Diplomová práce

České
vysoké
učení technické
v Praze

F3

Fakulta elektrotechnická
Katedra počítačů

Vícekamerový systém pro sledování a vysílání

Diplomová práce

Filip Toman

Vedoucí: Ing. Ivo Malý Ph.D.
Obor: Otevřená informatika
Leden 2024

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Toman** Jméno: **Filip** Osobní číslo: **483857**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Vícekamerový systém pro sledování a vysílání

Název diplomové práce anglicky:

Multi-camera monitoring and streaming system

Pokyny pro vypracování:

Analyzujte současné systémy pro správu a vysílání (streaming) videa. Zaměřte se na systémy pracující jak s více paralelními stopami (stream) tak i na vícekamerové vysílání. Na základě analýzy navrhněte systém pro přehrávání videí monitorujících určitou oblast více zdroji, např. pro účely sportovního přenosu, uživatelského testování mimo laboratoř nebo komplexní přednášky. Dále navrhněte přehrávač, který umožní na základě různých spouštěčů (trigger) přepínat mezi zdroji nebo sledovat určitou osobu mezi jednotlivými zdroji. Na základě návrhu implementujte jednotlivé moduly systému. Výsledný systém otestujte na alespoň 3 různých scénářích záznamu a přehrávání dat.

Seznam doporučené literatury:

RICHARDS, Paul William. The Unofficial Guide to Open Broadcaster Software.
KELLY, Philip, et al. Automatic camera selection for activity monitoring in a multi-camera system for tennis. In: 2009 Third ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC). IEEE, 2009. p. 1-8.
RODRIGUEZ-GIL, Luis, et al. Interactive live-streaming technologies and approaches for web-based applications. Multimedia Tools and Applications, 2018, 77: 6471-6502.

Jméno a pracoviště vedoucí(ho) diplomové práce:

Ing. Ivo Malý, Ph.D. katedra počítačové grafiky a interakce FEL

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **14.08.2023** Termín odevzdání diplomové práce: **09.01.2024**

Platnost zadání diplomové práce: **16.02.2025**

Ing. Ivo Malý, Ph.D.
podpis vedoucí(ho) práce

podpis vedoucí(ho) ústavu/katedry

prof. Mgr. Petr Páta, Ph.D.
podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

.

Datum převzetí zadání

Podpis studenta

© ČVUT v Praze, Design: ČVUT v Praze, VIC

Poděkování

Chtěl bych poděkovat zejména vedoucímu této práce, Ing. Ivu Malému Ph.D., díky kterému tento projekt existuje ve své nynější formě a který mi ho během celé naší spolupráce pomáhal rozvíjet. Kromě toho bych chtěl poděkovat také svému kolegovi Alexovi Nguyenovi, který mi během vývoje mnohokrát poradil s technickou stránkou implementace, a Vítu Říhovi, který mi pomáhal během psaní této práce. V neposlední řadě bych chtěl také poděkovat členům své rodiny za jejich stálou podporu v průběhu celého studia.

Prohlášení

Prohlašuji, že jsem práci vypracoval samostatně a použil jsem pouze podklady uvedené v přiloženém seznamu. Nemám závažný důvod proti užití tohoto školního díla ve smyslu §60 Zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon).

Abstrakt

Cílem tohoto projektu bylo navrhnutí a implementace aplikace pro správu a režii živých přenosů v prostředí webového prohlížeče se zaměřením na tvorbu vysoce přizpůsobitelného a ergonomického uživatelského rozhraní. Vytvořená aplikace tak nabízí možnosti pro konfiguraci vstupních multimediálních stop a následný lineární střih těchto stop v prostředí živého studia. Sestříhaný živý přenos pak mohou koncoví diváci sledovat přímo ve vestavěném přehrávači této aplikace nebo lze vysílat externí streamovací platformy jako YouTube nebo Twitch. Prostředí živého studia také nabízí pokročilé možnosti pro monitorování vstupních stop a umožňuje spojení několika kamerových úhlů do jediného souhrnného pohledu, ve kterém lze celý snímaný prostor sledovat z ptačí perspektivy.

Klíčová slova: živý přenos, livestreaming, lineární střih, web, webový prohlížeč, RTMP, WebRTC, HTTP Live Streaming, HLS, TypeScript, WebGL, ffmpeg, GStreamer, GraphQL, React, NestJS, Next.js, React Server Components

Vedoucí: Ing. Ivo Malý Ph.D.

Abstract

The goal of this project was to design and implement an application for the management and live editing of live broadcasts, running entirely inside a web browser, with a specific focus on creating a highly customizable and ergonomic user interface. As such, the application allows its user to configure the incoming multimedia tracks and to then linearly edit these track to produce the resulting live broadcast. The outgoing broadcast can be viewed inside the inbuilt player of this application itself or can be delivered to external streaming platforms such as YouTube or Twitch. The live studio environment also offers advanced options for the monitoring of the incoming tracks and is capable of merging multiple video feeds from different camera angles into a singular view where the director can observe the entire broadcast area from a top-down perspective.

Keywords: live broadcast, livestreaming, linear editing, web, web browser, RTMP, WebRTC, HTTP Live Streaming, HLS, TypeScript, WebGL, ffmpeg, GStreamer, GraphQL, React, NestJS, Next.js, React Server Components

Title translation: Multi-camera monitoring and streaming system — Master's thesis

Obsah

1 Popis projektu 1

1.1 Přehled 1

1.2 Studio 1

1.3 Floorplan 3

1.4 Editor 4

1.5 Player 5

1.6 Existující řešení 5

2 Návrh 7

2.1 Funkční požadavky 7

2.2 Případy užití 10

2.2.1 Vytvoření uživatelského účtu 10

2.2.2 Přidání zdroje 11

2.2.3 Přidání výstupu 11

2.2.4 Nastavení klávesové zkratky 12

2.2.5 Tvorba layoutu 13

2.2.6 Úprava Floorplan náhledu 15

2.2.7 Export záznamu 17

2.2.8 Odeslání zprávy v chatu 18

3 Uživatelské rozhraní 19

3.1 Login / Signup 19

3.2 Studio 19

3.2.1 Postranní panel 20

3.2.2 Layout 22

3.3 Floorplan Editor 22

3.4 Editor 24

3.4.1 Průzkumník 24

3.4.2 Náhled 25

3.4.3 Časová osa 26

3.5 Player 27

3.6 Settings 28

4 Technologické řešení 31

4.1 Gateway 32

4.2 API server 32

4.2.1 Framework 32

4.2.2 Jazyk 33

4.2.3 Databáze 33

4.2.4 Autentizace 34

4.3 Media server 35

4.4 Frontend server 35

4.4.1 Framework 35

4.4.2 Typování 37

4.4.3 Multimédia 37

4.4.4 State Management 38

4.4.5 Uživatelské rozhraní 39

5 Implementace **41**

5.1 Gateway 42

5.2 Main DB 42

5.3 API 43

5.3.1 Moduly 43

5.3.2 Factory providers 43

5.3.3 Entity 45

5.3.4 Middleware 46

5.3.5 Guards 47

5.3.6 DTOs 48

5.3.7 Resolvery 48

5.3.8 Controllery 51

5.3.9 Gateways 52

5.3.10 Services 54

5.4 Streamer 59

5.5 Client 62

5.5.1 Routing 62

5.5.2 Middleware 63

5.5.3 Skupina NoAuth 64

5.5.4 Skupina Main 66

5.5.5 Studio - Multimédia 69

6 Testování **93**

6.1 Scénáře 94

6.1.1 Vytvoření účtu 94

6.1.2 Vytvoření zdroje 94

6.1.3 Správa layoutu 95

6.1.4 Mixování zvuku 96

6.1.5 Spuštění přenosu 96

6.1.6 Tvorba nového layoutu 97

6.1.7 Tvorba Floorplan pohledu 98

6.1.8 Tvorba destinací 99

6.1.9 Klávesové zkratky 100

6.1.10 Audio monitoring 100

6.1.11 Úprava profilu 101

7 Závěr **103**

Literatura **105**

A Nasazení aplikace **111**

Obrázky

1.1 Původní pohled kamery 3
1.2 Promítnutý obraz z pozice kamery 4
3.1 Sekce Login 20
3.2 Sekce Signup 21
3.3 Schématický nákres obrazovky Studio 23
3.4 Obrazovka Studio 24
3.5 Sekce Layouts 25
3.6 Select Source menu 25
3.7 Keybind menu 26
3.8 Obrazovka Floorplan Editor . . . 27
3.9 Schématický nákres obrazovky Editor 28
3.10 Obrazovka Editor 29
3.11 Obrazovka Player 30
3.12 Obrazovka Settings 30
4.1 Vizualizace softwarové architektury 31
5.1 Graf Docker Compose konfigurace 76
5.2 Class diagram MailTransportProvider 76
5.3 Class diagram RedisProvider . . . 77
5.4 Class diagram MediasoupProvider 77
5.5 Class diagram třídy ImageUtilProvider 77
5.6 Class diagram třídy UserEntity . 77
5.7 Class diagram třídy SourceEntity 78
5.8 Class diagram třídy AuthGuard 78
5.9 Class diagram třídy GetUserGuard 78
5.10 Schéma rozhraní GraphQL 79
5.11 Class diagram třídy AuthResolver 80
5.12 Class diagram třídy UserResolver 80
5.13 Class diagram třídy SourceResolver 81
5.14 Class diagram třídy StreamResolver 81
5.15 Class diagram třídy AppGateway 82
5.16 Class diagram třídy ChannelGateway 82
5.17 Class diagram třídy AuthService 83
5.18 Class diagram třídy UserService 83
5.19 Class diagram třídy SourceService 83
5.20 Class diagram třídy MailService 84
5.21 E-mailová šablona pro ověření účtu 84
5.22 E-mailová šablona pro resetování hesla 84
5.23 Class diagram třídy SocketService 85

5.24 Class diagram třídy LiveService 85

5.25 Class diagram třídy MediaService 86

5.26 Class diagram třídy StreamerService 86

5.27 Graf GStreamer pipeline 87

5.28 Komponentová struktura aplikace 87

5.29 Souborová hierarchie složky `app` 88

5.30 Hlavičky komponenty `<PaneView>` 89

5.31 Filler element komponenty `<PaneView>` 90

5.32 Seskupené hlavičky komponenty `<PaneView>` 91

5.33 Výsledné uživatelské rozhraní ukázkového layoutu 91

5.34 Stromová reprezentace ukázkového layoutu 92

6.1 Požadovaný layout scénáře ”Správa layout” 95

6.2 Limitace správy oken 96

6.3 Požadovaný layout scénáře ”Tvorba nového layout” 98

6.4 Previzualizace kontrolních bodů pro škálování vrstev 99

6.5 Požadovaný layout scénáře ”Audio monitoring” 101

Tabulky

5.1 Schéma entity UserEntity 46

5.2 Schéma entity SourceEntity 46

Kapitola 1

Popis projektu

1.1 Přehled

V rámci tohoto projektu jsem se chtěl zaměřit na problematiku živých videopřenosů na internetu a konkrétně prozkoumat možnosti pro jejich správu a režii přímo v rozhraní webové aplikace bez nutnosti instalace dedikovaného desktopového softwaru. Zejména jsem pak cílil na správu multikamerových přenosů, kde systém v reálném čase přijímá množství audiovizuálních přenosů z mnoha různých zdrojů (typicky několika různých kamer) a režisérovy přenosu poskytuje agregovaný přehled o všech právě vysílajících zdrojích a dává mu možnost přepínat mezi tím, který ze zdrojů je právě viditelný koncovému divákovi livestreamu.

Mým cílem tak bylo vytvořit ucelenou webovou platformu, která obsáhne veškeré technologické aspekty potřebné pro livestreaming, od příjmu (ingest) nezpracovaných audiovizuálních dat přímo z jejich zdrojů (např. kamer), přes monitorování a režii výsledného přenosu, až pro archivaci a doručení (delivery) konečného zpracovaného obsahu. V rámci archivace by měl systém také nabídnou možnosti nad rámec pouhého exportu hotového záznamu proběhlého přenosu a měl by umožnit i dodatečný nelineární střih zdrojových přenosů daného záznamu a následný export této upravené verze.

1.2 Studio

Obrazovka Studio je jádrem celé platformy a slouží k monitorování a režii livestreamů. Právě v této sekci musí uživatel nejprve nakonfigurovat veškeré vstupy a výstupy, které bude jeho požadovaný přenos zahrnovat. Vstupy

mohou být libovolné audiovizuální přenosy na protokolu RTMP, které mohou přicházet například přímo od připojených kamer nebo od jiných streamovacích serverů. Mezi těmito vstupy bude moci režisér během přenosu přepínat a případně bude moci také kombinovat jejich audio stopy. Výstupy jsou pak všechny cílové destinace, kam budou data výsledného zpracovaného přenosu putovat. V základu bude podporované odesílání dat na libovolný RTMP endpoint, vysílání na význačné streamovací platformy jako YouTube nebo Twitch a streamování přímo do vestavěného přehrávače této platformy (viz sekce 1.5).

Vzhledem k tomu, že uživatelské potřeby pro vysílání každého živého přenosu jsou velice závislé na konkrétním vysílaném obsahu a mohou být extrémně rozdílné pro různé uživatele, tak bylo zřejmé, že uživatelské prostředí studia bude muset nabídnout vysokou míru flexibility. Místo fixního rozložení UI prvků tak bude studio obsahovat kompletně přizpůsobitelný systém oken a karet, kde si režisér bude moci veškeré právě monitorované zdroje libovolně rozložit do požadovaného layoutu, který mu poskytne optimální přehlednost pro konkrétní právě probíhající přenos. Vytvořené layouty si také bude možné uložit pro další budoucí použití.

Pro účely monitorování vyššího počtu zdrojů by mělo studio také podporovat souběžné spuštění na několika počítačích či obrazovkách, s rozdílným layoutem v každé z instancí. Systém tak musí podporovat komunikaci mezi všemi aktivními instancemi živého studia, aby stav zůstal konzistentní přes všechny karty prohlížeče, kde je aplikace právě otevřena. Pro přidanou flexibilitu by aplikace také měla podporovat tvorbu vlastních klávesových zkratek pro přepínání mezi zdroji a i zde bude synchronizace mezi instancemi klíčová, jelikož webové prohlížeče neposkytují API pro odposlech kláves mimo zaměřenou kartu prohlížeče a bude tak potřeba, aby aktivace zkratky v libovolné kartě způsobila reakci ve všech ostatních kartách.

Po stránce zvuku studio nabídne základní ovládání pro každý ze zdrojů s možnostmi Solo, Mute a Cue. Tlačítko Mute vypne zvuk pro daný zdroj a tlačítko Solo deaktivuje zvukovou stopu všech ostatních zdrojů kromě kromě toho vybraného. V základu může být Solo vždy aktivováno pouze u jednoho zdroje a pokud je Solo již aktivováno na jiném místě, tak se jeho aktivace u předchozího zdroje zruší při kliknutí. Toto chování lze přepnout, pokud uživatel při kliknutí na tlačítko Solo podrží klávesu Ctrl. Poté je možné aktivovat Solo u více zdrojů, které následně přehrávají zvuk dohromady. Studio také nabídne možnost pro automatické Solo právě aktivního zdroje, jehož obraz je zrovna vysílán koncovému divákovi. Tlačítko Cue je pak inspirováno např. mixážními pulty společnosti Pioneer a umožňuje monitorování audia určitých zdrojů, aniž by byly zvukové stopy slyšet v samotném živém přenosu [1]. Za tímto účelem je v rozhraní studia možné nakonfigurovat 2 různé výstupní audio zařízení: Master out a Monitor out. Zatímco na Master out výstupu může režisér slyšet finální výstupní audio, tak jak ho slyší koncoví diváci

přenosu, tak na výstupu Monitor out lze poslouchat zvuk ze všech zdrojů, které u sebe mají aktivované tlačítko Cue.

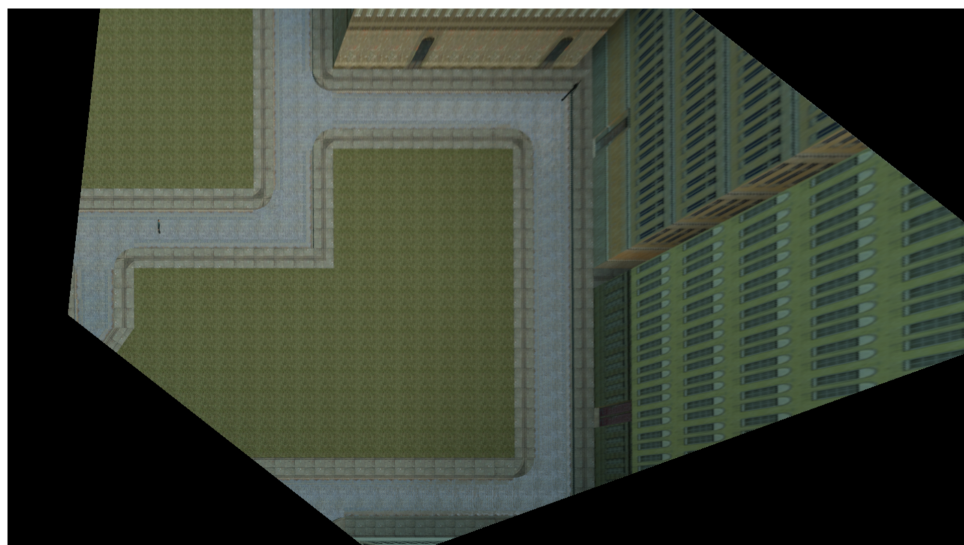
1.3 Floorplan

Obrazovka Floorplan je dodatečnou podsekcí vysílacího studia. Jejím účelem je poskytnout režisérovi přenosu ucelený přehled o prostoru, ve kterém se právě vysílaný přenos odehrává, a usnadnit mu tak monitorování probíhajících událostí. Funkce Floorplan je tak prospěšná zejména v případech, kdy jsou zdroji našeho přenosu statické kamery rozmístěné v jednotném prostoru jako je například. sportoviště, koncertní hala nebo výstavní plocha. Floorplan nám pak umožňuje zadat ke každé zdrojové kameře informace o jejím objektivu a pozici a orientaci ve 3D prostoru. Jako výsledek pak získáme vizualizaci celého kamerově pokrytého prostoru z ptačí perspektivy, a to tak, že aplikace promítne obraz každé kamery na virtuální podlahu.

Obrázek 1.1: Původní pohled kamery

Uživatel do tohoto pohledu bude moci také naimportovat libovolné obrázky a zadat jejich velikost a pozici v prostoru. To tak uživateli umožní vložit do Floorplan pohledu například schématický plán půdorysu budovy, který mu bude sloužit jako dodatečná prostorová reference k promítnutým obrazům z kamer.

Z hlediska organizace je Floorplan integrován uvnitř vysílacího studia a nastavenou vizualizaci je možné umístit libovolně do režisérovo layoutu, stejně jako v případě zdrojů. Pokud chce uživatel vytvořenou vizualizaci dalé upravit (např. vložit nové kamery), pak se tlačítkem “Edit”, umístěném v tomto náhledu, přesune na samostatnou obrazovku Floorplan Editor, na které



Obrázek 1.2: Promítnutý obraz z pozice kamery

může dále přidávat nebo upravovat kamery a obrázkové vrstvy. Po uložení změn se upravená vizualizace zobrazí ve studiovém náhledu.

1.4 Editor

O konečný export záznamů z proběhlých přenosů se stará sekce Editor. Uživatel zde v průzkumníku nalezne kompletní historii svých streamů a po rozkliknutí kteréhokoliv z nich získá nejen možnost si záznam prohlédnout a exportovat ho z aplikace ve formátu MP4, ale zejména také možnost obsah záznamu dále upravovat a dodatečně tak změnit jakákoliv režijní rozhodnutí, která proběhla během živého přenosu.

Úprava záznamů probíhá v rozhraní typickém pro nelineární střihové programy jako například Adobe Premiere, DaVinci Resolve nebo VEGAS Pro. Hlavním prvkem obrazovky tak bude časová osa, na které uživatel uvidí stopy pro veškeré audio a video zdroje [2]. Díky zaměření aplikace konkrétně na živé přenosy však tento editor na rozdíl od klasického střihového software vyžaduje, aby veškeré nahrané zdroje uvnitř záznamu zůstaly časově synchronizované a aby byl v jakoukoliv danou chvíli aktivní právě jeden z video zdrojů, stejně jako tomu je během živého přenosu. Výsledný sestřih tak nemůže obsahovat jakékoliv diskontinuity, časové posuny nebo přehrávat více video stop zároveň.

Z toho důvodu tak pro tento účel nedávalo smysl reprezentovat záznamy zdrojů ve formě klipů umístěných ve stopách, tak jako tomu dělá většina klasických video editorů. Místo toho je právě viditelný obsah ovládán pomocí klíčových snímků na envelope, která přepíná mezi zdroji. Oproti tomu, v

případě audio zdrojů je možné, aby v jeden moment hrálo více stop zároveň. Audio stopy jsou tak také ovládány pomocí envelopes, ale místo jediné sdílené pro všechny stopy má každá stopa svou vlastní envelope, která kontroluje, zda je daná stopa právě slyšitelná nebo ztlumená.

1.5 Player

Obrazovka Player obsahuje vestavěný přehrávač livestreamů, který lze nastavit jako jeden z možných výstupů pro přenos na obrazovce Studio. Přehrávač je vždy specifický pro konkrétní uživatelský profil a kromě právě probíhajícího streamu daného uživatele zde lze také nalézt základní informace o jeho profilu a živý chat pro diváky přenosu.

Chatové rozhraní kromě samotného odesílání a zobrazení zpráv nabídne i základní možnosti pro moderování obsahu. K těm bude mít přístup jednak samotný majitel vysílajícího profilu a jednak další uživatelské účty, kterým byla majitelem přiřazena role moderátora na obrazovce Nastavení. Dostupnými moderátorskými možnostmi budou permanentní zabanování vybraného uživatele v chatu a dočasný timeout, kterému po uběhnutí zvolené doby vyprší platnost.

1.6 Existující řešení

V dnešní době je profesionální režie živých přenosů v zásadě exkluzivní doménou desktopových aplikací. Zřejmě nejbližší webovou alternativou k navrhovanému systému by je platforma StreamYard, která také pracuje kompletně na straně prohlížeče a umožňuje multistreaming do více cílových destinací zároveň. Oproti navrhované platformě jsou však vlastnosti StreamYardu uzpůsobené spíše pro streamování videohovorů než pro správu arbitrárních obrazových přenosů [3]. Do přenosu je tak možné pozvat více hostů a každému z nich je umožněno streamovat zdroje přímo přístupné jejich webovému prohlížeči jako webkameru nebo obrazovku. StreamYard tak operuje na podobné bázi jako například Teams, Discord nebo Google Meet, pouze s možností přenášet výsledný obraz i na vnější streamovací platformy.

Více přesným ekvivalentem je v tomto případě projekt Studio Web Control od společnosti Vimeo, který narozdíl od StreamYardu poskytuje plnohodnotné produkční prostředí pro správu živých přenosů uvnitř webového prohlížeče. Avšak v tomto případě se nejedná o samostatnou streamovací platformu, ale pouze o webové rozhraní pro ovládání jejich desktopového software Livestream

Studio a finální zpracování a vysílání obsahu tak stále probíhá výhradně na počítači koncového uživatele [4].

Se svým projektem jsem chtěl cílit právě na více profesionálně zaměřenou stranu živých přenosů a hlavní inspirací pro funkcionalitu se tak staly programy jako Wirecast, vMix nebo i novější Multiview pohled v OBS, které jsou dnes průmyslovým standardem. Všechny tyto programy disponují právě přizpůsobitelným náhledem všech příchozích zdrojů, možnostmi pro monitorování audia a schopností přepínat mezi nimi pomocí konfigurovatelných klávesových zkratek (tvz. hotkeys) [5]. Z hlediska multiview pohledu však moje aplikace nabízí větší míru flexibility než většina ostatních řešení, jelikož Wirecast, OBS i StreamYard v tomto ohledu spoléhají pouze na několik předpřipravených layoutů. vMix oproti tomu možnost pro importování vlastních layoutů nabízí, nicméně organizaci zdrojů do více karet v rámci okna neumožňuje z programů žádný.

Oproti tomu, na straně vestavěného editoru existuje webových řešení poměrně velké množství jako například Clipchamp, Veed nebo Canva. Je však potřeba podotknout, že všechny tyto produkty se zaměřují na všeobecné nelineární editování, zatímco editor obsažený v mojí aplikaci je uzpůsobený konkrétně pro dodatečnou úpravu zaznamenaných přenosů a způsob fungování je tak poměrně odlišený. Oproti tomuto existujícímu softwaru tak editor mojí aplikace nereprezentuje střih ve formě mediálních klipů na stopách v časové ose, ale pomocí kontrolních envelopes, které mezi zdroji přepínají (viz sekce 1.4). Z hlediska uživatelského rozhraní je pak rozložení prvků nejvíce inspirováno výchozím layoutem desktopového editoru Adobe Premiere a fungování envelopes bylo převzato z programu VEGAS Pro.

Kapitola 2

Návrh

2.1 Funkční požadavky

Tento seznam popisuje základní požadavky, které musí aplikace splňovat, aby byla použitelná v praxi a mohlo dojít k jejímu nasazení do produkce:

- Uživatelské účty
  - Aplikace umožní vytváření uživatelských účtů
  - Aplikace ověří e-mailovou adresu nového uživatele odesláním verifikačního e-mailu
  - Aplikace umožní přihlášení a odhlášení z aplikace
  - Aplikace umožní změnu hesla
  - Aplikace umožní obnovení zapomenutého hesla odesláním resetovacího e-mailu
- Profil
  - Aplikace umožní úpravu profilových informací (popis, profilový obrázek, jméno)
  - Aplikace umožní přidávání a odebírání moderátorských účtů pro daný profil
- Studio
  - Stream
    - Aplikace umožní spuštění a ukončení přenosu
    - Aplikace umožní pojmenování přenosu pro účely vestavěného přehrávače a archivace

    - Aplikace bude v průběhu přenosu zobrazovat dosavadní trvání a počet aktuálních sledujících ve vestavěném přehrávači
  - Zdroje
    - Aplikace umožní přidávání a odebírání audiovizuálních zdrojů
    - Aplikace umožní pojmenování zdrojů
    - Aplikace umožní resetování streamovacícho klíče pro vybraný zdroj
    - Aplikace umožní drag-and-drop seřazení seznamu zdrojů
    - Aplikace umožní konfiguraci klávesových zkratek pro přepnutí aktivního zdroje
    - Aplikace zobrazí aktuální stav nakonfigurovaných zdrojů (online/offline)
  - Výstup
    - Aplikace umožní přidávání a odebírání výstupů
    - Aplikace umožní výstup přenosu na vestavěný přehrávač
    - Aplikace umožní výstup na libovolný RTMP server
    - Aplikace nabídne integraci pro výstup na populární streamovací platformy
    - Aplikace zobrazí náhled odchozího přenosu
  - Uživatelské rozhraní
    - Aplikace umožní horizontální a vertikální půlení oken
    - Aplikace umožní nastavit libovolnou proporcionální velikost mezi pravou a levou polovinou rozpůleného okna
    - Aplikace umožní přidávání karet do oken
    - Aplikace umožní drag-and-drop přesouvání karet mezi okny
    - Aplikace umožní drag-and-drop přiřazení zdroje z postranního panelu do vybrané karty
    - Aplikace nabídne možnost pro automatické zaostření karty právě aktivního zdroje
  - Audio
    - Aplikace umožní ztlumení audia vybraných zdrojů
    - Aplikace umožní “solo” pro audio vybraných zdrojů
    - Aplikace nabídne možnost pro automatické “solo” audia právě aktivního zdroje
    - Aplikace umožní konfiguraci 2 výstupních audio zařízení: Master Out a Monitor Out
    - Aplikace umožní poslech vybraných zdrojů na audio výstupu Monitor Output (možnost Cue)

- Floorplan editor

  - Aplikace umožní posouvání a zoom náhledu Floorplan

- Aplikace umožní přidávání, úpravu a odebírání kamerových a obrázkových vrstev
- Aplikace pro každou kameru nabídne nastavení zdroje, pozice a rotace v prostoru, korekce pro zkreslení čočky a náhledového obrázku
- Aplikace pro každý obrázek nabídne nastavení pozice a reálné velikosti v prostoru
- Aplikace umožní drop-and-drop seřazení vrstev
- Aplikace umožní skrytí vybraných vrstev
- Aplikace umožní uzamknutí ovládání vybraných vrstvy

- Editor
  - Aplikace umožní procházení, vyhledávání a řazení záznamů proběhlých přenosů
  - Aplikace umožní načtení, smazání, přejmenování a duplikaci záznamu
  - Aplikace umožní export aktuální verze záznamu ve formátu MP4
  - Aplikace umožní přehrávání a pozastavení záznamu
  - Aplikace nabídne ovládání posun na začátek a na konec záznamu a posun o snímek dopředu a zpět
  - Aplikace umožní libovolný posun kurzoru po časové ose
  - Aplikace umožní posouvání a zoom časové osy záznamu
  - Aplikace umožní vytváření klíčových snímků na časové ose pro přepnutí aktivní video stopy
  - Aplikace umožní vytváření klíčových snímků na časové ose pro ztlumení nebo aktivaci konkrétních audio stop
  - Aplikace umožní vynucené zobrazení vybrané video stopy
  - Aplikace umožní vynucené ztlumení nebo "solo" vybrané audio stopy
  - Aplikace umožní drag-and-drop seřazení audio a video stop

- Player
  - Aplikace umožní spuštění a pozastavení právě probíhajícího přenosu daného profilu
  - Aplikace zobrazí základní informace o probíhajícím přenosu a profilu vysílajícího uživatele (název přenosu, jméno profilu, popis profilu, profilový obrázek)
  - Aplikace zobrazí dosavadní trvání aktuálního přenosu a počet diváků
  - Aplikace přihlášeným uživatelům umožní odesílání real-time zpráv do chatu
  - Aplikace moderátorovi umožní permanentní zablokování uživatele v chatu
  - Aplikace moderátorovi umožní dát chatujícímu uživateli dočasný timeout na odesílání zpráv

2.2 Případy užití

Tato část popisuje hlavní procesy, které jsou používané při práci s aplikací:

2.2.1 Vytvoření uživatelského účtu

Aktér: Nepřihlášený uživatel

Vstupní podmínky: -

Hlavní scénář:

1. Uživatel naviguje na úvodní stránku aplikace
2. Systém zobrazí úvodní stránku s formulářem Login
3. Uživatel klikne nad přepínač Login / Signup nad formulářem
4. Systém skryje formulář Login a zobrazí formulář Signup
5. Uživatel do zobrazeného formuláře zadá své uživatelské jméno, heslo a svůj e-mail
6. Systém skryje formulář a odešle aktivační e-mail na zadanou e-mailovou adresu
7. Uživatel naviguje do své e-mailové schránky a v přijatém e-mailu klikne na tlačítko Aktivovat
8. Systém účet trvale aktivuje a zobrazí uživateli stránku se zprávou „Váš účet byl úspěšně aktivován“

Alternativní scénáře:

Neplatné údaje

6. Systém zobrazí modál s chybovou zprávou
7. Uživatel zavře modál

Návrat do bodu 5

Neaktivovaný účet

7. Systém smaže uživatelské údaje, pokud účet není aktivován do 24 hodin od vytvoření

Návrat do bodu 1

2.2.2 Přidání zdroje

Aktér: Přihlášený uživatel

Vstupní podmínky:

- Uživatel je na stránce Studio

Hlavní scénář:

1. Uživatel klikne na tlačítko Create Source v části Sources postranního panelu
2. Systém přidá novou kartu s názvem „Untitled Source“ do seznamu zdrojů
3. Uživatel klikne na ikonu tužky vedle názvu zdroje
4. Systém změní text názvu na editovatelné textové pole
5. Uživatel vyplní název zdroje a stiskne klávesu Enter
6. Systém skryje textové pole a na jeho místě zobrazí text s upraveným názvem
7. Uživatel zkopíruje vygenerovaný streamovací klíč do svého streamovacího software a začne vysílat na RTMP endpoint aplikace
8. Systém aktualizuje indikátor u karty zdroje na stav ONLINE a zobrazí obrazová data přenosu ve všech náhledových kartách zdroje

Alternativní scénáře:

Chyba přenosu

2. Systém ponechá indikátor u karty zdroje ve stavu OFFLINE

Návrat do bodu 7

2.2.3 Přidání výstupu

Aktér: Přihlášený uživatel

Vstupní podmínky:

- Uživatel je na stránce Studio

Hlavní scénář:

1. Uživatel klikne na tlačítko Create Destination v části Destinations postranního panelu
2. Systém přidá novou nevyplněnou kartu do seznamu výstupů
3. Uživatel klikne na dropdown menu Type u horního okraje karty výstupu
4. Systém zobrazí seznam všech podporovaných typů výstupů
5. Uživatel klikne na položku požadovaného typu
6. Systém skryje seznam typů a aktualizuje formulář uvnitř karty v závislosti na daném typu
7. Uživatel vyplní požadované informace pro daný výstup
8. Uživatel klikne na tlačítko Start Stream
9. Systém spustí stream a začne odesílat audiovizuální data přenosu na nakonfigurovaný výstup

Alternativní scénáře:

Neplatné údaje

2. Systém zobrazí modál s chybovou hláškou a stream zůstane vypnutý

Návrat do bodu 7

2.2.4 Nastavení klávesové zkratky

Aktér: Přihlášený uživatel

Vstupní podmínky:

- Uživatel je na stránce Studio

Hlavní scénář:

1. Uživatel klikne na tlačítko More u libovolné karty, uvnitř které je přiřazený vybraný zdroj, pro který chce uživatel nastavit klávesovou zkratku
2. Systém zobrazí dropdown menu karty
3. Uživatel klikne na položku Keybinds

4. Systém zobrazí menu pro tvorbu klávesových zkratek
5. Uživatel klikne na tlačítko Add Keybind
6. Systém do seznamu zkratek přidá novou položku s nápisem „Press keys...“
7. Uživatel stiskne klávesy požadované zkratky
8. Systém uvnitř přidané položky zobrazí popis stisknutých kláves a aktivuje používání této zkratky
9. Uživatel klikne do prostoru mimo menu
10. Systém skryje zobrazené menu

Alternativní scénáře: -

2.2.5 Tvorba layoutu

Aktér: Přihlášený uživatel

Vstupní podmínky:

- Uživatel je na stránce Studio

Hlavní scénář:

1. Uživatel zadá název layoutu do textového pole v sekci Layouts v horním navigačním panelu
2. Uživatel stiskne tlačítko Save na pravé straně sekce Layouts
3. Systém uloží nový layout a přidá ho seznamu existujících layoutů

Alternativní scénáře:

Přidání karty zdroje

2. Uživatel chytí kartu zdroje z postranního panelu a přetáhne jí do prostřední oblasti libovolného okna
3. Systém přidá novou kartu na konec lišty karet daného okna

Návrat do bodu 1

Přidání okna zdroje

2. Uživatel chytí kartu zdroje z postranního panelu a přetáhne jí k jedné ze stran existujícího okna
3. Systém rozpůlí existující okno a na zvolenou stranu přiřadí náhled vybraného zdroje

Návrat do bodu 1

Přidání karty (obecné)

2. Uživatel klikne na tlačítko + na liště karet
3. Systém vytvoří novou prázdnou kartu
4. Uživatel klikne na tlačítko More v tomto okně
5. Systém zobrazí dropdown menu
6. Uživatel vybere možnost Select Source
7. Systém zobrazí list dostupných náhledů
8. Uživatel klikne na požadovaný náhled
9. Systém přiřadí vybraný náhled k vytvořené kartě

Návrat do bodu 1

Přidání okna (obecné)

2. Uživatel klikne na tlačítko More ve existujícím okně
3. Systém zobrazí dropdown menu
4. Uživatel vybere možnost Horizontal Split nebo Vertical Split
5. Systém rozpůlí vybrané okno na dvě
6. Uživatel klikne na tlačítko More v nově vytvořeném okně
7. Systém zobrazí dropdown menu
8. Uživatel vybere možnost Select Source
9. Systém zobrazí list dostupných náhledů
10. Uživatel klikne na požadovaný náhled
11. Systém přiřadí vybraný náhled k první kartě vytvořeného okna

Návrat do bodu 1

Smazání karty

2. Uživatel klikne na tlačítko X vedle názvu vybrané karty

3. Systém kartu odebere z lišty karet daného okna

Návrat do bodu 1

Smazání okna

2. Uživatel klikne na tlačítko More uvnitř vybraného okna
3. Systém zobrazí dropdown menu
4. Uživatel vybere možnost Close All
5. Systém uzavře celé okno

Návrat do bodu 1

Změna velikosti okna

2. Uživatel chytí rozdělovací čáru mezi dvěma okny a přetáhne jí k jedné ze stran
3. Systém aktualizuje velikostní poměr mezi těmito dvěma okny

Návrat do bodu 1

Přesunutí karty

2. Uživatel chytí kartu z lišty karet vybraného okna a přetáhne jí do lišty karet jiného okna
3. Systém smaže kartu v původním okně a vytvoří novou ekvivalentní kartu v okně novém

Návrat do bodu 1

2.2.6 Úprava Floorplan náhledu

Aktér: Přihlášený uživatel

Vstupní podmínky:

- Uživatel je na stránce Studio
- Studiový layout obsahuje náhled Floorplan

Hlavní scénář:

1. Uživatel stiskne tlačítko Edit v náhledu Floorplan

2. Systém zobrazí obrazovku Floorplan Editor
3. Uživatel klikne na tlačítko Save
4. Systém uloží provedené změny
5. Uživatel klikne na tlačítko Back
6. Systém skryje obrazovku Floorplan Editor a vrátí uživatele na obrazovku Studio

Alternativní scénáře:

Přidání kamery

2. Uživatel vybere nástroj Add Camera z lišty nástrojů a klikne do editoru
3. Systém v místě stisku vytvoří novou kameru

Návrat do bodu 3

Přidání obrázku

2. Uživatel vybere nástroj Add Image z lišty nástrojů a klikne do editoru
3. Systém zobrazí systémové menu pro výběr obrazového souboru
4. Uživatel vybere zdrojový obrázek
5. Systém vloží obrazovou vrstvu na místo stisku

Návrat do bodu 3

Úprava vrstvy

2. Uživatel vybere nástroj Cursor a v okně editoru klikne na vybranou kameru nebo vybraný obrázek
3. Systém zobrazí možnosti vybrané vrstvy v sekci Options postranního panelu
4. Uživatel upraví nastavení vrstvy v postranním panelu
5. Systém aktualizuje náhled v závislosti na změnách

Návrat do bodu 3

2.2.7 Export záznamu

Aktér: Přihlášený uživatel

Vstupní podmínky:

- Uživatel je na stránce Editor

Hlavní scénář:

1. Uživatel vyhledá požadovaný záznam v průzkumníku a klikne na jeho kartu
2. Systém načte vybraný záznam do editoru a zobrazí jeho časovou osu
3. Uživatel stiskne tlačítko Export
4. Systém vyrenderuje hotový multimediální záznam a poté zobrazí systémové menu pro uložení souboru

Alternativní scénáře:

Střih

2. Uživatel zaměří časovou osu na cílový časový úsek a v místě střihu provede dvojklik na video envelope
3. Systém v místě stisku vytvoří klíčový snímek
4. Uživatel klíčový snímek přetáhne na cílovou video stopu
5. Systém zobrazí přechod envelope z původní video stopy na konečnou video stopu

Návrat do bodu 3

Úprava audio stopy

2. Uživatel zaměří časovou osu na cílový časový úsek a v místě úpravy provede dvojklik na audio envelope
3. Systém v místě stisku vytvoří klíčový snímek
4. Uživatel klíčový snímek přetáhne k horní nebo spodní hraně audio stopy
5. Systém zobrazí přechod envelope ze ztišené do slyšitelné nebo naopak

Návrat do bodu 3

2.2.8 Odeslání zprávy v chatu

Aktér: Přihlášený uživatel

Vstupní podmínky:

- Uživatel je na stránce Player daného přenosu

Hlavní scénář:

1. Uživatel napíše zprávu do textového pole u spodní hrany chatového panelu
2. Uživatel stiskne tlačítko Chat
3. Systém doručí zprávu všem divákům živého přenosu a zobrazí jí v chatovém panelu

Alternativní scénáře:

Neplatný vstup

2. Systém zobrazí modál s chybovou hláškou
3. Uživatel zavře modál

Návrat do bodu 1

Kapitola 3

Uživatelské rozhraní

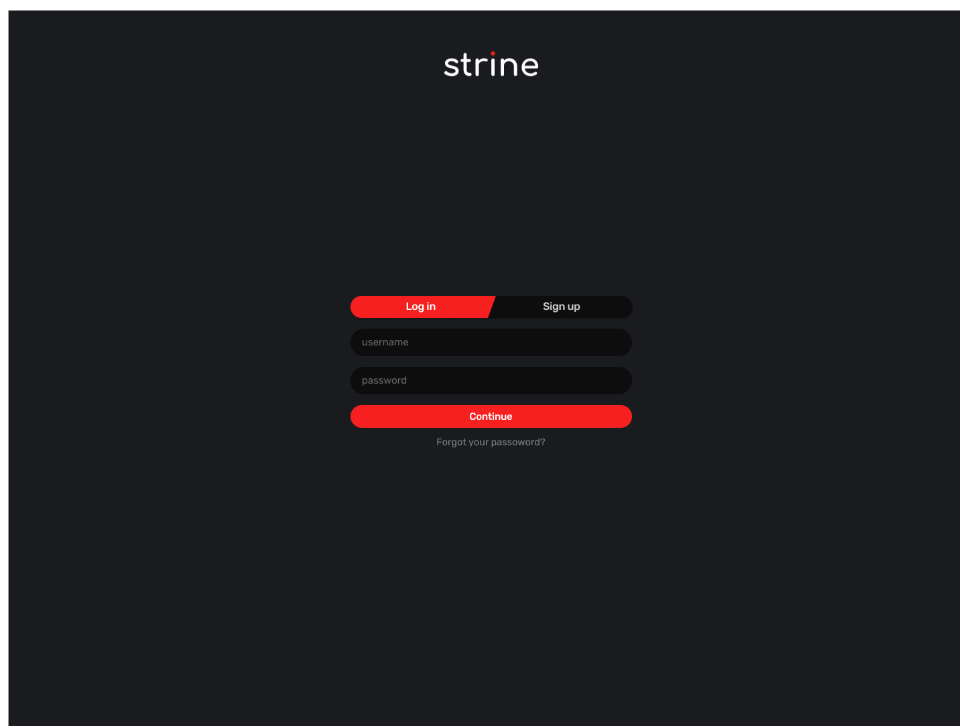
3.1 Login / Signup

Tato stránka je vstupním bodem celé aplikace a je jedinou, která neobsahuje horní navigační panel. Stránka je rozdělena do 2 sekcí, Login a Signup, mezi kterými lze navigovat pomocí velkého přepínače umístěného nad obsahem každé sekce. Samotný obsah je pak napozicován do úplného středu obrazovky a tím na sebe ihned upoutává návštěvníkovu pozornost. Obě sekce jsou tvořeny pouze základním formulářem pro přihlášení nebo registraci a uživateli představují vizuální jazyk celé platformy. Veškeré prvky formuláře tak obsahují zcela zaoblené rohy a drží se konzistentní barevné palety aplikace, která je tvořena červenou jako svou primární barvou, vyhrazenou pro důležité a interaktivní prvky, bílou a světlými odstíny šedé pro texty a sekundární prvky a tmavými odstíny šedé pro pozadí a neaktivní prvky.

Aby byl průchod nového uživatele co nejvíce usnadněn, tak je registrační formulář redukován pouze na ty nejdůležitější vstupní pole, která jsou absolutně klíčová pro tvorbu profilu. Aplikace tak snižuje bariéru pro připojení nového uživatele a další nekritické informace jako popis nebo profilový obrázek přenechává k dodatečnému nastavení až po vytvoření účtu.

3.2 Studio

Tato stránka slouží jako kontrolní centrum pro režii živých přenosů a její vzhled je do velké míry uživatelsky přizpůsobitelný. Rozhraní celé stránky je rozděleno do 2 hlavních částí: levého postranního panelu, který je vyhrazen k samotné konfiguraci livestreamu, a zbylého prostoru vpravo, který je ponechán



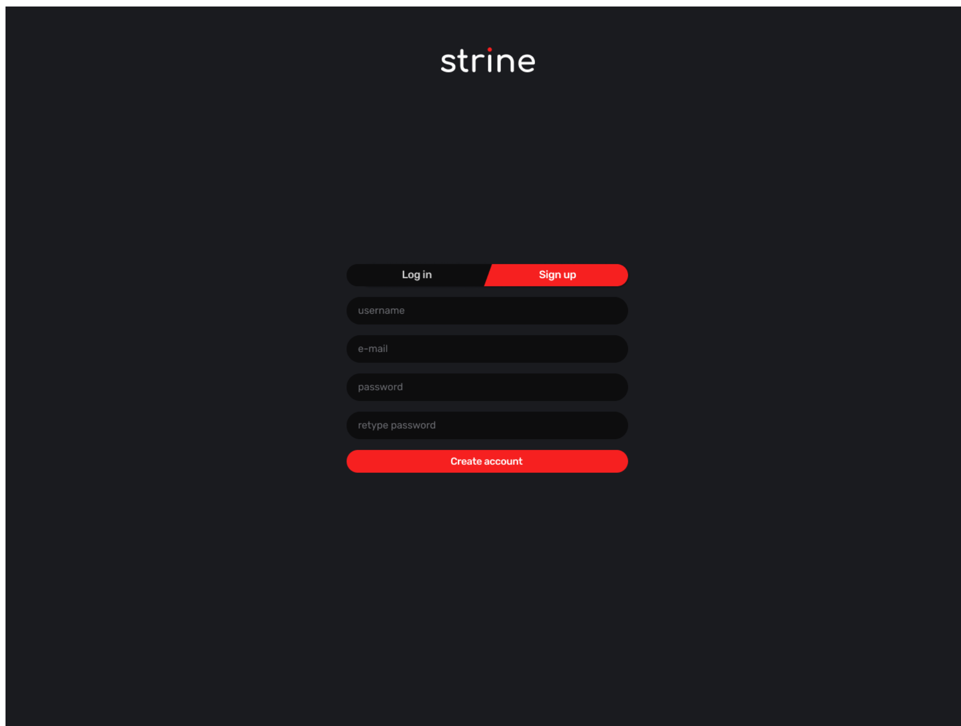
Obrázek 3.1: Sekce Login

pro tvorbu režisérského layoutu.

3.2.1 Postranní panel

První část postranního panelu se zabývá informacemi o právě probíhajícím přenosu. Lze zde tak nalézt velké tlačítko pro spuštění nebo ukončení streamu, textové pole pro pojmenování a konečný náhled odchozího obrazu. V průběhu streamu se zde také zobrazují informace o aktuálním počtu diváků a dosavadní doba vysílání.

Druhou částí panelu je seznam vstupních zdrojů, do kterého lze přidávat nové položky stisknutím tlačítka na vrchu celého listu. Každý zdroj je zde reprezentován ve formě karty, uvnitř které lze nastavit název daného zdroje, zobrazit jeho streamovací klíč a případně také klíč resetovat nebo zdroj zcela odstranit. Kromě toho také karta obsahuje ONLINE/OFFLINE indikátor, který uživatele informuje o tom, zda daný zdroj právě odesílá data na server nebo ne. Karty podporují interakci drag-and-drop, pomocí které je možné zdroje v seznamu libovolně seřadit nebo je lze přetáhnout přímo do vybraného okna uvnitř layoutu, čímž se pro tento zdroj automaticky vytvoří nová karta s náhledem. Oddělovač této části také obsahuje adresu streamovacího serveru. Narozdíl od streamovacího klíče zůstává tato adresa stejná pro veškeré zdroje a z hlediska rozhraní jí tak dávalo smysl hierarchicky nadřadit celé části,



Obrázek 3.2: Sekce Signup

místo toho aby byla obsažena mezi položkami v seznamu zdrojů.

Třetí částí v postranním panelu je seznam výstupů, na které se bude zpracovaný přenos odesílat. Tato sekce se vizuálně velmi podobá sekci zdrojů a každý výstup je také reprezentován kartou v seznamu. Uvnitř této karty lze pak vždy nalézt dropdown menu pro výběr typu cílové destinace (např. Custom RTMP, YouTube, Twitch,...) a tlačítko pro smazání výstupu. Další položky uvnitř karty pak záleží na konkrétním vybraném typu.

Poslední položkou v postranním panelu je část Config, která slouží k nastavení všeobecného chování živého studia. Právě zde je možné nakonfigurovat audio výstupy Master Out a Monitor Out a společně s tím tu lze také nalézt checkboxy, které zapnou nebo vypnou funkce automatického zaostření aktivního zdroje a automatické "solo" zvuku aktivního zdroj.

Veškeré kromě první části postranního panelu může uživatel libovolně zvětšovat nebo zmenšovat potažením za oddělovač dané části. Kliknutím na oddělovač může vybrané části také zcela skrýt. Tím je zajištěno, že prostor zbytečně nezabírají části, které pro uživatele nejsou v momentální chvíli relevantní, a prostor panelu je využit efektivně pro důležité části. Celý postranní panel lze také zcela skrýt kliknutím na tlačítko šipky u horní hrany panelu a uživatel tím může získat více prostoru pro zobrazení layoutu.

3.2.2 Layout

Tato oblast stránky je tvořena hierarchií oken, kterou si uživatel může nakonfigurovat dle svých potřeb. Uvnitř každého z těchto oken se pak nachází série přepínatelných karet, kde každá z nich může obsahovat náhled vybraného zdroje, náhled výsledného přenosu nebo pohled Floorplan. Horní hrana každého okna obsahuje kromě seznamu karet také tlačítka pro ovládání zdrojů. Jmenovitě zde lze nalézt možnosti mute (M), solo (S) a CUE, které provedou korespondující audio akci pro zdroj uvnitř právě zvolené karty, a tlačítko More, které při stisku otevře dropdown menu s dalšími možnostmi. V tomto menu se nachází možnosti pro změnu zdroje dané karty, pro nastavení klávesových zkratek, pro horizontální a vertikální rozpůlení okna a pro hromadné uzavření všech karet uvnitř daného okna.

Nové karty lze vytvářet kliknutím na tlačítko +, umístěné na liště karet uvnitř daného okna, nebo přesunutím karty zdroje z postranního panelu do vybraného okna. Karty v layoutu lze také přesouvat mezi okny nebo mezi sebou v rámci jednoho okna pomocí drag-and-drop.

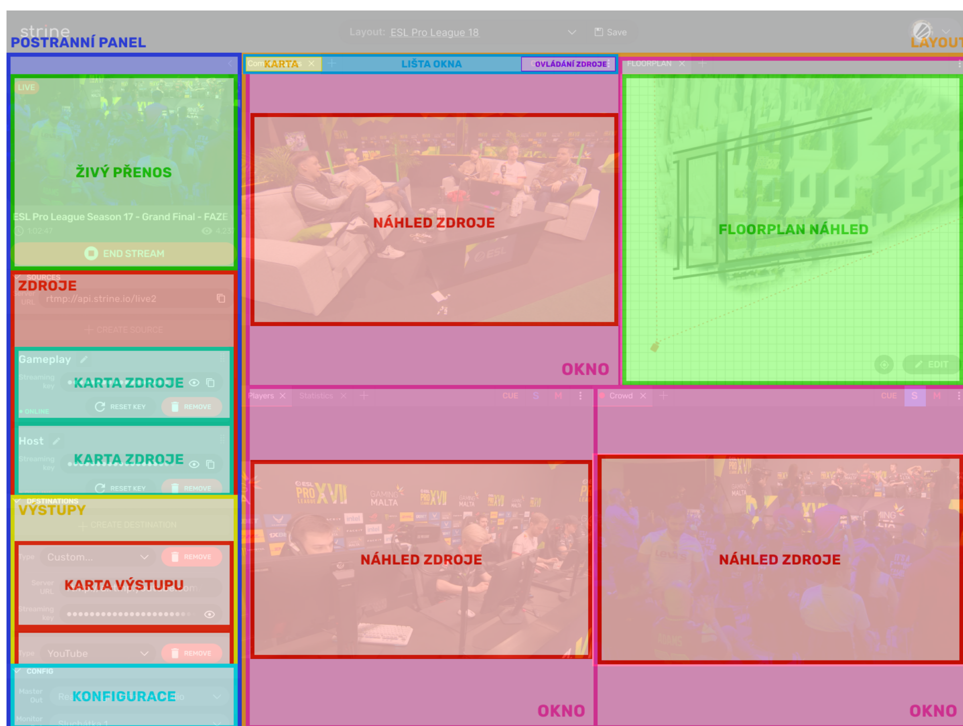
Nová okna lze v layoutu získat rozpůlením existujícího okna na dvě. Toho lze dosáhnout buď manuálním kliknutím na tlačítka Horizontal Split a Vertical Split uvnitř dropdown menu daného okna nebo drag-and-drop přetažením existující karty k jednomu z okrajů existujícího okna. V takovou chvílí se uživateli zobrazí indikátor rozpůlení a při puštění karty se okno automaticky rozdělí na dvě, s tím že k jedné z polovin se přiřadí přetahovaná karta.

Vytvořené layouty si uživatel může ukládat pro budoucí použití. K tomu slouží sekce Layouts, nacházející se uprostřed horního navigačního panelu. V této sekci lze aktuální layout pojmenovat nebo uložit a při stisku šipky se také otevře dropdown menu pro prohledávání existujících layoutů, ze kterého lze v minulosti vytvořené layouty znovu načíst.

3.3 Floorplan Editor

Na obrazovku Floorplan Editor je uživatel přesměrován při stisku tlačítka Edit uvnitř karty s Floorplan náhledem a jejím účelem je upravování obsahu uvnitř tohoto náhledu. Náhled Floorplan je tvořen hierarchií vrstev, kde každá z nich reprezentuje jednu kameru nebo obrázek, a obraz na vyšších vrstvách překrývá vrstvy nižší. Rozhraní je rozděleno na levou lištu pro výběr nástrojů, středový editor Floorplan náhledu a pravý postranní panel pro konfiguraci vrstev.

Nástrojová lišta obsahuje nástroje Cursor, Add Camera, Add Image a

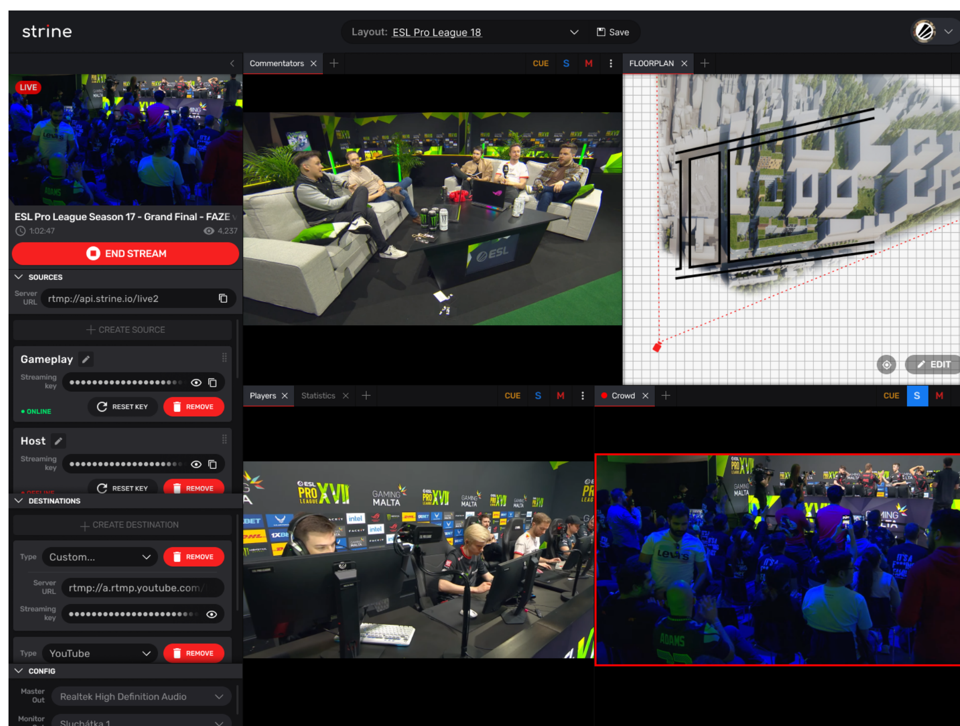


Obrázek 3.3: Schématický nákres obrazovky Studio

Center View. Při volbě nástroje Cursor může uživatel označovat vrstvy uvnitř editoru a kliknutím na vybraný obrázek nebo kameru se pro danou vrstvu zobrazí možnosti pro úpravu. Tato akce je ekvivalentní jako kliknutí na vrstvu v pravém postranním panelu. Pokud je právě aktivní nástroj Add Camera tak kliknutí uvnitř editoru vytvoří novou kameru v místě stisku. Ekvivalentně, použití nástroje Add Image nejprve zobrazí modál pro výběr cílového obrázku ze souborů a poté vybraný obrázek vloží na místo stisku. Nástroj Center View pak slouží jako dodatečný přepínač, jehož aktivace způsobí, že se při vybrání určité vrstvy pohled editoru přesune tak, aby byla tato vrstva vycentrována uprostřed obrazovky.

Spodní část nástrojové lišty navíc obsahuje ještě dvě další tlačítka. Prvním z nich je tlačítko s ikonou diskety, sloužící k uložení aktuálního stavu Floorplan náhledu. Klávesová zkratka Ctrl+S funguje jako ekvivalentní alternativa. Druhé tlačítko s ikonou šipky slouží k návratu na obrazovku Studio. Pokud v době stisku není stav Floorplanu uložený, pak se před navigací zobrazí modál s varováním neuložených změn.

Pravý postranní panel je rozdělen na dvě části. První z nich obsahuje seznam všech existujících vrstev seřazený od nejvyšší vrstvy po nejnižší a pořadí vrstev lze upravit drag-and-drop přetažením vybrané vrstvy v tomto seznamu. Každou z vrstev lze také dočasně skrýt kliknutím na ikonu oka nebo jí uzamknout proti úpravám kliknutím na ikonu zámku. Druhá část



Obrázek 3.4: Obrazovka Studio

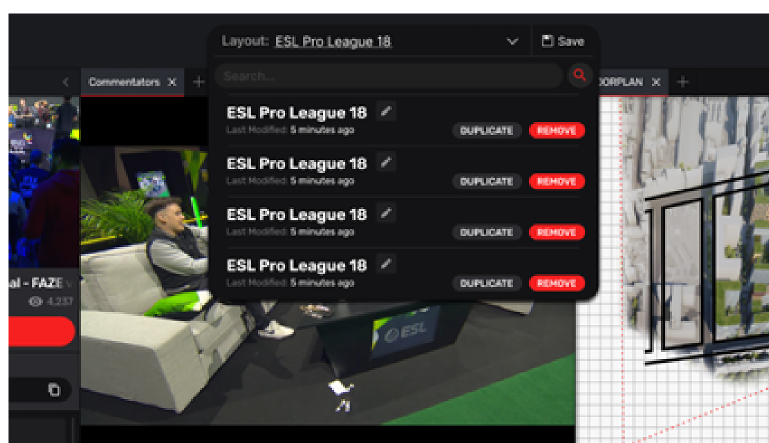
postranního panelu je závislá na kontextu a zobrazuje ovládání pro právě vybranou vrstvu. V případě kamery je tu tak možné nastavit její zdroj, náhledový obrázek, korekci pro zkreslení čočky a její pozici a rotaci v prostoru. V případě obrázku lze pak vybrat zdrojový soubor a nastavit pozici nebo velikost vrstvy. Stejně jako v případě obrazovky Studio je i zde možné změnit relativní velikost obou částí postranního panelu nebo některou z částí zcela skrýt.

3.4 Editor

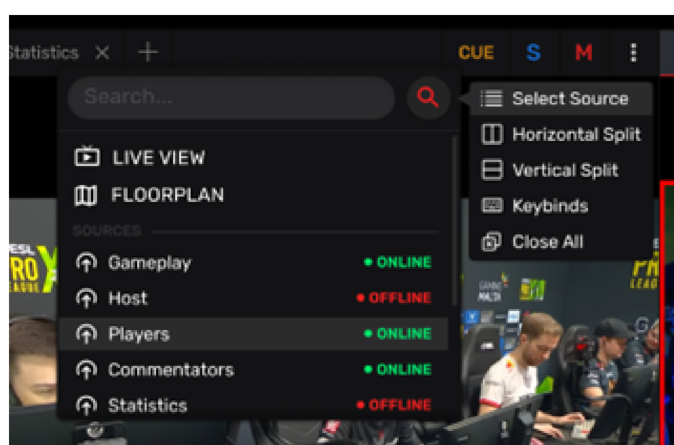
Stránka Editor slouží pro úpravu a export záznamů z proběhlých přenosů a je vertikálně rozdělena do dvou polovin. Spodní polovina obsahuje časovou osu editovaného záznamu, horní polovina je dále horizontálně rozdělena na dvě části: průzkumník a náhled.

3.4.1 Průzkumník

Sekce průzkumníku obsahuje seznam veškerých záznamů proběhlých přenosů a umožňuje v tomto seznamu vyhledávat podle názvu nebo jeho položky



Obrázek 3.5: Sekce Layouts

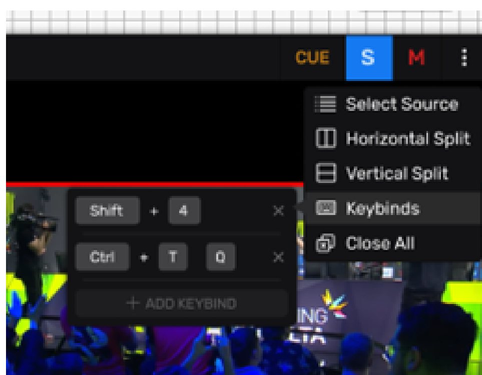


Obrázek 3.6: Select Source menu

řadit alfabeticky, podle délky záznamu nebo podle data vysílání. Každý záznam je v tomto seznamu reprezentován svoji kartou, na které lze nalézt jeho název, délku, datum vysílání a automaticky vygenerovaný náhledový obrázek. Kliknutím na kartu je zvolený záznam načten do editoru. Spodní část průzkumníku pak slouží k úpravám právě načteného záznamu a nabízí možnosti pro jeho přejmenování, smazání, duplikaci, export ve formátu MP4 a uložení provedených změn.

3.4.2 Náhled

Sekce náhledu je poměrně jednoduchá a obsahuje pouze samotné okno s video náhledem a lištu s tlačítky ovládání. Ovládací lišta nabízí možnosti pro přehrání a pozastavení záznamu, přetočení na začátek a na konec záznamu a posun o snímek dopředu a dozadu. Okno náhledu pak v jakoukoliv danou chvíli zobrazuje snímek obrazu na aktuální pozici časového kurzoru, umístěného na



Obrázek 3.7: Keybind menu

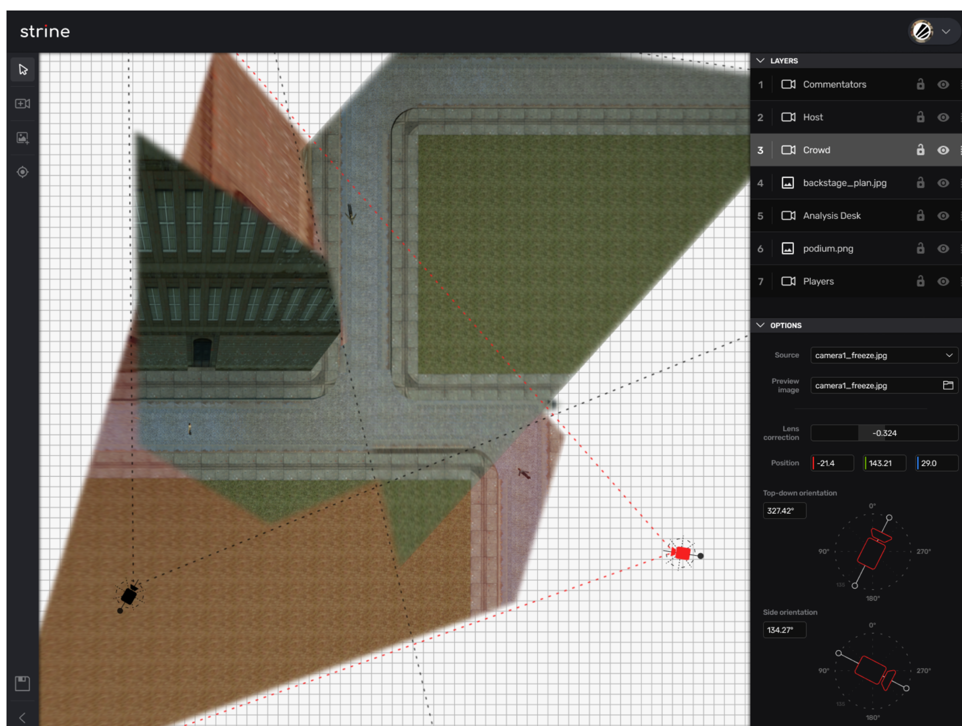
časové ose.

3.4.3 Časová osa

Časová osa editoru je tvořena dvěma oddělenými seznamy stop, jedním pro video stopy a druhým pro audio stopy. Každý existující zdroj v načteném záznamu je zde reprezentován právě jednou video a jednou audio stopou a pořadí stop v seznamu lze upravit drag-and-drop přetažením vybrané stopy na jinou pozici. Přes celou výšku časové osy se pak táhne červený kurzor, který indikuje aktuální pozici přehrávače a jeho přetažením doleva nebo doprava je možné záznam přetočit. Nad oběma seznamy stop se pak nachází časová lišta, ve které lze nalézt časovou značku pro momentální pozici kurzoru a měřítko, které zobrazuje periodické časové značky pro celou šířku časové osy. Zobrazení tohoto měřítka je adaptivní a vykreslené hodnoty se přizpůsobí aktuálnímu přiblížení a posunutí časové osy.

Seznam video stop obsahuje jedinou sdílenou envelope, která je reprezentována tenkou červenou linkou, která se táhne přes celou délku načteného záznamu. Klíčové snímky této envelope jsou reprezentovány malými červenými kruhy a každý z nich reprezentuje změnu aktivní video stopy. Obsah každé video stopy je na časové ose reprezentován šedým obdélníkem, jehož barva je světlejší právě v úsecích, kde je daná stopa zrovna aktivní. V hlavičce každé stopy lze také nalézt tlačítko s ikonou oka, které slouží k vynucení náhledu pro danou stopy. Dokud je tato možnost tedy aktivní, tak bude okno náhledu zobrazovat pouze obsah vybrané stopy nehledě na to, která stopa je právě aktivní podle nastavení envelope.

Seznam audio stop vypadá do značné míry identicky jako seznam video stop, s tím hlavním rozdílem, že místo jediné sdílené envelope přes celý seznam má zde každá audio stopa svou vlastní envelope. Tato envelope pak může být pouze ve dvou možných pozicích: u horní hrany stopy, čímž reprezentuje, že



Obrázek 3.8: Obrazovka Floorplan Editor

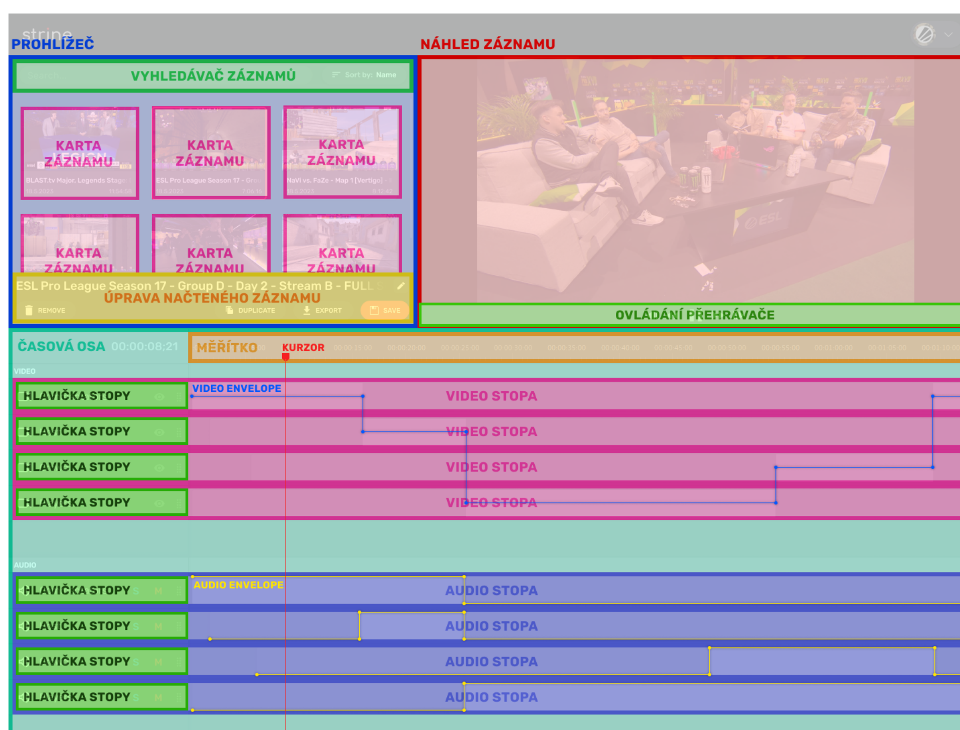
je daná stopa právě slyšitelná, nebo u spodní hrany, čímž reprezentuje, že je stopa ztlumená. Hlavička audio stopy pak obsahuje tlačítka Mute (M) a Solo (S) a aktivace daného tlačítka vynutí ztlumení nebo "solo" dané stopy nehledě na nastavení envelope.

3.5 Player

Stránka Player je s výjimkou Loginu jedinou stránkou, ke které má přístup i nepřihlášený uživatel, a její funkcí je zobrazení živého přenosu pro konkrétní uživatelský profil. Levá část stránky je vyhrazena samotnému videopřehrávači a pravý postranní panel obsahuje živý chat daného přenosu.

Přehrávač nabízí základní ovládání videa jako pozastavení a spuštění přenosu, nastavení hlasitosti a přepnutí do fullscreen. Pod oknem přehrávače se pak nachází základní informace o aktuálním přenosu jako jeho název, délka trvání a momentální počet diváků. Dále jsou zde obsaženy také základní informace o vysílajícím profilu jako jeho jméno, popis a profilový obrázek.

Uvnitř postranního panelu se nachází seznam zpráv v chatu. Každá zpráva zobrazuje nejprve uživatelské jméno odesílatele následované samotným obsahem zprávy. Obsah seznamu je omezený na posledních 50 zpráv a nové



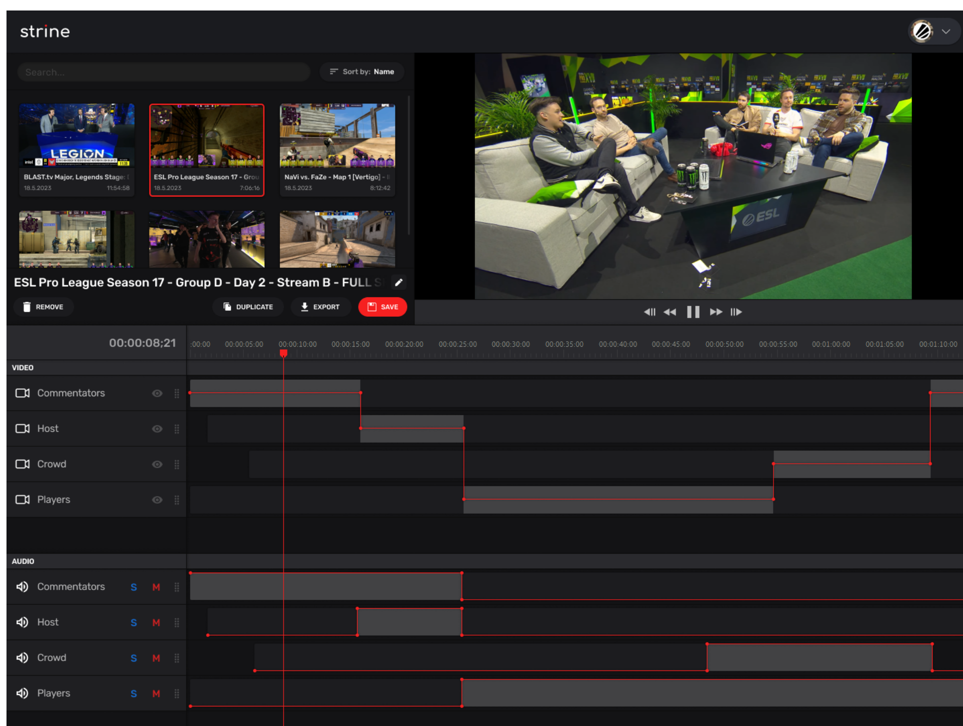
Obrázek 3.9: Schématický nákres obrazovky Editor

zprávy se zobrazují od chvíle, kdy uživatel naviguje na stránku Player. Není tak možné zobrazit historické zprávy z doby před připojením k přenosu. Dolní část postranního panelu poté obsahuje formulář s textovým polem pro psaní zpráv a tlačítko pro odeslání. Pokud uživatel není k aplikaci přihlášený, pak je tento formulář deaktivován.

3.6 Settings

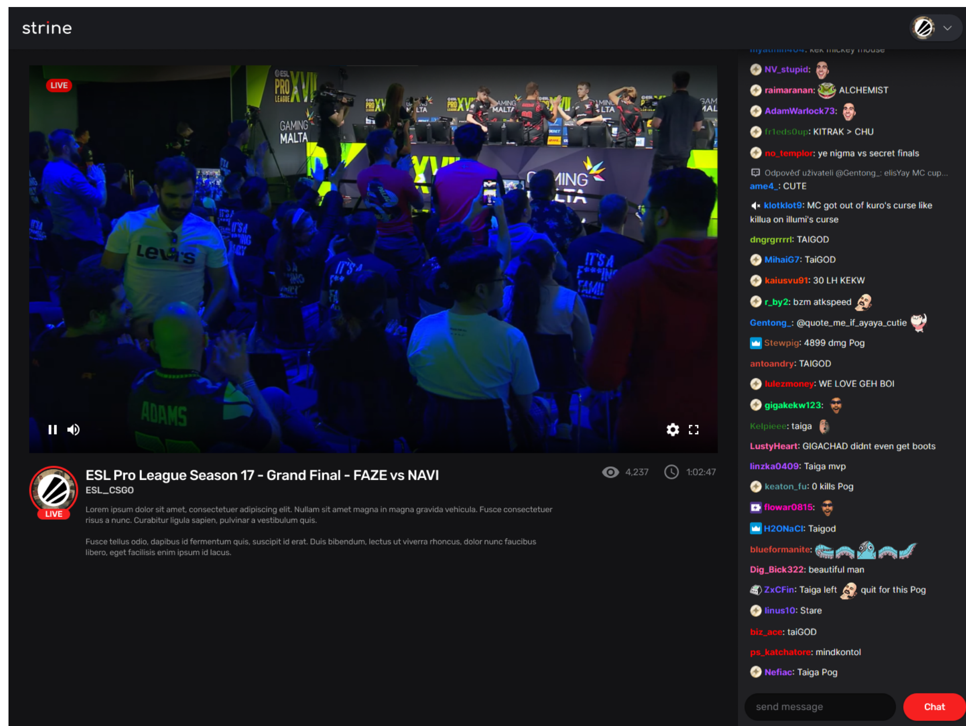
Stránka Settings nabízí základní možnosti pro správu uživatelského účtu a její obsah je rozdělen do dvou částí: formulář pro úpravu veřejných profilových informací a formulář pro změnu hesla.

První formulář nabízí možnosti pro změnu profilového jména, změnu popisku, úpravu profilového obrázku a přidávání nebo odebírání moderátorů. Pole formuláře jsou v tomto případě předvyplněna dříve zadanými informacemi. Druhý formulář obsahuje nevyplněná textová pole sloužící ke změně hesla. Pro úspěšné provedení této akce musí uživatel zadat nejprve své aktuální heslo, své nové heslo a poté znovu zopakovat nové heslo. Oba formuláře jsou následované tlačítky pro uložení změn a vzájemně jsou oddělené oddělovací čarou. Tím je vizuálně indikováno, že oba formuláře jsou vzájemně nezávislé.

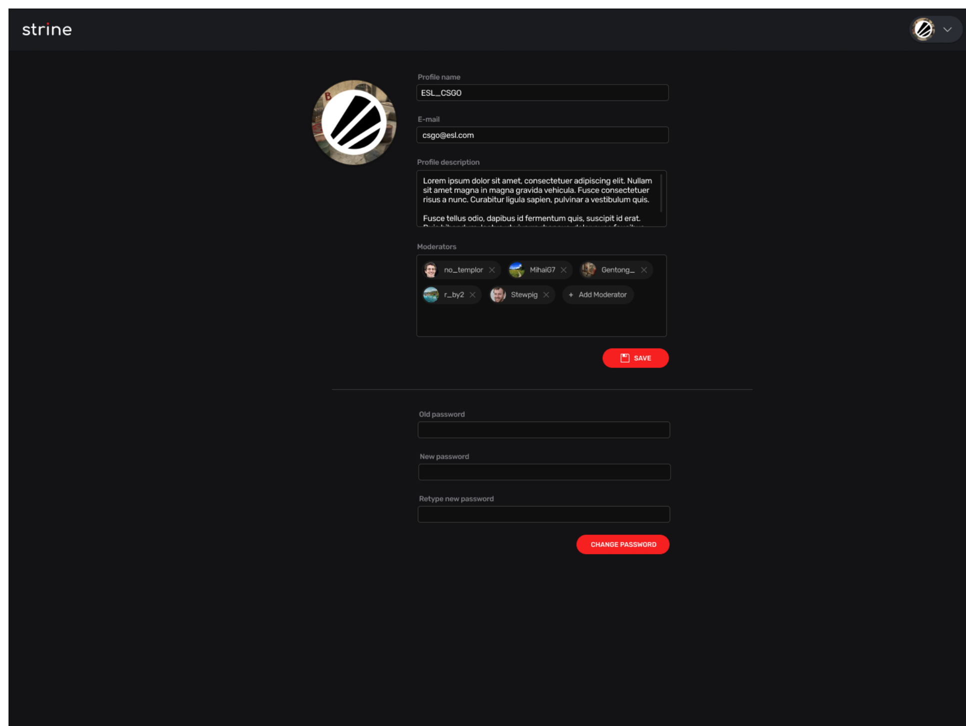


Obrázek 3.10: Obrazovka Editor

Na této stránce je oproti většině ostatních použitý alternativní vzhled textových polí, který je převzatý z obrazovky Floorplan Editor. Tento vzhled byl použit, jelikož lépe funguje pro víceřádková textová pole a byl vzhledově konzistentní se seznamem moderátorů.



Obrázek 3.11: Obrazovka Player

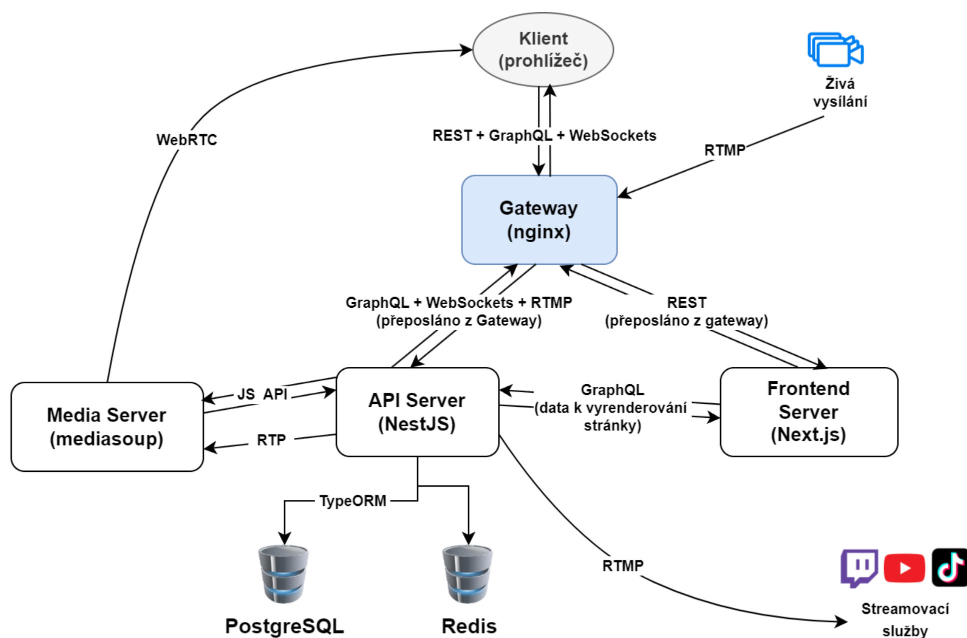


Obrázek 3.12: Obrazovka Settings

Kapitola 4

Technologické řešení

Jelikož je technologické řešení celé platformy tvořeno kombinací mnoha rozmanitých technologií, tak byla architektura aplikace navržena ve formě několika vzájemně komunikujících služeb, místo jako jediná monolitická codebase. Pro účely nasazení jsou pak jednotlivé služby sestaveny ve formě Docker kontejnerů a celková architektura platformy je definována konfigurací Docker Compose. Správu závislostí uvnitř kontejnerů má pak nastarosti nástroj pnpm v případě JS/TS projektů a v případě Pythonu je k instalaci použit nástroj pip uvnitř virtuálního prostředí venv. O finální nasazení aplikace na produkční server se stará nástroj Ansible.



Obrázek 4.1: Vizualizace softwarové architektury

4.1 Gateway

Vstupním bodem celého systému je webový server nginx, který slouží jako reverse proxy a příchozí požadavky přesměrovává na příslušné aplikační servery. Kromě toho se také stará o příjem příchozích RTMP streamů a validaci jejich streamovacích klíčů. Za tím účelem je nginx zkompilován s volitelným modulem `nginx-rtmp-module`, který přidává podporu pro protokol RTMP a umožňuje nastavení notifikačních callbacků na RESTové API, pomocí kterých nginx informuje API server o začátku a konci RTMP přenosů [51].

4.2 API server

Hlavní službou celé aplikace je služba API server. Ta poskytuje rozhraní v jazyce GraphQL pro veškeré uživatelské interakce, koordinuje zpracování multimédií s ostatními službami a stará se o správu dat v připojených databázích. Celé API je tak definováno ve formě datového grafu, nad kterým může konzument provádět komplexní dotazy a získávat požadované datové položky granulárním způsobem [6]. Použití jazyka GraphQL oproti typickému REST API pak nabízí nejen větší flexibilitu při dotazování dat ze strany klientské aplikace, ale také zaručuje typovou bezpečnost na obou stranách a nabízí pokročilé možnosti cachování dat na základě specifikace jejich datových typů.

Stejně jako v případě RESTových rozhraní probíhá komunikace v GraphQL také na protokolu HTTP a funguje tedy na modelu Požadavek-Odpověď. Část interakcí mezi serverem a klientem tak místo GraphQL využívá ještě obousměrnou komunikaci přes protokol WebSockets, pomocí které lze klienta informovat o nastalých událostech (např. nová zpráva v chatu, začátek livestreamu, změna stavu zdroje atd.), aniž by se musel sám dotazovat [7].

4.2.1 Framework

Samotná implementace tohoto serveru využívá aplikační framework NestJS pro runtime prostředí Node.js, který je v tomto případě nakonfigurován tak, aby jako svůj backend využíval web server framework Express. Díky tomu lze tak v rámci této aplikace využívat i existující middleware balíčky napsané právě pro Express. Hlavní výhodou Nestu oproti použití samotného Expressu je pak zejména jeho rozsáhlý ekosystém, který nabízí existující integrace pro velké množství populárních technologií, a také jeho vynucování projektové struktury a objektově orientované architektury, což ve výsledku vede k více konzistentnímu, udržitelnému a přehlednému kódu [8].

Ve výchozím stavu je však tento framework určen zejména k tvorbě RESTových API a implementaci jazyka GraphQL pro NestJS tak dodatečně poskytuje knihovna Apollo Server, která je sama integrována právě jako middleware nad frameworkem Express a slouží tedy jako mezivrstva mezi těmito dvěma frameworky. Platforma Apollo pak kromě této serverové knihovny také poskytuje klientskou knihovnu Apollo Client, jejíž použití na straně frontendu výrazně usnadňuje API integraci [9].

Obousměrná komunikace přes WebSockets je pak postavena na knihovně Socket.io, pro kterou NestJS také poskytuje vestavěnou integraci a pro kterou také existuje přidružená klientská knihovna Socket.io Client, která je využita na straně webové aplikace pro navázání komunikace se serverem [10].

4.2.2 Jazyk

Samotný kód aplikace je napsán plně v jazyce TypeScript, který je před spuštěním v Node.js přeložen do jazyka JavaScript, oproti kterému však nabízí systém pro statické typování a podporu pro moderní syntaxi, kterou JavaScript momentálně nepodporuje nebo zatím nebyla plně stadrardizována. Mimo to je TypeScript také výchozím jazykem frameworku NestJS, jehož architektura je vysoce závislá na použití dekorátorů, pomocí kterých probíhá většina interakcí mezi uživatelským kódem a samotným frameworkem [8]. Syntakticky se tak tento přístup podobá například Java frameworku Spring, který v podobné kapacitě využívá Java anotace [11].

Volba TypeScriptu je pak zejména výhodná v kontextu tvorby GraphQL API, jehož schéma je samo o sobě také zcela staticky typované a s využitím TypeScript reflekce je tedy možné toto schéma automaticky generovat na základě programových typů [12]. Oproti typickému přístupu, kdy je schéma definováno manuálně, tak nedochází k žádné duplikaci typů mezi jazyky GraphQL a TypeScript a předchází se možné divergenci mezi těmito dvěma typovými systémy.

4.2.3 Databáze

API server pro své fungování vyžaduje propojení se dvěma databázemi: relační SQL databázi PostgreSQL pro ukládání hlavních uživatelských dat a key-value databází Redis pro data bez relačního schématu a data s expirací.

O připojení k databázi Redis se stará stejnojmený balíček `redis`, který s ní při spuštění serveru naváže spojení a vytvořený klient je poté zpřístupněn zbytku systému skrz dependency injection systém v NestJS. Jelikož databáze

Redis nefunguje na bázi schématu, tak s API serverem nevyžaduje žádnou větší integraci.

Oproti tomu, v případě PostgreSQL je potřeba, aby API server dodržoval definované schéma databáze. Toto propojení je tak spravováno systémem pro objektově relační mapování, konrkétně knihovnou TypeORM. Jak už název napodívá, tak i tato knihovna je úsce spjata s jazykem TypeScript a stejně jako v případě GraphQL využívá reflekci k tomu, aby databázové schéma udržovala synchronizované s existujícími TypeScript typy. Databázové entity jsou tedy definovány ve formě tříd a TypeORM opět využívá funkcionalitu dekorátorů k definování tabulek a sloupců databáze [13]. Hlavním výhodou knihovny TypeORM oproti ostatním řešením je pak zejména její snadná integrace s jazykem GraphQL. Třídy databázových entit lze totiž anotovat TypeORM dekorátory i GraphQL dekorátory zároveň a tím tak k dané entitě automaticky vytvořit i odpovídající GraphQL typ.

4.2.4 Autentizace

Autentizace v rámci mojí aplikace se drží stadardního protokolu Oauth 2.0 a přístupová práva daného uživatele jsou tedy ověřována za pomoci přístupového (access) tokenu ve formátu JWT. API server má tak na starosti vydávání nových access a refresh tokenů v moment přihlášení k aplikaci, obnovování access tokenu při vypršení platnosti a deaktivaci refresh tokenů při odhlášení z aplikace nebo z jiných bezpečnostních důvodů [14]. Jedno z využití připojené Redis databáze je pak jako blacklist tokenů, pomocí kterého lze deaktivovat vydané refresh tokeny ještě před skončením jejich platnosti.

Při prvnotním přihlášení uživatele server vytvoří access token s platností 15 minut a refresh token s platností 7 dnů. Klientská aplikace poté každých následujících 15 minut odešle požadavek na vydání nového access tokenu a pro ověření přiloží klientův refresh token. Pokud tento refresh token není na blacklistu, pak je klientovy vydán nový access token. Hlavní výhodou tohoto modelu oproti klasickým sessions je to, že si server nemusí pamatovat jakékoliv údaje o přihlášených uživatelích a přístup k databázi je vyžadován pouze při refreshování tokenu. Tento model je také více vhodný pro budoucí rozvoj, pokud by funkcionalita aplikace byla rozdělena mezi více samostatných backend serverů nebo i v případě load balancování mezi několika instancemi API serveru. Tokenové ověření totiž oproti session nevyžaduje centralizovaný bod pro autentizaci a každý server může uživatelův access token ověřit samostatně a zcela nezávisle na zbytku systému.

4.3 Media server

Jelikož se celá platforma zabývá streamováním multimediálních dat, tak je mediální server další naprosto klíčovou komponentou tohoto systému. Tento server má na starosti správu přenosů přes rozhraní WebRTC, určené pro webovou real-time komunikaci, které aplikace využívá pro doručení audiovizuálních dat z jedontilvých zdrojů na obrazovku Studio [15]. V rámci mé aplikace jsem se pak rozhodl pro použití existujícího open-source serveru `mediasoup`, jehož backend je sice napsaný v jazyce C++, ale poskytuje rozhraní pro jazyk JavaScript a API server s ním tak může interagovat napřímo [16]. Oba tyto servery jsou tak úsce spjaty svou funkcionalitou a API server slouží jako správce pro tento media server.

Vstupní multimediální data jsou pak Media serveru doručena ve formě RTP přenosu, který je živě generován nástrojem ffmpeg pro každý právě vysílající zdroj. Správu jednotlivých instancí ffmepg má na starosti také API server. Jelikož rozhraní WebRTC využívá právě protokol RTP pro implementaci svých mediálních kanálů [17], tak je hlavní prací tohoto media serveru, aby přijímané RTP pakety dále přeposlal klientům, kteří s ním mají vytvořené WebRTC spojení. Na straně klienta je pak pro navázaní tohoto spojení použita přidružená knihovna `mediasoup-client`.

4.4 Frontend server

4.4.1 Framework

Klientská aplikace tohoto projektu je vytvořena ve frontendovém aplikačním frameworku Next.js, který je nadstavbou nad populární komponentovou knihovnou React. Tento framework se mimo jiné stará o server-side rendering aplikace (SSR), což je technika využívaná zejména pro zrychlení "Largest Contentful Paint" a vylepšení skóre SEO v populárních vyhledávačích [18]. Z toho důvodu tak frontend vyžaduje svůj vlastní webový server, který techniku SSR realizuje a který je vestavěnou součástí frameworku Next.js.

Samotným cílem SSR je pak dynamicky vygenerovat HTML dokument požadované stránky na základě definovaného React komponentového stromu. Zatímco u typické React "single page application" (SPA) by se tak při prvotním načtení webu nejprve odeslal prázdný HTML skeleton a všechen obsah by byl následně vytvořen staženým JavaScriptem na straně klienta [19], tak v případě SSR se již v samotném HTML dokumentu nachází prvotní obsah celé stránky (ačkoliv zcela statický bez jakékoliv interaktivity). Díky tomu tak

může uživatelův prohlížeč obsah této stránky ihned zobrazit a teprve následně čekat na stažení potřebného JavaScriptu. Po jeho stažení je pak tato existující stránka tvz. "hydratována", během čehož stažený kód převezme kontrolu nad obsahem a spustí veškerou interaktivitu [20]. Po dokočení hydratace se pak stránka chová jako klasická React SPA aplikace a veškeré následné navigace již probíhají na straně klienta.

SSR také umožňuje celkovou redukci odesílaných API požadavků ze strany klienta, jelikož lze potřebná data načíst z API již během renderování na straně serveru (např. uživatelská data pro zobrazení profilu). Načtená data jsou pak nejen použita při renderování samotných HTML elementů, ale jsou do HTML dokumentu také embedována ve formě serializovaného JSON objektu [21]. V průběhu hydratace tak může Next.js tato data deserializovat a předat je klientskému kódu pro další zpracování (např. naplnění lokálního cache). Oproti SPA lze tedy eliminovat dodatečné odesílání API požadavků po prvotním načtení stránky a zároveň tak již od první chvíle uživateli zobrazit hodnotná data místo načítacích spinnerů. V případě mě aplikace lze také během SSR očekávat řádově kratší trvání API požadavků, jelikož se v rámci mojí architektury Frontend server a API server nachází na sdílené lokální síti.

Klientská aplikace je pak implementována s využitím moderního routeru "App Router", který byl do Next.js nově přidán v říjnu roku 2022 [23] a oproti původnímu "Pages Router" přidal zejména podporu pro tvz. React Server Components (RSC). S použitím RSC lze pak vybrané komponenty se statickým obsahem nastavit pro renderování pouze na straně serveru během SSR a zcela se tak vyhnout hydrataci těchto částí komponentového stromu [22]. Díky tomu tak klientovi není potřeba odesílat jakýkoliv JavaScript kód pro vyreslení těchto komponent a lze tak snížit celkovou velikost JS bundle i trvání hydratace. Server Components oproti klientským komponentám (komponenty, které nejsou RSC) také nemusí svůj komponentový strom vracet synchronně, ale mohou vrátit Promise objekt. Díky tomu lze tedy funkce těchto komponent definovat jako "async" a poté je použít pro provedení asynchonních operací (v tomto případě fetchování dat z API serveru).

Na straně klientských komponent je pak k asynchronní komunikaci s GraphQL API použit modul Apollo Client, kterým přímo integruje se serverovou knihovnou Apollo Server a poskytuje React Hooks pro provádění GraphQL dotazů a mutací. Pro načtení dat v komponentách je pak konkrétně použit Hook `useSuspenseQuery`, který využívá Suspense API, představené v React 18, aby komponentu "suspendoval", dokud není API požadavek dokončen [24]. To má v kontextu SSR za následek, že React pozastaví renderování dané stránky do doby, kdy jsou data načtena, a uživatel tak uvidí až kompletní verzi webu se všemi informacemi. Při běhu na straně klienta pak tento Hook způsobí blokování navigace až do chvíle dokončení všech požadavků, které navigovaná stránka vyžaduje. Nová stránka je tedy po krátké chvíli odkryta celá s veškerým obsahem, místo aby byla odkryta instantně s řadou postupně

mizejících načítacích spinnerů. Pro integraci Apollo Client s App Routerem frameworku Next.js je pak v aktuální době vyžadován ještě přídavný balíček `@apollo/experimental-nextjs-app-support`, který se stará hlavně o automatické obnovení Apollo Client cache tak, aby cache po hydrataci obsahoval veškerá data, která byla stažena během SSR [25].

4.4.2 Typování

Stejně jako v případě API serveru je i na straně klientské aplikace použit jazyk TypeScript. V tomto případě je však dynamika mezi GraphQL a TypeScriptem opačná a místo toho, aby typové definice TypeScript sloužili jako zdroj pro GraphQL schéma, tak je naopak potřeba, aby zde TypeScript vynucoval GraphQL typy již definované na straně serveru. Veškeré GraphQL dotazy, mutace a fragmenty, které klientská aplikace využívá, jsou tak nejprve definovány ve formě `.graphql` souborů a následně je použit nástroj GraphQL Code Generator, který tyto definice validuje proti schématu staženému ze serveru a následně pro každou definici vygeneruje korespondující TypeScript typ a "Document" objekt, který je přímo kompatibilní s knihovnou Apollo Client a je použit pro spuštění daného query nebo mutace [26].

Jelikož musí aplikace často pracovat s uživatelem zadanými údaji, jelichž typovou správnost nelze zaručit během kompilace, tak je v kombinaci s TypeScriptem použita také validační knihovna Zod. Ta umožňuje definovat typy a pravidla, která jsou ověřována přímo za běhu aplikace a je specificky určena pro integraci s TypeScriptem. Z každého definovaného Zod schématu lze tak přímo odvodit korespondující TypeScript definici pomocí typu `z.infer <T>` [27]. V praxi je Zod integrován s knihovnou React Hook Form, která se stará o celkovou správu všech formulářů, aby validoval zadaná data ještě před odesláním na server. Validační pravidla pak ve většině případů přesně odpovídají server-side validaci.

4.4.3 Multimédia

Přehrávání multimediálního obsahu existuje v klientské aplikaci ve 2 formách: přehrávání jednotlivých zdrojů na obrazovce Studio a přehrávání výsledného livestreamu na obrazovce Channel.

Přehrávání zdrojů ve Studiu využívá rozhraní WebRTC, určené pro peer-to-peer real-time komunikaci s nízkou odezvou, které nejčastěji nachází uplatnění například v kontextu webových videokonferencí [28], které sami o sobě fungují na podobném principu jako obrazovka Studio. Pro počáteční navázání WebRTC spojení je pak nejprve použita komunikace na protokolu WebSockets přes knihovnu Socket.io Client, pomocí které jsou mezi klientem a serverem

vyjednány vzájamené požadavky a je nalezena optimální cesta po síti pro vytvoření peer-to-peer spojení (tvz. "signaling") [15]. Na základě této komunikace pak klient vytvoří nový Transport objekt z knihovny `mediasoup-client`. Pro každý živý zdroj jsou pak pro tento Transport vytvořeny dva různé Consumer objekty: jeden pro video stopu a druhý pro audio stopu [29]. Obě tyto stopy jsou poté spojeny do jediného MediaStream objektu, který existuje v rámci prohlížečového MediaStream API [30]. Tento MediaStream pak může být již přímo použit jako zdroj pro <video> element nebo AudioContext (viz sekce 5.5).

V případě obrazovky Channel je přenos doručen pomocí protokolu HLS. Ten funguje na bázi postupného generování multimediálních chunků ve formátu MPEG-TS a jejich průběžného stahování na straně klienta za použití popisného playlistu ve formátu m3u8 [31]. Tento protokol sice oproti WebRTC příchází s řádově vyšší dobou odezvy, a není tak vhodný pro real-time obrazovku Studio, ale jeho výpočetní náročnost je oproti WebRTC také výrazně nižší. Místo udržování aktivního WebRTC spojení s každým připojeným klientem totiž HLS multimedální chunky existují jen ve formě obyčejných statických souborů, které může triviálně doručovat přímo Gateway server nginx a do budoucna je možné i jejich cachování na serverch CDN. Tento přístup je tak výrazně lépe škálovatelný než real-time přenosy WebRTC a jedná se tedy o ideální řešení pro doručení streamu vysokému počtu souběžných diváků.

Hlavním problémem HLS je pak zejména roztříštěná podpora mezi webovými prohlížeči. Nativní podporu na desktopových platformách momentálně nabízí pouze Safari, avšak v ostatních prohlížečích je možné podporu přidat jako JavaScriptovou "Media Source Extension" (MSE), například ve formě knihovny `hls.js`. Některé z verzí iOS Safari však nopak nenabízí plnou podporu pro MSE a pro stabilní fungování HLS přenosů přes všechny platformy je tak potřeba přehrávač implementovat v obou variantách. Z toho důvodu jsem se tak rozhodl pro využití knihovny `video.js`, která nejen poskytuje vysoce stylovatelné uživatelské rozhrání přehrávače, ale také se stará o polyfillování HLS pro populární prohlížeče [32].

4.4.4 State Management

O správu globálního stavu této aplikace se společně starají 2 různá řešení: state management knihovna Zustand a již zmiňovaný Apollo Client, jehož cache sám o sobě slouží jako globální úložiště a nabízí i pokročilé funkce pro přímé čtení z a zapisování do cache.

Apollo tak v mojí aplikaci spravuje data, která jsou buď přímo načtená ze serveru nebo jsou ze serverových dat odvozená. V praxi to jsou tak zejména informace o uživatelských profilech a informace o definovaných zdrojích. Tyto

data jsou pak aktualizována pomocí optimistických updatů, skutečných odpovědí serveru nebo manuálně, například v reakci na přijetí WebSocket zprávy. Apollo Client mimo to umožňuje i uložení zcela samostatných klientských dat pomocí direktivu `@client` [33], avšak nutnost tato data stále definovat v rámci GraphQL schématu a nutnost k nim přistupovat pomocí GraphQL dotazů dělá z této volby poměrně neergonomickou možnost. Tento přístup je tak vhodný spíše pro občasné přidávání dotatečných dat do již existujího serverového schématu, spíše než pro ukládání zcela nezávislých dat.

Z toho důvodu je tak state management řešení mé aplikace doplněno knihovnou Zustand, která je extrémně "lightweight" 1.1kB (gzipped, verze 4.4.7) alternativou k populárním React state management knihovnám jako Redux nebo MobX [34]. Kromě velikosti je pak její hlavní výhodou také velmi snadné užití, které třeba oproti Reduxu nevyžaduje použití <Provider> komponent a umožňuje vytváření asynchronních akcí bez dodatečné konfigurace. Vytvoření nového store je tedy možné provést pouhým zavoláním funkce `create()`, která jako návratovou hodnotu vrátí Hook, který lze ihned naimportovat a použít v libovolných React komponentách. Klíčovou vlastností pro mou aplikaci pak byla také možnost interagovat s obsahem store i mimo kontext React komponenty pomocí funkcí `getState()` a `setState()`, díky čemuž jsem mohl definovat i vzájemné interakce mezi několika různými stores.

Vybrané stores také využívají vestavěný Zustand middleware pro integraci s knihovnou Immer, díky které lze úpravy stavu provádět imperativním způsobem. Stav uvnitř Zustand (stejně jako v React) totiž ve výchozím stavu musí zůstat vždy "neměnný" (immutable) a místo mutace existujícího stavu je tak potřeba vytvořit novou kopii tohoto stavu s požadovanými změnami. V případě použití Immeru lze však mutace provádět přímo na Immerem vytvořeném Proxy objektu aktuálního stavu a na základě provedených mutací se pak Immer sám postará o konstrukci nové kopie stavového objektu [35]. Tato integrace pak zejména usnadnila znovupoužití existujících funkcí z prvotního prototypu této aplikace, ve kterém byl pro state management místo Zustand použit Redux Toolkit, který sám interně Immer používá a rozhraní pro úpravu stavu je tak identické.

4.4.5 Uživatelské rozhraní

Jednou z klíčových součástí tohoto projektu byla i tvorba přehledného a vysoce přizpůsobitelného uživatelského rozhraní, a to zejména na obrazovce Studio, které režisérovi živého přenosu musí nabídnout maximální komfort a kontrolu nad vysíláním. Pro dosažení optimálního a jednotného UI jsem se tak rozhodl pro vytvoření vlastních znovupoužitelných komponent místo využití některé z již existujících komponentových knihoven. Veškeré styly těchto komponent jsou tak definovány ručně a k jejich psaní byl využit preprocesor SASS, konkrétně jeho syntaxe SCSS.

Narozdíl od většiny ostatních komponentových frameworků jako například Vue, Svelte nebo Angular však React nenabízí vestavěné řešení pro zapouzdření stylů uvnitř definovaných komponent. Za tímto účelem byl tak použit systém CSS Modules, který vytvořené styly umožňuje rozdělovat do izolovaných modulů [36]. V praxi tak typicky jeden CSS Modul odpovídá zhruba jedné React komponentě (až na výjimky, kdy odvozené komponenty čerpají styly ze stejného modulu). Z hlediska implementace se pak CSS Modules stará o prefixování jmen tříd unikátním ID daného modulu, a tudíž tak zamezuje konfliktům mezi názvy tříd v různých modulech.

Často používaným prvkem v uživatelském rozhraní této aplikace jsou také různorodé ikonografické znaky, které pomáhají s vysvětlením funkcionality interaktivních prvků nebo dodávají vizuální důraz na důležité informační prvky. Zdrojem těchto ikon je knihovna React Icons, která sama slouží jako rozáshlá kolekce existujících ikonografických setů a veškeré ikony poskytuje ve formě přizpůsobiltených React komponent. Použité ikony jsem pak čerpal nejčastěji ze setů VS Code Icons, Font Awesome a Material Design Icons.

Poslední použitou UI knihovnou je pak React Hot Toast, který se stará o zobrazování notifikačních ”toast” zpráv u horního okraje obrazovky. Pomocí nich je uživatel informován o právě nastalých událostech jako například validačních chybách formuláře nebo úspěšném provedení serverového požadavku. Vzhled těchto zpráv byl také modifikován, aby vizuálně korespondoval se zbytkem aplikace.

Kapitola 5

Implementace

Celá aplikace je organizována do 3 oddělených projektů: API, Streamer a Client. Tyto projekty jsou obsaženy v rámci jediného monorepo projektu a jejich kód lze naléz ve stejnojmenných složkách v kmenové složce repozitáře. Projekty API a Client také obsahují soubor Dockerfile pro sestavení svého vlastního Docker kontejneru a celou aplikaci lze spustit pomocí nástroje Docker Compose, pro který je v kmenové složce monorepa obsažený kofingurační soubor.

Kromě toho se v kmenové složce repozitáře nachází také složky `gateway` a `main-db`. Ty mají na starosti vytvoření modifikovaných Docker obrazů serveru nginx a databáze PostgreSQL se specifickou konfigurací pro integraci s ostatními Docker Compose službami.

Chování Docker Compose konfigurace lze upravit pomocí následujících proměnných prostředí:

- `APP_HOST` - Hostname, na kterém je celá aplikace veřejně hostována. Podle této proměnné jsou nastavené odkazy uvnitř verifikačních e-mailů a hodnota pole "Server URL", zobrazená v postranním panelu obrazovky Studio.
- `GATEWAY_PORT` – Port na straně host počítače, na kterém poběží gateway server.
- `MAIL_HOST` – Adresa e-mailového STMP serveru pro odesílání verifikačních e-mailů.
- `MAIL_PORT` – Port e-mailového serveru pro odesílání verifikačních e-mailů.
- `MAIL_SENDER` – E-mailová adresa pro odesílání verifikačních e-mailů.
- `MAIL_PASSWD` – Heslo pro přihlášení k účtu `MAIL_SENDER`.

- `MAIL_LOGO_URL` - URL adresa, na které je staticky hostovaný obrázek s logem aplikace. Tato adresa je použita v záhlaví odesílaných e-mailů.
- `ACCESS_TOKEN_SECRET` – Tajemství použité pro šifrování JWT access tokenu.
- `REFRESH_TOKEN_SECRET` – Tajemství použité pro šifrování JWT refresh tokenu.
- `RTMP_INGEST_PORT` – Port, na kterém gateway přijímá vstupní RTMP přenosy.
- `WEBRTC_RANGE_START`, `WEBRTC_RANGE_END` – Rozsah portů pro navázání WebRTC spojení. Vyžaduje rozash o velikosti právě 100 portů.

Graf celé Docker Compose konfigurace lze najít na obrázku 5.1.

5.1 Gateway

V případě nginx serveru je kontejner založený na Docker Hub obrazu `tiangolo/nginx-rtmp`, ve kterém je narozdíl od výchozího `nginx` obrazu server již zkompilovaný i s modulem `nginx-rtmp-module`. Do tohoto obrazu jsou pak překopírovány konfigurační soubory `/etc/nginx/nginx.conf` a `/etc/nginx/sites-enabled/default`. Pomocí těchto souborů je povoleno přijímání RTMP přenosů a callbacky pro vznik a ukončení přenosu jsou nasměrovány na API server. Pro přeposílání na API server jsou také nastaveny veškeré dotazy přicházející na URL cesty `/gql`, `/api` a `/socket.io`. `/gql` pak slouží jako endpoint pro celé GraphQL rozhraní, `/socket.io` je využíváno knihovnou `socket.io` pro navázání WebSocket spojení a `/api` momentálně nemá žádnou funkci, ale je zarezervováno pro budoucí užití. Cesta `/static` je poté nastavena pro sdílení statických souborů ze složky `/www/static` a tato složka samotná je namountovaná jako sdílený volume mezi Docker kontejnery [37] Gateway a API. Například při nahrání profilového obrázku tak může API server přijmutý obrázek oříznout, zmenšit a uložit do této složky a tento vytvořený soubor bude následně dostupný ze serveru nginx. Jakékoliv další URL cesty jsou přesměrovány na frontendový server.

5.2 Main DB

Složka `main-db` se stará o inicializaci produkční databáze v případě, že ještě nebyla vytvořena. Název výchozí databáze je tak upraven na ”strine” podle názvu aplikace a dovnitř obrazu je překopírován databázový dump schématu, ze kterého se tato databáze inicializuje. Datová složka sestaveného kontejneru je také namountována jako volume a při smazání kontejneru tedy uložená data zůstanou zachována.

5.3 API

5.3.1 Moduly

Základní organizační jednotkou NestJS aplikace jsou tvz. moduly, které jsou implementovány jako třídy označené dekorátorem `@Module()`. Tyto moduly slouží k zapouzdření určitého výseku celkové funkcionality serveru a jejich definice obsahuje seznam veškerých závislostí (tříd), který tento modul vyžaduje ke svému fungování [42]. Ve výchozím nastavení jsou pak veškeré závislosti pro daný modul instanciovány jako singleton a ostatní závislosti uvnitř tohoto modulu si vytvořenou instanci mohou vyžádat jako jeden z parametrů svého konstruktoru. Dependency injection systém uvnitř Nest.js se pak automaticky postará o její předání při konstrukci singletonu této třídy [41]. Tento systém tak funguje obdobně jako konstruktory s anotací `@Autowired` uvnitř Java frameworku Spring [11].

V konkrétním případě mé aplikace jsem se pak veškerou funkcionalitu rozhodl organizovat pouze do jediného modulu `AppModule`, který obsahuje veškeré závislosti mé aplikace dále popisované v této sekci, jelikož většina funkcionality vyžaduje úzkou spolupráci mezi více závislostmi a rozdělení do více samostatných modulů by vedlo pouze ke složitější organizaci kódu a nutnosti explicitně předávat závislosti mezi moduly.

Ačkoliv je však samotný kód aplikace organizovaný pouze do jediného modulu, tak tento modul sám importuje několik dalších modulů přímo vestavěných ve frameworku NestJS. Těmi jsou konrkétně `ConfigModule`, který se stará o načtení proměnných prostředí ze souboru `.env` [44], `TypeOrmModule`, který spravuje spojení mezi databází PostgreSQL a knihovnou pro objektově relační mapování TypeORM [43] a `GraphQLModule`, který má na starosti integraci mezi frameworkem NestJS a GraphQL serverem Apollo Server [45].

5.3.2 Factory providers

Provider v rámci NestJS představuje základní generickou jednotku pro poskytování dat v rámci vestavěného inversion of control systému skrz dependency injection. Koncept providerů pak v NestJS existuje v několika různých formách; v tomto případě se bavíme konrkétně o tvz. "factory providers", které jsou definovány objektem s metodou `useFactory()` a návratová hodnota této funkce je přímo poskytnuta ostatním závislostem daného modulu [46]. Má aplikace pak tyto factory providers využívá zejména ke správě dlouhožijících spojení navázaných při prvotním spuštění serveru a existujících po celou dobu jeho běhu.

MailTransportProvider

Provider MailTransportProvider (viz obrázek 5.2) využívá knihovnu Nodemailer, aby vytvořil spojení s e-mailovou adresou pro odesílání notifikačních zpráv aplikace (ověření účtu, reset hesla). Toto spojení je navázáno na základě proměnných prostředí `MAIL_HOST`, `MAIL_PORT`, `MAIL_SENDER` a `MAIL_PASSWD` (viz kapitola 5) a vytvořený `nodemailer.Transporter` objekt je konzumován službou MailService, která nad ním staví vyšší úroveň funkcionality.

RedisProvider

RedisProvider (viz obrázek 5.3) při spuštění serveru naváže spojení s přidruženou databází Redis a vytvořenou instanci klientu poskytuje zbytku systému. Spojení je nakonfigurováno pomocí proměnných prostředí `REDIS_HOST` a `REDIS_PORT`, ale ty jsou v rámci Docker Compose automaticky nastaveny tak, aby se klient připojil k databázi v kontejneru `redis`.

MediasoupProvider

Tento provider (viz obrázek 5.4) se stará o spuštění mediálního serveru `mediasoup` a jeho konfiguraci z hlediska používaných portů pro navázání WebRTC spojení s klientskou aplikací a nastavení používaných mediálních formátů. Ty jsou zde nastaveny jako formát VP8 pro přenos obrazových dat a formát Opus pro kódování audio stopy. Tyto formáty jsou pak při konverzi vstupních přenosů zrcadleny v nastavení nástroje `ffmpeg` (viz podsekce 5.3.10). MediasoupProvider zbytku systému poskytuje objekt `mediasoup.Worker`, který reprezentuje běžící proces mediálního serveru, a `mediasoup.Router`, který se stará o správu a propojení existujících WebRTC transportů.

ImageUtilProvider

ImageUtilProvider (viz obrázek 5.5) poskytuje základní funkcionalitu pro práci s profilovými obrázky uživatelů.

Pro zpracování a uložení obrázků poskytuje tento provider metodu `process()`, která přijímá objekt typu `FileUpload` z knihovny `graphql-upload`, která se stará o nahrávání souborů skrz GraphQL rozhraní. Provider tak nejprve vygeneruje náhodný unikátní název souboru pro daný obrázek, z předaného `FileUpload` objektu vytvoří Node.js `ReadStream` a ten je poté nejprve zpracován image processing knihovnou `sharp` a následně uložen do souboru ve složce

statických souborů (dostupná na URL `/static/image`). Výsledný obrázek je z originálních vstupních dat oříznut na velikost 300x300 pixelů a překonvertován do formátu JPEG.

Provider nabízí také metodu `delete()` pro odstranění obrázku na základě názvu souboru. Tato metoda je využita při změně profilového obrázku, kdy je původní obrázek odstraněn ze serveru a je nahrazen obrázkem novým.

5.3.3 Entity

Třídy entit představují databázové objekty knihovny TypeORM a společně s tím i korespondující GraphQL typy těchto objektů (viz podsekce 4.2.3) [47]. Každá entita sama o sobě dědí ze třídy `BaseEntity`, importované z knihovny TypeORM, která poskytuje funkce pro interakci s databází, jako např. vyhledání entity podle ID nebo uložení vytvořené instance entity do databáze. Databázové schéma připojené PostgreSQL databáze je pak přímo odvozeno z definic těchto tříd.

UserEntity

Tato entita (viz obrázek 5.6) reprezentuje ověřený uživatelský účet a je ve vztahu OneToMany vůči entitě SourceEntity. Z důvodu bezpečnosti pak databázový sloupec `password` existuje pouze na straně databáze a v rámci GraphQL schématu není vůbec odhalen. Oproti tomu, položka `live`, obsahující informace o právě vysílaném přenosu daného uživatele, je definována pouze uvnitř GraphQL schématu a při dotazu na API se o její doplnění do serverové odpovědi stará "field resolver" funkce uvnitř třídy UserResolver. Položka `image` pak obsahuje název souboru, do kterého byl uložen profilový obrázek uživatele, a lze k němu přistoupit na URL `/static/image/[image]`.

SourceEntity

Entita SourceEntity (viz obrázek 5.7) představuje příchozí zdroj do režisérského studia konkrétního uživatele. Tato entita je tedy ve vztahu ManyToOne vůči entitě UserEntity a tento vztah je v rámci databáze modelován jako sloupec `ownerId`, který obsahuje ID uživatelského účtu vlastnícího daný zdroj. Položka `isLive`, která reprezentuje, zda daný zdroj právě vysílá, je pak definována pouze na straně GraphQL schématu a o její doplnění se stará "field resolver" funkce uvnitř třídy SourceResolver.

Název	Unikátní	Nullable	PostgreSQL typ	GraphQL typ
id	ano (primární klíč)	ne	UUID	ID!
nick	ano (sekundární klíč)	ne	VARCHAR	String!
name	ne	ne	VARCHAR	String!
password	ne	ne	VARCHAR	- *neobsahuje* -
email	ne	ne	VARCHAR	String!
bio	ne	ano	TEXT	String
image	ano	ano	VARCHAR	String
sources	ne	ne	- *neobsahuje* -	[SourceEntity!]!
live	ne	ano	- *neobsahuje* -	LiveInfo

Tabulka 5.1: Schéma entity UserEntity

Název	Unikátní	Nullable	PostgreSQL typ	GraphQL typ
id	ano (primární klíč)	ne	UUID	ID!
createdAt	ne	ne	TIMESTAMP	Float!
title	ne	ne	VARCHAR	String!
key	ano (sekundární klíč)	ne	VARCHAR	String!
owner	ne	ne	UUID	UserEntity!
isLive	ne	ne	- *neobsahuje* -	Boolean!

Tabulka 5.2: Schéma entity SourceEntity

5.3.4 Middleware

Účelem middleware v NestJS je předzpracování přicházejícího HTTP požadavku ještě před invokací handler metody dané URL cesty [48]. Jediná middleware třída mé aplikace, ParseTokenMiddleware, pak zpracovává veškeré požadavky přicházející na GraphQL API (cesta `/gql`) a validuje platnost JWT tokenu obsaženého v hlavičce HTTP požadavku v atributu "Authorization". Pokud je tento atribut hlavičky nastaven a zároveň obsahuje platnou verzi access token, tak je obsah (payload) tohoto JWT tokenu uložen do zpracovávaného `Request` objektu, ze kterého může být znovu přečten v následujících metodách zpracovávajících tento požadavek.

Vzhledem k tomu, že tento middleware stojí před všemi GraphQL požadavky, nehledě na to, zda příchozí požadavek vyžaduje autorizaci nebo ne, tak

zde není řešeno samotné zamítnutí neautorizovaného požadavku a middleware veškeré příchozí požadavky propustí dále.

5.3.5 Guards

Gaurds naplňují do určité míry stejnou funkcionalitu jako middleware a také jsou využity ke zpracování požadavku ještě před invokací konečné handler metody, ačkoliv jsou spuštěny až po doběhnutí middleware [48]. Narozdíl od něj však nezachycují veškeré požadavky přicházející na danou URL cestu, ale mohou být nastaveny pouze pro vybrané GraphQL dotazy a mutace. Zároveň také místo generického `Request` objektu mohou přistupovat k `GqlExecutionContext` objektu, pomocí kterého lze předat dodatečné informace do následujících GraphQL handlerů.

AuthGuard

Tento guard (viz obrázek 5.8) je nastaven u všech GraphQL dotazů a mutací, u kterých je vyžadováno přihlášení uživatele. Při spuštění guard zkontroluje, zda je na přijmutém `Request` objektu nastaven JWT payload objekt, o jehož nastavení se dříve v pipeline postaral middleware ParseTokenMiddleware. Pokud nastavený není, znamentá to, že uživatel při odeslání požadavku neposkytnul platný access token, a tento guard tak vyhodí GraphQL chybu se zprávou "Missing authorization", čímž zkratuje zbytek pipeline a ke spuštění následujících handlerů tedy již nedojde.

GetUserGuard

Guard GetUserGuard (viz obrázek 5.9) se stará o získání dat právě přihlášeného uživatele z PostgreSQL databáze. Podle uživatelského ID obsaženého uvnitř JWT payload tedy provede databázový dotaz a získaná data zapíše do `GqlExecutionContext`, ze kterého mohou být přečtena dalšími handler metodami. Tento guard samotný se nestará o ověření autorizace a pokud tak příchozí `Request` objekt JWT payload neobsahuje, tak je požadavek propuštěn do zbytku pipeline bez úpravy `GqlExecutionContext`. Handler metody, které uživatelská data nutně vyžadují, tak musí být nakonfigurována s guardy AuthGuard a GetUserGuard zároveň.

5.3.6 DTOs

Veškeré vstupní argumenty jednotlivých GraphQL mutací jsou definovány v samostatných "data transfer object" (DTO) třídách, anotovaných NestJS dekorátorem `ArgsType()`. Tyto třídy pak nejen definují samotné argumenty a jejich datové typy v GraphQL schématu, ale také se starají o jejich základní validaci. Za tímto účelem je aplikace nakonfigurována s knihovnou `class-validator`, která poskytuje řadu dekorátorů, pomocí kterých lze jednotlivé argumenty anotovat např. s požadavky na rozsah čísel, délku řetězce nebo dodržení RegEx pravidel.

Mimo to lze z poskytnutých primitv této knihovny také vytvářet vlastní validační dekorátory, díky čemuž jsem mohl implementovat dekorátor `TrimLength()`, který se stará o ověření délky textu po oříznutí bílých znaků (whitespace) po obou stranách vstupního řetězce (pomocí metody `String.prototype.trim()`).

Validační pravidla specifikovaná v děchto DTO třídách jsou pak většinou přesně zrcadlena na straně klienta pomocí ekvivalentních funkcí ve validační knihovně Zod (viz podsekce 4.4.2).

5.3.7 Resolvery

Jedním ze základních stavebních bloků pro implementaci rozhraní v jazyce GraphQL jsou takzvané "resolver", funkce, které na straně serveru generují odpoveď pro jedno určité pole datového objektu, na které se klient dotazoval. Tato pole mohou být celé dotazy a mutace, které jsou sami o sobě považovány za pole na implicitiním datovém typu Root, který v GraphQL obsahuje veškeré vstupní body pro průchod datového graphu [49], nebo jenom specifická pole na programátorem definovaných datových typech (např. UserEntity).

AuthResolver

Třída AuthResolver (viz obrázek 5.11) shromažďuje resolver funkce týkající se uživatelské autentizace a konkrétně odpovídá na mutace `login`, `refresh` a `logout`. Pro samotné generování tokenů třída využívá službu AuthService.

Mutace:

- `login` – Mutace nejprve uživatele ověří na základě uživatelského jména a hesla a následně vygeneruje nový access a refresh token. Refresh token je poté uložen do HTTP `HttpOnly` cookie pod jménem `refresh` a access

token je vrácen v odpovědi mutace. Expirace cookie je nastavena na stejnou délku jako expirace JWT tokenu.

- `refresh` – Mutace ověří platnost refresh tokenu v cookie právě zpracovávaného HTTP dotazu. Poté vygeneruje nový access token a vrátí ho v odpovědi mutace.
- `logout` – Mutace odstraní uživatelovo `refresh` cookie a jeho refresh token přidá na blacklist v databázi Redis. Expirace tohoto Redis klíče je nastavena tak, aby položka expirovala ve stejnou chvíli jako skončí platnost JWT tokenu.

UserResolver

UserResolver (viz obrázek 5.12) seskupuje resolvery dotazů a mutací týkajících se správy uživatelských účtů a společně s tím také resolvery polí na GraphQL typu UserEntity.

Dotazy:

- `getUser` – Vrátí uživatelská data typu UserEntity na základě uživatelského jména.
- `me` – Vrátí uživatelská data typu UserEntity právě přihlášeného uživatele.
- `doesUserExist` – Vrátí boolean hodnotu reprezentující, zda existuje uživatelský účet s konrkétním uživatelským jménem.
- `isResetValid` – Vrátí boolean hodnotu reprezentující, zda je předaná hodnota platným ID právě probíhající session pro obnovení hesla.

Mutace:

- `signup` – Tato mutace se stará o vytvoření nové session pro ověření uživatele. Nejprve je tak pro tuto session vygenerováno náhodné ID pomocí knihovny `nanoid` a pod tímto ID je poté vytvořen záznam v databázi Redis obsahující zadané uživatelské jméno, e-mail a heslo. Tento záznam má platnost 24 hodin. Poté je uživateli na zadanou e-mailovou adresu odeslán ověřovací e-mail pro tuto nově vytvořenou session (s použitím služby MailService).
- `verify` – Tato mutace má na starosti dokončení ověřovací session a následné vytvoření uživatelského účtu. Mutace na základě zadaného ID

načte záznam dané session z Redis a pomocí načtených informací vytvoří nový záznam uživatelského účtu v databázi PostgreSQL. Session záznam je poté z databáze Redis odstraněn.

- `requestReset` – Na základě uživatelského jména vytvoří session pro obnovení hesla jako záznam v Redis. Tato session má platnost 20 minut.
- `reset` – Tato mutace přijímá ID obnovovací session a nové heslo. Pokud je zadané ID platné, pak je uživateli, který danou session vytvořil, změněno heslo na nové.
- `updateUser` – Volitelně přijímá hodnoty `name`, `bio`, `password` a `image`. Veškeré nastavené hodnoty jsou pak aktualizovány na objektu UserEntity přihlášeného uživatele a změny jsou persistovány do PostgreSQL databáze. Upravený objekt UserEntity je vrácen jako odpoveď této mutace.

Pole:

- `email` – Pokud uživatel dotazuje jiný uživatelský objekt UserEntity než svůj vlastní, pak je citlivé pole `email` nahrazeno prázným textovým řetězcem.
- `sources` – Pokud uživatel dotazuje jiný uživatelský objekt UserEntity než svůj vlastní, pak je seznam vytvořených zdrojů nahrazen prázným polem.
- `live` – Vrací objekt LiveInfo obsahující informace o právě probíhajícím živému přenosu daného uživatele.

SourceResolver

SourceResolver (viz obrázek 5.13) seskupuje resolvery dotazů a mutací týkajících se správy studiových zdrojů a společně s tím také resolvery polí na GraphQL typu SourceEntity.

Mutace:

- `createSource` – Slouží k vytvoření nového zdroje pro přihlášeného uživatele. Mutace nejprve vygeneruje náhodný streaming key pro nový zdroj pomocí kryptograficky bezpečné Node.js funkce `crypto.randomBytes()`. Tento klíč má délku 32 bytů a je uložen ve formě hexadecimálního řetězce. S tímto klíčem je poté vytvořena nová entita SourceEntity s výchozím názvem "Untitled". Tato entita je poté persistována do databáze a vrácena jako odpoveď této mutace.

- `deleteSource` – Odstraní z databáze vybraný záznam SourceEntity. Pokud tento zdroj v moment smazání právě vysílá, pak je přenos přerušen zavoláním metody `LiveService#setOffline()`. Pokud byl tento zdroj zároveň použit i jako jeden ze zdrojů právě probíhajícího livestreamu daného uživatele, pak je i tento koncový livestream ukončen.
- `renameSource` – Aktualizuje název zdroje na objektu SourceEntity se specifikovaným ID.
- `resetSourceKey` – Vygeneruje nový streaming key pro zdroj se specifikovaným ID. Pokud zdroj právě vysílá, pak se změna projeví až při dalším započetí vstupního přenosu.

Pole:

- `owner` – Vrátí objekt UserEntity s uživatelskými informacemi vlastníka daného zdroje.
- `isLive` – Vrátí booleanovou hodnotu reprezentující, zda jsou z tohoto zdroje právě vysílána vstupní data.

StreamResolver

Tento resolver (viz obrázek 5.14) slouží jako proxy mezi GraphQL rozhraním a službou StreamerService. Mutace `startStream` tedy zavolá korespondující funkci `StreamerService#startStream()` a mutace `endStream` zavolá `StreamerService #endStream()`.

5.3.8 Controllery

Controller třídy, anotovány dekorátorem `Controller()`, se ve frameworku NestJS starají o zpracování příchozích HTTP požadavků na specifikovaných URL cestách [50]. Controllery jsou tak typickým způsobem pro zacházení s požadavky v kontextu klasických RESTových rozhraní. V případě GraphQL však ekvivalentní funkcionalitu naplňují již zmiňované resolvery a celá aplikace tak využívá jen jediný controller `AppController`, který se stará pouze o zpracování notifikačních REST požadavků, přicházejících z Gateway serveru nginx. Pomocí nich je API server informován o spuštění a ukončení příchozích RTMP přenosů [51].

V případě spuštění nového přenosu je na API server odeslán POST požadavek na URL `/live/publish`, který ve svém těle obsahuje streaming key zdroje,

na který chce uživatel audiovizuální data doručit. Controller tak nejprve zkontroluje platnost tohoto klíče a zda na tento zdroj již nejsou odesílána data jiného vstupního přenosu. Dále také ověří, že uživatel, jež je vlastníkem daného zdroje, právě nevysílá koncovým divákům, jelikož aplikace momentálně nepodporuje dynamické přidávání nových zdrojů v průběhu živého přenosu. Pokud některá z těchto podmínek selže, pak controller odpoví stavovým kódem 400 Bad Request, čímž dá Gateway serveru najevo, že daný přenos nechce přijmout, a Gateway tak tento RTMP přenos zamítne. V případě úspěchu controller odpoví stavovým kódem 204 No Content a přepne cílový zdroj do stavu ONLINE zavoláním funkce `LiveService#setOnline()`.

V případě ukončení přenosu API server přijme požadavek na URL `/live/done`. Na tento požadavek vždy odpoví stavovým kódem 204 No Content a s přijmutým streaming key zavolá metodu `LiveService#setOffline()`, která se dále postará o přepnutí zdroje do stavu OFFLINE.

5.3.9 Gateways

Gateway třídy s dekorátorem `@WebSocketGateway` fungují v NestJS jako nastavba nad knihovnou Socket.io a starají se o prvotní instanciování WebSocket serveru při spuštění aplikace, navázání a ukončení WebSocket spojení s jednotlivými klienty a následný odposlech příchozích událostí na těchto spojeních [52]. Každá gateway v mojí aplikaci pak spravuje svůj vlastní Socket.io namespace [53].

AppGateway

Třída AppGateway (viz obrázek 5.15) vytváří Socket.io server pro přijímání spojení v namespace `ws`. Tento namespace je použit pro real-time komunikaci pouze s přihlášenými uživateli a pro navázání nového spojení tedy vyžaduje, aby klient poskytnul platný access token. Tento token je serveru předán v hlavičce prvotního HTTP handshake požadavku, který probíhá před upgradem na obousměrné WebSocket spojení. Gateway tedy ověří platnost tohoto tokenu a na základě něj požadavek na upgrade potvrdí nebo zamítne. V případě úspěšného upgrade je pak nově vytvořené spojení zaznamenáno ve službě SocketService zavoláním funkce `SocketService#registerNewConnection()`. V případě odpojení je záznam naopak odstraněn zavoláním funkce `SocketService#unregisterConnection()` a pokud bylo přes tento WebSocket již navázáno souběžné WebRTC spojení, tak je toto spojení také ukončeno metodou `MediaService#destroyTransport()`.

Odposlouchávané události `createTransport`, `deviceRtpCaps` a `connectTransport` pak slouží k navázání nového WebRTC spojení s klientem a tento proces je

započat na straně klienta vysláním události `createTransport`. V případě úspěšného dokončení tohoto procesu budou server i klient obsahovat korespondující `Transport` objekt knihovny `mediasoup`, reprezentující toto vytvořené WebRTC spojení, který je na straně serveru uložen zavoláním metody `MediaService#addTransport()`.

Socket.io události `viewStream` a `resume` pak slouží k tomu, aby si klient vyžádal doručení audiovizuálních dat vybraného zdroje přes již existující `Transport`. Na straně serveru i klienta je tak pro daný `Transport` vytvořen nový `Consumer` objekt konzumující data vyžádaného zdroje. Po vytvoření těchto objektů pak obě strany souběžně zavolají metodu `Consumer#resume()`, čímž je přenos dat zahájen.

Událost `closeAllConsumers` je ze strany klienta zavolána v moment, kdy uživatel opustí obrazovku Studio. V tu chvíli se server postará o uzavření veškerých právě aktivních `Consumer` objektů tohoto klienta, avšak jeho `Transport` je zachován pro budoucí použití. Korespondující `Consumer` objekty jsou zároveň uzavřeny také na straně klienta.

Události `setActive`, `setSolo` a `toggleMuted` nakonec slouží ke správě právě probíhajícího živého přenosu. Událost `setActive` klientovi umožňuje přepnout zdroj, který je právě viditelný koncovému divákovi, událost `setSolo` umožňuje nastavení audio možnosti ”solo” na zvukovou stopu určitého zdroje (případně vypnutí ”solo”, pokud je jako ID zdroje předána hodnota `null`) a `toggleMuted` umožňuje přepnutí mezi slyšitelným a ztlumeným stavem audio stopy daného zdroje.

ChannelGateway

Třída ChannelGateway (viz obrázek 5.16) spravuje WebSocket spojení v namespace `channel` a narozdíl od AppGateway nevyžaduje autentizaci uživatele. Autentizaci je však stále možné provést volitelně. Na straně klienta probíhá připojení k této gateway pouze na obrazovce Channel a slouží k přenosu dat týkajících se živého přenosu konkrétního uživatele. Během připojování WebSocketu je tak v HTTP hlavičce společně s volitelným access tokenem také odesláno uživatelské ID uživatele, na jehož stránce Channel se klient nachází. Server pak při navázání spojení tento socket rovnou přidá do Socket.io místnosti (room) tohoto uživatele. Díky tomu pak může rozesílat události všem divákům daného přenosu najednou.

Tato gateway obsahuje pouze jediný handler události a to pro událost `newMessage`. Tu klient vyšle ve chvíli odeslání nové zprávy do chatu a událost tedy vyžaduje, aby byl socket odesílatele autentizovaný svým access tokenem. Jelikož API momentálně neposkytuje zpětnou historii zpráv v chatu, tak přijatou zprávu není třeba ukládat na straně serveru. Zpráva je tak přímo

přeposlána do místnosti, do které je daný socket připojen, a tím je doručena všem ostatním divákům živého přenosu.

5.3.10 Services

Service třídy jsou v NestJS jednou z dalších specializací providerů, tedy hodnot, které modul poskytuje svým ostatním závislostem skrz dependency injection. Oproti "factory providers" však poskytované hodnoty nejsou definované návratovou hodnotou funkce `useFactory()`, ale třídami anotovanými dekorátorem `@Injectable()` [54]. Tyto třídy jsou pak použity pro zapouzdření určitého výseku funkcionality a poskytovaná funkcionalita je znovupoužitelná mezi závislostmi daného modulu.

AuthService

Třída AuthService (viz obrázek 5.17) poskytuje základní utility funkce pro práci s JWT tokeny. Dostupné funkce tak umožňují vydání nového access a refresh tokenu, ověření platnosti refresh tokenu a blacklistování refresh tokenu. Záznam blacklistu je v databázi Redis reprezentován klíčem "token:{HODNOTA_TOKENU}", ke kterému je přiřazena hodnota prázdného řetězce.

UserService

Služba UserService (viz obrázek 5.18) nabízí znovupoužitelné funkce pro práci s databázovou entitou UserEntity. Pomocí funkce `checkNickTaken()` lze ověřit, zda je konkrétní uživatelské jméno již zabrané jiným uživatelem, a pomocí `getUserByUid()` lze získat objekt UserEntity daného uživatele na základě jeho uživatelského ID. Pokud uživatel s daným ID neexistuje, pak vrácený Promise resolvuje s hodnotou `null`.

SourceService

SourceService (viz obrázek 5.19) nabízí funkci `getSource()`, pomocí které lze získat objekt SourceEntity podle zadaného ID zdroje a uživatelského ID vlastníka tohoto zdroje. Jako ID vlastníka je vždy předáno ID obsažené v payloadu JWT tokenu a je tak zabráněno, aby si uživatel mohl vyžádat informace o zdroji cizího uživatele. Pokud požadovaný zdroj není nalezen, pak vrácený Promise rejectuje s GraphQL chybou "Unable to find source".

MailService

Má aplikace používá e-mailové zprávy při ověřování uživatelských účtů a resetování hesel a jediním z hlavních cílů při tvorbě e-mailových šablon bylo zachování vizuální konzistence mezi těmito e-maily a uživatelským rozhraním samotné aplikace. Tento cíl tak vyžadoval tvorbu stylovaných HTML e-mailů s komplexním layoutem. Hlavní překážkou však byla velmi omezená podpora HTML a CSS ze strany e-mailových klientů a také vysoká úroveň nekonzistence mezi nimi. Pro dosáhnutí maximální podpory jsou tak e-mailové šablony typicky vytvářeny s použitím inlinovaných CSS atributů a samotného rozložení dokumentu je dosáhnuto pomocí HTML `<table>` elementů [39].

V mojí aplikaci jsem se tak pro tvorbu těchto šablon rozhodl využít knihovnu React Email, která funguje na stejném principu jako např. React Native, a umožňuje deklaraci e-mailového rozložení a stylu pomocí poskytnutých React komponent [55]. Vytvořený komponentový strom je touto knihovnou následně vyrenderován jako HTML řetězec, který dodržuje veškerá klasická pravidla pro tvorbu e-mailů a je vysoce kompatibilní s maximálním množstvím e-mailových klientů. Použití React také umožňuje tvorbu a kompozici vlastních znovupoužitelných komponent a obě mé e-mailové šablony tak sdílí základní layout díky komponentě `<EmailLayout>`, do které je konkrétní obsah dané šablony předán jako "prop".

Služba MailService (viz obrázek 5.20) má pak nastarosti zpracování těchto React komponent, jejich vyrenderování a následné odeslání vyrenderovaného obsahu pomocí `Transporter` objektu knihovny `nodemailer` (viz podsekce 5.3.2). Na straně některého z resolverů je tak nejprve zavolána funkcionální React komponenta a jsou jí předány patříčné "props" (např. jméno ověřovaného uživatele a ID ověřovací session). Tato funkce poté vrátí uzel komponentového stromu `JSX.Element` a ten je dále předán funkci `MailService#send()`. Ta se postará o vyrenderování tohoto uzlu pomocí funkce `render()` z balíčku `@react-email/components` a vytvořený textový řetězec je odeslán na uživatelovo emailovou adresu zavoláním metody `Transporter#sendMail()`.

E-mailové šablony lze najít na obrázcích 5.21 a 5.22.

SocketService

SocketService (viz obrázek 5.23) se stará o správu WebSocket spojení a má u sebe uložené obě instance Socket.io serveru: jeden předaný od AppGateway pro namespace `ws` a druhý od ChannelGateway pro namespace `channel`. Nad těmito instancemi pak poskytuje řadu utility funkcí, které mohou ostatní závislosti modulu využívat pro komunikaci s připojenými klienty.

Pro každé právě aktivní WebSocket spojení s gateway AppGateway si SocketService také ukládá záznam o tom, který uživatelský účet je vlasníkem daného spojení. Díky tomu pak může poskytovat metody jako `getUidBySocket()`, která umožňuje získat uživatelské ID na základě poskytnutého ID socketu, `getConnsByUid()`, která vrátí list všech existujících spojení pro daného uživatele, nebo `sendToUser()`, která vyšle událost všem spojením konkrétního uživatele.

LiveService

Třída LiveService (viz obrázek 5.24) má na starosti správu vstupních přenosů přicházejících do zdrojů a každý nový přenos si indexuje pomocí ID odpovídajícího zdroje, jeho streaming key a uživatelského ID vlastníka daného zdroje. Třída pak poskytuje základní getter funkce pro vyhledávání přenosů podle indexovaných klíčů.

Začátek přenosu

V moment započnutí nového přenosu je nejprve třídou AppController zavolána metoda `LiveService#setOnline()`. Tato metoda nejprve inicializuje 2 nové `PlainTransport` objekty knihovny `mediasoup`, nastaví je na odposlech na `localhost` a ke každému z nich vytvoří `Producer` objekt. Jeden `Producer` je nastaven na přijímání obrazových dat ve formátu VP8, druhý přijímá zvuková data ve formátu Opus. Vytvořením těchto objektů je Media server instruován, aby nalezl 4 volné porty, 2 pro každý transport, a začal na nich odposlouchávat příchozí data. Tyto porty jsou použity pro: RTP přenos obrazových dat, RTCP přenos obrazových dat, RTP přenos zvukových dat a RTCP přenos zvukových dat. Protokol RTCP pak slouží jako řídící protokol k odesílání kontrolních zpráv pro svůj korespondující RTP přenos [**?**].

Jakmile jsou tyto porty otevřeny, tak dojde ke spuštění příkazu `ffmpeg`. Ten je nakonfigurován tak, aby konzumoval data příchozího RTMP přenosu, snížil jejich bitrate a rozlišení pro rychlejší doručení ke klientovi, převedl stopy do očekávaných formátů VP8 a Opus a následně je ve formě RTP a RTCP paketů doručil na nově otevřené porty Media serveru. Ke spuštění a správě tohoto `ffmpeg` procesu je využita knihovna `fluent-ffmpeg`; korespodující shell příkaz bych však vypadal takto:

```
ffmpeg -i rtmp://localhost/live/[STREAMING_KEY] -map 0:a:0 -map
↪  0:v:0 -acodec libopus -ab 128k -ac 2 -ar 48000 -af
↪  adelay=delays=0:all=1 -af aresample=async=1 -vcodec libvpx
↪ -b:v 500k -vf scale=640:360  -cpu-used 4 -deadline realtime
↪ -f tee "[select=a:f=rtp:ssrc=11111111:payload_type=101]
↪  rtp://127.0.0.1:[AUDIO_RTP_PORT]?rtcpport=[AUDIO_RTCP_PORT]|
↪  [select=v:f=rtp:ssrc=22222222:payload_type=102]
↪  rtp://127.0.0.1:[VIDEO_RTP_PORT]?rtcpport=[VIDEO_RTCP_PORT]"
```

Po spuštění tohoto příkazu pak server odposlouchává události na vytvořených Transportech, podle kterých pozná, zda byla data z `ffmpeg` již doručena do Media serveru. Jakmile je potvrzeno spojení u všech 4 otevřených portů, tak si LiveService uloží veškerá data týkající se tohoto přenosu (Transporty, Producery, `ffmpeg` proces) a odešle WebSocket událost `status-change`, kterou připojené klienty daného uživatele informuje o tom, že jeden ze zdrojů právě přešel do stavu ONLINE.

Konec přenosu

V moment ukočení přenosu třída AppController zavolá metodu `LiveService#setOffline()`. Tato metoda nejprve zkontroluje, zda daný uživatel právě livestreamuje koncovým divákům, a pokud ano, tak je tento přenos přerušen zavoláním funkce `StreamerService#endStream()`. Aplikace v tuto chvíli nenabízí pokročilé možnosti pro error handling zdrojů a výpadek jednoho ze zdrojů tedy ukončí celý vysílaný přenos.

Dále je všem připojeným klientům odeslána WebSocket událost `status-change`, která je informuje o přechodu daného zdroje do stavu OFFLINE. Následně server ukončí probíhající `ffmpeg` proces tohoto přenosu a zavře veškeré Producery a Transporty.

MediaService

Služba MediaService (viz obrázek 5.20) se stará o správu `mediasoup Transport` a `Consumer` objektů, reprezentujících WebRTC spojení mezi serverem a klientem. Nové Transporty jsou v této službě zaregistrovány zavoláním metody `addTransport()`, která je volána třídou AppGateway po úspěšném vytvoření nového WebRTC spojení. Tato služba si registrované Tranporty a Consumery indexuje podle podle ID socketu, přes který byly vytvořeny.

Hlavní funkcionalitou této služby je pak otevírání nových Consumerů na již registrovaných Transportech a o tento proces se stará metoda `createConsumer()`. Tato metoda je volána také ze strany AppGateway a to v moment přijmutí události `viewStream`. V moment invokace tato metoda nejprve ověří, že je

požadovaný zdroj opravdu ve stavu ONLINE, a následně si vyžádá jeho `Producer` objekty zavoláním funkce `LiveService#getStreamBySource()`. Na Transportu daného klienta je pak zavolána funkce `Transport#consume()`, která slouží k vytvoření nového Consumera na tomto Transportu a které je jako argument předáno ID dotázaného Produceru. Stejně jako v případě Producerů jsou i Consumery vytvořeny vždy po dvou pro každý přenos: jeden pro audio stopu a jeden pro video stopu. Tímto procesem je tedy Media server nakonfigurován tak, aby data přijímaná z lokální instance `ffmpeg` přeposílal vzdáleným klientům přes WebRTC spojení. Nově vytvořené Consumery si pak služba uloží do svého indexu.

Existující Consumery lze ukončit zavoláním metody `closeAllConsumers()` a tato metoda volána třídou AppGateway v moment přijmutí stejnojmené WebSocket události `closeAllConsumers`. Tato metoda nejprve vyhledá veškeré Consumery asociované s daným socketem, každý z nich uzavře a následně je odstraní z indexu. K úplnému uzavření Transportu slouží metoda `destroyTransport()` a ta je zavolána ve chvíli, kdy se socket odpojí od AppGateway.

StreamerService

Cílem služby StreamerService (viz obrázek 5.26) je správa právě probíhajících livestreamů ke koncovým uživatelům a správa pravě běžících instancí programu Streamer (viz sekce 5.4) asociovaných s těmito přenosy. Informace o livestreamech jsou indexovány podle uživatelského ID jejich tvůrce.

Spuštění přenosu

Ke spuštění nového livestreamu slouží metoda `startStream()`, která je zavolána z resolveru StreamResolver a jsou jí předány informace potřebné k jeho spuštění. Těmi jsou název livestreamu, stav všech zdrojů, ze kterých je tento livestream složen (který zdroj je aktivní / které zdroje jsou ztlumené / který zdroj má "solo"), a specifikace všech externích destinací, na které má být livestream odesílán (dvojice RTMP URL a streaming key).

Jako první metoda zkontroluje, zda jsou veškeré specifikované zdroje ve stavu ONLINE, a následně na základě předaných dat sestaví příkaz pro spuštění programu Streamer. Tento příkaz je poté spuštěn Node.js funkcí `child_process.exec()` a na jeho standardní výstup je přidán event listener čekající na zprávu "ready" (viz sekce 5.4). Jakmile program Streamer vypíše tuto zprávu, tak je všem divákům právě se nachazejícím na obrazovce Channel vysílajícího uživatele odeslána událost `streamStart`. V reakci na ní může klientská aplikace začít zobrazovat živý přenos ve formátu HLS.

Ukončení přenosu

Zavolání funkce `endStream()` ukončí právě probíhající livestream specifikovaného uživatele. Nejprve je tak všem sledujícím klientům odeslána WebSocket událost `streamEnd`, informující je o konci přenosu, a následně je na standardní vstup běžícího Streamer procesu vepsána zpráva `exit`. V reakci na ní tento proces uzavře svou mediální pipeline a ukončí se s stavovým kódem 0. Metoda poté odebere veškeré informace o ukončeném livestreamu z indexu služby StreamerService.

Ostatní

Tato třída poskytuje také metody `setActive()`, `setSolo()` a `toggleMuted()`, které slouží ke úpravě přenosu v průběhu vysílání. Každá tato metoda při svém zavolání nejprve nalezne Streamer proces odpovídající požadovanému livestreamu a následně tomuto procesu odešle zprávu přes standardní vstup.

5.4 Streamer

Projekt Streamer slouží jako spojovací prvek mezi API serverem a frameworkem GStreamer a jeho úkolem je real-time správa dynamické mediální pipeline v průbehu probíhajícího přenosu. Právě tento projekt se tak stará o doručení zpracovaného přenosu koncovému uživateli a to jak ve formě HLS přenosu, který je využit v rámci vestavěného livestream přehrávače této aplikace, tak i ve formě RTMP přenosu, který je odesílán na streamovací platformy třetích stran jako YouTube nebo Twitch.

Framework GStreamer jako takový pracuje na bázi zdrojů (source) a spotřebičů (sink), kde každý zdroj slouží jako generátor multimediálních dat a každý spotřebič jako vstup pro určitý zpracovávající prvek. Konfigurace pipeline tak funguje na bázi tvorby orientovaného grafu, kterým data protékají od vstupních uzlu až po uzly koncové a každý uzel příchozí data určitým způsobem transformuje [38].

GStreamer v rámci svých nástrojů nabízí i CLI utilitu, díky které lze pipeline definovat přímo ve formě příkazových argumentů, podobně jako je tomu například u nástroje ffmpeg. Tento přístup je však limitován na tvorbu statických pipeline a není s ním tedy možné implementovat režisérskou funkcionalitu pro přepínání mezi zdroji nebo ztlumení audio stopy v průběhu živého přenosu. Projekt Streamer tak tuto pipeline vytváří programaticky a umožňuje její následovnou manipulaci za chodu.

Ačkoliv GStreamer samotný je implementován v jazyce C, tak využívá open-source knihovnu GObject, která poskytuje interoperabilitu pro další programovací jazyky. Můj projekt Streamer je tak napsaný v jazyce Python a jeho integrace s frameworkem GStreamer probíhá skrz balíček PyGObject.

Nová instance tohoto Python programu je API serverem naspawnována při každém započetí livestreamu a jsou jí předány příkazové argumenty popisující počáteční stav tohoto streamu. Následná komunikace s touto instancí poté probíhá přes standardní vstup.

Příkaz pro spuštení programu má nasledující strukturu:

```
[exec_cmd] [stream_id] [active_src_index] [solo_index]
↪  [src_list] --dest [dest_list]
```

- **[exec_cmd]** - Příkaz ke spuštění programu. Je nakonfigurován v API serveru pomocí proměnné prostředí `STREAMER_CMD`.
- **[stream_id]** – Náhodné ID, které API server vygeneruje v moment počátku livestreamu. Určuje cestu pro ukládání statických souborů HLS přenosu (např. URL playlistu bude `/static/live/[stream_id]/playlist.m3u8`).
- **[active_src_index]** – Index do seznamu [src_list]. Určuje, který zdroj je na počátku aktivní (tj. obraz kterého zdroje bude viditelný koncovému divákovi).
- **[solo_index]** - Index do seznamu [src_list] nebo hodnota "n" (NULL). Nastavuje SOLO na audio stopu vybraného zdroje (pokdu je NULL, tak SOLO není aktivní).
- **[src_list]** – Seřazený seznam zdrojů. Každá položka v seznamu se zkládá ze 2 mezerou oddělených hodnot: streaming key daného zdroje a stavu jeho audio stopy. Audio stav může nabývat hodnot "n" (NULL – tj. audio stopa je slyšitelná) a "m" (MUTED – tj. audio stopa je ztlumená).
- **–dest** – Oddělovač, který ukončuje seznam [src_list] a započíná seznam [dest_list]. Tato hodnota není vyžadována, pokud livestream není odesílán na žádné externí RTMP endpointy. V takovém případě není potřeba specifikovat ani oddělovač "–dest", ani následný seznam [dest_list].
- **[dest_list]** – Seznam externích RTMP endpointů, na které se má přenos odesílat. Každá položka je URL protokolu RTMP.

V praxi by tak výsledný příkaz pro spuštění mohl vypadat například takto:

```
python main.py abcd 0 n src1key n src2key m src3key n --dest
↪  rtmp://a.rtmp.youtube.com/live2/ytkey
```

Na základě tohoto příkazu program sestaví GStreamer pipeline v svém počátečním stavu. Graf této pipeline lze nalézt na obrázku 5.27.

V prvním kroku jsou všechny příchozí RTMP přenosy načteny pomocí prvku `rtmpsrc` a jeho výstup je následně rozdělen na samostatnou video a audio stopu prvkem `flvdemux`. Obě stopy jsou následně dekódovány do "raw" datového toku a jsou dále zpracovány. V případě video stopy doje k úpravě velikosti obrazu na 1280x720 pixelů a převodu snímkové frekvence na 25fps. Audio stopa je převedena na 2 kanálové audio o vzorkovací frekvenci 48kHz.

Zpracované video stopy jsou pak zavedeny do prvku `input-selector` a audio stopy do prvku `audiomixer`. Právě tyto prvky se starají o real-time přepínání aktivních přenosů a mixování audio stop a na počátku jsou tedy nastaveny podle zadaných argumentů programu.

Aktivní video stopa pak dále putuje do prvku `x264enc`, kde je enkódována do kódování H.264 a následně zavedena do prvku `tee`, který slouží jako rozbočovač. Jedna větev je pak použita pro tvorbu HLS přenosu a druhá pro tvorbu výstupních RTMP přenosů. Jelikož oba typy muxerů vyžadují odlišný typ kódování datového toku H.264 [38], tak ještě obě větve prochází prvkem `h264parse`, který datový tok upraví dle požadavků daného formátu. V případě HLS přenosu je pak video stopa zavedena přímo do prvku `hlssink2`, který slouží zároveň i jako muxer a tedy spojí video a audio stopu dohromady a začne generovat statické multimediální soubory. Pro RTMP přenos je video stopa zavedena do muxeru `flvmux`, kde vzniká výsledný odchozí stream a ten je poté odeslán na externí RTMP servery prvkem `rtmpsink`.

Smíchaná audio stopa je oproti tomu převedena do kódování AAC a další zpracování již nevyžaduje. Tento datový tok je tak již přímo zaveden do prvků `hlssink2` a `flvmux`.

Po sestavení a spuštění pipeline se pak program přesune do "event loop" smyčky, kde odposlouchává přicházející GStreamer eventy a čte příkazy ze standardního vstupu. Čtení z `stdin` bylo tedy potřeba realizovat neblokujícím zpsůobem, aby nedošlo k zablokování event loop. Za tímto účelem bylo použito POSIXové systémové volání `fcntl()`, pomocí kterého je file descriptoru `stdin` nastavena vlajka `O_NONBLOCK` [40]. Python modul `selectors` je poté použit k odposlechu eventu `READ` na tomto file descriptoru a v případě jeho nastání zavolá callback funkci `accept_cmd()`, která přečte a parsuje příkaz na vstupu a podle něj dynamicky rekonfiguruje běžící pipeline. Z důvodu použití volání `fctnl()` tedy tento program, oproti zbytku aplikace, není přenositelný na nePOISXové operační systémy jako například Windows.

Program momentálně podporuje 4 různé příkazy na standardním vstupu, kterými jsou:

- `switch [active_src_index]` – Přepne aktivní video stopu na prvku `input-selector`.
- `solo [solo_index]` - Pokud je zadaný index zdroje, tak ztlumí audio stopy všech ostatních zdrojů kromě vybraného. Pokud je zadána hodnota ”n”, tak je solo deaktivováno a všechny audio stopy se vrátí do svého normálního stavu (tj. slyšitelné na základě toho, zda je daný zdroj MUTED nebo ne).
- `mute [src_index]` – Přepíná stav MUTED pro vybraný zdroj (tj. pokud je zdroj slyšitelný, tak se přepne na MUTED; pokud je MUTED, tak se přepne na slyšitelný).
- `exit` – ”Gracefully” ukončí pipeline. Odešle signál EOS (End Of Stream) do video a audio enkodérů (prvky `x264enc` a `avenc_aac`) a tím způsobí EOS celého datového toku a následné ukončení programu.

Kromě přijímání příkazů na vstupu však bylo také potřeba, aby program nazpět informoval o stavu HLS přenosu. Přehrávání na straně koncového uživatele může totiž započít teprve v moment, kdy je dokončeno generování prvního chunku a společně s tím je vygenerován playlist ve formátu m3u8. Do té doby není možné přenos načíst a pokus na straně frontendu by vyústil v chybu přehrávače. Event loop tedy mimo jiné odposlouchává i GStreamer událost `splitmuxsink-fragment-opened`, která je vyslána vždy při vytvoření nového HLS chunku. Druhé vyslání této události tedy znamená, že `hlssink2` právě začal generovat druhý chunk a v tu chvíli je tak první chunk již dokončený a společně s ním je vygenerovaný i playlist. Program tak v tuto chvíli vypíše zprávu `ready` na svůj standardní výstup a API server při jejím přečtení aktualizuje stav přenosu tak, aby ho koncový divák mohl sledovat ve vestavěném přehrávači.

5.5 Client

5.5.1 Routing

App Router architektura v Next.js využívá pro definici cest takzvaný ”file-system routing” přístup, ve kterém jsou jednotlivé cesty aplikace definovány hierarchií souborů a složek umístěných uvnitř složky `app`. Název každé složky pak reprezentuje jeden konkrétní segment URL cesty a uvnitř každé složky se mohou nacházet speciální soubory `layout.tsx` a `page.tsx`.

Soubor `layout.tsx` představuje sdílené UI rozhraní, které se vztahuje nejen na jednu konkrétní cestu, ale také na veškeré potomky této cesty. Layout cesty

`/example` tak bude například vykreslen i na stránce `/example/user`. Layoutové komponentě je pak ze strany Next.js automaticky předán prop `children`, který reprezentuje již specifický obsah konkrétní stránky, který má být v tuto chvíli vyrenderován uvnitř tohoto layoutu. Soubor `page.tsx` pak reprezentuje přímo obsah konrkétní stránky a tento obsah je tedy zaobalen všemi nadřazenými layouty nad touto cestou.

Názvy vytvořených složek pak v typickém případě přímo odpovídají názvům segmetů URL cesty. Kromě toho však Next.js definuje také několik speciálních případů. Pro mou aplikaci jsou pak konkrétně relevantní složky, jejichž jméno je zaobaleno kulatými nebo hranatými závorkami (např. `(example)` nebo `[example]`).

Složky s kulatými závorkami představují takzvanou "route group", což je v Next.js organizační jednotka pro seskupení několika různých stránek bez ovlivnění jejich cesty. Stejně jako pro normální cestu může být i pro route group definována její vlastní layout komponenta, a díky tomu tak může několik stránek sdílet stejné uživatelské rozhraní, aniž by museli začínat sdíleným prefixem URL cesty. Například obsah komponenty `app/(my-group)/example/page.tsx` by tak byl ve výsledné aplikaci dostupný na cestě `/example`.

Oproti tomu, složky s hranatými závorkami definují dynamické parametry cesty a text mezi těmito závorkami slouží jako jméno daného parametru. Pod těmito jmény jsou pak jednotlivé parametry uloženy do objektu `params`, který je předán jako prop všem stránkám podřazeným této dynamické cestě. Například komponenta uvnitř souboru `app/user/[slug]/page.tsx` by tak byla dostupná na URL cestách začínajících prefixem `/user/` a pokud k ní uživatel přistoupil třeba z adresy `/user/test`, tak by jí byl předán `params` objekt `{ slug: "test" }`.

S pomocí těchto konstruktů je pak definována celá hierarchie mojí aplikace, která je vizualizována na obrázcích 5.28 a 5.29.

5.5.2 Middleware

Middleware v Next.js slouží ke zpracování a úpravě příchozího HTTP požadavku ještě před začátkem SSR. Oproti React Server Components (RSC) pak nabízí výrazně větší možnosti pro manipulaci s požadavkem i se serverovou odpovědí, jelikož komponenty oproti middlware nemají přímý přístup k serverovému `Request` objektu a jejich render nesmí způsobit žádné vedjější účinky (např. nastavení cookie). Middleware je v Next.js reprezentován pouze jedinou funkcí, kterou lze umístit do souboru `middleware.ts` v kořenové složce zdrojového kódu.

Můj middleware pak nejprve přečte URL adresu příchozího požadavku a

uloží ji do header záznamu `x-url`. Zatímco serverové komponenty totiž přímý přístup k URL adrese nemají, tak header záznamy oproti tomu číst mohou. Jedná se tak o hlavní způsob pro předávání dat mezi middleware a RSC. Následně server zkontroluje záznamy cookies a pokud požadavek obsahuje platný refresh token, ale platnost access tokenu již vypršela, tak se server pokusí o jeho obnovení. V případě úspěchu je pak token zapsán do cookie `access`.

5.5.3 Skupina NoAuth

V této podsekci se zaměřím na komponenty obsažené uvnitř route group složky `(noauth)` a vztahy mezi nimi. U této route group jsem se pak zejména snažil o maximální využití RSC a pouze minimální subset interaktivních prvků tedy vyžaduje klientský kód ke svému fungování.

Layout

Komponenta NoAuthLayout představuje layout pro celou route group `(noauth )` a je tak vyrenderována při navigaci na kteroukoliv ze stránek uvnitř této skupiny. Jejím hlavním cílem je pak zamezit návštěvě těchto stránek, pokud je uživatel již přihlášený ke svému uživatelskému účtu. Stránky obsažené v této skupině se tak týkají jen funkcionality nepřihlášeného uživatele. Tato komponenta je implementována jako RSC a během renderování na straně serveru kontroluje, zda cookies příchozího požadavku obsahují platný access token a uživatel je tedy přihlášen k aplikaci. Pokud ano, tak k vyrenderování požadované stránky vůbec nedojde a místo toho server odpoví stavovým kódem 307 Temporary Redirect, pomocí kterého je uživatel přesměrován na URL `/studio`.

Mimo svých potomků (prop `children`) tato komponenta renderuje také klientskou komponentu `<LoginRedirect>`. Ta má nastarosti vymazání Zustand storu `useLoginRedirectStore` (viz 5.5.3) ve chvíli, kdy uživatel naviguje pryč z této route group (např. v moment přihlášení).

Přihlašovací přesměrování

Tato funkcionalita musela být implementována kvůli fungování obrazovky Channel, kterou může navštívit i nepřihlášený uživatel a sledovat zde probíhající přenos, avšak bez přihlášení nemůže odesílat zprávy do chatu. Během tohoto stavu pak horní navigační lišta místo typického profilového obrázku zobrazuje tlačítko Login, pomocí kterého může uživatel přejít na přihlašovací

stránku. Po dokončení přihlašovacího procesu je však uživatel za normálních okolností přesměrován na obrazovku Studio a na původně navštívenou obrazovku Channel by se tedy musel vracet ručně.

Z toho důvodu byl tedy do aplikace přidán mechanismus pro změnu navigované stránky po přihlášení. Při stisku tlačítka Login na obrazovce Channel je tak nejprve do Zustand store `useLoginRedirectStore` uložena právě navštívená URL adresa a teprve poté dojde k přesměrování na stránku Login. V moment přihlášení je pak uživatel přesunut zpět na patřičnou obrazovku Channel a Zustand store je vyprázdněn komponentou `<LoginRedirect>` (viz 5.5.3).

Úvodní zpráva

Optimální serverová implementace se ukázala jako zejména obtížná u stránek Verify a Reset. U stránky Verify bylo mým cílem celý ověřovací krok provést plně na straně serveru a serverová komponenta tak s předaným parametrem `slug` zavolá GraphQL mutaci `verify()` a uživatele rovnou přesměruje na přihlašovací stránku stavovým kódem 307 Temporary Redirect. Reset stránka oproti tomu obsahuje uživatelské rozhraní pro zadání nového hesla; zde jsem však chtěl dosáhnout okamžitého přesměrování, pokud ID resetovací session není platné.

Kromě samotného přesměrování však bylo potřeba, aby následně zobrazená přihlašovací stránka ihned po načtení vykreslila potvrzovací nebo chybovou zprávu, která by uživatele informovala o úspěšném ověření účtu nebo chybně zadaném ID ověřovací nebo resetovací session. Cílová URL adresa tohoto přesměrování kromě samotné cesty obsahuje také query string s klíčem `toast`, uvnitř kterého se nachází serializovaný JSON objekt ve formátu:

```
{
    msg: string;
    err: boolean;
}
```

Uvnitř layoutové komponenty přihlašovací obrazovky (LoginLayout) se pak mimo jiné nachází i klientská komponenta `<IntialToast>`. Té jsou tímto serverovým layoutem předány informace o požadované úvodní zprávě, načtené právě z tohoto query stringu. Tato komponenta pak ihned po hydrataci vytvoří informační ”toast” zprávu pomocí knihovny React Hot Toast a provede client-side přesměrování, pomocí kterého query string odstraní.

5.5.4 Skupina Main

Route group (main) shromažďuje stránky aplikace určené pro přihlášeného uživatele. V tuto chvíli jsou to tak stránky Studio a Settings; během budoucího vývoje by se sem měla přidat také stránka Editor.

Layout

Hlavní layout této skupiny, MainLayout, je implementovaný ve formě serverové komponenty a kromě sdíleného uživatelského rozhraní má na starosti také celkovou inicializaci aplikace na straně klienta. Všechen obsah tohoto layoutu je tak zaobalen třemi klientskými komponentami: `<WebSocketProvider>`, `<WebRtcProvider>` a `<StoreInitializer>`. Uvnitř těchto wrapper komponent se pak layout stará o vykreslení horní navigační lišty, pod níž je umístěný již specifický obsah navštívené stránky.

WebSocketProvider

Komponenta WebSocketProvider se ihned po namountování pokusí navázat spojení s WebSocket bránou AppGateway na straně API serveru (viz podsekce 5.3.9). Jelikož toto spojení vyžaduje autentizaci uživatele, tak je v hlavičce prvotního požadavku také odeslán uživatelův access token, který komponenta získá zavoláním utility funkce `getAutorefreshed()`. Tato funkce pak před vrácením tokenu nejprve ověří jeho platnost a pokud již expiroval, tak se pokusí o jeho automatické obnovení. Požadavek na WebSocket spojení je tak vždy odeslán z platnou verzí access tokenu.

V případě úspěšného navázání spojení jak pak vytvořená instance `Socket` objektu uložena do Zustand storu `useWebSocketStore`, ze kterého mohou k soketu přistupovat ostatní komponenty. V případě opuštění skupiny Main a společně s tím unmountování komponenty WebSocketProvider se pak tato komponenta postará o uzavření WebSocket spojení, zničení vytvořeného soketu a vyprázdnění storu `useWebSocketStore`.

WebRtcProvider

Komponenta WebRtcProvider svou funkcionalitou závisí na nadřazeném WebSocketProvider a jejím cílem je využít navázané WebSocket spojení k tomu, aby navázala spojení přes WebRTC a vytvořila tak nový `Transport` objekt knihovny `mediasoup-client`. Komponenta tak nejprve čeká do chvíle, kdy je do storu `useWebSocketStore` uložena nová instance soketu a následně tento soket využije k provedení celého WebRTC handshake pomocí WebSocket událostí `createTransport`, `deviceRtpCaps` a `connectTransport` (viz podsekce 5.3.9). V případě úspěšného vytvoření WebRTC transportu je pak tento objekt uložen

do Zustand storu `useWebRtcStore`. V případě unmountování komponenty je tento transport uzavřen a store vymazán.

StoreInitializer

Zatímco obsah stránek obsažených ve skupině NoAuth byl zcela statický a jeho renderování na straně serveru bylo tedy triviální záležitostí, tak u skupiny Main (zejména stránky Studio) jsou možnosti pro serverové renderování výrazně více omezené. Hlavním důvodem je vysoká přizpůsobitelnost uživatelského rozhraní, která pro své fungování využívá řadu persistentních Zustand stores, jejichž obsah je kontinuálně zálohován v prohlížečovém úložišti `localStorage`. Tyto stores pak obsahují veškeré informace potřebné k popisu UI konfigurace právě přihlášeného uživatele.

Vzhledem k tomu, že tato data existují pouze v prohlížeči na straně klienta, tak není možné, aby bylo cílové uživatelské rozhraní vyrenderováno již během SSR. Ihned po načtení stránky je tedy nutné, aby byly veškeré požadované stores hydratovány a jejich předchozí stav byl obnovený z `localStorage`. Právě to je tedy cílem komponenty StoreInitializer, která se okamžitě po namountování postará o načtení stavu pro stores `useSidebarStore`, `useLiveStore` a `useFloorplanStore`. Jelikož je pak vytvořená UI konfigurace specifická pro konkrétního uživatele, tak k tomuto obnovení stavu musí dojít pokaždé, když se k aplikaci přihlásí nový uživatelský účet. Pokud se tedy uživatel v průběhu jedné návštěvy odhlásí ze svého účtu a poté se znovu přihlásí, tak StoreInitializer již nemusí provádět žádnou akci. Pokud se však přihlásí k účtu jinému, tak jsou veškeré stores rehydratovány s novými údaji.

Zbytek aplikace pak může stav hydratace sledovat pomocí Zustand store `useInitializerStore`, který definován přímo vedle komponenty StoreInitializer a obsahuje data ve formátu:

```
{
    isSidebarLoaded: boolean;
    isLiveLoaded: boolean;
    isFloorplanLoaded: boolean;
}
```

Komponenty, které na načítaných stores závisí, tak mohou `useInitializerStore` odebírat a pomocí něj čekat do doby, kdy je hydratace dokončena. Části aplikace jako postranní panel Studia, správce oken nebo Floorplan Editor jsou tak během SSR vyrenderovány pouze s načítacím spinnerem a ten je následně na straně klienta nahrazen skutečným UI, jakmile jsou požadované stores načteny z `localStorage`.

Studio - Layout

O definici layoutu stránky Studio se stará serverová komponenta StudioLayout. Uvnitř ní je jako první obsažena komponenta `<ActivationGuard>` a pod ní se nachází komponenta `<FloorplanSwitcher>`. Samotný UI obsah stránky je pak komponentě `<FloorplanSwitcher>` předán jako její potomek a skládá se z postranního panelu v komponentě `<StudioSidebar>` (viz podsekce 5.5.5) a samotného obsahu stránky Studio, obsaženého v souboru `page.tsx` (viz podsekce 5.5.5).

ActivationGuard

Část funkcionality, vztahující se k přehrávání multimediálního obsahu, je prohlížeči ihned po načtení stránky blokována, a to až do doby, kdy uživatel provede první interakci (např. kliknutí). Hlavním příkladem je zejména automatické přehrávání obsahu se zvukem [56]. Abych se tak těmto limitacím zcela vyhnul, tak se komponenta ActivationGuard stará o vynucení prvotní interakce na stránce Studio.

Tato komponenta tak nejprve pomocí prohlížečového User Activation API zkontroluje, zda již se stránkou proběhla dřívější interakce [57]. Pokud ne, tak je celý viewport stránky překryt poloprůhlednou overlay vrstvou s nápisem "Press anywhere" a načítání multimediálního obsahu přes WebRTC je odloženo. Teprve po stisku je pak tato vrstva skryta a multimediální obsah je spuštěn i se zvukem. V moment aktivace je také vyrenderována komponenta `<KeybindHandler>`.

KeybindHandler

Komponenta KeybindHandler implementuje funkcionalitu pro vytváření vlastní klávesových zkratek, sloužících pro přepínání aktivního zdroje. Tato komponenta tak v moment namountování přidává event listener funkce na globální objekt `Window`, pomocí kterých odposlouchává údálosti `keydown`, `keyup` a `blur`. Na základě těchto událostí si pak udržuje aktualizovaný set všech právě stisknutých kláves.

Za normálních okolností komponenta při každém novém stisku klávesy provádí detekci zkratek. Stisky jsou však ignorovány pokud uživatel právě zapisuje do polí `<input>` nebo `<textarea>`. Záznamy o jednotlivých nakonfigurovaných zkratkách se pak nachází uvnitř Zustand store `useLiveStore` a pro účely serializace do `localStorage` jsou reprezentovány ve formě textového řetězce, který obsahuje seřazenou sekvenci klávesových kódů, delimitovaných znakem `NULL`.

Set právě stisknutých kláves je tak nejprve převeden do odpovídajícího formátu a výsledný řetězec je předán funkci `matchBind()` uvnitř `useLiveStore`.

Ta se postará o prohledání všech záznamů a pokud nalezne odpovídající zkratku, tak aktualizuje ID aktivního zdroje. Jelikož pořadí položek uvnitř setu není garantováno, tak je během převodu na řetězec také potřeba veškeré kódy deterministicky seřadit, aby porovnávané řetězce přesně odpovídaly. Set kláves je tak nejprve převeden na pole a to je následně seřazeno funkcí `sortKeys()`.

Aby bylo porovnávání nezávislé na právě nastaveném rozložení klávesnice, tak jsou zkratky místo znaků kláves tvořeny univerzálními klávesovými kódy. Pro vizualizaci zkratek v rámci UI pak aplikace obsahuje mapování mezi kódy všech povolených kláves a jejich reprezentací na anglické QWERTY klávesnici. V ideálním případě by bylo zkratky možné dynamicky namapovat na konkrétní klávesový layout, který má daný uživatel právě nastavený. Toho by šlo dosáhnout pomocí funkce `getLayoutMap()`, obsažené v rámci prohlížečového Keyboard API. Tato funkce je však bohužel v tuto chvíli považována za experimentální a její podpora v prohlížečích je značně omezená [58]. Zatím jsou tak zkratky vizualizovány pouze na základě anglického QWERTY rozložení.

Komponenta KeyboardHandler může být také přepnuta do režimu pro nahrávání nových zkratek, což je provedeno zavoláním funkce `record()` uvnitř store `useKeyStore` a této funkci je předáno ID zdroje, pro který je nová zkratka nahrávána. Od chvíle přepnutí tak komponenta neprovádí jakoukoliv detekci a nové stisky jsou místo toho zaznamenány do vytvářené zkratky. Nahrávání je automaticky ukončeno ve chvíli, kdy je upuštěna první klávesa (událost `keyup`). Vytvořená zkratka je poté uložena do `useLiveStore` pomocí funkce `registerBind()`. Pokud právě nahraná zkratka přesně odpovídá již v minulosti použité zkratce, tak je původní záznam odstraněn ze store.

FloorplanSwitcher

Komponenta FloorplanSwitcher se stará o maximalizaci obrazovky Floorplan Editor ve chvíli, kdy je uvnitř kompaktního Floorplan náhledu stisknuto tlačítko Edit. V moment stisku je tak do store `useFloorplanActivation` zapsána hodnota `{ isActive: true }` a komponenta v reakci na to skryje veškerý obsah stránky Studio (předaný této komponentě jako její potomek) a místo něj zobrazí maximalizovanou verzi komponenty `<FloorplanEditor>`. Při stisku návratové šipky v postranním panelu editoru je pak do `useFloorplanActivation` opět zapsána hodnota `{ isActive: false }` a maximalizovaný editor je tedy znovu ukryt a Studio se vrátí do původního stavu.

5.5.5 Studio - Multimédia

Multimediální obsah zdrojů je na stránce Studio spravován pomocí dvou různých Zustand stores: `useLiveStore` a `useMediaStore`.

useLiveStore

Store `useLiveStore` slouží k ukládání veškeré konfigurace zdrojů a oproti `useMediaStore` je persistentní. Jeho data jsou tak kontinuálně synchronizována s úložištěm `localStorage`. Uvnitř tohoto store se nachází záznam pro každý právě existující zdroj a v tomto záznamu jsou uloženy informace o jeho stavu ve formátu:

```
{
    muted: boolean;
    cued: boolean;
    binds: string[];
}
```

`useLiveStore` si tak ukládá serializované záznamy všech uživatelem definovaných klávesových zkratek daného zdroje (viz podsekce 5.5.4 - KeybindHandler) a také informace o tom, zda jsou u zdroje aktivována tlačítka MUTE a CUE.

Kromě toho tento store ukládá také ID právě aktivního zdroje a případně ještě ID zdroje, u kterého je aktivováno tlačítko SOLO (obsahuje `null`, pokud možnost SOLO není použita). Dále je store synchronizovaný se sekcí Config v postranním panelu Studia a jsou v něm tak uložena prohlížečem poskytnutá ID audio výstupů Master Out a Monitor Out a také uživatelova konfiguraci možnosti "Solo source on switch".

Celkově tak tento store reprezentuje uživatelovo konfiguraci celého vysílání; nestará se však o samotné přehrávání korespondujícího multimediálního obsahu. Tento store však úzce spolupracuje se store `useMediaStore`, který kontinuálně informuje o změnách v konfiguraci Studia.

useMediaStore

Tento store má na starosti samotnou správu multimediálního obsahu vysílaného pomocí spojení WebRTC. Při načtení Studia jsou tak nejprve veškeré zdroje ve stavu ONLINE inicializovány pomocí funkce `initFromSources()` uvnitř tohoto store. Při změně stavu zdroje pak mohou být nové přenosy otevřeny pomocí funkce `initById()` nebo ukončeny pomocí `deinitById()`. Při úplném opuštění stránky Studio jsou přenosy veškerých zdrojů uzavřeny funkcí `deinitAll()`.

Při otevírání nového spojení pro každý zdroj jsou nejprve vytvořeny dva nové `Consumer` objekty na existujícím Transportu knihovny `mediasoup-client` (viz podsekce 5.5.4 - WebRtcProvider), jeden pro audio stopu a jeden pro video stopu. Oba tyto Consumery jsou poté spuštěny a ze získaných `MediaStreamTrack` objektů je sestavený fínání `MediaStream` objekt [30], obsahující audio i video

stopu. Ten je následně nastaven jako zdroj pro nově vytvořený `HTMLVideoElement` a dva nové `AudioContext` objekty.

Vytvořený `HTMLVideoElement` je sám o sobě nastavený jako ztlumený a zbytku aplikace slouží jako zdroj obrazových dat daného zdroje. Všechna náhledová okna zdrojů fungují tedy tak, že samotné okno obsahuje pouze element `<canvas>` a do něj jsou obrazová data každý snímek překopírována. Díky tomu jsou tak všechna okna zobrazující stejný obsah perfektně synchronizována a uživatelem vytvořené layouty mohou být instantně přepnuty bez jakékoliv prodlevy, jelikož aplikace nemusí inicializovat nový `<video>` element pro každé jednotlivé okno.

Vytvořené `AudioContext` objekty se pak samostatně starají o přehrávání audio stopy vytvořeného `MediaStream` objektu, jeden na audio výstupu Master Out a druhý na výstupu Monitor Out. Uvnitř těchto AudioContextů je poté vytvořena pipeline pro zpracování zvuku, uvnitř které audio prochází uzlem `GainNode` [59]. Pomocí těchto uzlů, jež umožňují úpravu hlasitosti zvuku, je pak implementována funkcionalita MUTE, SOLO a CUE.

Studio - Postranní panel

Postranní panel na stránce Studio je implementovaný komponentou `<StudioSidebar>` a skládá se z náhledu živého přenosu, formuláře pro spuštění a ukončení přenosu a tří konfiguračních sekcí: Sources, Destinations a Config. Všeobecný stav tohoto panelu je uložen uvnitř persistentního store `useSidebarStore` a nastavení v sekci Config jsou synchronizována s `useLiveStore`. Tato komponenta má na starosti také prvotních stažení všech uživatelovo zdrojů z API, které následně zobrazuje uvnitř sekce Sources a také pomocí nich inicializuje mediální přenosy zavoláním funkce `initFromSources()` uvnitř `useMediaStore`.

Jednotlivé sekce tohoto panelu jsou reprezentovány komponentou `<PaneView>`, jejíž chování a vzhled jsou inspirovány postranním panelem editoru Visual Studio Code, a rozložení těchto sekcí je vysoce uživatelsky přizpůsobitelné. Při prvním příhlášení nového uživatele tak sekce Sources, Destinations a Config vyplní volnou výšku postranního panelu v poměru 1:1:0. Sekce Config tak začíná zcela skrytá a sekce Sources a Destinations si výšku rozdělují rovnoměrně.

Nad obsahem každé sekce se pak nachází její hlavička, pomocí které lze vyplnění postranního panelu upravit (viz obrázek 5.30). Potažením za "handle" u horní hrany této hlavičky může uživatel změnit velikost dané sekce; kliknutím na ní lze sekci zcela kolapsovat nebo znovu expandovat. Při změně velikosti okna prohlížeče sekce vždy zachovávají poměr výšek a chování je navrhnuto tak, aby nikdy nedošlo k implicitnímu kolapsu nebo expanzi (tj. sekce nikdy

neexpanduje nebo nekolapsuje v jiném případě, než když uživatel explicitně kline na její hlavičku). Chování při kolapsu a expanzi je pak následující:

- Kolaps - Pokud pod kolapsovanou sekcí již není žádná další sekce, která by byla expandovaná (tj. tato sekce je buď nejvíce spodní sekcí v postranním panelu nebo jsou veškeré níže umístěné sekce již také kolapsované), tak je sekce kolapsována směrem dolu a její místo je předáno nejblížší horní sekci, která je expandovaná. V opačném případě sekce kolapsuje směrem nahoru a prostor je předán neblížší spodní sekci. Pokud byla tato sekce úplně poslední expandovanou sekcí v celém panelu a její prostor tedy není možný předat jiné expandované sekci, tak je všechen prostor postranního panelu předán práznému filler elementu, umístěnému pod všemi sekcemi (tj. kolapsované hlavičky všech sekcí se přesunou k horní hraně postranního panelu) (viz obrázek 5.31).
- Expanze - Pokud je filler element aktivní, tak je sekce expandována na plnou výšku postranního panelu. V opačném případě je výška expandována na poměr 1/N, kde N je celkový počet existujících sekcí. Sekce se vždy nejprve pokusí expandovat směrem dolu a převzít velikost od nejblížší spodní sekce o větší velikosti než 1/N. Pokud taková sekce neexistuje (tj. všechny níže umístěné sekce jsou zcela kolapsované nebo zmenšené na nižší velikost), pak sekce expanduje směrem nahoru a převezme velikost od nejbližší horní sekce o velikosti větší než 1/N. Výhodou expanze právě na velikost 1/N je pak zejména to, že je vždy během expanze zaručna existence aspoň jedné sekce, která má výšku větší než tento poměr, a lze jí tedy potřebný prostor odebrat a předat ho nově expandované sekci. Expanze tak nikdy nemusí změnit velikosti více než dvoum sekcím zároveň (expandované sekci a sekci, ze které je převzána výška).

Při úpravě velikosti tažením za ”handle” v hlavičce vybrané sekce je chování následující. Handle je aktivní pouze ve chvíli, kdy existuje nad ním i pod ním aspoň jedna expandovaná sekce. Pokud je tato podmínka splněna, pak je nejblížší horní i spodní expandovaná sekce nalezena. Při tažení handle směrem nahoru je pak horní expandované sekci výška postupně odebírána a ekvivalentní výška je předána sekci spodní. Při tažení směrem nahoru je naopak výška odebrána spodní sekci a horní sekci je předána. Celkovým výsledkem je, že pokud se hned pod sebou nachází několik hlaviček kolapsovaných sekcí, tak se veškeré tyto hlavičky posouvají společně a velikost se mění nejbližším expandovaným sekcím okolo nich (viz obrázek 5.32).

Studio - Správce oken

Kořenovou komponentou celého window manager systému je komponenta `<Window>`, která je obsažena v souboru `page.tsx` stránky Studio a představuje

kořenový uzel stromové struktury, kterou je celý systém tvořen. Stav tohoto správce oken se nachází uvnitř Zustand store `useWinmanStore` a jeho persistence je rozdělena mezi několik záznamů uvnitř `localStorage`. Jeden z nich slouží jako index, který si ukládá informace o všech uživatelem definovaných layoutech, a každý další záznam je serializovanou reprezentací jednoho z těchto layoutů.

Každý uzel uvnitř této stromové struktury je poté z hlediska UI reprezentován komponentou `<GroupNode>` a koresponduje s jedním ze tří existujících typů: Horizontal Split, Vertical Split a Tabs.

Uzly typu Horizontal Split a Vertical Split reprezentují rozpůlení okna na dvě poloviny, buď po horizontální nebo vertikální ose. Tento uzel si ukládá informace o tom v jakém poměru velikostí obě poloviny jsou a následně odkazuje na dva další uzly, které reprezentují pravého a levého potomka tohoto rozpůlení. Poměr těchto polovin je uvnitř `<GroupNode>` implementován pomocí CSS atributů `flex-shrink` a `flex-grow` a poměry jsou tak automaticky zachovány při změně velikosti okna prohlížeče.

Uzel typu Tabs pak slouží jako koncový uzel stromu a obsahuje pole objektů `Tab`, kde každý reprezentuje jednu kartu na horní liště daného okna. Kromě toho si také ukládá informaci o tom, která z těchto karet je právě aktivována. Každý objekt `Tab` pak obsahuje ID dané karty (automaticky generované storem `useWinmanStore`) a ID zdroje, který má tato karta zobrazovat. Pokud je ID zdroje nastaveno na hodnotu `null`, pak se jedná o prázdnou kartu a uživatelské rozhraní zobrazí na horní liště název "<Empty>". Speciálními případy ID zdroje jsou také hodnoty "LIVEVIEW" a "FLOORPLAN", které reprezntují náhled živého přenosu a kompaktní verzi pohledu Floorplan.

Stromová reprezentace ukázkového layoutu (viz obrázek 5.33) lze vidět na obrázku 5.34.

Floorplan Editor

Obrazovka Floorplan Editor je reprezentována stejnojmennou komponentou `<FloorplanEditor>`, které je jako "prop" předávána booleanová hodnota `compact`. Na základě té je pak tato komponenta vykreslena buď ve formě kompaktního read-only náhledu uvnitř některého z oken v uživatelovo layoutu nebo jako plný celoobrazovkový editor s lištou nástrojů a postranním panelem vrstev. Stav celého editoru se poté nachází uvnitř Zustand store `useFloorplanStore`.

Nástroje

Lišta nástrojů se nachází u levé hrany obrazovky a obsahuje celkem 4 různé nástroje: Kurzor, Přidat kameru, Přidat obrázek a Vycentrovat. Ve spodní

části nástrojové lišty se také nachází šipka pro opuštění obrazovky Floorplan Editor a návrat do Studia.

Nástroj Kurzor je výchozím nástrojem tohoto editoru a umožňuje výběr a posun vrstev kliknutím do sekce Náhled. Po kliknutí se vybraná vrstva červeně zvýrazní a v postranním panelu se zobrazí její konfigurační možnosti. Nástroj Vycentrovat pak pouze posune pozici náhledu zpět na souřadnice [0,0].

Při výběru nástroje Přidat kameru se uživatelův kurzor myši změní na typ crosshair a při kliknutí do sekce Náhled se otevře dialog pro výběr souboru s příponou `.fspy`. Pomocí něj může uživatel načíst projektový soubor externího programu fSpy, který je v odvětví 3D modelování často používán pro nalezení kamery na základě statického obrázku [60]. Do tohoto programu tak uživatel může naimportovat snímek z kamery vybraného zdroje a pomocí něj automaticky získat informace o pozici této kamery a úhlu jejího zorného pole. Při importu tohoto souboru tak moje aplikace na základě načtených dat sestaví odpovídající pohledovou a projekční matici, která přesně odpovídá kameře zdroje. Společně s tím je z tohoto souboru načten i náhledový obrázek, který uživatel použil pro detekci kamery. S těmito údaji je pak vytvořena vrstva uvnitř `useFloorplanStore` a její pozice je nastavena na místo, kde uživatel kliknul nástrojem Přidat kameru. Po přidání vrstvy se vybraný nástroj automaticky vrátí na typ Kurzor.

Nástroj Přidat obrázek funguje obdobně jako nástroj Přidat kameru a při kliknutí do sekce Náhled také zobrazí dialog pro výběr souboru. Tentokrát však dialog vyžaduje soubor typu `image/*`. Z hlediska implementace je pak vytvořená obrázková vrstva reprezentovaná identicky jako vrstva kamery, pouze s tím rozdílem, že je pohledovo-projekční matice nastavena na matici identity. Při vykreslení tak nedochází k žádné deformaci. Samotná obrázková data jsou pak uložena ve stejné formě jako náhledový obrázek kamerové vrstvy. Tato data však ani v jendom případě nejsou uložena přímo do store `useFloorplanStore`, jelikož ten je persistován do uložiště `localStorage`, jež podporuje pouze ukládání textových řetězců. Místo toho jsou tak data obrázků uložena samostatně v úložišti IndexedDB, které nabízí i podporu pro ukládání binárních dat.

Náhled

Náhledová sekce obrazovky Floorplan Editor je tvořena dvěma elementy `<canvas>`, které se vzájemně překrývají. Spodní vrstva pak využívá renderovací kontext `CanvasRenderingContext2D` a slouží pouze k vykreslení naváděcí mřížky. Horní vrstva oproti tomu využívá kontext `WebGL2RenderingContext` a stará se o vykreslení samotných vrstev kamer a obrázků.

Každá vrstva je vykreslena ve formě čtyřúhelníku, který vyplňuje celý

náhled. Samotný posun, rotace, škálování a deformace vrstvy jsou tak implementovány čistě na straně fragment shaderu. Během každého renderovacího průchodu je tak nejprve předvypočítaná pohledovo-projekční matice každé vrstvy upravena tak, aby brala v potaz aktuální posun a přiblížení náhledu. Tato matice je do shaderů následně předána jako uniform.

Uvnitř vertex shaderu je pak nejprve `vec2` souřadnice vertexu rozšířena na `vec4` a to tak, že je do z-komponenty uložena hodnota 0.0 a do w-komponenty hodnota 1.0. Výsledný vektor je následně vynásoben maticí předanou v uniformu a výsledek je předán jako výstupní parametr do fragment shaderu. Do proměnné `gl_Position` je oproti tomu souřadnice vertexu nastavena přímo bez jakékoliv transformace.

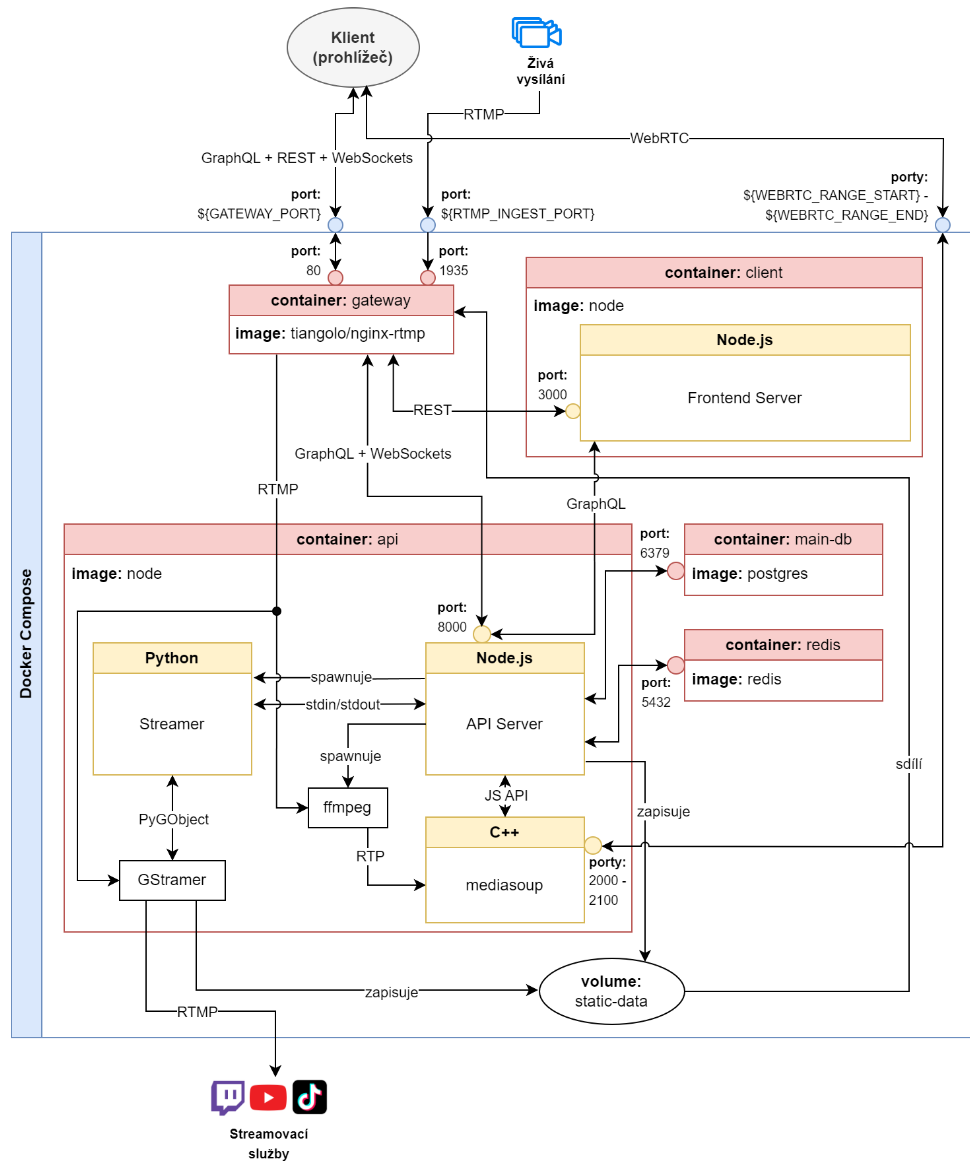
Na straně fragment shaderu je pak tento předaný vektor nejprve vydělen svou w-komponentou a následně přemapován z rozsahu -1.0 až 1.0 na 0.0 až 1.0. Komponenty x a y tohoto vektoru se pak v tuto chvíli dají použít pro samplování textury obsahující obrazová data zdroje a výsledný obraz vykreslený na čtyřúhelníku bude zdeformovaný takovým způsobem, aby z daného kamerového záběru vytvořil pohled z ptačí perspektivy.

Vrstvy

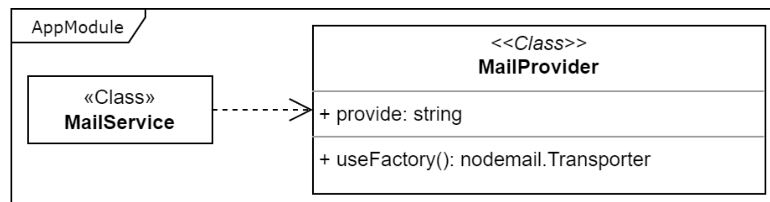
Postranní panel obrazovky Floorplan Editor využívá pro svou organizaci také komponetu `<PaneView>` (viz podsekce 5.5.5), aby obsah rozdělil do dvou sekcí: Layers a Options.

Sekce Layers obsahuje seřazený seznam všech vytvořených vrstev a jejich pořadí v tomto seznamu přesně odpovídá pořadí, ve kterém jsou v sekci Náhled vykresleny (vrstva na pozici 1 bude nejvíce v popředí). Tato sekce pak využívá knihovnu dnd kit, aby umožnila drag and drop funkcionalitu pro přesouvání jednotlivých vrstev, pomocí kterých může uživatel upravit jejich pořadí. Kliknutím na vrstvu lze danou vrstvu vybrat, stejně jako kdyby uživatel na tuto vrstvu kliknul v sekci Náhled nástrojem Kurzor. Každá vrstva také na pravé straně obsahuje tlačítko s ikonou oka, pomocí kterého lze danou vrstvu dočasně skrýt a následně znovu odkrýt.

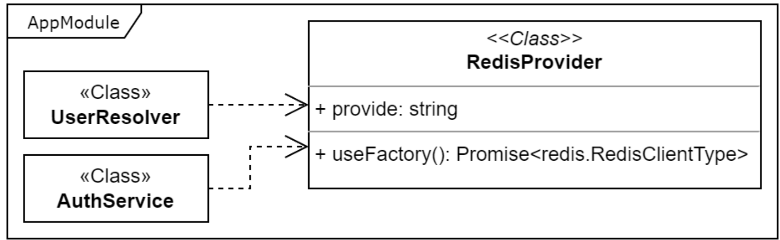
Sekce Options pak zobrazuje konfigurační možnosti pro právě vybranou vrstvu. Pokud tedy žádná z vrstev právě vybraná není, pak tato sekce zůstává prázdná. Sdílenými možnostmi pro oba typy vrstev jsou pak slidery pro rotaci a škálování dané vrstvy a tlačítko Remove pro její odstranění. V případě obrázkové vrstvy pak sekce Options obsahuje také textové pole, pomocí kterého lze vybranou vrstvu přejmenovat. U kamerové vrstvy je toto textové pole nahrazeno dropdown seznamem zdrojů, ve kterém uživatel vybere zdroj, který má být na této vrstvě zobrazen. Jakmile tedy tento zdroje přejde do stavu ONLINE, tak je náhledový obrázek této vrstvy nahrazen obrazovými daty příslušného živého přenosu.



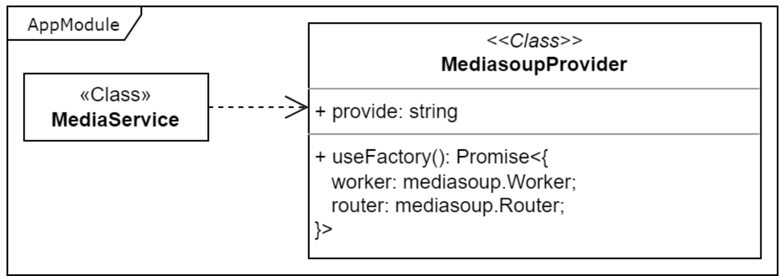
Obrázek 5.1: Graf Docker Compose konfigurace



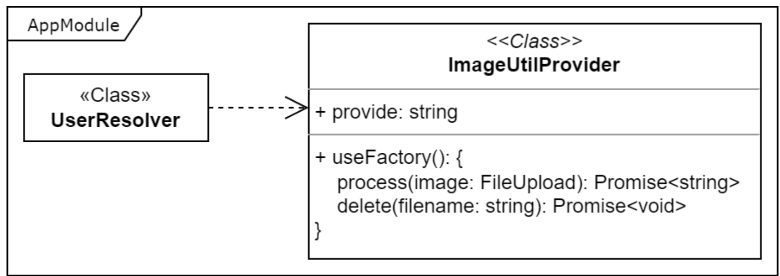
Obrázek 5.2: Class diagram MailTransportProvider



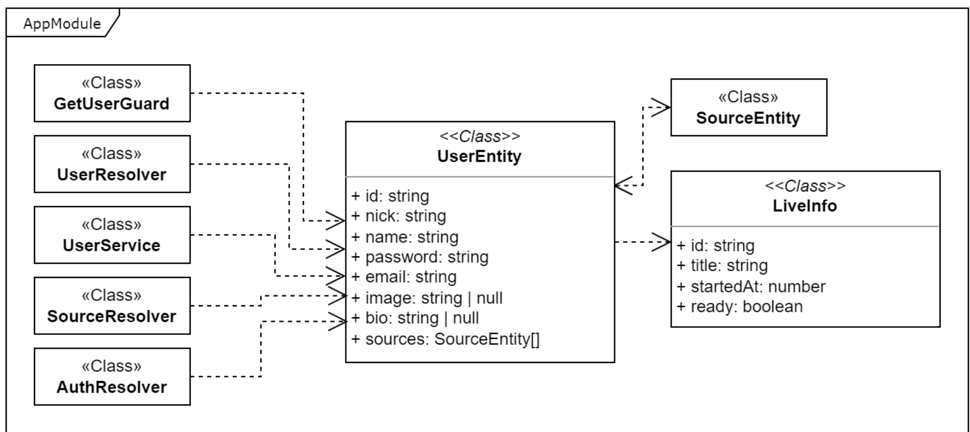
Obrázek 5.3: Class diagram RedisProvider



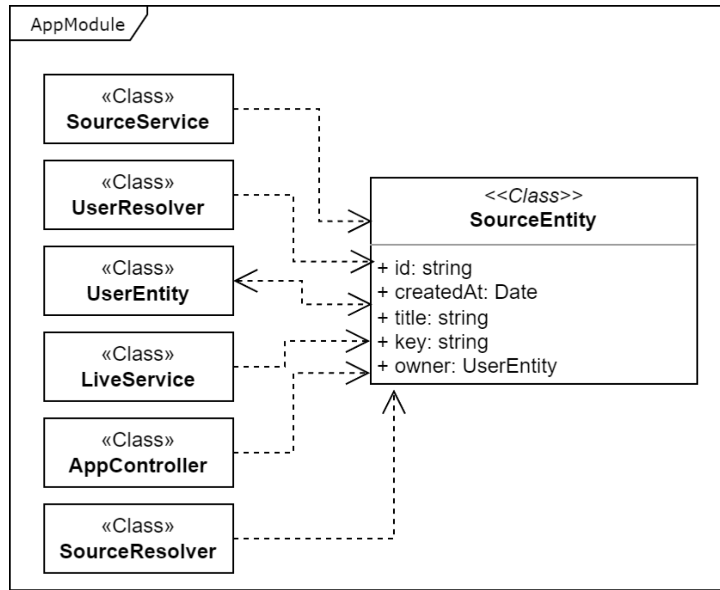
Obrázek 5.4: Class diagram MediasoupProvider



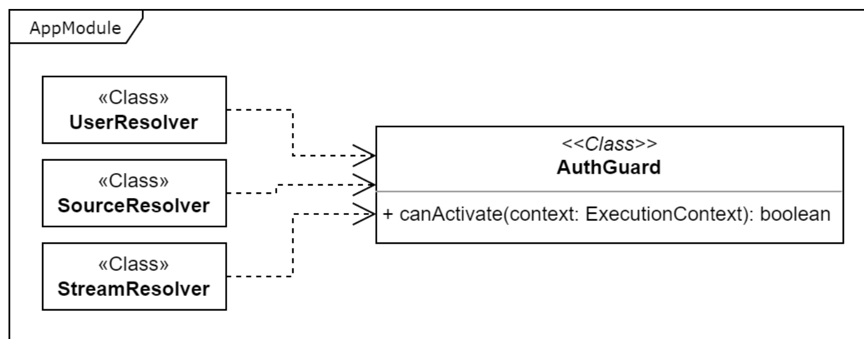
Obrázek 5.5: Class diagram třídy ImageUtilProvider



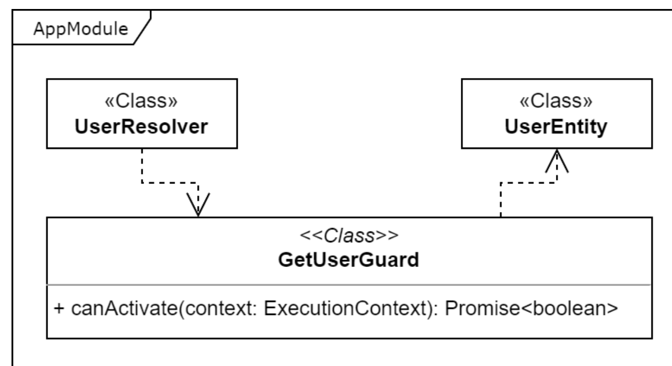
Obrázek 5.6: Class diagram třídy UserEntity



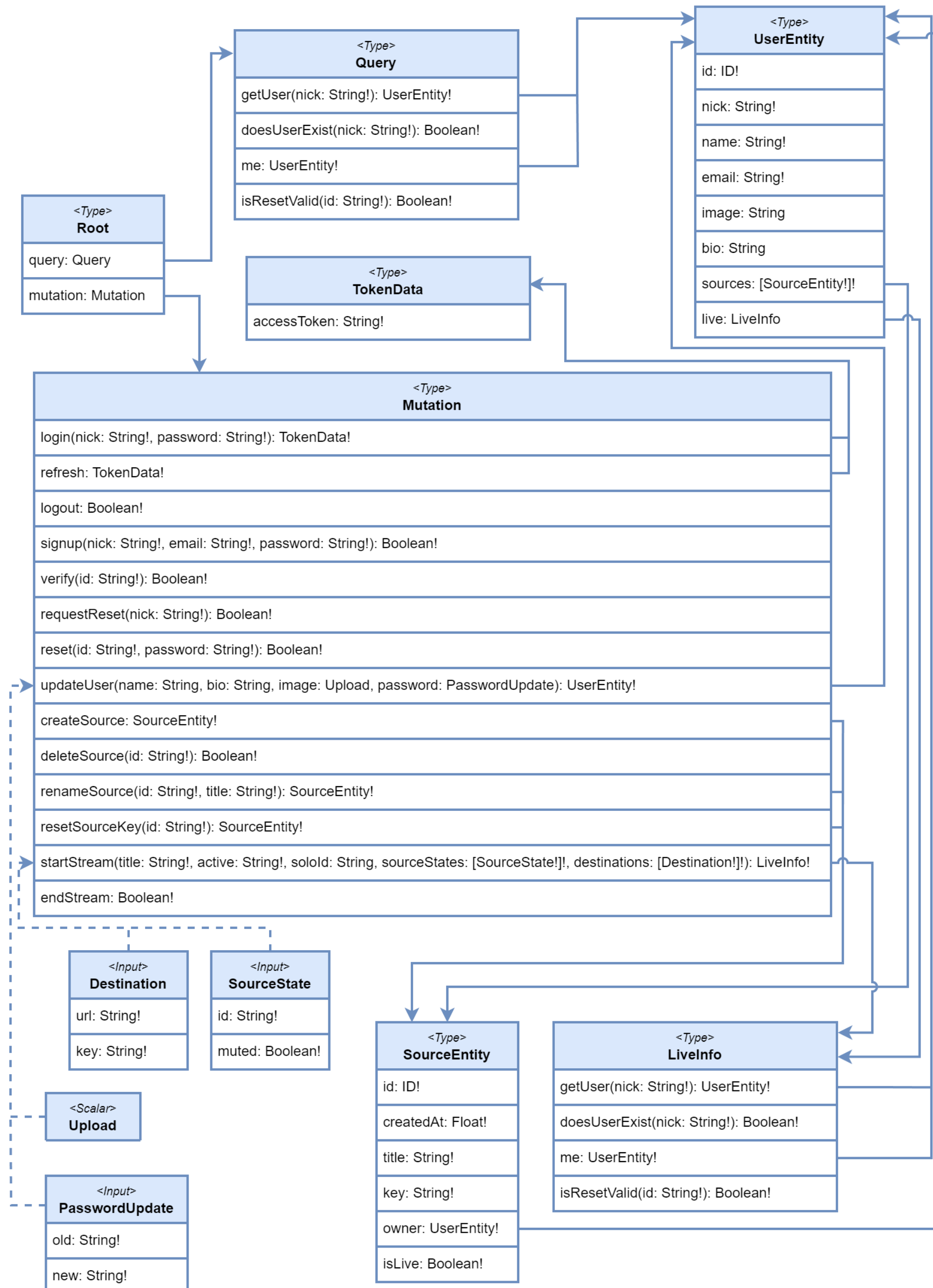
Obrázek 5.7: Class diagram třídy SourceEntity



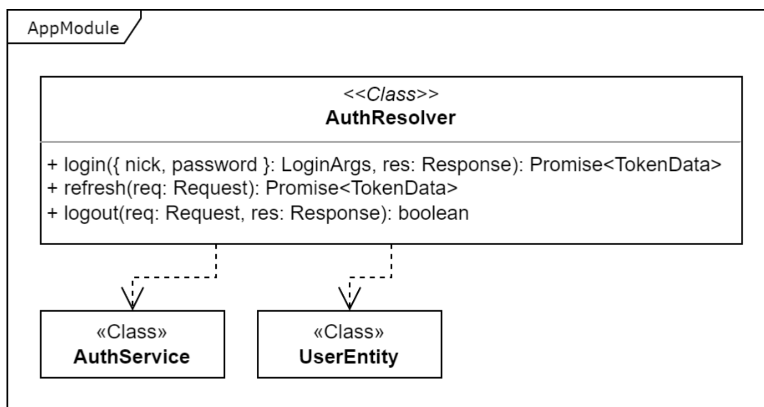
Obrázek 5.8: Class diagram třídy AuthGuard



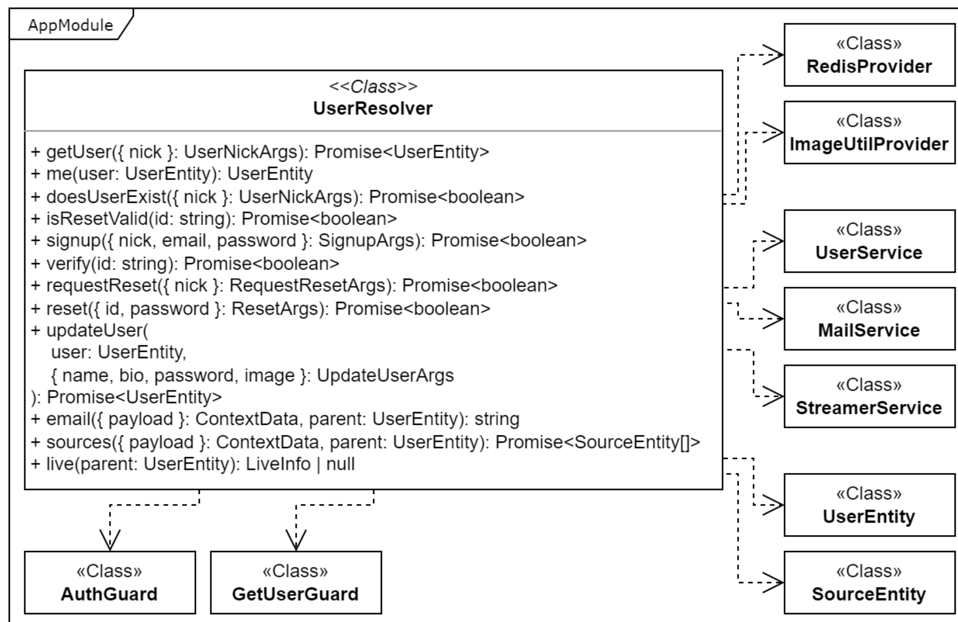
Obrázek 5.9: Class diagram třídy GetUserGuard



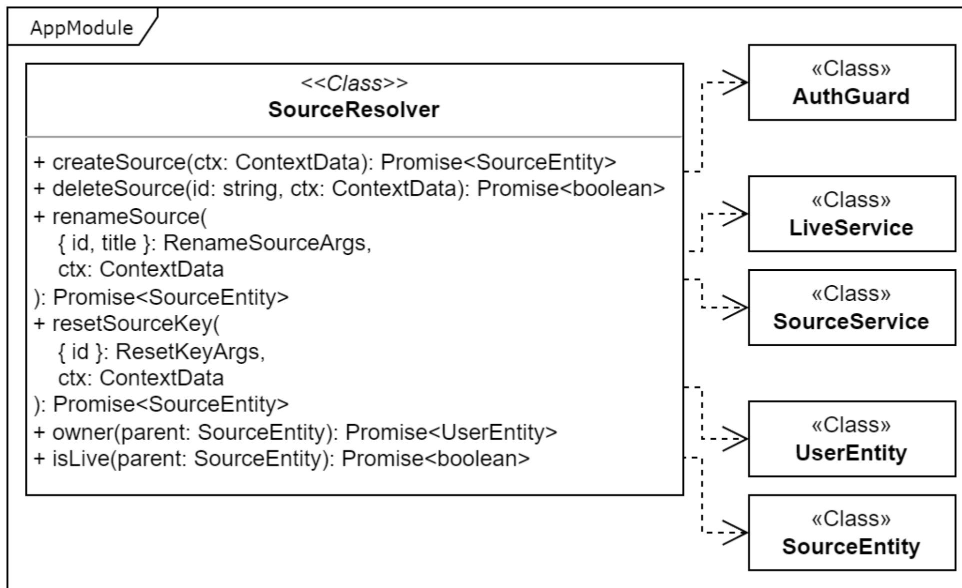
Obrázek 5.10: Schéma rozhraní GraphQL



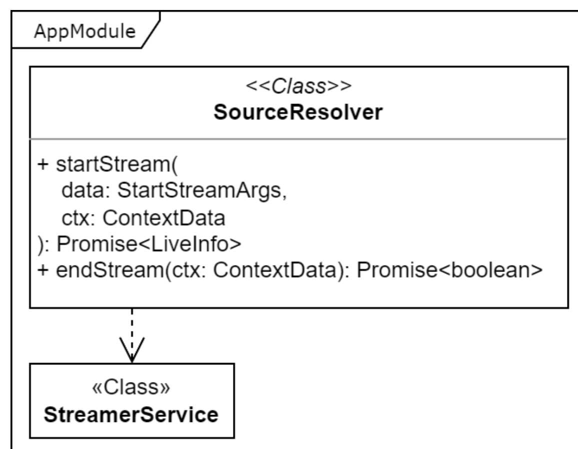
Obrázek 5.11: Class diagram třídy AuthResolver



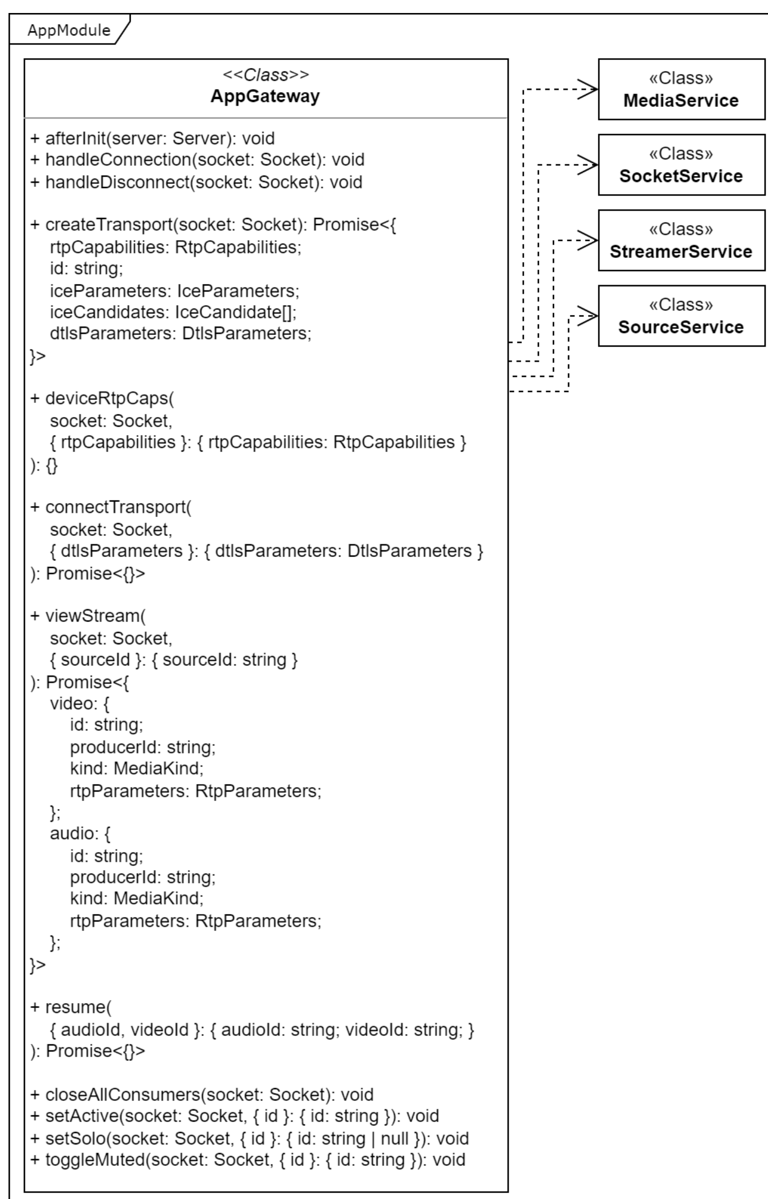
Obrázek 5.12: Class diagram třídy UserResolver



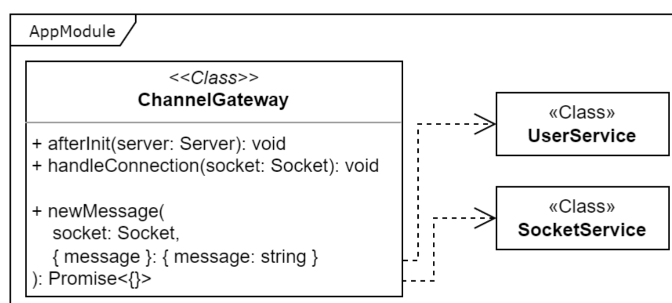
Obrázek 5.13: Class diagram třídy SourceResolver



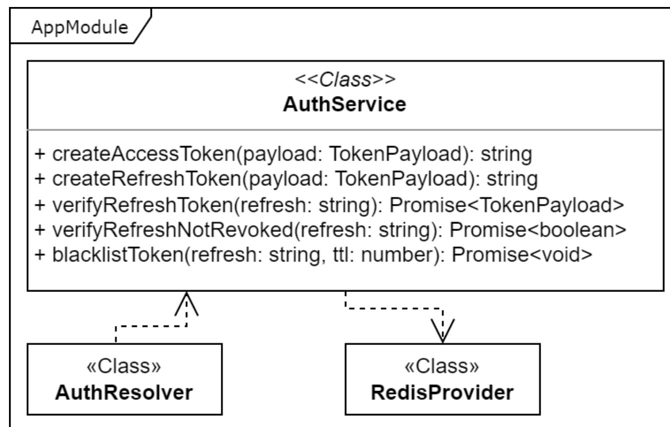
Obrázek 5.14: Class diagram třídy StreamResolver



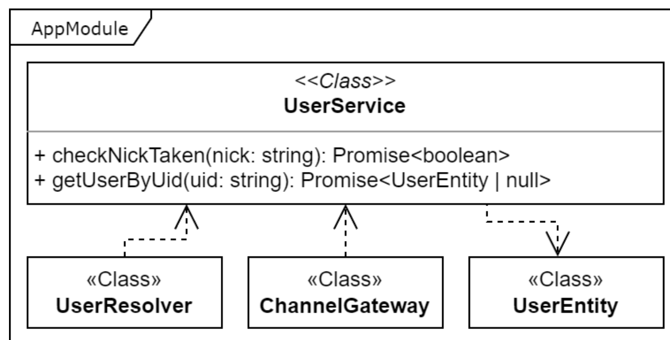
Obrázek 5.15: Class diagram třídy AppGateway



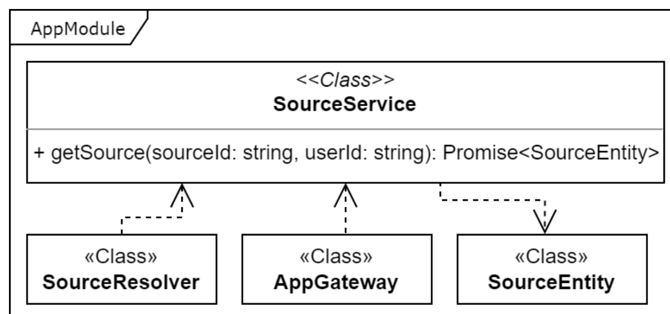
Obrázek 5.16: Class diagram třídy ChannelGateway



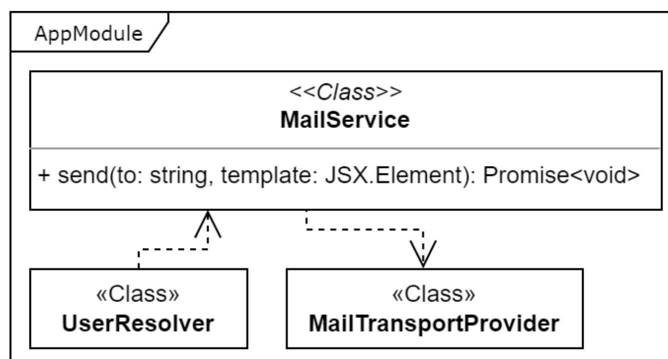
Obrázek 5.17: Class diagram třídy AuthService



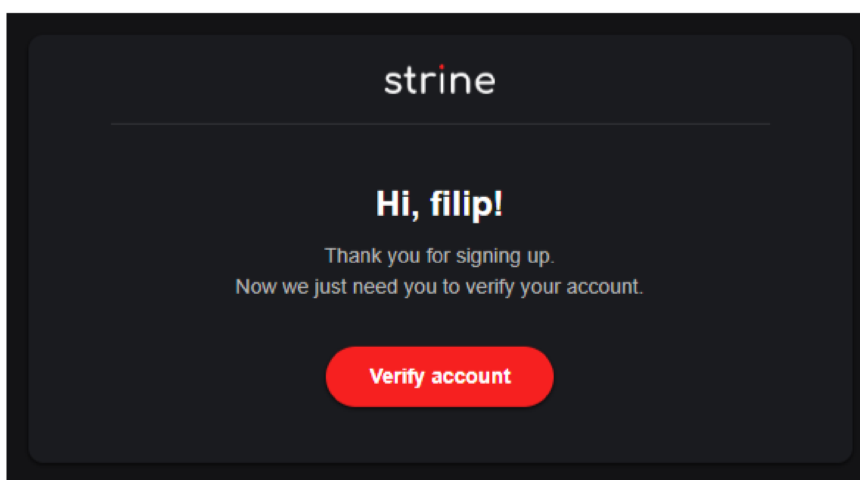
Obrázek 5.18: Class diagram třídy UserService



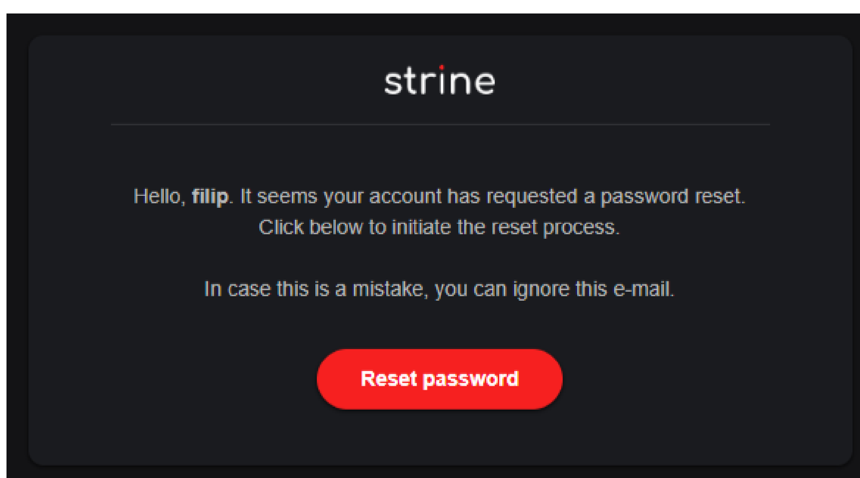
Obrázek 5.19: Class diagram třídy SourceService



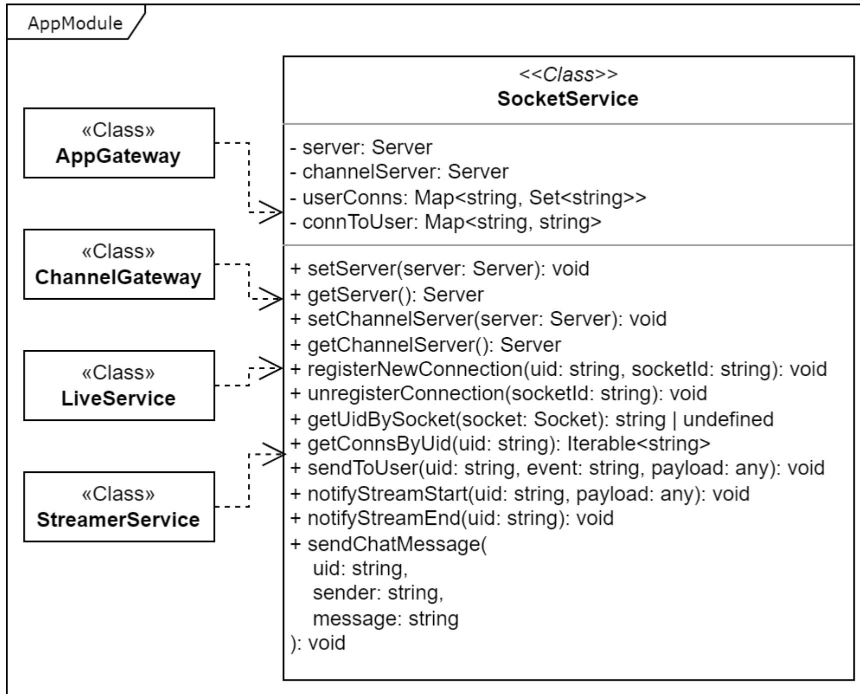
Obrázek 5.20: Class diagram třídy MailService



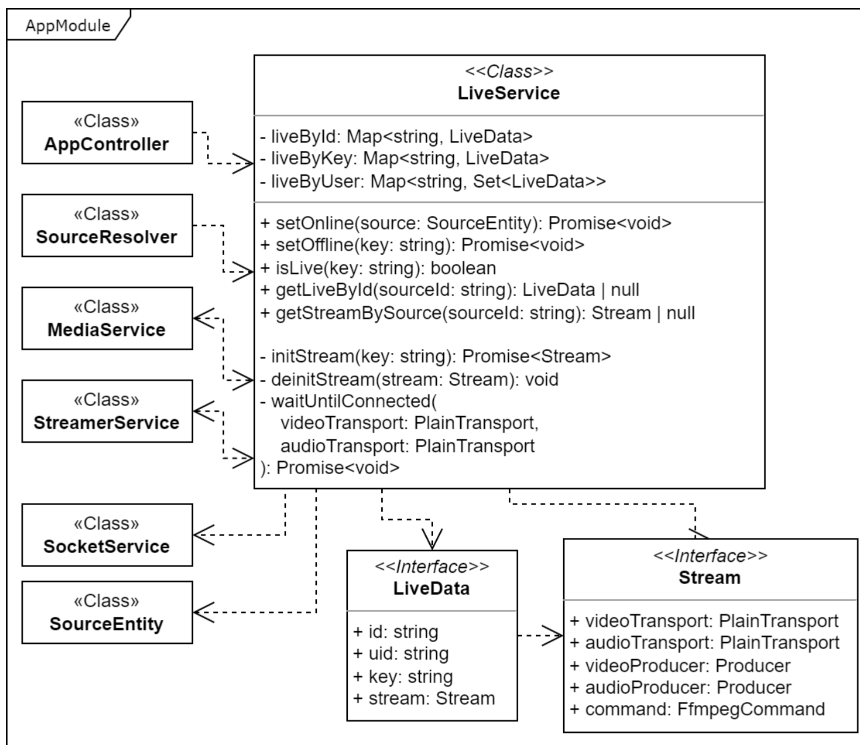
Obrázek 5.21: E-mailová šablona pro ověření účtu



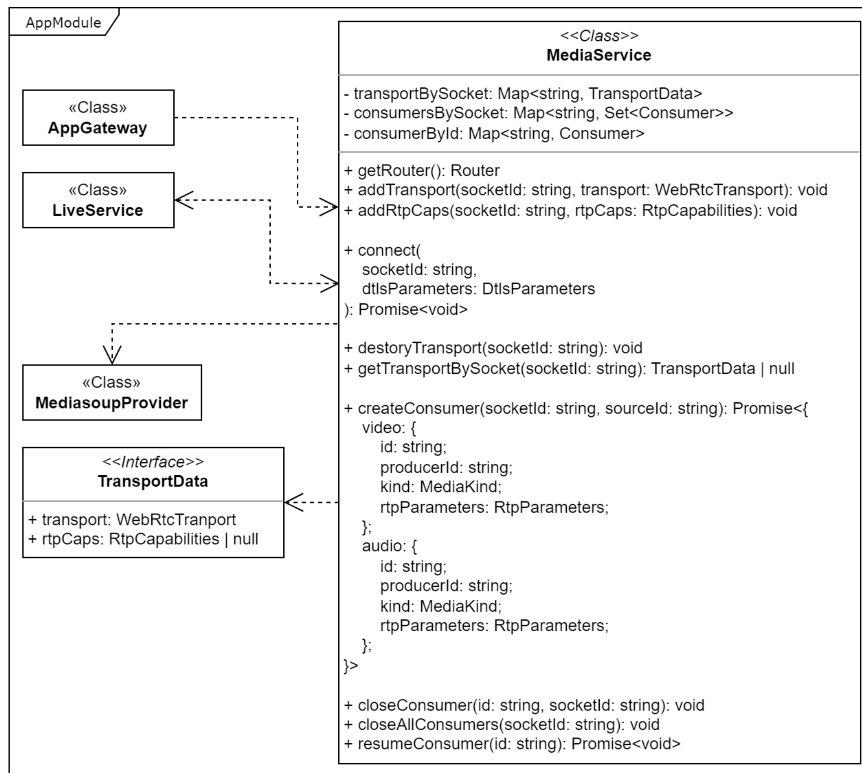
Obrázek 5.22: E-mailová šablona pro resetování hesla



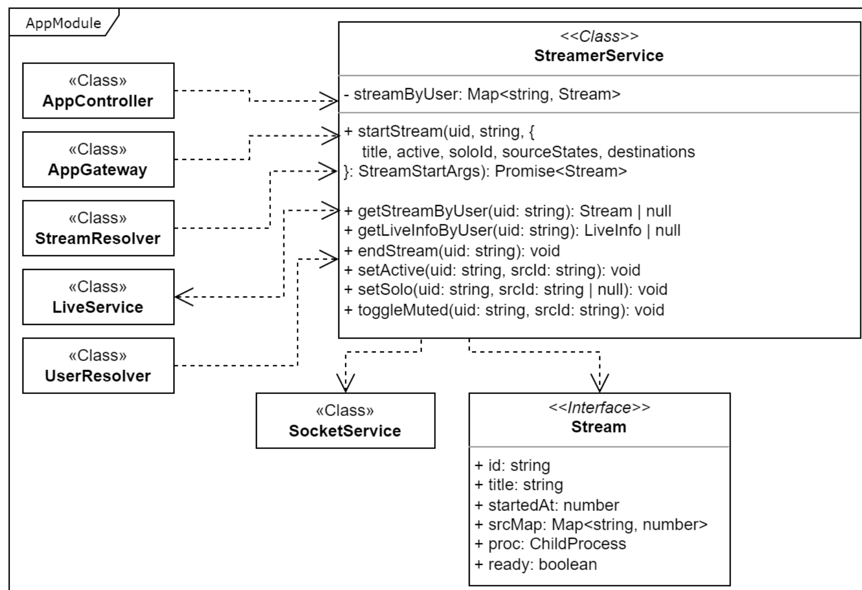
Obrázek 5.23: Class diagram třídy SocketService



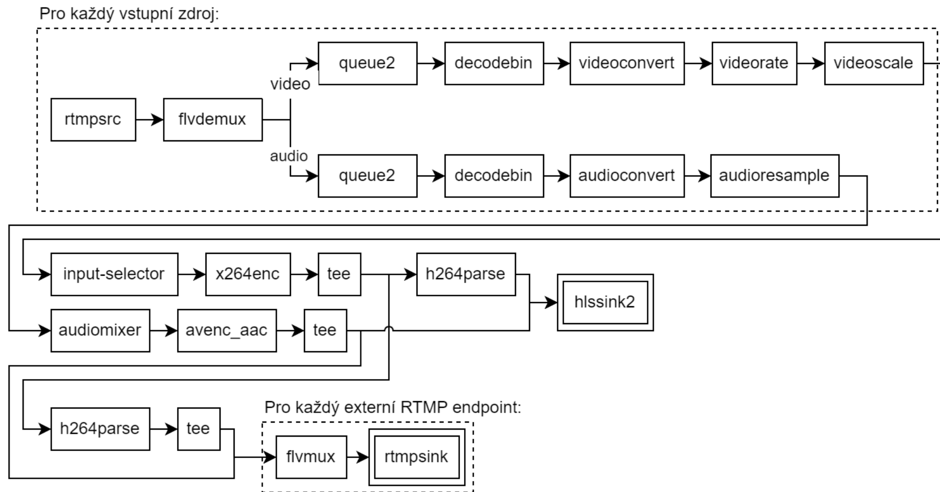
Obrázek 5.24: Class diagram třídy LiveService



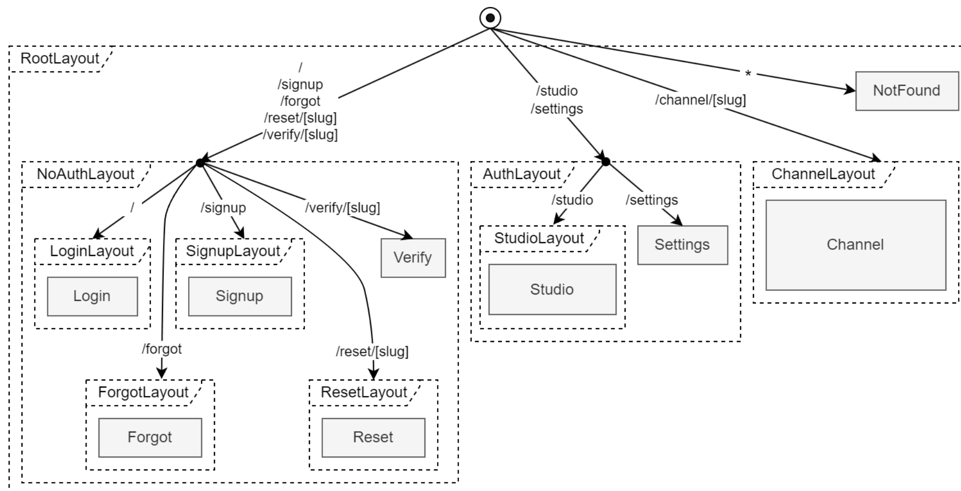
Obrázek 5.25: Class diagram třídy MediaService



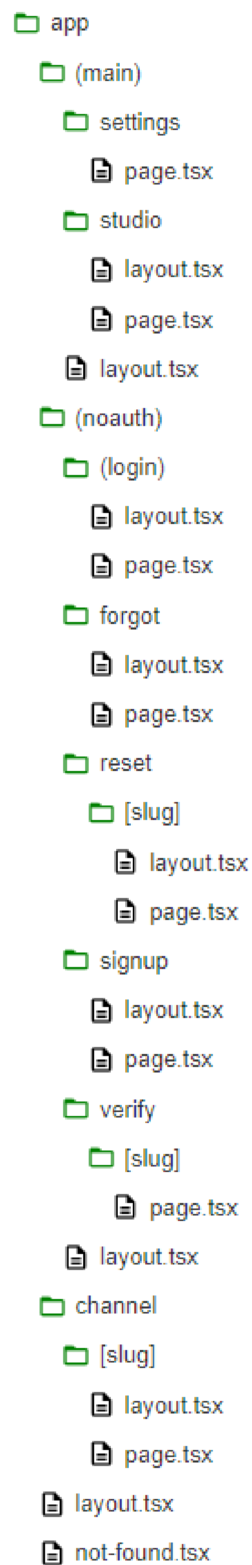
Obrázek 5.26: Class diagram třídy StreamerService



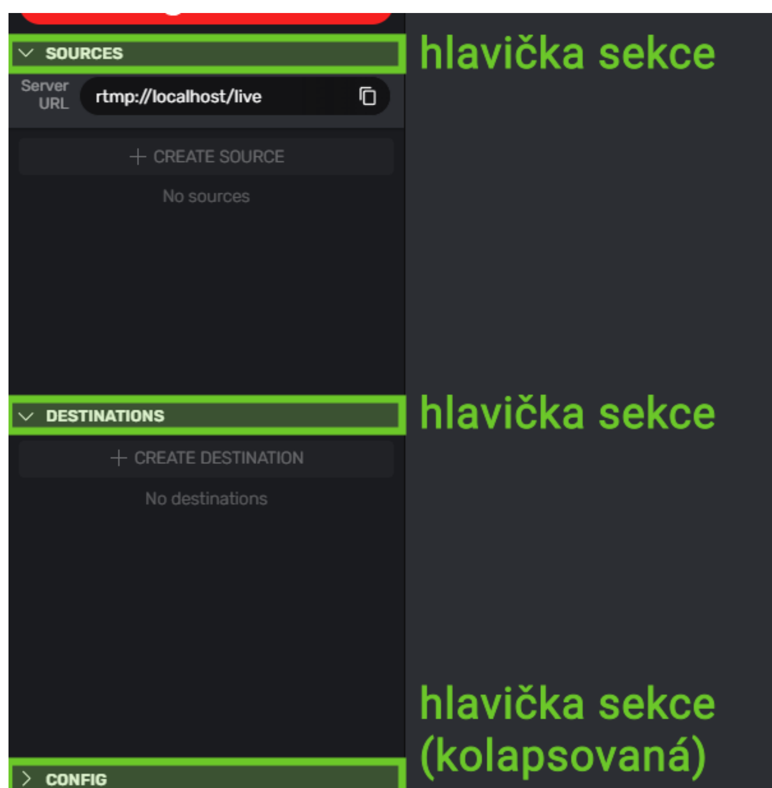
Obrázek 5.27: Graf GStreamer pipeline



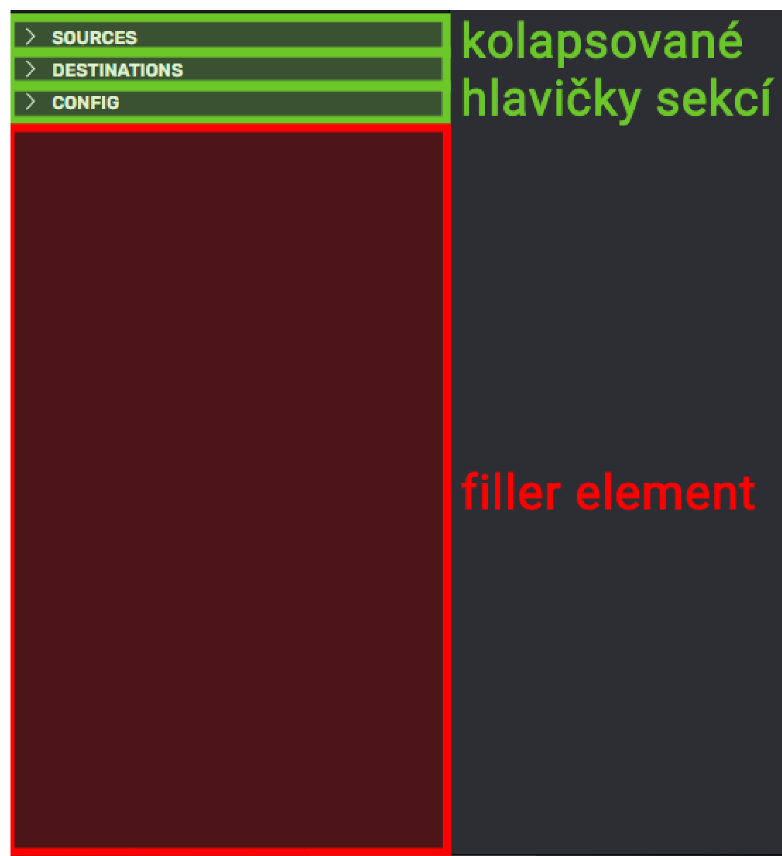
Obrázek 5.28: Komponentová struktura aplikace



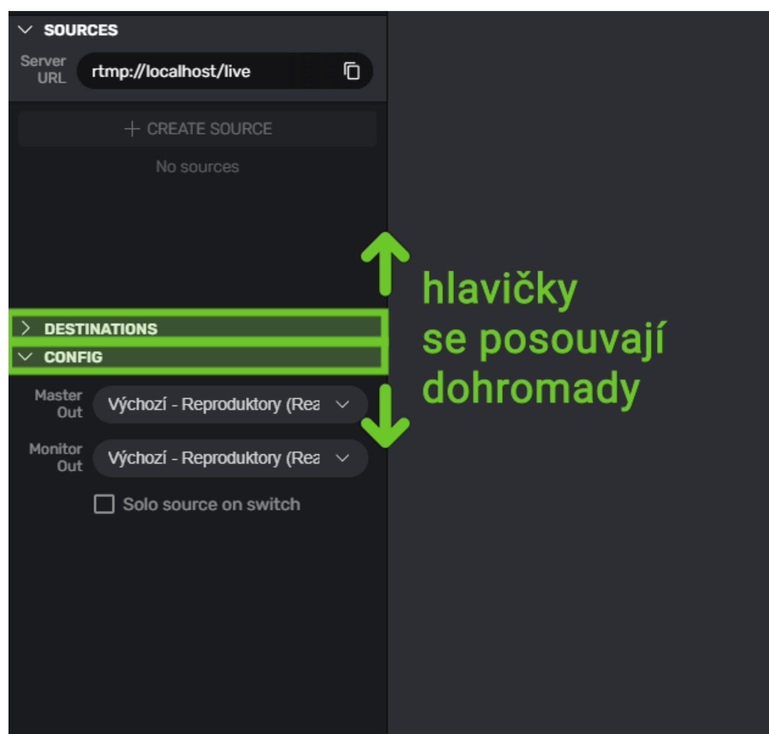
Obrázek 5.29: Souborová hierarchie složky `app`



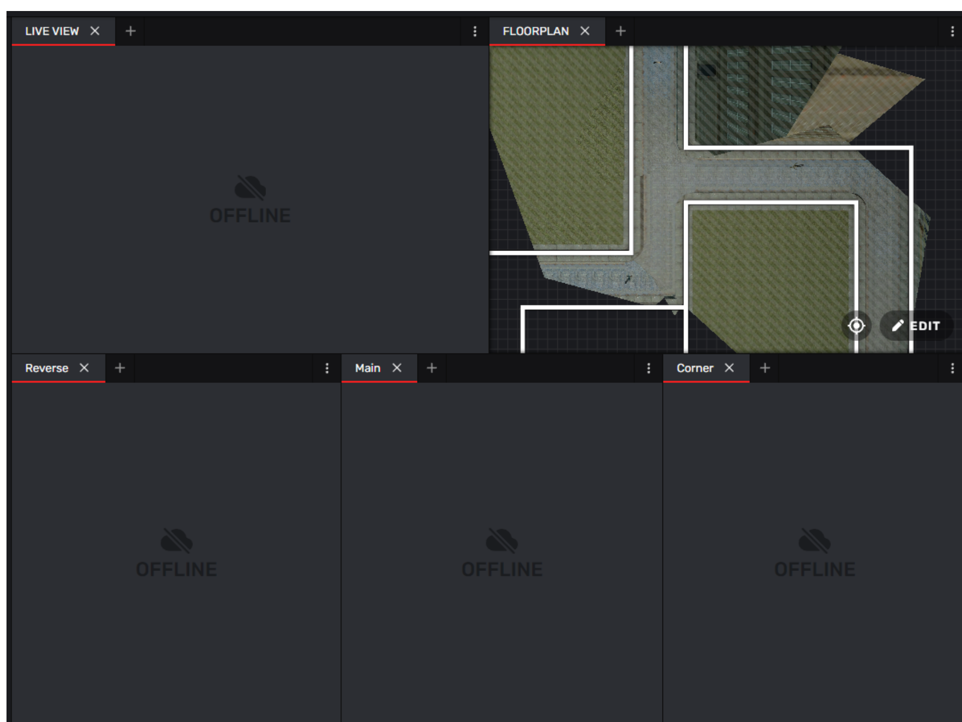
Obrázek 5.30: Hlavičky komponenty `<PaneView>`



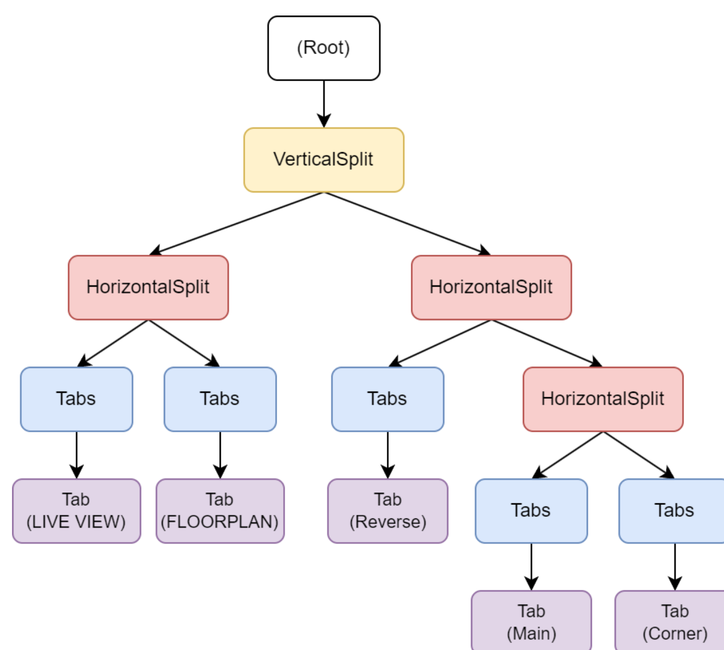
Obrázek 5.31: Filler element komponenty `<PaneView>`



Obrázek 5.32: Seskupené hlavičky komponenty `<PaneView>`



Obrázek 5.33: Výsledné uživatelské rozhraní ukázkového layoutu



Obrázek 5.34: Stromová reprezentace ukázkového layoutu

Kapitola 6

Testování

Testovací průchod aplikace se skládal celkem z jedenácti různých scénářů, které byli navrhnuty tak, aby co nejpřesněji emulovali skutečný prvotní zážitek nového uživatele a aby se dotkly co největšího množství existující funkcionality.

Do testování se dohromady zapojilo pět účastníků s různou úrovní znalostí probíhrané problematiky. Dva z uživatelů se s tématem livestreamování nikdy dříve nesetkali, dva uživatelé měli základní zkušenosti s osobním streamováním na platformě Twitch a poslední z uživatelů měl již profesionální zkušenost v oblasti produkce živých přenosů.

Před začátkem testů byla uživatelům představena pouze základní motivace za tuto aplikací a při vysvětlování funkcionality jsem se záměrně vyhnul technickým výrazům, které aplikace používá ve svém uživatelském rozhraní. Uživatelé tak testování započali pouze s velmi rudimentární představou o celkovém systému a při svém průchodu museli tedy spoléhat na intuitivní pochopení testovaných konceptů.

Samotné vysílání bylo uživateli vyzkoušeno ve třech různých variantách:

- streamování různorodého obsahu do vestavěného přehrávače (screenshare, webkamera, videosoubor)
- streamování kamer ve Floorplan pohledu na externí služby YouTube a Twitch
- streamování kombinace kamer ve Floorplan pohledu a ostatního obsahu s použitím monitorovacího audio výstupu

6.1 Scénáře

6.1.1 Vytvoření účtu

Uživatel si založí nový uživatelský účet a přihlásí se k němu.

Tento scénář nepředstavoval pro nikoho z uživatelů zásadní problém a všem se povedl dokončit bez jakékoliv externí pomoci. Dva z uživatelů však museli proces registace opakovat, jelikož poprvé zadali vymyšlenou e-mailovou adresu a teprve po dokončení byli notifikační zprávou informování o tom, že je e-mail vyžadován k ověření účtu. U jednoho z dalších uživatelů jsem si také všiml delší prodlevy než tuto notifikační zprávu u horního okraje obrazovky zpozoroval. V praxi by tak mohlo dojít k riziku, že notifikace zmizí (limit 3.5s), aniž by si toho uživatel všiml. Lepším řešením by tak mohla být samostatná stránka, na kterou je uživatel po dokončení registrace přesměrován.

Samotné ověření účtu a přihlášení pak již proběhlo bez jakýchkoliv zádrhelů a verifikační e-mailová zpráva se správě vykreslila ve všech uživateli používaných klientech (Seznam, Gmail, Yahoo).

6.1.2 Vytvoření zdroje

Uživatel vytvoří zdroj pojmenovaný "Obrazovka 1", začně na něj vysílat screenshare svojí obrazovky (předpřipravená OBS scéna) a vysílaný obraz zobrazí ve Studiu.

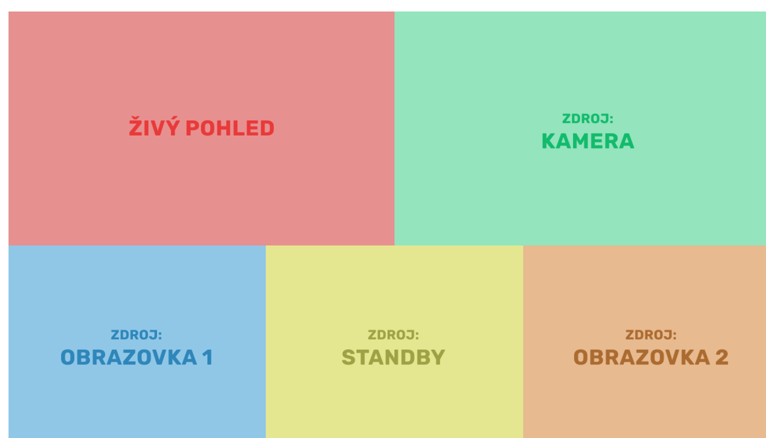
Všem uživatelům se snadno podařilo najít sekci Sources v postranním panelu, založit nový zdroj i jej přejmenovat na "Obrazovka 1". Všichni tři uživatelé se zkušenostmi s livestreamingem pak také samostatně dokázali zkopírovat zobrazený streamovací klíč a správně ho nakonfigurovat uvnitř OBS. Neznalým uživatelům bylo nejprve potřeba vysvětlit základy fungování streamovacího software, následně se jim však konfigurace také podařila bez větších problémů. U samotného kopírování jsem však pozoroval nežádoucí dopad na UX, jelikož v tuto chvíli pole „Stream key“ funguje tak, že se při jeho nakliknutí celý klíč automaticky vybere a zkopíruje do schránky. Každý z uživatelů se však kopírování pokusil započnout tažením kurzoru po textovém poli, čímž se automaticky vybraný text naopak deselectoval. Do budoucna bych tak doporučoval výběr textu zcela zamezit a udělat z textového pole pouze klikatelný prvek.

Uživatelé následně zpozorovali, že se stav zdroje změnil na ONLINE, a po krátké chvíli hledání se jim podařilo obraz tohoto zdroje zobrazit ve výchozím

okně. Pouze jediný uživatel však přišel na to, že lze karty zdrojů přetáhnout do okenního layoutu pomocí drag and drop. Zbytek uživatelů zdroj oknu přiřadil pomocí dropdown menu Select Source. Navrhoval bych tak přidání CSS stylů při najetí kurzoru myši na kartu zdroje, čímž by aplikace lépe indikovala, že je možné s kartou samotnou interagovat.

6.1.3 Správa layoutu

Uživatel přidá další zdroje a vytvoří layout podle dodaného obrázku. Následně nastaví zdroj Standby jako aktivní, tak aby byl vidět v okně Živý pohled.

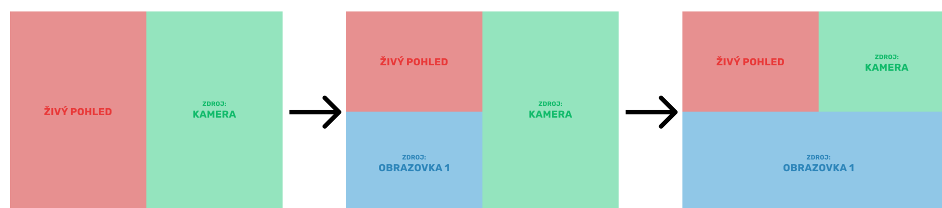


Obrázek 6.1: Požadovaný layout scénáře "Správa layout"

Zatímco drag and drop zdrojových karet u předchozího scénáře většina uživatelů neobjevila, tak u tohoto scénáře bylo ihned zjevné, že uživatelé jsou všeobecně více seznámeni s konceptem správy oken, např. díky používání systému Windows. Uživatelé tak nejprve vytvořili nové zdroje v postranním panelu, což bylo po provedení předchozího scénáře již zcela bezproblémové. Následně skoro ve všech případech sáhli po tlačítku + pro přidání nové karty okna a po chvíli experimentování se jim podařilo přijít na to, že lze novou kartu vytáhnout z horní lišty a dokovat jí k jedné ze stran, čímž je původní okno rozděleno na dvě. Pouze jediný uživatel tak celý layout vytvářel s použitím dropdown tlačítek Horizontal Split a Vertical Split, místo pomocí drag and drop. Celkově ale relativní rychlost, se kterou uživatelé dokázali požadovaný layout vytvořit, považuji za velmi dobrý výsledek, který dokazuje vysokou intuitivnost tohoto window management systému.

Jednou z opakovaně pozorovaných limitací byla však nutnost se vždy držet hierarchie rozdělených oken. To mělo za následek, že pokud se uživatel rozhodl kořenové okno nejprve rozdělit horizontálně do sekcí Živý pohled a Kamera, tak pak již nebylo možné celé okno rozdělit vertikálně, aby mohl uživatel přidat spodní trojici pohledů. Uživatel tak musel nejprve vertikálně rozpůlit

jednu z vytvořených horizontálních polovin a druhou nerozpůlenou polovinu předokovat do horní části tohoto nově rozpůleného okna.



Obrázek 6.2: Limitace správy oken

Tento problém jsem pozoroval opakovaně skoro u všech uživatelů, avšak nikdo z nich touto chybou nebyl výrazně zmatený a všichni uživatelé dokázali rozložení oken rychle přeorganizovat. Z mého pohledu si tedy myslím, že velmi snadná pochopitelnost této okenní hierarchie dalece předčívá její nevyhnutelné limitace a pro budoucí vývoj bych se jí rozhodl zachovat.

Uživatelům také chvíli trvalo naleznout položku pro nastavení okna na živý pohled, jelikož jí při vybírání z listu Select Source většina z nich nezaznamenala. Po chvíli hledání se však tuto možnost podařilo najít všem uživatelům a následné nastavení aktivního zdroje pomocí tlačítka SWITCH už se obešlo bez zdržení.

6.1.4 Mixování zvuku

Uživatel nastaví Studio tak, aby byla slyšet pouze audio stopa zdroje Obrazovka 2.

Tento scénář se bez jakýcholiv komplikací podařil dokončit čtyřem z pěti účastníků a to těm s předchozími zkušenostmi s livestreamováním a jednomu nezkušenému, který se však v minulosti již zběžně setkal se střihem videa. Poslední účastník byl s koncepty MUTE a SOLO zcela neseznámen; i jemu se však po chvíli podařilo příslušná tlačítka pochopit metodou pokus-omyl. Tři z účastníků pak požadovaného výsledku dosáhli tlačítkem SOLO, zbylí dva naopak aktivovali možnost MUTE na všech ostatních zdrojích.

6.1.5 Spuštění přenosu

Uživatel nastaví Studio tak, aby byl vždy automaticky slyšitelný právě aktivní zdroj, a následně spustí živý přenos do vestavěného přehrávače. Poté postupně přepne mezi všemi čtyřmi zdroji a přenos ukončí.

Pro tento scénář jsem uživatele nejprve instruoval, aby na nové kartě prohlížeče nalezli svoji profilovou stránku a tam tak mohli sledovat vysílaný přenos. Najít odpovídající sekci Channel v dropdown menu na horní liště se pak všem povedlo bez jakéhokoliv problému.

Oproti tomu, nastavení možnosti pro automatické "solo" aktivní stopy byl pro uživatele konzistentně jeden z největších problémů celého testu. Hlavní překážkou pak bylo zejména nalezení sekce Config v postranním panelu, kde se tato možnost nachází. Již během dřívějších scénářů jsem totiž pozoroval, že většina uživatelů nenašla možnost pro kolapsování sekcí v postranním panelu a s výjimkou jediného uživatele nechali všichni postranní panel ve výchozí konfiguraci, kdy je rovnoměrně rozdělený sekcemi Sources a Destinations. Vzhlem k tomu, že sekce Config je pak ve výchozím stavu kolapsovaná, tak zcela unikla pozornosti všech uživatelů. Zde bylo tak potřeba uživatele lehce nasměrovat do postranního panelu, kde pak již dokázali sekci Config nalézt a bez větších problémů si odvodili fungování checkboxu "Solo source on switch".

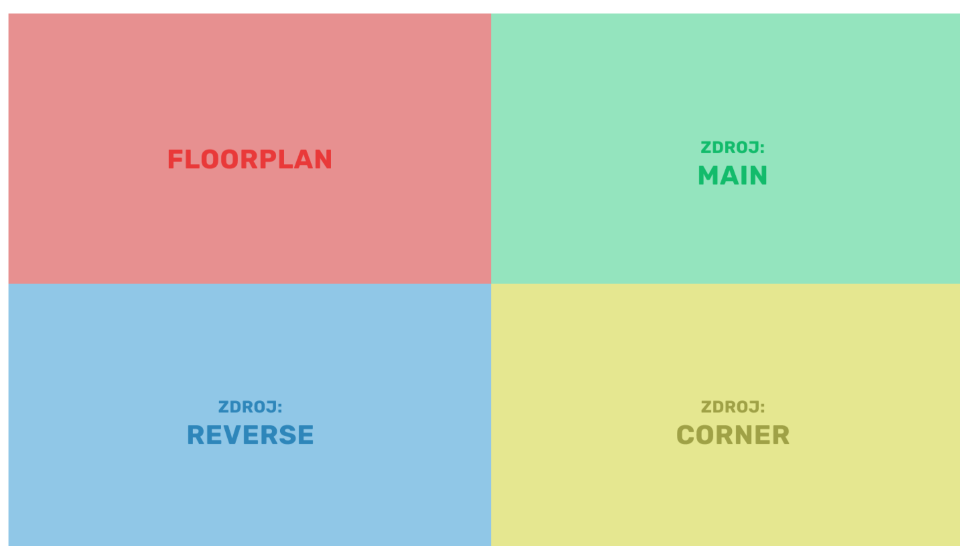
Samotné spuštění přenosu se pak uživatelům podařilo již bez problémů a společně s tím i následné přepínání zdrojů a ukončení. Delší prodleva mezi spuštěním přenosu a jeho zobrazením na stránce Channel, způsobená protokolem HLS, však některé z uživatelů poměrně znejistila a do budoucna bych tak doporučoval přidat vizuální indikátor, který se aktivuje ihned po spuštění přenosu.

6.1.6 Tvorba nového layoutu

Uživatel vytvoří nový layout (aniž by zničil původní) podle dodaného obrázku.

Najít lištu pro správu layoutů se všem uživatelům podařilo velmi rychle a snadno pak dokázali založit nový layout. Část však byla zmatená tím, že nový layout vypadá identicky jako jejich předchozí, jelikož při vytvoření nového layoutu doje k duplikaci toho právě aktivního. Možným řešením by mohlo být vytvoření prázného layoutu místo duplikace; v takovém případě by však bylo na místě přidat také tlačítko pro explicitní zduplikování vybraného layoutu.

Díky zkušenostem ze scénáře 6.1.3 pak již nalezení pohledu Floorplan nepředstavovalo problém a uživatelé ho ihned sami našli v menu Select Source. Kromě toho také uživatelé založili tři nové zdroje podle zadání na obrázku.



Obrázek 6.3: Požadovaný layout scénáře "Tvorba nového layout"

6.1.7 Tvorba Floorplan pohledu

Uživatel nakonfiguruje pohled Floorplan tak, aby zdroje viděl v jednotném pohledu z ptačí perspektivy. Zobrazené vrstvy budou v pořadí: 1. Corner, 2. Reserve, 3. Main

Pro tento scénář bylo potřeba větší zapojení z mé strany, abych uživatelům nejprve vysvětlil koncept tohoto pohledu a fungování nástroje fSpy. S mou pomocí tak uživatelé nejprve vytvořili `.fspy` projektové soubory pro všechny tři zdroje a následně už bylo jejich samostatným úkolem tyto soubory využít v pohledu Floorplan.

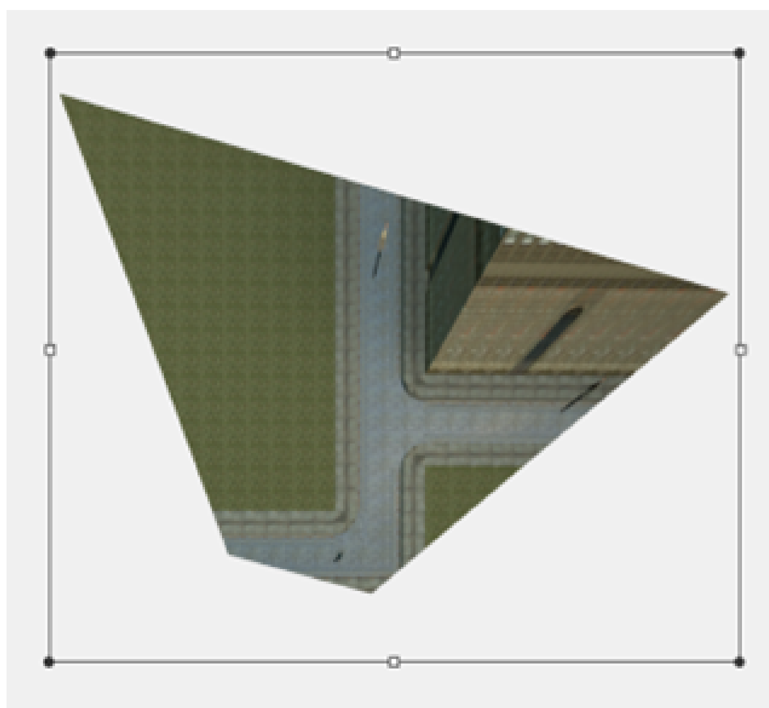
Všichni uživatelé pak ihned dokázali otevřít Floorplan Editor stisknutím tlačítka EDIT a přidat nový pohled tlačítkem Add Camera. Jeden z uživatelů jako první kliknul na tlačítko Add Image, ale jeho chyba mu došla, jakmile si všiml, že otevřený file picker nezobrazuje jeho vytvořené `.fspy` soubory. Tři uživatelé také dokázali sami přijít na to, že lze celý pohled editoru posouvat podržením kolečka myši; zbylím dvoum jsem základní ovládání sám vysvětlil.

Uživatelům pak již nedělalo problém koncept Floorplan pohledu pochopit, ale samotná tvorba byla stále poměrně obtížná. Uživatelé měli zejména problém vrstvy naškálovat na správnou velikost, aby spolu vytvořili ucelený pohled, jelikož slider velikosti nebyl dostatečně citlivý a malé pohyby způsobili příliš velké změny ve velikosti. Možnými vylepšeními do budoucna jsou tak například:

- "precision mode" - stisknutí klávesy (např. Shift) přepne slider do více

citlivého režimu

- přidat číselná pole pro manuální zadání velikosti
- ovládání velikosti pomocí kontrolních bodů zobrazených přímo okolo obrázku / pohledu kamery



Obrázek 6.4: Previzualizace kontrolních bodů pro škálování vrstev

Mimo to bych také doporučil přidání slideru průhlednost, díky kterému by uživatel mohl vidět, jak přesně na sebe okraje jednotlivých vrstev navazují. Bez této možnosti měli uživatelé problém najít ideální pozici pro každou vrstvu.

Oproti tomu, samotné seřazení vrstev podle zadání zde proběhlo velmi hladce. Na rozdíl od karet zdrojů (viz podsekce 6.1.2) totiž jednotlivé vrstvy v sekci Layers vizuálně indikují najetí myši a uživatelům tak bylo ihned zjevné, že je vrstvy možné přesouvat pomocí drag and drop.

6.1.8 Tvorba destinací

Uživatel nastaví vysílání na externí platformy YouTube a Twitch a spustí přenos. Uživatelům jsou předány údaje pro tato vysílání (server URL a streaming key).

Všichni uživatelé dokázali intuitivně odvodit význam a chování sekce Destinations, ve které podle zadání vytvořili dvě nové destinace a nakonfigurovali je s poskytnutými údaji. Jelikož však karty destinací nevyžadují žádné potvrzení při změně údajů, tak část uživatelů vyjádřila nejistotu nad tím, zda je nastavení destinací již hotové a zda mohou spustit livestream. V budoucí iteraci je tak rozhraní možné doplnit tlačítky pro explicitní editaci a uložení stavu destinace. Samotný přenos se pak již podařilo spusit bez problémů.

6.1.9 Klávesové zkratky

Uživatel nastaví zdrojům klávesové zkratky pro přepínání podle specifikace:

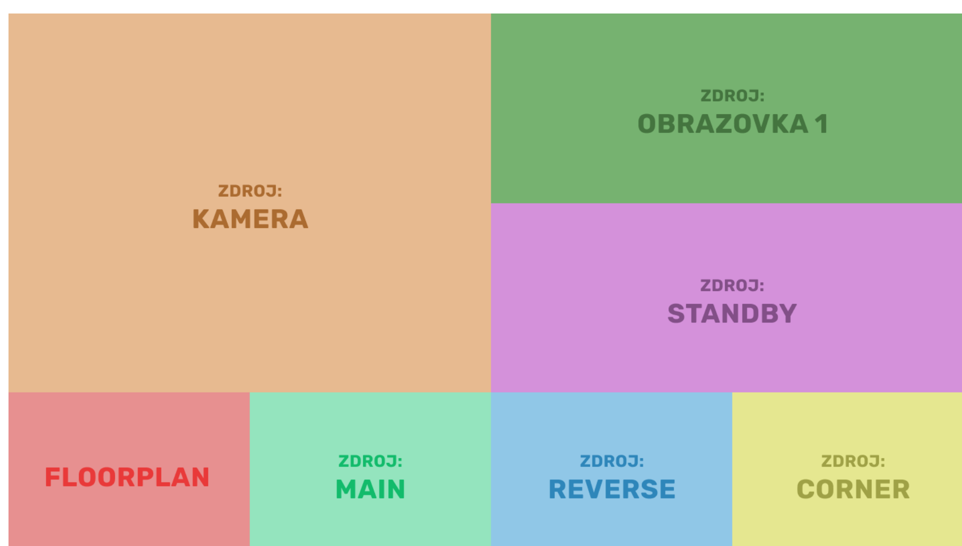
- **Ctrl + Alt + A - Main**
- **Ctrl + Alt + B - Reverse**
- **Ctrl + Alt + C - Corner**

Při hledání menu pro tvorbu klávesových zkratek skoro všichni uživatelé nejprve zamířili do sekce Sources, kde tuto možnost hledali u karty daného zdroje. Poté, co zde tuto možnost nenašli, pak menu dokázali po chvíli najít v dropdown menu na horní liště okna. Jako hlavní problém bych zde tak viděl roztříštěnou funkcionalitu těchto možností na horní liště, jelikož část z nich se týká přímo daného okna (Horizontal Split, Close All, atd.) a část se týká obsahu, které dané okno právě zobrazuje (MUTE, SOLO, Keybinds, atd.). Jednou z možností by bylo veškeré tyto možnosti přesunout do postranního panelu ke kartám zdrojů, avšak snadná a rychlá dostupnost z horní lišty okna se oproti tomu jeví jako výrazně ergonomičtější možnost. Ideální organizační struktura těchto možností tak bude v budoucích iteracích aplikace vyžadovat větší zamyšlení.

Samotné vytvoření zkratky po otevření menu Keybinds pak již proběhlo bez jakýchkoliv komplikací a všem uživatelům se podařilo zkratky nastavit a následně i vyzkoušet jejich fungování při přepínání aktivního zdroje.

6.1.10 Audio monitoring

Uživatel se vrátí do původně vytvořeného layoutu "default" a upraví ho podle dodaného obrázku. Následně nastaví zdroje tak, aby byly zdroje Kamera, Obrazovka 1 a Standby slyšet na audio výstupu



Obrázek 6.5: Požadovaný layout scénáře "Audio monitoring"

Master Out a zdroje Main, Reverse a Corner na výstupu Monitor Out.

Díky předchozí zkušenosti s tvorbou nového layoutu, a společně s tím i s použitím layout selectoru na horní liště obrazovky, již uživatelům nedělalo problém přepnout aktivní layout na výchozí "default". Požadované rozložení pak dokázali všichni vytvořit bez větších komplikací.

Uživatelé si díky scénáři 6.1.5 také již zapamatovali umístění selektorů audio výstupů v sekci Config postranního panelu. Pouze uživatel s profesní zkušeností v oblasti živé produkce však dokázal výstupy samostatně nastavit a následně i nakonfigurovat jednotlivé zdroje. Ostatním uživatelům nebyl ze zadání scénáře zcela jasný požadovaný výsledek a museli být tedy seznámeni s konceptem monitorovacího audio výstupu. Tuto funkcionalitu bych však považoval za pokročilou a neočekával bych její intuitivní pochopení od uživatelů bez předchozí zkušenosti. Výsledek tak nepovažuji za výrazně problematický.

Po vysvětlení testovaného konceptu se uživatelům již podařilo audio výstupy samostatně nakonfigurovat a všimli si i nově zobrazeného tlačítka CUE v horní liště u oken zdrojů. Zde pak bylo ještě překážkou pochopit vztah mezi tlačítkem CUE a tlačítky SOLO a MUTE; po chvíli experimentování však dokázali uživatelé sami odvodit, že MUTE a SOLO ovlivňují pouze výstup Master Out a CUE ovlivňuje pouze Monitor Out.

6.1.11 Úprava profilu

Uživatel změní svůj profilový obrázek a přidá popis.

Nalézt cílovou obrazovku Settings pro účastníky nepředstavovalo žádný problém a samotná úprava profilu také proběhla velmi hladce. Jediným možným vylepšením by zde mohlo být přidání samostatného ukládacího tlačítka nad textová pole pro změnu hesla, jelikož jsou tato pole momentálně separována od zbytku formuláře oddělovací čarou a jede z účastníků tak vyjádřil nejistotu nad tím, zda se ukládací tlačítko pod celým formulářem vztahuje pouze ke změně hesla nebo i ostatním vstupům v horní části.

Kapitola 7

Závěr

Vytvořená implementace, ačkoliv ne zcela kompletní, dokázala posloužit jako dobrý důkaz o proveditelnosti a potenciální přínosnosti celkového návrhu. Uživatelské rozhraní aplikace pak i přes určité nedostatky, odhalené v průběhu uživatelských testů, působilo poměrně intuitivním dojmem a novým uživatelům umožnilo tvorbu komplexních layoutů a použití pokročilých funkcí i po pouhých několika minutách používání. Během budoucího vývoje bych se tak chtěl i nadále držet vytvořeného návrhu a dále ho rozvíjet a modifikovat pro ještě snažší použití.

Po stránce funkcionality jsou pak hlavními chybějícími částmi aplikace schopnosti pro nelineární střih obsahu a s nimi související obrazovka Editor. Vzhledem k tomu, že však již v momentální době existuje řada editačních programů pro střih ve webovém prohlížeci, tak se tato část zdála jako méně klíčová než samotné zpracování živých přenosů. Budoucí doplnění této funkcionality by tak nemělo představovat zásadní překážku. Další možností by také mohlo být koncept vestavěného editoru zcela vynechat a místo něj uživatelům umožnit přímý export všech zaznamenaných stop. Poté by už uživatel mohl ke střihu využít svůj preferovaný editační software.

Největší nedostatky v aktuální implementaci se pak projevili zejména u obrazovky Floorplan Editor. Poskytované nástroje tohoto editoru nejsou totiž v momentální chvíli dostatečné na to, aby byla tvorba pohledů opravu ergonomická, a uživatelský zážitek je tak značně frustrující. Samotný koncept Floorplan pohledu, ve kterém může režisér sledovat celý snímaný prostor v rámci jediného okna, se však ukázal jako velmi zajímavý a jeho fungování v praxi předčilo má očekávání. Základní rozložení editoru na účastníky testu také působilo přehledným dojmem a s budoucím přidáním dodatečných nástrojů, jako např. přesnější škálování nebo nastavení průhlednosti vrstev, by se tak z pohledu Floorplan měla stát opravu použitelná možnost pro monitorování živého přenosu.

Celkově bych se pak chtěl v budoucím vývoji také zaměřit na zvýšení odolnosti vůči chybám a výpadkům v průběhu živého přenosu a vzhledem k výpočetní náročnosti živého zpracování audiovizuálního obsahu bych chtěl prozkoumat možnosti pro horizontální škálování aplikace a clusterové nasazení.

Literatura

[1] Roey Izhaki. *Mixing Audio (3rd Edition)*. Routledge, 2017.

[2] Lance Phillips. *Video Editing Made Easy with DaVinci Resolve 18: Create quick video content for your business, the web, or social media*. Packt Publishing, 2023.

[3] Neznámý autor. *Video, Camera, and Production Tools*. StreamYard, Neznámý rok.
`https://support.streamyard.com/hc/en-us/sections/4417411110292-Video-Camera-and-Production-Tools`

[4] Neznámý autor. *What is Studio Web Control?*. Livestream, Neznámý rok.
`https://help.livestream.com/hc/en-us/articles/12653798062609-What-is-Studio-Web-Control-`

[5] Paul William Richards. *The Unofficial Guide to Open Broadcaster Software: OBS: The World's Most Popular Free Live-Streaming Application (Live Streaming Book)*. Nezávisle publikováno, 2019.

[6] Eve Porcello, Alex Banks. *Learning GraphQL: Declarative Data Fetching for Modern Web Apps*. O'Reilly Media, 2018.

[7] Andrew Lombardi. *WebSocket: Lightweight Client-Server Communications*. O'Reilly Media, 2015.

[8] Sebastien Dubois, Alexis Georges. *Learn TypeScript 3 by Building Web Applications: Gain a solid understanding of TypeScript, Angular, Vue, React, and NestJS*. Packt Publishing, 2019.

[9] Ahmed Bouchefra. *Full Stack Development with Angular and GraphQL: Learn to build scalable monorepo and a complete Angular app using Apollo, Lerna, and GraphQL*. Packt Publishing, 2022.

[10] Rohit Rai. *Socket.IO Real-time Web Application Development*. Packt Publishing, 2013.

[11] Vývojářský tým Spring. *Annotation-based Container Configuration*. Spring, Neznámý rok.
https://docs.spring.io/spring-framework/reference/core/beans/annotation-config.html

[12] Remo H. Jansen. *Decorators & metadata reflection in TypeScript: From Novice to Expert*. Wolk Software, 2015.
https://www.wolksoftware.com/blog/decorators-metadata-reflection-in-typescript-from-novice-to-expert-part-4

[13] Parth Ghiya. *TypeScript Microservices*. Packt Publishing, 2018.

[14] Sebastin E. Peyrott. *JWT Handbook*. Auth0 by Okta, 2018.

[15] Salvatore Loreto, Simon Romano, Lorenzo Miniero. *Real-Time Communication with WebRTC: Peer-to-Peer in the Browser*. O'Reilly Media, 2014.

[16] Vývojářský tým mediasoup. *mediasoup v3 Design*. mediasoup, Neznámý rok.
https://mediasoup.org/documentation/v3/mediasoup/design/

[17] Henning Schulzrinne, Stephen L. Casner, Ron Frederick, Van Jacobson. *RTP: A Transport Protocol for Real-Time Applications*. IETF, 2003.
https://datatracker.ietf.org/doc/html/rfc3550

[18] Kirill Konshin. *Next.js Quick Start Guide*. Packt Publishing, 2018.

[19] Michael S. Mikowski, Josh C. Powell. *Single Page Web Applications*. Manning Publications, 2013.

[20] Josh Comeau. *The Perils of Hydration: An Eye-Opening Realization about React*. Nezávisle publikováno, 2020.
https://www.joshwcomeau.com/react/the-perils-of-rehydration/

[21] Liran Cohen. *Reducing HTML Payload With Next.js (Case Study)*. Smashing Magazine, 2021.
https://www.smashingmagazine.com/2021/05/reduce-data-sent-client-nextjs/

[22] Joseph Savona, Josh Story, Lauren Tan, Mengdi Chen, Samuel Susla, Sathya Gunasekaran, Sebastian Markbåge, Andrew Clark. *React Labs: What We've Been Working On – March 2023*. Meta Open Source, 2023.
https://react.dev/blog/2023/03/22/react-labs-what-we-have-been-working-on-march-2023

[23] Balázs Orbán, Delba de Oliveira, DongYoon Kang, Jiachi Liu, JJ Kasper, Lee Robinson, Maia Teegarden, Sebastian Markbåge, Shu Ding, Steven (@styfle), Tim Neutkens. *Next.js 13*. Vercel, 2022.
https://nextjs.org/blog/next-13

[24] Dan Abramov. *RFC: Suspense in React 18*. GitHub, 2022.
https://github.com/reactjs/rfcs/pull/213

[25] Patrick Arminio. *Using Apollo Client with Next.js 13: releasing an official library to support the App Router*. Apollo Graph Inc., 2023.
https://www.apollographql.com/blog/using-apollo-client-with-next-js-13-releasing-an-official-library-to-support-the-app-router

[26] Khalil Stemmler. *TypeScript GraphQL Code Generator – Generate GraphQL Types with Apollo Codegen Tutorial*. Apollo Graph Inc., 2021.
https://www.apollographql.com/blog/typescript-graphql-code-generator-generate-graphql-types

[27] Anderson Osayerie. *Schema Validation with Zod in 2023*. Turing, 2023.
https://www.turing.com/blog/data-integrity-through-zod-validation/

[28] Jozsef Vass. *How Discord handles two and half million concurrent voice users using WebRTC*. Discord, 2018.
https://discord.com/blog/how-discord-handles-two-and-half-million-concurrent-voice-users-using-webrtc

[29] Vývojářský tým mediasoup. *Communication Between Client and Server*. mediasoup, Neznámý rok.
https://mediasoup.org/documentation/v3/communication-between-client-and-server/

[30] Cullen Jennings, Bernard Aboba, Jan-Ivar Bruaroey, Henrik Boström, Youenn Fablet, Daniel C. Burnett, Adam Bergkvist, Anant Narayanan. *Media Capture and Streams*. W3C, 2023.
https://www.w3.org/TR/mediacapture-streams/

[31] Roger Pantos. *HTTP Live Streaming 2nd Edition*. IETF, 2023.
https://datatracker.ietf.org/doc/html/draft-pantos-hls-rfc8216bis

[32] Vývojářský tým video.js. *videojs-http-streaming (VHS) - Compatibility*. GitHub, Neznámý rok.
https://github.com/videojs/http-streaming?tab=readme-ov-file#compatibility

[33] Vývojářský tým Apollo. *Local-only fields in Apollo Client*. Apollo Graph Inc., Neznámý rok.
https://www.apollographql.com/docs/react/local-state/managing-state-with-field-policies/

[34] Daishi Kato. *Micro State Management with React Hooks: Explore custom hooks libraries like Zustand, Jotai, and Valtio to manage global states*. Packt Publishing, 2022.

[35] Michel Weststrate. *Introducing Immer: Immutability the easy way*. Medium, 2018.
https://medium.com/hackernoon/introducing-immer-immutability-the-easy-way-9d73d8f71cb3

[36] Michele Bertoli. *React Design Patterns and Best Practices*. Packt Publishing, 2017.

[37] Adrian Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media, 2016.

[38] Wim Taymans, Steve Bake, Andy Wing, Ronald S. Bultje, Stefan Kost. *GStreamer 1.10 Application Development Manual*. Samurai Media Limited, 2017.

[39] Mathew Patterson. *Create Stunning HTML Email That Just Works*. SitePoint, 2010.

[40] Michael Kerrisk. *Linux Programming Interface*. No Starch Press, 2010.

[41] Vývojářský tým NestJS. *NestJS Documentation - Injection scopes*. NestJS, Neznámý rok.
https://docs.nestjs.com/fundamentals/injection-scopes

[42] Vývojářský tým NestJS. *NestJS Documentation - Modules*. NestJS, Neznámý rok.
https://docs.nestjs.com/modules

[43] Vývojářský tým NestJS. *NestJS Documentation - Techniques - Database*. NestJS, Neznámý rok.
https://docs.nestjs.com/techniques/database

[44] Vývojářský tým NestJS. *NestJS Documentation - Techniques - Configuration*. NestJS, Neznámý rok.
https://docs.nestjs.com/techniques/configuration

[45] Vývojářský tým NestJS. *NestJS Documentation - GraphQL - Quick Start*. NestJS, Neznámý rok.
https://docs.nestjs.com/graphql/quick-start

[46] Vývojářský tým NestJS. *NestJS Documentation - Providers - Factory providers*. NestJS, Neznámý rok.
https://docs.nestjs.com/fundamentals/custom-providers#factory-providers-usefactory

[47] Vývojářský tým TypeORM. *TypeORM - Entities*. TypeORM, Neznámý rok.
https://typeorm.io/entities

[48] Vývojářský tým NestJS. *NestJS Documentation - FAQ - Request lifecycle*. NestJS, Neznámý rok.
https://docs.nestjs.com/faq/request-lifecycle

[49] Vývojářský tým GraphQL. *GraphQL - Execution - Root fields & resolvers*. The GraphQL Foundation, Neznámý rok.
https://graphql.org/learn/execution/#root-fields-resolvers

[50] Vývojářský tým NestJS. *NestJS Documentation - Controllers*. NestJS, Neznámý rok.
https://docs.nestjs.com/controllers

[51] Roman Arutyunyan. *Streaming with nginx-rtmp-module*. Blogger, 2017.
https://nginx-rtmp.blogspot.com/

[52] Vývojářský tým NestJS. *NestJS Documentation - WebSockets - Gateways*. NestJS, Neznámý rok.
https://docs.nestjs.com/websockets/gateways

[53] Vývojářský tým Socket.IO. *Socket.IO - Advanced - Namespaces*. Socket.IO, 2023.
https://socket.io/docs/v4/namespaces/

[54] Vývojářský tým NestJS. *NestJS Documentation - Providers - Services*. NestJS, Neznámý rok.
https://docs.nestjs.com/providers#services

[55] Bu Kinoshita, Zeno Rocha. *React Email - Introduction*. React Email, Neznámý rok.
https://react.email/docs/introduction

[56] François Beaufort. *Autoplay policy in Chrome*. Chrome for Developers, 2017.
https://developer.chrome.com/blog/autoplay

[57] Marcos Caceres. *The User Activation API*. WebKit, 2023.
https://webkit.org/blog/13862/the-user-activation-api/

[58] Gary Kacmarcik. *Keyboard Map*. W3C, 2023.
https://wicg.github.io/keyboard-map/

[59] Paul Adenot, Hongchan Choi, Raymond Toy, Chris Wilson, Chris Rogers. *Web Audio API*. W3C, 2021.
https://www.w3.org/TR/webaudio/

[60] Abdelilah Hamdani, Carlos Barreto. *3D Environment Design with Blender: Enhance your modeling, texturing, and lighting skills to create realistic 3D scenes*. Packt Publishing, 2023.

Nasazení aplikace

Spuštění aplikace vyžaduje instalaci nástrojů Docker a Docker Compose a následné nastavení proměnných prostředí podle specifikace na začátku kapitoly 5. Tyto proměnné lze také specifikovat v souboru `.env` v kořenové složce projektu. Aplikaci lze pak spustit zavoláním příkazu `docker-compose up`, také v kořenové složce projektu. Po spuštění všech kontejnerů lze k aplikaci přistoupit na portu `GATEWAY_PORT`. Aplikaci je následně možné ukončit příkazem `docker-compose down`.