FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

ZADÁNÍ BAKALÁŘSKÉ PRÁCE

Název:	Linky - Grafická knihovna
Student:	Jiří Košata
Vedoucí:	Ing. Jiří Chludil
Studijní program:	Informatika
Studijní obor:	Webové a softwarové inženýrství
Katedra:	Katedra softwarového inženýrství
Platnost zadání:	Do konce letního semestru 2019/20

Pokyny pro vypracování

Linky jsou světelnou instalací nacházející se na části fasády FEL ČVUT. Cílem práce je navrhnout a implementovat aplikační rozhraní rozšiřující stávající řešení o snadné zobrazování elementárních grafických objektů na virtuálním plátně, manipulaci s nimi a o další vizuální efekty a transformace.
1. Seznamte se s kompletní architekturou aktuálního ekosystému kolem projektu Linky.
2. Analyzujte problematiku vizualizace grafických útvarů v kontextu požadavků na knihovnu.
3. Pomocí metodik softwarového inženýrství navrhněte architekturu grafické knihovny s důrazem na škálovatelnost, rozšiřitelnost a spolehlivost.
4. Implementujte funkční prototyp grafické knihovny pomocí technologií Microsoft .NET se zohledněním nezávislosti na konkrétní platformě. Rozhraní vystavte pomocí REST API spolu s popisem dle specifikace OpenAPI pro snadnou konzumaci klienty.
5. Řešení dostatečně pokryjte vhodnými testy.
6. Zajistěte kvalitu celého systému díky technice Continues integration.

Seznam odborné literatury

Dodá vedoucí práce.

Ing. Michal Valenta, Ph.D.
vedoucí katedry

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
děkan

V Praze dne 2. ledna 2019

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Bakalářská práce

Linky – Grafická knihovna

Jiří Košata

Katedra softwarového inženýrství
Vedoucí práce: Ing. Jiří Chludil

14. května 2019

Poděkování

Chtěl bych poděkovat především vedoucímu práce Jiřímu Chludilovi za rady a konzultace při jejím vytváření. Děkuji také Robertu Berkovi za poskytnutí informací o současném stavu projektu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 14. května 2019 …………………

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2019 Jiří Košata. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí a nad rámec oprávnění uvedených v Prohlášení na předchozí straně, je nezbytný souhlas autora.

Odkaz na tuto práci

Košata, Jiří. *Linky – Grafická knihovna.* Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2019.

Abstrakt

Práce se zabývá analýzou současného stavu projektu světelné fasády Linky z pohledu technického zázemí, které je dostupné vývojářům k integraci s jejich aplikacemi či grafickými vizualizacemi. Analytická část práce popisuje převážně zvolené technologie pro realizaci rasterizace grafických objektů a další použité nástroje. Hlavním cílem praktické části je poskytnout takové rozhraní, které bude pochopitelné a umožní snadnou manipulaci s fyzickou fasádou pomocí virtuálního grafického plátna. S tím souvisí i návrh architektury aplikace se zaměřením na aspekt zpracování požadavků simulace vizuálních efektů v reálném čase. Závěrem práce jsou nastíněna další možná rozšíření tohoto systému o další grafické prvky či principy.

Klíčová slova Linky FEL ČVUT, aplikační rozhraní, rasterizace vektorové grafiky, škálovatelnost API, REST, OpenAPI, Docker, .NET Core, Redis, GitLab

Abstract

Main topic of this work is to analyse current state of light facade Linky from perspective of technical resources available to developers to integrate with their applications or visualizations. Analytic part is mostly describing technologies used to implement graphics objects conversion to raster form and other important tools. Main topic to implementation part is to provide such interface that will be easy understandable and will enable intuitive manipulation with physical facade using virtual canvas. With that comes architecture design that will handle requirement of real-time simulation of graphics effects. In conclusion are proposed extensions to this solution including new graphics effects and other features.

Keywords Linky FEL CTU, application interface, vector graphics rasterization, API scalability, REST, OpenAPI, Docker, .NET Core, Redis, GitLab

Obsah

Úvod **1**
Cíl práce . . . 2
Rozbor zadání . . . 2

1 Analýza **5**
1.1 Projekt Linky . . . 5
1.2 Současné řešení . . . 6
1.3 Použité technologie . . . 10
1.4 Nástroje . . . 22

2 Specifikace požadavků **29**
2.1 Funkční požadavky . . . 29
2.2 Nefunkční požadavky . . . 30

3 Návrh **33**
3.1 Model případů užití . . . 33
3.2 Doménový model . . . 35
3.3 Nasazení aplikace . . . 36
3.4 Proces interakce s administračním rozhraním . . . 38
3.5 Proces interakce s low level Linky API . . . 39
3.6 Pohyb dat v celém systému . . . 40

4 Implementace **41**
4.1 Vývojové prostředí . . . 41
4.2 Realizace jednotlivých komponent . . . 41
4.3 Nasazení aplikace . . . 53
4.4 Použití knihovny . . . 55

5 Testování **57**
5.1 Unit testy . . . 57

Závěr **59**

Seznam literatury **61**

A Seznam použitých zkratek **65**

B Ukázka Swagger UI **67**

C Ukázka OpenAPI specifikace **69**

D Obsah přiložené SD karty **71**

Seznam obrázků

1.1 Světelná fasáda Linky (převzato z [1]) 5
1.2 Schéma původního rozhraní (převzato a upraveno z [3]) 7
1.3 Část specifikace aktualizované verze rozhraní (převzato z [1]) . . . 9
1.4 Debugger aktualizované verze rozhraní (snímek webu z [1]) 10
1.5 Ukázka Swagger UI (převzato z [11]) 14

3.1 Diagram případů užití pro grafickou knihovnu 34
3.2 Datový model pro uživatele a plátno 35
3.3 Obecný diagram nasazení aplikace 37
3.4 Diagram aktivit pro administrační rozhraní 38
3.5 Diagram aktivit pro low level API 39
3.6 Diagram datového toku pro grafickou knihovnu 40

4.1 Vizuální podoba simulátoru 52

B.1 Swagger UI pro API grafické knihovny 68

Seznam tabulek

1.1 Srovnání výkonu grafických knihoven 21
1.2 Souhrn hodnocení rasterizačních knihoven 21

3.1 Kontrola splnění funkčních požadavků 35

Seznam ukázek kódu

1.1 Příklad Docker Compose definice 26
4.1 Docker Compose definice grafické knihovny 42
4.2 Definice controlleru v ASP.NET Core 43
4.3 Konfigurace autentizace a autorizace pro Swagger 45
4.4 Definice akce controlleru v ASP.NET Core 46
4.5 Konfigurace knihovny pro přístup k Redis databázi 47
4.6 Načtení objektu z Redis úložiště 47
4.7 Zpracování grafických dat snímku v jádře 48
4.8 Důležité hodnoty pro komunikaci s fasádou 49
4.9 Vykreslení elipsy pomocí knihovny SkiaSharp 50
4.10 Konfigurace SignalR hubu v rámci ASP.NET Core 51
4.11 Část implementace SignalR hubu pro simulátor 52
4.12 Část skriptu pro automatické sestavení obrazů 53
4.13 Část skriptu pro automatické nasazení aplikace 54
4.14 Část skriptu pro automatické testování aplikace 55
4.15 Část Dockerfile pro spuštění testů 55
5.1 Ukázka testu pomocí frameworku xUnit 58
C.1 Část OpenAPI specifikace pro API grafické knihovny 70

Úvod

Pokud chce vývojář nějakým způsobem provádět interakce s externími službami a nebo zařízeními, mezi které patří i světelná fasáda Linky, slouží mu k tomu již po několik let aplikační rozhraní založené na webových protokolech, což se stalo standardem pro tento druh integrace. Proto je potřeba se zaměřit na formu podobných rozhraní, kde se bohužel často uchyluje k implementaci pouze základních operací jako je například vytvoření prostředku, jeho přímá editace a nebo jeho smazání. To znamená, že veškerá logika aplikace musí být často vykonávána konzumentem daného rozhraní. S tím souvisí často i nesnadná dostupnost a nedostatečná dokumentace, protože u obecných operací se ani neočekává a to je problém.

Snahou práce je navrhnout a implementovat rozhraní, které zmíněné problémy eliminuje a obecně nastíní postupy, kterými se dají řešit. Nové rozhraní pro světelnou fasádu Linky nebude obsahovat pouze obecnou manipulaci s jejich fyzickou strukturou v podobě světelných bodů, ale rozšíří je i o další abstraktní struktury v podobě konkrétních grafických objektů. Konzument se tak může zaměřit na svůj hlavní účel a nemusí se zabývat operacemi, které jsou pro všechny konzumenty podobné. Tak, aby kdokoliv z řad studentů ČVUT, žáků středních škol a i různých umělců mohl vytvářet kreativní světelné instalace.

Hlavní motivací pro volbu tématu byly kritické nedostatky současného řešení, které není pro danou cílovou skupinu dostatečně intuitivní a je snahou většinu z nich odstranit a pro zbytek řešení alespoň navrhnout.

Práce začíná analýzou současného řešení, popisem všech pojmů důležitých pro pochopení souvislostí a výběrem vhodných technologií na kterých bude práce stavět. V návaznosti na současný stav bude navržena nová architektura řešící nedostatky se zohledněním rozšiřitelnosti a škálovatelnosti do budoucna. Na závěr bude obsažen souhrn implementace navržené architektury pomocí analyzovaných technologií a nástrojů včetně testování a nasazení výsledné aplikace.

Cíl práce

Cílem práce je podrobná analýza současného stavu projektu světelné fasády Linky na ČVUT FEL. Současně s tím analyzovat všechny dostupné přístupy k vizualizaci a především rasterizaci různorodých grafických útvarů v závislosti na vymezených funkčních požadavcích. Na základě analýzy bude navržena architektura nové komponenty pro grafickou knihovnu, která bude řešit případné nedostatky zjištěné při analýze současného stavu a podporovat nové požadované funkce. Celková architektura bude navržena s důrazem na škálovatelnost a snadnou rozšiřitelnost.

Výstupem práce bude implementovaný funkční prototyp grafické knihovny s ohledem na platformní nezávislost. Aplikace bude realizována v podobě veřejně dostupného REST API, které budou konzumovat jednotliví klienti. Bude obsahovat specifikaci dle standardu OpenAPI pro jeho snadnou čitelnost jak člověkem tak i strojem. Řešení bude pokryté vhodnými testy a kvalita kódu bude zajištěna díky technice Continuous Integration (CI), která zprostředkuje i automatizované nasazení do cílového prostředí.

Rozbor zadání

Práce se zabývá analýzou, návrhem a implementací komponenty, která se stane součástí projektu Linky. Bude ho doplňovat o snadno dostupné aplikační rozhraní umožnující intuitivní manipulaci s grafickým plátnem této světelné fasády. Výstupem práce bude nejen patřičná vývojová a uživatelská dokumentace, ale také implementovaná aplikace podrobená testům a nasazena do patřičného prostředí.

Následuje diskuse k jednotlivým bodům zadání práce.

Seznamte se s kompletní architekturou aktuálního ekosystému kolem projektu Linky

Součástí práce bude podrobná analýza všech komponent současného řešení včetně historie vývoje projektu jako celku.

Analyzujte problematiku vizualizace grafických útvarů v kontextu požadavků na knihovnu

Problematika rasterizace bude probírána v kapitole, kde budou zvažovány a porovnány již existující implementace ve srovnání s manuální implementací pro potřeby projektu.

Pomocí metodik softwarového inženýrství navrhněte architekturu grafické knihovny s důrazem na škálovatelnost, rozšiřitelnost a spolehlivost

Tento bod bude obsažen v kapitole o návrhu celkového řešení. Výstupem práce bude návrhová dokumentace, ve které bude kladen důraz na zde vyjmenované požadavky tak, aby se nenarazilo na problém až v implementační fázi.

Implementujte funkční prototyp grafické knihovny pomocí technologií Microsoft .NET se zohledněním nezávislosti na konkrétní platformě. Rozhraní vystavte pomocí REST API spolu s popisem dle specifikace OpenAPI pro snadnou konzumaci klienty

V implementační části se detailně popíše, jak byly jednotlivé navrhované části realizovány pomocí zadaných technologií. Popíší se i nutná omezení v rámci technologie Microsoft .NET s ohledem na platformní nezávislost. Shrnou se nutné náležitosti pro automatické generování OpenAPI specifikace pro výsledné API.

Řešení dostatečně pokryjte vhodnými testy

Implementace bude obsahovat primárně jednotkové testy pro jednotlivé komponenty. Dále bude doporučeno použití dalších druhů testů. V rámci porovnání přístupů k rasterizaci dojde i na testování výkonu.

Zajistěte kvalitu celého systému díky technice Continues integration. Pro nasazení využijte principů Continues delivery

Řešení bude obsahovat podporu pro GitLab–CI, díky kterému se zajistí kvalita kódu pomocí automatizovaného spouštění připravených testů. Stejně tak tento systém umožňuje i automatizované či poloautomatizované nasazení aplikace do zvoleného prostředí.

Kapitola 1

Analýza

1.1 Projekt Linky

Z pohledu kolemjdoucího se jedná o světelnou fasádu, která se jako součást fasády nachází na budově FEL ČVUT ze strany směrem k Vítěznému náměstí. Fyzicky je tvořena pěti sloupci s rovnoměrným rozestupem, takže vy výsledku tvoří skoro pravidelný čtverec. Každý sloupec je osazený světelnými body, které je možné programově ovládat a měnit jejich barvu dle RGB. Ve výsledku se jedná o velkoplošný LED panel s rozlišením 5×204 obrazových bodů.

Veřejně přístupná je také doprovodná webová aplikace [1], kde se nachází shrnutí a smysl projektu. Dále detailní informace jako historie a provozní řád. Má sloužit i jako místo, kde se uživatel dozví o akcích a možnostech interakce s fasádou.

Pro účely různých akcí existuje také mobilní aplikace a mnoho doprovodných menších aplikací, které umožňují uživatele zapojit v reálném čase, pokud se nachází poblíž fasády.

Technickým pohledem se jedná o kolekci několika menších projektů, které se snaží dohromady tvořit rozšiřitelný celek. Od veřejného webu, přes admi-

Obrázek 1.1: Světelná fasáda Linky (převzato z [1])

nistrační rozhraní jednotlivých aplikací této platformy až po *low level API*, které slouží k ovládání jednotlivých fyzických světelných bodů.

1.2 Současné řešení

1.2.1 Původní rozhraní fasády

Klíčovým prvkem je rozhraní fasády, které umožňuje aplikacím zobrazovat jednotlivé obrazové body, které se nachází ve sloupcích. Zde popisované rozhraní bylo součástí původní implementace a bylo provozované do konce roku 2018. Přístup je řešení pomocí REST API, kam lze na několik endpointů zasílat obrazová data ve speciálním formátu, kde jsou jednotlivé body jednoduše zakódované do RGB a sloučené za sebe. Tato data jsou buď odeslána pro jednotlivý bod, celý sloupec nebo celou fasádu. Obrazové body jsou namapovány přímo na fyzické světelné body na fasádě. Číslování a pozice jednotlivých bodů je zanesena na obrázku 1.2a. Jako barevný model je zvolen RGB s možným rozšířením na RGBW, protože každý bod má navíc bílé podsvícení, které lze regulovat odděleně. Kompletní specifikaci, ze které tato část práce čerpá, lze nalézt spolu se samotným API v podobě veřejně dostupné webové aplikace [2].

Přístup k API je řízen pomocí klíčů, které jsou vydávaný skrze administrační portál administrátorem. Všechny vydané klíče jsou platné a není možnost jak povolit v konkrétní čas pouze jeden klíč.

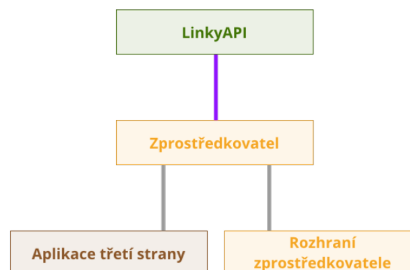
Jak je již zmíněno v práci řešící původní implementaci [3], rozhraní počítá s několika architekturami aplikací. Mezi základní patří ta, kde klient komunikuje přímo s rozhraním fasády. Tento způsob ale nevyhovuje scénářům, kde je v rámci jednoho sezení aplikace velké množství klientů a zároveň je snížena bezpečnost, protože autorizace probíhá přímo s klientským zařízením, které může být snadno napadnutelné. Proto existuje doporučený postup ve kterém klientská aplikace komunikuje s vlastní serverovou částí, která následně volá přímo rozhraní fasády – zodpovědnost je tedy koncentrována na jedno bezpečné místo. Poslední možností je existence nějakého prostředníka, který je také jediným bodem, který komunikuje přímo s fasádou, ale navíc obsahuje vlastní rozhraní, pomocí kterého poskytuje aplikacím třetích stran dodatečné funkce. Aplikace tak neimplementuje rozhraní fasády, ale pouze prostředníka. To je přesně ta varianta, ve které se bude nacházet naše grafická knihovna. Bude se jednat prostředníka, který bude poskytovat abstrakci nad low level rozhraním fasády. Tato architektura je zachycena na obrázku 1.2b.

1.2.1.1 Administrační portál

Pro zlepšení práce s API klíči k rozhraní fasády existuje portál, kde může administrátor jednoduše aplikace spravovat. V portálu existuje i role pro vývojáře, kde může mít přehled všech svých aplikací a klíčů.

	Column				
Column index	0	1	2	3	4
Light index	204	204	204	204	204
	203	203	203	203	203
	202	202	202	202	202
	•	•	•	•	•
	•	•	•	•	•
	•	•	•	•	•
	3	3	3	3	3
	2	2	2	2	2
	1	1	1	1	1
	0	0	0	0	0

(a) Číslování světelných bodů



(b) Jedna z architektur pro aplikace

Obrázek 1.2: Schéma původního rozhraní (převzato a upraveno z [3])

V současné době tento portál ale stojí paralelně vedle hlavního API pro fasádu a klíče na sebe nejsou napojené, takže nelze tyto funkce naplno využít. Nově proto vzniká *Linky core*, které bude mít na starosti kompletní administrativu kolem fasády včetně vydávání klíčů. Na nové API bude napojeno i nově vznikající administrační rozhraní. Kombinace těchto řešení by měla nedostatky současného portálu odstranit.

1.2.1.2 Hlavní nevýhody

Všechny aplikace jsou odkázané na manuální proces, kde jim administrátor musí vydat klíč pro testování a případně pro ostré zobrazování. Pokud již aplikace získá klíč, musí sama zjistit, kdy se má aktivovat. Po aktivaci použije získaný klíč k autentizaci s rozhraním fasády, kde může začít libovolně vykreslovat po omezenou dobu, kterou aplikace nezná ale pouze na jejím uplynutí začne dostávat chybové návratové kódy.

Každá aplikace musí implementovat logiku toho, jak požadovanou scénu zobrazit na takto specifickém zobrazovacím zařízení. V některých případech je to dostačující, nebo dokonce žádoucí, pokud aplikace opravdu využije pouze dostupné sloupce a nepotřebuje řešit například rozestup mezi sloupci. Pro většinu aplikací je to podle mého názoru komplikace, protože například pro pouhé vykreslení kružnice je potřeba provést spoustu úkonů včetně toho, jak vlastně ve 2D prostoru body kružnice dopočítat.

Dokumentace rozhraní je dostupná na stejné adrese jako API. Její obsah je ale bohužel formátován jako jednoduchý text a bez jakýchkoliv detailů. Není například jednoznačně popsané jaké rozměry fyzicky fasáda má a to že se během roku mění podle potřeby provozovatele. Zcela chybí ukázky komunikace a toho, jaké vstupy se očekávají a jaké výstupy mohou být vráceny. To lze například pozorovat na pouze stručně popsaném formátu pro barevné schéma. Některé funkce jsou označené jako budoucí a nejsou vůbec implementovány.

1.2.2 Aktualizované řešení fasády

Rozhraní pro ovládání fasády prošlo začátkem roku 2019 zásadní změnou a bylo kompletně přepsáno a přesunuto na jiné umístění. Struktura rozhraní je zcela odlišná a nekompatibilní s původním řešení, takže jakákoliv existující aplikace vyžaduje přepsání. Nová specifikace stejně jaké samotné API se nachází na adrese `https://linky.fel.cvut.cz/Api`. Je součástí webové aplikace, která slouží jako informační web o projektu jako celku a bližší informace lze najít přímo ve zdroji [1].

Jedná se o znatelně lépe formátovanou specifikaci než v původní podobě. Obsahuje seznam všech podporovaných funkcí včetně detailního popisu každé z nich s odkazy na další zdroje, kde lze zjistit dodatečné informace. Všechny vstupní parametry jsou taktéž zdokumentovány včetně uvedení datových typů a výchozích hodnot. Je k dispozici i popis všech možný odpovědí a návratových kódů s ukázkou jejich konkrétních podob. Její ukázka je na obrázku 1.3.

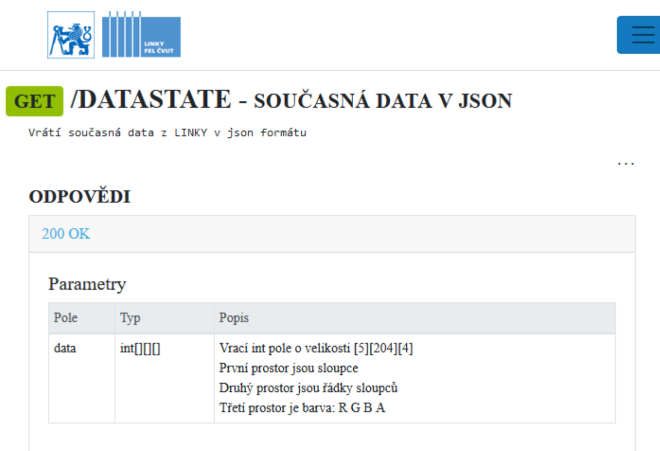
Oproti původní implementaci již obsahuje funkční podporu pro správu aplikačních klíčů, které již není potřeba řešit manuálně pomocí externí aplikace. Nyní je správa fronty aplikací a přidělování jednotlivých klíčů na jednom místě. Aplikace má možnost požádat o klíč a pokud je fasáda volná, dostane ihned nový klíč a může začít zobrazovat. V opačném případě je zařazená do fronty a získává klíč platný až od určitého časového okamžiku. Aplikace má také možnost si klíč prodloužit, pokud je fronta aktuálně prázdná.

Hlavní rozdíl je ve způsobu zobrazování. Nyní již se neposílá pouze jeden snímek nebo datová sada, která by se promítla na fasádě. Posílá se rovnou celá sekvence snímků, které se s určenou snímkovou frekvencí na fasádě zobrazí a aplikace nemusí řešit žádné časování, ale pouze vygeneruje snímky, které si přeje zobrazit. Se zavedením této abstrakce se ztratila možnost ovládat jednotlivé obrazové body a nebo logické sloupce a nejmenší jednotka je jeden snímek. Formát pro jednotlivé body stejně jako barevné schéma zůstalo zachováno.

Stejně jako v původní implementaci je i zde možné získat aktuální data a informace o stavu fasády. Pro strojové zpracování je k dispozici stav ve formátu JSON a pro vizuální zobrazování lze získat data v podobě obrázku ve formátu PNG. Jelikož je rozhraní zodpovědné i za frontu aplikací, je možné získat i informace o aktuálně běžící aplikaci. Lze zde zjisti i konfiguraci fasády, jako je časový rozsah, kdy je fasáda v provozu a nebo její přesné rozměry, které se v čase mohou měnit.

1.2.2.1 Prostředí pro vývojáře

Součástí je i uživatelský portál, kde si může kdokoliv založit účet s možností rozhraní otestovat. Testování je možné v odděleném prostředí, které je nazváno *Debugger*. Uživateli jsou vygenerovány přístupové údaje pro API a je možné ihned začít komunikovat, není v tomto případě řešeno nic jako fronta



Obrázek 1.3: Část specifikace aktualizované verze rozhraní (převzato z [1])

v produkčním prostředí. Tento účet ale nemá ihned přístup na fyzickou fasádu, ale výstup z API je směrován na speciální stránku, kde je vidět, co by se přibližně zobrazilo v případě napojení na fasádu.

K tomu aby měl uživatel šanci něco zobrazit v ostrém prostředí a nebo se zařadit do fronty je nutný zásah administrátora rozhraní, který musí přístup schválit. Jde tedy o klasický scénář, kde API si může vyzkoušet kdokoliv, ale v reálném prostředí ho mohou používat pouze schválené aplikace.

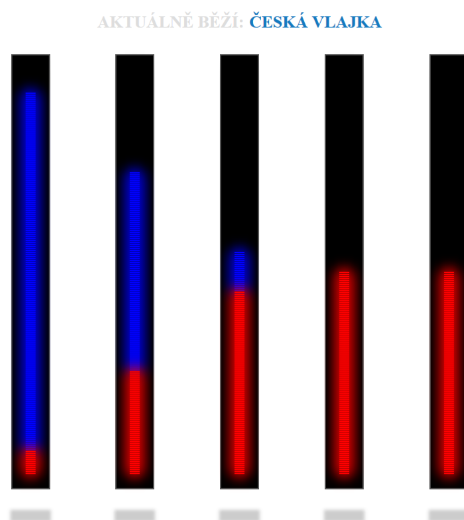
Prostředí, ve kterém je možné rozhraní testovat odpovídá podobě v jaké je zobrazen aktuální stav fasády na webu `https://linky.fel.cvut.cz/`. Jsou zde zobrazeny jednotlivé sloupce reflektující fyzické rozložení. Je naznačeno i oddělení individuálních světelných bodů. Vizualizace je v intervalu několika milisekund obnovována o zobrazuje tak stav blízký tomu aktuálně na fasádě, případně v testované aplikaci. Ukázka je na obrázku 1.4.

1.2.2.2 Hlavní nevýhody

I přesto, že tato podoba rozhraní obsahuje znatelně propracovanější podobu specifikace, stále jsou zde nedokonalosti bránící ve snadném použití. Pojmenování jednotlivých koncových bodů není konzistentní, ve specifikaci je vše pojmenované velkými písmeny. Reálně je ale vše dostupné pod adresami, které mají názvy v camel case konvenci a daná technologie je navíc case sensitive, takže je to pro vývojáře značně matoucí.

Dokumentace nikde nezmiňuje, kde rozhraní reálně běží, takže vývojář musí hádat, jaká je základní adresa pro API, aby ho mohl začít vůbec volat. Stejně tak není popsané, jestli je pro testování vytvořeno speciální prostředí a nebo lze vše provádět proti ostrému API.

Součástí specifikace není možnost interaktivně API z prohlížeče zavolat, což by odstranilo oba předchozí zmíněné problémy a zároveň udělalo rozhraní



Obrázek 1.4: Debugger aktualizované verze rozhraní (snímek webu z [1])

pro vývojáře atraktivní a snadno dosažitelné i bez potřeby nějakého externího nástroje. Celkově by tento přístup usnadnil testování aplikace před reálným nasazením a odpověděl na všechny otázky související s používáním API.

Další problém, který tato verze přináší oproti původní implementaci je dostupnost celé specifikace a dokumentace pouze v češtině. Takový přístup je nezvyklý a omezuje použití pouze na česky mluvící vývojáře, což je nyní pravděpodobně většina, ale do budoucna tomu tak být nemusí. Rozhraní ani na datové úrovni nepodporuje žádnou formu lokalizace a například texty popisující běžící aplikace jsou pouze v češtině.

1.3 Použité technologie

Následuje popis použitých technologií, které jsou zafixovány zadáním, pro pochopení jejich základních principů následně použitých v praktické části. Klíčová místa obsahující možnost volby jako například přístup k rasterizaci objektů zpracovávaných grafickou knihovnou obsahují detailní analýzu všech alternativ navrženou metodikou.

1.3.1 Framework a jazyk

Jakákoliv aplikace se může skládat z kombinace mnoha různých frameworků a programovacích jazyků – nicméně je výhodné si pro nějakou konkrétní část zafixovat pouze omezenou množinu z nich pro zjednodušení a urychlení vývoje. Součásti by měly být navržené tak, aby nebyl problém kombinovat více technologií nezávisle na sobě.

Pod pojmem framework se rozumí sada již vyřešených problémů zaměřená převážně na jeden konkrétní programovací jazyk. Může se skládat z dalších podpůrných programů, definicí základních rozhraní, různě rozsáhlých knihoven a nebo prostředků pro vynucení konkrétních návrhových vzorů. Jeho hlavním cílem je zjednodušení vývoje tak aby vývojář nemusel znovu vynalézat řešení na již známe problémy a mohl se zaměřit pouze na svou přidanou hodnotu.

1.3.1.1 .NET Core

Předchůdcem této technologie je .NET Framework, což je velice rozšířená a stále se rozvíjející platforma pro tvorbu aplikací v prostředí Microsoft Windows. Výhodou je možnost tvorby všech druhů aplikací pro tento operační systém a plná integrace s jeho základními komponentami a principy. Tyto vlastnosti ale mohou být v některých případech i hlavní nevýhodou, protože aplikace psané nativně pro .NET Framework nelze jednoduše spustit na ostatních platformách. Je to způsobeno buď tím, že aplikace nepočítá s jejich koncepty a nebo používá zrovna ty součásti, které nelze přenést na další operační systémy a také to, že se jedná o zcela uzavřený projekt bez dostupných zdrojových kódů. Tento problém se snaží řešit open-source projekt Mono, což je platformě nezávislá implementace vybraných komponent frameworku, které lze spouštět na operačních systémech jako je například linux, Android a nebo iOS. Framework jako takový nepředepisuje žádný programovací jazyk, ale slouží pouze k interpretaci pevně daného mezikódu nazvaného Common Intermediate Language (CIL), do něho se pak může překládat libovolný jazyk, pro který se vytvoří potřebný překladač. Souhrnné informace o frameworku byly čerpány ze zdroje [4].

Nedostatky potřebné pro aktuální moderní aplikace řeší .NET Core. Jedná se o plně multiplatformní a open-source alternativu pro .NET Framework. Část rozhraní je převzatých z plného frameworku, ale obecně není zpětně kompatibilní a v těchto rozhraních mohou být nekompatibilní změny. Stávající aplikace tedy nelze ve většině případů pouze aktualizovat na tuto platformu, ale je nutný částečný přepis pro funkci s novým rozhraním stejně jako aktualizace všech knihoven třetích stran – není to tedy doporučená cesta pro existující aplikace, ale pro tvorbu nových.

Klíčové je kromě podpory více platforem také to, že došlo ke kompletní reimplementaci celého běhového prostředí a většiny základních knihoven, což se velmi pozitivně podepsalo na výkonu aplikací. Jedná se tedy o ideální platformu pro aplikace, kde je vyžadován maximální možný výkon na úrovni vysokoúrovňových jazyků. Samotné běhové prostředí (CoreCLR) je implementováno v jazyce C++. Sada základních knihoven (CoreFX) je již implementována plně v jazyce C#. Je možné zde vytvářet omezené množství aplikací, pro verzi 2.0 se jedná o ASP.NET Core webové aplikace, konzolové aplikace, knihovny a univerzální aplikace pro Windows 10 (UWP). Podpora programo-

vacích jazyků není nikterak omezena a jedná se o plnohodnotný interpret pro CIL mezikód stejně jako u plného frameworku. Tento odstavec se zakládá na informacích ze zdroje [5]

1.3.1.2 C#

Vysokoúrovňový, staticky typovaný programovací jazyk s podporou více paradigmat určený k použití řešení velmi obecných problémů. Vyvinutý byl společností Microsoft a jeho hlavním návrhářem byl Anders Hejlsberg. Inspirací pro něj byly převážně jazyky Java a C++. Informace v tomto odstavci jsou převzaty ze zdroje [6].

Jazyk je kompilovaný do CIL mezikódu, takže je možné ho interpretovat na jakékoliv implementaci CLR běhového prostředí kam patří .NET Framework a nebo .NET Core. Ke kompilaci je možné využít open-source kompilátor Roslyn, který je napsaný kompletně v jazyce C#, což umožňuje jeho snadnou rozšiřitelnost bez potřeby jakýchkoliv dalších nástrojů. Právě kombinace jazyka C# a prostředí .NET Core bude použita k tvorbě převážné většiny praktické části práce, jak je určeno zadáním. Tento odstavec se zakládá na zdroji [5].

Pro platformu .NET se jedná o zde nejpoužívanější programovací jazyk, jak je patrné i z průzkumu [7]. Mnoho využití má však i jinde, například pro tvorbu nativních mobilních aplikací díky platformě Xamarin (Společnost, která aktuálně stojí za vývojem projektu Mono). Je hlavním skriptovacím jazykem ve známém herním enginu Unity.

1.3.2 Databáze

Službeb databáze je v rámci toho projektu nutné především k persistenci dat. Informace o aktuálním stavu musí být zachovány i po ukončení jakékoliv komponenty, cože je v distribuovaném prostředí naprosto běžný jev, proto uchovávání pouze v operační paměti nepřichází v úvahu. Stejně tak z důvodu paralelního přístupu ke stejným datům není vhodné ukládání pouze na diskové pole, kde zároveň chybí možnost pro snadné škálování aplikace.

1.3.2.1 Redis

Redis je open-source implementace distribuované NoSQL databáze. Hlavní myšlenka je založená na principu úložiště, které se celé nachází v operační paměti pro maximalizaci odezvy operací s daty. Data jsou průběžné dávkově ukládána na běžné diskové úložiště pro zachování dat při případném selhání celého systému. Základní struktura dat se skládá z jednoduchého modelu klíč–hodnota, kde pod libovolným klíčem může být libovolný podporovaný datový typ.

Základním datovým typem je textový řetězec, do kterého lze uložit jakákoliv data o maximální velikosti 512 MB. Kromě toho jsou k dispozici i další

více abstraktní datové typy, ale vždy slouží k ukládání dat v podobě textových řetězců. Patří sem klasický seznam hodnot, množina hodnot bez opakování, hash tabulka (vnořené úložiště typu klíč–hodnota) a další specifické struktury. Mimo to je možné tuto databázi použít i jako jednoduchou implementaci fronty zpráv, ať už jde o klasickou frontu, která se zpracovává postupně a nebo mechanizmus pub–sub, kdy jsou zprávy vždy posílány přímo konkrétním poslouchajícím konzumentům.

Na úrovni instance databáze je podporována replikace dat mezi více uzlů. Existuje možnost je nakonfigurovat do režimu master/slave, kde jsou data jak replikována, tak i rozprostřena mezi více uzlů s možností přesměrování provozu při výpadku některého z nich. Komunikace na úrovni síťové vrstvy je realizována pomocí udržovaného TCP spojení, po kterém se posílají jednotlivé příkazy definované vlastním protokolem RESP. Ten si zakládá na tom, aby byl snadno implementovatelný klienty, rychle zpracovatelný a zároveň čitelný člověkem. Komunikace se řídí modelem požadavek–odpověď, kde server běžící výhradně pouze na jednom vlákně zpracovává požadavky klientů jeden po druhém a ihned jim posílá zpět odpověď. Informace jsou převzaty z oficiální dokumentace reprezentované pramenem [8].

1.3.3 Podpůrné frameworky a knihovny

Kromě základních prvků je nutné použít i řadu různých dalších podpůrných nástrojů. Jsou neméně důležité, takže ke všem z nich následuje stručný popis, aby bylo jasné i jejich použití v kontextu zbytku práce.

1.3.3.1 OpenAPI a Swagger

OpenAPI je specifikace [9] sloužící k popisu API, které jsou výhradně typu REST. Výsledný soubor obsahuje veškeré klíčové vlastnosti popisovaného API, jako jsou:

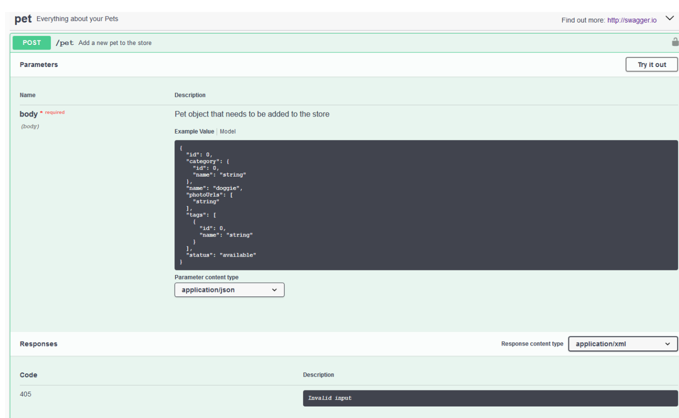
- informace o každém dostupném endpointu včetně všech HTTP operací, které podporuje (GET, POST, …)
- schéma všech parametrů operací včetně vstupních a výstupních dat
- všechny autentizační metody, které lze v rámci API použít
- dodatečné informace o rozhraní jako celku (kontakt, licence, podmínky použití, …)

Specifikace rozhraní může být buď ve formátu YAML a nebo JSON. V obou případech se jedná o popis snadno zpracovatelný strojem a čitelný člověkem bez potřeby dalších nástrojů. Zároveň je zcela nezávislé na programovacím jazyce a technologii, pomocí kterých je dané API implementované. Naopak pro hodně jazyků existuje podpora, jak pro v něm vyvíjené API automaticky

generovat tuto specifikaci na základě zdrojových kódů a případně dalších speciálně formátovaných komentářů. Aktuální verze specifikace popisovaná v této sekci je OpenAPI 3.0 [9].

Swagger je sada nástrojů, které usnadňují použití OpenAPI specifikace. Původně i ona byla součástí této sady pod názvem Swagger Specification, ale došlo k jejímu oddělení do samostatného projektu, do kterého nyní přispívá několik společností (Google, IBM, Microsoft, ...), jak uvádí zdroj [10].

Hojně využívaným nástrojem je Swagger UI [11], který slouží ke zpřístupnění OpenAPI specifikace interaktivní formou v podobě jednoduché webové aplikace. Ta umožňuje zobrazení všech specifikovaných prvků a i jejich používání v rámci ladění a nebo snaze nějakému rozhraní více porozumět. Uživatel tak může volat jednotlivé endpointy z pohodlí svého webového prohlížeče pomocí formuláře vygenerovaného ze specifikace a ihned vidí odpověď serveru.



Obrázek 1.5: Ukázka Swagger UI (převzato z [11])

Dalším užitečným nástrojem je Swagger Codegen [12], ten umožňuje automatické generování nativních klientských knihoven určených ke konzumaci daného API popsaného specifikací. Podporováno je široké množství jazyků, takže je možné vytvořit knihovnu obsahující pro jazyk specifické struktury odpovídající přesně datovému modelu API. Současně je vytvořena i kostra pro volání všech existujících endpointů. Druhým využitím tohoto nástroje je generování ukázkové implementace daného API – tzv. stubbing. Jedná se o funkční aplikaci, která obsahuje všechny specifikované prvky, ale je schopna vracet pouze ukázkové výstupy pro konkrétní vstupy, které je také možné uvádět v OpenAPI specifikaci. Není zde tedy obsažena žádná logika a takové API není ničeho schopné, jedná se ale o ideální nástroj k prototypování aplikačních rozhraní předtím, než dojde k jejich kompletní implementaci.

1.3.3.2 SignalR

Jde o volitelnou komponentu pro technologii ASP.NET, kterou lze doinstalovat v podobě knihovny. Cílem je umožnit serverové aplikaci asynchronně posílat zprávy do klientské části aplikace. Komunikace je zaměřena především na klienty v podobě webové JavasSript aplikace, kde lze se serverem díky HTML5 navázat WebSocket spojení pro posílání zpráv. Proto také existuje SignalR klientská knihovna pro JavaScript a TypeScript. Klient ale nemusí nutně běžet na webu, ale může se jednat i o jiný typ aplikace – nativní knihovna existuje například pro jakékoliv .NET Framework aplikace.

Model komunikace nemusí být nutně založený na neustálém WebSocket spojení a pokud jedna ze stran daný typ komunikace nepodporuje, dojde k přepnutí na jiný způsob (například polling), takže je zajištěna i zpětná podpora se staršími webovými prohlížeči už na úrovni této komponenty a vývojář problém vůbec nemusí řešit. Kromě standardního posílání zpráv je zde umožněno i obousměrné volání funkcí (RPC) ze serveru na klientovi. Nechybí ani podpora pro systémové události (připojení, odpojení), vytváření skupin klientů a nebo podpora autorizace. Sekce byla zpracována na základě zdroje [13].

1.3.4 Volba rasterizační knihovny

Jedna z komponent výsledné aplikace bude zodpovědná za převod grafických objektů, které budou popsané pouze určitým seznamem atributům a ty je potřeba převést do rastrové podoby. Problematika by se dala přirovnat k převodu vektorové grafiky, kde je pouze matematický popis objektů do výsledné podoby bitmapy, která se naopak skládá pouze z matice obrazových bodů, které obsahují pouze barvu bez jakéhokoliv dalšího kontextu.

Jedním z přístupů je celý proces realizovat vlastní implementací, jak uvádí například zdroj [14, Kapitola 3]. Tento způsob není implementačně příliš atraktivní, jelikož se jedná o již známé algoritmy, které lze použít v podobě již existujících knihoven. Další sekce se tedy zaměří právě na tento přístup, který je vhodným kandidátem na použití v tomto specifickém případě, kde plátno dostatečně malé na to, aby šlo využít výhod vlastní implementace. A bude podroben patřičnému testu.

Rozbor technologií bude zaměřený na technologii .NET, která obsahuje dostatečnou škálu přístupů k problematice a zároveň poskytuje i prvky moderního vývoje a je multiplatformní. Musíme skupinu zvažovaných technologií porovnat a využít pouze tu co z porovnání vyjde nejlépe.

1.3.4.1 Hodnotící kritéria

Aby bylo možné jednotlivé možnosti měřitelně porovnat, je nutné zvolit konkrétní kritéria a jejich klíčové metriky. U každého z nich bude popsané, za co lze získat kolik bodů, maximum pro kritérium jako celek je 5. Na konci hod-

nocení bude uvedena souhrnná tabulka s body a případně dalšími měřeními na základě které bude vyvozen finální závěr.

Dokumentace a komunita

Důležitým kritériem při prvotní implementaci a následném rozšiřování je kvalitní dokumentace v některé z existujících podob.

Možné získané body:

- **1 b** – existence dokumentace a dostupnost online
- **2 b** – ukázky kódu s použitím
- **1 b** – existence oficiálního diskuzního fóra
- **1 b** – aktivní komunita na github.com/stackoverflow.com

Srozumitelnost API

Při integraci knihovny a případnému proniknutí do problematiky někomu dalšímu pomůže intuitivnost poskytovaného API.

Možné získané body:

- **1 b** – dodržování konvencí daného jazyka (pojmenování, paradigma, …)
- **2 b** – knihovna obsahuje inline dokumentaci (intellisense)
- **1 b** – dlouhé konstrukce nahrazeny extension metodami

Adaptace na nové požadavky

Aby bylo možné aplikaci v budoucnu rozšířit o další funkce a případně další grafické objekty, je nutné ověřit že je daná knihovna dostatečně univerzální a nezaměřuje se pouze na omezenou podmnožinu požadovaných funkcí.

Možné získané body:

- **1 b** – abstrakce nad vykreslovanými obrazci
- **2 b** – možno přidat vlastní útvary
- **1 b** – abstrakce nad plátnem (možnost vykreslovat v paměti)

Snadnost integrace

Pokud se daná knihovna zvolí je důležité to, jak obtížné ji bude do projektu jako celku zaintegrovat a zda to bude mít nějaké negativní důsledky.

Možné získané body:

- **2 b** – knihovna je multiplatformní

- **1 b** – distribuce skrze balíčkovací systém (NuGet)
- **1 b** – žádné další závislosti na instalovaný software

Výkon

Pro konkrétní aplikaci není výkon tak důležitý jako pro jiné graficky náročné aplikace, které běží přímo na grafické kartě uživatele a kde se očekává minimální odezva. Nicméně i tak je důležité, aby vykreslení i komplikovanější scény na našem omezeném plátně nezabralo mnoho času a prostředků, protože musí dojít k jejímu dalšímu zpracování odeslání po síti na fyzické zobrazovací zařízení, kde teprve dojde k zobrazení.

Výkon je měřen na testovací scéně v prostředí, kde je vygenerována náhodná scéna s daným počtem objektů a sleduje se průměrný čas vykreslení jednoho takového snímku. Prostředí je pro všechny zvažované technologie stejné, pouze se mění daná implementace elementárních vykreslovací operací.

Bude provedena sada testů a na základě relativního umístění rozděleny body:

- **1 − 4 b** – podle průměrného pořadí (nejhorší má 1 b a nejlepší 4 b)
- **1 b** – řádově rychlejší než ostatní

1.3.4.2 Hodnocení nativní implementace

Dokumentace a komunita

Protože se jedná přímo o součást .NET Framework, tak není stejně jako s jakoukoliv jeho součástí problém najít online dokumentaci. V tomto případě je dostupná na adrese https://docs.microsoft.com, takže za to získává **1 bod**. Co se týče ukázek kódu, tak ty u většiny hlavních tříd buď kompletně chybí a nebo nejsou dostatečně komplexní. Chybí také složitější scénáře použití na reálných příkladech. Zde tedy získává pouze **1 bod**. Fórum zaměřené přímo pro tuto součást bohužel neexistuje, takže zde nezískává žádné body. Na webu stackoverflow.com se vyskytuje přibližně 1000 otázek obsahující tag souvisejícím se `System.Drawing` a existuje i aktivita za tento rok, takže zde získává **1 bod**. V součtu tedy za toto kritérium celkem **3 body**.

Srozumitelnost API

Stejně jako v předchozím bodě zde není problém s dodržováním konvencí, protože se jedná o součást vytvořenou přímo tvůrci .NET Framework, takže zde získává **1 bod**. Inline dokumentace je na stejně dobré úrovni, jako ostatní komponenty frameworku, takže získává **2 body**. Neobsahuje žádné zjednodušující metody a vše je nutné psát klasickým stylem, takže zde žádné body. V součtu tedy za toto kritérium celkem **3 body**.

Adaptace na nové požadavky

Není zde žádná abstrakce nad vykreslovanými obrazci a vše je nutné uvádět explicitně, zde tedy nezískává žádné body. Není možné přidat vlastní útvary a rozhraní je poměrně uzavřené. Jediná možnost rozšíření je vykreslení útvaru jako křivky popsané jednotlivými body. Takže za tuto část získává pouze **1 bod**. Obsahuje přímo podporu pro vykreslení pouze v paměti s výstupem do bitmapy, ve které lze efektivně přistupovat k jednotlivým bodům. Takže zde získává **1 bod**. V součtu tedy za toto kritérium celkem **2 body**.

Snadnost integrace

Rozhraní je součástí platformy .NET Core od veze 2.1, takže se jedná o plně multiplatformní komponentu, kterou lze bez problémů použít na všech podporovaných platformách. Zde získává **2 body**. Komponenta je v .NET Core dostupná pomocí NuGet balíčku `System.Drawing.Common`, takže získává **1 bod**. Tento balíček je závislý pouze na dalších systémových komponentách distribuovaných také skrze NuGet, takže získává opět **1 bod**. V součtu tedy za toto kritérium celkem **4 body**.

Výkon

Výkonově se v rámci navrženého testu jedná o druhou nejrychlejší metodu vykreslování (viz tabulka 1.1), získává tedy celkem **3 body**.

1.3.4.3 Hodnocení knihovny ImageSharp

Dokumentace a komunita

Veškeré informace o projektu jsou soustředěny na platformu github.com a nechybí zde ani odkaz na online dokumentaci generovanou ze zdrojového kódu dostupnou na drese `https://sixlabors.github.io/docs/`. Takže zde získává **1 bod**. Součástí online dokumentace jsou i ukázky použití knihovny včetně odkazu na repositář s několika vzorovými aplikacemi, ty by ale mohly být lépe okomentovány a doplněny o ilustrační příklady již přímo na úrovni dokumentace, získává zde tedy pouze **1 bod**. Přímo na stránce je možnost diskutovat a komunikovat s vývojáři pomocí platformy Gitter a ze strany vývojářů existují reakce na vytvářené issues v rámci platformy github.com, takže zde získává **1 bod**. V kombinaci s předchozím bodem a s aktivními 44 otázkami na webu stackoverflow.com získává za tuto položku **1 bod**. V součtu tedy za celé kritérium získává **4 body**.

Srozumitelnost API

Knihovna používá nestandardní přístup v rámci ostatních i systémových knihoven – většina operací je řešena pomocí funkcionálního paradigmatu a nemusí

být intuitivní jejich použití. V důsledku malé sady ukázkových příkladů může snadno vývojář vytvářet neoptimální kód, takže zde žádné body nezískává. Inline dokumentace je v tomto případě na klasické úrovni a všechny třídy obsahují stručný popis a nic není vynecháno, jedná se ale o totožnou dokumentaci jako je ta webová, takže získává pouze **1 bod**. Knihovna je navržena v duchu funkcionálního paradigmatu, takže nehrozí zbytečně dlouhé konstrukce a většiny akcí je dosaženo pomocí extension metod, takže získává **1 bod**. V součtu tedy za celé kritérium obdrží **2 body**.

Adaptace na nové požadavky

Všechny obrazce v rámci knihovny jsou reprezentovány jako polygony, což zajišťuje podporu pro libovolně složité vlastní útvary, které lze takto popsat a následně s nimi provádět standardní operace. Takže za abstrakci získává **1 bod.** Jak je již zmíněno v předchozím bodu, veškeré operace jsou plně abstraktní a není problém si přidat vlastní – stačí k tomu pouze implementovat rozhraní, které s objektem umožní základní operace tak aby mohl interagovat s ostatními prvky. Zde získává maximální **2 body**. Existuje podpora pro převod aktuální podoby plátna do fyzické bitmapy a lze přistupovat i k jednotlivým virtuálním obrazovým bodům, takže získává **1 bod**. V součtu tedy za celé kritérium obdrží **4 body**.

Snadnost integrace

Knihovna si klade za cíl být plně multiplatformní na rozdíl od původní implementace, která je součástí .NET Framerork. Veškerý kód je psaný dle specifikace .NET Standard 1.3, takže je multiplatformní podpora zajištěna a získává **2 body**. Distribuce je realizována pomocí oficiálního NuGet balíčku skrze oficiální galerii a za to získává **1 bod**. Balíček s knihovnou nemá žádné další systémové závislosti kromě dalších vlastních podpůrných balíčků, takže získává **1 bod**. V součtu tedy za celé kritérium obdrží **4 body**.

Výkon

Výkonově se v rámci navrženého testu jedná o nejpomalejší metodu vykreslování (viz tabulka 1.1) i přes snahu eliminovat problém s případným nesprávným použitím. S množstvím vykreslovaných objektů výkon rapidně klesal zřejmě díky příliš vysoké abstrakci, která není připravena na takový typ použití. Získává tedy pouze **1 bod**.

1.3.4.4 Hodnocení knihovny SkiaSharp

Dokumentace a komunita

Jako rozcestník implementace této grafické knihovny pro platformu .NET slouží github.com, kde je mimo jiné i odkaz na online dokumentaci. Jelikož je

SkiaShaprp součásti projektu Mono, které nyní patří pod společnost Microsoft, je dokumentace kompletně součástí https://docs.microsoft.com, za to tedy získává **1 bod**. Dokumentace obsahuje u klíčových tříd velké množství ukázek toho, jak je správně používat. Součástí repositáře se zdrojovými kódy knihovny je k dispozici široká sada ukázek použití napříč všemi podporovanými platformami. Navíc na webu `https://skia.org` je celá řada ukázek použití knihovny co se týká grafických možností, které si lze i interaktivně vyzkoušet přímo na webu a poté zdrojové kódy s drobnou úpravou použít i v této konkrétní implementaci. Za to získává celé **2 body**. Oficiálně podporovaný komunikační kanál je platforma Gitter a problémy či návrhy je možné řešit skrze issues v rámci platformy github.com, zde tedy získává **1 bod**. Na webu stackoverflow.com existuje přes 300 otázek související s knihovnou a její implementací, takže získává **1 bod**. V součtu tedy za celé kritérium obdrží **5 bodů**.

Srozumitelnost API

Knihovna používá klasický procedurální přístupný podobný nativní implementaci, takže nebudí žádnou nejistotu ohledně jejího použití, takže získává **1 bod**. Inline dokumentace obsahuje základní popis všech použitých tříd a datových typů, zde tedy získává **1 bod**. Knihovna neobsahuje žádnou sadu předem připravených funkcí, které by zjednodušovaly používání knihovny a vše je tedy nutné psát klasickým stylem, takže zde nezískává žádné body. V součtu tedy za celé kritérium obdrží **2 body**.

Adaptace na nové požadavky

Základní sada grafických útvarů v rámci knihovny neobsahuje žádnou zastřešující abstrakci a jedná se samostatně stojící třídy, takže zde nezískává žádné body. Pokud jde o rozšiřitelnost vlastními grafickými útvary, je zde k dispozici možnost použít polygon sestávající se z několika bodů a nebo kompletně abstraktní křivku, kterou je možné též klasicky vykreslit a provádět nad ní díky společnému rozhraní základní operace. Za rozšiřitelnost tedy získává celé **2 body**. Existuje možnost aktuální stav plátna převést do bitmapy a tu následně na surová data. Zároveň je možné přistupovat k jednotlivým obrazovým bodům pomocí bitmapy, která nad kterou se nachází virtuální plátno, takže zde získává **1 bod**. V součtu tedy za celé kritérium obdrží **3 body**.

Snadnost integrace

Jelikož je knihovna od samého počátku součástí projektu Mono, nemá žádné problémy s podporou na více platformách. Dokonce zde značně přesahuje samotnou platformu .NET a má podporu i na mobilních platformách jako je Android a nebo iOS. Takže za multiplatformnost získává celé **2 body**. Distribuce je zajištěna výhradně skrze oficiální NuGet balíček dostupný ve standardní galerii, takže získává **1 bod**. Co se týká dalších závislostí, tak záleží

na platformně, kde se knihovna používá. V rámci OS Windows není potřeba instalovat žádné další závislosti, ale například u systémů Linuxového typu je potřeba doinstalovat další závislosti a dodatečné balíčky, které navíc ani nejsou nikde zmíněny a je nutné poměrně složitě hledat, takže zde nezískává žádné body. V součtu tedy za celé kritérium obdrží **3 body**.

Výkon

Výkonově se v rámci navrženého testu jedná o nejrychlejší metodu vykreslování (viz tabulka 1.1), kde nedochází k problémům ani při vysokém počtu grafických objektů. Získává tedy **4 body.**

1.3.4.5 Shrnutí hodnocení

Součástí porovnávací metodiky byl i test výkonu, jehož výsledky jsou již zahrnuty v hodnocení jednotlivých kandidátů. Shrnující naměřená data jsou obsažena spolu s dalšími relevantními informacemi v tabulce 1.1.

Knihovna	1000 objektů	Průměrné FPS
Nativní	12,21 ms	82
ImageSharp	4236 ms	0.23
SkiaSharp	**6,72 ms**	**148**

Tabulka 1.1: Srovnání výkonu grafických knihoven

Celkové hodnocení vytvořené ze součtu všech kritérií pro každého kandidáta je shrnuté v tabulce 1.2. U všech kritérií je zvýrazněný kandidát s nejvyšším skóre a v posledním řádku jsou agregovány součty napříč všemi kritérii. Z tabulky je tedy patrné, že podle navržené metodiky těsně vyhrála knihovna SkiaSharp a proto bude použita dále v implementační části.

Kritérium	Nativní	ImageSharp	SkiaSharp
Dokumentace a komunita	3	4	**5**
Srozumitelnost API	**3**	2	2
Adaptace na nové požadavky	2	**4**	3
Snadnost integrace	**4**	**4**	3
Výkon	3	1	**4**
Celkem	15	15	**17**

Tabulka 1.2: Souhrn hodnocení rasterizačních knihoven

Jako mezivýsledek porovnání knihoven lze pozorovat, že nativní implementace je téměř ideální, ale nedosahuje dostatečného výkonu. Naopak knihovna ImageSharp obsahuje nadměrnou míru abstrakce, takže i přes efektivnější implementaci pro menší počet objektů není vhodná pro naše použití. Vítězná

knihovna SkiaSharp kombinuje oba tyto přístupy, kdy je přítomna jak efektivní implementace, tak přiměřená míra abstrakce, která nevytváří úzké hrdlo.

1.4 Nástroje

Při realizaci práce je nutné použít také patřičné nástroje a zde si některé klíčové specifikujeme a popíšeme. Jedná se především o vývojové nástroje použité v implementační části práce, které velké množství procesů zjednodušují a obsahují podporu takové procesy vůbec realizovat ideálně v automatizované formě.

Výběr nástrojů je založený čistě na osobní preferenci a zkušenostech autora s jejich aplikací na řešené problémy. Jsou zde popsány hlavní zvolené nástroje se zmíněním existujících alternativ spolu s naznačením hlavních rozdílů mezi nimi. Volba konkrétních nástrojů není něco, co by ovlivňovalo projekt jako takový a často lze bez větších problémů, na rozdíl od volby technologií, přejít od jednoho na druhý.

1.4.1 Visual Studio

Visual Studio je vývojový nástroj (IDE) vytvořený firmou Microsoft. Cílí převážně na tvorbu různých druhů aplikací především pro platformu Microsoft Windows. Od příchodu platformy .NET Core, která je multiplatformní se nyní jedná i o nástroj pro tvorbu široké škály multiplatformních aplikací. Prostředí není omezené pouze na jeden programovací jazyk, ale podporuje jich v základu rovnou několik (C, C++, C#, VB .NET, …) a další je možné přidat formou různých rozšíření (Python, Ruby, Java, …). Provozování nástroje je zatím omezené pouze na operační systém Windows, to ale neomezuje výsledné aplikace.

Každá verze má vždy dostupnou variantu zdarma. Do verze 2015 se jednalo o edice s názvem *Express*, ty umožňovaly základní použití s omezením pokročilých funkcí a vývojem pouze pro nekomerční účely. Od verze 2015 existuje edice s názvem *Community*, která je zcela zdarma a obsahuje téměř všechny funkce jako vyšší placené edice s omezením pouze těch funkcí, které pro jednotlivce a menší týmy nejsou klíčové a již lze využít i pro komerční účely s omezením pouze na velikost týmu a či organizace.

Nástroj jako takový v sobě obsahuje celou řadu funkcí, které jsou praktické nejen pro samotnou editaci zdrojového kódu, ale i pro integraci s dalšími důležitými nástroji. Tyto funkce je možné rozdělit do několika hlavních kategorií:

- editace kódu s podporou IntelliSense (autocomplete pro zdrojový kód),
- úpravy kódu a refaktorizace,
- integrovaný debugger a profiler,

- grafický designer (GUI, datové modely, class diagramy, …),
- integrace nástrojů (Git, Docker, Microsoft Azure, …).

Veškeré informace a vlastnosti nástroje pochází z oficiální dokumentace zastoupené pramenem [15].

Volba tohoto nástroje jako hlavního pro realizaci implementační části byla poměrně jednoznačná, jelikož pro vývoj na platformě .NET neexistuje žádná srovnatelná konkurence, která by pokryla stejnou funkcionalitu v odpovídající kvalitě. Podle průzkumu [7] mezi vývojáři se jedná o nejoblíbenější IDE za rok 2018. Jako alternativa se nabízí editace kódu v některém z běžných textových editorů – tam ale přicházíme o integraci s dalšími nástroji. Na této úrovni je s omezením na .NET Core možné použít multiplatformní textový editor *Visual Studio Code*. Jako konkurenční IDE je dobré zmínit *JetBrains Rider*. Je to multiplatformní nástroj pro tvorbu aplikací na platformě .NET, ale je omezený existencí pouze komerční placené edice a jednotlivé integrované nástroje nejsou na takové úrovni jako je tomu u Visual Studio, jedná se ale o velice mocný nástroj v oblasti refaktorizace.

1.4.2 GitLab

GitLab je open-source software který podporuje celý cyklus při vývoji aplikace. Jedná se o poměrně komplexní nástroj, který je k dispozici pomocí víceuživatelského webového rozhraní se širokou možností nastavení práv. Je možné ho používat zdarma v podobě hostované verze na `https://gitlab.com`, kde je k dispozici vždy aktuální verze. Zároveň je možné si nástroj nasadit kdekoliv jinde na vlastní infrastruktuře – je to velmi častý případ použití ve firemním prostředí pro interní projekty. Implementační část této práce bude využívat možnosti studentům dostupné a fakultou spravované GitLab instance nacházející se na adrese `https://gitlab.fit.cvut.cz`.

Všechny komponenty, které jsou volitelně dostupné (pokud je správce instance nakonfiguroval) pro každý vytvořený projekt je možné shrnout do následujících kategorií:

- správa Git repozitářů,
- dokumentace a wiki,
- issue tracker,
- nástroje pro CI/CD.

1.4.2.1 Správa kódu

Hlavní součástí je možnost hostování Git repositářů, které obsahují kód aplikace a umožňují jeho verzování. Jedná se standardní Git repozitář, který lze

najít kdekoliv jinde a je plně integrovaný se systémem uživatelů a práv v rámci GitLab instance. Existuje role *Reporter*, která má pouze přístup ke čtení zdrojového kódu a slouží především k používání společně s issue tracing nástrojem. Kdokoliv s rolí *Developer* má možnost měnit zdrojový kód v rámci všech větví.

Nad jednotlivými větvemi lze nastavovat další politiky a omezení. Hlavní větev je tak například možné chránit proti přímým změnám a vynutit tak *merge requesty*, které mohou obsahovat další sadu pravidel a nebo dokonce vyžadovat schválení další osobou. Často se například využívá možnosti, kdy do hlavní větve se dostane pouze kód, který projde všemi automatizovanými testy.

Co se týče samotné struktury větví a to jak s nimi vývojář pracuje zde nejsou žádná omezení. Je zde široká možnost nastavení pro podporu snad jakéhokoliv Git workflow. Mezi hlavní používané z nich jsou především Git flow[1] a GitHub flow[2].

1.4.2.2 Continues integration

Další významnou součástí, kterou lze využít pro potřeby implementační části je GitLab–CI. Komponenta obsahuje integraci přímo do GitLab instance, ale vyžaduje konfiguraci a povolení správcem. Je nutné nakonfigurovat tzv. *runnery*, které jsou zodpovědné za spouštění vývojáři definovaných úloh, které se v určitých okamžicích spustí a vykonají různorodé akce od integračních úloh až po nasazení aplikace na jednotlivá prostředí.

Výhodou je, že konfigurace toho, jak se GitLab–CI má chovat je součástí přímo zdrojového kódu dané aplikace. Do kořene souborové struktury Git repositáře stačí umístit soubor s názvem `.gitlab-ci.yml`, který je při provedení operace *git push* do vzdáleného repositáře přečten a jsou provedeny veškeré zde uvedené akce, pokud jsou splněny podmínky k jejich aktivaci. Jednoduchý příklad může být například to, že při nahrání kódu do hlavní větve dojde k jeho kompilaci a výsledek je publikován v podobě *artefaktu* pro další použití – například pro nasazení na některé z prostředí.

Co se týče obsahu úloh, které se v rámci GitLab–CI mají spustit, může se v základní variantě jednat o pouhé bash skripty, které provedou požadované akce. V tomto případě je omezen pouze na nástroje, které obsahuje administrátorem nakonfigurovaný runner, případně si je musí před provedením každého skriptu nainstalovat v rámci konfiguračního souboru sám. Pro eliminaci závislostí na připraveném prostředí lze použít Docker obraz. Může se jednat o již veřejně dostupný obraz a nebo nějaký vlastní. V tomto případě jsou skripty spouštěné ve vývojářem definovaném prostředí a nemůže dojít k problém s některou z externích závislostí.

Součástí nástroje GitLab je i integrované Docker registry. To je možné jak pro uchování obrazů použitých v rámci GitLab–CI, tak i pro publikací

[1] https://nvie.com/posts/a-successful-git-branching-model

[2] http://scottchacon.com/2011/08/31/github-flow.html

jakýchkoliv obrazů například jako artefaktů pro vyvíjené aplikace. Do registry je možné ukládat libovolné množství obrazů (pokud administrátor nenastaví jinak) a libovolně je strukturovat. Registry lze vystavit do internetu (případně v rámci intranetu) a při použití patřičné autorizace použít k nasazování obrazů na cílové stroje, kde aplikace reálně běží.

GitLab obsahuje všechny potřebné součásti, od správy zdrojových kódů s případným issue trackingem, po automatizované CI úlohy až po publikování artefaktů do integrovaného Docker registry a jejich nasazení do cílových prostředí. To vše zabalené do přehledného webového rozhraní dostupného všem existujícím uživatelům. Jedná se tedy o ideální nástroj pro implementaci metodik CI/CD v rámci našeho projektu. Celá sekce o nástroji GitLab čerpá informace ze zdroje [16].

1.4.3 Docker

Docker je aktuálně nejpoužívanejší a open-source nástroj pro tvorbu a provoz kontejnerových aplikací podle reportu [17] služby Sysdig, poskytující nástroje pro monitoring systémů. Princip kontejnerů se často také nazývá jako virtualizace na úrovni operačního systému. V základu se jedná o využívání již existujících funkcí linuxového jádra, tak aby se dosáhlo maximální izolace aplikace běžící v kontejneru. Jde především o `cgroups`, které umožňují správu přidělených prostředků (CPU, RAM, I/O, síť, …) pro daný proces. Dále pak o `namespaces`, které slouží k izolaci na úrovni skupin prostředků, které proces reálně uvidí – například jde o vlastní prostor pro procesy, souborový systém a nebo síťová rozhraní. Izolovaná aplikace pak nevidí všechny prostředky hostitelského operačního systému, ale pouze svůj namespace (případně sdílené prostředky, podle konfigurace).

Kromě spouštění kontejnerů je možné také pomocí nástroje nové obrazy pro kontejnery vytvářet. Již s příchodem prvních nástrojů vznikla Open Container Initiative (OCI), která má za cíl formát obrazů sjednotit. Není tedy problém obrazy vytvořené přes Docker konzumovat v ostatních kontejnerových nástrojích (podman, cri-o, …) a naopak. K popisu obrazů slouží v tomto případě soubor nazvaný `Dockerfile`, který obsahuje krok za krokem postup jak pomocí integrovaných příkazů takový obraz vytvořit. Obrazy na sebe lze libovolně vrstvit, většinou při vytváření nějakého nového vycházíme již z existujícího, který obsahuje potřebné nástroje – například `debian:stretch`[3].

Hotové obrazy se následně uchovávají v některém z Docker registry. Veřejně hostovaná variantou je oficiální Docker Hub na adrese hub.docker.com. Je možné si registry spustit lokálně na vlastní infrastruktuře a mít ho tak plně pod kontrolou. Jedna z možností je například využít registry, které je součástí GitLab o kterém se zmiňuje kapitola 1.4.2.2.

[3] `https://hub.docker.com/_/debian`

Aby mohl cílový stroj spouštět a provozovat Docker kontenery, musí mít nainstalovaný software nazývaný *Docker daemon*, který zajišťuje vlastní funkcionalitu. Ten následně vystavuje API pomocí kterého je možné ho ovládat. K tomu slouží další software nazývaný *Docker CLI*, který se na API daemona po autorizaci připojí a ovládá jej. Tato klient–server architektura je nutná i v případě kdy daemon i jeho klient běží v rámci stejné instance operačního systému.

Opět se jedná o ideální nástroj pro realizaci implementační části práce. Je tak možné docílit jednotného a neměnného prostředí jak pro sestavení aplikace, tak pro její provoz na cílovém operačním systému bez žádných dalších závislostí. Zároveň se nabízí i možnost jeho použití v rámci CI/CD procesů jakožto zabudované součásti do GitLab–CI. Celá sekce je založena na informacích ze zdroje [18].

1.4.3.1 Docker Compose

Docker Compose je oficiální rozšíření nástroje Docker pro správu komplexnějších aplikací skládající se z více komponent – v tom tomto případě je každá komponenta reprezentována jedním kontejnerem. Takto strukturovaná aplikace je definována souborem `docker-compose.yml`, který obsahuje všechny informace nutné k sestavení i ke spuštění celé aplikace.

```
version: '3'
 services:
   web:
     build: .
     ports:
     - "5000:5000"
     volumes:
     - .:/code
     links:
     - redis
   redis:
     image: redis
```

Ukázka kódu 1.1: Příklad Docker Compose definice

Námi vytvořené součásti jsou tak z centrálního konfiguračního souboru odkazovány pomocí již zmíněného `Dockerfile`, který se jednoduše použije pro sestrojení obrazu podle naší definice. Další možností je odkázání se na externí již existující obrazy třetích stran, které pouze konzumujeme – může se například jednat o databáze a podobná hotová řešení. K sestavení celé aplikace slouží příkaz `docker-compose build` a ten provede stažení všech externích obrazů a případně sestaví naše interní komponenty.

Pro výsledné spuštění je zde obsažena i definice všech nastavení potřebných pro běh jednotlivých kontejnerů. U každé komponenty lze specifikovat například proměnné prostředí, připojené diskové oddíly a nebo třeba mapování síťových portů. Důležitou součástí je propojení jednotlivých komponent mezi sebou, pokud to dává v rámci aplikace jako celku smysl, pomocí tzv. *linků* – například webová aplikace může potřebovat komunikovat s databází, ale jedná se o dva rozdílné kontejnery, takže musíme vytvořit propojení. Ke spuštění celé aplikace slouží příkaz `docker-compose up`, který spustí všechny komponenty a stará se o to, aby neustále běžely ve správné konfiguraci i v případě výpadku některého z nich.

Nástrojů pro orchestraci kontejnerů existuje mnoho a Docker Compose je pouze jedním z nich. Výhodou je jeho snadné použití, které dostatečně pokrývá potřeby implementační části projektu. Alternativou může pokročilejší nástroj Docker Swarm, který je velmi podobný, ale nabízí mnohem více funkcí a podporuje provoz kontejnerů v rámci clusteru složeného z několika strojů. Pro opravdu rozsáhlé projekty ani tyto nástroje nestačí a je nutné použít alternativy typu Kubernetes, OpenShift a další, které jsou navržené cíleně pro systémy s velkým množstvím pracovních uzlů a běžících kontejnerů. Informace o nástroji Docker Compose pochází z oficiální dokumentace [19].

Kapitola 2

Specifikace požadavků

Nyní si popíšeme požadavky, kterou jsou na grafickou knihovnu kladeny a na základě kterých se budou odvíjet další části práce. Při specifikaci požadavků je nutné vycházet především z analýzy současného stavu, kde byly identifikovány nedostatky a prostory pro zlepšení. Dále je nutné zohlednit všechny požadavky v zadání, tak aby každý z nich mohl být splněn. Celá tato kapitola je vypracována za použití informací a doporučení ze zdroje [20].

Požadavky jsou klasicky rozděleny na funkční a nefunkční protože se jedná kromě samotné funkcionality i o návrh celkové architektury, takže dává smysl si stanovit oba typy požadavků.

2.1 Funkční požadavky

Mezi funkční požadavky patří především samotné dílčí funkcionality, které jsou kladeny na systém jako celek a lze jednoznačně vymezit jejich rozsah a rozhodnout následném splnění či nesplnění. Z velké části odráží specifikace dané zadáním práce.

F1 Vytvoření virtuálního grafického plátna

Uživatel bude přistupovat k celistvému pevně velkému plátnu. Nad fyzickou strukturou fasády tedy existuje virtuální vrstva, takže vše pod ní nemusí uživatel vůbec řešit a je zajištěno knihovnou. Obsah plátna je následně vykreslen na fyzickou část fasády.

F2 Manipulace s objekty v rámci plátna

Uživatel má možnost do plátna umisťovat libovolné množství objektů z pevně dané sady námi definovaných typů. Je možné zpětně manipulovat s již vytvořenými objekty a i s plátnem jako celkem.

F3 Nastavení vlastností a efektů pro každý objekt

Každý objekt může obsahovat mnoho vlastností podle toho jak určuje jeho definice. Uživatel může libovolně tyto vlastnosti číst a editovat. Stejně tak může uživatel aplikovat na objekty námi definované efekty, které opět mohou mít sadu svých stejným způsobem editovatelných vlastností.

F4 Simulace plátna v reálném čase bez vnější interakce

Objekty nemusí být nutně pouze statické a editovatelné uživatelem, ale především za použití efektů pro jejich pohyb či transformace mohou být i dynamické. Tato dynamičnost musí být zajištěna i bez interakce uživatele musí být promítnuta do výsledného zobrazovaného stavu plátna.

F5 Náhled na aktuální stav plátna

Pro zkušební účely má uživatel možnost zobrazit si aktuální stav plátna i jinde než na fyzické fasádě se zachováním totožného komunikačního rozhraní.

F6 Strojově zpracovatelná definice aplikačního rozhraní

Uživatel má k dispozici strojově čitelnou a kompletní definici komunikačního rozhraní s knihovnou

F7 Dynamická správa izolovaných prostředí aplikace

Administrátor knihovny má možnost dynamicky přidávat prostředí pro jednotlivé aplikace. Ty jsou od sebe kompletně izolované a každé z nich nabízí totožné možnosti se zachováním stejného komunikačního rozhraní. Administrátor určuje, která aplikace bude mít přístup k prostředí napojenému přímo na fyzickou fasádu.

2.2 Nefunkční požadavky

K nefunkčním požadavkům se řadí více abstraktní nároky na systém, které nelze snadno vyhodnotit na základě samotné implementace, ale lze je například sledovat podle některých z klíčových metrik.

NF1 Aplikace bude rozdělena do škálovatelných komponent

Protože se jedné o aplikaci operující převážně v reálném čase, jsou předpokládány vysoké nároky na výpočetní čas. Proto je nutné zajistit, aby jednotlivé komponenty bylo možné nezávisle na sobě škálovat až na úroveň toho, že každá aplikace bude mít vlastní dedikovaný hardware.

NF2 Odezva aplikace bude co nejblíže reálnému času

Výstup aplikace je převážně grafického rázu s možným výskytem rychle se měnících stavů plátna. Proto je nutné se zaměřit na snížení odezvy zpracování požadavků od uživatele a jejich promítnutí do fyzického světa tak, aby pozorovatel nezaznamenal viditelné zpomalení, které může komplikovat především interaktivní aplikace.

Kapitola 3

Návrh

V této části práce se budeme věnovat návrhu výsledného řešení tak, aby byly splněny požadavky z předchozí kapitoly. Nejprve budou navrženy možné případy užití rozdělené podle jednotlivých identifikovaných aktérů. Díky nim bude možné zkontrolovat splnění požadavků. Následovat bude návrh datového modelu pro reprezentaci plátna a všech jeho souvisejících objektů. Závěrem dojde na návrh nasazení celkové aplikace a procesů týkajících se integrace s dalšími již existujícími komponentami.

3.1 Model případů užití

Celá tato sekce včetně diagramů a tabulek byla vypracována na podle doporučení ze zdroje [20].

3.1.1 Aktéři

Na základě funkčních požadavků bylo identifikováno několik klíčových aktérů, kteří budou v případech užití figurovat. Nyní následuje jejich výčet a stručný popis.

3.1.1.1 Uživatel

Uživatelem je označen konzument rozhraní grafické knihovny. Ve skutečnosti to může být buď přímo vývojář konkrétní aplikace a nebo ona samotná. Jedná se o primárního aktéra v rámci naší aplikace.

3.1.1.2 Testovací uživatel

Pro vývojové účely existuje speciální role, která má stejně možnosti jako klasický uživatel kromě zobrazování obsahu na fyzickou fasádu. Protože se jedná

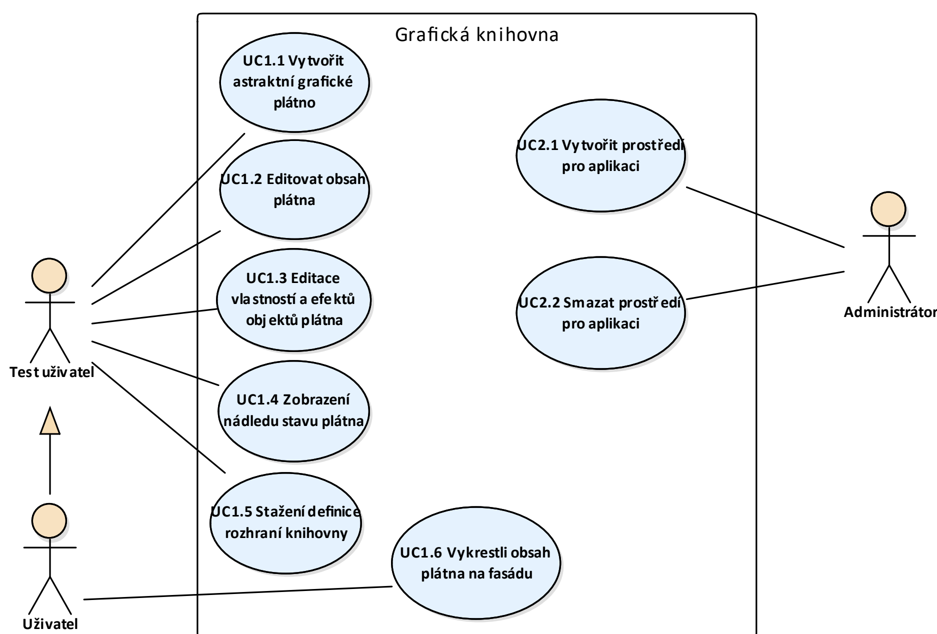
především o otevřenou platformu pro vývojáře tak je třeba rozlišovat i takové druhy aktérů.

3.1.1.3 Administrátor

Administrátor spravuje jednotlivá prostředí pro uživatele. Jde tedy o servisní roli, která není přímo určena pro třetí strany ale pro integraci s ekosystémem linky. Může jím být buď přímo jednotlivec a nebo například *Linky core*.

3.1.2 Případy užití

Jednotlivé případy užití včetně rozdělení mezi popsané aktéry je možné vidět na obrázku 3.1.



Obrázek 3.1: Diagram případů užití pro grafickou knihovnu

3.1.3 Splnění požadavků

Jako jeden z nástrojů pro kontrolu splnění funkčních požadavků lze použít jejich provázání s jednotlivými případy užití. Tím se ověří, jestli každý z požadavků bude realizován a naopak i zda případy užití dávají smysl v kontextu funkčních požadavků. Shrnutí je realizováno pomocí tabulky 3.1.

Jak můžeme vidět, tak všechny případy užití řeší některý z funkčních požadavků. Ke všem kromě jednoho požadavku také existuje alespoň jeden případ užití. Funkční požadavek **F4** není namapovaný na žádný případ užití, jelikož se jedná o interní proces, který není iniciován žádným z aktérů ani v podobě

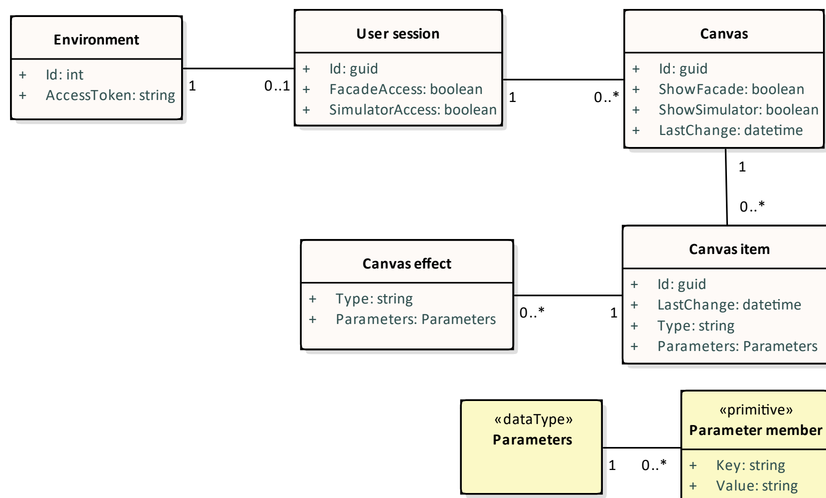
	UC1.1	UC1.2	UC1.3	UC1.4	UC1.5	UC1.6	UC2.1	UC2.2
F1	+					+		
F2		+						
F3			+					
F4								
F5				+				
F6					+			
F7							+	+

Tabulka 3.1: Kontrola splnění funkčních požadavků

externího systému a jeho popis následuje dále v této kapitole. Tabulkou jsme tedy funkční požadavky úspěšně ověřili.

3.2 Doménový model

V rámci domény je nutné zachytit klíčové entity, se kterými bude aplikace interně i veřejně operovat. U každé takové entity jsou vždy znázorněny její definující atributy včetně datových typů a případných vlastních typů. Přehled doménového modelu pro aktuální stav plátna grafické knihovny, se kterým bude uživatel a aplikace manipulovat je zachycený na obrázku 3.2. Doménový model byl vytvořen podle informací ze zdroje [21].



Obrázek 3.2: Datový model pro uživatele a plátno

3.2.1 Entity

3.2.1.1 Prostředí

Aplikační prostředí je jednoznačně identifikovatelné pomocí svého pořadového čísla. Spravováno je administrátorem, který určuje přístupový klíč, který bude aktuálně pro dané prostředí aktivní.

3.2.1.2 Uživatelská session

V rámci každého prostředí může existovat maximálně jedna session, která je definována unikátním náhodně generovaným identifikátorem. Je v rámci ní pevně nastaveno, jaké možnosti uživatel má a kam všude může přistupovat.

3.2.1.3 Plátno

Uživatel si teoreticky v rámci své session může vytvořit libovolný počet pláten. Každé plátno má též svůj unikátní identifikátor. Uživatel může u každého plátna určit jeho nastavení, které je ale omezené nastavením dané session. Pro účely překreslování existuje ještě atribut evidující datum a čas poslední změny plátna.

Plátno v sobě obsahuje jednotlivé položky. Ty jsou také identifikovány identifikátorem včetně evidence data a času poslední změny. Druh každého objektu je určen typem z daného rozsahu předem definovaných typů. Položka může obsahovat libovolné množství parametrů typu klíč–hodnota.

Na položku plátna lze aplikovat efekt, ten se nachází přímo uvnitř položky a je určený podobně jako položka samotná typem efektu z existujícího rozsahu. Parametry jsou zde totožné struktury jako u položky plátna.

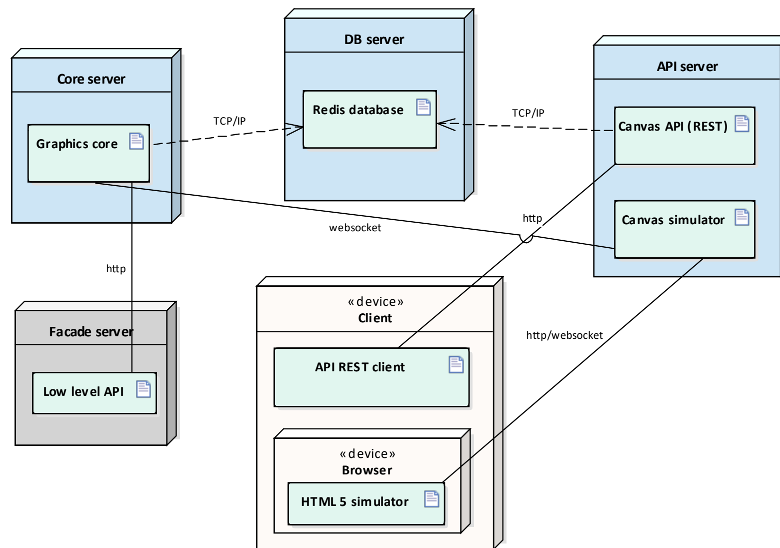
3.3 Nasazení aplikace

Aplikaci je nutné rozdělit na jednotlivé oddělené funkční celky, tak aby bylo možné dosáhnout požadované škálovatelnosti. Na nejvyšší úrovni jde o rozdělení na tři hlavní komponenty, které budou blíže popsány v následujících podkapitolách. Toto vysokoúrovňové rozdělení včetně externích systémů a klientského zařízení je zachyceno na obrázku 3.3 (diagram byl vypracován na základě zdroje [22]).

3.3.1 Komponenty aplikace

3.3.1.1 Datové úložiště

Úložiště je klíčová komponenta, která zde zajišťuje jako jediná persistenci dat. V úložišti se nachází veškerá dynamická konfigurace provedená administrátorem systému. Hlavní je ale aktuální stav plátna pro všechny prostředí a uživatelské session.



Obrázek 3.3: Obecný diagram nasazení aplikace

3.3.1.2 Aplikační rozhraní

Jak můžeme vidět již ze zmíněného diagramu nasazení, s datovou vrstvou komunikuje především aplikační rozhraní, které umožňuje uživatelům číst aktuální stav plátna a administrátorům i stav prostředí. Hlavní funkcí je ale editace stavu plátna (respektive prostředí), které se okamžitě promítnou do datového úložiště.

3.3.1.3 Výpočetní jádro

Ústřední komponentou je jádro celého systému, které také komunikuje s datovou vrstvou, ale pouze v režimu pro čtení. Hlavním úkolem je načtení aktuálního stavu plátna z datové perzistentní vrstvy a vytvoření či aktualizace reálného stavu plátna, které je drženo pouze v operační paměti. Na základě definice tak prování vykreslování definovaných objektů a aplikace efektů.

Výstupem jsou surová data připravená pro zobrazení na cílových konzumentech. Tato data jsou následně v závislosti na nastavení odesílána buď to na simulátor a nebo přímo na fyzickou fasádu prostřednictvím externího systému.

3.3.1.4 Simulátor

Jde o komponentu kompletně nezávislou na zbytku systému. Slouží pouze ke zobrazování obrazových dat v reálném čase, které jsou ji zasílány grafickým jádrem. V rámci simulátoru je možné vytvářet nezávislé instance, které jsou rozlišovány pomocí unikátního identifikátoru. Takže je HTML5 klient pouze připojí do nějaké existující instance na základě identifikátoru který obdrží.

Pomocí něho jsou mu následně nezávisle zaslána obrazová data jádrem a on je pouze zobrazí na klientovi.

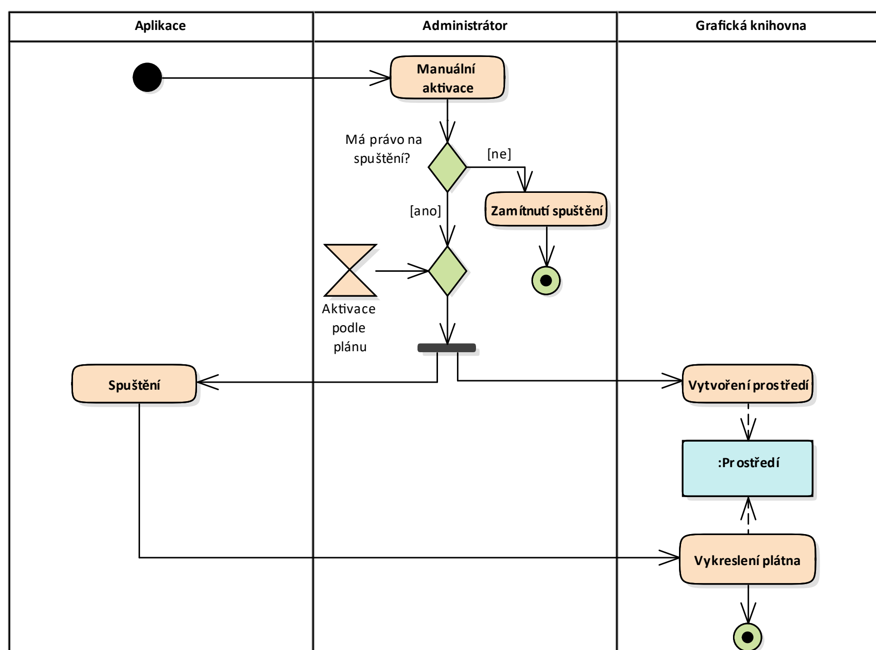
3.3.2 Externí systémy

3.3.2.1 Linky API

Jediným čistě externím systémem, se kterým musí systém aktivně komunikovat a o kterém musí vědět jsou samotné fyzické Linky dostupné pomocí low level aplikačního rozhraní. S tímto systémem komunikuje pouze jádro systému, které mu podobně jako v případě simulátoru posílá surová obrazová data ke zobrazení.

3.4 Proces interakce s administračním rozhraním

Jednotlivá prostředí a jejich nastavení je nutné spravovat administrátorem, který je může určovat dynamicky podle potřeby. Bylo tedy nutné navrhnout proces, který pokryje všechny oblasti od cílové aplikace, která chce zobrazovat, přes administrátora, který rozhodne jak takový požadavek zpracovat až po grafickou knihovnu, která ho reálně vykoná.



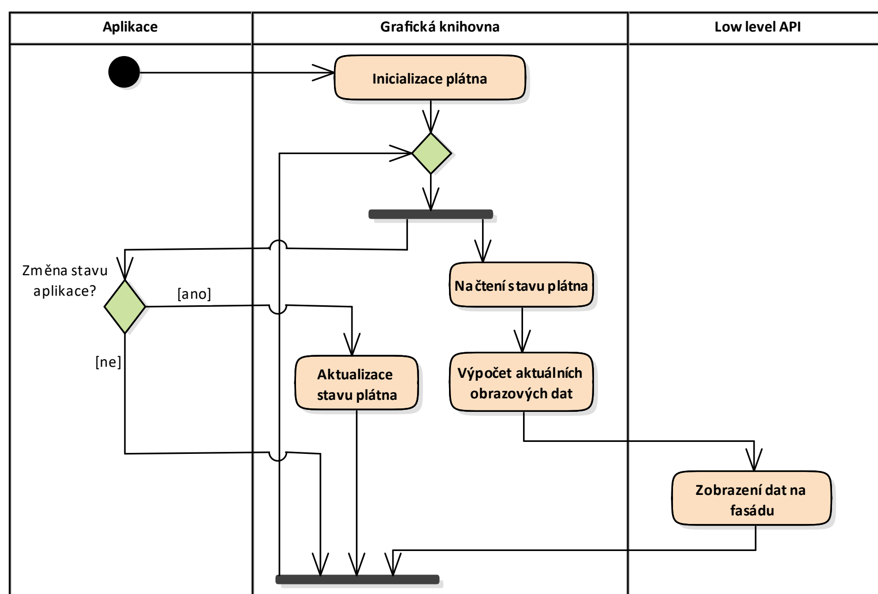
Obrázek 3.4: Diagram aktivit pro administrační rozhraní

Výsledná podoba procesu je zachycena na obrázku 3.4 (diagram byl vypracován na základě doporučení ze zdroje [23]). Rozhodující logika ke spuštění

aplikace je v kompetencích administrátora. Záleží pouze jestli je o něj požádáno manuálně a nebo jde o spuštění podle časového plánu, který je kompletně spravován administrátorem. Pro realizaci spuštění dojde k vytvoření prostředí v grafické knihovně a k samotné informaci aplikaci, že se má spustit. Následně již komunikuje aplikace přímo s grafickou knihovnou, kam odesílá data k vykreslení v rámci daného prostředí.

3.5 Proces interakce s low level Linky API

Pro zachycení komunikace přímo s grafickou knihovnou, která následně komunikuje již přímo s low level rozhraním existuje speciální proces. Bylo nutné zachytit způsob zobrazení požadovaných dat a jejich aktualizace od aplikace skrze grafickou knihovnu až na fyzickou fasádu.

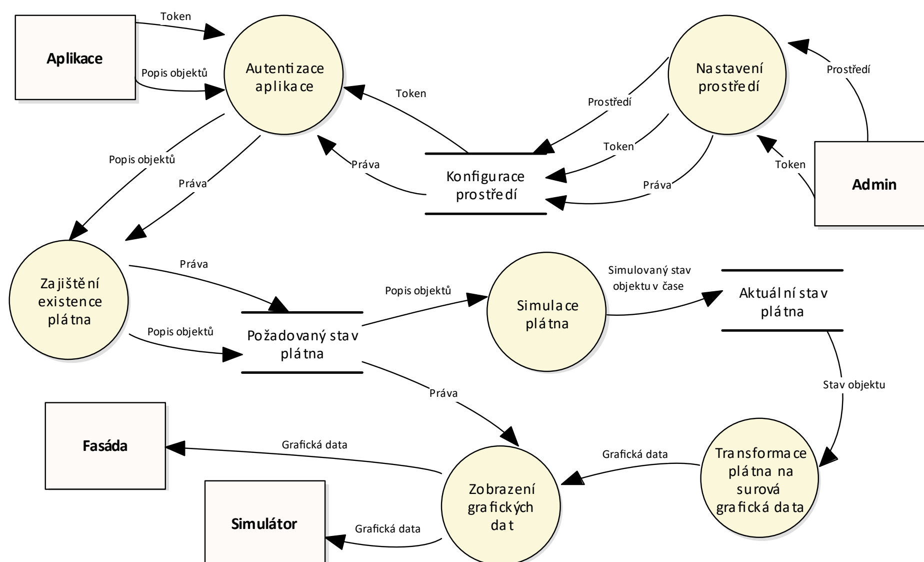


Obrázek 3.5: Diagram aktivit pro low level API

Navržený proces zachycuje obrázek 3.5 (diagram byl vypracován na základě doporučení ze zdroje [23]). Jeho iniciátorem je vždy samotná aplikace, která má zájem zobrazovat grafická data. Na začátku dojde k inicializaci grafické knihovny a vytvoření nového virtuálního grafického plátna. Následně jsou paralelně zpracovávány požadavky aplikace na změny a výpočet aktuálních dat, které jsou následně odesílány pomocí daného rozhraní přímo na fasádu.

3.6 Pohyb dat v celém systému

Pro přehled všech důležitých procesů a jejich vztahu s konkrétními částmi dat, se kterými celý systém pracuje je součástí návrhu i diagram datových toků zachycený na obrázku 3.6. Znázorněny jsou zde všechny důležité vystupující entity, které jsou zdroji a konzumenty konkrétních částí dat. Data jsou následně zpracovávána jednotlivými procesy a během zpracování ukládána do datových úložišť. Diagram je typu *„Yourdon and Coad"* vytvořený podle informací ze zdroje [24].



Obrázek 3.6: Diagram datového toku pro grafickou knihovnu

Kapitola 4

Implementace

Předmětem této kapitoly práce je realizace již navržených komponent z předešlé části práce. Při implementaci je nutné zmínit i všechny detaily a případné odklony od navrhovaného řešení. Výsledkem by měla být mimo jiné i specifikace pro nasazení celého systému.

4.1 Vývojové prostředí

V rámci prostředí ve kterém byla aplikace vyvíjena by neměly být žádné specifické závislosti, pro úplnost je ale dobré uvést jeho klíčové parametry a verze jednotlivých nástrojů, na kterých byla implementační část realizována:

Windows 10 Pro build 1809;

Visual Studio Enterprise 2017 verze 15.8.9;

Docker Desktop Community verze 2.0.0.3, engine verze 18.09.2;

Docker Compose verze 1.23.2.

U dalších použitých knihoven na verzi buď přímo nezáleží a nebo to bude zmíněno při implementaci konkrétní komponenty či přímo patrné z konfiguračních souborů, které budou součástí přiložených zdrojových kódů implementační části.

4.2 Realizace jednotlivých komponent

Před tím, než přejdeme k jednotlivým navrženým komponentám je nutné zmínit důležitou součást systému, která bude jednotlivé části propojovat a zajišťovat že každá z nich je k dispozici. To je zajištěno díky nástroji Docker Compose, který slouží k orchestraci kontejnerových aplikací v podobě kterých jsou zastoupeny jednotlivé dále popsané komponenty.

```yaml
version: '3.4'
services:
 canvas.data:
  image: redis:5-alpine
 canvas.api:
  image: ${DOCKER_REGISTRY}canvasapi:${IMAGE_TAG:-latest}
  build:
   context: .
   dockerfile: src/Services/Canvas/Canvas.API/Dockerfile
  depends_on:
   - canvas.data
 canvas.simulator:
  image: ${DOCKER_REGISTRY}canvassimulator:${IMAGE_TAG:-latest}
  build:
   context: .
   dockerfile: src/Services/Canvas/Canvas.Simulator/Dockerfile
 graphics.core:
  image: ${DOCKER_REGISTRY}graphicscore:${IMAGE_TAG:-latest}
  build:
   context: .
   dockerfile: src/Services/Graphics/Graphics.Core/Dockerfile
  depends_on:
   - canvas.data
   - canvas.simulator
```

Ukázka kódu 4.1: Docker Compose definice grafické knihovny

Popis celého systému založený na `docker-compose.yml` definici je přiložený jako ukázka kódu 4.1. Každé komponentě odpovídá jedna definice služby, která je definována svým názvem a kontejnerovým obrazem, který ji implementuje. V našem případě jsou součástí i odkazy na předpisy jak takové obrazy sestavit. Z ukázky jsou také patrné závislosti mezi jednotlivými součástmi systému, které zachycuje diagram nasazení 3.3 v návrhové části.

Součástí konfigurace jsou i proměnné prostředí, kterými lze ovlivňovat označení obrazů komponent. Proměnná `DOCKER_REGISTRY` určuje adresu repositáře, kde bude výsledný obraz uložen. V případě lokálního vývoje je proměnná prázdná a všechny obrazy tak existuje pouze v rámci lokálního stroje. Druhá proměnná `IMAGE_TAG` slouží k označení konkrétní verze obrazu pro jejich odlišení například v rámci verzovacího systému. V lokálním případě obsahuje výchozí hodnotu `latest`.

4.2.1 Aplikační rozhraní

Hlavní rozhraní sloužící k ovládání celé grafické knihovny je implementováno jako ASP.NET Core 2.1 aplikace. Jde o klasický způsob jak zde realizovat API dodržující návrhový vzor MVC. Každá entita obsahuje vlastní controller, který umožňuje provádění určitých operací. View je realizováno jednoduchými DTO objekty, které slouží pouze k přenosu dat a neobsahují žádnou logiku. Model zastupuje servisní vrstva, kde se nachází veškerá business logika a je zde i obsažena komunikace s datovou vrstvou skrze repositáře vztahující se ke konkrétním entitám. Implementace je obsažena v projektu `Canvas.API`.

```
[ApiVersion("1.0")]
[Route("api/v{version:apiVersion}/[controller]")]
[ApiController]
[Authorize]
public class CanvasController : ControllerBase
{
  private readonly ICanvasService canvasService;

  public CanvasController(ICanvasService canvasService)
  {
    this.canvasService = canvasService
      ?? throw new ArgumentNullException(nameof(canvasService));
  }
}
```

Ukázka kódu 4.2: Definice controlleru v ASP.NET Core

V ukázce 4.2 je vidět část definice aplikačního controlleru. Každý je tvořen samostatnou třídou, která dědí z `ControllerBase` cože je základní třída obsahující společnou funkcionalitu jako například reakce na příchozí požadavek a nebo pomocné metody pro vrácení odpovědi. Dále je třída konkrétního controlleru vždy obalena atributy, které ji přidávají další vlastnosti a chování. Postupně jde o atribut `ApiVersion` který zajišťuje verzování API, což je následně promítnuto do generování OpenAPI specifikace. Dalším důležitým je atribut `Route`, který definuje přesnou cestu, kde bude daný controller dostupný v rámci URL. Směrovací šablona může jako v tomto případě obsahovat proměnné například pro přítomnost verze nebo pro název daného controlleru. Atribut `ApiController` označuje daný controller určený pouze pro potřeby poskytnutí API což zajistí automatickou validaci vstupního modelu a mapování vstupních DTO na objekty. Poslední atribut `Authorize` omezuje dostupnost všech operací daného controlleru pouze pro autorizované požadavky pomocí výchozího schématu. V ukázce je vidět i získávání závislostí v podobě dalších tříd či rozhraní pomocí DI, zde při získávání instance `ICanvasService`.

4.2.1.1 Dostupné operace

Velká část funkčních požadavků na systém je realizována právě v této komponentě. Proto zde nyní uveďme stručný přehled všech operací poskytovaných v rámci API:

- správa prostředí administrátorem
  (api/v1/Admin/environment – GET/POST/DELETE);
- správa plátna aplikací (api/v1/Canvas – GET/PATCH);
- správa všech položek plátna aplikací
  (api/v1/Canvas/items – GET/DELETE);
- správa jednotlivých položek aplikací
  (api/v1/Canvas/item – GET/POST/DELETE);
- informace o simulátoru pro aplikaci (api/v1/Canvas/simulator – GET);
- vytvoření testovacího účtu uživatelem (api/v1/User/testToken – POST).

4.2.1.2 Autorizace a autentizace

Jelikož se jedná o veřejně dostupné rozhraní, tak je nutné ho řádně zabezpečit a umožnit provádění určitých operací pouze některým klientům. Autentizace je řešena kompletně bezstavově pomocí aplikačních tokenů. Každému klientovi je přidělen unikátní řetězec, který je znám pouze jím a rozhraním. Díky tomu je možné klienta jednoznačně identifikovat.

Pro přístup k API jsou podporována dvě autentizační schémata. První je určeno pouze pro administrační přístup. Tokeny pro tento druh ověření jsou manuálně generované a přímo součástí konfigurace aplikace, protože není nutné je často měnit. Ověření je prováděno pomocí kontroly HTTP hlavičky `X-Token` a je použit pro autorizaci pouze k operacím v rámci admin controlleru.

Druhé podporované schéma je také realizováno pomocí HTTP hlavičky. V tomto případě jde o `Linky-Token`, což je token nastavený administrátorem a svázaný s konkrétním prostředím, které má v sobě určitě nastavení práv. Následná autorizace probíhá na úrovni claimů, které jsou klientovi přiděleny při procesu autentizace. Jde především o `linky.sessionId`, který popisuje unikátní identifikátor uživatele v kontextu prostředí. Dále pak o `linky.simulator` a `linky.facade`, které určují práva daného klienta pro zobrazování pouze do simulátoru a nebo přímo na fyzickou světelnou stěnu.

Autorizace je pak následně kontrolována na úrovni celého controlleru nebo pouze jeho vybrané akce. Pomocí atributu `Authorize` je možné určit jaká jsou povolena schémata pro autorizaci a nebo zda jsou vyžadovány nějaké speciální claimy. V případě že nejsou požadavky pro autorizaci splněny dostane klient

automaticky odpověď pomocí HTTP kódu 401 – Unauthorized v případě kompletně chybějící autorizace a nebo 403 – Forbidden v případech nedostatečného oprávnění.

Aby se požadované chování promítlo i do Swagger respektive OpenAPI specifikace, bylo nutné provést i zde speciální konfiguraci ve. V ukázce 4.3 je vidět, že nejprve je aplikován filtr `SecurityRequirementsOperationFilter`, který se provede nad všemi operacemi všech controllerů a podle přítomnosti `Authorize` atributu takové akce označí jako autorizované a přidá odpovídající HTTP kódy (401, 403), které takové operace mohou produkovat. Následně jsou přidány obě již zmíněné autentizační metody, tak aby byly obsažené ve specifikaci včetně způsobu jejich použití a informačního popisu. Daná konfigurace se nachází v `Startup.cs` a je součástí komplexnějšího nastavení pro Swagger v rámci tohoto projektu.

```
setup.OperationFilter<SecurityRequirementsOperationFilter>();
setup.AddSecurityDefinition(LinkyAuthentication.Sheme,
    new ApiKeyScheme()
    {
      In = "header",
      Name = LinkyAuthentication.Header,
      Type = "apiKey",
      Description = LinkyAuthentication.Description
    });
setup.AddSecurityDefinition(LinkyAdminAuthentication.Sheme,
    new ApiKeyScheme()
    {
      In = "header",
      Name = LinkyAdminAuthentication.Header,
      Type = "apiKey",
      Description = LinkyAdminAuthentication.Description
    });
```

Ukázka kódu 4.3: Konfigurace autentizace a autorizace pro Swagger

4.2.1.3 OpenAPI specifikace

Pro vygenerování kompletní OpenAPI specikace je nutné dodržet konvence pro komentování především akcí v controllerech a jejich označení správnými atributy. Jak je názorně vidět v ukázce 4.4, kromě komentování samotné metody a jejich vstupních parametrů jsou popsány i jednotlivé návratové kódy, které mohou při operaci nastat. To je kromě samotných komentářů nutné označit i pomocí atributu `ProducesResponseType`, jehož hodnotou je HTTP kód možné produkované odpovědi. Pro označení HTTP metody, pomocí které je

operace dostupná slouží atribut HttpGet a jeho další varianty pro ostatní metody. Součástí může být i vlastní šablona pro URL na které bude akce v rámci controlleru dostupná. Swagger UI vygenerované právě na základě této specifikace je ukázáno v příloze B.1 a její část pro jednu z operací v příloze C.1.

```
/// <summary>
/// Gets state of specified environment.
/// </summary>
/// <param name="envId">Id of environment.</param>
/// <response code="404">No environment with such ID exists.
/// </response>
/// <returns>Environment state.</returns>
[HttpGet("environment/{envId:int}")]
[ProducesResponseType(StatusCodes.Status404NotFound)]
[ProducesResponseType(StatusCodes.Status200OK)]
public async Task<ActionResult<EnvironmentStateResponse>>
               GetApplication(int envId)
{
  var result = await adminService.GetEnvironmentState(envId);
  if (result != null)
    return Ok(result);
  else
    return NotFound();
}
```

Ukázka kódu 4.4: Definice akce controlleru v ASP.NET Core

4.2.2 Datová vrstva

Veškeré operace, které se týkají načítání a ukládání dat jsou realizovány pomocí sady repositářů. Ty jsou součástí samostatného projektu Canvas.Data a ten je následně používán v dalších projektech.

Pro umožnění persistence všech dat je použita databáze Redis, která se svým primárním úložištěm typu klíč–hodnota ideálně hodí pro manipulaci s entitami v rámci grafické knihovny a je zajištěno především jejich velmi rychlé čtení. Pro přístup k databázi je použita knihovna StackExchange.Redis, která implementuje protokol pro komunikaci s Redis databází. Aby bylo možné krom primitivních typů ukládat i komplexní datové struktury, je nutné použít knihovnu, která zajistí převod struktur na podporované datové typy. V našem případě je použitý NewtonsoftSerializer, který veškeré objekty mapuje na textové řetězce obsahující jejich JSON reprezentaci. Pro použití stačí knihovnu zaregistrovat v rámci DI, jak můžeme vidět v ukázce 4.5 a následně pak používat pomocí rozhraní ICacheClient. V konfiguraci je kromě klienta

a prostředníka pro mapování objektů nutné nastavit i údaje nutné k pro přijení kam patří URL databáze, identifikace, přístupové údaje a další...

```
services.AddSingleton<ICacheClient, StackExchangeRedisCacheClient>();
services.AddSingleton<ISerializer, NewtonsoftSerializer>();
services.ConfigureOptions<ConfigureRedis>();
```

Ukázka kódu 4.5: Konfigurace knihovny pro přístup k Redis databázi

Co se týče přímo vrstvy samotných repositářů, tak ty operují přímo s rozhraním `ICacheClient`, které je zmíněno dříve. Sdružují pouze operace pro manipulaci s daty a neobsahují žádnou logiku. Jedná se tedy o vrstvu realizující základní CRUD operace nad entitami v podobě DTO objektů. Jak vypadá načtení stavu plátna, které je určeno pouze svým identifikátorem je obsaženo v ukázce 4.6. Další operace jsou velice podobné, proto není jim věnovat další pozornost.

```
public async Task<CanvasSessionInfo> GetSessionInfo(Guid id)
{
    return await this.cacheClient
      .GetAsync<CanvasSessionInfo>(GetSessionKey(id));
}
```

Ukázka kódu 4.6: Načtení objektu z Redis úložiště

4.2.3 Jádro systému

Grafické jádro je realizováno jako klasická .NET Core 2.1 konzolová aplikace. Jde tedy pouze o proces běžící na pozadí a komunikující s ostatními komponentami bez toho aby vystavoval své části. Kód implementace je situován do projektu `Graphics.Core` a vše se odehrává ve dvou na pozadí běžících úlohách.

První z nich je úloha `MainRenderingTask` zajišťující aktualizaci stavu virtuálního plátna a produkci obrazových dat, které jsou dále zpracovány jednotlivými konzumenty. Implementace se skládá z nekonečné smyčky, ve které jsou zpracovávány jednotlivé snímky pro dostupné instance plátna. Proces aktualizace se skládá z načtení požadovaného stavu pomocí datové vrstvy popsané v sekci 4.2.2. Tento stav je poté namapován na reprezentaci jednotlivých objektů v paměti se zachováním jejich interního stavu, pokud již existují. Stejně tak jsou namapovány i jakékoliv efekty a jejich parametry. Po té dojde k aplikaci či aktualizaci efektů na dané objekty. To je provedeno

jednoduchým spuštěním logiky, kterou obsahuje efekt definovaný v odpovídající třídě. Efekt je implementace rozhraní `IShapeEffect`, které předepisuje pouze aktualizaci z předaného seznamu parametrů a aplikaci na předaný grafický objekt. Hlavním výstupem jsou kromě aktualizovaného stavu i obrazová data současného snímku. Ta jsou přidána do fronty realizované rozhraním `IOutputBufferService`, kde jsou následně zpracována další nezávislou úlohou.

Druhou neméně důležitou úlohou je `OutputProcessingTask` starající se o zpracování obrazových dat z první úlohy. Ústředním prvkem je opět nekonečná smyčka, ve které se tentokrát načítají jednotlivé výsledky z fronty pomocí stejného rozhraní `IOutputBufferService`, kterým byla data vložena. Logika pro tento popisovaný proces se nachází ve třídě `OutputSenderService`, jejíž část můžeme vidět v ukázce 4.7. Po načtení jednotlivých záznamů z fronty dojde k jejich optimalizaci, tak aby v případě zpoždění nedošlo ke kumulaci snímků a zpomalení celého systému. Realizováno je pomocí použití pouze fixního počtu snímků a k zahození zbývajících. Pokud by se posílaly pouze změněné body, mohlo by být realizováno například slučováním snímků. Dále jsou výsledné snímky po jednom předány konzumentům, kteří je jsou schopni zpracovat. Konzumentem může být jakákoliv třída implementující rozhraní `IResultConsumer` registrována v rámci DI kontejneru. Součástí základní implementace jsou dva hlavní konzumenti. Prvním je simulátor, kde jsou data pouze přeposlána pomocí SignalR klienta komponentně simulátoru, která zajišťuje zbytek. Druhým konzumentem je fasáda jejíž implementační detaily jsou blíže popsány dále v této sekci.

```
public async Task ProcessResults(CancellationToken cancellationToken)
{
  var results = await outputBuffer
    .DequeueAllResults(cancellationToken);
  results = OptimizeResults(results);
  if (results.Any() && consumers.Any())
  {
    foreach (var result in results)
      foreach (var consumer in this.consumers
        .Where(c => c.CanComsumeResult(result)))
      {
        await consumer.ConsumeResult(result);
      }
  }
}
```

Ukázka kódu 4.7: Zpracování grafických dat snímku v jádře

4.2.3.1 Komunikace s fasádou

Komunikace s fyzickou fasádou probíhá skrze aktualizované aplikační rozhraní popsané během analýzy v sekci 1.2.2. Veškerá data jsou vyměňována pomocí HTTP protokolu a základní URL je součástí konfigurace grafického jádra. To jaké je nutné používat endpointy je patrné z ukázky 4.8, která zachycuje konstanty použité pro komunikaci s rozhraním.

```
static class FacadeApiConstants
{
  public const string PlayEndpointPath = "v1/Play";
  public const string CreateTokenEndpointPath = "v1/Token/Create";
  public const string RefreshTokenEndpointPath = "v1/Token/Refresh";
  public const string StateEndpointPath = "v1/State";
  public const HttpStatusCode BadCredentialsHttpCode
          = HttpStatusCode.Unauthorized;
}
```

Ukázka kódu 4.8: Důležité hodnoty pro komunikaci s fasádou

Jako první je nutné, aby grafické jádro získalo přístupový klíč pro zobrazování na fasádu. K tomu slouží endpoint `Token/Create`, kterému jsou předány údaje dostupné po registraci v rámci administračního portálu. Jde o unikátní identifikátor uživatele a s ním související *tajná* část. To je následně vyměněno za přístupový klíč a ten je jádrem uložen a používán k další komunikaci. Pokud se klíč blíží ke konci, dojde k jeho automatickému prodloužení jádrem pomocí endpointu `Token/Refresh`, kterému je předán dříve získaný obnovovací klíč. K obnovení dojde i pokud rozhraní odpoví HTTP stavovým kódem signalizujícím neplatný klíč. Před prvním odesláním obrazových dat je načten aktuální stav fasády pomocí ednpointu `State`. Pro jádro jsou důležité informace o reálně dostupném rozměru fasády, která se může měnit v závislosti na omezeních podle provozního řádu. Data jsou tomu přizpůsobena už na úrovni jádra a může dojít k jejich oříznutí či posunutí. Implementace zapouzdřující komunikaci s API se nachází ve třídě `FacadeApiAdapter`.

Hlavní částí je komunikace s endpointem `Play`, na který jsou odesílána vždy aktuální obrazová data produkovaná grafickým jádrem. Data jsou pouze oříznuta podle aktuálního stavu fasády a pak převedena do tvaru odpovídajícímu požadavkům rozhraní. Odesílán je vždy pouze jeden snímek a pomocí parametru `clear_buffer` je potlačeno jakékoliv ukládání a kumulace snímků na straně fasády. Veškerá logika pro odesílání dat na fasádu realizuje třída `FacadeConsumer`, což je implementace již známého rozhraní `IResultConsumer`.

4.2.3.2 Vykreslování plátna

Pro převod grafických objektů do rastrové podoby je na základě výběru v sekci 1.3.4 analýzy použita knihovna SkiaSharp. Implementace vykreslovací vrstvy se nachází v projektu `Graphics.Providers`, kde je řešeno pomocí implementace obecného rozhraní `IDrawingProvider` poskytujícího metody pro vykreslení základních geometrických útvarů a pro převod plátna na soubor jednotlivých bodů. Implementace jedné takové metody je znázorněna na ukázce 4.9 při vykreslení vyplněné elipsy konkrétní barvou. Grafické objekty se pomocí tohoto rozhraní umí vykreslit a je tak možné ho nezávisle na zbytku systému vyměnit například za jinou implementaci vykreslování.

```
public void FillEllipse(Point corner, Size size, Color color)
{
    var paint = new SKPaint()
    {
        Color = color.ToSkiaColor(),
        IsAntialias = false,
        Style = SKPaintStyle.Fill
    };
    var rect = new SKRect()
    {
      Location = new SKPoint(corner.X, corner.Y),
      Size = new SKSize(size.Width, size.Height)
    };
    canvas.DrawOval(rect, paint);
}
```

Ukázka kódu 4.9: Vykreslení elipsy pomocí knihovny SkiaSharp

Součástí hlavní vykreslovací logiky, která pomocí znázorněného postupu vykreslí všechny grafické objekty a jejich případné efekty je následně převedení celého plátna do dvourozměrného pole obrazových bodů. Z tohoto pole jsou vybrány přesně ty, které odpovídají sloupcům na fyzické fasádě a jsou definované v konfiguraci grafického jádra. Proces rasterizace virtuálního plátna je tedy realizován pomocí vykreslení celého prostoru v paměti a následným vybráním pouze relevantních data. Protože se jedná obecně o operace v nízkém rozlišení, neznamená to žádné výkonové problémy což bylo potvrzeno i testem rychlosti v sekci 1.3.4.5 analýzy.

4.2.4 Simulátor

Simulátor pro grafickou knihovnu je vytvořený jako ASP.NET Core 2.1 aplikace podobně jako aplikační rozhraní. Jedná se o zcela nezávislou součást,

která pouze slouží k přeposílání z zobrazování grafických dat na klientech v podobě webových prohlížečů.

Na straně serveru jde o použití technologie SignalR, díky které lze navázat obousměrné spojení jak s producenty dat, tak s jejich konzumenty. Integrace technologie do ASP.NET Core projektu je zajištěnou instalací jednoho NuGet balíčku `Microsoft.AspNetCore.SignalR`, který bývá součástí běžně používaného meta balíčku `Microsoft.AspNetCore.App`. Dále pak stačí pouze zaregistrovat jednotlivé komunikační uzly (huby), které budou dostupné pro klienty na pevně dané URL. Registrace se prování standardně v rámci `Startup.cs` a konfigurace použitá pro `CanvasHub`, který realizuje hlavní uzel pro simulátor zachycuje ukázka 4.10.

```
app.UseSignalR(routes =>
{
  routes.MapHub<Hubs.CanvasHub>("/canvasHub");
});
```

Ukázka kódu 4.10: Konfigurace SignalR hubu v rámci ASP.NET Core

Jako hlavní komunikační prvek slouží již zmíněný `CanvasHub` a ten obsahuje veškerou logiku potřebnou pro správné směrování dat. Mezi základní operace patří připojení resp. odpojení do instance simulátoru. Dále pak existuje možnost odeslat obrazová data pouze do jedné konkrétní instance. Všechny požadavky jsou pokryty. Ukázka 4.11 zachycuje metodu `JoinCanvasSession`, která slouží k připojení klienta a je realizována jeho přidáním do skupiny podle identifikátoru instance. Naopak ten kdo chce data odeslat použije metodu `UpdateClientCanvas`, pomocí které jsou data odeslána do požadované instance podle skupiny do které se klient připojil. Tyto operace jsou na sobě nezávislé a připojit se lze do jakékoliv skupiny i když do ní nikdo data neposílá. Stejně tak můžou být data posílána do prázdné skupiny.

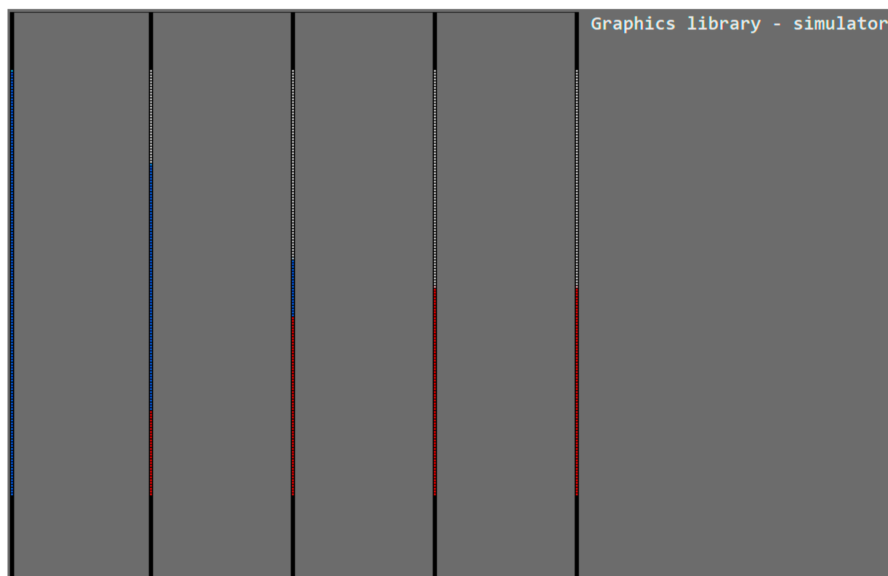
Součástí simulátoru je i klient, ten je realizován jako čistě HTML5 aplikace a je poskytována v rámci stejného prostředí jako SignalR. Jde pouze o odeslání jednoduché statické stránky, která obsahuje reference na Javascript a veškerá logika se dále odehrává ve webovém prohlížeči uživatele simulátoru. Komunikace se SignalR hubem je prováděna díky knihovně `aspnet/signalr` dostupné pomocí `npm` systému pro balíčky.

Při příchodu uživatele na stránku se simulátorem je zahájena komunikace s hubem pro zobrazování dat z plátna. Identifikace instance je určena buďto z URL fragmentu nebo se použije výchozí hodnota pro virtuální instanci reprezentující data z reálné fasády. Pro inicializaci je zavolána metoda `JoinCanvasSession`. Následně se jen vyčkává na přijetí dat po navázaném spojení. Existují dva druhy zpráv, které je nutné zpracovávat. První je inicializační zpráva, která obsahuje nastavení plátna jako je jeho velikost a umístění

```
public class CanvasHub : Hub<ICanvasClient>
{
  public async Task UpdateClientCanvas(
    CanvasMessageHeader<UpdateCanvasMessage> message)
  {
    await Clients.Group(message.SessionId).Update(message.Payload);
    if (!string.IsNullOrEmpty(message.NamedSession))
    {
      await Clients.Group(message.NamedSession)
        .Update(message.Payload);
    }
  }
  public async Task JoinCanvasSession(string sessionId)
  {
    await Groups.AddToGroupAsync(Context.ConnectionId, sessionId);
  }
}
```

Ukázka kódu 4.11: Část implementace SignalR hubu pro simulátor

jednotlivých sloupců. Druhou je hlavní datová zpráva, která obsahuje obrazová data ke zobrazení na simulátoru. Tím vnikne přesná reprezentace aktuálních dat na simulovaném plátně a výsledný stav je zachycen na obrázku 4.1.



Obrázek 4.1: Vizuální podoba simulátoru

4.3 Nasazení aplikace

Protože jsou zdrojové kódy aplikace uchovávané pomocí nástroje GitLab, kde se nachází Git repositář, je snadné využít i jeho dalších možností k realizaci nasazení aplikace do cílového prostředí a k automatizovanému spouštění testů. Tyto jednotlivé dílčí části budou popsány v následujících podsekcích.

4.3.1 Cílové prostředí

Nasazení bude zaměřené do prostředí OS linuxového typu, protože je to stále dostupnější varianta i díky struktuře ostatních alternativ, které nejsou tak časté mezi poskytovateli virtuálních strojů a nebo obecně kladou vyšší nároky. Protože je ale nasazení řešeno primárně pomocí technologie Docker, nemělo by to mít na aplikaci jako takovou vliv a nasazení by šlo s drobnými úpravami převést i jinam.

Jako konkrétní prostředí je použitý OS `Ubuntu 18.04.2 LTS`. Jde tedy o systém typu Linux a o distribuci vyvíjenou společností Canonical. Stroj samotný má jako klíčové parametry přidělené 2 jádra CPU a 4 GB operační paměti. Virtuální stroj běží v rámci privátního hostingu bigcloud.cz poskytnutého pro vývojové účely v rámci práce.

4.3.2 Continuous deployment

Při provedení změn ve větvi `master` dojde ke spuštění GitLab–CI procesu, který je definovaný v konfiguračním souboru `.gitlab-ci.yml`. V našem případě se jako první spustí proces zodpovědný za sestavení obrazů pro všechny komponenty aplikace a jejich nahrání do Docker registry také v rámci GitLab. Důležitou část procesu můžeme vidět v ukázce 4.12, kde je provedeno sestavení pomocí nástroje `docker-compose`, který díky již dříve popsané konfiguraci v `docker-compose.yml` umí dané obrazy sestavit. Označení obrazů je řízeno proměnnou prostředí `IMAGE_TAG`, která vzniká na základě názvu větve kde sestavení vzniklo v kombinaci s pořadím spuštěného procesu. Obrazy jsou například označené jako `master-31`. Výsledek sestavení je následně uložen do interního registry pomocí přístupových údajů dostupných v kontextu GitLab–CI agenta, který celou úlohu vykonává.

```
script:
  - docker-compose -f docker-compose.yml build
  - docker login -u gitlab-ci-token -p $CI_JOB_TOKEN $CI_REGISTRY
  - docker-compose -f docker-compose.yml push
```

Ukázka kódu 4.12: Část skriptu pro automatické sestavení obrazů

Následně po sestavení je ihned aktivován proces, který obrazy nasadí do cílového prostředí. Protože se jedná o vývojové verze je tento proces zcela automatický bez manuálního zásahu a pro produkční nasazení by bylo vyžadováno více interakce s uživatelem. Nasazovací proces, jehož část je opět možné vidět v ukázce 4.13, se skládá z vytvoření adresáře s konfigurací pro prostředí, kam bude aplikace nasazena. Klíčové je předat proměnnou `IMAGE_TAG`, díky které cílový stroj pozná, jakou verzi aplikace má spustit. Jako další nesmí samozřejmě chybět ani URL pro registry, kde se obrazy nachází, předaná pomocí `DOCKER_REGISTRY`. Součástí je i konfigurace celé aplikace popsané v `docker-compose.yml`. Následné je celá složka nakopírována na cílový stroj pomocí nástroje `scp`. V tento okamžik je vše připraveno a chybí už jen pomocí protokolu SSH vzdálenému serveru přikázat, aby se přihlásil k registry pomocí aktuálně předaných přihlašovacích údajů, stáhl všechny požadované obrazy podle aktuální verze a spustil aplikaci. V případě že aplikace již běží je díky nástroji `docker-compose` zajištěno že dojde pouze k aktualizaci změněných komponent a zbytek jako například perzistentní součásti zůstanou zachovány.

```
script:
  - mkdir ./deploy &&
    echo "IMAGE_TAG=$IMAGE_TAG" > ./deploy/.env &&
    echo "DOCKER_REGISTRY=$DOCKER_REGISTRY" >> ./deploy/.env
  - cp docker-compose.yml docker-compose.override.yml ./deploy/
  - 'scp -r ./deploy gitlab-ci@$DEPLOY_DEV_HOST:'
  - ssh gitlab-ci@$DEPLOY_DEV_HOST "docker login $CI_REGISTRY \
    -u $CI_DEPLOY_USER -p $CI_DEPLOY_PASSWORD"
  - ssh gitlab-ci@$DEPLOY_DEV_HOST "cd ./deploy \
    && docker-compose pull && docker-compose up -d"
```

Ukázka kódu 4.13: Část skriptu pro automatické nasazení aplikace

4.3.3 Continuous Integration

Vynucení tohoto procesu je docíleno pomocí nastavení GitLab projektu tak, že hlavní větev `master` je chráněná proti změnám. Tím je vývojář donucen své změny vytvářet v oddělených větvích. Když chce své změny zahrnout do hlavní větve vytvoří požadavek a tím vznikne *merge request*. Podle nastavené politiky musí každý takový požadavek úspěšně projít procesem definovaným v GitLab–CI, aby mohl být dokončen. To poskytuje ideální místo pro začlenění automatizovaných testů. Čímž dojde k zamezení zanesení chybného kódu do projektu.

V konfiguračním souboru `.gitlab-ci.yml` je tedy definován proces pro otestování aplikace. Testování probíhá, jak je vidět v ukázce 4.14, pomocí speciální verze Docker obrazů pro testované komponenty. K ověření všech

testů pak následně dojde pouhým spuštěním daných obrazů. Pokud některý z testů selže, je celý proces označený jako neplatný a požadavek na zahrnutí změn je odmítnut dokud nedojde k opravení všech testů.

```
script:
  - docker build --target testrunner -f \
    src/Services/Canvas/Canvas.API/Dockerfile -t canvas-api:test .
  - docker build --target testrunner -f \
    src/Services/Graphics/Graphics.Core/Dockerfile \
    -t graphics-core:test .
  - docker run --rm canvas-api:test
  - docker run --rm graphics-core:test
only:
  - /^features\/.*$/
```

Ukázka kódu 4.14: Část skriptu pro automatické testování aplikace

Ke spouštění testů v Docker prostředí je použit více fázový build proces [25]. Díky tomu lze už na úrovni každé komponenty v jejím Dockerfile specifikovat speciální cíl, pomocí kterého se komponenta otestuje. V ukázce 4.15 je vidět definice takového cíle pro komponentu aplikačního rozhraní.

```
FROM build AS testrunner
WORKDIR /src/Services/Canvas/Canvas.UnitTests
ENTRYPOINT ["dotnet", "test"]
```

Ukázka kódu 4.15: Část Dockerfile pro spuštění testů

4.4 Použití knihovny

Jak již vychází z návrhu celého systému, pokud chce uživatel s grafickou knihovnou komunikovat, musí k tomu využít aplikačního rozhraní, které slouží jako hlavní vstupní bod. Jediným požadavkem je vlastnit aktivní klíč s přístupem ke knihovně. Ten může být v případě testovacího prostředí vygenerován kýmkoliv a nebo je uživateli předán administrátorem. Po té už je možné libovolně zobrazovat požadované objekty s výstupem na simulátor.

Informace uživatel získá primárně ze specifikace samotného rozhraní dostupné pomocí Swagger–UI a nebo hrubého OpenAPI JSON dokumentu. Podrobná uživatelská dokumentace je dostupná v podobě generovaných HTML stránek, které jsou dostupné na titulní stránce spolu s API. Dokumentace

je také součástí příloh této práce – konkrétně se jedná o adresář `doc`, který obsahuje její offline verzi.

4.4.1 Ukázka zobrazení objektu

Pro názornost si ukážeme konkrétní postup, který musí uživatel následovat, aby bez předchozích znalostí mohl zobrazit pomocí knihovny jednoduchý grafický objekt.

Jako první krok musí uživatel získat přístupový klíč k rozhraní. Předpokládáme, že ho ještě nemá a pokud by měl, tak tento krok přeskočí. Pro získání testovacího klíče je nutné povést požadavek v podobě:

```
curl -X POST "http://localhost:5000/api/v1/User/testToken"
```

Odpovědí bude jednoduše strukturovaný JSON obsahující jeden atribut `token`, který jsme potřebovali. A použijeme ho ve všech následujících požadavcích.

Druhým krokem bude zobrazení požadovaného objektu. Jako příklad budeme chtít zobrazit červený obdélník přibližně ve středu plátna. Půjde o další požadavek na aplikační rozhraní v podobě:

```
curl -X POST "http://localhost:5000/api/v1/Canvas/item" \
-H "Linky-Token: t8VmAwzBTiPEUa3xlse/VkOkTu/0+FpkvMpd/u4gqVo=" \
-H "Content-Type: application/json-patch+json"
```

Jako tělo požadavku je nutné uvést popis požadovaného objektu včetně všech vlastností. Lze odeslat pomocí předchozího příkazu při uvedení přepínače `-d`. Všechny dostupné objekty a jejich vlastnosti jsou též součástí uživatelské dokumentace.

```
{
  "type": "rectangle",
  "parameters": {
    "y": "80",
    "width": "100",
    "height": "30",
    "color": "#FFFF0000",
    "filled": "true"
  }
}
```

Výsledek je možné vidět v simulátoru, jehož URL lze získat pomocí následujícího požadavku (hodnota odpovědi v atributu `displayUrl`):

```
curl -X GET "http://localhost:5000/api/v1/Canvas/simulator" \
-H "Linky-Token: t8VmAwzBTiPEUa3xlse/VkOkTu/0+FpkvMpd/u4gqVo="
```

Kapitola 5

Testování

Pro zajištění určité požadované kvality kódu je nutné ho podrobit řadě testů. Tím bude zaručena především klíčová funkcionalita a bude zachována v čase. Vhodně zvolené testy mohou zabránit nechtěným efektům způsobených změnami kódu v budoucnosti. Testy by tedy měly být co nejvíce izolované od okolí, aby se v průběhu času automaticky nestaly neplatnými.

5.1 Unit testy

V našem případě je nutné otestovat minimálně klíčové komponenty pomocí jednotkových testů. Jde o testy s nejvyšším stupněm izolace a testováno je většinou na úrovni samotných tříd. Tedy bude pokryto aplikační rozhraní na servisní vrstvě a klíčové součásti grafického jádra. Datovou vrstvu není třeba těmito testy pokrývat, protože neobsahují žádnou logiku. Komponenta simulátoru je na tom podobě a slouží pouze k distribuci dat bez složité logiky.

Testy byly sestaveny na základě osvědčených postupů ze zdroje [26]. Jako testovací framework byl zvolen `xUnit`, který poskytuje všechny požadované funkce. Ostatní alternativy na tom byly hodně podobě a nenabízely nic navíc, co by se dalo využít v rámci projektu. Jako nástroj pro vytváření dočasných zástupných implementací pro potřeby testování posloužila knihovna `Mocq`. Jde o nejpopulárnější balíček toho typu podle statistik ve správci NuGet.

Přesná podoba testů je zachycena v ukázce 5.1. Skládá se standardně z metody, která reprezentuje jeden testovací scénář. Metoda musí je označena atributem `Theory`, protože se jedná o test na více vstupních datech, která jsou postupně dána atributy `InlineData` a určují parametry s jakými bude metoda testována. V první části dochází k připravení všech prerekvizit k testu a především o vytvoření *falešných* implementací závislostí testované jednotky. To je možné nadefinováním návratových hodnot pro konkrétní metody a jejich parametry. Jestli došlo ke správném zavolání lze pak zpětně verifikovat. V druhé části dochází k vytvoření instance samotné jednoty nad připravenými objekty

a zavolání testované funkcionality. Nakonec dojde k ověření požadované stavu výsledku a použitých *falešných* objektů.

```
[Theory]
[InlineData("hello-world", UserTypes.AdminManaged)]
[InlineData("hello-world", UserTypes.ManualUser)]
public async Task Get_user_status_by_type_success(string token,
  string userType)
{
  // Arrange
  userRepositoryMock.
    Setup(x => x.GetUser(It.Is<string>(t => t == token),
      It.Is<string>(t => t == userType)))
    .ReturnsAsync(new UserSession());
  // Act
  var userService = new UserService(userRepositoryMock.Object,
    capabilitiesOptionsMock.Object);
  var result = await userService.GetUserStatus(token, userType);
  // Assert
  Assert.NotNull(result);
}
```

Ukázka kódu 5.1: Ukázka testu pomocí frameworku xUnit

Stejným způsobem jsou otestovány veškeré další již zmíněné komponenty aplikace a lze do řešení snadno přidávat testy další, které budou automaticky spuštěny pomocí CD procesu. Konkrétně se jedná o následující rozdělení testů:

- Canvas.API – 25 testů,
- Graphics.Core – 7 testů.

Závěr

Cílem práce bylo analyzovat současný stav projektu světelné fasády Linky na ČVUT FEL. Dále analyzovat metody, jak zjednodušit možnosti zobrazování obrazových dat. Na základě toho navrhnout a implementovat prototyp řešení, které všechny detekované nedostatky odstraní s ohledem i na možné požadavky budoucí. Cílové řešení musí být tvořeno pomocí webového aplikačního rozhraní se strojově zpracovatelnou definicí a vhodně pokryté testy a uživatelskou dokumentací.

V práci bylo podrobně zanalyzováno jakým způsobem projekt funguje a z jakých se skládá komponent. Na základě nalezených nedostatků byly specifikovány funkční požadavky a na nich založen návrh nové komponenty rozšiřující současné řešení. Vybrány byly na základě analýzy nejvhodnější technologie a postupy pro realizaci. Nakonec vnikl prototyp grafické knihovny, která uživatelům umožní požadovaný přístup k ovládání světelné fasády. Implementován byl i základ simulátoru, který umožňuje uživatelům náhled v reálném čase. Řešení je realizováno pomocí platformě nezávislé technologie .NET Core a celé řešení je navíc zabalené do sady Docker kontejnerů, takže byly splněny i požadavky na formu řešení. Díky téměř nativní podpoře ve zvolených technologiích obsahuje rozhraní kompletní OpenAPI specifikaci. Součástí je i uživatelská dokumentace pokrývající veškeré dostupné možnosti knihovny. Kvalita je zajištěna pomocí jednotkových testů, které jsou součástí automatizovaného procesu při interakci vývojáře se zdrojovým kódem. Ve výsledku tedy byly splněny všechny body zadání.

Budoucím využitím práce může být její nasazení do ostrého prostředí a po otestování mezi cílovými uživateli její používání jako alternativy k současnému řešení. Podle potřeb by bylo možné přidávat další grafické objekty a efekty díky snadné rozšiřitelnosti. Je připravena i integrace s dalšími komponentami systému určeným především pro správu. Díky OpenAPI specifikaci lze vygenerovat a zpřístupnit knihovny pro konkrétní programovací jazyky a tím se přiblížit uživatelům ještě více.

Seznam literatury

1. *ČVUT FEL - Linky* [online]. ČVUT FEL IoT Workshop, © 2019 [cit. 2019-04-20]. Dostupné z: https://linky.fel.cvut.cz.
2. *Linky API* [online]. ČVUT FEL IIM, © 2018 [cit. 2018-12-10]. Dostupné z: https://service.iim.cz/linkyapi.
3. KIRSCHNER, Filip. *Systém řízení světelné instalace LINKY*. Praha, 2017. Diplomová práce. České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra počítačové grafiky a interakce. Vedoucí práce Roman Berka.
4. *Introduction to ASP.NET Core* [online]. Microsoft Corporation, © 2019 [cit. 2019-04-12]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/?view=aspnetcore-2.1.
5. *.NET architectural components* [online]. Microsoft Corporation, © 2019 [cit. 2019-04-12]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/standard/components.
6. *Introduction to the C# Language and the .NET Framework* [online]. Microsoft Corporation, © 2015 [cit. 2019-04-16]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/csharp/getting-started/introduction-to-the-csharp-language-and-the-net-framework.
7. *Developer Survey Results: 2018* [online]. Stack Overflow, © 2018 [cit. 2019-05-11]. Dostupné z: https://insights.stackoverflow.com/survey/2018.
8. *Introduction to Redis* [online]. © 2019 [cit. 2019-04-17]. Dostupné z: https://redis.io/topics/introduction.
9. *OpenAPI Specification: Version 3.0.0* [online]. OpenAPI Initiative, © 2019 [cit. 2019-04-20]. Dostupné z: https://github.com/OAI/OpenAPI-Specification/blob/771adff68185c563118c4649efeb98767d71c404/versions/3.0.0.md.

10. What Is the Difference Between Swagger and OpenAPI? *SwaggerBlog* [online]. 2017 [cit. 2019-04-20]. Dostupné z: https://swagger.io/blog/api-strategy/difference-between-swagger-and-openapi.

11. *Swagger UI* [online]. © 2019 [cit. 2019-04-20]. Dostupné z: https://swagger.io/tools/swagger-ui.

12. Swagger Codegen. *GitHub, Inc.* [online] [cit. 2019-04-20]. Dostupné z: https://github.com/swagger-api/swagger-codegen.

13. *Introduction to ASP.NET Core SignalR* [online]. 2018 [cit. 2019-04-21]. Dostupné z: https://docs.microsoft.com/en-us/aspnet/core/signalr/introduction.

14. ŽÁRA, Jiří. *Moderní počítačová grafika*. 2. vydání. Brno: Computer Press, 2004. ISBN 80-251-0454-0.

15. *Welcome to the Visual Studio IDE: Visual Studio 2017* [online]. 2019 [cit. 2019-04-24]. Dostupné z: https://docs.microsoft.com/en-us/visualstudio/get-started/visual-studio-ide?view=vs-2017.

16. *GitLab Documentation: User documentation* [online]. © 2019 [cit. 2019-05-11]. Dostupné z: https://docs.gitlab.com/ee/user/index.html.

17. *Docker usage report* [online]. Sysdig, 2018 [cit. 2019-05-11]. Dostupné z: https://sysdig.com/blog/2018-docker-usage-report.

18. *Docker overview* [online]. © 2019 [cit. 2019-05-11]. Dostupné z: https://docs.docker.com/engine/docker-overview.

19. *Overview of Docker Compose* [online]. © 2019 [cit. 2019-05-11]. Dostupné z: https://docs.docker.com/compose/overview.

20. MLEJNEK, Jiří. *Softwarové inženýrství: Analýza a sběr požadavků* [online]. 2018 [cit. 2019-05-11]. Dostupné z: https://moodle.fit.cvut.cz/pluginfile.php/88280/mod_resource/content/3/03.prednaska.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém médiu].

21. MLEJNEK, Jiří. *Softwarové inženýrství: Analýza problémové domény* [online]. 2018 [cit. 2019-05-11]. Dostupné z: https://moodle.fit.cvut.cz/pluginfile.php/88283/mod_resource/content/3/04.prednaska.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém médiu].

22. *UML Deployment Diagrams* [online]. © 2018 [cit. 2019-05-06]. Dostupné z: https://www.uml-diagrams.org/deployment-diagrams.html.

23. MLEJNEK, Jiří. *Softwarové inženýrství: Modelování obchodních procesů* [online]. 2018 [cit. 2019-05-11]. Dostupné z: https://moodle.fit.cvut.cz/pluginfile.php/88277/mod_resource/content/2/02.prednaska.pdf [Soubor přístupný po přihlášení do sítě ČVUT – kopie souboru uložena na přiloženém médiu].

24. *What is a Data Flow Diagram* [online]. Lucidchart, © 2019 [cit. 2019-04-30]. Dostupné z: https://www.lucidchart.com/pages/data-flow-diagram.
25. *Use multi-stage builds* [online]. © 2019 [cit. 2019-05-11]. Dostupné z: https://docs.docker.com/develop/develop-images/multistage-build.
26. *Unit testing best practices with .NET Core and .NET Standard* [online]. Microsoft Corporation, 2018 [cit. 2019-05-12]. Dostupné z: https://docs.microsoft.com/en-us/dotnet/core/testing/unit-testing-best-practices.

Příloha A

Seznam použitých zkratek

API Application Programming Interface

CD Continuous Deployment

CI Continuous Integration

CIL Common Intermediate Language

CLR Common Language Runtime

CRUD Create, Read, Update, Delete

DI Dependency injection

DTO Data transfer object

FPS Frames per second

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IDE Integrated Development Environment

JSON JavaScript Object Notation

LED light–emitting diode

MVC Model–view–controller

OCI Open Container Initiative

OS Operating system

PNG Portable Network Graphics

REST Representational State Transfer

RGB Red–green–blue

RGBW Red–green–blue–white

RPC Remote procedure call

SSH Secure Shell

TCP Transmission Control Protocol

URL Uniform Resource Locator

YAML YAML Ain't Markup Language

Příloha B

Ukázka Swagger UI

V této ukázce je zachyceno grafické rozhraní, které je součástí nástroje Swagger a je vygenerované na základě specifikace pro aplikační rozhraní grafické knihovny.

swagger Select a spec V1

Graphics library API 1.0 1.0

/swagger/v1/swagger.json

Graphics library to allow drawing of specific shapes to Linky facade through REST API.

Contact Jiří Košata

Authorize

Admin

GET /api/v1/Admin/environment/{envId} Gets state of specified environment.

Parameters — Try it out

Name	Description
envId * required integer($int32) (path)	Id of environment.

Responses — Response content type: text/plain

Code	Description
200	Success Example Value | Model { "status": "OK", "simulatorUrl": "string" }
401	Unauthorized
403	Forbidden
404	No environment with such ID exists.

POST /api/v1/Admin/environment/{envId} Set specified environment to required state.

DELETE /api/v1/Admin/environment/{envId} Deletes specified environment.

Canvas

GET /api/v1/Canvas Gets information about current canvas session.

PATCH /api/v1/Canvas Updates canvas session settings.

GET /api/v1/Canvas/items Gets all items in current canvas session.

DELETE /api/v1/Canvas/items Deletes all items in canvas session.

GET /api/v1/Canvas/item/{id} Gets one specific item from canvas session.

DELETE /api/v1/Canvas/item/{id} Deletes specific canvas item.

POST /api/v1/Canvas/item Create or update sent canvas item.

GET /api/v1/Canvas/simulator Gets information about simulator instance for current session.

POST /api/v1/Canvas/image Display image on canvas and remove all other items.

Obrázek B.1: Swagger UI pro API grafické knihovny

Příloha C

Ukázka OpenAPI specifikace

```
{
  "swagger": "2.0",
  "info": {
      "version": "1.0",
      "title": "Graphics library API 1.0"
  },
  "paths": {
    "/api/v1/Admin/environment/{envId}": {
      "get": {
        "tags": [ "Admin" ],
        "summary": "Gets state of specified environment.",
        "operationId": "GetApplication",
        "consumes": [],
        "produces": [ "text/json" ],
        "parameters": [
        {
            "name": "envId",
            "in": "path",
            "description": "Id of environment.",
            "required": true,
            "type": "integer",
            "format": "int32"
          }
        ],
        "responses": {
          "404": { "description": "No environment with such ID." },
          "200": {
            "description": "Success",
            "schema": {
              "$ref": "#/definitions/EnvironmentStateResponse" }
          }
          },
          "security": [ { "linky-admin": [] } ]
        }
      }
  },
  "definitions": {
      "EnvironmentStateResponse": {
          "type": "object",
          "properties": {
              "status": {
                  "description": "Environment status.",
                  "enum": [ "OK", "ERROR" ],
                  "type": "string" }}}}}
```

Ukázka kódu C.1: Část OpenAPI specifikace pro API grafické knihovny

Příloha D

Obsah přiložené SD karty

```
readme.txt ........................ stručný popis obsahu SD
doc ............................... dokumentace ve formátu HTML
src
  impl ............................ zdrojové kódy implementace
  thesis .......................... zdrojová forma práce ve formátu LaTeX
text .............................. text práce
  BP_kosatjir.pdf ................. text práce ve formátu PDF
  ref ............................. offline verze některých referencí
```