

ZÁPADOČESKÁ UNIVERZITA V PLZNI
FAKULTA ELEKTROTECHNICKÁ
KATEDRA ELEKTROENERGETIKY A EKOLOGIE

DIPLOMOVÁ PRÁCE

Vizualizace dat z meteostanice

Visualization of weather station data

Autor práce: Bc. Martin Zlámal
Vedoucí práce: Ing. Martin SÝKORA, Ph.D.

Plzeň 2017

Abstrakt

Cílem této diplomové práce je vytvoření takové aplikace, která zvládne zpracovávat výstupní data z meteorologických stanic, dokáže je zobrazit ve webovém prohlížeči a dokáže tato data doplnit o záznam z webové kamery.

Pro vyřešení tohoto zadání byl vytvořen jeden hlavní server, který se chová jako zdroj pravdy a poskytuje vnějšímu světu GraphQL API. Webové rozhraní i všechny meteostanice komunikují se serverem pouze prostřednictvím tohoto API. Kromě hlavního serveru byla vytvořena také minimalistická mikroslužba pro zpracovávání videa z webových kamer.

Vytvořené řešení nabízí možnost připojit jakoukoliv meteorologickou stanicí, protože API není závislé na typu stanice. Server se zaměřuje na přijímané fyzikální veličiny, nikoliv na jejich původ, takže je možné pracovat s libovolným počtem zdrojů dat od libovolných výrobců. API neslouží pouze pro přijímání dat, ale také pro jejich vybavování. Server mimo jiného počítá agregace historických dat a umožňuje jejich export.

Klíčová slova

API, Docker, FFmpeg, GraphQL, HLS, meteorologická stanice, Nette Framework, PHP, React, webová kamera

Abstract

The aim of this diploma thesis is to create an application that can process the output data from weather stations, can show these data in a web browser and can handle and show web camera view.

To solve this problem the master server was created. It acts as the source of truth and provides GraphQL API to the outside world. Web interface and all weather stations communicate with the server only through this API. Minimalist microservice for video processing from webcams was also created in addition to the main server.

Created solution offers the ability to connect any weather station because the API is not dependent on the type of station. Server focuses on the received physical quantities, not their origin, so it can work with any number of data sources from any company. API is not only able to receive data, but it can also provide these data. Server among other things allows the aggregation of historical data and allows them to be exported.

Key Words

API, Docker, FFmpeg, GraphQL, HLS, weather station, Nette Framework, PHP, React, web camera

Prohlášení

Předkládám tímto k posouzení a obhajobě diplomovou práci zpracovanou na závěr studia na Fakultě elektrotechnické Západočeské univerzity v Plzni.

Prohlašuji, že jsem svou závěrečnou práci vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této závěrečné práce je veškerý použitý software legální.

.....
Podpis

V Plzni dne 15. 5. 2017

Bc. Martin Zlámal

Obsah

Seznam obrázků	vi
Seznam symbolů a zkratk	vii
1 Úvod	1
2 Druhy a formáty sdílených dat	3
2.1 Meteostanice v laboratoři EU 411	3
2.1.1 Binární formát dat	4
2.2 IP kamera	6
2.2.1 Streamování videa	8
2.2.2 Způsob připojení kamery do veřejné sítě	10
3 Návrh aplikace	11
3.1 Přenos a ukládání dat z meteostanic	11
3.2 Architektura serverové části aplikace	13
3.2.1 Hexagonální architektura	15
3.2.2 Konkrétní příklad ovládání kontextu	16
3.2.3 Kompletní adresářová struktura	18
3.2.4 Testování aplikace a CI provoz	19
3.2.5 Návrh databáze a databázové migrace	22
3.2.6 Zabezpečení API pro obousměrnou komunikaci	24
3.3 Architektura uživatelského rozhraní	25
3.3.1 Základy práce s Reactem	25
3.3.2 Dočasné Redux úložiště	27
3.3.3 JSX	29
3.3.4 Komunikace se serverem pomocí GraphQL	31
3.4 Architektura streamovací aplikace	33
3.4.1 Zapnutí a vypnutí zpracování streamu	33
3.5 Architektura konkretizačního uzlu	35
3.6 GraphQL API	37
3.6.1 Typy (Types)	39
3.6.2 Dotazy (Queries)	40
3.6.3 Mutace (Mutations)	41

3.6.4	Stránkování v GraphQL	42
3.6.5	Pokročilé možnosti GraphQL	43
3.6.5.1	Aliases	44
3.6.5.2	Fragmenty	44
3.6.5.3	Direktivy	45
3.6.5.4	Introspekce	45
3.6.6	Chybové stavy v GraphQL	46
4	Výsledná aplikace a její provoz	48
4.1	Uživatelské rozhraní	48
4.2	Doporučená infrastruktura	51
4.3	Možnosti rozšíření díky API	54
5	Hodnocení a závěr	56
	Literatura	58

Seznam obrázků

2.1	Meteostanice Technoline WS-3650-IT	4
2.2	IP kamera airCam od společnosti Ubiquity Networks	6
3.1	Základní návrh aplikace	12
3.2	Základní návrh aplikace s mezivrstvou pro API	13
3.3	Hexagonální architektura	17
3.4	UML diagram konkrétních implementací repozitáře	18
3.5	Continuous Integration	21
3.6	Verzování kódu v Gitu	22
3.7	Problém verzování databáze	23
3.8	Aktualizace komponent prostřednictvím Reduxu	28
3.9	GraphQL komunikace se serverem přes API	32
4.1	Celkový pohled na výslednou aplikaci	48
4.2	Úvodní stránka klientské aplikace	49
4.3	Detail meteorologické stanice v aplikaci	50
4.4	Způsoby interpolace grafů	51
4.5	Přehled všech video streamů v aplikaci	52
4.6	Infrastruktura použitá při vývoji	53
4.7	Doporučená infrastruktura	54

Seznam symbolů a zkratek

API	Application Programming Interface
AUFS	Advanced Multi-Layered Unification Filesystem
CDN	Content Delivery Network
CI	Continuous Integration
CLI	Command-Line Interface
DDD	Domain-Driven Design
DHCP	Dynamic Host Configuration Protocol
DI	Dependency Injection
DIC	DI Container
DOM	Document Object Model
HLS	HTTP Live Streaming
HTML	HyperText Markup Language
HTTP	Hypertext Transfer Protocol
HTTPS	HTTP Secure
I/O	Input/Output
ID	Identifier
IP	Internet Protocol
JSON	JavaScript Object Notation
JSX	JavaScript XML
JWT	JSON Web Token
MSE	Media Source Extensions
NAT	Network Address Translation
OOP	Object-Oriented Programming
PoE	Power over Ethernet
REST	Representational State Transfer
RTSP	Real Time Streaming Protocol

SRP	Single Responsibility Principle
SSH	Secure Shell
URL	Uniform Resource Locator
UI	User Interface
UML	Unified Modeling Language
UUID	Universally Unique Identifier
VCS	Version Control System
XML	Extensible Markup Language
XSS	Cross-site scripting

1

Úvod

V dnešní době existují tisíce různých zdrojů dat a tisíce meteorologických stanic od různých výrobců. Přesto není běžně rozšířený žádný společný komunikační protokol. Tato diplomová práce si klade za úkol zamyslet se nad touto skutečností, navrhnout řešení a tento nápad prakticky zrealizovat. Cílem je umožnit získávání dat z meteostanic, jejich ukládání a vizualizace prostřednictvím webové aplikace. Na takový problém je třeba dívat se dostatečně široce. Proto je jako doplňkový úkol nutné obohatit získaná data z meteorologických stanic o obrazový záznam z webových kamer.

Bylo by naivní snažit se vytvořit takovou aplikaci, která bude znát každou meteostanici. Jednak je to vzhledem k jejich počtu nadlidský úkol a potom ne každá meteostanice dokáže data odesílat přes síť na případný server. Výsledná aplikace na základě zadání této diplomové práce vykrystalizovala v poměrně zajímavý návrh. Základem je server, který vnějšímu světu nabízí GraphQL API a to je také jediný způsob, jak lze se serverem komunikovat. Toto API přijímá primárně informace o fyzikálních veličinách s tím, že nezáleží na jejich původu. Server tak tvoří jediný zdroj pravdy. Stále však existuje problém s velkým množstvím různých meteostanic. Ten je vyřešen pomocí převodníků (tzv. konkretizačních uzlů), které mají za úkol porozumět jedné konkrétní meteostanici a odesílat její data přes API na server. Tento konkretizační uzel je doslova několik desítek řádek kódu.

Data přijatá prostřednictvím API jsou zobrazována ve webovém prohlížeči. K tomu je použit systém React komponent, který z toho samého API dokáže data velmi elegantním způsobem získat. Ona elegance spočívá právě v GraphQL, které poskytuje grafové API. Díky tomu je možné dotázat se serveru na konkrétní podmnožinu dat namísto předem definované množiny dat.

Celou situaci příjemně komplikují webové kamery. Ty se totiž zcela vymykají způsobům, kterými komunikují meteostanice. Webová kamera poskytuje tok dat (video záznam), který nelze rozumným způsobem odesílat přes žádné API. Navíc tento tok dat není vhodný pro přímé zpracování v přehrávači videa a je nutné nejprve provést konverzi formátu. Součástí práce je tedy také

mikroslužba, která se o toto předzpracování záznamu stará. Hlavní API server pak pouze informuje tuto službu s žádostí o nové zpracování streamu dat.

Výsledkem je překvapivě komplexní aplikace, která skutečně dokáže zpracovávat data z libovolných meteostanic a dokáže přehrávat videozáznamy z webových kamer v prohlížeči. Největší síla je však v GraphQL API, které z hlediska uživatele sice není nijak vidět, ale z hlediska celého návrhu tvoří hlavní pilíř řešící téměř jakýkoliv myslitelný problém. Důkazem tohoto tvrzení jsou například exporty dat. Exportování nebylo nikdy v plánu vytvářet, ale díky GraphQL lze exportovat jakákoliv data dostupná prostřednictvím API.

Velký důraz při vypracovávání této diplomové práce byl kladen na moderní technologie a postupy. Byly použity poslední verze Nette Frameworku (PHP), Reactu (JS), PostgreSQL a dalších nástrojů a knihoven. Stejně tak byl kladen velký důraz na dobrý návrh architektury, která z velké části vychází z principů DDD a automaticky testovatelného kódu [1].

2

Druhy a formáty sdílených dat

Pro ověření funkčnosti návrhu v této práci jsou k dispozici dvě meteorologické stanice a jedna webová kamera. Z dalších kapitol bude jasné, že na zdroji dat nezáleží, což je hlavní myšlenka, na které vše stojí. Tato úvodní část tedy slouží spíše pro ukázkou toho, jak je oblast meteostanic různorodá z pohledu formátů dat a přenosu informací. Díky tomuto pozorování je výsledná aplikace sice mnohem složitější, ale není závislá na konkrétních zařízeních.

2.1 Meteostanice v laboratoři EU 411

Meteostanice v laboratoři je značky Technoline WS-3650-IT. Tato stanice umí měřit základní fyzikální veličiny týkající se počasí, jako jsou například teplota (venkovní i vnitřní), tlak a relativní vlhkost (opět venkovní i vnitřní). Tato meteostanice umí měřit také rychlost větru (včetně směru a poryvu) a srážky. Pro toto měření je však nutné mít k dispozici přídatný anemometr a srážkoměr. Ty nejsou v laboratoři EU 411 nainstalovány.

K meteostanici bohužel není k dispozici žádná technická dokumentace a výrobce není ochoten tuto dokumentaci poskytnout. Tím pádem je téměř nemožné dobře implementovat programové zpracování dat, které putují přes hardwarové rozhraní COM.

Pro příjem dat musí být stanice připojena k počítači, na kterém je spuštěn program HeavyWeatherPro V1.1. Tento program zobrazuje historii naměřených dat, což jinými slovy znamená, že si je někde musí uchovávat. A právě zde je jedno z možných řešení problému s chybějící technickou dokumentací. HeavyWeatherPro si totiž data ukládá v binárním formátu do jednoho souboru, který je možné programově přečíst a získat všechny potřebné informace. Ve výsledku je tento postup dokonce možná lepší, než snažit se rozluštit probíhající komunikaci. K dispozici je totiž celá naměřená historie, která sahá až na začátek roku 2015.



Obrázek 2.1: Meteostanice Technoline WS-3650-IT

2.1.1 Binární formát dat

Již zmíněný program HeavyWeatherPro V1.1, který je na počítači nainstalován, si ukládá historii dat do souboru `history.dat`. Jedná se o binární formát, kdy každý řádek je 56 bytů dlouhý a obsahuje informace, které jsou podrobně rozepsány v tabulce 2.1.

Parsování takového souboru je např. v PHP možné pomocí vestavěné funkce `unpack`¹. K dispozici jsou základní informace o prostředí, ve kterém se stanice (resp. její senzory) nachází. U většiny položek je z názvu jasné, co znamenají. Výjimku tvoří pouze časové razítko a směr větru. Časové razítko je dáno ve dnech od 30. 12. 1899. To je velmi netradiční a je nutné provést jednoduchý přepočít. Například záznam, který má hodnotu 42116.364583333 dnů

¹Příklad PHP programu, který je schopen parsovat tento formát souboru je k dispozici na adrese <https://gist.github.com/mrtznzlm1/624c55f792f8b821a000bb04f0b0e0ac>. JavaScript varianta je pak popsána později v rámci konkretizačního členu.

Tabulka 2.1: Binární formát dat meteostanice v EU 411

Pozice	Typ (délka)	Význam	Jednotka
00	Double (8)	Časové razítko	Dny od 12/30/1899
08	Float (4)	Absolutní tlak	hPa (mBar)
12	Float (4)	Relativní tlak	hPa (mBar)
16	Float (4)	Rychlost větru	m/s
20	ULong (4)	Směr větru	viz tabulka 2.2
24	Float (4)	Poryv větru	m/s
28	Float (4)	Celkové srážky	mm
32	Float (4)	Nové srážky	mm
36	Float (4)	Vnitřní teplota	Celsius
40	Float (4)	Venkovní teplota	Celsius
44	Float (4)	Vnitřní vlhkost	procento
45	Float (4)	Venkovní vlhkost	procento
52	ULong (4)	neznámé	vždy nula

ve skutečnosti znamená datum 22. 4. 2015 s časem 8:45. Ideální je původní hodnotu přepočítat na sekundy, takže se přičtením této hodnoty zachová dostatečně vysoká přesnost:

```
let days = 42116.364583333;
let seconds = Math.round(days * 24 * 3600);
new Date(Date.UTC(1899, 11, 30, 0, 0, seconds))
// vrátí: 2015-04-22T08:45:00.000Z
```

Směr větru stanice odesílá stanice jako celé číslo v rozsahu 0 - 15. Jedná se tedy o 16 hodnot, kdy azimut je násobkem tohoto čísla a 22.5, jak je vidět v tabulce 2.2.

Tabulka 2.2: Formát větrné růžice a odpovídající azimuty

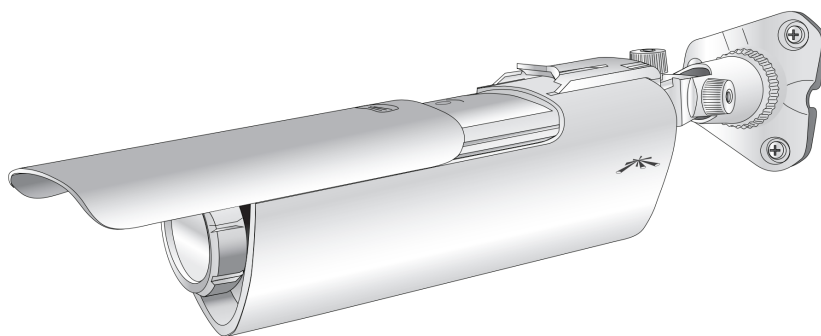
Data	0	1	2	3	4	5	6	7
Směr	S	SSV	SV	VSV	V	VJV	JV	JJV
Azimut	0	22.5	45	67.5	90	112.5	135	157.5
Data	8	9	10	11	12	13	14	15
Směr	J	JJZ	JZ	ZJZ	Z	ZSZ	SZ	SSZ
Azimut	180	202.5	225	247.5	270	292.5	315	337.5

Bohužel všechny zmíněné informace nejsou oficiální a pochází ze zdrojových kódů jiných programů, které se snaží tomuto konkrétnímu formátu

dat také porozumět. Dokonce stanice od stejného výrobce nemají formát dat jednotný a dokumentace není nikde k dispozici. Navíc tyto informace platí pro stanici La Crosse WS-3610 a je jen štěstí, že formát dat je stejný s meteostanicí v laboratoři EU 411. I tak jsou některé oblasti nejasné a stejně tomu je i u mezních hodnot, které může stanice odesílat. Jeden příklad za všechny: tato stanice může odeslat informaci obsahující rychlost větru 51 m/s. V takovém případě to však znamená, že je hodnota neplatná a je mimo rozsah (nebo není připojen anemometr). Proto je nutné znát alespoň základní rozsahy a hodnoty mimo rozsahy prohlásit za neplatné a vyřadit je. Ani tyto rozsahy bohužel nejsou k dispozici pro všechny hodnoty, které stanice dokáže změřit.

Vzhledem ke všem těmto překážkám je tato stanice spíše nevhodná pro napojení do libovolného informačního systému. Je důležité odesílaným datům bez pochyb rozumět a to se zde říci nedá. I přes všechny tyto komplikace je však meteostanice do systému zapojena a je možné s daty pracovat právě díky souboru `history.dat`. Detailněji je způsob přenosu informace popsán v třetí části této práce zabývající se architekturou tzv. konkretizačního uzlu.

2.2 IP kamera



Obrázek 2.2: IP kamera airCam od společnosti Ubiquity Networks
zdroj: Ubiquity Networks airCam User Guide

IP kamera, která byla k dispozici v této práci je *airCam* [2] od americké společnosti Ubiquiti Networks. Jedná se o dříve běžně dostupnou kameru pro vnitřní i vnější použití. Tato IP kamera je napájena prostřednictvím PoE (24 V, 0.5 A). Tento ethernetový kabel slouží zároveň pro komunikaci s IP kamerou. Na těchto kamerách je běžně spuštěný Linuxový server a tak je tomu i zde. Ten slouží zejména pro streamování (spojité odesílání videozáznamu).

Kamera však nabízí i webový server s jednoduchou aplikací pro ovládání a nastavování parametrů záznamu a v neposlední řadě obsahuje i SSH server, díky čemuž lze zjistit o kameře podrobnější informace (zjednodušený výpis):

```
$ ssh ubnt@192.168.0.123
ubnt@192.168.0.123's password:

BusyBox v1.18.4 (2013-06-09 00:59:08 EEST) built-in shell

AirCam.v1.2# cat /proc/version
Linux version 2.6.28 (buildd@builder) (gcc version 4.5.2
↪ (Linaro GCC 4.5-2011.02-0) ) #1 PREEMPT Sun Jun 9 01:03:42
↪ EEST 2013
AirCam.v1.2# cat /proc/cpuinfo
Processor       : FA626TE rev 1 (v5l)
BogoMIPS        : 532.48
Features        : swp half thumb
CPU implementer : 0x66
CPU architecture: 5TE
CPU variant     : 0x0
CPU part        : 0x626
CPU revision    : 1
Hardware        : Faraday GM8126
AirCam.v1.2# netstat -lnp
Active Internet connections (only servers)
Proto Local Address Foreign Address State PID/Program name
tcp 0.0.0.0:554 0.0.0.0:* LISTEN 6022/ubnt-streamer
tcp 0.0.0.0:80 0.0.0.0:* LISTEN 6016/lighttpd
tcp 0.0.0.0:22 0.0.0.0:* LISTEN 6020/sshd
tcp 0.0.0.0:443 0.0.0.0:* LISTEN 6016/lighttpd
tcp :::22 :::* LISTEN 6020/sshd
udp 0.0.0.0:10001 0.0.0.0:* 6014/infctld
```

Z předchozího výpisu (zejména pak z poslední `netstat` části) je vidět, že na IP kameře je otevřeno pět portů. Port 10001 je speciální port pro ostatní zařízení společnosti Ubiquiti Networks. Slouží k odhalování známých zařízení v síti. Dále je zde klasický port 22 pro SSH a 80 resp. 443 pro HTTP resp. HTTPS komunikaci. Poslední port je 554, který slouží právě ke streamování videa.

2.2.1 Streamování videa

Streamování u této kamery probíhá prostřednictvím protokolu RTSP (Real Time Streaming Protocol). Ve výchozím nastavení jsou k dispozici čtyři adresy rozlišené podle požadované kvality obrazu:

```
rtsp://192.168.0.123:554/live/ch00_0 - Plné rozlišení
rtsp://192.168.0.123:554/live/ch01_0 - Poloviční rozlišení
rtsp://192.168.0.123:554/live/ch02_0 - Čtvrtinové rozlišení
rtsp://192.168.0.123:554/live/ch03_0 - Malý náhled
```

Přímo v IP kameře v souboru `/etc/aircam-playlist.json` lze zjistit, jaké mají tyto streamy nastavení. Toto nastavení je pro každý stream téměř stejné, jen se liší rozměry výsledného videa a přenosová rychlost (výchozí hodnoty):

```
live/ch00_0: 1280x720 @ 25 FPS (max bitrate 4096)
live/ch01_0: 640x368 @ 25 FPS (max bitrate 1024)
live/ch02_0: 320x176 @ 25 FPS (max bitrate 256)
live/ch03_0: 160x96 @ 25 FPS (max bitrate 64)
```

Nejdůležitější jsou adresy jednotlivých streamů pro pozdější zpracování. Jelikož se jedná o RTSP protokol, tak nelze jednoduše spustit tento stream ve webovém prohlížeči, protože tento protokol není podporován. Bude tedy nutné vytvořit mezipřehrávač, který bude přijímat data ze streamu IP kamery a bude je transformovat na něco srozumitelného pro prohlížeče. Srozumitelným formátem je například HLS, čehož lze snadno dosáhnout pomocí programu FFmpeg [3]:

```
$ ffmpeg -i rtsp://user:pass@192.168.0.123:554/live/ch01_0 \
-hls_flags delete_segments+append_list \
-use_localtime 1 \
stream.m3u8
```

Tento příkaz na Linuxovém serveru začne přijímat stream z dané adresy a průběžně jej bude ukládat ve formátu připraveném pro HLS. Konkrétně se vždy vytvoří soubor `stream.m3u8`, který obsahuje seznam datových souborů, které obsahují jednotlivé fragmenty videa:

```
#EXTM3U
#EXT-X-VERSION:3
#EXT-X-TARGETDURATION:17
```

```
#EXT-X-MEDIA-SEQUENCE:351
#EXTINF:16.666667,
stream-1488616019.ts
#EXTINF:16.666667,
stream-1488616036.ts
#EXTINF:16.666667,
stream-1488616053.ts
#EXTINF:16.666667,
stream-1488616069.ts
#EXTINF:5.133333,
stream-1488616086.ts
#EXT-X-ENDLIST
```

FFmpeg se zde (kromě převodu RTSP na sekvence videa) chová jako krátká dočasná paměť. Vstupní stream rozděljuje do menších datových souborů a po nějakém čase je po sobě zase maže. Videopřehrávači v prohlížeči potom stačí pouze poslat umístění souboru `stream.m3u8`. Zodpovědností tohoto přehrávače je jednou za čas stahovat aktualizovaný seznam fragmentů videa a zároveň tyto fragmenty stahovat, správně seřadit a spouštět. Výsledný efekt je nepřetržité a nikdy nekončící video. Následuje ukázka minimálního HTML kódu, který je potřeba pro přehrání takového streamu v prohlížeči s využitím knihovny Clappr [4]:

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <script
    type="text/javascript"
    src="https://cdn.jsdelivr.net/clappr/latest/clappr.min.js">
  </script>
</head>
<body>
  <div id="player1"></div>
  <script>
    var player = new Clappr.Player({
      source: "stream.m3u8",
      parentId: "#player1",
    });
  </script>
</body>
</html>
```

Tato knihovna interně využívá knihovnu HLS.js [5], která zajišťuje kompatibilitu se všemi prohlížeči, které podporují MSE (HTML 5). To jsou v dnešní době tyto prohlížeče (a další):

- Chrome 34+
- Firefox 42+
- Internet Explorer 11+
- Edge 10+

Na závěr je důležité zmínit se o vznikající prodlevě při zpracování videa. Z toho důvodu, že FFmpeg musí nejdříve vstupní stream zpracovat a vytvořit krátkou frontu útržků videa, tak prodleva mezi realitou a pozorováním na monitoru je asi jedna minuta. Toto je však běžný jev i u společností, které se zabývají real-time přenosem videa, takže to nelze považovat za nevýhodu, ale spíše za nutné zlo. Je zkrátka potřeba vytvořit alespoň jeden fragment, aby bylo možné něco na straně prohlížeče přehrát. Prakticky je však potřeba vytvořit alespoň dva fragmenty videa, protože je třeba začít přehrávat první a zároveň na pozadí stahovat druhý pro plynulé navázání obrazu.

2.2.2 Způsob připojení kamery do veřejné sítě

Po připojení kamery do sítě běžně získává kamera svojí dočasnou IP adresu z DHCP serveru. V případě domácnosti je to zpravidla router. Stejným způsobem získávají IP adresu ve vnitřní síti také počítače a jiná zařízení. Pro pohodlnější obsluhu je vhodné nastavit této kameře statickou IP adresu, takže se nebude po čase měnit. Takto byla v předchozích příkladech kameře nastavena IP adresa 192.168.0.123. Problém však nastává u kamer, které musí být viditelné z veřejné sítě, což je většina kamer pro tuto aplikaci. V takovém případě je nutné mít k dispozici veřejnou IP adresu a na vstupním routeru nastavit NAT tak, aby překládal např. určené porty na statické adresy IP kamer ve vnitřní síti. To znamená, že každá kamera bude mít jinou privátní IP adresu s porty 554 (RTSP), ale NAT je bude vnějšímu světu ukazovat na jedné veřejné IP adrese, ale pokaždé s jiným portem. Do aplikace pro zpracování streamu se pak bude zadávat právě tato veřejná adresa.

3

Návrh aplikace

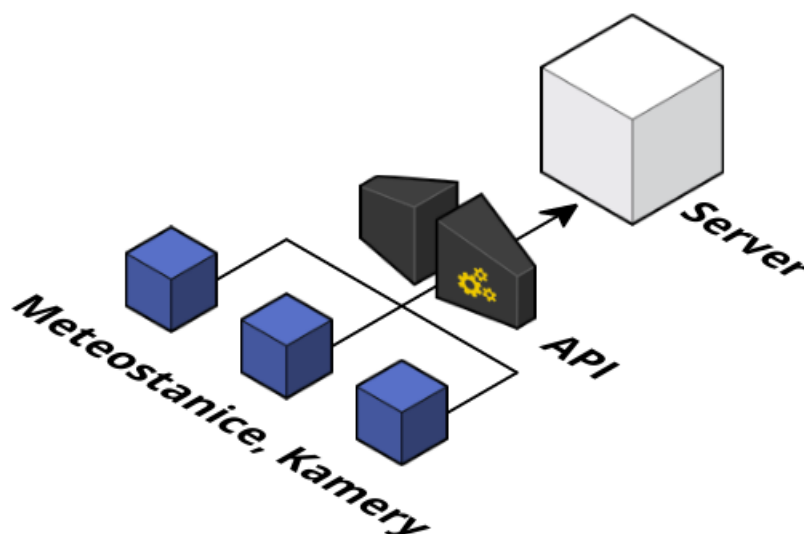
Před vývojem aplikace je klíčové dobře promyslet veškeré aspekty, které následně ovlivňují návrh řešení. V tomto konkrétním případě jde o tři otázky. Jak bude server data přijímat? Kam je bude ukládat? Jak se budou data zobrazovat uživateli? Vše ostatní je svým způsobem „pouze“ kód, který tento návrh umožňuje. Samotný kód je ovšem neméně důležitý. Špatně naprogramovaná aplikace může znamenat narůstající vytváření technického dluhu [13], který povede k nepřehledné a špatně udržitelné aplikaci. Následující sekce postupně poodhalují návrhy jednotlivých částí aplikace a myšlenky, které k tomuto návrhu vedly.

3.1 Přenos a ukládání dat z meteostanic

Ať už se jedná o webovou kameru nebo o meteorologickou stanici, aplikace musí nějak data získat a uložit je pro pozdější zpracování. Existují v zásadě dva přístupy, jak data získávat. Aplikace může data od koncových zařízení sbírat, nebo je může přijímat. Jedná se o zcela protichůdné procesy. Zatímco při sbírání musí aplikace vynaložit nemalé úsilí pro dotazování se na nová data, tak při naslouchání pouze čeká až nová data přijdou a ty následně uloží. Proto se tento druhý přístup používá v praxi mnohem častěji a stejně je tomu i v této práci.

Aplikace tedy funguje stejně jako je znázorněno na obrázku 3.1 - jako posluchač. Koncová zařízení odesílají na server požadavky a server je přijímá a ukládá pro pozdější zpracování. Každé zařízení však komunikuje svým originálním způsobem. Neexistuje jednotný formát dat, jsou dány pouze komunikační protokoly.

Aby se předešlo tomu, že server bude muset vědět jak funguje každý zdroj dat, který existuje, tak je vhodnější vytvořit jednotné veřejné API. Toto API bude dostupné pro každého, kdo chce se serverem komunikovat a důležité je, že není závislé na typu zařízení. Z toho plyne, že se každé zařízení musí podřídit a komunikovat předem daným způsobem. Obrovskou výhodou

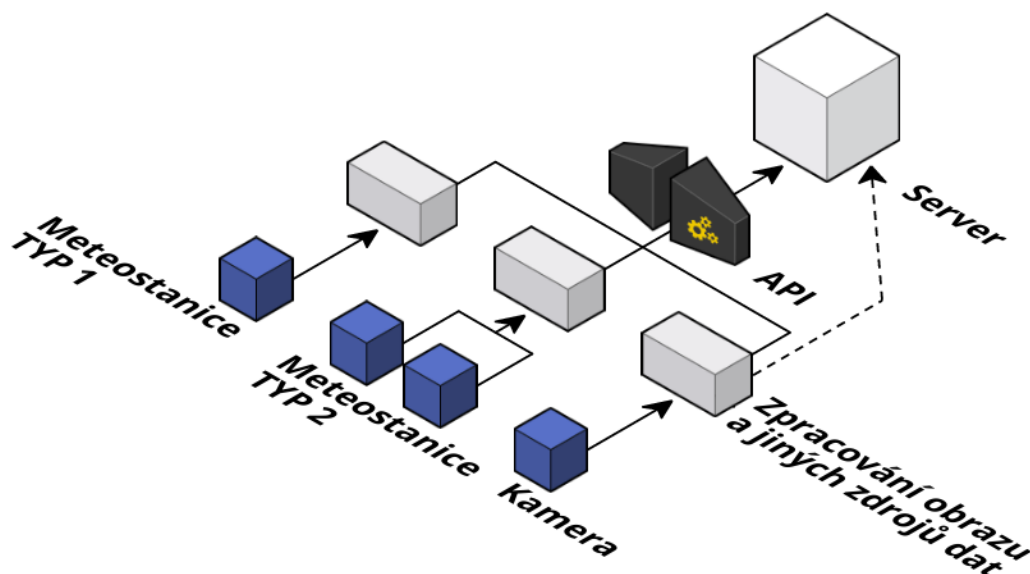


Obrázek 3.1: Základní návrh aplikace
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

tohoto přístupu je úleva zodpovědnosti serveru. Nemusí totiž vědět odkud data pocházejí, protože jsou pořád stejná. Místo toho se může soustředit na vykonávání své hlavní činnosti - zpracovávání a ukládání jednotného formátu dat. V celém systému se tak stává zdrojem pravdy.

Zůstává však otázka, jak z meteostanice tato data odeslat na API? Některé meteostanice odesílají UDP datagramy se svým vlastním formátem dat, jiné zase fungují jen přes sériovou linku (a tedy po síti neodesílají nic) a třeba takové kamery streamují obraz přes RTSP protokol. Ani jeden zmíněný způsob není s API kompatibilní. Proto je nutné představit celému API vrstvu, která kompatibilitu zajistí. Ta může být součástí infrastruktury, nesmí však být součástí serveru. Tuto myšlenku zachycuje lépe obrázek 3.2. Veškerá data z meteostanice, která neumí komunikovat s API přímo, musí nejdříve projít programem, který zajistí kompatibilitu s API. Tento program může využívat více meteostanic, pokud jsou stejného typu. Speciální případ potom tvoří kamera, kde je nutné nejdříve zpracovat stream dat a serveru předat pouze adresu tohoto nově vytvořeného a upraveného streamu. Server potom tento stream nabízí k dispozici pro prohlížeč, ale sám se nestará o jeho zpracování (viz popis streamování videa z první kapitoly).

Server následně ukládá data do databáze, kterou v tomto případě zastupuje PostgreSQL [6]. Volba databáze není ničím ovlivněna. Bylo možné vybrat téměř libovolnou relační databázi (například MySQL). V této práci se



Obrázek 3.2: Základní návrh aplikace s mezivrstvou pro API
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

však hojně využívá UUID identifikátorů a PostgreSQL je na rozdíl od jiných databází přímo podporuje. Toto je pouze výhoda zvolené databáze, nikoliv důvod zvolení databáze.

3.2 Architektura serverové části aplikace

U jakékoliv webové aplikace se vždy řeší stejný problém. Klient (webový prohlížeč, CLI) pošle požadavek na server (data z meteostanice), server předá tento požadavek aplikaci, ta na něj náležitě zareaguje a odešle klientovi zpět odpověď. Tento cyklus se neustále opakuje a napsat se dá téměř jakkoliv. Aby měl však program nějakou hodnotu, je nutné věnovat se dostatečně dlouho jeho návrhu. Nejde tedy pouze o napsání několika řádek kódu. Jde o hledání správného řešení.

Podívejme se detailněji na celou cestu od webového prohlížeče přes server zpět k prohlížeči. Otevřením stránky <http://example.com/> webový prohlížeč odesílá GET požadavek.

```
GET / HTTP/1.1
Host: example.com
```

```
Connection: keep-alive
Cache-Control: no-cache
Upgrade-Insecure-Requests: 1
User-Agent: Chrome/56.0.2924.87
Accept: text/html,application/xhtml+xml,application/xml;q=0.9
Accept-Encoding: gzip, deflate, sdch
Accept-Language: en-US,en;q=0.8
```

Server tento požadavek přijímá na portu číslo 80 a předává jej běžící službě, která na základě tohoto požadavku spouští aplikaci (v případě této práce PHP 7.1). Stejná cesta funguje i pro odpověď, pouze opačným směrem. Vše ostatní se odehrává v rámci aplikace. Bude proto zajímavější podívat se, co se děje právě zde (popisováno na vytvořené aplikaci v této DP práci).

Webový server je nastaven tak, aby vždy posílal požadavek na soubor `www/index.php` v adresářové struktuře aplikace. Zde se vytvoří DIC kontejner a aplikace se spustí:

```
<?php declare(strict_types = 1);

/** @var Nette\DI\Container $container */
$container = require __DIR__ . '/../bootstrap.php';
$container->getByType(Nette\Application\Application::class)
    ->run();
```

Toto se odehrává při každém požadavku. Vnitřně se aplikace podívá na URL adresu a podle nastavení dokáže přiřadit konkrétní GET požadavek konkrétnímu kontroleru. Jakmile aplikace dojde až do kontroleru, je čas vykonat nějakou operaci. V tomto jednoduchém případě by se jednalo pouze o vykreslení šablony a odeslání zpět. Představme si ale případ, kde je potřeba uložit něco do databáze.

Nebyl by žádný problém začít vykonávat tuto operaci přímo v kontroleru. Je to ten nejjednodušší přístup, ale má svá zrádná úskalí. Úkolem kontroleru je delegovat práci někam dál. Na nějaké místo v aplikaci, které je na tento úkol specializované a provádí pouze tuto jednu věc. To je velmi důležitý princip (SRP). Pokud dokážeme vytvářet kousky aplikace tak, že jsou samostatné a na jednu věc specializované, bude jednoduché je udržovat a automaticky testovat. Navíc to přispívá přehlednosti a kontrole nad programem.

V této práci je kromě SRP použit ještě koncept kontextů. Aplikace je rozdělena na menší logické celky (kontexty), které fungují jako samostatné zapouzdřené jednotky. Pokud potřebuje nějaký kontroler vykonat např. vložení do databáze jako v předchozím příkladu, tak pouze na programovou sběrnici

odešle příkaz, který se automaticky odešle do správného kontextu a ten jej zpracuje.

```
$commandBus->dispatch(new RegisterNewUser(  
    'username',  
    '53cr3tPa55w0rd'  
));
```

`RegisterNewUser` je pouze obyčejný value objekt [7], který zapouzdřuje přenášená data. Po spuštění se nevrací žádná hodnota a předpokládá se, že příkaz proběhne v pořádku. Zároveň je takto zajištěna databázová transakce i kontrola oprávnění (opět uvnitř kontextu).

3.2.1 Hexagonální architektura

Je dobrý nápad oddělovat jednotlivé kontexty uvnitř aplikace. Jak toho ale dosáhnout bez využití mikroslužeb jen na úrovni společného kódu? Odpovědí může být právě hexagonální architektura. Místo toho, aby mohl v kódu kdokoli dělat cokoli, tak stanovíme hranice pomocí příkazů, které se spouští díky sběrnici. Takto vypadá malý výřez adresářové struktury v aplikaci:

```
src/  
├── Authentication.....řízení oprávnění  
│   ├── Application.....I/O porty balíčku  
│   ├── DomainModel.....doménový model  
│   └── Infrastructure.....implementační detaily doménového modelu  
├── Devices.....zařízení (meteostanice, kamery, apod.)  
│   ├── Application  
│   ├── DomainModel  
│   └── Infrastructure
```

`Authentication` i `Devices` jsou kontexty, které se starají o uživatelské účty včetně oprávnění uživatelů resp. o zařízení (meteostanice, webkamery, apod.). Každý tento kontext obsahuje tři hlavní části: `Application`, `DomainModel` a `Infrastructure`.

Část `Application` obsahuje již dříve zmíněné příkazy (commandy) a jejich obsluhu. S kontextem je tedy možné komunikovat pouze prostřednictvím této aplikační vrstvy¹.

¹Ve skutečnosti je možné programově zasáhnout i přímo do kontextu. Tomu bohužel nejde nijak zabránit, protože by aplikace nešla testovat. Je tedy třeba dodržovat určitá pravidla. Toto je jedno z nich.

`DomainModel` potom obsahuje veškerou logiku, která **není závislá na implementačních detailech**. Tato poslední poznámka je pro dobrý návrh klíčová. Model jednoduše nesmí vědět o tom, že existuje nějaká databáze, do které se data ukládají. Nesmí vědět, že existuje nějaký konkrétní typ meteorologické stanice. Může ale vědět, že existuje pojem meteorologické stanice a s ním pracovat. Pokud dokážeme odlišit tento rozdíl, pak bude doménový model fungovat pro jakoukoliv meteostanici a nemusíme řešit implementační detaily. Dosáhneme toho využitím rozhraní v PHP [8].

A konečně se dostáváme k **Infrastructure**. Právě zde jsou umístěny jednotlivé implementační detaily. Detaily typu jakou databázi aplikace používá, jak se do ní data ukládají, popř. jak se aplikace napojuje na CLI nebo prezentační část.

Hexagonální architektura se ve svém původním znění používá spíše pro vytváření kontextů u jednotlivých mikroslužeb pro složitější návrhy. To však neznamena, že nelze stejné postupy aplikovat v rámci jednoho kódu a dodržovat stejná pravidla. Jedna z velkých předností hexagonálního návrhu je možnost odložit rozhodnutí. Pokud totiž vytváříme aplikaci, tak by mělo být jedno do jaké databáze se následně data uloží. Můžeme bez větších problémů navrhnout celý doménový model a až na samotném konci se rozhodnout a zvolit tu správnou databázi. Databáze je totiž jen implementační detail [9]².

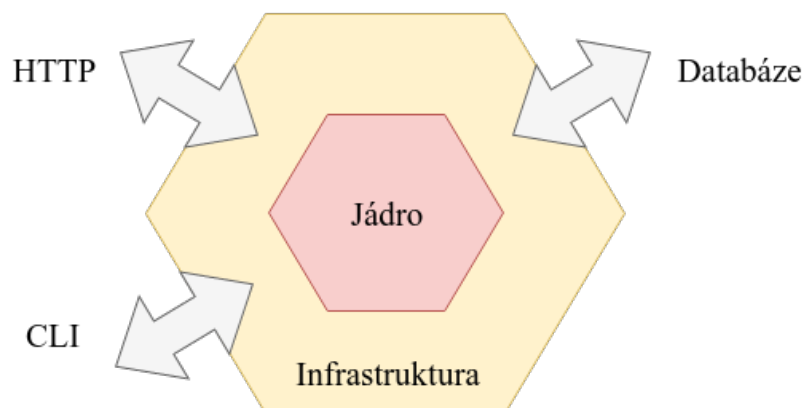
3.2.2 Konkrétní příklad ovládání kontextu

Hexagonální architekturu si lze představit jako cibuli s vrstvou slupek. Pokud se chceme dostat k samotnému jádru, musíme projít přes několik vrstev (slupek). Samotné jádro (doménový model) neví o implementačních detailech, ty jsou ve vyšší vrstvě obalující jádro. Tato vrstva tak tvoří prostředníka mezi konkrétní technologií a nezávislým jádrem programu, jako je znázorněno na obrázku 3.3. Z levé strany přicházejí požadavky z webového prohlížeče, popř. z konzole nebo API, a z pravé strany probíhá komunikace s databází.

Z toho je vidět, že každý tento vstup má vlastní komunikační protokol, a proto je znázorněn v jiné části šestiúhelníku. Úkolem infrastrukturní vrstvy je tyto požadavky přeložit a komunikovat s jádrem³. Prakticky to probíhá tak, že požadavek z webového prohlížeče přichází jako HTTP zpráva do kontroleru v aplikaci. Kontroler pouze vytvoří jednoduchý objekt reprezentující

²V praxi se ukazuje, že je rozumné mít na paměti s jakou databází se pracuje a využívat toho. Pro většinu aplikací však tato znalost nepřináší větší výhody než odstínění se od této skutečnosti.

³V anglických textech se vstupům říká porty a překlady požadavků probíhají pomocí adaptérů. Např. požadavek z webového prohlížeče přichází na HTTP port, kde adapterem je kontroler.



Obrázek 3.3: Hexagonální architektura

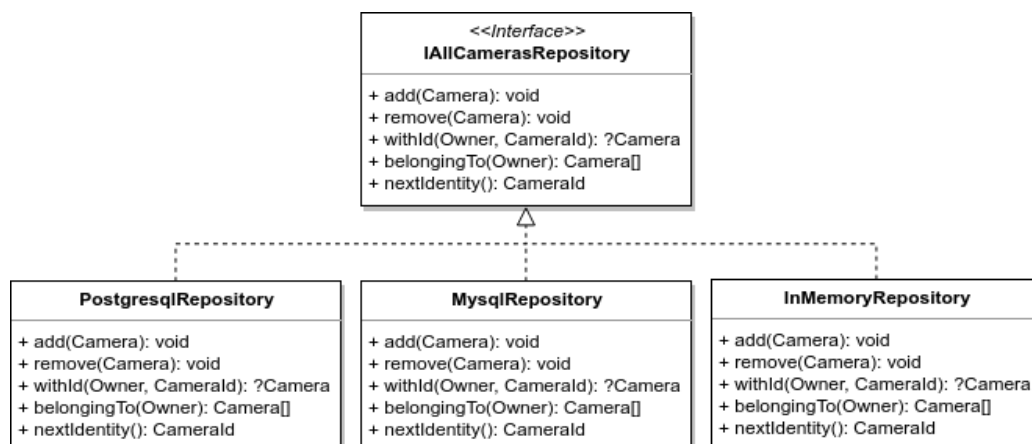
požadavek a odešle jej jako příkaz jádru. V tomto případě vytvoření nové meteostanice:

```
$commandBus->dispatch(new CreateWeatherStation(
    'Weather Station Name',
    UserId::createFromString($userId) // UUID
));
```

Takto odeslaný příkaz jádru je již zbavený informace o původní technologii. Díky tomu je možné použít stejný princip pro přicházející komunikaci z jakéhokoli zdroje. Jde pouze o vytvoření objektu `CreateWeatherStation` a odeslání na sběrnici příkazů. Samotné vytváření objektu je právě záležitostí infrastrukturní vrstvy. Je důležité uvědomit si, že komunikace probíhá pouze od vyšších vrstev k nižším. Tak je zajištěno, že jádro neví o infrastruktuře. Problém ale nastává při komunikaci s databází. Pokud mají požadavky putovat pouze směrem do středu, jak může jádro něco ukládat do databáze, pokud o databázi nesmí vědět?

První naivní postup by byl v jádru zavolat nějaký objekt, který by rovnou uložil data do databáze. Říkejme mu repozitář [10]. Tento repozitář by však musel vědět, že se používá PostgreSQL a tedy jádro volá část programu obsahující implementační detaily. To je nežádoucí. Místo toho je mnohem vhodnější vyžadovat rozhraní [8], které neobsahuje informace o databázi, ale pouze dává k dispozici jakýsi předpis toho, co bude konkrétní implementace umět. Tato myšlenka je zachycena na obrázku 3.4 pomocí UML diagramu.

K dispozici je jedno rozhraní `IAllCamerasRepository`, které je využíváno jádrem. Vedle rozhraní existují i konkrétní implementace, které již vědí o technologii. Jádro vyžaduje rozhraní, ale DIC se postará o to, aby bylo toto



Obrázek 3.4: UML diagram konkrétních implementací repozitáře

rozhraní nahrazeno konkrétní implementací. Efekt je pak takový, že v jádru je možné programovat technologicky nezávisle, ale program bude fungovat např. díky PostgreSQL. Tomuto postupu se říká Inversion of Control [11].

Využívání rozhraní má ještě jednu obrovskou výhodu. Při automatickém testování může být nežádoucí pracovat přímo s databází z důvodu rychlosti. Testy musí být rychlé. Proto je možné místo PostgreSQL implementace použít implementaci, která využívá pouze dočasnou paměť. Testy se tak násobně zrychlí. Takové testování je naprosto v pořádku pokud nechceme testovat funkčnost repozitáře, ale nějakou třídu, která jej využívá.

3.2.3 Kompletní adresářová struktura

Celá hlavní aplikace je v této práci postavena s využitím Nette Frameworku [12]. Celá adresářová struktura je znázorněna v následující části. Je zde vidět dříve popisované jádro aplikace ve složce `src`.

```

/
├── bin
│   └── console ..... spouštěč CLI
├── config
│   ├── config.local.neon ..... lokální konfigurační soubor
│   └── config.neon ..... obecný konfigurační soubor
├── extensions ..... implementace služeb třetích stran
├── migrations ..... databázové migrace
│   └── basic-data ..... základní data

```

└─ dummy-data	testovací data
└─ structures	základní struktury
src	
└─ Authentication	
└─ Devices	
└─ ...	
tests	testy aplikace
var	logy a cache
vendor	služby třetích stran
www	veřejná složka
└─ index.php	vstupní bod aplikace
bootstrap.php	vytváří DIC kontejner
composer.json/.lock	definice závislostí třetích stran

Vstupním bodem je soubor `www/index.php`. Tímto souborem projde veškerá HTTP komunikace, která je následně zpracovávána v jednom z kontextů ve složce `src`. Komunikace s aplikací prostřednictvím CLI zase probíhá přes `bin/console`:

```
$ bin/console dbal:run-sql "SELECT * FROM user_accounts"
array(1) {
  [0]=>
  array(3) {
    ["id"]=>
    string(36) "00000000-0000-0000-0000-000000000001"
    ["password_hash"]=>
    string(60) "$2y$10$8nJLFVEGIFRnSyb5vjukQ/tJ/V4Yupa30..."
    ["username"]=>
    string(4) "test"
  }
}
```

Soubor `index.php` dělá pouze to, že vytváří DI kontejner (DIC), který se stará o správné složení objektového grafu a obstarávání závislostí napříč aplikací. Následně z tohoto kontejneru získá aplikační službu, která se stará o běh aplikace. Význam ostatních částí je zřejmý z popisků, jen testy a migrace jsou rozebrány ve zvláštních částech, které následují.

3.2.4 Testování aplikace a CI provoz

Napsání kódu je jen polovina problému. Každá správná aplikace musí mít automatické testy. Testy mají jediný význam. Kontrolují, jestli dříve na-

psaný kód funguje tak, jak bylo původně zamýšleno. Mnohdy se význam testů překrucuje a říká se, že díky testům neobsahuje aplikace chyby. To není správně. Díky testům neobsahuje pouze chyby, o kterých již víme - jinak chyby stále obsahuje.

U takových testů je důležité jejich neustálé spouštění. Každá nová řádka, dokonce i nový znak v programu může způsobit nefunkčnost nějaké části aplikace a není v lidských silách vždy kontrolovat vše. Spuštění testů se v této aplikaci provádí pomocí příkazu `tests/run`:

```
$ tests/run
```

```
TESTBENCH edition
```

```
-----
|_  _/_/ _/(_/_/_/_/ _/ _/)_|_  _/
|_| \___/_/___) |_| \___ |_| \___ v2.0.x
```

```
PHP 7.1.1 (cli) | php | 8 threads | /tests/bootstrap.php
```

```
[OK] GraphQL/DomainModel/DateTimeTypeTest.phpt
```

```
[OK] DI/Nette/ExtensionEnumTest.phpt
```

```
[OK] DomainModel/Humidity/RelativeHumidityTest.phpt
```

```
[FAIL] DomainModel/Pressure/PressureTest.phpt
```

```
Failed: MissingServiceException was expected but got ...
```

```
[OK] Infrastructure/DomainModel/CommandBusTest.phpt
```

```
...
```

```
FAILURES! (149 tests, 1 failure, 4.7 seconds)
```

Automatické testy jsou vlastně pouze obyčejné funkce, které mají za úkol spustit omezenou podčást aplikace a zkontrolovat, jestli se na základě daných vstupů vrací očekávaná odpověď. Testy proto musí být deterministické a mělo by být možné je spouštět ve více vláknech procesoru (mimo speciální případy) pro zvýšení rychlosti. Níže je pro ukázkou zjednodušený test, který ověřuje, že objekt reprezentující kameru lze inicializovat a je možné vstupní hodnoty zase získat.

```
public function testCameraEntity() {
    $uuid = Uuid::fromString('58d200ad-6376-4c01-9b6d');

    $cameraEntity = Camera::create(
        CameraId::create($uuid),
        new Owner(new User(UserId::create(), 'User Name')),
    );
}
```

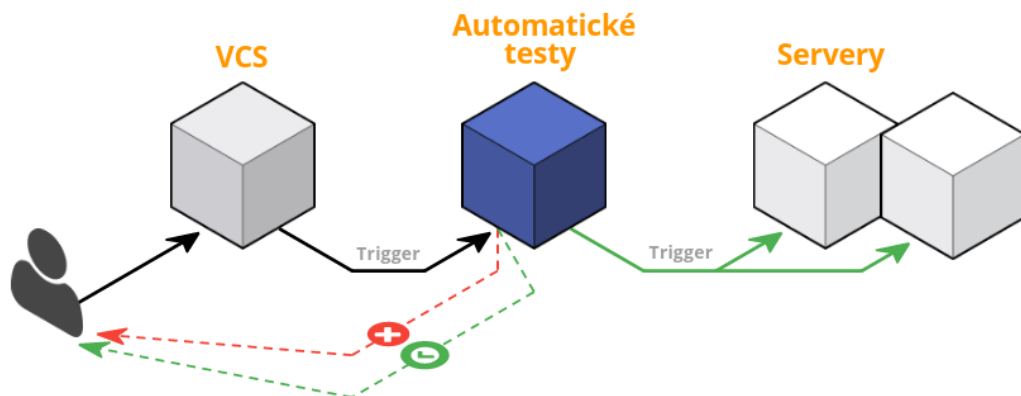
```

    'Camera Name',
    Stream::create('rtsp://a', Uuid::uuid4(), 'hls'),
    new \DateTimeImmutable
);

Assert::type(CameraId::class, $cameraEntity->id());
Assert::same(
    '58d200ad-6376-4c01-9b6d',
    $cameraEntity->id()->toString()
);
Assert::same('Camera Name', $cameraEntity->cameraName());
}

```

V první polovině testu se vytvoří objekt `Camera` a v druhé polovině se pomocí `Assert` funkcí ověřuje platnost hodnot. Pokud například vrácené ID z objektu nesouhlasí s očekávaným ID, test selže a je nutné hledat a opravit chybu.



Obrázek 3.5: Continuous Integration
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

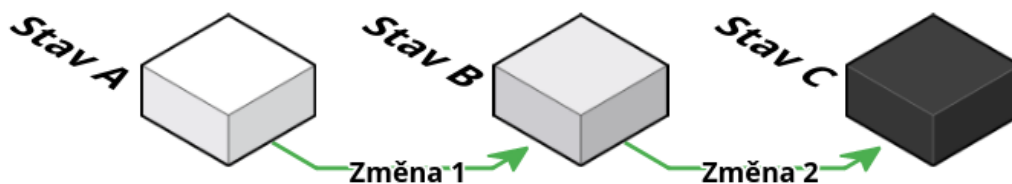
Z existujících testů se však dá těžit mnohem více. Automatické testy se hodí i při spolupráci více lidí, kdy je možné veškeré zaverzované změny na vzdáleném serveru okamžitě automaticky kontrolovat a v případě, že aplikace funguje správně, můžeme spouštět například automatické nasazení nové verze aplikace na ostrý server (jako je to na obrázku 3.5). Zde platí, že čím více je kód pokryt testy, tím je menší šance na vznik nějaké chyby a je možné automatickým testům více věřit. Nemá smysl snažit se o automa-

tické nasazování nové verze na ostrý server, pokud testy nejsou, nebo je jich nedostatečně málo.

Podle obrázku 3.5 programátor odesílá své kódy např. pomocí Gitu [14] do vzdáleného repozitáře. Po odeslání se automaticky spouštějí testy a programátor tak získává zpětnou vazbu o tom, jestli je vše stále funční či nikoliv. Celý proces může být složitější, ale základní myšlenka je pořád stejná. Důležité je mít pouze dostatečné pokrytí aplikace testy tak, aby se dalo testům důvěřovat, a pak je možné spoustu kroků zautomatizovat.

3.2.5 Návrh databáze a databázové migrace

Stejně tak jako se běžně verzují kódy aplikace, je třeba nějakým způsobem verzovat databázi. To je však (na rozdíl od kódu) velký problém. Verzování kódu totiž funguje stejně jako na obrázku 3.6. Pokud se kód nachází ve stavu A, tak naše změny představují rozdíl od této verze a po aplikování změn se dostáváme do dalšího stavu B. Důležité je, že v každém kroku je kód jasně daný a tedy aplikujeme změny na známý stav.



Obrázek 3.6: Verzování kódu v Gitu
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

Změny databáze však fungují spíše podle obrázku 3.7. Databáze se neustále mění. Zejména pak její data, kvůli kterým je tento problém zcela odlišný od verzování kódu. Po aplikování změn totiž databáze nezůstane v původním stavu. Vnitřní stav databáze se přirozeně mění a nový stav není nikdy přesně známý. Mění se z toho důvodu, že uživatelé databázi používají - přidávají, mažou a aktualizují data. Takže zatímco v kódu můžeme bez obav změny vrátit, u databáze to není možné a veškeré změny probíhají jen vpřed.

Z toho důvodu je chytrou strategií vždy dělat zpětně kompatibilní změny. To znamená, že nově nasazená verze databáze vždy funguje i se starou verzí aplikace (z hlediska aplikace je to dopředná kompatibilita). V této práci jsou migrace zajišťovány pomocí knihovny Nextras Migrations [15]. Tato knihovna zajišťuje kontrolu nad spouštěním databázových migrací ve



Obrázek 3.7: Problém verzování databáze
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

správném pořadí. Je pak na programátorovi, aby nedělal destruktivní změny. Databázové migrace jsou v adresářové struktuře uloženy takto:

```

migrations/
├── basic-data
│   ├── 2017
│   │   └── 02
│   │       └── 2017-02-03-104919-ws-series-ws3600.sql
├── dummy-data
│   ├── 2017
│   │   ├── 01
│   │   │   └── 2017-01-31-091915-test-user.sql
│   │   └── 02
│   │       ├── 2017-02-14-104504-weather-stations.sql
│   │       └── 2017-02-23-161812-cameras.sql
└── structures
    ├── 2017
    │   ├── 01
    │   │   └── 2017-01-31-090810-initial.sql
    │   └── 02
    │       ├── 2017-02-03-104848-ws-series.sql
    │       ├── 2017-02-14-085713-ws-creation-date.sql
    │       ├── 2017-02-15-095854-physical-quantities.sql
    │       ├── 2017-02-21-155618-ws-record-creation-date.sql
    │       └── 2017-02-23-095726-cameras.sql
  
```

Jak je vidět, tak jsou změnové skripty označeny datem. Zároveň se v databázi uchovává informace o předchozích migracích. Při spuštění migrací přes CLI se tedy začnou provádět pouze zatím neaplikované změny a to v pořadí podle času. Aplikace se tak dostane z neznámého stavu do jiného neznámého stavu jasně definovanou cestou. Díky tomu, že jsou migrace součástí kódu,

je možné je spouštět také v průběhu automatického testování a nasazování nové verze aplikace na server.

3.2.6 Zabezpečení API pro obousměrnou komunikaci

Aby mohla klientská aplikace komunikovat se serverem, musí komunikovat prostřednictvím API. A je naprosto nezbytné, aby tato klientská aplikace posílala autorizované požadavky. Jednak je nutné vědět, kdo požadavky posílá, a potom data, která server v API nabízí, nejsou veřejně přístupná. V této práci se využívá JW token (JWT) [16]. Uživatel se nejdříve přes HTTPS přihlásí a po úspěšném ověření přihlašovacích údajů server vrátí JWT, který se uloží do „local storage“ v prohlížeči. JWT vypadá takto:

eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9.eyJpYXQiOjE0ODg3MzE5MDcsImV4cCI6MTQ0Dgx0DMwNywidXVpZCI6IjAwMDAwMDAwLTAwMDAtMDAwMC0wMDAwLTAwMDAwMDAwMDAwMSJ9.odmV31XcAzVaKZpepNoSfnmavq3FDnZDF12zhfR6J54Yh8eFw2mj1Meq9nBUe5amtZRp45j9_xe8pIsJe7z7Jw

Jedná se o dlouhý řetězec, který obsahuje dvě tečky (barevně zvýrazněné). Tyto tečky dělí řetězec na další tři části. První část je zakódována pomocí base64 [17], takže pro odhalení obsahu je možné použít funkci `atob` z JavaScriptu:

```
atob('eyJ0eXAiOiJKV1QiLCJhbGciOiJIUzUxMiJ9');
```

Tato funkce vrátí JSON, který obsahuje informace o tokenu, zejména pak o použitém šifrovacím algoritmu:

```
{"typ": "JWT", "alg": "HS512"}
```

Podobně je tomu i u druhé části tokenu. Ta však navíc vrací libovolné informace, které si aplikace potřebuje předat:

```
{
  "iat": 1488731907,
  "exp": 1488818307,
  "uuid": "00000000-0000-0000-0000-000000000001"
}
```

V tomto případě se jedná o UUID uživatele, čas vystavení tokenu `iat` a čas expirace token `exp` (trvanlivost). Zatím se tedy jedná o veřejně přístupné informace, které může kdokoli získat. V tokenu by neměla být žádná citlivá informace, protože uživatel (webový prohlížeč) pak tento token odesílá při

každém požadavku a server ověřuje jeho pravost. Není tak nutné neustále posílat a někde uchovávat heslo. Aby nedošlo k podvržení, tak token obsahuje ještě poslední část, která slouží jako ochrana proti napadení. K vygenerování poslední části je nutné znát nejen data, ale i privátní klíč (v předchozím příkladu je to slovo **secret**). Tímto privátním klíčem se token na serveru také ověřuje a pokud došlo ke změně informací v tokenu, tak jej nepůjde ověřit a považuje se za nevalidní.

Webová aplikace posílá na server (resp. na API) HTTP požadavky s hlavičkou **authorization**, která obsahuje JWT token. Tento způsob komunikace musí dodržet každý, kdo chce se serverem komunikovat. Server tak ví o jakého uživatele se jedná a jestli je oprávněn danou operaci provést. Uživatel musí mít například přístup k meteostanicím, ale jen k těm, které vlastní.

3.3 Architektura uživatelského rozhraní

Uživatelské rozhraní je napsáno s pomocí React knihovny od společnosti Facebook [18]. Tato knihovna umí velmi dobře pracovat s DOM v prohlížeči pomocí deklarativního přístupu. To znamená, že odpovědností programátora je nadefinovat, jak by měl určitý stav aplikace vypadat a odpovědností React knihovny je co nejlepším způsobem se do tohoto stavu dostat⁴.

3.3.1 Základy práce s Reactem

Celé uživatelské rozhraní se skládá z komponent. Komponenty získávají data a podle těchto dat automaticky mění stav DOMu. Taková jednoduchá komponenta v Reactu by mohla vypadat třeba takto:

```
let Component = (props) =>
  <div className="fullname">
    I am {props.firstname}{', '}
    <strong>{props.surname}</strong>
  </div>;

ReactDOM.render(
  <Component firstname="John" surname="Doe"/>,
  document.getElementById('root')
);
```

⁴Opakem je imperativní programování. Zde programátor říká, co se má vykonávat. U deklarativního programování naopak programátor říká, jak má vypadat koncový stav.

V první části kódu je vytvořena komponenta s názvem `Component` (**musí** začínat velkým písmenem), která obsahuje obyčejný `div` s vypsáním jména a příjmení. Tyto hodnoty se dostanou do komponenty prostřednictvím tzv. vlastností (properties - props). Vlastnosti vždy přicházejí do komponenty zvenku, což je vidět v druhé části kódu, kde se komponenta napojuje do DOMu. Zároveň jsou vlastnosti neměnné.

Tímto stylem je možné připravovat komponenty a pouze měnit jejich vlastnosti. React se postará o správné vykreslení. Předchozí spuštěný kód vykreslí v prohlížeči následující HTML:

```
<div class="fullname">I am John <strong>Doe</strong></div>
```

Možná to na první pohled nevypadá jako nic zajímavého. Dokonce se to může zdát jako příliš složité a nepraktické. Je třeba si však uvědomit, že taková komponenta lze použít na více místech a pouze změnou jejich vlastností ovládat výsledný stav. Navíc pracujeme jen s minimálním HTML, takže je menší šance na vytvoření chyby. Celé uživatelské rozhraní se potom skládá z drobných komponent (které se zase skládají z jiných komponent) a tak vznikne výsledná stránka. Komponenty je také velmi jednoduché testovat.

Ačkoliv by se z předchozích ukázek mohlo zdát, že se v komponentě pracuje přímo s HTML, tak to není pravda. Jedná se o JSX. Tomu, co je JSX, je věnována celá následující sekce. Pro teď stačí, že předchozí kód je nutné ještě přeložit do čistého JavaScriptu, například pomocí nástroje Babel [19]. Díky tomu je možné používat moderní JavaScript ve starších prohlížečích. Výsledný kód pak vypadá takto:

```
var Component = function Component(props) {
  return React.createElement(
    "div", { className: "fullname" },
    "I am ", props.firstname, ' ',
    React.createElement(
      "strong",
      null, // nemá žádné vlastnosti
      props.surname
    )
  );
};

ReactDOM.render(React.createElement(
  Component, { firstname: "John", surname: "Doe" }
), document.getElementById('root'));
```

To je sice mnohem nepřehlednější kód, ale je zde hezky vidět, co se děje na pozadí. Každý element podobý HTML se ve skutečnosti překládání na volání funkce `createElement`, jejíž vstupní parametry jsou: název komponenty (elementu), její vlastnosti a výčet dalších elementů. Již na takto malé ukázce je vidět způsob zanořování komponent. To je také důvod, proč původní ukázka obsahuje mezeru za křestním jménem. Mezery se totiž zachovávají pouze pokud jsou v řádce. Ostatní se zahazují a je třeba je pomocí JSX vynutit.

Existuje ještě jeden způsob komponent, které se vytváří pomocí tříd. Zde je ukázka stejné komponenty tímto novým způsobem:

```
let Component = class extends React.Component {
  render = () =>
    <div className="fullname">
      I am {this.props.firstname}{' '}
      <strong>{this.props.surname}</strong>
    </div>;
};
```

Způsob použití je stejný, výsledný kód je podobný, liší se pouze přístup k vlastnostem. Tyto komponenty mají však navíc ještě tzv. stavy (anglicky states). To jsou vlastnosti komponenty, které jsou zapouzdřeny pouze v jedné konkrétní komponentě a drží její vnitřní stav. Typicky se tak obsluhují třeba formuláře. Komponenta ví, co se s formulářem děje a jak se mění, ale nikdo mimo komponentu k této informaci nemá přístup.

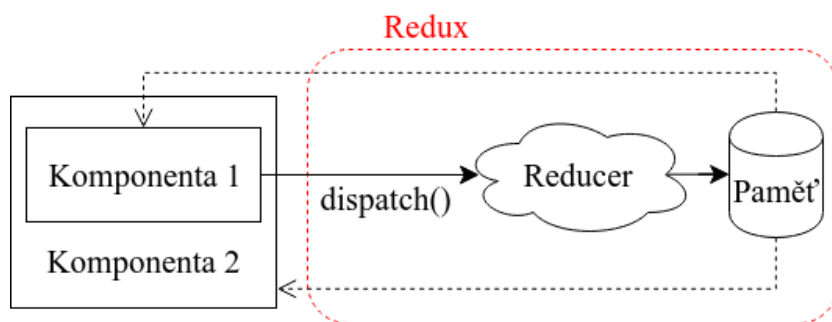
Z předchozích ukázek a vysvětlování by mělo být jasné, že data tečou skrz komponenty pouze směrem dolů od nadřazených komponent k listům pomyslného grafu komponent. Ačkoliv je toto správný přístup, občas je potřeba ze zanořených komponent předávat informaci nadřazeným komponentám. Typickým zástupcem knihoven, které toto řeší, je Redux [20]. V tomto případě již stav komponenty není uchován v komponentě, ale v globálním úložišti a veškeré změny stavu probíhají prostřednictvím tohoto úložiště. Zejména kvůli slovu *globální* zní tento přístup jako slevení ze všech dobrých postupů, Redux je však mezi React vývojáři velmi oblíbený a v této práci se hojně využívá.

3.3.2 Dočasné Redux úložiště

Díky tomu, že je React deklarativní, tak stačí někde udržovat stav aplikace a při jeho změně se potřebná část aplikace automaticky překreslí. Stav aplikace se běžně uchovává uvnitř každé komponenty. U složitějších aplikací je však tento přístup spíše nevhodný, protože komponenty mohou předávat

data pouze vnořeným komponentám a nikoliv nadřazeným. Redux [20] je knihovna, která se k uchovávání stavu aplikace staví zcela jinak. Místo toho, aby se držel stav v každé komponentě, tak si Redux pamatuje stav celé aplikace v jednom úložišti. Díky tomu je možné v průběhu životního cyklu aplikace postupně upravovat tento stav a React se postará o překreslení DOMu.

Udržovat velké množství informací v jednom úložišti může rychle přerůst programátorům přes hlavu. Redux proto obsahuje způsob, jak úložiště rozumným způsobem aktualizovat. Každá komponenta má možnost vytvořit a odeslat akci pomocí funkce `dispatch`. Redux se následně postará o to, aby tato akce došla do tzv. reduceru, pokud nějaký existuje. Ten má za úkol příchozí akci nějakým způsobem zpracovat a aktualizovat úložiště (paměť). Následně dojde k automatickému překreslení na všech místech, které tuto část paměti používají. Díky tomuto způsobu lze aktualizovat i nadřazené komponenty, jak je vidět na obrázku 3.8.



Obrázek 3.8: Aktualizace komponent prostřednictvím Reduxu

Nadřazené komponenty by šlo aktualizovat i bez centrálního úložiště, horší případ by nastal u komponent, které jsou ve stromu komponent na úplně jiném místě a nemají úzkou provázanost - přesto se nějak ovlivňují. Např. každá komponenta může vytvořit chybovou hlášku (také komponentu) napříč celou aplikací.

Akce jsou v Reduxu obyčejné JavaScript objekty, pokud je však potřeba v rámci akce spustit nějakou funkci (dotázat se API), je možné tak učinit. Pro akce, které nejsou jen objekty, je zapotřebí rozšiřujících knihoven jako je třeba Redux Thunk [21]. Reducer je potom úplně obyčejná funkce, která má k dispozici původní stav úložiště a akci a na základě těchto informací vytváří novou podobu úložiště. Reducer by nikdy neměl měnit vstupní argumenty, ale vždy vrátit novou instanci, neměl by volat jiné funkce (data přicházejí v akci) a měl by být deterministický (tzn. nepracovat s datumem, náhodnými čísly, apod.).

Obrovskou výhodou je možnost řetěžit jednotlivé akce případně rozkládat reducery. Spuštění akce nemusí znamenat jen jeden úkol, ale může spustit celou sekvenci akcí. Přidání nové meteostanice tak znamená zavolání API, ukázání průběhu zpracovávání, zobrazení zprávy o úspěšném založení stanice a přesměrování - to vše vyvoláno jedním příkazem typu „založ meteostanici s názvem XY“.

Oproti uchovávání stavu uvnitř komponent je zde ještě jedna výhoda. Díky tomu, že jsou všechna data na jednom místě, je možné je uchovávat v normalizované podobě a teoreticky tak ušetřit potřebné místo v paměti.

3.3.3 JSX

V roce 2009 byl poprvé zveřejněn projekt XHP jako open-source. Cílem bylo vytvořit způsob, jak eliminovat velké množství chyb a bezpečnostních problémů v PHP [22]. Takže zatímco původní kód v PHP pro vypsání krátkého HTML by vypadal takto:

```
echo "<i>Hello <b>$user_name</b>!</i>";
```

Tak v XHP by stejný zápis vypadal následovně:

```
echo <i>Hello <b>{$user_name}</b>!</i>;
```

Na počet znaků se jedná o stejné zápisy. Rozdíl je pouze v uvozovkách a ve způsobu vypsání proměnné. Je zde však velký rozdíl ve významu. V případě PHP se jedná pouze o skládání řetězce, v případě XHP se jedná o celou gramatiku, která je součástí jazyka. Díky tomu bude v případě XHP výstup vždy validní HTML a nebude obsahovat bezpečnostní chybu jako příklad s PHP. Ten je náchylný na XSS útok.

XHP se překládá na PHP třídy reprezentující danou značku. Stejná myšlenka byla převzata do JavaScriptu ve formě JSX. Opět se HTML značky zapisují v XML notaci a následně se překládají na jiný kód, který se stará o stavbu DOMu a zabezpečení:

```
let variable = 'class';
<div attribute={variable}>
  <Component/>
</div>
```

Z původního zápisu v JSX se tak stane obyčejný JavaScript:

```
var variable = 'class';
React.createElement(
```

```

    'div',
    { attribute: variable },
    React.createElement(Component, null)
  );

```

Aby se rozlišil obyčejný HTML tag a React komponenta, tak je nutné rozlišovat velikost počátečního písmena. Musí také existovat vždy jedna nadřazená komponenta (stačí ve formě HTML):

```

<div>
  <CapitalizedComponentName/>
  <justHtmlTag/>
</div>

```

Z výsledného přeloženého kódu bude jasné, proč tomu tak je:

```

React.createElement(
  "div",
  null,
  React.createElement(CapitalizedComponentName, null),
  React.createElement("justHtmlTag", null)
);

```

HTML tagy se překládají jako obyčejné řetězce, ale komponenty zůstávají jako proměnné. Takže aby předchozí kód fungoval, tak musí být někde dříve tato komponenta vytvořená, protože zde už je jen její použití, které v případě JSX pouze vypadá hezky.

JSX je velmi intuitivní a veškeré možnosti jsou velmi dobře popsány v dokumentaci [24]. Proto si myslím, že je bezpředmětné psát do této práce to samé a raději bych se ještě zdržel u jedné zvláštnosti, která má úzkou souvislost s React vykreslováním DOMu a zaslouží si vysvětlení. Představme si následující kód:

```

ReactDOM.render(
  <ul>
    {[1, 2, 3, 4, 5].map((item) =>
      <li>{item}</li>
    )}
  </ul>,
  document.getElementById('root')
);

```

Sám o sobě je tento kód naprosto v pořádku. Vykreslí se bodový seznam pěti čísel. React si však bude stěžovat: *Warning: Each child in an array or iterator should have a unique 'key' prop. Check the top-level render call using 'ul'*. Opravený kód podle chybové hlášky by vypadal takto:

```
ReactDOM.render(  
  <ul>  
    {[1, 2, 3, 4, 5].map((item) =>  
      <li key={item}>{item}</li>  
    )}  
  </ul>,  
  document.getElementById('root')  
);
```

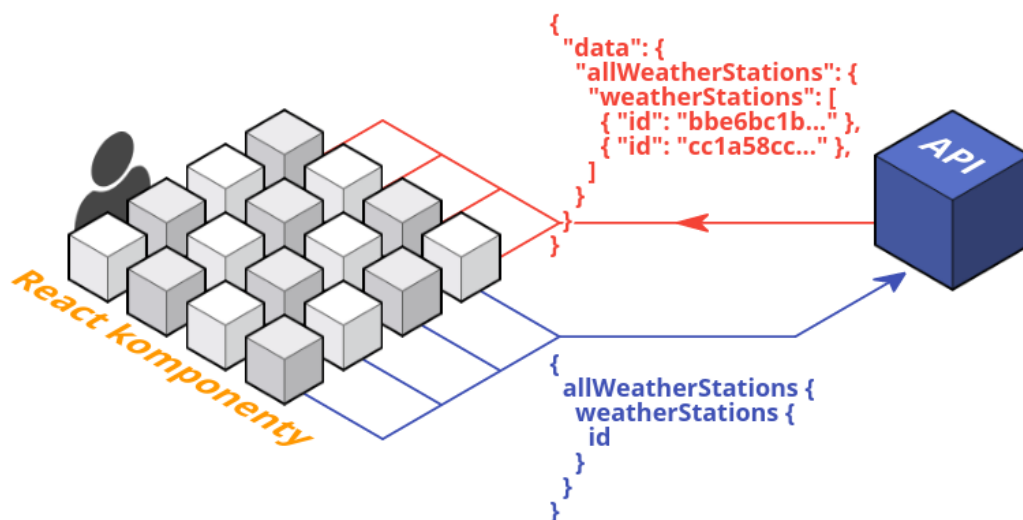
V čem je problém? React totiž už při vykreslování ví, že pokud každá položka neobsahuje svůj jedinečný klíč, tak při změně dat v určité situaci bude překreslení DOMu velmi výpočetně náročné. V předchozím příkladu je v seznamu pět čísel, ale těchto položek mohou být stovky. Pokud se tento výčet změní (například prostřednictvím vlastností komponenty), tak bude React hledat, kde je změna a na tomto místě aplikuje do DOMu opravu. Vymění se tak jen nutná část pro překreslení a původní položky zůstanou. To se stane i bez klíčů, pokud přidáme položku na konec seznamu.

Jenže existuje mnohem horší případ a tím je přidání položky na začátek seznamu. V tom případě React porovná první položku a zjistí, že je DOM jiný. Druhá položka je však také jiná (protože se celý seznam posunul) a tak to jde dál až do konce. Je tedy nutné překreslit celý seznam. To je ale zbytečné kvůli jedné nové položce. Právě kvůli tomuto problému se používají klíče. Pokud je klíč unikátní, React může porovnat pouze klíče a snadno tak zjistí, že je nová položka na začátku seznamu. Překreslení je pak triviální.

Důležité je použít pro klíč nějakou hodnotu, která dostatečně dobře charakterizuje obsah HTML tagu. Je tedy nevhodné použít indexy pole, protože ty se mohou změnit (i když data ne). Stejně tak je nevhodné použít náhodnou hodnotu. Prakticky se pro tyto účely většinou používá ID z databáze. V předchozí ukázce je to sama hodnota, to je však krajní případ.

3.3.4 Komunikace se serverem pomocí GraphQL

Všem komponentám z předchozích ukázek je možné říkat prezentační komponenty. Tyto komponenty pouze prezentují (zobrazují) určitý stav na základě vstupů. Jedná se tedy o naprosto předvídatelné komponenty, které v mnoha případech nemusí mít ani vnitřní stav. Tímto stylem by však šlo napsat akorát statickou webovou stránku.



Obrázek 3.9: GraphQL komunikace se serverem přes API
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

Většinou je potřeba data někde získat. Od toho jsou nadřazené komponenty - tzv. kontejnery. Je nutné zdůraznit, že komponenta a kontejner je pořád to samé a píšou se stejně. Pro lepší návrh se však často rozlišuje jaké má komponenta závislosti. Takže pokud komponenta něco vykresluje, tak se stará **pouze** o vykreslování a data získává pomocí vlastností. Pokud se komponenta stará o získávání dat, tak jí říkáme kontejner (nebo nadřazená komponenta) a neměla by dělat nic jiného. V této práci se pro komunikaci se serverem využívají asynchronní Redux akce [20]. Jak funguje Redux bylo podrobněji rozepsáno dříve. Nadřazené komponenty (kontejnery) pro komunikaci se serverem využívají GraphQL API. Toto API je vysvětleno podrobně později v samostatné sekci.

Příkladem kontejneru je komponenta pro získání všech kamer, které jsou k dispozici. Protože se tato komponenta nebude starat o vykreslování, ale předá pouze data další komponentě na nižší úrovni, tak sama o sobě není moc zajímavá. Zajímavější je její napojení do stromu komponent. Většinou je každá komponenta ve vlastním souboru a pouze se exportuje pro použití jinde. Kontejner je však ještě obalen do nadřazené komponenty, který ji připojuje do Redux úložiště. Díky tomu je pak možné spustit akci pro načtení všech kamer přesně v okamžiku, kdy se komponenta připojuje do DOMu:

```
export const AllCamerasContainer = class extends
  ↪ React.Component {
```

```

componentWillMount() {
  this.props.dispatch(loadAllCameras()); // <-
}

render = () => { /* ... */ }
};

export default connect()(AllCamerasContainer);

```

Tímto stylem se vlastně z původní komponenty `AllCamerasContainer` stane ta samá komponenta, která již však (na rozdíl od původní) může pracovat s Reduxem. Došlo tedy k oddělení závislostí: kontejner získává data ze serveru a o nic víc se již nestará. Naopak předává tyto data nějaké další komponentě. Tato prezentační komponenta je díky tomu velmi jednoduchá a hlavně jí nezáleží na původu dat. To se hodí i pro testování.

3.4 Architektura streamovací aplikace

Streamovací aplikace slouží pouze pro zpracovávání obrazu z kamer. Tato menší aplikace je naprogramována jako mikroslužba se kterou se komunikuje prostřednictvím jednoduchého API. Ačkoliv by mohl být tento kód součástí hlavního serveru, není tomu tak. Streamování totiž vyžaduje zcela jiné nároky na výpočetní výkon, a proto se vyplatí mít samostatný server, který se o tento problém stará.

Ve svém jádru je tato aplikace velmi podobná hlavnímu serveru. Na rozdíl od něj je však naprogramována minimalisticky, takže je rychlá a moc toho neumí. Prakticky je možné pouze zapnout zpracování streamu a následně jej zase vypnout. Počítá se s tím, že aplikace bude fungovat pouze v rámci vnitřní sítě, takže není potřeba řešit ani žádné zabezpečení. Hlavní server jednoduše odešle na tuto službu požadavek na zpracování streamu a služba odpoví URL adresou, ze které je možné zpracované video přehrávat.

3.4.1 Zapnutí a vypnutí zpracování streamu

Vnitřně se hlavní aplikace ovládá pomocí příkazů, které vytvářejí nové záznamy meteorologických nebo třeba nový záznam pro webovou kameru. Vytvářením je myšleno uložení nového záznamu do databáze. Právě tyto příkazy jsou vhodné místo pro vykonání dalších operací jako je třeba zapnutí konverze streamu videa. Server to dělá tak, že v době ukládání informace o nové kameře

do databáze odesílá na streamovací server POST požadavek na zapnutí streamování. Vše probíhá v jedné transakci, takže kdyby se náhodou nový stream na vzdálené službě nepodařilo zapnout, tak se ani v databázi nevytvoří nová kamera.

Streamovací server má dvě URL adresy: `/startStream` a `/stopStream`. První zmíněná adresa očekává v těle požadavku adresu původního streamu (tedy to, co nabízí webová kamera) a druhá adresa očekává ID streamu, který má být zastaven. Následuje ukázka zapnutí streamu pomocí příkazu `curl`, který umí POST požadavky posílat z příkazové řádky [23]:

```
curl --data "source=rtsp://stream.source" \
      http://stream.adeira.loc/startStream
```

Streamovací aplikace odpoví ve formátu JSON, který obsahuje ID nového streamu (pro pozdější zastavení), dále původní adresu a hlavně novou adresu HLS playlistu. Ta je relativní, takže je možné měnit adresu tohoto serveru:

```
{
  "data": {
    "id": "d736a5ff-7b91-4526-93cd-64c2fed230e8",
    "source": "rtsp://stream.source",
    "hls": "\hls/Y9c7gqdbevAzww7LfJYbJg/stream.m3u8"
  }
}
```

Obdobně funguje zastavování:

```
curl --data "identifier=d736a5ff-7b91-4526-93cd-64c2fed230e8"
      http://stream.adeira.loc/stopStream
```

Tentokrát se vrátí pouze ID smazaného streamu jako potvrzení úspěšné operace:

```
{
  "data": {
    "identifier": "d736a5ff-7b91-4526-93cd-64c2fed230e8"
  }
}
```

Tato aplikace si zachovává podobné chování jako hlavní server, takže pokud například dojde k chybě, tak aplikace odpoví v podobném JSON formátu, který obsahuje `error` pole:

```
{
  "errors": [
    {
      "message": "Stream with identifier
                  'd736a5ff-7b91-4526-93cd-64c2fed230e8'
                  is not registered!"
    }
  ]
}
```

Co se v této aplikaci děje po přijetí požadavku na zpracování streamu? Aplikace obsahuje minimalistickou databázi SQLite [25], do které se ukládají streamy pro zpracování. Na pozadí potom běží tolik procesů, kolik existuje zaregistrovaných streamů v databázi. Jedná se o procesy programu FFmpeg, jehož jediným úkolem je přijímat stream a překládat jej na HLS playlist a příslušné útržky videa.

3.5 Architektura konkretizačního uzlu

Konkretizační uzel je jednoduchý program, který slouží pro převod unikátního formátu dat meteostanice na formát, kterému rozumí GraphQL API. Toto je jediné místo v celém systému, kde existují konkrétní implementační detaily týkající se jedné konkrétní meteostanice. Cílem bylo, aby byl tento uzel co nejjednodušší. Pokud by byl potřebný program delší než několik desítek řádek kódu, nebyla by tato myšlenka moc použitelná. Implementace konkretizačního uzlu musí být pro koncového uživatele velmi jednoduchá. Jako příklad je zde uveden konkretizační člen stanice zmíněné v úvodní části této práce.

Před samotným odesláním prvních dat je nutné získat JWT token pomocí přihlašovacího jména a hesla. K tomu slouží jednoduchá `login` mutace, která se na server odešle jako POST požadavek:

```
mutation {
  login (username: "user", password: "pass") {
    token
  }
}
```

Tomu co znamenají jednotlivé části dotazu se podrobně věnuje následující kapitola. Důležité je, že server odpoví JWT tokenem, který se používá pro ověřování následujících odesílaných požadavků. Tento token je možné někam

dočasně uložit, protože jeho expirace chvíli trvá. Ničemu však nevadí získat pokaždé nový token. Jen je nutné ptát se přes zabezpečené HTTPS spojení.

Nyní zbývá odeslat nějaká data na server prostřednictvím POST požadavku. Tato meteorologická stanice má bohužel jedno specifikum. Kromě počítače, který je připojený přímo, neumí data nikam odesílat. Konkretizační uzel by mohl běžet na nějakém vzdáleném serveru, ale v tomto případě musí být puštěn přímo na počítači u meteostanice. Zde vždy získá poslední záznam ze souboru `history.dat` a odešle jej přes GraphQL API na server. Data se opět odesílají díky mutaci, jejíž vstupní data mají tuto podobu:

```
{
  id: '00000000-0001-0000-0000-000000000001',
  input: {
    absolutePressure: 966.4,
    relativePressure: 1006.4,
    indoorTemperature: 24.4,
    outdoorTemperature: null,
    indoorHumidity: 32,
    outdoorHumidity: null,
    windSpeed: null,
    windAzimuth: null,
    windGust: null,
    pressureUnit: 'PASCAL',
    humidityUnit: 'PERCENTAGE',
    windSpeedUnit: 'KMH',
    temperatureUnit: 'CELSIUS'
  }
}
```

Z ukázky je vidět, že meteostanice neumí všechny fyzikální veličiny. Chybějící položky je možné odeslat s hodnotou `null`, nebo je úplně vypustit. Asi nejzajímavější částí jsou jednotky jednotlivých fyzikálních veličin. Hlavní server totiž umí velmi dobře pracovat s jednotkami a interně provádí přepočty podle potřeby. V současné době je možné použít tyto jednotky:

- tlak: pascal, bar, torr, fyzikální atmosféra
- teplota: Celsius, Fahrenheit, Kelvin
- vlhkost: procento
- rychlost: km/h, míle/h, m/s

Tyto jednotky není potřeba přes API odesílat, pokud se jednotky jednotlivých hodnot neliší od výchozích. Předchozí data by tedy bylo možné zjednodušit takto:

```
{
  id: '00000000-0001-0000-0000-000000000001',
  input: {
    absolutePressure: 966.4,
    relativePressure: 1006.4,
    indoorTemperature: 24.4,
    indoorHumidity: 32
  }
}
```

Význam je stejný. Samozřejmě platí, že odesílaná data jsou v JSON formátu a je tedy možné z požadavku odstranit bílé znaky. Zde jsou ukázky rozepsány pouze pro přehlednost. Na rychlost by to zde nemělo žádný zásadní vliv, v praxi se však ukazuje, že odpovědi ze serveru je vhodné takto minifikovat, protože velikost přenášených dat může klesnout až na polovinu⁵.

Slovo architektura je v případě konkretizačního uzlu až příliš nadnesené. V součtu má program asi 100 řádek kódu a většina je pouze definice binárního formátu dat. Kód, který něco skutečně vykonává má jen asi 20 řádek. Bez okolního kontextu by však nedával smysl (protože využívá externí knihovny) a je proto k dispozici na elektronickém médiu, které je přiloženo k této závěrečné práci.

3.6 GraphQL API

GraphQL [26] je API, které bylo vynalezeno společností Facebook pro chytřejší komunikaci klientských částí se serverem. Jeho velkou předností je, že se jedná o dotazovací jazyk. Je tedy možné doptat se serveru na libovolnou podmnožinu dat. To je rozdíl například oproti REST API (nebo téměř jakémukoliv jinému API). Většinou je totiž API navrženo tak, že nabízí jasné definované koncové body na které se uživatel může ptát. Z těchto bodů se také většinou vrací jasné definovaná data. GraphQL ale funguje úplně jinak.

GraphQL API nabízí pouze jednu URL adresu, na kterou je možné posílat dotazy. To je zpravidla adresa `/graphql`. Na tuto adresu se posílají POST HTTP požadavky. Tělo těchto požadavků pak nese informaci o tom, jaká data chceme získat. Tato informace může být téměř libovolná. Na straně serveru se pak vytvoří graf všech dat, která jsou v API k dispozici. Jako příklad uveďme jednoduchý dotaz na seznam všech meteorologických stanic. Klient pošle POST požadavek na server v tomto znění (zjednodušeno):

⁵Prakticky ověřeno na odpovědi vrácené z GraphQL API společnosti Kiwi.com. Původní odpověď o velikosti 522 KB se odstraněním bílých znaků zmenšila na 266 KB.

```
POST /graphql HTTP/1.1
Host: connector.adeira.loc
Connection: keep-alive
Content-Length: 91
accept: application/json
content-type: application/json
Accept-Encoding: gzip, deflate
Accept-Language: en-US,en;q=0.8
```

Tělo požadavku obsahuje následující JSON data:

```
{
  "query": "{allWeatherStations{weatherStations{id}}}",
  "variables": null,
  "operationName": null
}
```

Za povšimnutí stojí hodnota `query`, která obsahuje GraphQL dotaz. Ten má vlastní zápis, který na první pohled vypadá jako zjednodušený JSON:

```
{
  allWeatherStations {
    weatherStations {
      id
    }
  }
}
```

V tomto dotazu se ptáme na všechny meteorologické stanice a u těchto stanic nás zajímá pouze jejich identifikátor. Server následně odpoví návratovým kódem HTTP/1.1 200 OK a společně s hlavičkami pošle požadovaná data:

```
{
  "data": {
    "allWeatherStations": {
      "weatherStations": [
        { "id": "bbe6bc1b-386f-4b6d-91e4-379c7882792c" },
        { "id": "cc1a58cc-1fd9-46f9-a193-d7e942a01019" }
      ]
    }
  }
}
```

Z této krátké ukázky by měla být vidět obrovská výhoda GraphQL. Data totiž nejsou nijak vázána na URL adresu, ale výběr požadovaných dat se provádí popsáním podmnožiny grafu, který nás zajímá. Ten je poslán jako tělo POST požadavku, takže se může kdykoliv libovolně měnit a podle toho se změní také odpověď.

Předchozí ukázka obsahuje pouze dotazování se na data. To by však bylo pro API málo. Server je potřeba i ovládat a od toho jsou zde mutace. Tyto dvě hlavní skupiny jsou kromě dalších vlastností API popsány dále.

3.6.1 Typy (Types)

Aby bylo možné získávat z API nějaká data, je potřeba definovat tzv. typy. Ty formují strukturu navrácených dat. Předchozí příklad s dotazem na všechny identifikátory meteostanic obsahoval tři typy. Ty sice na první pohled nejsou vidět, ale pomocí nich je API nadefinováno. Předpis těchto typů by vypadal následovně:

```
type Query {  
  allWeatherStations: WeatherStationsConnection  
}  
  
type WeatherStationsConnection {  
  totalCount: Int!  
  weatherStations: [WeatherStation]  
}  
  
type WeatherStation {  
  id: ID!  
  name: String!  
}
```

Tyto typy toho obsahují ve skutečnosti více, ale pro pochopení to pro teď stačí. První typ `Query` definuje sadu všech dotazů, které můžeme položit. Pokud se zeptáme jako v předchozím případě na všechny meteostanice, tak dostaneme `Connection` typ. Na této úrovni můžeme doplnit nějaké další potřebné informace, jako je celkový počet dostupných stanic a samozřejmě samotné stanice. Vykřičník v návratové hodnotě značí povinnou položku. Konečně dotazem na typ meteostanice můžeme získat ID stanice nebo třeba její jméno. Všechny typy jsou k dispozici v interaktivní dokumentaci API, takže sestavit správný dotaz není složité.

3.6.2 Dotazy (Queries)

Již v předchozích ukázkách byl použit jednoduchý dotaz pro získání ID všech meteostanic. Tyto dotazy umožňují získat z API téměř libovolnou informaci. Jednou z nejužitečnějších vlastností jsou argumenty dotazu. Zatímco vytažení všech meteostanic nepotřebuje žádné argumenty, tak pokud chceme pouze jednu konkrétní stanici, musíme specifikovat jakou. To jde v GraphQL udělat třeba takto:

```
{
  weatherStation(id: "bbe6bc1b-386f-4b6d-91e4-379c7882792c") {
    id
    name
  }
}
```

Odpověď je opět JSON obsahující dotázané informace (ID a název). Užitečnost argumentů je pak ještě lépe vidět ve spojení s proměnnými.

```
query ($id: ID!) {
  weatherStation(id: $id) {
    id
    name
  }
}
```

Toto je plnohodnotný zápis dotazu a lze jednoduše použít jako takový předpis pro jeden typ dotazů. Souběžně s tímto dotazem se totiž posílá i JSON, který obsahuje ID. Toto ID se může libovolně měnit a dotaz zůstává pořád stejný.

Další příjemnou vlastností je možnost zeptat se na více věcí najednou:

```
query ($stationId: ID!, $userId: ID!) {
  weatherStation(id: $stationId) {
    id
    name
  }
  user(id: $userId) {
    id
    username
    token
  }
}
```

Opět je nutné poslat souběžně hodnoty proměnných a opět se vrátí JSON obsahující požadovaná data. GraphQL je velmi intuitivní a díky interaktivní dokumentaci API je pokládání dotazů velmi jednoduché a dokonce i s napovídáním.

Díky tomu, že se API skutečně chová jako graf, tak je možné pokládat dokonce rekurzivně zanořené dotazy (uživatel vlastní meteostanice, kde každá meteostanice je vlastněna nějakým uživatelem, kde každý uživatel vlastní nějaké meteostanice...). Proto je nutné na serveru dávat pozor na přílišnou složitost položeného dotazu a případně odpovědět chybou.

3.6.3 Mutace (Mutations)

Podobně jako dotazy získávají ze serveru data, tak mutace data na serveru mění. Zároveň však data stejně jako dotazy získávají. Rozdíl je pouze v tom, že mutace mají trochu jiný zápis a mohou něco změnit. Navíc mutace jsou zpracovávány sériově, kdežto dotazy mohou být na serveru pracovány souběžně (nehrozí kolize). Příkladem velmi jednoduché mutace je přihlašování:

```
mutation {  
  login(username: "test", password: "test") {  
    id  
    username  
    token  
  }  
}
```

Je vidět, že vnitřek dotazu je stejný jako u „queries“. Dotaz však začíná klíčovým slovem `mutation` a volá nějakou konkrétní mutaci (zde `login`). Server v případě úspěšného přihlášení odpovídá mimo jiné JWT tokenem, který je dále používán pro ověřování místo přihlašovacího jména a hesla. O něco složitější mutace je potřeba pro vytvoření nového záznamu meteostanice:

```
mutation create(  
  $stationId: ID!, $quantities: PhysicalQuantitiesInput!  
) {  
  createWeatherStationRecord(  
    id: $stationId, quantities: $quantities  
  ) {  
    id, creationDate  
  }  
}
```

Jak je vidět, tak v argumentech není nutné posílat pouze skalární hodnoty, ale je možné poslat celý objekt. Ten musí být vždy odeslán souběžně ve formátu JSON. Pro předchozí mutaci by tělo požadavku mohlo vypadat třeba takto:

```
{
  "stationId": "00000000-0001-0000-0000-000000000001",
  "quantities": {
    "absolutePressure": 100000,
    "relativePressure": 100000,
    "pressureUnit": "PASCAL"
  }
}
```

Návratová hodnota ze serveru je také ve formátu JSON a obsahuje ID nového záznamu a čas vytvoření tohoto záznamu:

```
{
  "data": {
    "createWeatherStationRecord": {
      "id": "4b9126b3-cbed-4a63-9acf-49a61f6a9fc0",
      "creationDate": "2017-03-12T14:37:31+01:00"
    }
  }
}
```

Mutace jsou velmi podobné dotazům. Stačí pouze pochopit jak fungují dotazy a uvědomit si, že mutace mohou měnit stav aplikace. Pomocí těchto dvou jednoduchých metod je možné vykonat jakoukoliv myslitelnou operaci, kterou API umožňuje. To je vlastně také hlavní myšlenka celé serverové aplikace - poskytnout dostatečně mocné API a žádné uživatelské rozhraní.

3.6.4 Stránkování v GraphQL

Stránkování je speciální požadavek na API, který jde mimo všechny ostatní. Můžeme například chtít získávat meteostanice po deseti tak, aby bylo možné vytvořit jednotlivé stránky s jejich kratším přehledem. Podobně jako se to dělá na internetových obchodech pro stránkování produktů. Přitom většina API je postavená tak, že zobrazí jeden záznam nebo všechny záznamy. Jak si s tímto problémem poradit? Řešením jsou uzly a hrany grafu:

```
query paginateStations {
  allWeatherStations(first: 10) {
```

```

    totalCount
    edges {
      cursor
      node {
        id # konkrétní meteostanice
      }
    }
  }
}

```

Takto lze získat prvních deset meteostanic. Je vidět, že samotné tělo dotazu je mnohem složitější než obvykle. Na první úrovni jsou hrany grafu (edges) a jejich celkový počet. Celkový počet hran je zároveň celkový počet meteostanic. Každá hrana následně obsahuje kurzor a uzel grafu. Uzel grafu obsahuje jednotlivé položky meteostanice, které nás zajímají. Kurzor potom představuje ukazatel na konkrétní hranu grafu. Skvělé na tomto návrhu je to, že získáme požadovaných deset meteostanic a zároveň víme jejich celkový počet a známe kurzor poslední hrany (např. CUr50r), takže dotaz na dalších deset meteostanic je velmi přímočarý:

```

query paginateStations {
  allWeatherStations(first: 10, after: "CUr50r") {
    totalCount
    edges {
      cursor
      node {
        id # konkrétní meteostanice
      }
    }
  }
}

```

Existuje více způsobů jak pracovat se stránkováním a i předchozí způsob je nutné pomocí GraphQL vytvořit. Je tedy na programátorovi jaký způsob zvolí. Kurzorový přístup se však velmi osvědčil a pro GraphQL API je vhodným kandidátem. Může se zdát, že je to až příliš složité, ale pokud si uvědomíme, kolik informací lze jedním dotazem získat, tak to za tu námahu stojí.

3.6.5 Pokročilé možnosti GraphQL

Všechny dříve popsané možnosti GraphQL API plně dostačují pro téměř jakýkoliv požadavek. Existuje však celá řada vylepšení a pomůcek, které

umožňují pokládané dotazy zpřehlednit, případně doplnit o nějaké další vlastnosti. Následuje krátký výčet těchto pokročilých možností, který je zde uveden spíše pro úplnost.

3.6.5.1 Aliasy

Pomocí aliasů lze změnit názvy jednotlivých hodnot, které API vrací. Aliasy se uvádějí před výraz a končí dvojtečkou:

```
query getName {
  uživatel: user(id: "00000000-0000-0000-0000-000000000001") {
    jmeno: username
  }
}
```

Původní dotaz by vrátil odpověď obsahující slova *user* a *username*. Nově však vrátí jejich české varianty:

```
{
  "data": {
    "uzivatel": {
      "jmeno": "John Doe"
    }
  }
}
```

3.6.5.2 Fragments

Některé dotazy mohou být velmi složité a může se stát, že se některé části dotazu budou opakovat. V GraphQL si lze ušetřit práci pomocí fragmentů. Každý fragment obsahuje výčet polí nad určitým typem a tento fragment lze používat na více místech. Fragments se uplatní až u rozsáhlejších dotazů, ale pro ukázkou zápisu následuje jednoduchý příklad:

```
query getUserInfo {
  user(id: "00000000-0000-0000-0000-000000000001") {
    ...UserFields
  }
}

fragment UserFields on User {
  id, username, token
}
```

Jak lze očekávat, tak API vrátí ID uživatele, jeho uživatelské jméno a JWT token. Pokud by bylo potřeba získat v jiné části dotazu ta stejná data, tak je možné volání fragmentu opakovat. Fragment lze také zapsat jako součást dotazu:

```
query getUserInfo {  
  user(id: "00000000-0000-0000-0000-000000000001") {  
    ... on User {  
      id, username, token  
    }  
  }  
}
```

V tomto jednoduchém příkladu to však nedává smysl, protože stejného efektu lze dosáhnout i bez fragmentu. Tato varianta se hodí pouze v případě, že by daný uzel grafu vracel více typů. Pak je možné na základě typu pracovat s jinými hodnotami. Ve spojení s aliasy to umožňuje pokládat poměrně zajímavé dotazy.

3.6.5.3 Direktivy

Pomocí direktiv lze podmiňovat, případně jinak měnit chování položeného dotazu:

```
query getUsernameOrToken($switch: Boolean!) {  
  user(id: "00000000-0000-0000-0000-000000000001") {  
    username @skip(if: $switch)  
    token @include(if: $switch)  
  }  
}
```

Tento příklad ukazuje dvě vestavěné direktivy podle specifikace. Pomocí přepínače je tak možné získat uživatelské jméno nebo token, ale ne obojí najednou. Díky tomu je možné vytvořit složité dotazy a podmiňovat, jestli chceme úplný výběr nebo jen podvýběr. Tyto direktivy je možné přidávat i vlastní. Zde je však nutné se nejdříve zamyslet, jestli je to skutečně nutné.

3.6.5.4 Introspekce

Nástroje jako GraphQL [27] umí zobrazit celé schéma API a napovídají jeho možnosti při psaní dotazu. Jak je to ale možné? Jak tento nástroj sám od sebe ví, co API umí? K tomu slouží tzv. introspekce. GraphQL totiž nabízí

možnost, jak API prozkoumávat a zjišťovat potřebné podrobnosti. Princip je pořád stejný, jen tyto systémové typy začínají dvojicí podtržíték:

```
{
  __schema {
    directives {
      name, description, locations
      args { name, description }
    }
  }
}
```

Tímto dotazem lze například získat informace o dříve použitých direktivách - jejich názvy, popis funkce, kde je lze použít a jaké mají argumenty. Stejným způsobem je možné zeptat se na téměř cokoliv. Celé API je tedy velmi transparentní a samo sobě dělá jednoduchou dokumentaci.

3.6.6 Chybové stavy v GraphQL

V průběhu pokládání dotazů se může stát, že vznikne nějaká chyba. Chyby vznikají a není potřeba se jich bát, ale zejména v API je s nimi potřeba dobře zacházet. To znamená, že úspěšná odpověď je pouze tehdy, pokud server vrátí HTTP kód 200. Cokoliv jiného je pravděpodobně známka chyby. V GraphQL mohou nastat tři hlavní chybové stavy. První je „syntax error“, pokud uživatel odešle na server nevalidní formát dotazu. Kromě toho, že server odpoví kódem 422, tak odešle také JSON popisující, kde je problém:

```
{
  "errors": [
    {
      "message": "
Syntax Error (1:14) Expected :, found String \"User\"
1: {__type(name \"User\")}{kind}}
      ^",
      "locations": [
        { "line": 1, "column": 14 }
      ]
    }
  ]
}
```

Odpověď je ve formátu JSON, takže může působit na první pohled nepřehledně, ale hned v zápětí je vidět jaký dotaz byl odeslán a kde je chyba. Toto je důvod, proč jsou vrácená data z tohoto API vždy zanořena v klíči **data**. Chyby se totiž posílají s klíčem **errors**. Podobně může vzniknout validační chyba, kde je sice dotaz zapsán správně, ale nesplňuje předem dané požadavky. Příkladem může být hodnota s jiným datovým typem v argumentu, než je očekáván: `Argument 'name' got invalid value User. Expected type 'String', found User.`

Posledním příkladem je vnitřní chyba serveru. V tomto případě je dotaz validní a správný, ale server selhal a proto vrací pole errorů jako v předchozím příkladu, ale navíc i pole dat s NULL hodnotami na místě selhání:

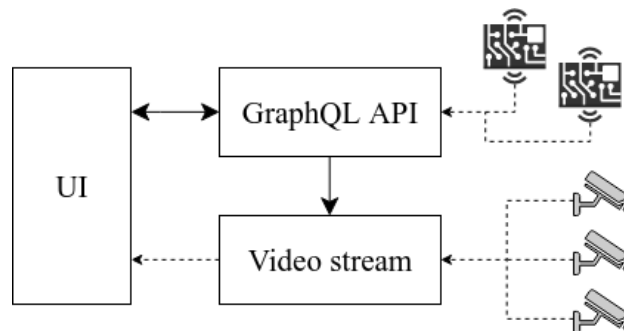
```
{
  "data": null,
  "errors": [
    {
      "message": "Internal Server Error."
    }
  ]
}
```

Nyní je na uživateli tohoto API, aby se k takovým chybám náležitě choval. Může je zobrazit uživateli ve webové aplikaci, nebo je tiše ignorovat. Záleží na konkrétním případě.

4

Výsledná aplikace a její provoz

Výsledkem této práce jsou tři nezávislé projekty. Nejsložitější je server, který poskytuje GraphQL API a celkově se stará o uchovávání a zpracování dat. Sám o sobě nemá žádné užitečné uživatelské rozhraní. Od toho je zde druhá aplikace, která tvoří toto uživatelské rozhraní a sama o sobě nemá žádnou databázi. Pro své fungování vyžaduje přítomnost API serveru, protože s ním neustále komunikuje. Poslední aplikací je samostatná služba pro konvertování streamu videa. Díky tomu se každá část stará pouze o své věci a vzájemně spolu pouze komunikují, viz obrázek 4.1.



Obrázek 4.1: Celkový pohled na výslednou aplikaci

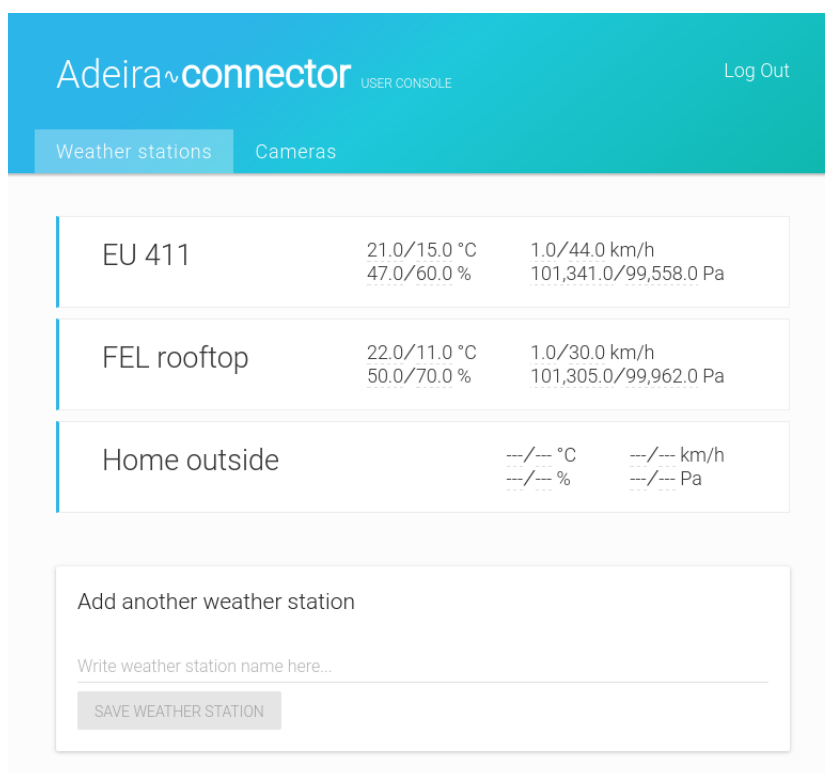
Navíc většina zdrojů dat má k dispozici konkrétní členy, které se starají o převod individuálního formátu dat zdroje pro API. Tak je zajištěna slučitelnost dvou různých systémů a samotné API může být na zdroji dat nezávislé.

4.1 Uživatelské rozhraní

Uživatelské rozhraní bylo vytvářeno záměrně velmi jednoduše podle hesla „méně je někdy více“. První co uživatel při první návštěvě aplikace spatří

je přihlašovací formulář. Přihlášení je přesně ten okamžik, kdy uživatel získá JWT token popisovaný dříve a uloží si jej do lokálního úložiště v prohlížeči. Veškerá následující komunikace probíhá pouze díky tomuto tokenu.

Po přihlášení uživatel hned vidí přehled všech meteostanic, jejich poslední záznam a má také možnost novou meteostanici vytvořit (obr. 4.2). Jak je vidět, tak při vytváření nové meteostanice lze zadat pouze její název, nikoliv typ. Jak bylo vysvětleno v dřívějších kapitolách, tak samotná aplikace nijak nepotřebuje řešit typ meteostanic. Převod dat do jednotného formátu je vyřešen pomocí konkretizačních členů. Po vytvoření stanice uživatel získá nové ID meteostanice, tím i způsob, jak přes GraphQL API zasílat nová data.

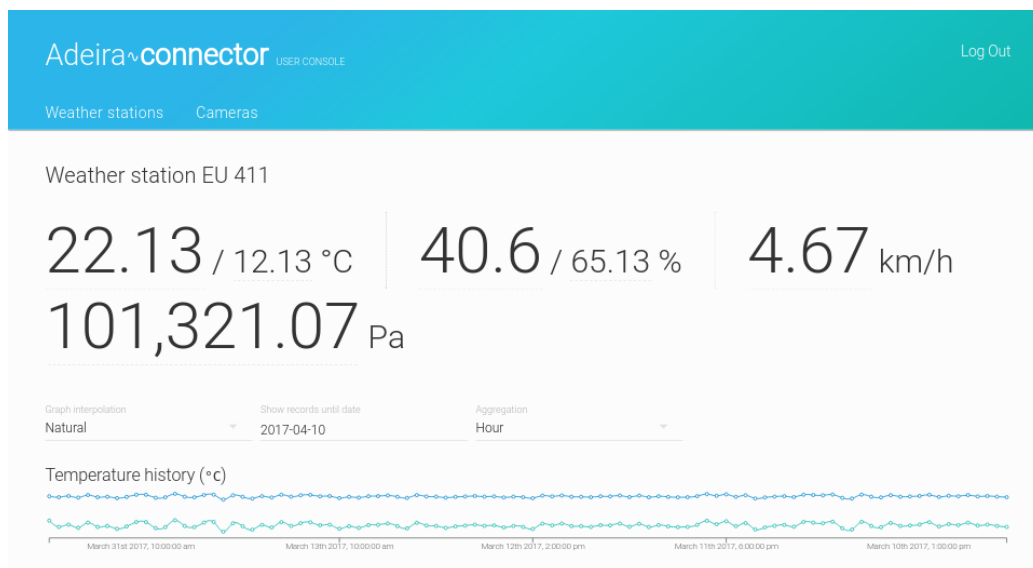


Obrázek 4.2: Úvodní stránka klientské aplikace

Vybráním jedné stanice je hned vidět její podrobný přehled. Zde je mimo jiné možné nahlédnout do historie naměřených záznamů od určitého data s možností výběru typu agregace dat. Záznamů totiž může být velmi mnoho, proto se na serveru počítají agregované přehledy. Data je možná prohlížet si s přesností na hodiny, popř. zvětšit časový rámec a data sdružovat po dnech, víkendech a měsících. Tak se lze teoreticky podívat na posledních 100 měsících

(pokud jsou data k dispozici) v rámci jednoho grafu.

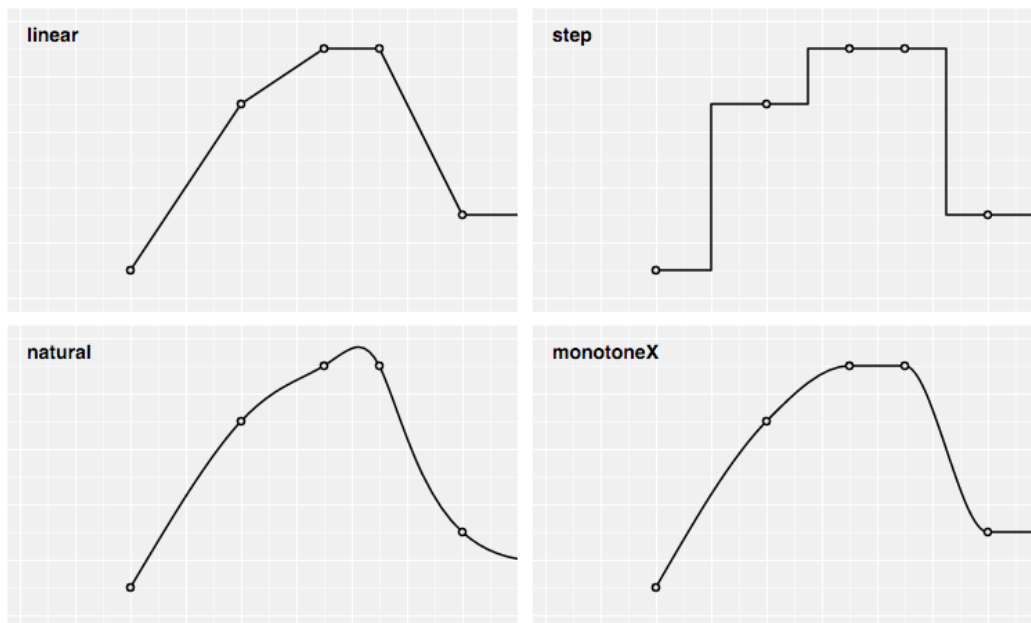
Kromě výběru počátečního data a typu agregace dat je také možné zvolit způsob interpolace bodů grafů. Body lze prokládat lineární křivkou, kubickými interpolacemi, případně krokovou (skokovou) interpolací s bodem uprostřed roviny. Záleží pak na uživateli, jaká interpolace mu vyhovuje nejvíce (obrázek 4.4).



Obrázek 4.3: Detail meteorologické stanice v aplikaci

Přehled všech IP kamer je podobný jako u meteostanic. Zde je možné kameru přidat, přehrát její stream a kameru zase odebrat. Prakticky se jedná pouze o seznam video přehrávačů (viz obr. 4.5), kdy každý přehrávač umí přehrát vytvořený HLS stream. Popis tohoto streamu a jeho převedení z důvodního formátu IP kamery je popsán v dřívějších kapitolách.

Celé toto uživatelské prostředí funguje díky spojení Reactu a Reduxu. K načtení aplikace dojde pouze při prvním otevření stránky a následně všechny aktualizace dat probíhají bez viditelného a obtěžujícího obnovení celé stránky. Jedno z možných vylepšení by bylo renderovat načítanou stránku na serveru. Uživatel by tak získal hotovou stránku včetně naplněného Redux úložiště. Každý další požadavek by byl obsloužen přes API stejně, jako je tomu teď. Výhoda by byla v rychlejším prvním načtení stránky. Rendrování na straně serveru je nutné také v okamžiku, pokud budou webovou stránku indexovat webové vyhledávače. To ale není případ této administrace, která je schována za přihlášením.

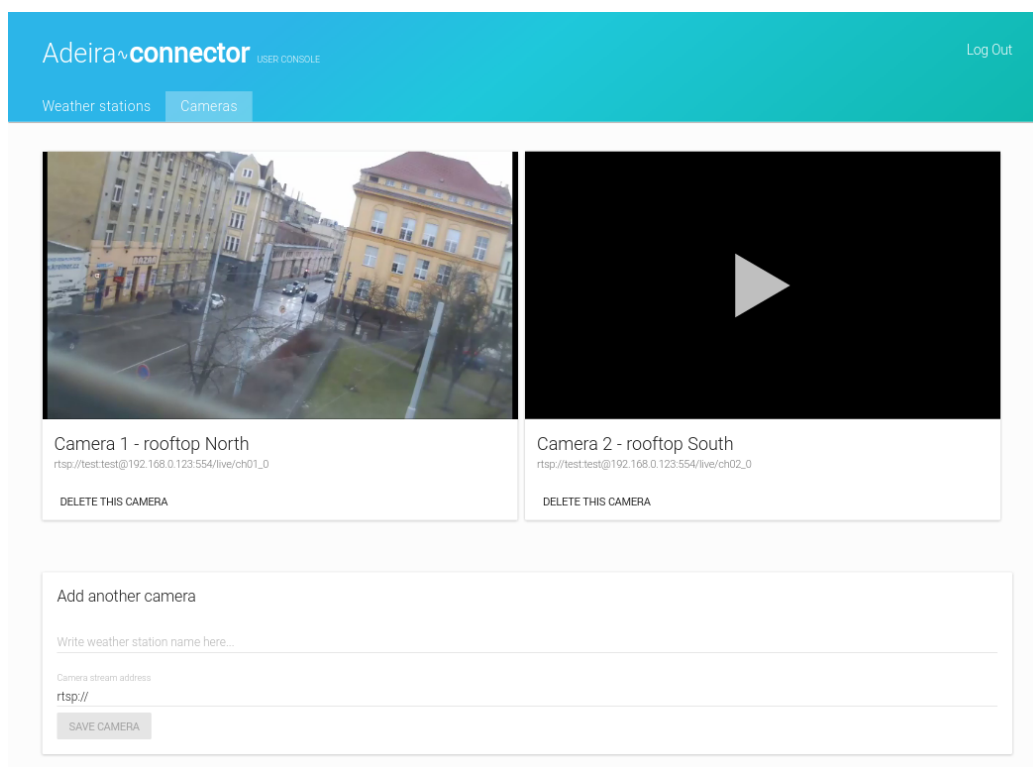


Obrázek 4.4: Způsoby interpolace grafů
zdroj: <https://github.com/d3/d3-shape>

4.2 Doporučená infrastruktura

Na začátku této práce jsem měl veškeré potřebné nástroje pro vývoj nainstalované na svém vlastní počítači. V průběhu vývoje jsem však postupně přešel s celým PHP, databázemi i webovými servery do prostředí Dockeru [28] a ze svého počítače PHP odinstaloval. Rozdíl je v tom, že dříve bylo nutné veškerý software obstarávat na jednom PC. Nově však stačí mít nainstalovaný Docker, který se chová jako velmi tenká virtualizační vrstva. Takto lze při vývoji zapnout na svém PC několik virtuálních serverů. Díky tomu je možné pohodlně využívat různé databáze nebo dokonce několik různých verzí PHP. Velký přínos je v tom, že tyto virtuální stroje jsou jasně definovány a mohou mít naprosto stejnou podobu, jakou má produkční prostředí. Toho by se s jedním PC dosáhlo těžko.

Docker se nechová jako běžně známé virtualizační platformy. Je totiž běžné, že si virtuální stroj alokuje potřebné prostředky a stroj, na kterém běží, tak velmi vytíží. Docker však využívá prostředky mnohem hospodárněji. Takže zatímco běžných virtuálních strojů je možné zapnout jednotky až desítky, tak tzv. Docker kontejnerů je možné spustit stovky až tisíce na jednom počítači. A to už by byla hodně velká infrastruktura - rozhodně

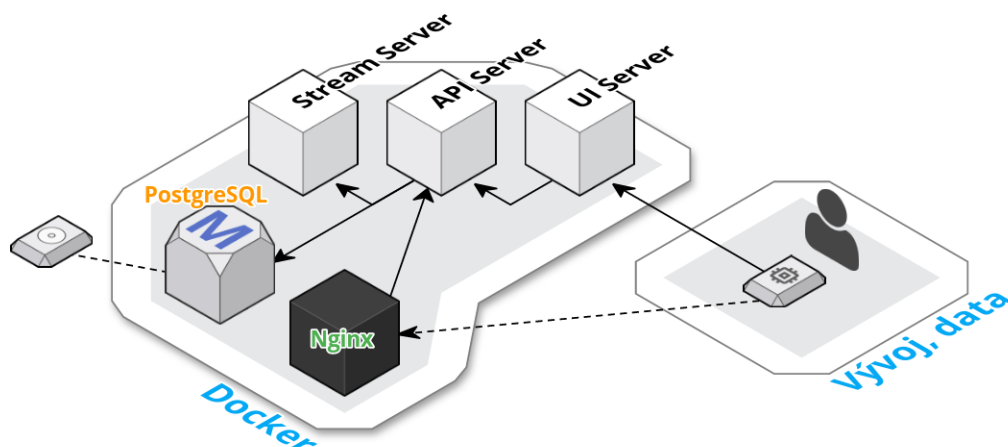


Obrázek 4.5: Přehled všech video streamů v aplikaci

dostatečná pro potřeby této aplikace. Na obrázku 4.6 je znázorněno Docker prostředí, které je pro tuto aplikaci vhodné.

Jak je možné, že Docker zvládne tak velké množství virtuálních strojů? Docker může používat více souborových systémů, ale takovým hlavním zástupcem je AUFS [29]. AUFS je vrstvený souborový systém, jehož hlavní předností je možnost mít sdílené vrstvy operačního systému pouze pro čtení a individuální tenké vrstvy s možností zápisu, které obsahují odlišnosti jednotlivých strojů. Pokud tedy server sdílí společnou část svého systému s jiným serverem, tak si nenárokuje dodatečné prostředky, ale sdílí je. Servery zároveň nemohou nijak kolidovat, protože spodní vrstvy souborového systému jsou pouze pro čtení. Sdílení prostředků není jediný přínos. Nastartování nového serveru v rámci Dockeru trvá zhruba jednu sekundu (mnohdy ani to ne)¹. Je tedy možné spustit proces z příkazové řádky, ale ve skutečnosti spustit celý Docker kontejner a proces spustit izolovaně v kontejneru. Díky extrémní

¹Záleží na tom, co se v kontejneru startuje za procesy. Doporučený postup je však držet kontejnery co nejméně náročné a starající se o jednu věc.



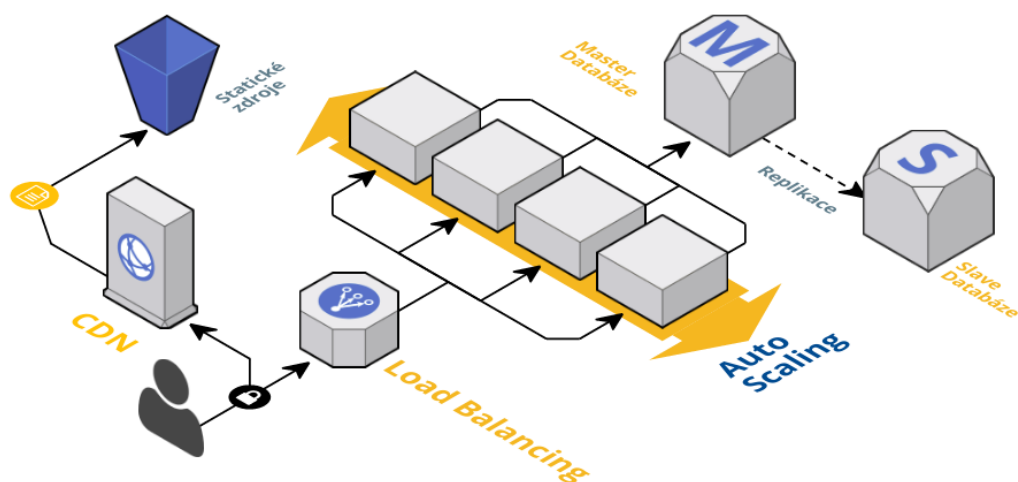
Obrázek 4.6: Infrastruktura použitá při vývoji
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

rychlosti startování je to téměř nezatelná změna.

Ale zpět ke vhodné infrastruktuře. Doposud popisované řešení pomocí Dockeru není jediné možné. Pokud by to bylo nutné, může aplikace fungovat pouze na jednom serveru. Je totiž potřeba pouze webový server, který bude přijímat HTTP požadavky a směřovat je na PHP. Dále je potřeba PostgreSQL databáze a to je vlastně vše. Webové rozhraní může být zatím vybavováno jako statická stránka (neprobíhá renderování na serveru), takže nepotřebuje žádné zvláštní služby. Stejně tak streamovací server pro zpracovávání videa potřebuje pouze PHP. Je však vhodné oddělit jednotlivé závislosti, aby bylo možné škálovat aplikaci nezávisle podle potřeby. Na to se opět perfektně hodí Docker, kde je běžné mít jeden běžící proces v jednom kontejneru (nebo alespoň jeden druh procesu). Proto má webový server i PHP vlastní kontejnery.

Vývojové prostředí může být zcela totožné s produkčním prostředím, ale nemusí tomu tak být. Na produkci je často potřeba dalších služeb, které různým způsobem optimalizují běh aplikace např. pro vysokou zátěž (CDN, vyvažování zátěže, replikace databází). Tento stav je znázorněn na obrázku 4.7.

Takové rozložení strojů je jednoznačně násobně dražší než provoz na jednom serveru. Při rozhodování se o infrastruktuře je nutné dobře propočítat a rozhodnout se, jak velká investice dává smysl. Pokud bych se rozhodl tuto aplikaci provozovat jako službu zákazníkům, tak by bylo nutné se k něčemu podobnému přibližovat. Pro soukromé účely by naopak stačil nějaký obyčejný



Obrázek 4.7: Doporučená infrastruktura
zdroj: Vlastní tvorba - <https://cloudcraft.co/>

sdílený hosting.

4.3 Možnosti rozšíření díky API

Na začátku programování hlavní aplikace nebylo úplně jasné, jak vše dopadne. V současné době jsem však přesvědčen, že vytvoření uzavřené aplikace s API bylo jednoznačně správné řešení. Aplikace se stará **jen** o ukládání a vybavování dat, popř. o autorizaci. Díky tomu je možné soustředit se na to, aby vše fungovalo co nejlépe. Není třeba řešit velké množství různorodých meteo-ostanic. Aplikace vlastně rozumí libovolné meteostanici, protože ona odesílá do API data a ta už mají jednotný formát.

Klíčové je využití GraphQL, které nevytváří statické koncové výstupy, ale umožňuje získávat z API libovolná data. Příkladem využití API mohou být exporty dat. Bylo by možné vytvořit další funkcionalitu a data umožnit exportovat. Ale to nebylo cílem. Server se stará o API. A když existuje grafové API, tak je možné data také exportovat. Například následující GraphQL dotaz vrátí všechny kamery, meteostanice a jejich data:

```
{
  allCameras {
    id, name, stream { source, hls }
  }
}
```

```

allWeatherStations {
  totalCount, weatherStations {
    id, name, allRecords(first: 10) {
      totalCount, returnedCount, records {
        aggregatedDate, absolutePressure, relativePressure
        indoorTemperature, outdoorTemperature,
        indoorHumidity, outdoorHumidity, windSpeed,
        windAzimuth, windGust
      }
    }
  }
}

```

Ze serveru se vrátí JSON odpověď, která je velká několik MB a obsahuje všechny potřebné informace. Není nutné mít předem definovanou strukturu odpovědi. Pouze pokud by bylo třeba exportovat data např. ve formátu XML a mít export velmi rychlý, vyplatí se tuto funkcionalitu dodělat.

Na rozdíl od jiných běžných API je GraphQL (jak název napovídá) grafové. Je nutné dotazovat se na konkrétní cestu v grafu a tato data se vrátí. Neexistuje tedy pevná struktura odpovědi a to je jeden z důvodů, proč toto API není třeba verzovat [30]. Nové vlastnosti lze jednoduše přidat a zastaralé lze schovat (i když budou dále fungovat). Prakticky tedy neexistuje nic jako jedna verze API. API může být každý den jiné, ale vždy zpětně kompatibilní - podobně jako je nutné dělat změny v rámci databázových migrací.

Myšlenka grafového API je natolik mocná, že se dnes začíná GraphQL používat jako proxy pro jiná (starší) API. Funguje to tak, že klienti posílají dotazy na GraphQL a toto API pouze překládá požadavky na např. REST API, které má pevnou strukturu. Tak lze získat všechny vlastnosti GraphQL i zpětně. Není pak složité postupně začít staré API odstraňovat. API lze tedy libovolně rozšiřovat díky jednotnému rozhraní, které je připraveno pro neustálé změny.

5

Hodnocení a závěr

Prvotním cílem práce bylo nalézt způsob, jak získávat data z různých meteorologických stanic a následně je zobrazovat ve vytvořené webové aplikaci. Aby měl uživatel aplikace také vizuální představu o aktuálním počasí, data bylo nutné doplnit o záznam z kamery. Již na samotném začátku práce se ukázalo, že získávat data ze starších meteorostanic je velice komplikované. Takové meteorostanice totiž nejsou nijak připojeny k síti a jediný způsob, jak komunikují, je přes sériovou linku přímo s připojeným počítačem, na kterém musí běžet software od výrobce stanice. Rychle se tak ukázalo, že není možné napsat aplikaci, která by znala konkrétní implementační detaily jednotlivých zařízení. Tento poznatek následně ovlivnil celé výsledné řešení.

Výsledkem této diplomové práce je hlavní server, který nemá žádnou znalost o jednotlivých meteorostanicích. Rozumí však velmi dobře fyzikálním veličinám a s okolním světem komunikuje pouze prostřednictvím GraphQL API. Na toto API jsou z jedné strany napojeny konkretizační členy, které obsahují jednotlivé implementační detaily meteorologických stanic. Z druhé strany API je napojena webová aplikace, která je napsána v Reactu a kde může uživatel celý systém ovládat. API tak slouží pro obousměrnou komunikaci a jeho velkou předností je grafový charakter. Data jsou k dispozici v podobě grafu, takže je možné získat pouze konkrétní podmnožinu tohoto grafu a ušetřit tak přenášená data.

Kromě serveru, který poskytuje GraphQL API, byla vytvořena také aplikace pro streamování videa z webových kamer. Tyto kamery zpravidla neposkytují formát, který je vhodný pro přímé zpracování ve webovém prohlížeči a je proto nutné provádět tzv. „near real-time“ konverzi obrazu. Streamovací server je postaven jako samostatná mikroslužba. Díky tomu, že není součástí serveru, který poskytuje GraphQL API, tak je možné tuto část aplikace nazávisle škálovat a dosahovat tak požadovaného výkonu.

Zadání práce bylo tedy nejen úspěšně vyřešeno, ale také rozšířeno o implementaci GraphQL, které se stalo ústředním bodem celé práce. Výsledná aplikace je díky API a hlavně díky systému konkretizačních členů schopna pracovat s jakoukoliv meteorologickou stanicí, která je na trhu.

Do budoucna by bylo vhodné vylepšit systém konkretizačních členů. Je běžné, že kvůli jedné meteostanici musí běžet celé dny zapnutý kancelářský počítač. Přitom důvod je pouze ten, že obsahuje software od výrobce, sériový port a připojení k internetu. V tom vidím velkou rezervu. Stačilo by připojit stanici k jednoduchému převodníku, který by disponoval sériovým portem a uměl data odesílat do sítě. Velkou překážkou jsou však samotní výrobci meteorologických stanic. Často totiž není k dispozici dostatečně podrobná technická dokumentace a porozumět všem náležitostem komunikace přes sériovou linku může být nepřekonatelný úkol. Naštěstí novější meteostanice umějí odesílat data přímo do sítě a tím tento problém zcela mizí.

Literatura

- [1] BUENOSVINOS, Carlos, Christian SORONELLAS a Keyvan AKBARY. *Domain-Driven Design in PHP* [online]. 2016 [cit. 2017-02-07]. ISBN 978-0-9946084-3-7. Dostupné z: <https://leanpub.com/ddd-in-php>
- [2] *AirCam Security Camery User Guide* [online]. [cit. 2017-03-04]. Dostupné z: https://dl.ubnt.com/guides/AirCam/airCam_UG.pdf
- [3] *A complete, cross-platform solution to record, convert and stream audio and video.* [online]. [cit. 2017-03-04]. Dostupné z: <https://ffmpeg.org/>
- [4] *An extensible media player for the web. (source code)* [online]. [cit. 2017-03-04]. Dostupné z: <https://github.com/clappr/clappr>
- [5] *MSE-based HLS client (source code)* [online]. [cit. 2017-03-04]. Dostupné z: <https://github.com/dailymotion/hls.js>
- [6] *PostgreSQL - the world's most advanced open source database.* [online]. [cit. 2017-03-04]. Dostupné z: <https://www.postgresql.org/>
- [7] *ValueObject* [online]. [cit. 2017-03-04]. Dostupné z: <https://martinfowler.com/bliki/ValueObject.html>
- [8] *Object Interfaces* [online]. [cit. 2017-02-09]. Dostupné z: <http://php.net/manual/en/language.oop5.interfaces.php>
- [9] MARTIN Robert. *Screaming Architecture* [online]. [cit. 2017-02-09]. Dostupné z: <https://8thlight.com/blog/uncle-bob/2011/09/30/Screaming-Architecture.html>
- [10] Microsoft *The Repository Pattern* [online]. [cit. 2017-02-09]. Dostupné z: <https://msdn.microsoft.com/en-us/library/ff649690.aspx>
- [11] *What is Inversion of Control?* [online]. [cit. 2017-03-26]. Dostupné z: <http://stackoverflow.com/questions/3058/what-is-inversion-of-control>
- [12] *Nette Framework - Quick 'n' Comfortable Web Development in PHP* [online]. [cit. 2017-03-05]. Dostupné z: <https://nette.org/>

- [13] *TechnicalDebt* [online]. [cit. 2017-03-05]. Dostupné z: <https://martinfowler.com/bliki/TechnicalDebt.html>
- [14] *Git - free and open source distributed version control system* [online]. [cit. 2017-03-26]. Dostupné z: <https://git-scm.com/>
- [15] *Nextras Migrations* [online]. [cit. 2017-03-05]. Dostupné z: <https://nextras.org/migrations/docs/3.0/>
- [16] *JSON Web Tokens* [online]. [cit. 2017-03-05]. Dostupné z: <https://jwt.io/>
- [17] JOSEFSSON S. *The Base16, Base32, and Base64 Data Encodings* [online]. [cit. 2017-03-05]. Dostupné z: <https://tools.ietf.org/html/rfc4648>
- [18] *A JavaScript library for building user interfaces - React* [online]. [cit. 2017-03-05]. Dostupné z: <https://facebook.github.io/react/>
- [19] *Babel - JavaScript compiler* [online]. [cit. 2017-03-11]. Dostupné z: <https://babeljs.io/>
- [20] *Redux - predictable state container for JavaScript applications* [online]. [cit. 2017-03-11]. Dostupné z: <http://redux.js.org/>
- [21] *Thunk middleware for Redux* [online]. [cit. 2017-03-27]. Dostupné z: <https://github.com/gaearon/redux-thunk>
- [22] YAMAUCHI, Owen. *Hack and HHVM: programming productivity without breaking things*. ISBN 978-1-491-92087-9.
- [23] *curl - transfer a URL* [online]. [cit. 2017-03-14]. Dostupné z: <https://linux.die.net/man/1/curl>
- [24] *Introducing JSX* [online]. [cit. 2017-03-11]. Dostupné z: <https://facebook.github.io/react/docs/introducing-jsx.html>
- [25] *SQLite - self-contained, serverless, zero-configuration and transactional SQL database engine* [online]. [cit. 2017-03-14]. Dostupné z: <https://www.sqlite.org/about.html>
- [26] *GraphQL - A query language for your API* [online]. [cit. 2017-02-08]. Dostupné z: <http://graphql.org/>
- [27] *GraphiQL - A graphical interactive in-browser GraphQL IDE*. [online]. [cit. 2017-03-12]. Dostupné z: <https://github.com/graphql/graphiql>

- [28] *Docker - the open-source application container engine* [online]. [cit. 2017-03-19]. Dostupné z: <https://www.docker.com/>
- [29] *Aufs4 - advanced multi layered unification filesystem version 4.x* [online]. [cit. 2017-04-10]. Dostupné z: <http://aufs.sourceforge.net/>
- [30] *GraphQL Versioning* [online]. [cit. 2017-04-10]. Dostupné z: <http://graphql.org/learn/best-practices/#versioning>