Česká zemědělská univerzita v Praze
Provozně-ekonomická fakulta
Katedra informačního inženýrství

Diplomová práce
Kontejnerizace vývojového prostředí

Bc. Lukáš Brabenec

©2021 ČZU

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Lukáš Brabenec

Systémové inženýrství a informatika
Informatika

Název práce

Kontejnerizace vývojového prostředí

Název anglicky

Containerization of development environment

Cíle práce

Cílem práce je navrhnout, naprogramovat a nasadit webovou aplikaci pro podporu kontejnerizace vývojového prostředí. Aplikace bude umožňovat vygenerovat skupinu souborů určených ke spuštění kontejnerů pro jednodušší vývoj, kde si uživatel potom pouze vybere, které programovací jazyky, databázové řešení nebo přímo hotová řešení typu ElasticSearch či MailCatcher bude potřebovat. Aplikace se bude skládat ze dvou částí, serverové a klientské. Serverová část bude vyvíjena v jazyce PHP za pomocí frameworku Symfony. Klientská část bude vyvíjena v jazyku JavaScript za pomocí knihoven React a Redux. Data budou uložena v relační databázi MariaDB.

Metodika

V první části práce bude provedena analýza požadavků a přehled použitých teoretických nástrojů a technik a také softwarových aplikací, především vývojových prosředí. V druhé části práce bude proveden návrh aplikace pomocí standardních datových a drátových modelů. Dále bude následovat implementace serverové a poté klientské části. Pro verzování aplikace bude použit GitHub. Výsledek včetně základní dokumentace bude veřejně dostupný na internetu.

Doporučený rozsah práce

60 – 100 stran

Klíčová slova

Kontejnerizace; Docker; Docker Compose; PHP; Symfony; MariaDB; JavaScript; TypeScript; React; GitHub

---

Doporučené zdroje informací

COPES, Flavio (2019) The React Handbook [online]. Copes. Dostupné z: https://flaviocopes.com/page/react-handbook/

NICKOLOFF, Jeff a KUENZLI, Stephen (2019) Docker in Action. 2nd ed. Shelter Island: Manning Publications Company. ISBN 978-1617294761.

POTENCIER, Fabien (2020) Symfony 5: The Fast Track. V1.0.6. Clichy: Symfony SAS, 2020. ISBN 9782918390374.

POULTON, Nigel (2020) Docker Deep Dive: Zero to Docker in single book. Poulton, 2020. ISBN 9781521822807.

RICHARDSON, Leonard, AMUNDSEN, Michael a RUBY, Sam (2013) RESTful Web APIs. Sebastopol: O'Reilly, 2013. ISBN 9781449358068.

Předběžný termín obhajoby

2020/21 LS – PEF

Vedoucí práce

doc. Ing. Vojtěch Merunka, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 7. 3. 2021

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 7. 3. 2021

Ing. Martin Pelikán, Ph.D.

Děkan

V Praze dne 09. 03. 2021

Čestné prohlášení

Prohlašuji, že svou diplomovou práci „Kontejnerizace vývojového prostředí“ jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 28. 03. 2021 ____________________________

Poděkování

Rád bych touto cestou poděkoval doc. Ing. Vojtěchu Merunkovi, Ph.D. za možnost volby tématu a následné cenné rady a poznatky. Dále bych chtěl poděkovat za podporu a zázemí celé mé rodině.

Kontejnerizace vývojového prostředí

Abstrakt

Tato práce se zabývá kontejnerizací vývojových prostředí a vytvořením aplikace určené k její podpoře. Aplikace umožňuje uživateli výběr technologií, které bude nově vyvíjený program potřebovat a následně vygeneruje předpis pro vytvoření prostředí, určeného pro systém *Docker*. V teoretické části je popsána samotná kontejnerizace, její principy a předchůdci, společně se systémem *Docker*, který s touto technologií pracuje. Čtenář je zde dále seznámen s architektonickým stylem REST API, jazykem PHP a JavaScript společně s jejich knihovnami. Pomocí těchto technologií je dále navržena a implementovaná aplikace, která je výsledkem celé této práce. Praktická část se zabývá návrhem, tvorbou a nasazením samotné aplikace. Navržena je podle stylu REST API a je tedy rozdělena do dvou částí, serverové a klientské. Serverová část je implementována pomocí jazyku PHP s frameworkem Symfony. Klientská část jazykem JavaScript a TypeScript s knihovnou React a Redux. Všechna data potřebná pro správný běh aplikace jsou uložena v relační databázi MariaDB. Závěrem jsou vytvořeny kontejnery určené pro produkční prostředí aplikace, které jsou následně nasazeny na adrese `https://docker.lukasbrabenec.cz`.

Klíčová slova: Kontejnerizace, Docker, Docker Compose, PHP, JavaScript, Symfony, TypeScript, React, Redux, GitHub, REST API

Containerization of development environment

Abstract

This thesis deals with the containerization of development environments and the creation of an application designed to support containerization. The application allows the user to select the technologies that the newly developed program will need and then generate a build recipe of environment for the Docker system. The theoretical part of this thesis describes the containerization itself, its principles and predecessors, together with system Docker that works with this technology. The reader is acquainted with the architectural style REST API, programming language PHP and JavaScript, together with their libraries. Result of this work is an application designed and developed by using these previosly mentioned technologies. The practical part deals with the draft, creation and deployment of the application itself. It is designed with the REST API style in mind and therefore it is divided into two parts, server and client. The server side is implemented using the PHP language with Symfony framework and the client side with JavaScript and TypeScript using React and Redux library. All data needed for the application to function is stored in relational database MariaDB. Finally, containers are created for the production environment and then deployed at `https://docker.lukasbrabenec.cz`.

Keywords: Containerization, Docker, Docker Compose, PHP, JavaScript, Symfony, TypeScript, React, Redux, GitHub, REST API

Obsah

1 Úvod **13**

2 Cíl práce a metodika **14**
2.1 Cíl práce . . . 14
2.2 Metodika práce . . . 14

3 Kontejnerizace **16**
3.1 Historie . . . 16
3.2 Open Container Initiative . . . 17
3.2.1 Image specifikace . . . 17
3.2.2 Runtime specifikace . . . 18
3.3 Kontejnery na operačních systémech . . . 19
3.3.1 Linuxové kontejnery . . . 19
3.3.2 Windows kontejnery . . . 19
3.3.3 Linux a Windows kontejnery . . . 19
3.3.4 Mac kontejnery . . . 19
3.4 Kontejnerová orchestrace . . . 19
3.5 Přínosy kontejnerizace . . . 20

4 Docker **21**
4.1 Tvůrce . . . 21
4.2 Docker Engine . . . 21
4.3 Moby . . . 21
4.4 Obrazy . . . 22
4.4.1 Operace s obrazy . . . 22
4.5 Kontejnery . . . 23
4.6 Dockerfile . . . 24
4.7 Docker Compose . . . 25
4.7.1 Struktura . . . 26

5 REST API **29**
5.1 Rozdělení klient a server . . . 29
5.2 Bezstavovost . . . 29
5.3 Komunikace . . . 29
5.3.1 Požadavky . . . 29
5.3.2 Odpovědi . . . 31

6 PHP **32**
6.1 Historie . . . 32
6.2 Dnes . . . 32
6.3 Knihovny . . . 32
6.4 Frameworky . . . 33
6.4.1 Nette . . . 33
6.4.2 Symfony . . . 33

6.4.3 Laravel . . . 34

7 JavaScript 35
7.1 ECMAScript 6 . . . 35
7.2 TypeScript . . . 35
7.3 Knihovny . . . 35
7.4 React . . . 36
7.4.1 Životní cyklus . . . 37
7.4.2 Hooky . . . 38
7.4.3 Redux . . . 39

8 Zadání aplikace 41
8.1 Serverová část . . . 41
8.1.1 Požadavky . . . 41
8.2 Datová struktura . . . 41
8.3 Klientská strana . . . 42
8.3.1 Požadavky . . . 43
8.3.2 Rozhraní . . . 43
8.4 Správa verzí . . . 44

9 Implementace serverové strany 45
9.1 Příprava prostředí . . . 45
9.2 Vytvoření projektu . . . 47
9.3 Struktura . . . 47
9.4 Knihovny . . . 50
9.5 Koncové body . . . 51
9.6 Proces vytvoření popisu Docker prostředí . . . 52
9.6.1 Diagram . . . 53
9.6.2 Controller . . . 53
9.6.3 Validátor . . . 54
9.6.4 Tvorba Docker souborů . . . 55
9.7 Testy . . . 57

10 Implementace klientské strany 58
10.1 Příprava prostředí . . . 58
10.2 Vytvoření projektu . . . 58
10.3 Knihovny . . . 58
10.4 Struktura projektu . . . 59
10.5 Princip fungování . . . 60
10.5.1 Výběr obrazu . . . 61
10.5.2 Změny stavu . . . 62
10.5.3 Validace . . . 63
10.5.4 Dokončení výběru obrazů . . . 63
10.5.5 Design . . . 64

11 Nasazení **66**
11.1 Příprava obrazů 66
11.2 Vytvoření a odeslání obrazů 68
11.3 Nasazení na produkční server 69

12 Závěr **71**

13 Požadavky a odpovědi koncových bodů **75**
13.1 Všechny obrazy 75
13.2 Detail obrazu 75
13.3 Všechny Compose verze 76
13.4 Všechny možnosti restartu 76
13.5 Vytvoření Docker prostředí 77

14 Twig šablony **77**
14.1 Docker Compose 77
14.2 Dockerfile 78

Seznam tabulek

1 Verze *Compose* formátů a *Docker Engine* podle (Docker, 2021a) . . 26

Seznam obrázků

1 Porovnání virtualizace (vpravo) a kontejnerizace (vlevo) aplikací podle (Docker, 2021f) 17
2 Extrahovaný obraz `alpine:latest` 17
3 Velikost Docker obrazů 22
4 Seznam spuštěných kontejnerů 23
5 React: životní cyklus podle (Wojciech, 2021) 38
6 Entitně vztahový model 42
7 Drátový model 43
8 Struktura nového projektu Symfony 48
9 Struktura zdrojových souborů projektu API 50
10 Diagram tvorby popisu Docker prostředí 53
11 Struktura projektu klienta 60
12 Tok dat v aplikaci 60
13 Design stránky - světlý režim 64
14 Design stránky - tmavý režim 65

Seznam zdrojových kódů

1 Spuštění Ubuntu kontejneru 23
2 Zastavení a opětovné spuštění Ubuntu kontejneru 23
3 Spuštění nové aplikace v kontejneru 24
4 Zobrazení metadat kontejneru 24
5 Smazání kontejneru 24
6 Příklad souboru *Dockerfile* 24
7 Příklad souboru *Dockerfile* s kombinací instrukcí (Docker Community, 2021) 25
8 Vytvoření WordPress aplikace pomocí Docker Compose podle (Docker, 2021d) 25
9 Composer: přidání knihovny 33
10 Composer: instalace a aktualizace knihoven 33
11 Zahrnutí PHP knihoven 33
12 React: tvorba elementu bez použití JSX 36
13 React: tvorba elementu s použitím JSX 37
14 React: JavaScript uvnitř JSX 37
15 React: Vytvoření stavu pomocí Hooku 38
16 React: Hook pro životní cykly 39
17 Dockerfile obrazu API 45
18 Konfigurační soubor webového serveru 46
19 Docker Compose serverového prostředí 46

20 Composer: vytvoření projektu 47
21 Knihovny serverového prostředí 50
22 Koncový bod tvorby archivu s popisem Docker prostředí 54
23 Serverová validace portů 55
24 Funkce obalující tvorbu Docker souborů a archivu 55
25 Archivace souborů 56
26 Test koncového bodu pro typy restartu 57
27 Docker Compose klientského prostředí 58
28 Vytvoření projektu React 58
29 Knihovny klientského prostředí 59
30 Akce vytvoření požadavku pro obrazy 61
31 Vytvoření stavu pro zvolenou verzi obrazu 62
32 Aktualizace stavu při změně portů 62
33 Definice vstupní komponenty svazku 63
34 Změna tmavého a světlého režimu 64
35 Produkční Dockerfile API 66
36 Produkční Dockerfile klienta 67
37 Produkční Dockerfile webového serveru 67
38 Nastavení projektu klienta na webovém serveru 67
39 Docker Compose pro sestavení produkčních obrazů 67
40 Označení obrazu 68
41 Odeslání obrazu 68
42 Skript vytvoření obrazů 68
43 Nastavení reverzní proxy 69
44 Docker Compose pro produkční server 70
45 Odpověď všech obrazů 75
46 Odpověď detailu obrazu 75
47 Odpověď všech Compose verzí 76
48 Odpověď všech typů restartu 76
49 Požadavek vytvoření prostředí 77
50 Twig šablona Docker Compose 77
51 Twig šablona Dockerfile 78

1 Úvod

Kontejnerizace se v poslední době stala velikým trendem ve světě softwarového vývoje, čím dál více společností přesouvá své aplikace do kontejnerů a zrychluje tak jejich tvorbu a nasazení. Tento způsob umožňuje zabalit softwarový kód a všechny jeho závislosti do takzvaných obrazů a kontejnerů, které je možné spustit jednotně na jakékoli infrastruktuře.

Koncept kontejnerizace je starý desítky let, avšak s jeho zpřístupněním pro masy přišla až společnost *Docker* v roce 2013, společně s průmyslovými standardy a nástroji, díky které se urychlila adopce této technologie.

Podle (Gartner, 2020) a jejich predikce bude do roku 2022 používat kontejnerizaci, na produkčních prostředí, více než 75 % celosvětových organizací, oproti současných 30 %. Zároveň predikují, že příjmy ze správy kontejnerů rapidně vzrostou z aktuálních 465,8 milionů amerických dolarů na 944 do roku 2024. Rychlejší převzetí této technologie ve velkých podnikových aplikací však brzdí jejich technický dluh a rozpočtová omezení. Rychlost převzetí bude tedy záviset na době, ve které budou aplikace upraveny nebo vyměněny.

V rámci této práce se budu zabývat problematikou kontejnerizace softwaru, její historií, adopcí a přínosy. Práce bude primárně zaměřená na kontejnerizaci pro vývojové prostředí, ale z částečně i na produkční, při nasazování projektu. Praktické použití kontejnerizace bude realizováno pomocí otevřeného softwaru společnosti *Docker*, který bude vysvětlen v teoretické části a poté použit v praktické. Výsledkem této práce bude webová aplikace, která bude umožňovat tvorbu dokumentů obsahujíce instrukce k vytvoření obrazu prostředí podle výběru uživatele.

Program bude navržen a implementován podle architektury REST API. Serverová část bude v jazyku PHP s použitím knihoven *Symfony* a klientská část bude v jazyku TypeScript a knihovnami *React* a *Redux*. Popis těchto technologií bude též obsahem této práce.

Aplikace bude dostupná na adrese `https://docker.lukasbrabenec.cz` a její dokumentace API na `https://docker.lukasbrabenec.cz/api/doc`.

2 Cíl práce a metodika

2.1 Cíl práce

Primárním cílem této práce je navrhnout, naprogramovat a nasadit webovou aplikaci pro podporu kontejnerizace vývojového prostředí. Aplikace bude umožňovat vygenerovat skupinu souborů obsahujících sérii instrukcí, určených ke tvorbě kontejnerů pro jednodušší přípravu vývojového prostředí. Uživatel si pouze vybere, jaké programovací jazyky, databázová řešení nebo přímo aplikace třetích stran bude pro vlastní vývoj potřebovat. Projekt se bude skládat ze dvou částí, serverové a klientské. Serverová část bude vyvíjena v jazyku PHP s frameworkem *Symfony*. Klientská část bude vyvíjena v jazyku JavaScript a TypeScript společně s knihovnami *React* a *Redux*, určených ke tvorbě uživatelského rozhraní a správy stavů. Data budou uložena v relační databázi *MariaDB*.

Vedlejšími cílem práce je vysvětlení kontejnerizace softwaru s jejím použitím a nástroji. Dalším cílem je představení jazyků PHP, JavaScript a TypeScript s příslušnými knihovnami. Závěrem práce bude popsána příprava aplikace pro produkční prostředí.

2.2 Metodika práce

Metodika je založena na studiu a analýze odborných informačních zdrojů, na jejichž základě bude navržena, implementována a nasazena webová aplikace. Práce se bude převážně zabývat relativně moderními technologiemi, u kterých jsou typické časté aktualizace, a proto bude většina zdrojů informací buď oficiální dokumentace nebo nově vydané knihy.

V první části práce bude představena softwarová kontejnerizace, její předchůdci a systém *Docker*, který tuto techniku používá ve svém provozu. Poté bude popsán jazyk PHP a JavaScript společně s jejich knihovnami, pomocí kterých bude dále implementována aplikace, určená k vytváření souborů s instrukcemi pro tvorbu vývojového prostředí.

Druhá část bude obsahovat návrh, implementaci a nasazení webové aplikace určené pro generování dokumentů, pomocí kterých lze spouštět kontejnery s vybraným prostředí. Výsledná aplikace, včetně její dokumentace a zdrojovými kódy bude veřejně dostupná na internetu.

I. Teoretická východiska

3 Kontejnerizace

Kontejnerizací se podle (IBM, 2019) rozumí zapouzdření nebo zabalení softwarového kódu, společně s jeho závislostmi, do jednotné a konzistentní jednotky, které mohou běžet na jakékoli infrastruktuře. Technologie kolem kontejnerů velice rychle dospívají a výsledkem jsou okamžitě měřitelné výhody, jak pro vývojové tak i provozní týmy a celkovou softwarovou infrastrukturu.

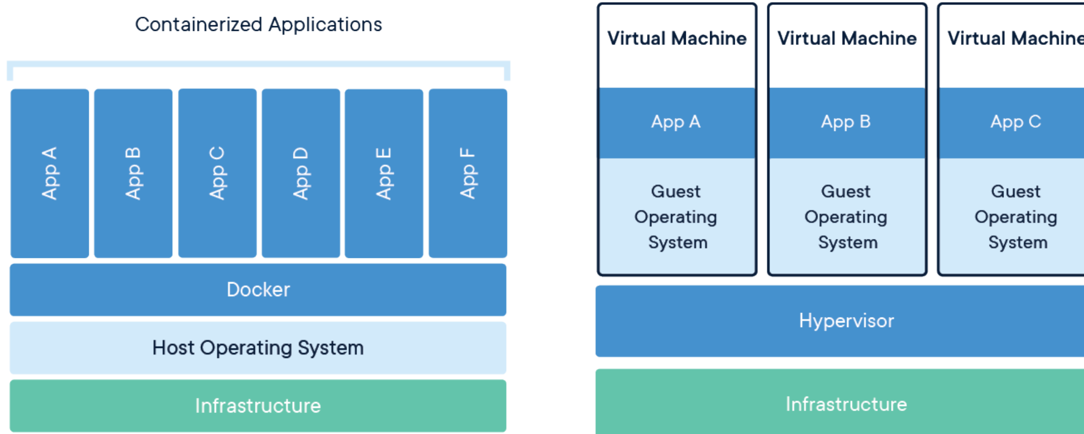
3.1 Historie

Před *kontejnerizací* a *virtualizací* běžely vyvíjené programy přímo na nainstalovaném systému serveru. Tímto přístupem vznikaly problémy pro bezpečný provoz více aplikací na jednom systému. Většinou se předem neznala náročnost aplikace, a proto se musely odhadovat potřebné hardwarové požadavky na výkon serveru. Výsledkem byl nákup robustních a drahých serverů, protože poslední, co by kupující chtěl je, aby měl pro svůj program nedostatečný výkon. Znamenalo by to neschopnost provádět různé operace a výsledkem by byla potenciální ztráta zákazníků a zisku. Servery tak mnohokrát běžely jen na 5 až 10 % své výpočetní kapacity, způsobující vysokou ztrátu finančních prostředků. (Poulton, 2020, s. 8)

Následně přišla *virtualizace* systému, která se vyřešila problém s během více aplikací na jednom serveru a důsledkem toho i nevyužití plné kapacity. Nevýhodou tohoto přístupu ovšem je, že každá aplikace musí běžet ve vlastním operačním systému, ve virtuálním stroji (Virtual Machine). Každý operační systém vyžaduje pro svůj běh další prostředky, způsobující využití procesoru, operační paměti a úložného prostoru, které by jinak mohli být využité pro běh samotných aplikací. Zároveň operační systémy potřebují opravy, monitoring a v jistých případech i licence. Další nevýhodou je pomalý start virtuálního stroje a komplikovaná přenosnost. Migrace a přenesení pracovního vytížení virtuálního stroje mezi cloudovými platformami je složitější, než by teoreticky mohlo být. (Poulton, 2020, s. 8 - 9)

Kontejnerizace řeší mnoho nevýhod virtualizace. Kontejner je ve své podstatě to samé jako virtuální stroj. Klíčovým rozdílem je, že nevyžaduje svůj vlastní, úplný operační systém, ale sdílí ho se svým hostitelem. Uvolní se tak prostředky procesoru, operační paměti a úložného prostoru, které by jinak byli využity pouze k virtualizaci systému. Výsledkem je snížení potencionálních nákladů na licence, opravy a údržbu. Spuštění kontejnerů je rychlé, protože nevyžadují zavedení vlastního operačního systému. Přenosnost kontejnerů je také velice jednoduchá, každý kontejner je vytvořen z takzvaného obrazu a obrazy lze vytvořit pomocí série instrukcí nebo stáhnout z repozitáře. Grafické porovnání virtualizace a kontejnerizace je zobrazené na obrázku 1. (Poulton, 2020, s. 9 - 10)

Zjednodušeně řečeno, kontejnerizace umožňuje aplikacím, aby byly „napsány jednou a spuštěny kdekoli“. Tato přenositelnost je důležitá z hlediska procesu vývoje a kompatibility prodejců. Nabízí také další významné přínosy, jako izolaci chyb, snadnou správa a zabezpečení. (IBM, 2019)



Obrázek 1: Porovnání virtualizace (vpravo) a kontejnerizace (vlevo) aplikací podle (Docker, 2021f)

3.2 Open Container Initiative

Kontejnery byly ve světě softwaru známé už desítky let. Nárůst na jejich popularitě v posledních letech způsobil až systém společnosti *Docker*. Byl vydán v roce 2013 a rychle se stal průmyslovým standardem, díky jeho nástrojům a jednoduchosti. Proces kontejnerizace se později, v roce 2015, standardizoval projektem OCI[1] konsorcia *Linux Foundation*. Momentálně obsahuje dvě specifikace, *image* (Linux Foundation, 2016a) a *runtime* (Linux Foundation, 2016b).

3.2.1 Image specifikace

Image specifikace definuje obsah archivu kontejnerových obrazů. Skládá se z manifestu, indexu obrazu, sady vrstev souborového systému a konfigurace. Cílem této specifikace je umožnit vytvoření interoperabilních nástrojů pro vytváření, přenos a přípravu kontejneru ke spuštění. (Chen, 2019)

Obraz je ve své podstatě archivní soubor, který má po rozbalení následující strukturu (obrázek 2).

```
❯ tree
.
├── 065271f6ba27473148a86d60e1cbb1e734bbb6adb5aae1861651558ae80a2f68
│   ├── json
│   ├── layer.tar
│   └── VERSION
├── e50c909a8df2b7c8b92a6e8730e210ebe98e5082871e66edd8ef4d90838cbd25.json
├── manifest.json
└── repositories
```

Obrázek 2: Extrahovaný obraz `alpine:latest`

[1] Open Container Initiative

Soubor `manifest.json` obsahuje:

- Cestu ke konfiguraci umístěné v JSON souboru stejného adresáře.
- Označení v repozitáři (`alpine:latest`).
- Seznam vrstev obrazu uložených v podadresářích.

Specifikace definuje pro vrstvy dvě věci:

1. Jak vrstvu reprezentovat.
   (a) Archivovat všechen obsah hlavní vrstvy.
   (b) Uložit výpočet změn vedlejší vrstvy oproti hlavní.
2. Jak sjednotit všechny vrstvy.

Aplikováním všech změn vedlejších vrstev nad hlavní získáme finální souborový systém.

3.2.2 Runtime specifikace

Rozbalením obrazu do souborového systému dostaneme něco, co je možné spustit. Proces od této části popisuje specifikace *runtime*, která si klade za cíl určit konfiguraci, prováděcí prostředí a životní cyklus kontejneru.

Konfigurace obsahují informace nezbytné k vytvoření a spuštění kontejneru. Zahrnují:

- Proces co se má spustit.
- Environmentální proměnné.
- Omezení zdrojů.
- Funkce oddělení od hosta. které se mají použít.

Některé konfigurace jsou stejné napříč všemi platformami, jiné jsou pouze pro konkrétní platformy, například pouze pro operační systém Windows. (Chen, 2019)

Specifikace *runtime* také definuje životní cyklus kontejneru, což je řada událostí, ke kterým dochází od vytvoření kontejneru, až po jeho ukončení. Kontejnery mají definované čtyři stavy: vytvořený, spuštěný, zastavený a smazaný. Mezi jednotlivými stavy se spouštějí takzvané „hooky“, které jsou určeny například pro vytváření, či mazání sítí.

3.3 Kontejnery na operačních systémech

3.3.1 Linuxové kontejnery

Počátky moderních kontejnerů vznikly ve světě Linuxu a jsou výsledkem dlouhodobé práce mnoha přispěvatelů. Příkladem je společnost Google, která přispěla několika technologiemi, spjatými s kontejnery, do linuxového jádra. Bez těchto a dalších příspěvků by neexistovaly moderní kontejnery, které jsou v tuto dobu dostupné.

Nejdůležitější technologiemi, které umožnily masivní vzrůst užívání kontejnerů, patří například jmenné prostory linuxového jádra (`kernel namespaces`), control groups (`cgroups`), souborový systém Union (`UnionFS`) a také Docker. Přes to všechno, kontejnerizace zůstala komplexní a mimo dosah většiny. Přístup ke kontejnerům a práci s nimi zjednodušil až příchod systému *Docker*. (Poulton, 2020)

3.3.2 Windows kontejnery

Společnost Microsoft v posledních letech usilovala o příchod Dockeru a kontejnerových technologií na platformu Windows. Kontejnerizace je v tuto chvíli dostupná v systémech Windows 10 a Windows Server 2016. Microsoft toho dosáhl ve spolupráci se společností Docker a její komunitou.

Práce s Windows kontejnery je stejná jako práce s linuxovými. Díky tomu mohou vývojáři a správci, kteří jsou seznámeni s funkcemi Dockeru, bez problému pracovat s kontejnery v tomto systému. (Poulton, 2020)

3.3.3 Linux a Windows kontejnery

Kontejnery sdílí svoje prostředky s hostujícím systémem. Pokud je aplikace navržena pro běh v jádru Windows, nelze kontejner s takovou aplikací spustit v systému Linux. Ovšem na počítači se systémem Windows lze spustit kontejnery pro Windows i Linux. Tato kompatibilita je umožněna díky virtualizaci linuxového jádra. Linuxové kontejnery spuštěné na hostujícím systému Windows mají mezi sebou ještě virtuální stroj, a tomu odpovídá i výkonnost. (Poulton, 2020)

3.3.4 Mac kontejnery

Kontejnery operačního systému *Mac* neexistují. Tento operační systém je ovšem možné použít jako hostující pro linuxové kontejnery, ale je toho dosaženo stejným způsobem jako u systému Windows, pomocí virtualizace linuxového jádra. (Poulton, 2020)

3.4 Kontejnerová orchestrace

Přenositelnost a agilnost kontejnerového procesu znamená, že máme možnost přesouvat a škálovat kontejnerové aplikace napříč cloudy a datovými centry. Kontejnery účinně zaručují, aby aplikace běžely všude stejným způsobem, což umožňuje rychlé a snadné využívání těchto prostředí. Aplikace se ale neustále rozšiřují, a

tak jsou potřeba nástroje automatizující údržbu, jako nahrazování vadných kontejnerů, spravování instalací aktualizací a konfigurování kontejnerů, během jejich životního cyklu. (Docker, 2021c)

Nástroje pro správu, škálování a údržbu kontejnerových aplikací se nazývají *orchestrátory* a jejich nejběžnějším příkladem je *Kubernetes* nebo *Docker Swarm.*

3.5 Přínosy kontejnerizace

Kontejnerizace přináší mnoho významných výhod pro vývojové a integrační týmy. Podle (IBM, 2019) jsou to například tyto:

- **Přenositelnost**: Kontejner vytváří spustitelný balíček softwaru, který je oddělen od hostitelského operačního systému (není na něj vázán ani na něm není závislý), je tedy přenosný a schopný běžet jednotně a konzistentně na jakékoli platformě.
- **Agilnost**: Docker Engine, dále popsaný v kapitole 4.2, započal tvorbu průmyslového standardu pro kontejnery pomocí jednoduchých nástrojů pro vývojáře a univerzálního přístupu k balení, který funguje v operačních systémech Linux i Windows.
- **Rychlost**: Kontejnery se často označují jako „lightweight“ (odlehčené), což znamená, že sdílejí jádro operačního systému počítače a nejsou zahlceny jeho další režií. Výsledkem je zvýšení efektivity hostujícího systému, snížení nákladů a zrychlení startu operačního systému, protože ho není potřeba opakovaně zavádět.
- **Izolace chyb**: Každá kontejnerizovaná aplikace je izolovaná a funguje nezávisle na ostatních. Porucha jednoho kontejneru nemá vliv na další provoz jiných kontejnerů. Vývojové týmy mohou identifikovat a opravit jakékoli technické problémy v rámci jednoho kontejneru bez výpadků v jiných kontejnerech.
- **Efektivita**: Software spuštěný v kontejnerových prostředích sdílí jádro operačního systému hostitele a aplikační vrstvy uvnitř kontejneru lze sdílet s dalšími kontejnery. Kontejnery jsou tedy ze své podstaty menší kapacity než virtuální stroj a mají kratší spouštěcí proces, což umožňuje provozovat mnohem více kontejnerů na stejné výpočetní kapacitě jako jeden virtuální počítač.
- **Snadná správa**: Platformy pro orchestraci kontejnerů automatizují instalaci, škálování a správu kontejnerových úloh a služeb. Orchestrátory mohou usnadnit úkoly správy jako je škálování kontejnerových aplikací, zavedení nových verzí a mimo jiné poskytování monitorování, protokolování a ladění.
- **Zabezpečení**: Izolace aplikací jako kontejnerů inherentně brání ostatní kontejnery a hostitelský systém při vniku škodlivého kódu. Navíc lze definovat oprávnění zabezpečení, která budou automaticky blokovat nežádoucí komponenty ve vstupu do kontejnerů nebo omezovat zbytečnou komunikaci.

4 Docker

Docker je open-source projekt určený k vytváření, spravování a organizování kontejnerů. Používá se k automatizaci nasazení aplikací jako přenosných a samostatných kontejnerů, které mohou běžet v cloudu nebo místně. Docker je zároveň společnost, která propaguje a vyvíjí tuto technologii ve spolupráci s dodavateli cloudových řešení a systémem Linux i Windows. (Microsoft, 2021b)

4.1 Tvůrce

Společnost Docker založil Americký vývojář jménem Solomon Hykes v Kalifornském městě San Fransisco. Původně byla společnost Docker platforma jako služba (PaaS) jménem dotCloud. Jelikož pro svojí činnost používali linuxové kontejnery, vytvořili si pro práci s nimi nástroj, který byl eventuálně pojmenován Docker. Společnost dotCloud avšak nebyla moc úspěšná, a tak se tvůrci rozhodli soustředit se na svůj nástroj Docker, s cílem zpřístupnit ho ostatním. Společnost se později přejmenovala na Docker, Inc. (Poulton, 2020)

Dnes je Docker světově uznávaná technologická společnost s tržním oceněním přes 1 miliardu amerických dolarů.

4.2 Docker Engine

Když se mluví o „Dockeru“, bývá většinou na mysli Docker Engine, open-source technologie, určená k vytváření a spravování kontejnerů, vytvořená v programovacím jazyku Golang. Funguje jako klient – server. Skládá se z dlouhodobě běžícího procesu - démona `dockerd`, API a klientského rozhraní, ovládaného pomocí příkazového řádku.

Klientské rozhraní používá Docker API, k ovládání nebo interakci s démonem `dockerd` prostřednictvím skriptování nebo přímých příkazů. Démon vytváří a spravuje objekty jako obrazy, kontejnery, sítě a svazky.

Docker Engine je dostupný na oficiálních stránkách Dockeru nebo ho lze vytvořit přímo ze zdrojového kódu, umístěném v repozitáři GitHub. Existují dvě edice, *Enterprise* a *Community*. Oboje dostávají nové stabilní verze každé čtvrtletí. Každá Community edice dostává podporu 4 měsíce a každá Enterprise edice 12 měsíců. Od prvního čtvrtletí roku 2017 dodržuje každý název formát YY.MM-xx podobný verzím linuxového systému *Ubuntu*. (Docker, 2021b)

4.3 Moby

Termín Docker býval označením pro skupinu open-source nástrojů, jejichž složením vznikne démon `dockerd` a klient. Tento projekt byl v roce 2017, na konferenci DockerCon, přejmenován na Moby. (Poulton, 2020)

Cílem *Moby* projektu bylo vytvořit takzvaný *upstream*, který umožňoval častější aktualizace a rozdělení do více samostatných komponent. Projekt je dostupný na repozitáři GitHub a tvoří ho skupina subprojektů a nástrojů. Hlavní projektem je *Docker Engine*, který je dále dělen do menších částí. (Docker, 2021e)

4.4 Obrazy

Obrazy kontejnerů lze chápat jako šablony virtuálního stroje. Šablona virtuálního stroje je jako zastavený virtuální stroj, stejně jako Docker obraz je nespuštěný kontejner. Z pohledu objektově-orientovaném programování je lze chápat jako třídy. Obrazy lze stahovat z různých repozitářů, tím nejznámějším je Docker Hub. Operace stažení obrazu se nazývá *pull*, pomocí které se obraz stáhne z registru repozitáře a uloží do lokálního úložiště, kde je z něj následně možné spustit kontejner. (Poulton, 2020, s. 49)

Obrazy se skládají z několika vrstev, reprezentující jednotný objekt. Každá vrstva je závislá na té předchozí. Uspořádání vrstev je klíčem ke zvýšení efektivnosti životního cyklu obrazu. Vrstva, která mění nejvíce věcí by měla být mezi prvními. Pořadí se může ovlivnit například pořadím instrukcí v souboru *Dockerfile*, více o něm v kapitole 4.6.

Uvnitř obrazu je tedy „osekaný“ operační systém a všechny soubory, společně se závislostmi, potřebnými ke spuštění aplikace. Kontejnery by měli být rychlé a přenosné, proto je vhodné nevkládat příliš mnoho věcí do obrazů. Obrazy obvykle obsahují pouze jeden interpretr příkazového řádku nebo dokonce žádný. Zároveň nemají svůj kernel, protože všechny kontejnery sdílí přístup k hostitelovu jádru. Tímto jsou dosaženy malé velikosti. Příkladem malého obrazu je například oficiální distribuce Alpine Linux (`alpine:latest`), který zabírá přibližně 6 MB nebo Ubuntu (`ubuntu:latest`), která má aktuálně velikost přibližně 73 MB.

Obrazy systému Windows bývají větší z důvodu, jakým tento operační systém funguje. Aktuální, nekomprimovaný obraz pro vývojovou platformu .NET, s označením `microsoft/dotnet:latest`, má přes 1,7 GB.

```
REPOSITORY         TAG       IMAGE ID       CREATED        SIZE
microsoft/dotnet   latest    0d8f4befaa1a   2 weeks ago    1.74GB
alpine             latest    389fef711851   2 weeks ago    5.58MB
ubuntu             latest    f643c72bc252   5 weeks ago    72.9MB
```

Obrázek 3: Velikost Docker obrazů

4.4.1 Operace s obrazy

- **build**: Vytvoření obrazu pomocí instrukcí v souboru *Dockerfile*.
- **history**: Vypsání instrukcí, kterými byl obraz vytvořen.
- **import**: Vložení obsahu z archivu ve formátu *tar* do souborového systému obrazu.
- **inspect**: Detailní informace o obrazu (proměnné, svazky, sítě a další).
- **list**: Zobrazení aktuálně stažených obrazů. Výstup na obrázku 3.
- **prune**: Vymazání nepoužívaných obrazů (bez vytvořených kontejnerů).

- **pull**: Stažení specifikovaného obrazu z repozitáře. Obraz se popisuje ve formátu <repozitář>:<tag>. V případě vynechání tagu se stáhne obraz s označením *latest*, tedy poslední verze.
- **push**: Odeslání obrazu do registru repozitáře.
- **remove**: Vymazání z lokálního uložiště.
- **save**: Uložení obrazu do *tar* archivu.
- **tag**: Označení cílového obrazu.

4.5 Kontejnery

Kontejner je spuštěná instance obrazu. Z každého lze spustit jeden a více kontejnerů. Nejjednodušším způsobem spuštění kontejneru je příkaz: docker run <obraz> <aplikace>.

```
$ docker run -d  --name ubuntu_kontejner ubuntu:latest /bin/bash
$ docker run -it --name ubuntu_kontejner ubuntu:latest /bin/bash
```

Zdrojový kód 1: Spuštění Ubuntu kontejneru

Volba `-it` nechá otevřený standardní vstup `stdin` a připojí ho k aktuálnímu (hostitelovu) terminálu, tento kontejner bude spuštěn v popředí. Naopak volba `-d` (*detached*) ho spustí na pozadí. Název kontejneru bude `ubuntu_kontejner`, toho se docílí volbou `name`. V případě, kdy nebude obraz `ubuntu:latest` stažen na lokálním stroji, Docker se ho automaticky pokusí najít v registru repozitáře a následně ho stáhne nebo vypíše chybu v případě nenalezení.

Kontejnery jsou spuštěny do té doby, dokud spouštěná aplikace neskončí. Zobrazit aktuálně spuštěné kontejnery lze příkazem `docker ps`. Volbou `-a` lze zobrazit i zastavené kontejnery. Jeho výstup je zobrazen v obrázku 4.

```
❯ docker ps
CONTAINER ID   IMAGE           COMMAND       CREATED      STATUS       PORTS     NAMES
3bdf7af53da5   ubuntu:latest   "/bin/bash"   2 days ago   Up 1 second            ubuntu_kontejner
```

Obrázek 4: Seznam spuštěných kontejnerů

Kontejner lze operací `stop` a následně spustit pomocí operace `start`. Cílový kontejner lze identifikovat buď jeho identifikátorem (sloupec `CONTAINER_ID`) nebo názvem (sloupec `NAMES`). Identifikátory lze získat z výstupu zobrazení spuštěných kontejnerů, viz. obrázek 4.

```
$ docker stop ubuntu_kontejner
$ docker start ubuntu_kontejner
```

Zdrojový kód 2: Zastavení a opětovné spuštění Ubuntu kontejneru

Spuštění nové aplikace na již běžícím kontejneru se provádí příkazem `exec`. Běžným použitím je spuštění interpretru příkazové řádky a připojení aktuální konzole.

```
$ docker exec -it ubuntu_kontejner /bin/bash
```

Zdrojový kód 3: Spuštění nové aplikace v kontejneru

Informace o kontejneru, jako třeba seznam svazků, sítí nebo otevřených portů, je možné zobrazit pomocí příkazu `inspect`.

```
$ docker inspect ubuntu_kontejner
```

Zdrojový kód 4: Zobrazení metadat kontejneru

Zastavený kontejner lze smazat příkazem `rm`.

```
$ docker rm ubuntu_kontejner
```

Zdrojový kód 5: Smazání kontejneru

4.6 Dockerfile

Dockerfile je soubor popisující prostředí aplikace a říká systému Docker, jak vytvořit obraz. Obvykle bývá ukládán do kořenového adresáře aplikace, označovaného jako *build context*, který definuje, kam bude mít *Docker Engine* při tvorbě obrazu přístup.

Dockerfile, podle (Poulton, 2020, s. 134), lze použít i jako formu dokumentace. Skládá se ze série instrukcí, které lze snadno číst a může tak, například novému vývojáři, pomoci rychleji pochopit prostředí projektu. Vytvoření jednoduchého obrazu pro aplikaci v prostředí *Node* je zobrazeno v následujícím kódu 6.

```
# Toto je komentar
FROM alpine
LABEL maintainer="email@email.com"
RUN apk add -update nodejs nodejs-npm
COPY . /src
WORKDIR /src
RUN npm install
EXPOSE 8080
ENTRYPOINT ["node", "./app.js"]
```

Zdrojový kód 6: Příklad souboru *Dockerfile*

Instrukce `FROM` říká, ze kterého obrazu se bude vycházet. V tomto případě se vychází z `alpine:latest`. Druhý řádek přidá do obrazu informaci o správci. Dalším krokem je nainstalování prostředí *nodejs* a *nodejs-npm* instrukcí `RUN`. Následuje zkopírování kódu aplikace, který je uložen na stejném místě jako Dockerfile, do adresáře `/src` v kontejneru pomocí `COPY`. Adresář `src` se nastaví jako pracovní, instrukcí `WORKDIR`. Poté se nainstalují knihovny aplikace pomocí `RUN` a odkryje se port 8080, `EXPOSE 8080`. Soubor `app.js` se nastaví jako základní aplikace pro spuštění, pomocí systému *Node*, `ENTRYPOINT ['node', './app.js']`. Komentáře jsou označeny znakem `#`.

Některé instrukce přidávají vrstvu do obrazu a jiné pouze metadata. Příkladem přidání vrstvy jsou instrukce `FROM`, `RUN` a `COPY`. Metadata se přidávají instrukcemi

LABEL, WORKDIR, ENV, EXPOSE a ENTRYPOINT. Jednoduše řečeno, pokud instrukce přidává obsah do obrazu, jako třeba aplikaci nebo soubor, vytvoří se nová vrstva. Naopak, pokud instrukce pouze popisuje, jak pracovat s obrazem, přidají se metadata. Každá přidaná vrstva zvětšuje velikost programu, proto je vhodné minimalizovat jejich používání buď kombinací více příkazů do jednoho nebo rozdělením souboru *Dockerfile* na více stupňů (*staging*). Běžným příkladem je kombinace více RUN instrukcí do jedné, pomocí dvojích ampersandů (&&) společně se zpětnými lomítky pro řádkování. Ukázka v kódu 7. (Poulton, 2020, s. 134 - 137)

```
FROM php:7.4-fpm
RUN apt-get update \
   && apt-get install -y libfreetype6-dev \
                         libjpeg62-turbo-dev \
                         libpng-dev \
   && docker-php-ext-configure gd --with-freetype --with-jpeg \
   && docker-php-ext-install -j$(nproc) gd
```

Zdrojový kód 7: Příklad souboru *Dockerfile* s kombinací instrukcí (Docker Community, 2021)

4.7 Docker Compose

Compose je nástroj určený k definici a spouštění skupiny Docker kontejnerů. Pro popis se používá soubor formátu YAML, kde se konfigurují součásti aplikace. Compose je používán k usnadnění spuštění skupin servis, příkladem může být PHP aplikace *WordPress* s *MySQL* databází. Díky Compose lze tuto skupinu spustit jedním příkazem: docker-compose up.

```
version: "3.3"

services:
   db:
      image: mysql:5.7
      volumes:
         - db_data:/var/lib/mysql
      restart: always
      environment:
         MYSQL_ROOT_PASSWORD: somewordpress
         MYSQL_DATABASE: wordpress
         MYSQL_USER: wordpress
         MYSQL_PASSWORD: wordpress
      expose:
         - 3306

   wordpress:
      depends_on:
         - db
      image: wordpress:latest
      ports:
         - "8000:80"
      restart: always
      environment:
         WORDPRESS_DB_HOST: db:3306
```

```
            WORDPRESS_DB_USER: wordpress
            WORDPRESS_DB_PASSWORD: wordpress
            WORDPRESS_DB_NAME: wordpress

volumes:
    db_data: {}
```

Zdrojový kód 8: Vytvoření WordPress aplikace pomocí Docker Compose podle (Docker, 2021d)

4.7.1 Struktura

Struktura konfiguračního souboru se skládá z šesti nejvyšších elementů. Dvou povinných, verze (`version`) a služby (`services`) a čtyř nepovinných, sítě (`networks`), svazky (`volumes`), konfigurace (`configs`) a tajemství (`secrets`). (Docker, 2020)

4.7.1.1 Version

Na začátku konfiguračního souboru musí být vždy definovaná verze Compose formátu, aby systém věděl, jak s konfigurací nakládat a jaké jsou dostupné elementy. Každá verze formátu koresponduje s verzí Docker Engine.

Compose formát	Docker Engine
3.8	19.03.0+
3.7	18.06.0+
3.6	18.02.0+
3.5	17.12.0+
3.4	17.09.0+
3.3	17.06.0+
3.2	17.04.0+
3.1	1.13.1+
3.0	1.13.0+
2.4	17.12.0+
2.3	17.06.0+
2.2	1.13.0+
2.1	1.12.0+
2.0	1.10.0+
1.0	1.9.1+

Tabulka 1: Verze *Compose* formátů a *Docker Engine* podle (Docker, 2021a)

4.7.1.2 Services

V této části se definují a konfigurují součásti aplikace. Každá služba obsahuje své popisující vlastnosti. Nejběžnější jsou následující:

- **build**: Konfigurační možnosti, které se aplikují při vytváření obrazu. Uvnitř uzlu `build` je nutné definovat parametr `context`.

  - **context**: Obsahuje buď cestu k adresáři obsahující soubor *Dockerfile* nebo URL *git* repozitáře.
  - **dockerfile**: Alternativní název souboru *Dockerfile*.
  - **args**: Argumenty pro *Dockerfile*. Lze definovat jako `klíč: hodnota` nebo `klíč=hodnota`. V případě vynechání hodnoty ji bude nutno zadat při vytváření obrazu.

- **image**: Specifikace názvu již existujícího obrazu. V případě, kdy je definován parametr `build`, určuje název nového obrazu.
- **container_name**: Řetězec specifikující vlastní název kontejneru. V případě vynechání se název vygeneruje automaticky.
- **depends_on**: Seznam služeb, na kterých je tato služba závislá. Zároveň určuje pořadí, ve kterém se kontejnery spouští a zastavují.
- **env_file**: Cesta k souboru obsahujícím enviromentální proměnné.
- **environment**: Seznam environmentálních proměnných a jejich hodnoty.
- **expose**: Seznam síťových portů, které kontejner odhalí ostatním službám, ale ne hostujícímu stroji.
- **extends**: Název služby, kterou aktuální služba rozšiřuje.
- **networks**: Definice sítí definovaných pod nejvyšším elementem `networks`, ke kterým se aktuální služba připojí.
- **ports**: Seznam síťových portů, které budou odhaleny hostujícímu počítači a na které porty budou přesměrovány.
- **restart**: Určení chování při ukončení kontejneru. Povolené hodnoty:
  - **no**: Základní chování, kontejner se nikdy sám nerestartuje.
  - **always**: Kontejner se bude vždy restartovat do té doby dokud není smazán.
  - **on-failure**: Kontejner se restartuje pokud jeho výstupní kód indikuje chybu.
  - **unless-stopped**: Kontejner se restartuje dokud není zastaven a smazán.
- **user** Určení uživatele použitého ke spuštění kontejnerového procesu. V případě vynechání se použije uživatel *root*.
- **volumes** Definice svazků hostujícího stroje, ke kterým se má aktuální služba připojit.
- **configs** Název konfigurace definované v nejvyšším elementu `configs`, ke kterým dostane tato služba přístup.

- **secrets** Název tajemství definovaného v nejvyšším elementu `secrets`, ke kterému dostane tato služba přístup.
- **command** Přepíše původní příkaz při spuštění ve zvoleném obrazu.

4.7.1.3 Networks

Nejvyšší element určený k definici názvu sítí. Ve výchozím stavu se vytvoří jedna společná síť pro všechny definované služby, kde spolu mohou komunikovat pomocí definovaného názvu a portu. Například v konfiguraci kódu 8 bude databáze dostupná pod adresou `http://db:3306`.

Compose umožňuje vytvoření více sítí, pro oddělení jednotlivých služeb nebo připojení na již existující síť.

4.7.1.4 Volumes

Nejvyšší element `volumes` je určen k definici názvu svazků. Svazky mohou být uloženy na hostujícím stroji nebo na sdíleném serveru. Přístup k ním bude mít každá služba, která bude mít definovaný stejnojmenný svazek v uzlu `volumes`.

4.7.1.5 Configs

Konfigurace umožňují službám adaptovat své chování, bez nutnosti aktualizace svého obrazu. Uloženy jsou v orchestrátoru a jejich použití je možné pouze při jeho zapnutí. Přístup k ním je podobný implementaci svazků, každá služba obsahující element `configs` se stejnojmenným názvem konfigurace bude mít k němu přístup.

4.7.1.6 Secrets

Tajemství jsou typy konfigurace obsahující citlivá data, jako hesla, SSH klíče nebo SSL certifikáty. Během přenosu jsou zakódovány a uloženy v orchestrátoru. Přístupné jsou pouze těm službám, co mají definovaný element `secrets` a v něm stejnojmenné tajemství.

5 REST API

REST (REpresentational State Transfer) je architektonický styl, určený pro definici standardů komunikace počítačových systémů na webu. Charakteristikami REST systémů (typicky označovaných jako *RESTful*) je jejich bezstavovost a rozdělení na klienta a server. (Codeacademy, 2021)

5.1 Rozdělení klient a server

V architektonickém stylu REST lze implementace klienta a serveru provádět nezávisle, aniž by každý o sobě věděl. Znamená to, že kód na klientské straně může být kdykoli pozměněn bez toho, aby ovlivnil běh serveru a naopak.

Dokud každá strana zná formát zpráv, které si navzájem posílají, mohou být udržovány odděleně. Oddělení uživatelského rozhraní od konceptů ukládání dat zvyšuje flexibilitu platformy a vylepšuje škálovatelnost díky zjednodušení serverových komponent. Rozdělení umožňuje nezávislý rozvoj každé komponenty. (Codeacademy, 2021)

5.2 Bezstavovost

Systémy, které dodržují paradigma REST jsou bezstavové, což znamená, že server nemusí vědět nic o tom, v jakém stavu se klient nachází, a naopak. Tímto způsobem můžou oba porozumět jakékoli přijaté zprávě, aniž by znali předchozí zprávy. Omezení bezstavovosti je vynuceno díky použití zdrojů (*resources*), nikoli příkazů. Zdroje jsou podstatou webu, popisují objekt, dokument nebo nějakou *věc*, kterou je potřeba uložit nebo odeslat jiné službě. REST systémy interagují prostřednictvím standardních operací se zdroji, nespoléhají se tak na implementace rozhraní.

Tato omezení pomáhají *RESTful* aplikacím dosáhnout spolehlivosti, rychlého výkonu a škálovatelnosti, protože komponenty mohou být spravovány, aktualizovány a opakovaně použity bez ovlivnění systému jako celku, a to i během provozu. (Codeacademy, 2021)

5.3 Komunikace

V REST architektuře, klient posílá požadavky pro získání či úpravu zdroje a server na ně odesílá odpovědi.

5.3.1 Požadavky

REST vyžaduje, aby klient v případě potřeby získání nebo modifikování dat na serveru, vytvořil a odeslal požadavek. Ten se skládá z:

- HTTP metody definující typ operace k provedení.
- Hlavičky umožňující klientovi přidat informace o požadavku.
- Cesty ke zdroji.
- Nepovinného těla zprávy obsahující data.

5.3.1.1 HTTP metody

HTTP definuje skupinu metod požadavků, které popisují operace k provedení na určitém zdroji. (Mozilla, 2021b)

- **GET** - Požadavek na reprezentaci specifikovaného zdroje. Požadavky používající tuto metodu by měly být použity pouze k získání dat.
- **HEAD** - Metoda požadující odpověď identickou k *GET* požadavku, ale bez těla zprávy.
- **POST** - Odeslání entity do specifického zdroje, způsobující změnu stavu nebo jiných vedlejších účinků na serveru.
- **PUT** - Výměna celé reprezentace zdroje na serveru za odesílanou.
- **DELETE** - Smazání specifikovaného zdroje.
- **CONNECT** - Vytvoření tunelu na server identifikovaného pomocí cílového zdroje.
- **OPTIONS** - Požadavek o vrácení možností komunikace pro cílový zdroj.
- **TRACE** - Provedení testu zpětné smyčky zpráv podél cesty k cílovému zdroji.
- **PATCH** - Částečná modifikace cílového zdroje odeslanými daty.

5.3.1.2 Hlavičky

Hlavičky umožňují klientům a serverům přidávat informace o požadavku nebo odpovědi. Každé pole hlavičky se skládá z názvu, nerozlišující velká a malá písmena, následujíce dvojtečkou a hodnotou. (Mozilla, 2021a)

Hlavičky jsou podle standardu RFC 2616 (Fielding et al., 1999) kontextově seskupeny na:

- **Obecné**, určené pro požadavky i odpovědi, ale bez vztahu k datům v tělu zprávy. Příkladem je `Cache-Control` pro určení způsobu ukládání do mezipaměti.
- **Požadavkové**, obsahující více informací o požadovaném zdroji nebo o klientovi požadující zdroj. Příkladem je `Accept`, pro určení přijatelného formátu odpovědi nebo `Authorization`, pro autorizaci.
- **Odpověďové**, obsahující přídavné informace o odpovědi, jako jejich lokace nebo informace o serveru. Patří sem například `Location`, určující adresu pro přesměrování.
- **Entitní**, poskytující informace o tělu zdroje. `Content-Length`, popisující velikost zdroje nebo `Content-Type`, definující typ média, jsou jejich příkladem.

5.3.1.3 Cesty

Požadavky musí obsahovat cestu ke zdroji u kterého má být provedena určitá operace. V *RESTful* aplikačním rozhraní by měla být struktura cesty navržena tak, aby klient věděl, o jaký zdroj se jedná. (Codeacademy, 2021)

5.3.2 Odpovědi

Každá odpověď serveru musí obsahovat hlavičku `Content-Type`, obsahujíce označení internetového typu těla odpovědi. Textové typy jsou například `text/html` nebo `application/json` označující, že tělo je v HTML nebo JSON formátu. (Codeacademy, 2021)

5.3.2.1 Stavové kódy

Odpovědi serveru musí obsahovat stavové kódy, které informují klienta o úspěšnosti jeho požadavku. Rozdělené jsou do pěti skupin:

- **100 - 199** informační
- **200 - 299** úspěšné
- **300 - 399** přesměrovací
- **400 - 499** chyba klienta
- **500 - 599** chyba serveru

Pro metodu GET se obvykle vrací kód 200, označeným jako `ok`. Pro metodu POST se většinou vrací 201 - `created`, protože bývá používán k vytvoření zdroje a pro PUT nebo DELETE, 204 - `no content`. V případě chybného těla požadavku se obvykle vrací 400 - `bad request` a při nečekané chybě serveru 500 - `internal server error`.

6 PHP

PHP je interpretovaný skriptovací jazyk. Napsaný kód se nahraje na webový server a poté se spouští interpretem. Webovým serverem typicky bývá *Apache* nebo *Nginx*. Zároveň je možné spustit *PHP* kód v příkazovém řádku podobně jako lze třeba *Bash*, *Ruby* nebo *Python*. (Lockhart, 2015, s. 16)

6.1 Historie

Jazyk vytvořil Rasmus Lerdorf. Původně jen jako skupinu CGI[2] skriptů, určených ke sledování návštěv webových stránek. Mezi lety 1994 a 1998 byl jazyk *PHP* přepsán na plnohodnotný programovací jazyk s konzistentnější syntaxí a základní podporou objektového programování. Finální produkt byl pojmenován *PHP 3* a byl vydán na konci roku 1998. Byla to první verze jazyka připomínající moderní verze. Poskytovala mnoho rozšíření pro různé databáze, protokoly a aplikační rozhraní. Ke konci roku 1998 bylo *PHP 3* nainstalováno na přibližně 10 % webových serverů celého světa. (Lockhart, 2015, s. 16)

6.2 Dnes

Dnes je *PHP* jedním z nejpoužívanějších jazyků pro webový vývoj. Dříve bylo běžné napsat *PHP* soubor a nahrát ho na produkční server pomocí rozhraní FTP, což není dobrá praktika. Dnes se běžně používají systémy správy verzí[3]. Společně s *PHP* bývá často používán nástroj správy knihoven *Composer*, umožňující uživateli definovat potřebné knihovny. Pro správný zápis se používají standardy *PSR* a kód se testuje pomocí nástrojů jako *PHPUnit*. Aplikace je možné používat s protokolem *FastCGI*, pomocí oblíbeného webového serveru *nginx*. Pro zvýšení rychlosti aplikace se používá *opcode* cache. Jako interpretr je od verze čtyři použit *Zend Engine* napsaný v jazyku C. (Lockhart, 2015, s. 17)

Nejnovější verze *PHP* je osm. Přináší oproti sedmým verzím striktnější typování datových typů a mnoho dalších funkcí, jako třeba výraz `match`, pro striktnější porovnávání oproti `switch` výrazu nebo atributy, které nahrazují *PHPDoc* anotace s nativní syntaxí *PHP*. (PHP Group, 2020)

6.3 Knihovny

Hlavní repozitář *PHP* knihoven se nazývá *Packagist*. O jejich stažení pro aplikaci a další správu se stará manažer komponent *Composer*, který kontroluje kompatibilitu mezi knihovnami. Všechny potřebné knihovny, společně s jejich verzemi, se definují do souboru `composer.json`. Po prvotním nastavení a instalaci knihoven se vytvoří soubor `composer.lock`, který uzamkne definice potřebných verzí knihoven a zajistí tak jednotné instalace na všech zařízení.

[2]Common Gateway Interface - protokol určený k propojení externích aplikací s webovým serverem

[3]Version Control System (VCS), například systém *git*

Knihovny se přidávají následujícím příkazem nebo manuální úpravou v souboru `composer.json`.

```
$ composer require <vendor>/<package>:<version>
```

Zdrojový kód 9: Composer: přidání knihovny

Po prvotní instalaci pomocí instalačního příkazu se vytvoří soubor `composer.lock`. Ten je dále možný aktualizovat aktualizačním příkazem `update`.

```
$ composer install
$ composer update
```

Zdrojový kód 10: Composer: instalace a aktualizace knihoven

Composer všechny nainstalované knihovny umístí do adresáře `vendor` a vytvoří soubor `autoload.php`, který poté stačí zahrnout do kódu a aplikace tak dostane přístup ke všem staženým knihovnám.

```
<?php
    require 'vendor/autoload.php';
```

Zdrojový kód 11: Zahrnutí PHP knihoven

6.4 Frameworky

Framework je sada knihoven, tvořící celek pro usnadnění a sjednocení vývoje. Součástí frameworků je injekce závislostí (*dependency injection, DI*), databázová vrstva, MVC (*Model View Controller*) architektura a routování. Nevýhodou je přínos věcí, které pro projekt nemusejí být potřeba a pro menší projekty tak mohou přinést vyšší složitost. Naopak jejich výhodou je přínos mnoha dobrých praktik, které se mohou postarat například o dobře čitelný kód a konzistentnost.

Pro větší projekty je inkluze frameworku téměř nutností, u menších je to téma k zamyšlení.

6.4.1 Nette

Nette je framework od českých vývojářů. Obsahuje šablonový systém *Latte*, ladící nástroj *Tracy* a vlastní testovací systém *Nette Tester*. (Nette Foundation, 2020)

6.4.2 Symfony

Symfony je jeden z nejúspěšnějších frameworků *PHP*. Je složený z mnoha modulů, které používají i jiné frameworky. Používá komponenty jako `HTTPClient`, sloužící jako obal pro datové proudy a `cURL`[4], také obsahuje nástroje pro API. `Mailer`, určený k usnadnění odesílání emailů a poskytující integraci na nejpopulárnější mailové služby. `Serializer`, pro převod objektů na specifický formát jako třeba XML, JSON, YAML a naopak. `Messenger`, pro usnadnění odesílání a příjem zpráv od ostatních aplikací nebo přes takzvané fronty zpráv (*Message Queue - MQ*). (Potencier, 2019) (Symfony, 2020)

[4]Program určený k přenosu dat pomocí různých síťových protokolů

6.4.3 Laravel

Laravel je jedním z nejpoužívanějších frameworků na celém světě. Mnoho používaných modulů, jako třeba `BrowserKit`, `Console`, `Debug` nebo `FileSystem`, přebírá od frameworku *Symfony*. Používá rozhraní příkazové řádky *Artisan*, které pomáhá při tvorbě webové aplikace. Objektově relační mapování (*ORM*) nazývané *Eloquent*, šablonovací systém *Blade* a další vlastní komponenty usnadňující vývoj. (Laravel, 2020)

7 JavaScript

JavaScript je jazyk interpretovaný nebo kompilovaný pomocí JIT[5] kompilátoru. Jeho použití je nejvíce známé jako skriptovací jazyk pro webové stránky, ale v poslední letech je populární i v prostředích mimo internetové prohlížeče, jako třeba prostředí *Node.js*. (Mozilla, 2021c)

Jazyk je prototypem-založené, což je styl objektově orientovaného programování, kde třídy nejsou explicitně definované, ale derivované přidáváním vlastností a metod do instancí jiné třídy nebo přidáním do prázdného objektu. Umožňuje tedy vytvoření nového objektu bez definice třídy. Je multiparadigmatický[6], jednovláknový, podporující imperativní a funkcionální programování. (Mozilla, 2021c)

Standard jazyku se nazývá *ECMAScript* (*ES*). Od roku 2012 podporují všechny internetové prohlížeče standard ES 5.1. Starší prohlížeče podporují alespoň verzi ES 3. V roce 2015 byla vydána verze šest, která se oficiálně nazývá *ECMAScript 2015*, původně označovaná jako *ECMAScript 6*. Od té doby vycházejí v ročních cyklech. (Mozilla, 2021c)

7.1 ECMAScript 6

Od ES 6 dostal jazyk mnoho vylepšení jako třídy a moduly, iterátory, kolekce nebo třeba šipkové a asynchronní funkce. (ECMA International, 2021)

ES 6 podporuje většina moderních prohlížečů, problémem ale je, že ne všichni uživatelé používají poslední verze. Pro zpětnou kompatibilitu je nutné využít transpilátor, který zajistí přepsání do starší verze jazyka. Transpilátorů je celá řada, jedním z nich je například *Babel* nebo *TypeScript*.

7.2 TypeScript

TypeScript je open-source programovací jazyk založený na jazyku *JavaScript*, kam přidává statické typování. Typy přidávají způsob popisu stavu objektu s lepší dokumentací a umožňují tak validaci kódu. Všechen validní *JavaScript* kód je zároveň *TypeScript* kódem. Pro transpilaci do jazyka *JavaScript* je možné použít jak *TypeScript*, tak *Babel* kompilátor. (Microsoft, 2021a)

7.3 Knihovny

Pro správu *JavaScript* knihoven se používají nástroje nazvané *npm*[7] nebo *yarn*. Zahrnuté knihovny se ukládají do souboru `package.json` nebo `yarn.lock`. Po prvotní instalaci se vytvoří soubor `package.lock` nebo `yarn.lock`, který se postará o uzamknutí verzí knihoven. Stažené knihovny se ukládají do adresáře `node_modules`. (NPM inc., 2021)

[5]Just In Time - speciální metoda překladu programu. V době spuštění a provádění je přeložen přímo do nativního strojového kódu a zrychlí tak jeho běh.

[6]Podporující více než jeden programovací způsob.

[7]Node Package Manager

7.4 React

React je framework, určený ke tvorbě uživatelských rozhraní. Byl vydán v roce 2013 společností Facebook. Hlavním cílem bylo rozdělit uživatelské prostředí webové aplikace na skupinu komponent, obsahující různé stavy. *React* je určen primárně pro tvorbu SPA[8] nebo mobilních aplikací, díky jeho optimalizaci pro práci s rychle se měnícími daty. Samotný *React*, bez dalších knihoven, je určen pouze pro management stavů a vykreslování těchto stavů do DOM. (Facebook, 2021a)

DOM (*Document Object Model*) je stromová reprezentace webové stránky, začínající značkou `<html>`, následující potomky nazývanými uzly (*node*).

Mnoho frameworků před *Reactem* přímo upravovali DOM při každé změně obsahu. *React* přišel s technologií zvanou *virtuální DOM*. Tento způsob umožňuje prohlížeči používat méně prostředků, protože aktualizace se provádí pouze pokud komponenta změní svůj stav. (Copes, 2020)

Při změně stavu označí *React* komponentu jako *dirty* a provede následující operace:

- Aktualizuje virtuální DOM podle komponent označených jako *dirty* (včetně dalších operací, jako provedení funkce `shouldComponentUpdate()`).
- Spustí *diffing* algoritmus pro vyhodnocení změn.
- Aktualizuje opravdový DOM.

Klíčovou operací je seskupení všech změn a provedení jednotné aktualizace DOM. Prohlížeč tak nemusí překreslovat každou změnu zvlášť. (Copes, 2020)

React používá technologii nazvanou JSX[9]. Syntaxe je podobná kombinaci HTML, CSS a jazyku JavaScript, ale výsledkem je pouze JavaScript. Zjednodušuje tak tvorbu uživatelských rozhraní. Umožňuje používat výrazy, vkládání větších bloků HTML a snadné použití stylů pomocí CSS. Překlad provádí transpilátor *Babel*. Všechny komponenty v *JSX* musí být vždy uvnitř jednoho elementu, nejčastěji `<div>` nebo speciální `<React.Fragment>`[10], umožňující vrácení skupiny elementů bez uzlů navíc.

```
ReactDOM.render(
  React.DOM.div(
    { id: 'bez-jsx' }
    React.DOM.h1(null, 'Nadpis'),
    React.DOM.p(null, 'Obsah')
  ),
  document.getElementById('root')
)
```

Zdrojový kód 12: React: tvorba elementu bez použití JSX

[8]Single-page application - jednostránková aplikace, kde je veškerý obsah tvořen jednou stránkou.

[9]JSX - JavaScript eXtension

[10]také lze použít krátkou syntaxi <>

```
ReactDOM.render(
  <div id="jsx">
    <h1>Nadpis</h1>
    <p>Obsah</p>
  </div>
  document.getElementById('root')
)
```

Zdrojový kód 13: React: tvorba elementu s použitím JSX

Pro použití *JavaScriptu* v *JSX* je jej nutné umístit do složených závorek `{}`.

```
const title = 'nadpis';
const elements = ['jedna', 'dva', 'tri'];
ReactDOM.render(
  <div>
    <h1>{title}</h1>
    <ul>
      {elements.map((value, i) => {
        return <li>{value}</li>
      })}
    </ul>
  </div>
  document.getElementById('root');
)
```

Zdrojový kód 14: React: JavaScript uvnitř JSX

7.4.1 Životní cyklus

Životní cyklus komponenty má podle (Facebook, 2021b) tři fáze: připojení, aktualizace a odpojení.

Fáze připojení se stará o připojení komponenty do DOM a používá čtyři funkce, `constructor`, `getDerivedStateFromProps`, `render` a `componentDidMount`.

- `Constructor` je první metoda, která se zavolá při připojení komponenty, obvykle se zde inicializuje stav.
- `getDerivedStateFromProps()` se používá pro aktualizaci stavu komponenty na základě proměnné `props`[11]. Vrací objekt s aktualizovanými údaji stavu nebo `null` v případě, kdy se stav nemění.
- `render()` je funkce vracející *JSX*.
- `componentDidMount()` je obvykle místo, kde se provádí volání externího aplikačního rozhraní. Spustí se po připojení komponenty do DOM.

Fáze aktualizace používá pět funkcí. Volají se při každé aktualizaci komponenty. (Facebook, 2021b)

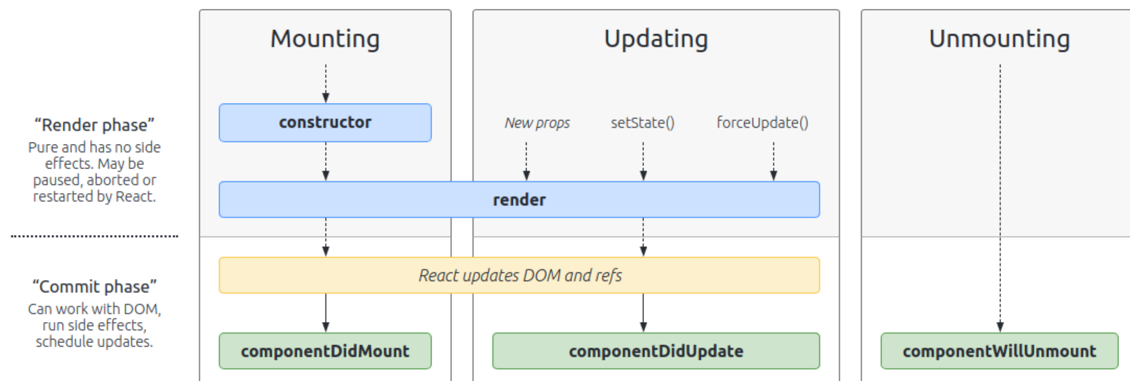
- `getDerivedStateFromProps()` a `render()` má stejnou funkci jako při připojení.

[11]Pole hodnot předaných komponentě.

- `getSnapshotBeforeUpdate()` poskytuje přístup k předchozímu stavu a `props`[11] společně s aktuálními.
- `componentDidUpdate()` je podobná `componentDidMount`, spustí se před aktualizací DOM.

Fáze odpojení používá pouze 1 funkci, `componentWillUnMount`.

- `componentWillUnmount()` se spustí před odstraněním komponenty z DOM. Používá se například pro třídění nebo čištění.



Obrázek 5: React: životní cyklus podle (Wojciech, 2021)

7.4.2 Hooky

Hooky jsou jedna z funkcí přidaných do *Reactu* ve verzi 16.7. Mění způsob vytváření stavů komponent a reagování na události životního cyklu aplikace. Dříve mohly mít stav pouze komponenty vytvořené pomocí tříd. Díky *hookům* je možné definovat stav v každé komponentě, použitím funkce `useState`, jenž vrací proměnnou stavu a funkci pro změnu stavu. (Copes, 2020)

```
const Counter = () => {
  const [count, setCount] = useState(0);

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

Zdrojový kód 15: React: Vytvoření stavu pomocí Hooku

Druhým důležitým hookem je `useEffect`, určený k reakci na události životního cyklu. Prvním jeho parametrem je funkce, která se provede při změně komponenty a druhým parametrem je pole stavových proměnných na které má Hook reagovat, aby se mohlo případně omezit její provedení. Pokud `useEffect` nedostane druhý parametr, bude se definovaná funkce provádět při každé změně komponenty. (Copes, 2020)

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

Zdrojový kód 16: React: Hook pro životní cykly

7.4.3 Redux

Redux je knihovna pro správu stavů. Pomáhá jim udržovat konzistenci, usnadňuje běh v různých prostředí a je jednoduchá pro testování. Poskytuje mnoho prostředků usnadnění pro vývojáře, jako zobrazení a vrácení se do předchozího stavu. (Abramov et al., 2021)

II. Vlastní práce

8 Zadání aplikace

Aplikace bude modelována podle architektonického stylu REST. Serverová část bude připojena k relační databázi MariaDB a bude poskytovat aplikační rozhraní. Klientská strana bude svá data získávat pomocí poskytnutého API. Celá aplikace bude umístěna v kontejnerech.

8.1 Serverová část

Serverová část bude implementovaná pomocí programovacího jazyku PHP a frameworku *Symfony*. Důvodem tohoto výběru jsou vlastní zkušenosti s těmito technologiemi. Použité verze budou PHP 8 a *Symfony* 5.2, které jsou v době psaní těmi nejnovějšími. Webovým serverem bude Nginx a procesovým manažerem PHP bude FPM[12]. Databázovým řešením bude relační systém MariaDB verze 10.5.

8.1.1 Požadavky

Serverová část musí splňovat následující požadavky:

- Připojení k databázi
- Funkcionální testy
- Dostupné API s následujícími koncovými body:
  - Listování všech obrazů a jejich verzí
  - Získání detailů jedné verze obrazu
  - Listování verzí Docker Compose
  - Vygenerování archivu s konfiguračními soubory Docker
- Dokumentace API
- Validaci vstupních dat

8.2 Datová struktura

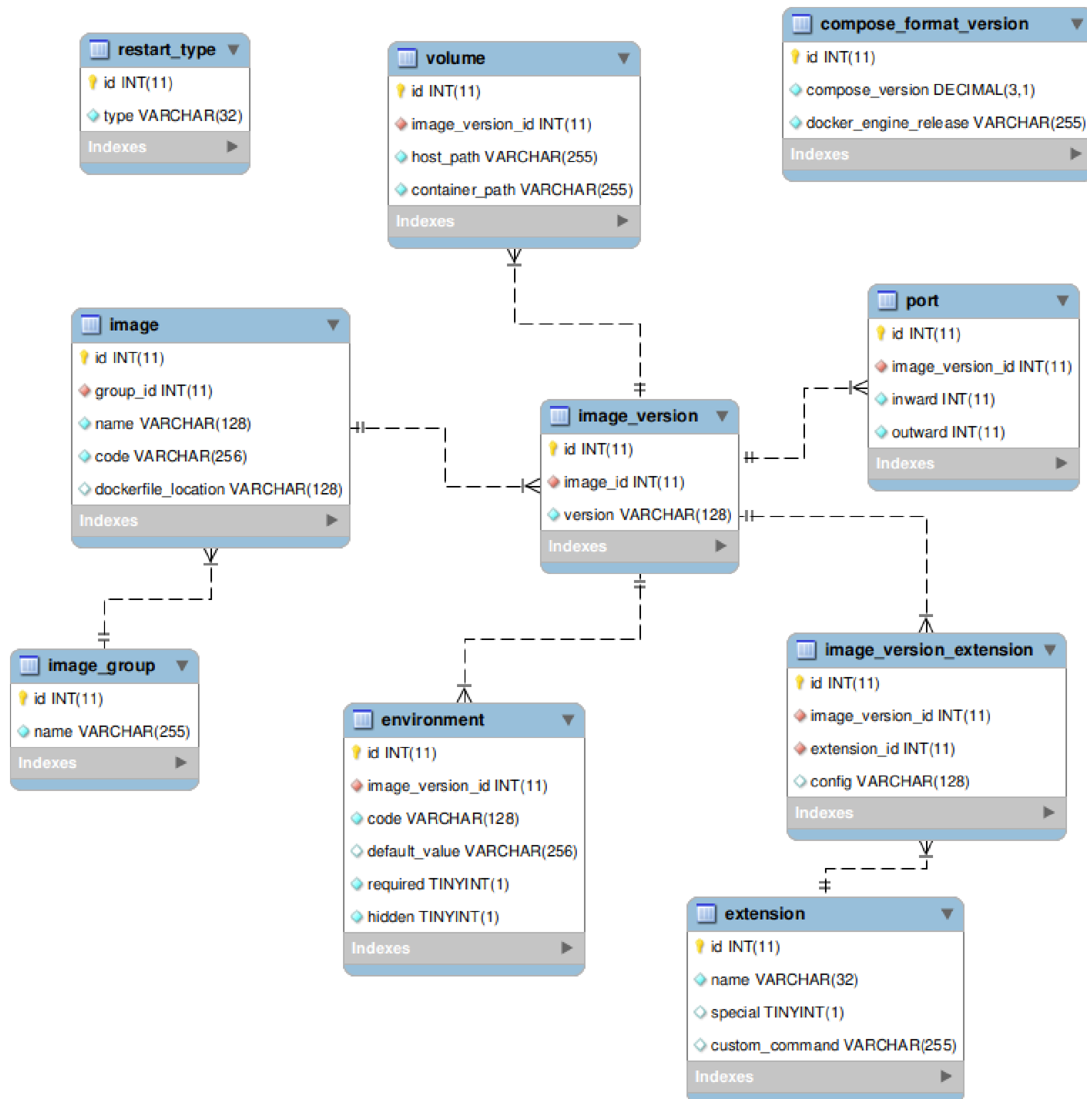
Hlavní entitou bude obraz, která bude obsahovat svůj název, kód, skupinu pro snazší uspořádání, kolekci svých verzí a pro případ, kdy bude potřeba vytvořit svůj vlastní obraz pomocí souboru *Dockerfile*, atribut cesty jeho umístění.

Každá verze obrazu bude mít svůj název, podporované rozšíření, environmentální proměnné, porty a svazky. Rozšíření budou rozdělené do dvou skupin. Obecné, určené pro všechny verze daného obrazu a speciální, specifické pro určitou verzi. Environmentální proměnné bude mít každá verze vlastní, z důvodu možných rozdílů mezi jednotlivými verzemi. Entita environmentální proměnné bude obsahovat svůj kód, výchozí hodnotu, označení jestli je proměnná povinná a zdali bude pro

[12] FastCGI Process Manager - procesový manažer PHP, který společně s Nginx nahrazuje obvyklý webový server, Apache.

uživatele skryta. U portů bude definováno výchozí číslo pro aplikaci uvnitř kontejneru a číslo pro hostující počítač, každá entita tak bude definovat směrování portu. Svazky budou mít údaj o typické cestě k datům aplikace uvnitř kontejneru a výchozí cestu v hostujícím počítači, pro případné zrcadlení.

Další entitou bude verze Compose formátu, která bude skladovat informace o všech dostupných verzí Compose a příslušných verzí Docker Engine, vypsaných v kapitole 4.7.1.1. Všechny možné typy chování při restartu budou uloženy v entitě typů restartů.



Obrázek 6: Entitně vztahový model

8.3 Klientská strana

Klientská strana bude implementována pomocí jazyku *TypeScript*, společně s knihovnami *React* a *Redux*. Výběr tohoto prostředí je z důvodu jednoduché správy stavů v aplikaci, která bude mít velký vliv například při implementaci závislostí mezi

obrazy. Pro stylování komponent bude zahrnuta knihovna *Material UI* obsahující předdefinované styly. O čitelnost a udržování standardů při psaní kódu se bude starat nástroj *ESLint* a *Prettier*.

8.3.1 Požadavky

Klientská část musí splňovat následující požadavky:

- Připojení k API
- Výběr jednoho a více obrazů
- Nastavení vlastností obrazu
- Validaci odesílaných dat
- Responzivní rozhraní

8.3.2 Rozhraní

Rozhraním klientské strany bude interaktivní formulář s kontrolovanými stavy.

Obrázek 7: Drátový model

Aplikace se pokusí zobrazit dva bloky obrazů vedle sebe. V případě nižších rozlišení je zobrazí po jednom, pod sebe.

8.4 Správa verzí

Pro správu verzí kódu aplikace bude použit systém *Git*, pomocí kterého bude snadné prohlížet historii kódu. V případě potřeby umožní vrácení kódu do předchozí verze. Zároveň bude jednoduché udržet všechny části projektu pohromadě.

Každý projekt bude mít svůj repozitář, klient i server. Oba dva pak budou přidány do třetího jako submoduly, čímž se docílí sjednocení celého projektu. Třetí repozitář bude obsahovat popisy pro tvorbu všech obrazů a konfigurační i záznamovými soubory. Ve vývojovém prostředí bude navíc obsahovat data z databáze, připojené pomocí svazku.

Webovým repozitářem bude *GitHub* od společnosti *Microsoft*. Další volbou by mohl být například *GitLab* či *Bitbucket*. Všechny tyto repozitáře by splňovaly potřeby pro projekt v této práci a výběr zrovna *GitHubu* nemá žádný bližší důvod.

Všechny projekty této práce budou veřejně dostupné na následující adrese: `https://github.com/lukasbrabenec/docker-generator`.

9 Implementace serverové strany

9.1 Příprava prostředí

Pro běh aplikace bude zapotřebí prostředí *PHP* verze 8, databáze *MariaDB*, a protože bude použit procesový manažer FastCGI, bude potřeba webový server Nginx, jelikož nebude k PHP žádný přibalen. K vytvoření vývojového prostředí bude použit *Docker* a *Docker-Compose*. Serverová část bude rozdělena do tří částí, API, databáze a webového serveru. Každá část bude mít svůj kontejner. Hostující počítač bude mít s kontejnery vytvořené svazky pro kořenový adresář API, konfigurační soubory webového serveru a logy. Databáze bude mít pro hostující systém otevřený port 3306, určený ke správě dat pomocí databázového správce jako je *DBeaver* nebo *Navicat*. Kontejner webového serveru bude mít otevřený port 8080, pomocí kterého se k němu, a tedy i k API bude přistupovat.

Databázový kontejner bude založen na obrazu `mariadb:10.5.8` dostupném v repozitáři *Docker Hub*. Obraz přijímá čtyři environmentální proměnné, pomocí kterých bude vytvořená databáze, nový uživatel a heslo pro uživatele *root*.

Pro kontejner API bude vytvořen vlastní obraz vycházející z oficiálního obrazu `php:8-fpm-alpine`. Bude potřebovat program *Composer* a *PHP* rozšíření jako `zip`, pro tvorbu ZIP souborů, `intl`, pro normalizaci textu a `pdo_mysql`, pro připojení k databázi. Program *Composer* se zkopíruje z obrazu `composer`. Instalace rozšíření se provedou použitím instalátoru `php-extension-installer` od *mlocati* (Locati, 2021). Při tvorbě obrazu se zároveň nainstalují knihovny a spustí se migrace nad databází, na kterou bude mít tento kontejner závislost. Přístup do tohoto kontejneru bude potřebovat webový server, a proto se otevře port číslo 9000. Soubor `Dockerfile`, pomocí kterého se obraz vytvoří, je zobrazen v kódu 17

```
FROM php:8-fpm-alpine

COPY wait-for-it.sh /usr/bin/wait-for-it

RUN chmod +x /usr/bin/wait-for-it &&\
   apk --update --no-cache add zip

COPY --from=mlocati/php-extension-installer:latest /usr/bin/
   ↪ install-php-extensions /usr/local/bin/
COPY --from=composer /usr/bin/composer /usr/bin/composer

RUN install-php-extensions zip pdo_mysql intl

CMD composer install ;  wait-for-it database:3306 -- bin/console
   ↪  doctrine:migrations:migrate && bin/console doctrine:
   ↪ fixtures:load --quiet ; php-fpm
```

Zdrojový kód 17: Dockerfile obrazu API

Webový kontejner bude založen na obrazu `nginx:1.19-alpine`, opět dostupném v repozitáři *Docker Hub*. Přístup do kontejneru API se nastaví pomocí konfiguračního souboru `default.conf` (kód 18) umístěném v adresáři `/etc/nginx/conf.d`. Pro vývojové prostředí budou veškeré kódy a konfigurační soubory do kontejnerů

umístěné pomocí svazků, z důvodu snadnějších a rychlejších úprav. Produkční prostředí bude mít tyto soubory uložené přímo v obrazu.

```
server {
  listen 8080 default_server;

  root /var/www/public;
  index index.php index.html index.htm;

  location / {
    try_files $uri $uri/ /index.php$is_args$args;
  }

  location ~ \.php$ {
    try_files $uri /index.php =404;
    include fastcgi_params;
    fastcgi_pass api:9000;
    fastcgi_index index.php;
    fastcgi_buffers 16 16k;
    fastcgi_buffer_size 32k;
    fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
    fastcgi_read_timeout 600;
  }

  location ~ /\.ht {
    deny all;
  }
}
```

Zdrojový kód 18: Konfigurační soubor webového serveru

Kompletní soubor `docker-compose` pro serverové vývojové prostředí je zobrazen v následujícím kódu 19.

```
version: "3"

services:
  database:
    image: mariadb:10.5.8
    environment:
      - MYSQL_DATABASE=${DATABASE_NAME}
      - MYSQL_USER=${DATABASE_USER}
      - MYSQL_PASSWORD=${DATABASE_PASSWORD}
      - MYSQL_ROOT_PASSWORD=${DATABASE_ROOT_PASSWORD}
    ports:
      - "3306:3306"
    restart: "unless-stopped"

  api:
    build:
      context: ./api
    depends_on:
      - database
    environment:
      - DATABASE_URL=mysql://${DATABASE_USER}:${
        ↪ DATABASE_PASSWORD}@database:3306/${DATABASE_NAME
        ↪ }?serverVersion=mariadb-10.5.8
```

```
        expose:
            - "9000"
        restart: "unless-stopped"
        user: "1000:1000"
        volumes:
            - ./api:/var/www
        working_dir: /var/www

    nginx:
        image: nginx:1.19-alpine
        depends_on:
            - api
        ports:
            - "8080:8080"
        restart: "unless-stopped"
        volumes:
            - ./api/public:/var/www/public
            - ./nginx/nginx.conf:/etc/nginx/nginx.conf
            - ./nginx/conf.d/:/etc/nginx/conf.d
            - ./logs:/var/log/nginx
        working_dir: /var/www
```

Zdrojový kód 19: Docker Compose serverového prostředí

9.2 Vytvoření projektu

Projekt byl vytvořen pomocí programu *Composer* příkazem `create-project`, který se postará o vytvoření kostry nového projektu.

```
$ composer create-project symfony/skeleton api
```

Zdrojový kód 20: Composer: vytvoření projektu

9.3 Struktura

Projekt serverové části má následující strukturu:

- **bin** - Spouštěcí soubory jako `console`, pro provádění konzolových příkazů nebo `phpunit`, pro spouštění testů.
- **composer.*** - Soubory správce knihoven.
- **config** - Konfigurační soubory *Symfony* a dalších knihoven.
- **Dockerfile** - Série instrukcí pro vytvoření *Docker* obrazu. Konkretní obsah tohoto souboru pro API je v kódu 17.
- **phpunit.xml.dist** - Konfigurace testů.
- **.env** - Environmentální proměnné projektu. Vloží se do něj proměnné definované v souboru `docker-compose`.
- **.gitignore** - Adresáře a soubory, které má správce verzí *Git* ignorovat.

- **public** - Statické soubory jako obrázky a styly.
- **src** - Všechny zdrojové soubory aplikace.
- **symfony.lock** - Speciální soubor knihovny *Symfony flex*, který je určen k uzamknutí takzvaných *Symfony* receptů. Recepty jsou určeny k aktualizaci konfiguračních souborů mezi jednotlivými verzemi knihoven.
- **templates** - Šablony *Dockerfile* a *Docker Compose*.
- **tests** - Veškeré automatické testy, v případě tohoto projektu, funkcionální testy koncových bodů.
- **var** - Dočasné soubory jako cache a logy.
- **vendor** - Adresář pro stažené knihovny projektu.
- **wait-for-it.sh** - Skript zjišťující, zdali je zadaná služba spuštěná. V tomto projektu je použit při vytváření obrazu, kde zjišťuje, zdali je spuštěná databáze, aby se mohly spustit migrace.

```
bin
composer.json
composer.lock
config
Dockerfile
phpunit.xml.dist
public
src
symfony.lock
templates
tests
var
vendor
wait-for-it.sh
```

Obrázek 8: Struktura nového projektu Symfony

Adresář `src` (source) je dále rozdělen podle účelu jednotlivých souborů. Pro tento projekt jsem se rozhodl na následující struktuře (obrázek 9):

- **Controller** - Controllery podle MVC architektury. Obsahují definice koncových bodů aplikace a spojují entity se zobrazením. V tomto případě se zobrazením rozumí odpovědi serveru typu JSON nebo ZIP archivu.
- **DataFixtures** - Datové migrace. Pouze pro vývojové prostředí, na produkčním se data vkládají přímo, pomocí vygenerovaných SQL skriptů.
- **Entity** - Entitní třídy, které se díky objektově relačnímu mapování (ORM) napojí na databázi. V *Symfony* se používá *Doctrine* ORM. Do tohoto adresáře jsem dále vytvořil podadresář, určený pro objekty přenosu dat DTO (Data Transfer Object).

  - **DTO** - Objekty, které slouží jako vrstva nad entitními třídy. V této aplikaci jsou použity z důvodu oddělení entit pro vytváření *Docker* souborů a ORM entit napojených na databázi. Validace požadavků se bude zpracovávat oproti těmto objektům.

- **EventListener** - Třídy, které naslouchají určitým událostem. V tomto projektu jsem vytvořil posluchače výjimek, který odchytí každou vyhozenou výjimku aplikace a vytvoří z ní JSON odpověď.

- **Exception** - Vlastní předpisy výjimek, které jsou v různých částí aplikace vyhozeny. Definovány jsou dvě, jedna pro chyby vzniklé při nevalidních požadavcích a druhá pro chyby při tvorbě souborů `Dockerfile` a `docker-compose`.

- **Form** - Formulářové objekty. Formuláře by se mohly vykreslit v šablonách HTML, ale tady jsou použity pouze pro spuštění validace požadavků a následné automatické vytvoření DTO.

- **Http** - Definice API odpovědi. Všechny objekty odpovědí koncových bodů budou vycházet z tohoto objekt. Obsahuje formátování těla, které zajistí, aby každé mělo totožnou strukturu.

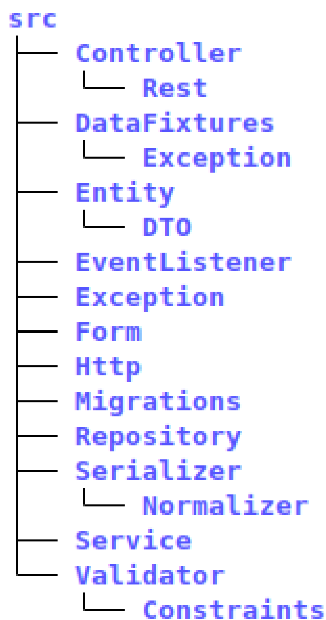
- **Migrations** - Schématické migrace. Obsahují skripty pro vytvoření tabulkové struktury databáze.

- **Repository** - Repozitáře entit. Vrstva mezi databází a entitami. Definuje způsoby, jak dostat data z databáze.

- **Serializer** - Předpisy, jak formátovat objekt. Pro tento projekt jsou vytvořeny dva normalizátory, které se starají o normalizaci objektů před transformací do příslušného formátu (JSON). Jeden pro chyby z formuláře při validaci a druhý pro obecné výjimky, které se mohou stát při běhu programu.

- **Service** - Servisní třídy aplikace. Tento adresář je určen pro služby, které vytváří soubory `Dockerfile` a `docker-compose` nebo ZIP archivy pro následné odeslání souborové odpovědi.

- **Validator** - Předpisy, jak provádět validaci nad příchozími požadavky a v případě nalezení nesrovnalosti, vytvoření chybové hlášky ve formuláři.



Obrázek 9: Struktura zdrojových souborů projektu API

9.4 Knihovny

Pro správný běh aplikace je potřeba prostředí PHP 8 a rozšíření normalizace textu, PDO[13], kontroly datových typů a tvorby ZIP archivů. Zároveň budou nutné knihovny *Symfony* a *Doctrine*, protože na nich bude postavená celá aplikace. Vývojové a testové prostředí má oproti produkčnímu navíc kontrolu syntaxe kódu a další různé knihovny, určené pro testování a ladění. Dokumentaci vytvoří knihovna `api-doc-bundle` od *nelmio*. API musí umožňovat přístup z externího zdroje, musí se tedy vyřešit CORS[14], o to se postará knihovna `cors-bundle`. Všechny potřeby aplikace jsou vypsány v následujícím kódu 21 umístěném v souboru `composer.json`.

```
...
"require": {
  "php": "^8.0",
  "ext-ctype": "*",
  "ext-iconv": "*",
  "ext-pdo_mysql": "*",
  "ext-zip": "*",
  "doctrine/common": "^3",
  "doctrine/doctrine-bundle": "^2",
  "doctrine/doctrine-fixtures-bundle": "^3.4",
  "doctrine/doctrine-migrations-bundle": "^3",
  "doctrine/orm": "^2.8",
  "nelmio/api-doc-bundle": "^3.0",
  "nelmio/cors-bundle": "^2.1",
  "sensio/framework-extra-bundle": "^5.5",
  "symfony/asset": "5.2.*",
  "symfony/console": "^5.2",
  "symfony/dotenv": "^5.2",
```

[13]PHP rozhraní pro práci s SQL databází
[14]Cross-Origin Resource Sharing - sdílení zdrojů z jiné domény

```
    "symfony/flex": "^1.3.1",
    "symfony/form": "^5.2",
    "symfony/framework-bundle": "^5.2",
    "symfony/mime": "^5.2",
    "symfony/monolog-bundle": "^3.5",
    "symfony/serializer": "^5.2",
    "symfony/twig-bundle": "^5.2",
    "symfony/validator": "^5.2",
    "symfony/yaml": "^5.2"
  },
  "require-dev": {
    "squizlabs/php_codesniffer": "3.*",
    "symfony/browser-kit": "5.2.*",
    "symfony/debug-bundle": "^5.2",
    "symfony/maker-bundle": "^1.14",
    "symfony/phpunit-bridge": "^5.2",
    "symfony/var-dumper": "^5.2"
  },
  ...
```

Zdrojový kód 21: Knihovny serverového prostředí

9.5 Koncové body

Rozhraní API poskytuje pět koncových bodů. Adresa každého bodu začíná řetězcem „/api“, následovaným verzí bodu „/v1“ a končící specifikací zdroje, podle stylu REST API, kapitola 5.

- **/api/v1/images**
  - Metoda GET.
  - Vrací všechny dostupné obrazy. Příklad v kapitole 13.1.
- **/api/v1/images/{imageID}**
  - Metoda GET.
  - Povinný parametr identifikátoru obrazu.
  - Vrací detail obrazu.
  - Ukázka v příloze 13.2.
- **/api/v1/compose-versions**
  - Metoda GET.
  - Vrací všechny dostupné verze Docker Compose.
  - Ukázka v příloze 13.3.
- **/api/v1/restart-types**
  - Metoda GET.
  - Vrací všechny dostupné typy restartu.

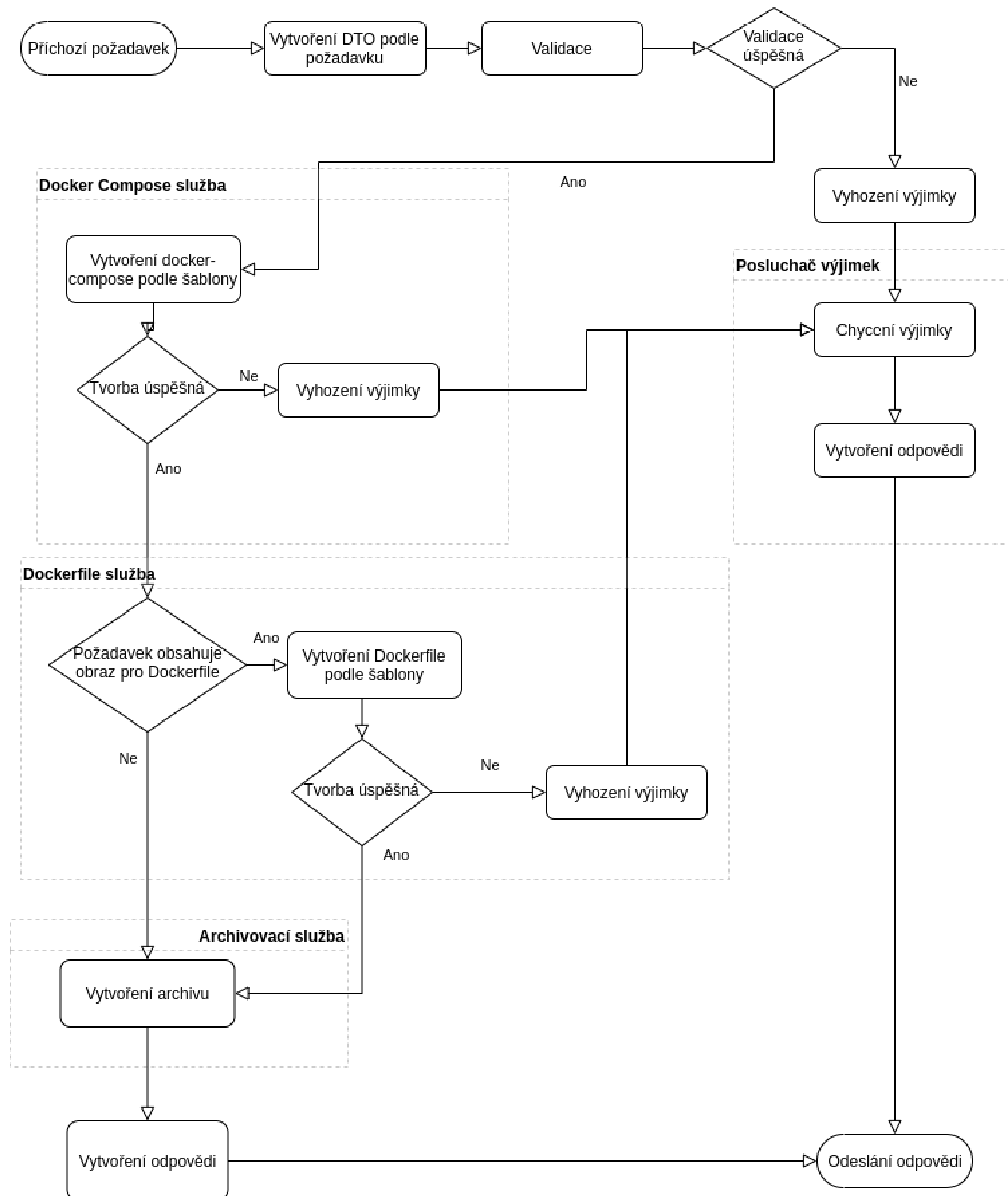
  - Ukázka v příloze 13.4.

- **`/api/v1/generate`**
  - Metoda POST.
  - V těle požadavku definice požadovaného prostředí.
  - Ukázka v příloze 13.5.
  - Vrací ZIP archiv.
  - Proces vytvoření je podrobněji popsán v následující kapitole 9.6.

Kompletní dokumentace aplikačního rozhraní je dostupná na následující adrese: `https://docker.lukasbrabenec.cz/api/doc`. Byla vytvořena pomocí knihovny `api-doc-bundle`, která ji vygenerovala ve formátu OpenAPI verze 2, což je standard projektu *Swagger*, starající se o popisování REST API.

9.6 Proces vytvoření popisu Docker prostředí

9.6.1 Diagram



Obrázek 10: Diagram tvorby popisu Docker prostředí

9.6.2 Controller

Proces vytvoření archivu s popisem *Docker* prostředí začíná v *controlleru*. Po příchodu požadavku se vytvoří nová instance formuláře a vloží se do ní data z těla požadavku, což spustí validaci. Pokud validace nalezne chybu, vyhodí se výjimka společně s obsahem formuláře, o který se následně postará posluchač výjimek, kde se vytvoří odpověď pro klienta. V případě, kdy se požadavek uzná jako validní,

přejde se do služby pro vytvoření příslušných popisů *Docker* prostředí. Poté se vytvořený archiv odešle klientovi a ihned po odeslání se smaže, aby nezabíral místo na serveru.

```
#[Route(
    '/generate',
    requirements: ['version' => 'v1'],
    methods: ['POST']
)]
public function generate(
    Request $request,
    GeneratorService $generatorService
): BinaryFileResponse {
    $generateDTO = $this->processForm($request, GenerateFormType
        ↪ ::class);
    $zipFilePath = $generatorService->generate($generateDTO);

    $response = new BinaryFileResponse($zipFilePath);

    $disposition = HeaderUtils::makeDisposition(
    HeaderUtils::DISPOSITION_ATTACHMENT,
        sprintf('%s.zip', $generateDTO->getProjectName())
    );
    $response->headers->set('Content-Disposition', $disposition);
    $response->deleteFileAfterSend(true);

    return $response;
}

protected function processForm(
    Request $request,
    string $formClass
): GenerateDTO {
    $form = $this->createForm($formClass);
    $form->submit($this->getJSON($request)[$form->getName()]);
    if (!$form->isValid()) {
        throw new FormException($form);
    }

    return $form->getData();
}
```

Zdrojový kód 22: Koncový bod tvorby archivu s popisem Docker prostředí

9.6.3 Validátor

Validace se spustí ihned po vytvoření instance formuláře a vložení dat. Nad parametrem verze Compose je vytvořen vlastní validátor, který zkontroluje, zdali příchozí identifikátor existuje v databázi. Další validátor je vytvořen nad celým objektem verze *Docker* obrazu. Posledně zmíněný validátor zkontroluje existenci následujících věcí:

- Identifikátoru verze obrazu.

- Verze obrazu v databázi.
- Rozšíření a jestli existují pro danou verzi obrazu.
- Environmentálních proměnných a jejich platnost pro verzi. Zároveň se kontroluje, zdali požadavek obsahuje všechny povinné proměnné.
- Portů a v případě, kdy je požadováno odkrytí portu pro hosta nebo ostatní kontejnery, zdali příslušný port existuje v požadavku.
- Svazků.
- Typu restartu.
- Závislého obrazu a také, zdali takový obraz existuje v požadavku.

```
private function validatePorts(
    ImageVersionConstraint $constraint,
    ImageVersionDTO $imageVersionDTO,
): void {
    /** @var PortDTO $portDTO */
    foreach ($imageVersionDTO->getPorts() as $portDTO) {
        if ($portDTO->isExposedToHost()
            && (!$portDTO->getInward() || !$portDTO->getOutward())
        ) {
            $this->context->buildViolation($constraint->
                ↪ missingInwardAndOutwardPort)
                ->atPath('ports')
                ->addViolation();
        }
        if ($portDTO->isExposedToContainers()
            && !$portDTO->getOutward()
        ) {
            $this->context->buildViolation($constraint->
                ↪ missingInwardPort)
                ->atPath('ports')
                ->addViolation();
        }
    }
}
```

Zdrojový kód 23: Serverová validace portů

9.6.4 Tvorba Docker souborů

Objekt přenosu dat DTO se po validaci dostává do služby tvorby *Docker* souborů. Nejprve se vytvoří soubor instrukcí pro *Docker Compose* a poté, v případě potřeby, soubory `Dockerfile`. Následně se vytvoří ZIP archiv a vloží se do něj tyto předpisy. Každý krok tohoto procesu má vlastní službu.

```
public function generate(GenerateDTO $requestObject): string
{
    $this->getDockerComposeGenerator()->generate($requestObject);
    $this->getDockerfileGenerator()->generate($requestObject);
```

```
    return $this->getZipGenerator()->generateArchive(
        ↪ $requestObject);
}
```

Zdrojový kód 24: Funkce obalující tvorbu Docker souborů a archivu

Služba pro tvorbu docker-compose souboru použije šablonovací systém *Twig*, který je volitelnou součástí *Symfony* frameworku. Šablona obsahuje skupinu podmínek a cyklů, které projdou atributy objektu přenosu dat DTO a vloží jej do šablony. Výsledkem je soubor formátu YAML. Celá šablona je v příloze 14.1.

Dockerfile služba se spustí pouze tehdy, obsahuje-li objekt přenosu dat informaci o umístění *Dockerfile*, kterou vezme z entity obrazu v databázi. Každý takový obraz potřebuje vlastní šablonu. V případě, kdy by obraz obsahoval informaci o *Dockerfile* a nebyla by vytvořena související šablona, systém by vyhodil výjimku. Šablona obrazu obvykle obsahuje instrukce, jak nainstalovat knihovny specifické pro dané prostředí a pro obecné aplikace typu *Git* nebo *Bash*. Ukázka takové šablony je v příloze 14.2.

Po vygenerování souborů z šablon následuje vložení do archivu. Služba nejprve vytvoří dočasný soubor na serveru, do kterého následně začne vkládat jednotlivé soubory systému Docker. Cestu k vytvořenému archivu nakonec vrátí zpět do controlleru. Odesláním klientovi se tento soubor automaticky vymaže.

```
public function generateArchive(
    GenerateDTO $generateDTO
): string {
    $zipFilePath = stream_get_meta_data(tmpfile())['uri'];

    $zip = new ZipArchive();
    $zip->open($zipFilePath, ZipArchive::CREATE);
    $zip->addFromString(
        'docker-compose.yml',
        $generateDTO->getDockerComposeText()
    );

    /** @var ImageVersionDTO $imageVersion */
    foreach ($generateDTO->getImageVersions() as $imageVersionDTO
        ↪ ) {
        if ($imageVersionDTO->getDockerfileLocation()) {
            $zip->addFromString(
                $imageVersionDTO->getDockerfileLocation().'
                    ↪ Dockerfile',
                $imageVersionDTO->getDockerfileText()
            );
        }
    }
    $zip->close();

    return $zipFilePath;
}
```

Zdrojový kód 25: Archivace souborů

9.7 Testy

Pro všechny koncové body byly vytvořeny funkcionální testy, které otestují aplikaci jako celek. U každého se otestuje, zdali vrací odpověď se správným stavovým kódem. Koncové body vracející entity z databáze mají navíc napsané testy na tělo odpovědi, kde se kontroluje struktura a obsah. U koncového bodu pro vygenerování *Docker* popisu se otestuje, jestli odpověď obsahuje hlavičku `Content-Type` s hodnotou `application/zip` a `Content-Disposition` obsahující `attachment; filename=test.zip`, které říkají, že odpověď je ZIP archiv s názvem `test.zip`.

```
public function testListGet()
{
    $client = static::createClient();

    $client->request('GET', '/api/v1/restart-types');

    /** @var stdClass $responseBodyObject */
    $responseBodyObject = json_decode($client->getResponse()->
        ↪ getContent());

    $this->assertEquals(200, $client->getResponse()->
        ↪ getStatusCode());
    $this->assertResponseHeaderSame('Content-Type', 'application/
        ↪ json');
    $this->assertObjectHasAttribute('message',
        ↪ $responseBodyObject);
    $this->assertObjectHasAttribute('data', $responseBodyObject);
    $this->assertGreaterThan(0, count($responseBodyObject->data))
        ↪ ;

    foreach ($responseBodyObject->data as $image) {
        $this->assertObjectHasAttribute('id', $image);
        $this->assertObjectHasAttribute('type', $image);
    }
}
```

Zdrojový kód 26: Test koncového bodu pro typy restartu

10 Implementace klientské strany

10.1 Příprava prostředí

Vývojové prostředí pro jazyk TypeScript a React potřebuje mít nainstalovaný *Node*. Součástí *Node* jsou i nástroje *npm*, *yarn* a *npx*[15]. Použije se *Docker* obraz, dostupný v repozitáři *Docker Hub*, `node:14`. Knihovny *Reactu* vytvoří při spuštění vývojový server s výchozím portem 3000, který se napojí na hostující počítač a ten tak dostane přístup k vyvíjeným stránkám. Ve chvíli, kdy bude tento projekt připraven na produkční prostředí, spustí se skripty knihovny *React*, které zkompilují všechny soubory do statických. Takové soubory se poté pouze přidají do webového serveru *Nginx*, který k nim bude poskytovat přístup.

```
...
client:
   image: node:14
   depends_on:
      - nginx
   ports:
      - "3000:3000"
   volumes:
      - ./client:/usr/src/app
   working_dir: /usr/src/app
   command: ["npm", "start"]
...
```

Zdrojový kód 27: Docker Compose klientského prostředí

Nastavení *Docker Compose* v kódu 27 je součástí předchozího, z kódu 19. Klient je závislý na službě *nginx*, protože pro svůj správný běh potřebuje přístup k serverové straně a jejím datům z databáze.

10.2 Vytvoření projektu

Vytvoření projektu *React* se provádí pomocí nástroje *npx*. Do příkazu se přidá argument `template` s hodnotou `typescript`, který zajistí, aby byl projekt připraven na vývoj s jazykem *TypeScript*.

```
$ npx create-react-app client --template typescript
```

Zdrojový kód 28: Vytvoření projektu React

10.3 Knihovny

Mimo již zmíněné knihovny *React* a *TypeScript* bude projekt potřebovat *Redux* pro správu stavů aplikace a *Material-UI*, pro předdefinované styly uživatelského rozhraní. Vývojové prostředí bude mít závislost na nástroji *ESLint* a *Prettier* pro udržení standardů a čitelnosti kódu. Všechny knihovny k sobě také dostanou svoje typy pro *TypeScript*, které nejsou jejich automatickou součástí.

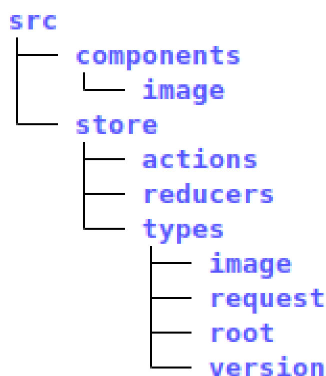
[15] nástroj určený ke spouštění *Node* balíčků

```
...
"dependencies": {
  "@material-ui/core": "^4.11.0",
  "@material-ui/icons": "^4.11.2",
  "@material-ui/lab": "^4.0.0-alpha.56",
  "@material-ui/styles": "^4.10.0",
  "@types/material-ui": "^0.21.8",
  "@types/node": "^14.14.2",
  "@types/react": "^16.9.53",
  "@types/react-dom": "^16.9.8",
  "@types/react-helmet": "^6.1.0",
  "@types/react-redux": "^7.1.9",
  "react": "^16.13.1",
  "react-dom": "^16.13.1",
  "react-redux": "^7.2.1",
  "react-scripts": "^4.0.0",
  "redux": "^4.0.5",
  "redux-thunk": "^2.3.0",
  "typescript": "4.0.3"
},
"devDependencies": {
  "@typescript-eslint/eslint-plugin": "^4.14.1",
  "@typescript-eslint/parser": "^4.14.1",
  "eslint": "^7.18.0",
  "eslint-config-airbnb-typescript-prettier": "^4.1.0",
  "eslint-config-prettier": "^7.2.0",
  "eslint-import-resolver-typescript": "^2.3.0",
  "eslint-plugin-import": "^2.22.1",
  "eslint-plugin-jsx-a11y": "^6.4.1",
  "eslint-plugin-prettier": "^3.3.1",
  "eslint-plugin-react": "^7.20.0",
  "eslint-plugin-react-hooks": "^4.2.0",
  "prettier": "^2.2.1",
  "source-map-loader": "^2.0.0",
  "ts-loader": "^8.0.14"
}
...
```

Zdrojový kód 29: Knihovny klientského prostředí

10.4 Struktura projektu

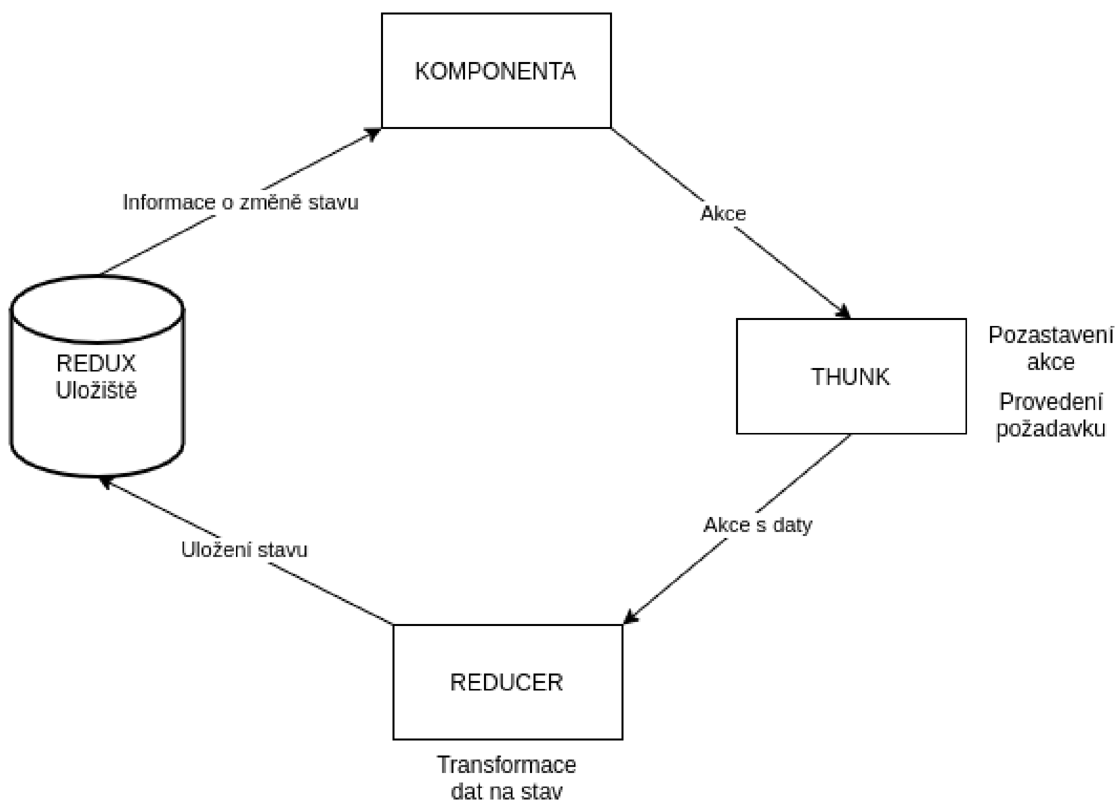
Kořenový adresář obsahuje popis závislostí, skriptů a metadat umístěných v souboru `package.json`. Stejně jako serverová, má klientská strana všechny vývojové zdroje v adresáři `src`. Ten je dále rozdělen na dvě složky, `components` a `store`. `Components` je určena pro všechny JSX komponenty *Reactu* a `store` je pro objekty obsahující logiku ohledně správy stavů, datové typy a odesílání požadavků na server.



Obrázek 11: Struktura projektu klienta

10.5 Princip fungování

Otevřením stránky ve webovém prohlížeči se provedou dvě věci, vykreslení základního formuláře a vytvoření úložiště pro správu stavů pomocí knihovny *Redux*. Stavy jsou rozděleny do tří částí: obrazy, verze *Compose* a požadavek. Všechny tyto stavy dostanou své výchozí hodnoty, které jsou definovány v *reducerech*. Objekty *reducerů* se starají o zachycení určitého typu akce a podle toho vykonají definovanou operaci nad stavem aplikace. Mezi komponentou a *reducerem* je navíc, pomocí knihovny `redux-thunk`, další proces, který pozastaví akci a provede další operace jako asynchronní požadavek na server.



Obrázek 12: Tok dat v aplikaci

Vykreslením formuláře začne komponenta naslouchat změnám stavů a zároveň se automaticky vyšle akce pro získání obrazů a verzí *Compose*. Při zachycení akce se odešle požadavek na server, vyhodnotí se jeho odpověď a předá se *reduceru*, který uloží data obrazů a verzí do stavu aplikace. Komponenty formuláře následně mění svojí strukturu na základě stavu. V případě, kdy je odpověď úspěšná, vytvoří prvek rozbalovacího seznamu se všemi obrazy. V případě, kdy je odpověď neúspěšná, zobrazí chybovou hlášku.

```
export const initImages = (): AppThunkAction => {
  return (dispatch: AppThunkDispatch<ImagesState>) => {
    fetch('https://docker.lukasbrabenec.cz/api/v1/images')
      .then((res: Response) => {
        if (!res.ok) {
          throw new Error('Loading images failed!');
        }
        return res.json();
      })
      .then((data) => {
        dispatch({
          type: INIT_IMAGES_SUCCESS,
          images: data.data
        });
      })
      .catch((err) => {
        dispatch({
          type: INIT_IMAGES_ERROR,
          imagesError: err.message,
        });
      });
    };
};
```

Zdrojový kód 30: Akce vytvoření požadavku pro obrazy

10.5.1 Výběr obrazu

Výběrem obrazu se vytvoří další akce, která odešle požadavek na server pro detail obrazu, jehož odpověď se opět uloží do stavu a vykreslí se komponenta obalující možnosti pro zvolený obraz. Obal v tomto stavu zprvu obsahuje pouze výběr s verzí obrazu. Po zvolení verze se do komponenty obalu přidají prvky pro výběr rozšíření, environmentálních proměnných, portů, svazků a závislostí na jiných obrazech. Každá změna v detailu obrazu vytvoří akci pro úpravu stavu požadavku.

Komponenta obalu obrazu používá mimo stav v *Redux* i vlastní, z důvodu udržení informace o aktuálně zvolené verzi obrazu. Tento stav poté ovlivňuje, zdali se možnost měnit vlastnosti obrazu vůbec zobrazí. V případě, kdy ještě není zvolená verze, uživatel uvidí pouze její výběr, jakmile ji ale zvolí, aktualizuje se lokální stav verze, zobrazí se ostatní vlastnosti a zároveň se odešle výběr do *Redux*, pro pozdější odeslání na server.

Lokální stavy v projektu jsou vytvořené pomocí „hooků“ podle kapitoly 7.4.2. Proměnná stavu aktuálně zvoleného obrazu povoluje dva datové typy, předem definovaný typ verze nebo nedefinovaný (`undefined`), který se použije pouze při

vykreslení, jako jeho výchozí hodnota.

```
const [selectedVersion, setSelectedVersion] = useState<
  ImageVersion | undefined
>(undefined);
```

Zdrojový kód 31: Vytvoření stavu pro zvolenou verzi obrazu

10.5.2 Změny stavu

Stavy v *Reactu* je nutné brát jako immutabilní objekty, tedy neměnné. Při každé změně je potřeba vytvořit nový objekt a ten vložit do úložiště. V případě tohoto projektu do *Redux*. Důvodů pro to je několik, za prvé to zjednoduší ladění aplikace a sníží možný počet zdroje chyb, protože změna stavů bude probíhat pouze na jednom místě (v *reduceru*). Dalším důvodem je přehlednější a srozumitelnější kód. Žádná funkce nezmění hodnotu stavu bez toho, aniž bychom o tom věděli, čímž získáme předvídatelnost aplikace. Navíc, *JavaScript* je rychlejší při výměně reference staršího objektu za nový, než při úpravě již existujícího. (Copes, 2020)

Typické změny stavů potom vypadají jako v kódu 32. Všechny možné změny stavu jsou definovány jako konstanty. Když přijde akce do *reduceru*, vyhodnotí se její typ pomocí příkazu `switch` a aktualizuje se stav. Akce změny portů odesílá identifikátor aktuálně vybrané verze obrazu a pole nově zvolených portů. Podle přijatého identifikátoru verze se z aktuálního stavu nalezne celý objekt verze a uloží se do proměnné `selectedImageVersion`, pomocí předem definované funkce `getImageVersionFromState`. Zmíněná funkce pouze použije operaci `find` nad polem. Podobně jako výběr zvolené verze se vyberou ostatní obrazy mimo zvolený, pomocí funkce `filter`. Následně se ověří, zdali byla zvolená verze obrazu opravdu nalezena, což by měla logicky být vždy, protože byla předem do stavu přidána při výběru obrazu a nikdy by se uživatel neměl dostat ke změně portů, ale pro úplnost tu musí být. Nakonec se komponentě vrátí její nový stav. Použije se zde takzvaný *spread* operátor v podobě tří teček, který vezme objekt či pole a jeho vlastnosti „roztáhne“ do jiného.

```
const getImageVersionFromState = (
  imageVersionID: number,
): ImageVersion | undefined => {
  return state.imageVersions.find(
    (imageVersion: ImageVersion): boolean =>
      imageVersion.id === imageVersionID,
  );
};
const getImageVersionsWithoutSelected = (
  imageVersionID: number,
): ImageVersion[] => {
  return state.imageVersions.filter(
    (imageVersion: ImageVersion) =>
      imageVersion.id !== imageVersionID,
  );
};

...
case CHANGE_PORTS: {
```

```
    const selectedImageVersion = getImageVersionFromState(
      action.imageVersionID,
    );
    const otherImageVersions = getImageVersionsWithoutSelected(
      action.imageVersionID,
    );
    if (selectedImageVersion !== undefined)
      return {
        ...state,
        imageVersions: [
          ...otherImageVersions,
          { ...selectedImageVersion, ports: action.ports },
      ],
    };
    return state;
  }
  ...
```

Zdrojový kód 32: Aktualizace stavu při změně portů

10.5.3 Validace

Validace na klientské straně probíhá ihned při manipulaci se zobrazenými prvky komponenty. Funkce reagující na změny ve formuláři hned zkontrolují zadanou hodnotu a v případě neplatnosti, vůbec neaktualizují stav v úložišti, ale pouze lokální. Zároveň označí vstupní komponentu jako chybnou a povinnou, aby formulář nešel odeslat.

```
<TextField
  label="Container path"
  value={volume.containerPath}
  onChange={(e: React.ChangeEvent<HTMLInputElement>) =>
    handleVolumesStateChange(e, volume.id, 'containerPath')
  }
  id={`${volume.id}-containerPath`}
  error={volume.containerPath === ''}
  helperText={
    volume.containerPath === ''
      ? 'Path cannot be empty'
      : null
    }
  required={volume.containerPath === ''}
/>
```

Zdrojový kód 33: Definice vstupní komponenty svazku

Komponenta v kódu 33 zobrazí chybu v případě, kdy cesta svazku je prázdná a znemožní odeslání formuláře, díky označení komponenty jako povinné.

10.5.4 Dokončení výběru obrazů

Každým výběrem komponenty ve formuláři se aktualizuje stav požadavku v úložišti. Odeslání požadavku a vytvoření archivu s popisem prostředí pro *Docker* se provede stisknutím tlačítka `GENERATE`. Provedením této operace se odešle akce do

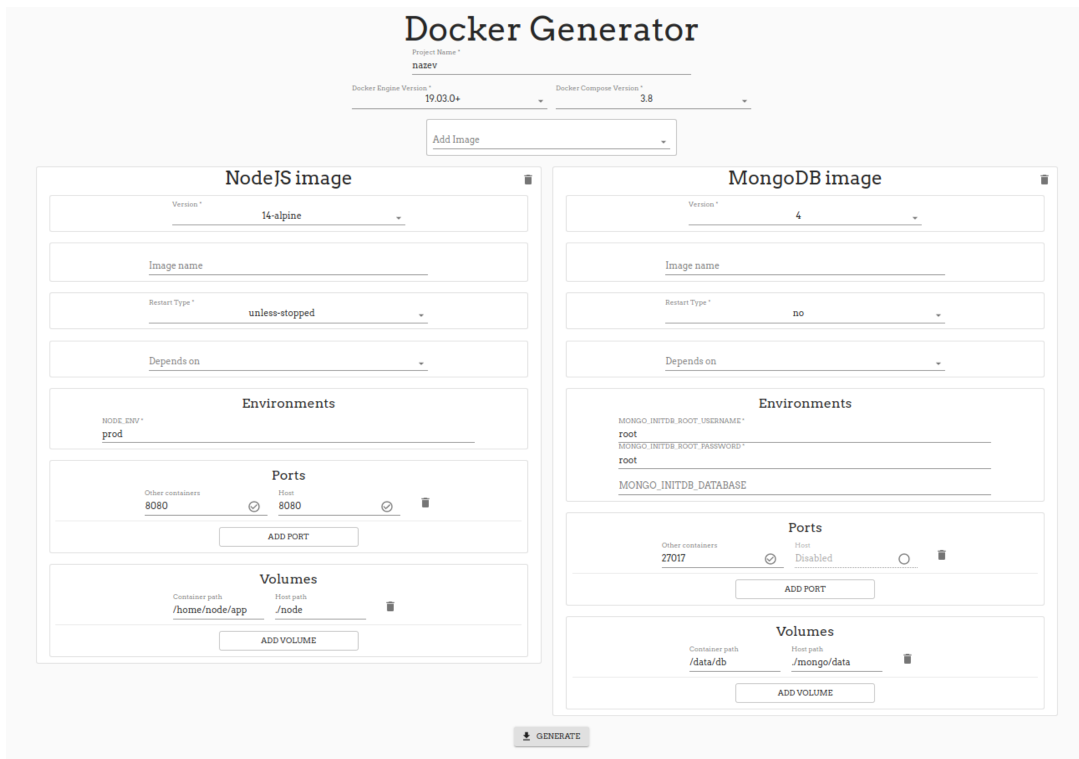
thunk, který vstoupí do centrálního stavu, k objektu požadavku a odešle POST požadavek na server. Po vytvoření archivu na serveru, podle sekce 9.6, se vrátí jako binární soubor v těle odpovědi, načež se uživateli zobrazí dialog pro stažení souboru.

10.5.5 Design

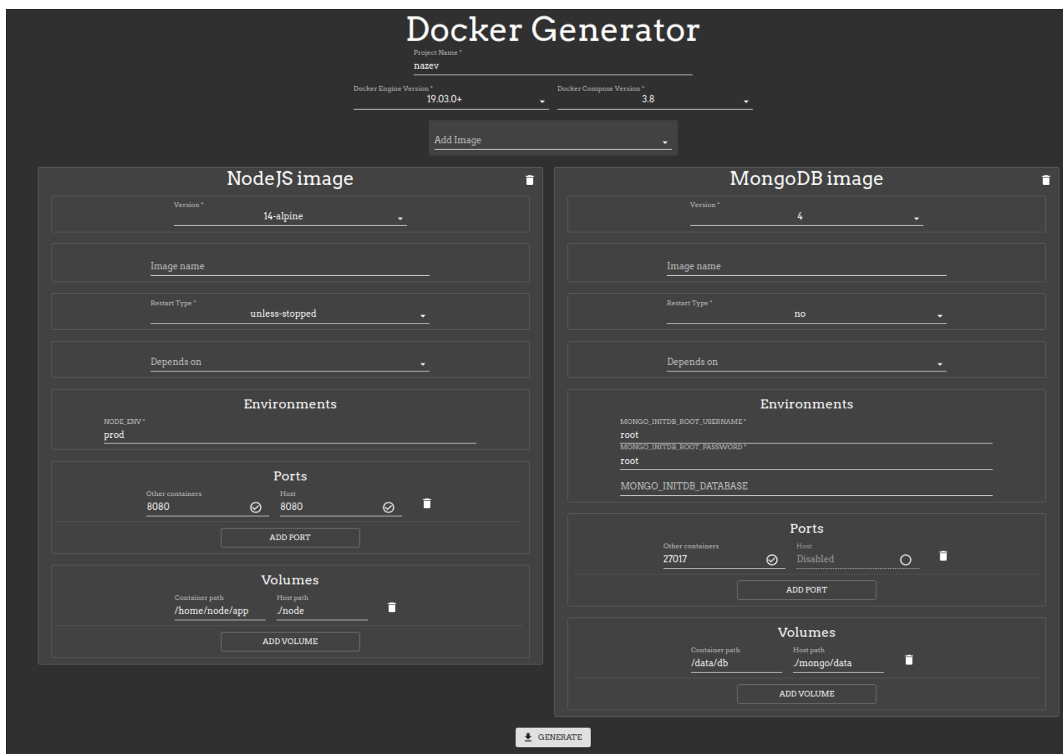
Uživatelské rozhraní je vytvořené pomocí knihovny *Material-UI*, obsahující předdefinované styly. Knihovna obsahuje mnoho komponent, které dle výběru dokážou automaticky vytvářet HTML tagy s CSS styly a usnadní se tak práce při jejich tvorbě. Rozhraní je možné přepínat mezi světlým a tmavým režimem, což je realizováno pomocí globálních stylů *Material-UI* a úložiště `localStorage` prohlížeče, díky kterému je možné zapamatovat si uživatelovu volbu.

```
const [darkState, setDarkState] = useState(
  localStorage.getItem('darkState') === 'true',
);
...
const handleThemeChange = () => {
  setDarkState(!darkState);
  localStorage.setItem('darkState', JSON.stringify(!darkState));
};
```

Zdrojový kód 34: Změna tmavého a světlého režimu



Obrázek 13: Design stránky - světlý režim



Obrázek 14: Design stránky - tmavý režim

Stránka je responsivní a mění své rozhraní na základě rozlišení obrazovky uživatele. V případě menších umístí jednotlivé obrazy pod sebe.

11 Nasazení

Před nasazením na produkční server je vhodné optimalizovat obrazy aplikace, jelikož vývojové obsahují přebytečné věci jako správce knihoven a zbytečně zabírají více místa. Každému obrazu jsem vytvořil nový `Dockerfile` a všechny jsem zahrnul do nového popisu *Docker Compose*, který se postará o sestavení aplikace. Celý postup by bylo vhodné umístit do CI/CD[16] a až tam vytvořit jednotlivé kroky k vytvoření, ale v rámci této práce jsem se rozhodl předvést schopnosti *Docker Compose* a celý postup jsem umístil do *Bash* skriptu.

11.1 Příprava obrazů

Pro obraz API jsem použil takzvaný *multistage build*, který by se dal popsat jako vícestupňové vytvoření obrazu. Podstatou této tvorby je rozdělení postupu na více částí a následný přenos požadovaných souborů mezi stupni. Obraz je rozdělen do tří částí. První vychází z oficiálního obrazu PHP a postará se o nainstalování potřebných PHP rozšíření. Druhý stupeň vychází z obrazu vytvořeného v prvním a jeho úkol je nainstalovat správce knihoven *Composer*, zkopírovat celý API projekt dovnitř obrazu a spustit instalaci závislostí. Třetí stupeň opět vychází z prvního a jediné, co se zde provede je zkopírování celého projekt s již nainstalovanými knihovnami z druhého stupně a otevření portu 9000, aby s ním mohl pracovat webový server. Výsledkem je menší obraz, pouze s nutnými závislostmi pro běh na produkčním prostředí. Všechny kroky stupňů jsou vypsány v následujícím kódu 35, kde jsou označeny pomocí komentářů.

```
# Prvni stupen
FROM php:8-fpm-alpine AS base
COPY --from=mlocati/php-extension-installer:latest \
   /usr/bin/install-php-extensions /usr/local/bin/
RUN apk --update --no-cache add zip &&\
   install-php-extensions zip pdo_mysql intl

# Druhy stupen
FROM base AS build
COPY --from=composer /usr/bin/composer /usr/bin/composer
COPY . /var/www/api
WORKDIR /var/www/api
RUN export APP_ENV=prod \
   && composer install --no-dev --optimize-autoloader \
   && composer dump-env prod

# Treti stupen
FROM base
COPY --from=build /var/www/api /var/www/api
EXPOSE 9000
```

Zdrojový kód 35: Produkční Dockerfile API

Obraz klienta se postará o vygenerování statických souborů, které se poté přesunou přímo do obrazu webového serveru. Bylo by možné vytvořit datový obraz pouze se

[16] kontinuální integrace a nasazení

soubory klienta a napojit ho na kontejner webového serveru, ale pro jednoduchost jsem se to rozhodl udělat takto. Projekt klienta se nakopíruje do obrazu a následně se spustí příkaz `npm install` pro nainstalování knihoven. Poté se příkazem `npm run build` spustí skript definovaný v souboru `package.json`. Vznikne tak nový adresář `build`, který obsahuje statické soubory, vhodné k umístění na web.

```
FROM node:14
COPY . /usr/src/app
WORKDIR /usr/src/app
RUN npm install && npm run build
```

Zdrojový kód 36: Produkční Dockerfile klienta

Webový server *Nginx* bude servírovat oboje aplikace, API i klienta. Obraz proto potřebuje statické soubory obou dvou aplikací a nové, upravené konfigurace. Nastavení vývojového prostředí, v kódu 18, bylo pouze rozšířeno o klientské, v kódu 38

```
FROM nginx:1.19-alpine

#staticke soubory API
COPY ./api/public/ var/www/api/public/
#staticke soubory klienta
COPY ./client/build var/www/client
#konfiguracni soubory
COPY ./nginx/nginx.conf /etc/nginx/nginx.conf
COPY ./nginx/conf.d/ /etc/nginx/conf.d

EXPOSE 8080 8081
```

Zdrojový kód 37: Produkční Dockerfile webového serveru

```
server {
  listen 8081;

  root /var/www/client;
  index index.html index.htm;

  location / {
    try_files $uri $uri/ =404;
  }
}
```

Zdrojový kód 38: Nastavení projektu klienta na webovém serveru

Databáze nepotřebuje vlastní obraz, ten se stáhne přímo z repozitáře *Docker Hub* a pouze se mu předají environmentální proměnné pro jméno a heslo uživatele, pod kterým bude API do databáze přistupovat.

Všechny nové `Dockerfile` jsem umístil do souboru `docker-compose-build.yml`, určeného k snazšímu vytvoření obrazů.

```
version: '3'

services:
  api:
    build:
```

```
            context: ./api
            dockerfile: Dockerfile.prod

    client:
        build:
            context: ./client
            dockerfile: Dockerfile.prod

    nginx:
        build:
            context: ..
            dockerfile: ./nginx/Dockerfile.prod
```

Zdrojový kód 39: Docker Compose pro sestavení produkčních obrazů

Služba *nginx* potřebuje mít přístup do projektu klienta i API, proto má v kódu 39 `context` nastavený jako předchozí adresář a ne přímo specifickou složku jako to mají ostatní obrazy.

11.2 Vytvoření a odeslání obrazů

Posledním krokem je spuštění vytvořeného `docker-compose` v minulém kroku a odeslání do vlastního repozitáře na *Docker Hub*. Obrazy budou uložené v privátním repozitáři, a proto je nejdříve nutné se přihlásit příkazem `docker login`. Po zadání se příkaz zeptá na přihlašovací jméno s heslem a následně bude možné odesílat obrazy do soukromých repozitářů. Poté je potřeba znovu naklonovat repozitáře z GitHubu a až nad nimi spustit `docker-compose`, aby se vytvořila nová, čistá instalace aplikací, bez závislostí vývojového prostředí. Vytvořené obrazy se označí podle názvu repozitáře, příkazem `docker tag` (kód 40). Označené obrazy se odešlou do repozitáře příkazem `docker push` (kód 41).

```
$ docker tag docker-generator_nginx:latest lukasbrabenec/docker-generator:
  ↪ nginx
```

Zdrojový kód 40: Označení obrazu

```
$ docker push lukasbrabenec/docker-generator:nginx
```

Zdrojový kód 41: Odeslání obrazu

Postup jsem automatizoval jednoduchým skriptem. Skript provede všechny zmíněné operace a poté vymaže naklonovaný repozitář a vytvořené obrazy z lokální stanice.

```
#!/bin/bash

git clone --recurse-submodules git@github.com:lukasbrabenec/docker-generator.
  ↪ git build

cd build

docker-compose -f docker-compose-build.yml build --no-cache

docker tag build_nginx:latest lukasbrabenec/docker-generator:nginx && docker
  ↪ tag build_api:latest lukasbrabenec/docker-generator:api
```

```
docker push lukasbrabenec/docker-generator:nginx && docker push lukasbrabenec
  ↪ /docker-generator:api

docker rmi lukasbrabenec/docker-generator:nginx lukasbrabenec/docker-
  ↪ generator:api build_client build_nginx build_api

cd .. && rm -rf build
```

Zdrojový kód 42: Skript vytvoření obrazů

11.3 Nasazení na produkční server

Produkční stroj už má nastavený webový server ve stylu reverzní proxy, jejímž principem je rozdělení požadavků klientů na další servery. Zároveň se stará o SSL certifikáty pro zabezpečení komunikace. Nově vytvořený kontejner *Nginx* je tedy nutné napojit na reverzní proxy.

Když se uživatel bude snažit připojit přes protokol HTTP, a tedy port 80, bude přesměrován do zabezpečeného HTTPS, na port 443. Tomu odpovídá první blok `server` v kódu 43. Poté se rozhoduje, kam bude příchozí požadavek přesměrován, zdali do API nebo na webovou stránku. O to se stará druhý blok. Pokud bude v cestě požadavku klíčové slovo `api` nebo `bundles`, přesměruje se do API, ostatní požadavky budou přesměrovány na webovou stránku. Klíčové slovo `bundles` je zde z důvodu statických souborů dokumentace, a proto je nutné přesměrovat tyto požadavky do API.

```
server {
  listen 80;

  server_name docker.lukasbrabenec.cz;

  return 301 https://docker.lukasbrabenec.cz$request_uri;
}

server {
  listen 443 ssl http2;

  server_name docker.lukasbrabenec.cz;

  location ~ ^/(api|bundles) {
    proxy_pass http://nginx:8080;
  }

  location / {
    proxy_pass http://nginx:8081;
  }
}
```

Zdrojový kód 43: Nastavení reverzní proxy

Nastavení serveru je tímto hotové a stačí pouze stáhnout obrazy z repozitáře *Docker Hub*, zadat jim environmentální proměnné a vytvořit z nich kontejnery. Byl zde vytvořen nový soubor `docker-compose`, který se postará o zbývající nastavení,

jako napojení portu databáze na hostující počítač, typy restartu a připojení na již existující síť, aby měla reverzní proxy přístup k webovému serveru.

```yaml
version: '3'

services:
  database:
    image: mariadb:10.5.8
    environment:
      - MYSQL_DATABASE=${DATABASE_NAME}
      - MYSQL_USER=${DATABASE_USER}
      - MYSQL_PASSWORD=${DATABASE_PASSWORD}
      - MYSQL_ROOT_PASSWORD=${DATABASE_ROOT_PASSWORD}
    ports:
      - "8000:3306"
    restart: 'unless-stopped'

  api:
    image: lukasbrabenec/docker-generator:api
    depends_on:
      - database
    environment:
      - DATABASE_URL=mysql://${DATABASE_USER}:${
        ↪ DATABASE_PASSWORD}@database:3306/${DATABASE_NAME
        ↪ }?serverVersion=mariadb-10.5.8
    restart: 'unless-stopped'

  nginx:
    image: lukasbrabenec/docker-generator:nginx
    depends_on:
      - api
    restart: 'unless-stopped'

networks:
  default:
    external:
      name: nginx-external
```

Zdrojový kód 44: Docker Compose pro produkční server

Environmentální proměnné jsou přidány pomocí souboru .env, umístěným ve stejném adresáři jako soubor docker-compose. Port 8000 hostujícího počítače je přesměrován do portu 3306 kontejneru databáze, z důvodu umožnění přístupu k databázi z „venku“. Tato definice by se jinak dala vynechat.

Posledním krokem je zadání příkazu docker-compose up -d, který stáhne poslední verze obrazů a spustí je na pozadí. Aktualizovat stažené obrazy lze pomocí příkazu docker-compose pull.

12 Závěr

Cílem této práce bylo popsat kontejnerizaci softwaru a její nástroje. Výsledkem měla být aplikace, která má zjednodušit přechod existujících nebo začínajících softwarových projektů do kontejnerů. Cílem aplikace byl vytvořit generátor popisů prostředí pro systém Docker, určených do vývojových prostředí. Kontejnery pro produkční prostředí by se měly tvořit spíše manuálně, podle potřeb projektu, aby byly optimální. Dílčími cíli bylo představení programovacích jazyků s knihovnami aplikace a architektonického stylu REST.

Kontejnerizace, její historie a přínosy byly popsány v první sekci teoretické části. Následně byl popsán její nejpoužívanější nástroj, systém *Docker*. Sekce systému Docker byla dále rozdělena na jeho vysvětlení a na jednotky se kterými pracuje, obrazy a kontejnery. Dále byly vysvětleny soubory a jejich instrukce, pomocí kterých tento systém pracuje. Programovací jazyky PHP a JavaScript, společně s jejich knihovnami a frameworky, byly taktéž popsány v teoretické části této práce. Architektonickým stylem byl zvolen REST, který je vysvětlen v kapitole 5. Tímto byly splněny cíle o popisu kontejnerizace, představení programovacích jazyků a architektonického stylu.

Praktická část byla rozdělena do tří částí, návrh, implementace a nasazení aplikace. Návrh a implementace byly dále rozděleny do dvou dalších částí, podle zvolené architektury aplikace, na serverovou a klientskou.

V návrhu jsou popsány požadavky aplikace, datová struktura, náčrt rozhraní pomocí drátového modelu a systém správy verzí, který byl pro vývoj použit.

Sekce implementace serverové části popsala prostředí, ve kterém bylo vyvíjeno, s použitím kontejnerů a systému Docker. Projekt byl vytvořen pomocí frameworku Symfony, jeho souborová struktura a použité knihovny jsou popsány v této sekci. Dále byly vypsány poskytované koncové body, jaké požadavky přijímají a co vrací za odpovědi. Požadavky a odpovědi jsou dále blíže popsány v dokumentaci API, která je též součástí této sekce. Koncem implementace serverové části byl popsán proces, který generuje instrukce tvorby prostředí systému Docker.

Implementace klientské strany byla zprvu popsána podobným stylem jako sekce serverové, nejdříve příprava prostředí ve kterém bylo vyvíjeno, poté tvorba a struktura projektu. Dále byl popsán proces fungování aplikace a všechny jeho kroky. Závěrem byl představen způsob vytvoření uživatelského rozhraní.

Poslední sekce praktické části obsahuje přípravu všech částí projektu pro produkční prostředí a poté jejich nasazení a zpřístupnění, čímž byl splněn hlavní cíl této práce, vytvoření aplikace podporující kontejnerizaci vývojového prostředí. Veškeré zdrojové kódy projektu jsou dostupné ve veřejném repozitáři popsaném v kapitole číslo 8.4.

Projekt lze snadno rozšiřovat o další obrazy pomocí migračních skriptů databáze.

Seznam použitých zdrojů

ABRAMOV, Dan et al., 2021. *Getting Started with Redux* [online] [cit. 2021-01-29]. Dostupné z: https://redux.js.org/introduction/getting-started.

CODEACADEMY, 2021. *What is REST?* [Online]. Codeacademy [cit. 2021-01-03]. Dostupné z: https://www.codecademy.com/articles/what-is-rest.

COPES, Flavio, 2020. *The React Beginner's Handbook: A React.js Handbook for Front End Developers* [online] [cit. 2021-01-07]. Dostupné z: https://flaviocopes.com/page/react-handbook/.

DOCKER, 2020. *The Compose Specification* [online]. GitHub [cit. 2021-01-03]. Dostupné z: https://github.com/compose-spec/compose-spec/blob/master/spec.md.

DOCKER, 2021a. *Compose file* [online] [cit. 2021-02-09]. Dostupné z: https://docs.docker.com/compose/compose-file/.

DOCKER, 2021b. *Docker Engine overview* [online] [cit. 2021-01-03]. Dostupné z: https://docs.docker.com/engine/.

DOCKER, 2021c. *Orchestration* [online] [cit. 2021-01-08]. Dostupné z: https://docs.docker.com/get-started/orchestration/.

DOCKER, 2021d. *Quickstart: Compose and WordPress* [online] [cit. 2021-02-09]. Dostupné z: https://docs.docker.com/compose/wordpress/.

DOCKER, 2021e. *The Moby Project* [online] [cit. 2021-01-10]. Dostupné z: https://github.com/moby/moby.

DOCKER, 2021f. *What is a Container?* [Online] [cit. 2021-01-08]. Dostupné z: https://www.docker.com/resources/what-container.

DOCKER COMMUNITY, 2021. *php* [online] [cit. 2021-01-03]. Dostupné z: https://hub.docker.com/_/php.

ECMA INTERNATIONAL, 2021. *ECMAScript® 2021 Language Specification* [online] [cit. 2020-12-25]. Dostupné z: https://tc39.es/ecma262/.

FACEBOOK, 2021a. *Getting Started: This page is an overview of the React documentation and related resources.* [Online] [cit. 2020-12-25]. Dostupné z: https://reactjs.org/docs/.

FACEBOOK, 2021b. *React.Component* [online] [cit. 2021-02-25]. Dostupné z: https://reactjs.org/docs/react-component.html.

FIELDING, R. et al., 1999. *Hypertext Transfer Protocol – HTTP/1.1* [Internet Requests for Comments]. RFC Editor, 1999-06. RFC, 2616. RFC Editor. ISSN 2070-1721. Dostupné také z: http://www.rfc-editor.org/rfc/rfc2616.txt.

GARTNER, 2020. Adoption of Cloud-Native Applications and Infrastructure will Drive Growth. *Gartner Forecasts Strong Revenue Growth for Global Container Management Software and Services Through 2024* [online] [cit. 2021-02-07]. Dostupné z: https://www.gartner.com/en/newsroom/press-releases/2020-06-25-gartner-forecasts-strong-revenue-growth-for-global-co.

CHEN, Bin, 2019. *Open Container Initiative (OCI) Specifications* [online]. Alibaba Cloud [cit. 2021-02-10]. Dostupné z: https://www.alibabacloud.com/blog/open-container-initiative-oci-specifications_594397.

IBM, 2019. *Containerization* [online]. 2019-05 [cit. 2021-01-08]. Dostupné z: https://www.ibm.com/cloud/learn/containerization.

LARAVEL, 2020. *Laravel: The PHP Framework For Web Artisans* [online] [cit. 2020-12-20]. Dostupné z: https://laravel.com/docs/8.x/installation.

LINUX FOUNDATION, 2016a. *Image Specification* [online] [cit. 2021-01-10]. Dostupné z: https://github.com/opencontainers/image-spec/blob/master/spec.md.

LINUX FOUNDATION, 2016b. *Runtime Specification* [online] [cit. 2021-01-10]. Dostupné z: https://github.com/opencontainers/runtime-spec/blob/master/spec.md.

LOCATI, Michele, 2021. *Docker PHP Extension Installer: Easy installation of PHP extensions in official PHP Docker images* [online] [cit. 2021-01-29]. Dostupné z: https://github.com/mlocati/docker-php-extension-installer.

LOCKHART, Josh, 2015. *Modern PHP*. Sebastopol: O'Reilly & Associates. ISBN 978-1-4919-0499-2.

MICROSOFT, 2021a. *TypeScript Documentation* [online] [cit. 2020-12-25]. Dostupné z: https://www.typescriptlang.org.

MICROSOFT, 2021b. *What is Docker?* [Online] [cit. 2020-12-29]. Dostupné z: https://docs.microsoft.com/cs-cz/dotnet/architecture/containerized-lifecycle/what-is-docker.

MOZILLA, 2021a. *HTTP headers* [online] [cit. 2021-01-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Headers.

MOZILLA, 2021b. *HTTP request methods* [online] [cit. 2021-01-03]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/HTTP/Methods.

MOZILLA, 2021c. *JavaScript* [online] [cit. 2020-12-25]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript.

NETTE FOUNDATION, 2020. *Proč používat Nette?* [Online] [cit. 2020-12-20]. Dostupné z: https://doc.nette.org/cs/3.1/why-use-nette.

NPM INC., 2021. *npm Docs* [online] [cit. 2020-12-25]. Dostupné z: https://docs.npmjs.com/.

PHP GROUP, 2020. *PHP 8.0 Released* [online] [cit. 2021-01-03]. Dostupné z: https://www.php.net/releases/8.0/en.php.

POTENCIER, Fabien, 2019. *Symfony 5: the fast track*. Clichy: Symfony SAS. ISBN 978-2-918390-37-4.

POULTON, Nigel, 2020. *Docker Deep Dive: Zero to Docker in single book*. Germany: Nigel Poulton. ISBN 978-1-5218-2280-7.

SYMFONY, 2020. *Symfony Documentation* [online] [cit. 2020-12-18]. Dostupné z: https://symfony.com/doc/current/index.html.

WOJCIECH, Maj, 2021. *React Lifecycle Methods diagram* [online] [cit. 2020-12-25]. Dostupné z: https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/.

III. Přílohy

13 Požadavky a odpovědi koncových bodů

13.1 Všechny obrazy


```
{
  "message": "OK",
  "data": [
    {
      "id": 1,
      "name": "PHP",
      "code": "php",
      "group": {
      "id": 1,
      "name": "Development Environment"
    }
},
...
```


Zdrojový kód 45: Odpověď všech obrazů

13.2 Detail obrazu


```
...
{
  "id": 13,
  "name": "MariaDB",
  "code": "mariadb",
  "group": {
    "id": 6,
    "name": "Database"
  },
  "imageVersions": [
    {
      "id": 113,
      "version": "latest",
      "extensions": [],
      "environments": [
        {
          "id": 202,
          "code": "MYSQL_ROOT_PASSWORD",
          "defaultValue": null,
          "required": true,
          "hidden": false
        },
        ...
        ],
      "volumes": [
        {
          "id": 110,
          "hostPath": "./mariadb/data",
          "containerPath": "/var/lib/mysql"
        }
      ],
```

```
      "ports": [
        {
          "id": 116,
          "inward": 3306,
          "outward": 3306,
          "exposedToHost": false,
          "exposedToContainers": true
        }
      ],
      "dependsOn": []
    },
...
```

Zdrojový kód 46: Odpověď detailu obrazu

13.3 Všechny Compose verze

```
{
  "message": "OK",
  "data": [
    {
      "id": 28,
      "composeVersion": 3.8,
      "dockerEngineRelease": "19.03.0+"
    },
    {
      "id": 27,
      "composeVersion": 3.7,
      "dockerEngineRelease": "18.06.0+"
    },
  ...
```

Zdrojový kód 47: Odpověď všech Compose verzí

13.4 Všechny možnosti restartu

```
{
  "message": "OK",
  "data": [
    {
      "id": 5,
      "type": "no"
    },
    {
      "id": 6,
      "type": "always"
    },
  ...
```

Zdrojový kód 48: Odpověď všech typů restartu

13.5 Vytvoření Docker prostředí


```
{
  "dockerVersionID": 14,
  "projectName": "projekt",
  "imageVersions": [
    {
      "id": 1,
      "version": "5.6-alpine",
      "environments": [],
      "volumes": [
        {
          "id": 1,
          "hostPath": "./php",
          "containerPath": "/var/www/html"
        }
      ],
      "ports": [
        {
          "id": 1,
          "inward": 80,
          "outward": 80,
          "exposedToHost": false,
          "exposedToContainers": true
        }
      ],
      "dependsOn": [],
      "imageName": "PHP",
      "restartType": {
        "id": 1,
        "type": "no"
      }
    }
  ],
  }
}
```


Zdrojový kód 49: Požadavek vytvoření prostředí

14 Twig šablony

14.1 Docker Compose

```
  version: '{{ dockerVersion }}'

  services:
     {%- for imageVersion in imageVersions ~%}
     {{ imageVersion.imageName|lower }}:
        {%- if imageVersion.dockerfileLocation is not empty ~%}
        build:
           context: {{ imageVersion.dockerfileLocation }}
        {%- else ~%}
```

```
        image: {{ imageVersion.imageCode }}:{{ imageVersion.
           ↪ version }}
        {%- endif ~%}
        container_name: {{ imageVersion.imageName|lower }}
        restart : '{{ imageVersion.restartType.type }}'
        {%- if imageVersion.environments is defined and
           ↪ imageVersion.environments is not empty ~%}
        environment:
           {%- for environment in imageVersion.environments ~%}
           - {{ environment.name }}={{ environment.value }}
           {%- endfor ~%}
        {%- endif ~%}
        {%- if imageVersion.ports is defined and imageVersion.
           ↪ ports is not empty ~%}
        {%- if imageVersion.anyPortExposedToHost == true ~%}
        ports:
           {%- for port in imageVersion.ports ~%}
           {%- if port.exposedToHost == true ~%}
           - {{ port.inward }}:{{ port.outward }}
           {%- endif ~%}
           {%- endfor ~%}
        {%- endif ~%}
        {%- if imageVersion.anyPortExposedToContainers == true ~%}
        expose:
           {%- for port in imageVersion.ports ~%}
           {%- if port.exposedToContainers == true ~%}
           - {{ port.outward }}
           {%- endif ~%}
           {%- endfor ~%}
        {%- endif ~%}
        {%- endif ~%}
        {%- if imageVersion.volumes is defined and imageVersion.
           ↪ volumes is not empty ~%}
        volumes:
           {%- for volume in imageVersion.volumes ~%}
           - {{ volume.hostPath }}:{{ volume.containerPath }}
           {%- endfor ~%}
        {%- endif ~%}
        {%- if imageVersion.dependsOn is defined and imageVersion.
           ↪ dependsOn is not empty ~%}
        depends_on:
           {%- for dependency in imageVersion.dependsOn ~%}
           - {{ dependency|lower }}
           {%- endfor ~%}
        {%- endif ~%}
    {%- endfor ~%}
```

Zdrojový kód 50: Twig šablona Docker Compose

14.2 Dockerfile

```
FROM {{ imageCode }}:{{ version }}

{% if extensions is defined and extensions is not empty %}
```

```
{% if extensions.system.main is defined and extensions.system.
   ↪ main is not empty %}
{# if image is alpine, change command #}
RUN {% if 'alpine' in version %}apk --update --no-cache add {%
   ↪ else %}apt-get update && apt-get install -y {% endif %}{%
   ↪ for systemMainExtension in extensions.system.main %}{{
   ↪ systemMainExtension.name }} {% endfor %}
{% endif %}
{% if extensions.system.custom is defined and extensions.system.
   ↪ custom is not empty %}
{# if there already are system main extensions -- connect with
   ↪ their RUN command #}
{% if extensions.system.main is defined and extensions.system.
   ↪ main is not empty %}
{{ ' &&\\\n    '|raw }}{% for systemCustomExtension in
   ↪ extensions.system.custom %}{{ systemCustomExtension.
   ↪ customCommand }} {% if loop.first == false %}{{ '&&\\\n'|
   ↪ raw }}{% endif %} {% endfor %}
{% else %}
RUN {% for systemCustomExtension in extensions.system.custom
   ↪ %}{{ systemCustomExtension.customCommand }} {% if loop.
   ↪ first == false %}{{ '&&\\\n'|raw }}{% endif %} {% endfor
   ↪ %}
{% endif %}
{% endif %}

{% if extensions.special is defined and extensions.special is
   ↪ not empty %}

COPY --from=mlocati/php-extension-installer:latest /usr/bin/
   ↪ install-php-extensions /usr/local/bin/

RUN install-php-extensions {% for specialExtension in extensions
   ↪ .special.main %}{{ specialExtension.name }} {% endfor %}
{% endif %}
{% endif %}
```

Zdrojový kód 51: Twig šablona Dockerfile