytváří webový server a získává veškeré (relevantní) informace z HTTP dotazu.

```
const http = require('http');

// request a response jsou stream
const server = http.createServer((request, response) => {
  const { headers, method, url } = request;

  let body = [];

  request.on('error', (err) => {
    console.error(err);
  });

  request.on('data', (chunk) => {
    body.push(chunk);
  });

  request.on('end', () => {
    body = Buffer.concat(body).toString();

    response.on('error', (err) => {
      console.error(err);
    });

    const responseBody = { headers, method, url, body };

    // HTTP status, hlavicka
    response.writeHead(200, {'Content-Type': 'application/
        json'})
    response.write(JSON.stringify(responseBody));
    response.end();
  });
});

server.listen(8080);
```

Tvorbu webového serveru usnadňuje modul `express`, který umožňuje snadnější zpracování HTTP požadavků a routování.[181]

[181] `expressjs.com`

```
const express = require('express');
const bodyParser = require('body-parser');

const app = express();
const port = 3000;
```

```
app.use(bodyParser.urlencoded({ extended: true }));
app.use(bodyParser.json());

// GET
app.get('/', (req, res) => {
  res.send(JSON.stringify(req.query));
});

// POST
app.post('/', (req, res) => {
  res.send(JSON.stringify(req.body));
});

app.listen(port, () => {
  console.log(`Listening on port ${port}`)
});
```

Komplexnější příklad využívající modul `expres` následuje.[182]

```
1  const express = require('express');
2  const bodyParser = require('body-parser');
3
4  const app = express();
5  const port = 3000;
6
7  app.use(express.json());
8  app.use(express.urlencoded({ extended: true }));
9
10 // log
11 const log = function(req, res, next) {
12   console.log(`${req.method} ${req.url}`);
13   next(); // předání řízení
14 }
15
16 // obsluha všech
17 const handler = function(req, res, next) {
18   const { controler, method, params } = req.params;
19
20   // vypsání POST
21   console.log(JSON.stringify(req.body));
22
23   res.send(`controler: ${controler}, method: ${method},
         method: ${params}`);
24   next();
25 }
26
27 app.all('/:controler/:method/:params(*)', [log, handler]);
28
```

[182] V uvedeném kódu jsou na řádku 27 ukázány pokročilé možnosti routování modulu `express`. Při volání serveru je nejprve volána funkce `log()`, která následně předá řízení funkci `handler()`. Do té jsou předány parametry z URL (řádek 18).

```
29 app.listen(port, () => {
30   console.log(`Listening on port ${port}`)
31 });
```

Napojení na databázi

Pro úplnost si ještě ukážeme jednoduchý příklad napojení na MySQL databázi.

```
var mysql = require('mysql2');

var connection = mysql.createConnection({
  host: 'localhost',
  port: '8889',
  user:'root',
  password:'root',
  database:'test'});

connection.connect();

var queryString = 'SELECT * FROM users';

connection.query(queryString, (err, rows, fields) => {
  if (err) throw err;
  for (let i in rows) {
    console.log(`ID: ${rows[i].id}, ${rows[i].name} ${rows[i
        ].surname}`);
  }
});

connection.end();
```

Závěr

V této krátké kapitole jsme představili základní koncepty a práci s Node.js. Dodejme, že k Node.js existuje modernější (ale ne až tak rozšířená) alternativa *Deno.js*.[183] Autorem je Ryan Dahl, který vytvořil Node.js. Hlavní myšlenkou Deno.js je odstranit komplikace v architektuře Node.js a přinést další vylepšení, zejména v oblasti rychlosti a bezpečnosti. Deno.js navíc nativně podporuje TypeScript a JSX syntaxi.

Úkol 31
Implementujte v Node.js REST API z úkolu 28.

Průvodce studiem
TypeScript je populární programovací jazyk vytvoření společností Microsoft, který je transpilován do jazyka JavaScript. Jak název napovídá rozšiřuje jazyk JavaScript o statickou typovou kontrolu, která je velmi výhodná zejména u větších projektů. Kromě typové kontrolo přidává i řadu dalších rozšíření.

[183] Pojmenování vzniklo přehozením pořadí písmen ve jméně Node.js. Deno.js je dostupné na stránce `deno.com`.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to Node.js?
2. Jaké jsou základní vlastnosti Node.js?

Next.js

V předchozích kapitolách jsme představili knihovnu React a technologii Node.js. Nyní se zaměříme na možnosti jejich propojení. To se může na první pohled jevit jako nesmysl, jelikož React je JavaScriptová front-end knihovna, zatímco Node.js je běhové prostředí pro JavaScript typicky používané na straně serveru (back-end). Zde je na místě si uvědomit, že veškerá funkcionalita React knihovny je naprogramována v JavaScriptu. Předpokládejme, že chceme spustit aplikaci vytvořenou v knihovně React. Nejprve se dotážeme na webový server, který nám pošle v odpovědi na náš dotaz HTML, CSS a JavaScript. Následně dochází k renderování webové aplikace a spuštění JavaScriptovéhé kódu. V rámci toho je spuštěna knihovna React a je vygenerováno uživatelské rozhraní.[184] Po skončení toto fáze je možné v prohlížeči vidět uživatelské rozhraní. Jelikož je ale renderování uživatelského rozhraní pomocí React knihovny pouze spuštění JavaScriptového kódu, nabízí se možnost jej přesunout na back-end. To se běžně označuje jako *server-side rendering*.

Samozřejmě ne vše je možné na back-end přesunout, jelikož změnou běhového prostředí ztrácíme přístup k prohlížeči.[185] JavaScriptový kód, který je možné vykonat nezávisle na běhovém prostředí se obvykle označuje jako *izomorfní kód*. Velká část renderování pomocí knihovny React je izomorfní. Knihovna React používá pro server-side rendering knihovnu *Next.js*, kterou si v následující kapitole představíme.[186]

Průvodce studiem
Dříve jsme ukázali, že se back-end přesunul na front-end. Nyní se opět front-end přesouvá na back-end. Zde se ale v principu jedná o jiný typ přesunu. Zatímco dříve se přesouvala celá funkcionalita, nyní přesouváme pouze část procesu renderování webové uživatelského rozhraní.

[184] Pomocí JavaScriptu je vygenerován HTML a CSS kód, přesněji řečeno je modifikován DOM.

[185] Připomeňme, že v Node.js nemáme přístup k objektu `window` (a tudíž ani k objektu `document`), `fetch` a řadě dalších webových API.

[186] Knihovna Next.js patří mezi doporučené nástroje při používání knihovny React.

Server-side rendering

V principu může server-side rendering nabývat různých podob.[187] Next.js provádí server-side rendering u každé stránky. To znamená, že Next.js vygeneruje HTML a CSS pro každou stránku předem a nenechává to na klientovi. Ke každé na serveru vyrenderované stránce je připojen minimalistický JavaScriptový kód, který má za úkol zabezpečovat interaktivitu stránky. Když je stránka načtena, je tento kód spuštěn. Tento proces se běžně nazývá *hydratace*.

Server-side rendering se může provádět při sestavení aplikace. To-

[187] Samotný název je poněkud nepřesný. Přesnější označení je *pre-rendering*. Toto nepřesné označení se ale běžně používá.

muto server-side renderingu se říká *static site generation*. Toto se typicky využívá pro statické aplikace, kde nepředpokládáme častou změnu obsahu.[188] Plnohodnotný server-side se provádí při každém požadavku na zobrazení aplikace.[189]
Dodejme, že při server-side renderingu můžeme provádět i různé optimalizace, například automaticky nahradit všechny obrázky moderními formáty, či provádět analýzu výkonnosti.

[188] Například e-shopy, blogy nebo dokumentace.

[189] Kromě uvedených se můžeme setkat s *incremental static site rendering*, kdy je aplikace renderována při sestavení a následně postupně přerenderována při každém požadavku.

Zprovoznění aplikace

Pro vytvoření projektu používající knihovnu Next.js je možné použít nástroj `create-next-app`.[190] Nejprve si vytvoříme aplikaci.

[190] Ten je nejprve nutné nainstalovat.

```
npx create-next-app aplikace
```

Výše uvedený příkaz spustí jednoduchého průvodce, který nás provede vytvořením aplikace.[191]
Aplikaci spustíme ve vývojovém módu následujícím příkazem.[192]

[191] Můžeme si například zvolit použití TypeScriptu, Tailwindu a dalších technologií.

[192] Ve vývojovém módu se změny ve zdrojovém kódu automaticky promítají do právě běžící aplikace.

```
npm run dev
```

Případně můžeme aplikaci sestavit a spustit příkazy.

```
npm run dev
npm run start
```

Struktura projektu

Jednotlivé stránky webové aplikace jsou React komponenty uložené v adresáři `pages`. Tyto komponenty jsou renderovány na straně serveru a posílány prohlížeči. Nyní vytvoříme novou stránku `test`. Ve složce `pages` vytvoříme soubor `test.js` s následujícím kódem.[193]

[193] Pokud při vytváření projektu pomocí `create-next-app` není zvolena doporučená možnost `useapprouter`, není složka `pages` vytvořena a ani není zajištěno routování, které si musíme vytvořit.

```
export default function Test() {
  return (
    <div>
      <h1>Test page</h1>
    </div>
  );
}
```

Next.js se automaticky postará o routování. Výše vytvořená stránka je dostupná přes URL `/test`.[194] Takto vytvořená stránka je statická.

[194] Dodejme, že přes URL `/` je dostupná stránka `index.js`.

Získání dat

Dynamicky generovanou stránku je možné vytvořit několika způsoby, jelikož máme více možností, kdy budeme stránku renderovat. Pro získání dat při sestavení aplikace (static site generation) slouží funkce `getStaticProps()`. Jednoduchý příklad získání dat následuje.

```
export async function getStaticProps() {
  const res = await fetch('https://jsonplaceholder.typicode.com/users/');
  const data = await res.json();

  return {
    props: {
      data,
    },
  };
}
```

Získaná data jsou dostupná v proměnné `data`. Kód níže ukazuje rozšíření naší testovací stránky o výpis těchto dat.

```
export default function Test({data}) {
  return (
    <div>
      <h1>Test page</h1>
      <ul>
      {data.map((user, index) => {
        return (
          <li key={user.id}>{user.name}</li>
        );
      })}
      </ul>
    </div>
  );
}
```

Dalším okamžikem, kdy můžeme získat data je server-side renderingu, tedy při požadavku na zobrazení aplikace. V tomto případě se ke získání dat používá funkce `getServerSideProps()`. Použití je analogické jako v případě `getStaticProps()`.

```
export async function getServerSideProps() {
  const res = await fetch('https://jsonplaceholder.typicode.com/users/');
  const data = await res.json();
```

```
  return {
    props: {
      data,
    },
  };
}
```

Doposud jsme získávali data na straně serveru. Data můžeme získat i na straně klienta, například následovně.

```
import { useState, useEffect } from 'react'

export default function getUsers() {
  const [data, setData] = useState(null);

  async function loadData() {
    const res = await fetch('https://jsonplaceholder.typicode.com/users/');
    const data = await res.json();
    setData(data);
  }

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

  if (!data) return <p>No data</p>

  return (
    <Users data={data}/>
  );
}

function Users({data}) {
  return (
    <ul>
    {data.map((user) => {
      return (
        <li key={user.id}>{user.name}</li>
       );
    })}
    </ul>
  );
}
```

Výše uvedený kód využívá React hooks. Jednoduší alternativou je použít SWR,[195] což je speciální React hook pro získávání dat poskytovaný Next.js.

[195] `swr.vercel.app`

```
import useSWR from 'swr';
const fetcher = url => fetch(url).then(r => r.json())

export default function Test() {
  const { data, error } = useSWR('https://jsonplaceholder.typicode.com/users/', fetcher);

  if (error) return <div>failed to load</div>;
  if (!data) return <div>loading...</div>;

  return (
    <Users data={data}>;
  )
}
```

Komponenty

Jednotlivé stránky mohou obsahovat uživatelské či vestavěné komponenty. Knihovna Next.js disponuje několika vestavěnými komponentami, který usnadňují vývoj. Příkladem je komponenta `Image` umožňuje vložit obrázek. Tato komponenta provádí různé optimalizace, například provede změnu velikosti obrázku, převede jej do formátu WebP a také předchází CLS (Cumulative layout shift).

```
import Image from 'next/image';

const YourComponent = () => (
  <Image
    src="/images/profile.jpg" // cesta k obrázku
    height={42} // výška
    width={42} // šířka
    alt="popisek"
  />
);
```

Další užitečné komponenty jsou `Head`, která slouží pro vložení hlavičky webové stránky nebo komponenta `Script` umožňující vložení externího JavaScriptového kódu.

Vizualizace komponent

Vizualizace komponent je v knihovně Next.js založena na CSS modulech. To, zjednodušeně řečeno, znamená, že vizualizace jednotlivých komponent je nezávislá na ostatních.[196]
Abychom vizualizovali komponentu, je nutné nejprve vytvořit CSS modul. V knihovně Next.js to uděláme jednoduše tak, že vytvoříme soubor s příponou `.module.css`. Například pro naši testovací

[196] Next.js podporuje SASS, Tailwind, a různé CSS-in-JS knihovny.

stránku vytvoříme soubor `test.module.css` ve složce `styles` s následujícím obsahem.[197]

[197] Případně jej můžeme uložit přímo u komponenty samotné.

```
.textColorPrimary {
    color: blue;
}
```

Připomeňme, že CSS moduly pracujíc s CSS jako s JavaScriptovými objekty, není tedy možné používat pomlčky v selektorech. Navíc není možné používat selektory neobsahující třídu.[198] Následující kód ukazuje načtení CSS modulu a jeho použití.

[198] Například selektor `h1` je neplatný.

```
import styles from '../styles/test.module.css';

export default function Test() {
  return (
    <div>
      <h1 className={styles.textColorPrimary}>Test page</h1>
    </div>
  );
}
```

CSS pravidla uvedená v souboru `global.css` jsou dostupná ve všech komponentách. Ve skutečnosti se jedná o běžný CSS soubor a je zde možné používat selektory bez omezení.

Dynamické routování

Pro úplnost zmiňme ještě jednu pokročilejší funkcionalitu knihovny Next.js. Při tvorbě webových aplikací často chceme zobrazit stránku dle zadaných parametrů. V případě static site generation je třeba tyto stránky předem vygenerovat. Next.js toto umožňuje pomocí funkce `getStaticPaths()`, která vrací parametry pro generování stránky. Jednoduchý příklad následuje.
Pro stránku, kterou budeme chtít zobrazovat, vytvoříme soubor `pages/users/[id].js` s následujícím obsahem.[199]

[199] Next.js zajistí automatické nahrazení `[id]` v názvu.

```
export default function Post({ data }) {
  return (
    <div>{data.name}</div>
   );
}

// vytvoříme stránky /user/1 a /user/2
export async function getStaticPaths() {
  return {
    paths: [
```

```
      {params: { id: '1' }},
      {params: { id: '2' }}
    ],
    fallback: false
  }
}

// předáme data do stránky
export async function getStaticProps({params}) {
  const res = await fetch(`https://jsonplaceholder.typicode.
      com/users/${params.id}`);
  const data = await res.json();

  return {
    props: { data }
  }
}
```

Výše uvedený kód vygeneruje stránky `user/1` a `user/2`.

Závěr

Stručně jsme představili knihovnu Next.js a server-side rendering. Projekt vytvořený pomocí Next.js je možné poměrně jednoduše nasadit na platformu *Vercel*.[200] Dodejme, že k této knihovně existuje několik alternativ. Například knihovna *Gatsby*, která se spíše zaměřuje na static site rendering nebo knihovna *Remix*.

[200] Knihovna Next.js je vyvíjena společností Vercel.

Úkol 32
Zprovozněte projekt pomocí `create-next-app` a vyzkoušejte příklady uvedené v této kapitole.

Kontrolní otázky
Odpovězte na následující otázky:

1. Co je to server-side rendering?
2. Jaké jsou možnosti server-side renderingu?

Mobilní a desktopové webové aplikace

V poslední kapitole tohoto textu se podíváme na možnosti využití webových technologií, které jsme představili dříve, pro tvorbu mobilních a desktopových aplikací. Ty v současné době v této oblasti poskytují širokou škálu možností.

Než ale začneme, položme si jednoduchou otázku: „Je pro uživatele přívětivější používat webovou aplikaci (či stránku) skrze webový prohlížeč, nebo použít nativní aplikaci, která zprostředkovává stejnou funkcionalitu?“ Mnohé studie prokazují, že v případě mobilních, ale i desktopových zařízení je nativní aplikace pro uživatele výrazně přívětivější z pohledu User Experience (UX). Nativní aplikace je nainstalovaná, má na ploše ikonu, typicky funguje v režimu offline a respektuje zásady uživatelského rozhraní daného operačního systému.[201]

Z výše uvedených důvodů je tedy velmi zajímavé kromě webové aplikace vyvinout i její nativní mobilní či desktopovou verzi. Samozřejmě vyvinout nativní aplikaci k existující webové aplikaci znamená aplikaci vytvořit (od zátku) pomocí jiných technologií. To může být velmi obtížné,[202] ale jen touto cestou je v současnosti možné dosáhnout na z pohledu UX nejlepší řešení. Je ale nutné zmínit, že nevýhodou tohoto řešení je platformová závislost. Aplikace například pro Android by se měla chovat a vypadat jinak než aplikace pro iOS, pokud má respektovat zvyklosti daného systému. Další možností je zvolit multiplatfromní přístup, kdy je aplikace vytvářená pro více platforem současně. To je pomocí webových technologií možné zařídit, nicméně to komplikuje vývoj (zejména samotný kód) a typicky to vede na ústupky v UX.[203] V mnoha případech je výhodné zvolit právě tento přístup. Vývoj multiplatformních aplikací je rychlý (nemusíme vyvíjet dvě různé aplikace).[204] Pojďme se nyní podívat na možnosti vývoje multiplatformních aplikací pomocí webových technologií.

Průvodce studiem
Dnes se běžně setkávám s desktopovými a mobilními aplikacemi, které jsou vytvořený pomocí webových technologií. Například Visual studiu code, Spotify, MS Teams a řada dalších.

[201] Zde je třeba si uvědomit, že UX je klíčový aspekt ovlivňující celkovou použitelnost aplikace.

[202] A také nákladné.

[203] Dodejme, že v lepším případě jsou tyto ústupky pouze drobné.

[204] S tím se pojí i náklady na vývoj a údržbu aplikace.

Multiplatformní mobilní aplikace

V současné době existují dva základní přístupy k tvorbě multiplatformní mobilních aplikací. První možností je takzvaný *hybridní vývoj*.[205] Ten je založen na velmi jednoduchého principu. Aplikace je tvořena nativní skořápkou, což je minimální nativní kód nutný pro běh aplikace. Veškerý obsah je zobrazen ve speciální komponentě `WebView`, která zpřístupňuje jádro webového prohlížeče. Celá aplikace je v podstatě jenom webová aplikace bežící v kontejneru tvořeném zmíněnou skořápkou, `WebView` a frameworkem, který zprostředkovává komunikaci se zařízení. Velkou výhodou výše popsaného je snadná přenositelnost, nevýhodou pak omezený přístup k nativním funkcím zařízení. Přesto je tento přístup poměrně populární. Příkladem technologie fungující na tomto principu je *Apache Cordova*.[206]

[205] Takto vytvořené aplikace se často označují jako *hybridní aplikace*.

Druhá možnost spočívá ve vytváření nativních komponent pomocí jazyka JavaScript. Veškerý kód je přeložen do nativního kódu dané platformy. Tento přístup nabízí o něco lepší výkon a přístup k nativním funkcím, na druhou stranu ale vyžaduje více platformově specifického kódu. Příkladem technologie využívající tohoto přístupu je *React Native*.[207]

[206] Případně její komerční verze *PhoneGap*.

To, jaký přístup zvolit velmi záleží na konkrétním projektu, jeho velikosti a cílech. Obecně ale můžeme pozorovat trend, kdy druhá možnost postupně utlačuje tu první.

[207] Základní představení technologie React Native je uvedeno v příloze tohoto textu.

Multiplatformní desktopové aplikace

Analogicky jako mobilní aplikace, můžeme vytvářet i desktopové aplikace. Tyto aplikace jsou snadno přenositelné mezi různými operačními systémy. Princip jejich fungování je velmi podobný jako u hybridního mobilního vývoje. Pro aplikaci je vytvořen nativní kontejner a UI aplikace je zobrazeno ve `WebView` komponentě. Dvě nejznámější technologie, které se vyžívají, jsou `Electron` a `NW.js`. Pojďme si ukázat instalaci a zprovoznění jednoduché aplikace pomocí technologie `Electron`.

Nejprve si vytvoříme kostru aplikace. Se základní konfigurací nám pomůže `npm`.

```
# vytvoření složky projektu
mkdir my-electron-app && cd my-electron-app

# inicializace
npm init

npm install --save-dev electron
```

Následně je třeba upravit vygenerovaný soubor `manifest.json`, aby bylo možné aplikaci snadno spustit. Obsah `manifest.json` následuje.

```
{
  "name": "my-electron-app",
  "version": "1.0.0",
  "main": "index.js",
  "scripts": {
    "start": "electron ."
  },
  "author": "Me",
  "devDependencies": {
    "electron": "^21.3.1"
  }
}
```

Nyní máme vše připraveni pro vývoj a můžeme začít programovat. Nejprve vytvoříme jednoduchou webovou stránku, která bude sloužit jako okno aplikace.

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8">
    <!-- Nastavení CSP: https://developer.mozilla.org/en-US/docs/Web/HTTP/CSP -->
    <meta http-equiv="Content-Security-Policy" content="default-src 'self'; script-src 'self'">
    <meta http-equiv="X-Content-Security-Policy" content="default-src 'self'; script-src 'self'"
        >
    <title>Hello from Electron!</title>
  </head>
  <body>
    <h1>Hello from Electron!</h1>
    <p>X</p>
  </body>
</html>
```

Následně vytvoříme skript, který vytvoří okno aplikace.

```
const { app, BrowserWindow } = require('electron');

// vytvoření okna
const createWindow = () => {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
  });
```

```
  win.loadFile('index.html'); // načtení obsahu okna
};

// ovládání a a manipulace s oknem
app.whenReady().then(() => {
  createWindow();

  // aplikace na MacOS mohou běžet i bez okna,
  // v takovémto případě znovuspuštění aplikace
  // pouze renderuje nové okno
  app.on('activate', () => {
    if (BrowserWindow.getAllWindows().length === 0) {
      createWindow();
    }
  });
});

// zavření okna na Windows a Linux,
// na MacOS (darwin) se nepoužívá
app.on('window-all-closed', () => {
  if (process.platform !== 'darwin') {
    app.quit();
  }
});
```

Aplikaci spustíme následujícím příkazem.

```
npm run start
```

Hlavní instance Elektron aplikace běží v omezeném Node.js, který má přístup k operačnímu systému. Z bezpečnostních důvodů je renderování UI prováděno mimo Node.js a stará se o něj jádro prohlížeče. K dispozici je celá řada Electron modulů (také známé jako *Elektron API*),[208] které zabezpečují různé (běžné) funkce a UI prvky, například `Notification` (notifikace), `Menu` (kontextové menu), `net` (zpracování HTTP požadavků) a mnoho dalších.

[208] `www.electronjs.org/docs/latest/api/app`

Propojení všech částí Elektron aplikace slouží *preload skript*, který nyní vytvoříme. Přidáme nový soubor `preload.js`, jenž bude sloužit jako most mezi front-endem a back-endem. Jeho obsah následuje.

```
const { contextBridge } = require('electron')

contextBridge.exposeInMainWorld('versions', {
  node: () => process.versions.node,
  chrome: () => process.versions.chrome,
  electron: () => process.versions.electron
  team: Array('Jack', 'Samantha', 'Daniel', 'Teal\'c')
```

```
})
```

Použití toho skriptu je nutné uvést v metodě `createWindow()`, kterou jsme uvedli dříve.

```
const path = require('path')

// vytvoření okna
const createWindow = () => {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      // potřebné kvůli správným adresám na různých OS
      preload: path.join(__dirname, 'preload.js'),
    },
  });

  win.loadFile('index.html'); // načtení obsahu okna
};
```

Nyní vytvoříme nový skript `render.js`, který bude do souboru `index.html` renderovat jména členů týmu. Tento skript poté musíme integrovat do HTML souboru.

```
const information = document.getElementById('info');

information.innerText = `This app is using Chrome (v${data.chrome()}),
                         Node.js (v${data.node()}),
                         and Electron (v${data.electron()})`;

const team = document.getElementById('team');
data.team.forEach((t) => team.innerHTML += `${t}`);
```

Propojení mezi aplikací a oknem je nutné realizovat pomocí IPC, jelikož se jedná o samostatné procesy. Rozšíříme `preload.js` a ukažme si jednoduchou obsluhu reagující na stisknutí tlačítka. Nejprve do souboru `index.html` přidáme tlačítko a do souboru `render.js` jeho obsluhu. Stisk tlačítka bude vyvolávat událost `ping`, na kterou by měla aplikace nějak zareagovat.

```
const button = document.getElementById('button');

button.addEventListener('click', async () => {
  const response = await window.data.ping();
  information.innerHTML = response;
});
```

Dále je nutné rozšířit skript `preload.js` o novou událost `ping`.

```
const { contextBridge, ipcRenderer } = require('electron')

contextBridge.exposeInMainWorld('data', {
  node: () => process.versions.node,
  chrome: () => process.versions.chrome,
  electron: () => process.versions.electron,
  team: Array('Jack', 'Samantha', 'Daniel', 'Teal\'c'),
    ping: () => ipcRenderer.invoke('ping'), // IPC
});
```

Posledním krokem je přidání obsluhy události `ping` do okna.

```
const { app, BrowserWindow, ipcMain } = require('electron');
const path = require('path')

// vytvoření okna
const createWindow = () => {
  const win = new BrowserWindow({
    width: 800,
    height: 600,
    webPreferences: {
      preload: path.join(__dirname, 'preload.js'),
    },
  });

  ipcMain.handle('ping', () => 'pong'); // obsluha
  win.loadFile('index.html'); // načtení obsahu okna
};
```

Tímto způsobem tedy dochází k základní komunikaci mezi oknem a samotnou aplikací. Pokud vše funguje, měl by výsledek vypadat tak, jak můžeme vidět na obrázku 11.

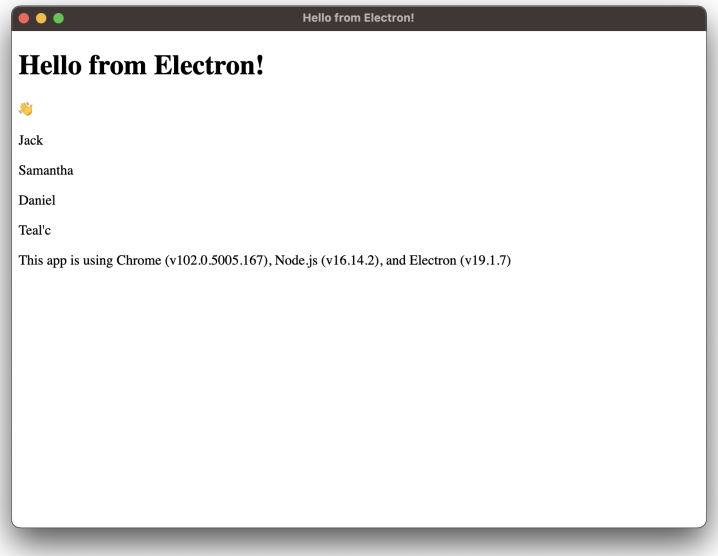
Progresivní webové aplikace

Díky neustále narůstajícím možnostem webových prohlížečů a jednotlivých webových technologií je možné na místo výše uvedených frameworků (interface mezi hardware a software) použít samotný webový prohlížeč.[209] Tímto přístupem je možné z běžné webové stránky/aplikace vytvořit mobilní nebo desktopovou aplikaci. Aplikace tohoto typu se označují jako *progresivní webové aplikace (PWA)*.[210]
PWA mají částečného nativní UI, lze je nainstalovat na plochu, fungují offline, při změně se samy aktualizují a umí v omezené míře

Průvodce studiem
Použít prohlížeč jako univerzální (multiplatformní) interface mezi hardware a software je velmi zajímavá myšlenka. Webový prohlížeče je dnes dostupný téměř na všech platformách.

[209] Přesněji řečeno jeho jádro.

[210] Ukázka jednoduché PWA aplikace je dostupná na stránce `github.com/martin-trnecka/pwa`.



Obrázek 11: Výsledná Elektron aplikace.

využívat nativních funkcí zařízení. Toho je docíleno pomocí souboru `Manifest.json` a technologie `ServiceWorker`.[211] `ServiceWorker` implementuje JavaScriptový kód, který slouží jako prostředník mezi webovou aplikací a webovým serverem. V případě, že je `ServiceWorker` podporován webovým prohlížečem, dojde nejprve k jeho stažení, instalaci a následné registraci a aktivaci. Následuje kód provádějící celý proces.

[211] `ServiceWorker` je webové API `developer.mozilla.org/en-US/docs/Web/API/Service_Worker_API`.

```
const registerServiceWorker = async () => {
//pokud lze ServiceWorkera zaregistrovat
if ("serviceWorker" in navigator) {
  try {
    const registration = await navigator.serviceWorker.
        register("/sw.js", {scope: "/"});

    // odchytávání stavu
    if (registration.installing) {
      console.log("Service worker installing");
    } else if (registration.waiting) {
      console.log("Service worker installed");
    } else if (registration.active) {
      console.log("Service worker active");
    }
  } catch (error) {
    console.error(‘Registration failed with ${error}‘);
  }
}};
```

```
registerServiceWorker();
```

`ServiceWorker` zajišťuje uložení celé, nebo alespoň části, aplikace do keše prohlížeče. Při výpadku připojení k síti `ServiceWorker` aplikaci poskytuje data z keše místo toho, aby je stahoval z webu.[212] Tímto je aplikacím umožněno fungovat i offline. Následuje velmi jednoduchý příklad kešování základních dat.

[212] Funguje tedy jako jakási proxy.

```
const addResourcesToCache = async (resources) => {
  const cache = await caches.open("v1");
  await cache.addAll(resources);
};

self.addEventListener("install", (event) => {
  // počkáme až je ServiceWorker nainstalovaný
  event.waitUntil(
    addResourcesToCache([ // přidáme soubory do cache
      "/",
      "/index.html",
      "/style.css",
      "/app.js",
      "/image-list.js",
    ]),
  );
});
```

Soubor `Manifest.json` popisuje metadata aplikace. Mezi ty patří například autor, název aplikace nebo ikonky, které jsou použity při instalaci.

```
{
  "short_name": "My app",
  "name": "My app is PWA",
  "icons": [
    {
      "src": "icons/icon-512.png",
      "type": "image/png",
      "sizes": "512x512"
    },
    {
      "src": "icons/maskable-icon-192.png",
      "type": "image/png",
      "sizes": "196x196"
    }
  ],
  "id":".",
  "start_url": ".",
  "display": "standalone",
```

```
  "theme_color": "#000000",
  "background_color": "#ffffff",
  "orientation": "portrait-primary",
  "description":"Simple PWA"
}
```

Přestože pojem progresivní webová aplikace není v současné době příliš známý, setkáváme s těmito aplikacemi na denní bázi. Stávají se totiž čím dál populárnějším řešením. Mezi nejznámější patří aplikace Spotify, MS Teams, Pinterest nebo Starbucks.

Závěr

Stručně jsme představili možnosti multiplatformního mobilního vývoje pomocí webových technologií. Dodejme, že v dnešní době jsou zde uvedené postupy stále více a více populární. Pro úplnost zmíníme ještě framework *Meteor*, který podobně jako React nativ, umožňuje vytvářet mobilní a desktopové aplikace.

Kontrolní otázky
Odpovězte na následující otázky:

1. Jaké jsou možnosti multiplatformního vývoje pomocí webových technologií?
2. Co je PWA?
3. Co je to Elektron?

Závěr

„It is the end of something simple and the beginning of everything else.“
Jennifer Baker

Tímto se dostáváme na samotný konec tohoto kurzu. Přestože se může jeho obsah jevit jako velmi nabitý, z řady důvodů jsme v mnoha případech pouze klouzali po povrchu dané problematiky. Hlavním cílem kurzu je poskytnou základy tvorby webových aplikací a ukázat stěžejní principy, na kterých jsou soudobé webové aplikace vystavěny. Pravdou ale je, že každý, kdo se chce této oblasti vývoje věnovat, se nyní ocitá na začátku dlouhé a fascinující cesty.

Příloha: CSS preprocesory

Technologii CSS představil v roce 1994 norský výzkumník Håkon Wium Lie.[213] O dva roky později, v roce 1996, se zrodil první standard této technologie, který byl zároveň prvním webovým standardem konsorcia W3C. Zatímco možnosti, a stejně tak i potřeby, vizualizovat HTML kód se od té doby velmi vyvinuly, samotná technologie CSS se příliš neměnila.[214]

Ačkoliv stálost a jednoduchost CSS technologie měly své výhody, v době největšího rozvoje vizualizace webových stránek,[215] se ukázaly jako zásadní omezení. Webový vývojáři potřebovali pokročilejší konstrukce, jako například proměnné, či funkce, které by jim usnadnili psaní CSS kódu. Reakcí na toto ze strany W3C nevyslyšené volání byl příchod *CSS preprocesorů*.

CSS preprocesory jsou nástroje, které umožňují transpilovat kód preprocesoru, zapsaný v jednoduchém programovacím jazyku (jazyk preprocesoru), do CSS kódu. Jazyk preprocesoru disponuje konstrukcemi pro řízení běhu programu, jako například podmínky a cykly, umožňuje vytvářet znovupoužitelné části CSS kódu pomocí funkcí a mixin, používat plnohodnotnou dědičnost (nikoliv pouze dědičnost na úrovni HTML) a významně rozšiřuje syntaktické možnosti zápisu CSS pravidel.

Preprocesory získaly obrovskou popularitu mezi webovými vývojáři a postupem času se staly běžnou webovou technologií.[216] Na toto později, poněkud neochotně, zareagovalo W3C a nastartovalo vývoj technologie CSS jako takové.[217] Postupně se v CSS objevily proměnné, funkce `calc()`, vnořování a další prvky, se kterými přišly preprocesory. Tímto rozvojem byl význam preprocesoru mírně oslaben. Přesto jsou dnes možnosti preprocesorů daleko nad možnostmi běžně podporovaného CSS. Navíc preprocesory umožňují generovat CSS kód, což pravděpodobně nikdy nebude pomocí běžného CSS možné.

[213] Håkon Wium Lie byl spolupracovníkem Tima Berners-Leeho, kterého netřeba představovat, a Roberta Cailliaua, který stojí za vznikem služby WWW.

[214] Tím máme na mysli, že postupně přibývaly nové vlastnosti, ale syntaxe zápisu CSS pravidel, se kromě několika drobných změn, prakticky nezměnila.

[215] Roky 2004–2014, tedy období vývoje HTML 5.

Průvodce studiem
Příchod CSS preprocesorů byl motivován zanedbaným vývojem CSS.

[216] Zajímavé je, že se ale nikdy nestaly základní technologií. Prohlížeče je nikdy nezačaly nativně podporovat.

[217] Nutno dodat, že W3C mělo poměrně omezené možnosti. Kompletně změnit CSS by bylo vzhledem k rozšířenosti této technologie velmi komplikované. Navíc CSS již bylo v této době rozděleno do nezávislých modulů, což komplikuje vývoj, jelikož neexistuje žádný modul popisující syntaxi CSS jako celek.

Preprocesor SASS

CSS preprocesorů existuje hned několik, mezi ty nejznámější patří například SASS, LESS a Stylus. Dříve se mezi nimi vedla dlouhá

válka o post nejlepšího, respektive nejpoužívanějšího preprocesoru, kterou po letech bojů vyhrál preprocesor SASS (Syntactically Awesome Style Sheets),[218] jenž je stručně představen v této příloze. Ostatní preprocesory se dnes prakticky nepoužívají.

[218] `sass-lang.com`

Instalace

Instalace preprocesoru SASS je poměrně jednoduchá. Existuje celá řada nástrojů a rozšíření do různých editorů, které SASS obsahují, případně je možné jej nainstalovat pomocí `npm`.

```
npm install -g sass
```

Preprocesor je možné volat pomocí API i z JavaScriptového kódu a provádět transpilaci na klientovi.[219] Pro vyzkoušení je také možné použít `sass-lang.com/playground` nebo CodePen.
Pro úplnost dodejme, že existuje několik implementací SASS: Ruby Sass, LibSass a Dart Sass. První je již nevyvíjená, druhá je stále funkční, ale je považována za zastaralou. V nových projektech by měla být používána Dart Sass, kterou budeme v tomto textu uvažovat.

[219] I když je to možné, není moc dobrých důvodů proč to dělat. Dodejme, že existují i neoficiální API pro jiné jazyky.

Základy SCSS

Preprocesor SASS umožňuje zapisovat kód preprocesoru pomocí dvou syntaxí: SASS, která je starší a dnes již nahrazena SCSS (Sassy CSS) syntaxí.
Syntaxe SCSS je plně kompatibilní se syntaxí CSS, ale navíc rozšiřuje stávající konstrukty CSS (například přidává nová at-pravidla) a přináší zcela nové konstrukty do CSS (například zanořování). Samotný jazyk CSS preprocesoru je poměrně jednoduchý a omezený programovací jazyk.[220] Obsahuje běžné operátory, umožňuje práci s řetězci a čísly. V následující části si jej stručně představíme.

Průvodce studiem
Stejné jméno syntaxe preprocesoru a preprocesoru často působí nedorozumění. Přestože je stále možné používat SASS syntaxi, pokud dnes někdo řekne, že používá SASS má na mysli SCSS syntaxi.

[220] Nenajdeme zde nic co bychom neviděli v jiných programovacích jazycích.

Proměnné

Jedním z hlavních přínosů CSS preprocesorů bylo zavedení proměnných, které v CSS znatelně chyběly. V SCSS se proměnné se zapisují s prefixem `$`.

```
SCSS kód

$color-main: #58508d;
$color-contrast: #ffa600;

.box {
  background-color: $color-main;
  color: $color-contrast;
```

```
}

.box--inverse {
  background-color: $color-contrast;
  color: $color-main;
}
```

Vygenerovaný CSS kód

```
.box {
  background-color: #58508d;
  color: #ffa600;
}

.box--inverse {
  background-color: #ffa600;
  color: #58508d;
}
```

Ve výše uvedeném kódu jsme deklarovali globální proměnou. Proměnné definované v deklaračním bloku jsou lokální v dané deklaračním bloku.[221]
Později se v CSS objevila nativní podpora proměnných. Z programátorského hlediska, jsou ale tyto proměnné poněkud odlišené od proměnných v běžných programovacích jazycích. CSS proměnné jsou deklarativní. Změna jejich hodnoty změní i předchozí použití dané proměnné. Proměnné v SCSS jsou imperativní, tedy změna jejich hodnoty neovlivní jejich předchozí použití. Dalším zásadním rozdílem je, že preprocesor při transpilaci proměnné nahrazuje jejich hodnotami. V případě nativních CSS proměnných, jsou tyto proměnné uloženy ve výstupním CSS.[222]

[221] Pro změnu globální proměnné z lokálního prostředí je možné použít příznak `!global`.

[222] Prohlížeč je tedy musí zpracovat. V případě použití preprocesoru má prohlížeč méně práce, což má za následek zvýšení rychlosti. Ve své podstatě zde můžeme vidět analogii server-side renderingu.

Komentáře

V SCSS je možné používat kromě běžných CSS komentářů (zapisovaných pomocí `/* */`) i komentáře uvozené symboly `//`. Běžné CSS komentáře budou ve výsledném CSS kódu, zatímco komentáře začínající symboly `//` budou odstraněny.[223]

[223] Jedná se o komentáře zdrojového kódu preprocesoru, ve výsledném CSS nemají význam.

SCSS kód

```
// komentář pro preprocesor
/* komentář pro CSS */
```

Vygenerovaný CSS kód

```
/* komentář pro CSS */
```

Interpolace

Výsledek vyhodnocení SCSS kódu může být snadno předán do CSS pomocí `#{}`. Toho se využívá zejména v případě, kdy chceme CSS vlastnost určit pomocí proměnné tak, jak je ukázáno níže.

SCSS kód

```scss
$where: top;

.box {
  #{$where}: 1em;
}
```

Vygenerovaný CSS kód

```css
.box {
  top: 1em;
}
```

Nesting

Jedním ze zásadních vylepšení CSS syntaxe, které přinesly CSS preprocesory, je *nesting* (vnořování).[224] Ten umožňuje zanořovat CSS pravidla. Zanoření umožňuje zkrátit zápis CSS selektorů a zpřehlednit kód.[225] Kód níže generuje (klasické) třídy pro navigační menu.

[224] Nesting je považován za jeden z největších přínosů CSS preprocesorů.

[225] Související části kódu jsou do sebe zanořeny.

SCSS kód

```scss
.nav {
  height: 3em;

  ul {
    margin: 0;
    padding: 0;

    li {
      list-style-type: none;

      a {
        text-decoration: none;
      }
    }
  }
}
```

Vygenerovaný CSS kód

```
.nav {
  height: 3em;
}

.nav ul {
  margin: 0;
  padding: 0;
}

.nav ul li {
  list-style-type: none;
}

.nav ul li a {
  text-decoration: none;
}
```

Pomocí *rodičovského selektoru*, který se zapisuje symbolem `&`, je možné se odkázat na rodiče v hierarchii, která je vytvořena zanořením. Například.

SCSS kód

```
a {
  text-decoration: underline;

  &:hover {
    text-decoration: none;
  }
}
```

Vygenerovaný CSS kód

```
a {
  text-decoration: underline;
}

a:hover {
  text-decoration: none;
}
```

Při použití je rodičovský selektor nahrazen rodičem. Je tedy možné jej použít i jako hodnotu funkce.

SCSS kód

```
.text-normal {
  color: black;

  :not(&) {
    color: red;
  }
}
```

Vygenerovaný CSS kód

```
.text-normal {
  color: black;
}

:not(.text-normal) {
  color: red;
}
```

Rovněž je možné rodičovský selektor použít pro vygenerování tříd dle BEM metodiky.[226]

[226] Odpůrci CSS preprocesorů často vytýkají nepřehlednost kódu v případě, že je použit nesting a CSS metodiky. Částečně jim lze dát za pravdu, ale spíše je to záležitost vkusu.

SCSS kód

```
.nav {
  & {
    /* společné pro všechny navigace */
  }

  &--secondary {
    /* specifika pro sekundární navigaci */
  }

  &__item {
    /* položka navigace */

    &--active {
      /* specifika pro aktivní položku navigace */
    }
  }
}
```

Vygenerovaný CSS kód

```
.nav {
  /* polečné pro všechny navigace */
}

.nav--secondary {
```

```
  /* specifika pro sekundární navigaci */
}

.nav__item {
  /* položka navigace */
}

.nav__item--active {
  /* specifika pro aktivní položku navigace */
}
```

Nesting je možné využít i pro zkrácení zápisu více vlastností se stejným prefixem. Syntaxe se ale v tomto případě mírně liší.

SCSS kód

```
body {
  font: {
    family: Arial;
    size: 1em;
    style: italic;
  }
}
```

Vygenerovaný CSS kód

```
body {
  font-family: Arial;
  font-size: 1em;
  font-style: italic;
}
```

Pro úplnost dodejme, že je možné pomocí at-pravidla `@at-root` potlačit nesting a zpřístupnit hlavní, nezanořenou úroveň.

Dědičnost

SCSS syntaxe umožňuje používat dědičnost.[227] Pomocí at-pravidla `@extend` je možné zdědit CSS vlastnosti.

[227] Označení dědičnost je v tomto případě mírně zavádějící, ale běžně se používá.

SCSS kód

```
.box {
  border-width: 1px;
  border-style: solid;
  border-color: black;
}

.box--emphasize {
```

```
  @extend .box;
  border-color: red;
}
```

Vygenerovaný CSS kód

```
.box, .box--emphasize {
  border-width: 1px;
  border-style: solid;
  border-color: black;
}

.box--emphasize {
  border-color: red;
}
```

V některých situacích je žádoucí, aby ve vygenerovaném CSS nefigurovala pravidla, ze kterých bylo děděno. K tomu je možné využít symbol %.[228]

[228] Uvedený kód je pouze demonstrační. Z pohledu BEM metodiky by bylo lepší jej řešit bez dědičnosti.

SCSS kód

```
%box {
  border-width: 1px;
  border-style: solid;
}

.box--normal {
  @extend %box;
  border-color: black;
}

.box--emphasize {
  @extend %box;
  border-color: red;
}
```

Vygenerovaný CSS kód

```
.box--emphasize, .box--normal {
  border-width: 1px;
  border-style: solid;
}

.box--normal {
  border-color: black;
}
```

```
.box--emphasize {
  border-color: red;
}
```

Mixiny

Mixiny umožňují vytvářet opakovaně použitelné části CSS kódu. Pro definici mixiny se požívá at-pravidlo `@mixin` a pro její použití `@include`.

SCSS kód

```
@mixin flex-center {
  display: flex;
  justify-content: center;
  align-items: center;
}

.box {
  @include flex-center;
}
```

Vygenerovaný CSS kód

```
.box {
  display: flex;
  justify-content: center;
  align-items: center;
}
```

Mixiny mohou mít i parametry, kterým je možné určit výchozí hodnoty. Argumenty je možné předávat pozičně, ale i jménem parametru.[229]

[229] Analogicky jako tomu je v jazyce Python.

SCSS kód

```
@mixin flex-center($flex-direction: row) {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: $flex-direction;
}

.box--row {
  @include flex-center();
}

.box--column {
```

```
  @include flex-center(column);
}

.box--column {
  @include flex-center($flex-direction: column);
}
```

Vygenerovaný CSS kód

```
.box--row {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: row;
}

.box--column {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
}

.box--column {
  display: flex;
  justify-content: center;
  align-items: center;
  flex-direction: column;
}
```

Průvodce studiem
Na první pohled je patrné, že mixiny je možné použít stejně jako at-pravidlo `@extend`. Obecně se `@extend` používá v případě, že je žádoucí explicitně určit závislosti mezi částmi kódu.

Mixiny mohou mít i proměnlivý počet argumentů. Příklad ukážeme později.

Funkce

Stejně jako v jiných programovacích jazycích, je možné v SCSS používat funkce. Pro definici funkce se požívá at-pravidlo `@function`.[230]

SCSS kód

```
1 $base: 1em;
2
3 @function add-m($x: 1em) {
4   @return $base + $x
5 }
6
7 .box {
8   margin: add-m(2em);
9 }
```

[230] Upozorněme, že SASS neumí počítat s různými jednotkami. Pokud bychom volání funkce na řádku 8 nahradili `add-m(2em)` dojde k chybě. V CSS nativní funkce `calc()` takové omezení nemá. V SASS je možné toto omezení částečně obejít pomocí vestavěných modulů.

Vygenerovaný CSS kód

```
.box {
  margin: 3em;
}
```

V některých případech je žádoucí, aby argumenty předané mixinám a funkcím splňovali určité podmínky. At-pravidla `@error` a `@warn` umožňují generovat chybu nebo varování při transpilaci. Pro testování, zda je, či není splněna daná podmínka slouží podmínky.

Konstrukce pro řízení běhu programu

SASS umožňuje využívat běžné konstrukce pro řízení běhu programu jako jsou podmínky a cykly. Pro zápis podmínek se využívají at-pravidla `@if` a `@else`. Příklad použití následuje.

SCSS kód

```
@mixin theme-colors($light-theme: true) {
  @if $light-theme {
    background-color: $light-background;
    color: $light-text;
  } @else {
    background-color: $dark-background;
    color: $dark-text;
  }
}
```

V SCSS existuje analogie ternárního operátoru.[231]

[231] V ukázce se využívá toho, že pokud je v SCSS nastavena vlastnost na hodnotu `null`, není předána do výsledného CSS.

SCSS kód

```
$show-border: false;

.box {
  padding: 1em;
  border: if($show-border, 1px solid black, null);
}
```

Vygenerovaný CSS kód

```
.box {
  padding: 1em;
}
```

Pro zápis cyklů se používají at-pravidla `each`, `for` a `while`. Příklad foreach cyklu následuje.

SCSS kód

```scss
$icons: "up", "right", "bottom", "left";

@each $icon in $icons {
  .icon--#{$icon} {
    background: url($icon+".png");
    width: 2em;
    height: 2em;
  }
}
```

Vygenerovaný CSS kód

```css
.icon--up {
  background: url("up.png");
  width: 2em;
  height: 2em;
}

.icon--right {
  background: url("right.png");
  width: 2em;
  height: 2em;
}

.icon--bottom {
  background: url("bottom.png");
  width: 2em;
  height: 2em;
}

.icon--left {
  background: url("left.png");
  width: 2em;
  height: 2em;
}
```

Použití cyklu for ukážeme na slíbeném příkladu mixiny s volitelným počtem parametrů.[232]

[232] Funkce `nth()` je součástí modulu `sass:list`.

SCSS kód

```scss
@mixin sizes($width, $selectors...) {
  @for $i from 0 to length($selectors) {
    #{nth($selectors, $i + 1)} {
      width: $width * ($i+1);
    }
```

```
  }
}

@include sizes(20ch, ".small", ".medium", ".large");
```

Vygenerovaný CSS kód

```
.small {
  width: 20ch;
}

.medium {
  width: 40ch;
}

.large {
  width: 60ch;
}
```

Vestavěné moduly

SASS obsahuje několik vestavěných modulů, které implementují užitečnou funkcionalitu. Nalezneme zde například modul pro práci s barvou, řetězci, selektory nebo modul implementující základní matematické operace. Moduly se načítají pomocí at-pravidla `@use`.[233]

[233] V ukázce je použito at-pravidlo `@debug`, které vypíše výsledek SCSS výrazu do konzole.

SCSS kód

```
@use 'sass:math';

@debug math.ceil(4.2);
```

Podrobný popis všech vestavěných modulů je k dispozici v oficiální dokumentaci preprocesoru SASS.[234]

[234] `sass-lang.com/documentation`

Organizace kódu

SCSS kód je možné rozdělit do více souborů a ty následně vložit pomocí pravidel `@import`, `@forward` nebo `@use`. `@import` přidává SCSS kód k existujícímu kódu.[235] `@forward` umožňuje načíst SCSS kód jako modulu pomocí `@use` a případně připojit namespace v podobě prefixu. Detailní rozbor použití `@forward` a `@use` již přesahuje základní znalosti SCSS a ponecháme jej na samostudium.

[235] Jedná se o analogii CSS at-pravidla `@import`, ale na rozdíl od něj je přidání kódu prováděno při transpilaci.

Závěr

Stručně jsme představili základní možnosti preprocesoru SASS. Pro úplnost ještě zmíníme zajímavou alternativu CSS preprocesorů a to stylované komponenty.[236] Zjednodušeně řečeno stylované komponenty umožňují generovat CSS kód pomocí JavaScriptu. JavaScript nahrazuje omezený kód preprocesoru, čímž získáme výrazně větší možnosti generování CSS. Dodejme, že se ale nejedná o technologii, která je stejně rozšířená jako CSS preprocesory.

[236] `styled-components.com`

Příloha: React Native

Po vzniku populární knihovny React, která výrazně usnadnila tvorbu webového front-endu, se objevil podobný požadavek na urychlení a zjednodušení vývoje mobilních multiplatformních aplikací. V návaznosti na tuto potřebu představila společnost Facebook (dnes Meta) v roce 2015 technologii React Native, která vychází z knihovny React. Oproti předchozím technologiím umožňuje React native vývoj výkonných mobilních aplikací, které se vzhledově velmi podobají aplikacím nativním. Toho je dosaženo překladem JavaScriptového kódu, ve kterém je aplikace vyvíjena, do nativního kódu cílové platformy. Díky tomu jsou zpřístupněny i nativní funkce zařízení, ke kterým aplikace bežící ve webovém prohlížeči nemají přístup. Pro lepší představu si technologii React Native stručně představíme.

Zprovoznění

Vytvoření první jednoduché aplikace v React Native je velmi snadné a lze jej realizovat dvěma způsoby. První a jednodušší metoda spočívá ve využití mobilní aplikace *ExpoGo*, která je doporučená pro začátečníky v oblasti mobilního vývoje.[237] Tuto metodu si pro svou rychlost a jednoduchost zprovoznění představíme. Alternativně je také možné použít React Native CLI, avšak pro tuto variantu je nutné mít nainstalované a nakonfigurované vývojové prostředí Android Studio nebo Xcode, což je proces vyžadující více času.
Při využití `ExpoGo` je nutné mít nainstalován Node.js a mobilní zařízení. Kostru aplikace s již nainstalovanými závislostmi vytvoříme a spustíme pomocí následujících příkazů.

[237] Expo je sada nástrojů, knihoven a dalších služeb vyvinutých pro React Native. Expo velmi zjednodušuje vývoj a jeho největší výhodou je, že aplikace, které jej používají je možné spustit ihned bez dalších konfigurací projektu.

```
# vytvoření projektu
npx create-expo-app AwesomeProject

# otevření a spuštění aplikace
cd AwesomeProject
npx expo start
```

Nyní máme vytvořenou první aplikaci a spuštěn vývojový server. V příkazové řádce se zobrazí QR kód, který použijeme při zprovoznění aplikace na našem zařízení.
Pro zprovoznění aplikace na vlastním zařízení je nutné provést několik kroků. Nejprve musíme připojit telefon do stejné sítě, ve které je připojen počítač, na kterém vyvíjíme. Následně je potřeba si na zařízení nainstalovat aplikaci *ExpoGo*, která je dostupná na GooglePlay a AppleStore. Nyní je potřeba vyfotit QR kód z příkazové řádky. V případě Android zařízení je potřeba vyfotit QR kód z aplikace ExpoGo. U iOS postačí použití nativního fotoaparátu. Po naskenování a načtení již máme aplikaci dostupnou na našem zařízení. Velkou předností tohoto přístupu je, že se aplikace automaticky načte s každým uložením změn.

Základní komponenty a stylování

Vzhledem k tomu, že React Native vychází z knihovny React, mají spoustu společných konceptů. Základní rozdíl však spočívá v používaných značkách a stylování. V React Native se k tvorbě layotu používají základní komponenty `<View>`, `<Text>`, `<Image>`, `<TextInput>` a `<ScrollView>`, které odpovídají základním prvkům UI používaných v mobilním vývoji. Základní komponentou je `<View>`, která slouží jako kontejner. Následuje příklad jednoduché komponenty.

```
export default function SimpleComponent(){
  return (
    <View>
      <Text>Hello, React Native!</Text>
    </View>
  );
};
```

Pro definování stylů se používá objekt `StyleSheet`. Tento objekt zajišťuje překlad stylů do nativního kódu. V menších projektech je zvykem využívat inline stylování, případně externích knihoven. Ukažme si příklad inline stylování komponenty z předchozího příkladu.

```
export default functionSimpleComponent(){
  return (
    <View style={styles.container}>
      <Text style={styles.text}>Hello, React Native!</Text>
    </View>
  );
};
```

```
const styles = StyleSheet.create({
  container: {
    flex: 1,
    justifyContent: 'center',
    alignItems: 'center',
  },

  text: {
    fontSize: 20,
    fontWeight: 'bold',
  },
});
```

Navigace

V React Native je možné využít mnoho knihoven pro práci s navigací. Mezi ty nejznámější patří *React Navigation* a *React Native Navigation*. Jakou knihovnu zvolit záleží na složitosti projektu. Vzhledem k vytvoření projektu pomocí Expo máme k dispozici navíc knihovnu *Expo Router*, která nám umožní navigaci mezi jednotlivými stránkami bez nutnosti instalace externích knihoven.
Detailní postup zprovoznění je přehledně popsán v oficiální dokumentaci na `reactnavigation.org`. Při vytvoření projektu pomocí `create-expo-app` již máme všechny prerekvizity nainstalované. Samotnou knihovnu a potřebné závislosti nainstalujeme pomocí následnicích příkazů.

```
npm install @react-navigation/native
npx expo install react-native-screens react-native-safe-area-context
npm install @react-navigation/stack
npx expo install react-native-gesture-handler
```

Po instalaci již můžeme začít knihovnu používat. Nejprve si naimportujeme potřebné závislosti do hlavní komponenty aplikace. Následně vytvoříme instanci zásobníkové navigace pomocí příkazu `createStackNavigatior()`, poté obalíme celou aplikaci do komponenty `NavigationContainer`, která zajišťuje navigaci v celé aplikaci. Do této komponenty již přidáme samotné obrazovky, které budeme v rámci navigace používat.

```
import React from 'react';
import { NavigationContainer } from '@react-navigation/
    native';
import { createStackNavigator } from '@react-navigation/
    stack';
```

```
import HomeScreen from './components/HomeScreen';

const Stack = createStackNavigator();

export default function App() {
  return (
    <NavigationContainer>
      <Stack.Navigator>
        <Stack.Screen  name="Home" component={HomeScreen} />
      </Stack.Navigator>
    </NavigationContainer>
  );
}
```

Jako výchozí obrazovka je ve výše uvedením příkladu obrazovka `Home`.[238]

[238] Výchozí obrazovka se vyskytuje jako první v seznamu obrazovek. Kromě toho je ale také možné přepsat výchozí obrazovku pomocí atributu `initialRoute` u komponenty `Stack.Navigator`.

V případě, že chceme navigovat ze stránky na jinou stránku, musíme využít navigačního objektu. Ten můžeme buď předat jako argument (každá komponenta ho dostane přiřazen automaticky), nebo můžeme využít `useNavigation` hooku. Následuje náznak řešení.

```
// použití objektu v props
export default function HomeScreen({navigation}) {
  // kód
}

// získání navigace z hooku
export default function HomeScreen() {
  const navigation = useNavigation()
}
```

Uložení dat

Nativní aplikace mají v závislosti na platformě přidělené určité místo v paměti zařízení, které mohou využít pro ukládání malého množství dat.[239] Vzhledem k tomu, že jsou aplikace napsané v React Native překládány do nativního kódu dané platformy, je možné tohoto úložiště využít.

[239] Velikost závisí na konkrétní verzi operačního systému a zařízení.

Pro uložení malých, často nestrukturovaných dat, jako je nastavení aplikace, je k dispozici persistentní úložiště `AsyncStorage`. V případě potřeby uložení větších a strukturovaných dat je vhodné využít databázi. Pro vývoj je poté možné si vybrat z poměrně velkého množství různých databází. Mezi ty nejznámější patří *Realm*, *SQLite*, *Firease*, *PouchDB* a *WatermelonDB*. Každá z uvedených uvedených nabízí jiné možnosti a jejich integrace do projektu může být

více či méně složitá. Vzhledem k jednoduchosti použití se nyní podívejme na integraci databáze SQLite.

SQLite

V současné sobě existuje celá řada knihoven poskytující SQLite databázi. My vzhledem k jednoduchosti integrace do projektu použijeme knihovnu pro Expo projekty, která poskytuje SQLite databázi a základní práci s ní. Nainstalujeme ji pomocí následujících příkazů.

```
# instalace knihovny
npx expo install expo-sqlite
```

Nyní máme připraveno vše pro práci s databází. Nový databázový objekt vytvoříme následujícím způsobem.

```
export const db = SQLite.openDatabase('db.myDB');
```

Po vytvoření databázového objektu můžeme začít s databází pracovat. K vytvoření dotazu nad databází slouží metoda `executeSql()`. Následující kód ukazuje provedení dotazu a zpracování výsledků dotazu.

```
tx.executeSql(
  'SELECT * FROM users WHERE id = ?',
  [userId],
  (tx, results) => {
    const len = results.rows.length;
    for (let i = 0; i < len; i++) {
      const row = results.rows.item(i);
      console.log(`User ID: ${row.id}, Name: ${row.name}`);
    }
  },
  (tx, error) => {
    console.error('Error executing SELECT query:', error);
  }
);
```

Ve výše uvedeném kódu je proveden dotaz nad databází a vrácené výsledky jsou vytisknuty do konzole. Druhý (nepovinný) parametr metody `executeSql()` slouží k předání pole argumentů, které obsahuje všechny potřebné hodnoty parametry dotazu (znaky `?` v SQL dotazu). Tímto způsobem je zajištěno bezpečné zpracování dat v dotazu.[240]

[240] Minimalizuje se riziko SQL injection útoku.

Závěr

Velmi stručně jsme ukázali základní práci s technologií React Native, která umožňuje vytvářet nativní mobilní aplikace pomocí jazyka JavaScript a knihovny React.