VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

APLIKACE PRO FESTIVALOVÉ NÁVŠTĚVNÍKY NA IOS

IOS APPLICATION FOR FESTIVAL VISITORS

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **JAN MENŠÍK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. PETR BOBÁK**
SUPERVISOR

BRNO 2021

Vysoké učení technické v Brně
Fakulta informačních technologií

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2020/2021

Zadání bakalářské práce

Student: **Menšík Jan**
Program: Informační technologie
Název: **Aplikace pro festivalové návštěvníky na iOS**
iOS Application for Festival Visitors
Kategorie: Uživatelská rozhraní

Zadání:

1. Seznamte se s architekturou iOS, jazykem Swift a frameworky SwiftUI a UIKit.
2. Prozkoumejte klient-server architekturu a nástroje vhodné pro tvorbu webových služeb.
3. Iterativním způsobem navrhněte uživatelské rozhraní aplikace a klient-server architekturu informačního systému pro festivalové návštěvníky.
4. Navržený systém a klientskou aplikaci implementujte.
5. Otestujte funkcionalitu a použitelnost výsledného řešení.
6. Zhodnoťte dosažené výsledky, vytvořte plakát a krátké prezentační video. Dále navrhněte možné pokračovaní.

Literatura:

- dle pokynů vedoucího

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Bobák Petr, Ing.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2020
Datum odevzdání: 12. května 2021
Datum schválení: 30. října 2020

Abstrakt

Cílem této práce je implementovat iOS aplikaci sdružující informace o festivalech na jedno místo. Uživatelům má poskytnout přehled o konaných událostech a pomoct jim při organizaci času. Práce popisuje proces tvorby aplikace od návrhu uživatelského rozhraní a systémové architektury až po implementaci včetně závěrečného testování. Výsledkem je klient-server aplikace umožňující organizátorům zaregistrovat do systému libovolný typ festivalu prostřednictvím webového rozhraní, který se následně zobrazí uživatelům v mobilní aplikaci. Serverová část je implementovaná ve frameworku Django, klientská aplikace ve frameworku SwiftUI.

Abstract

The aim of this thesis is to implement iOS application aggregating information about festivals to one place. Application should provide general overview about individual festival events and help users with time planning. Thesis is describing process of creation from user interface desing and system architecture to implementation with final testing. The result is client-server application providing web interface to organizers for any kind of festival registration which is later available in the mobile application. Server is implemented in Django framework, client application in framework SwiftUI.

Klíčová slova

Apple, iOS, aplikace, Swift, SwiftUI, Xcode, uživatelské rozhraní, model klient-server, webový server, Django, Python, REST API

Keywords

Apple, iOS, application, Swift, SwiftUI, Xcode, user interface, client-server model, web server, Django, Python, REST API

Citace

MENŠÍK, Jan. *Aplikace pro festivalové návštěvníky na iOS*. Brno, 2021. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Petr Bobák

Aplikace pro festivalové návštěvníky na iOS

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana inženýra Petra Bobáka a uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

.....
Jan Menšík
5. května 2021

Poděkování

Rád bych poděkoval svému vedoucímu za odbornou pomoc a pravidelné konzultace, které mě při tvorbě práce navedly správným směrem a pomohly mi práci dokončit.

Obsah

Kapitola 1

Úvod

Žijeme v době informací. Téměř každý v této době vlastní chytrý mobilní telefon a denně ho využívá, ať už k práci, zábavě či právě k zjišťování informací prostřednictvím internetu. Informace se za tuto dobu pomalu dostávají do aplikací tvořených na míru nejrůznějším službám a obchodům.

Dnešním moderním trendem je návštěva festivalů a počet jejich návštěvníků v průběhu let stoupá [30]. Lidé se chtějí bavit a mít spoustu společných zážitků. Účastníci festivalů si nejprve svoji událost musí najít. Zajímá je, kdo na festivalu vystupuje či jaká představení a akce se na něm konají. Následně si zakoupí lístky a na festival se vydají. pro zorientování se na dané akci následně musí vyhledat, která místa a vystoupení chtěli navštívit. K tomu jim poslouží internet, aplikace konkrétního festivalu či informační rozpisy v místě konání. Větší festivaly se mohou konat i na více místech zároveň. Návštěvníci si tudíž potřebují rozvrhnout, kterých událostí v rámci festivalu se chtějí účastnit a naplánovat si čas na delší přesuny.

Cílem práce je vytvořit klient-server aplikaci pro festivalové účastníky, která klade důraz sjednotit veškeré informace o festivalech na jedno místo. Klíčovou motivací tvorby aplikace je absence obdobné služby na trhu mobilních aplikacích. Organizátoři si budou moci svůj festival zaregistrovat přes webové rozhraní. Poskytnou informace o samotném festivalu a dále o jednotlivých jeho akcích. Uživatelé pak tyto údaje uvidí v mobilní aplikaci, která jim pomůže se v rámci festivalu lépe zorientovat a utvořit si vlastní plán. Generika návrhu aplikace poskytne možnost registrovat jakýkoliv festival bez ohledu na jeho tématiku, ať už se jedná o filmový, hudební či například food festival. Aplikace tak bude schopna poskytnout prostor i menším událostem. Pomůže jim v jejich zviditelnění a nabízí alternativu vlastního řešení na míru, na které tito organizátoři menších festivalů nemusí mít finanční prostředky.

Práce na začátku popisuje architekturu klient-server modelu (2) a platformy iOS (3), dále se zabývá zkoumáním a analýzou podobných existujících aplikací společně s návrhem uživatelského rozhraní (4). Následuje tvorba architektury systému a rozložení povinností mezi klientskou a serverovou část, ke které je přiložen popis výběru implementačních technologií (5). na to práce navazuje technickým popisem samotné tvorby aplikace (6). Ke konci je provedeno testování funkčnosti a uživatelského rozhraní aplikace (7) a je diskutováno nad finálním produktem (8).

Kapitola 2

Model klient-server

Tato kapitola pojednává o modelu klient-server a jeho smyslem v moderním světě. V úvodu kapitoly se čtenář dozví základní informace o modelu, následuje vysvětlení, z jakých částí je model složen a jakou roli v něm hrají. V neposlední řadě jsou uvedeny základní architektury tohoto modelu, jejich členění a reálné uplatnění v dnešní době. na závěr je vysvětleno, jak v modelu probíhá komunikace a jaké protokoly a pravidla se při této komunikaci uplatňují.

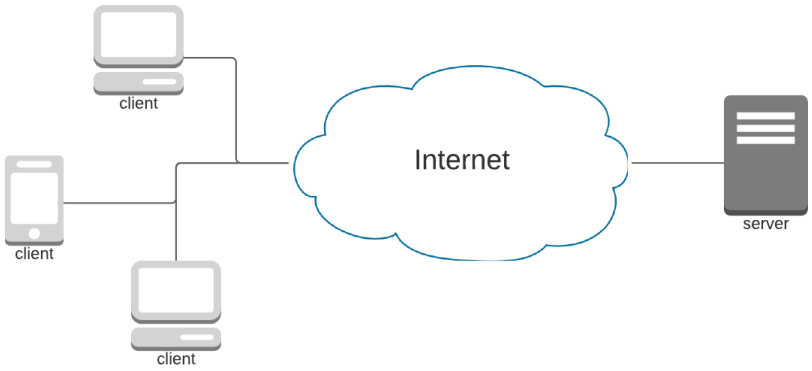
2.1 Definice modelu a jeho rozdělení

Klient-server model je dle definice [20] síťová architektura, ve které server poskytuje prostředky a služby jednomu či více klientům. Klient zasílá požadavky na server s cílem získat od něj určité informace a server na žádost klienta odpovídá. Klient-server aplikace definujeme jako distribuované, jelikož ve většině případů platí, že klientský program běží na jednom zařízení, zatímco serverový program běží na zařízení jiném. Využití modelu najdeme například ve službách jako jsou bankovnictví, zasílání e-mailů nebo přístupu na internet. Hierarchické uspořádání jednotlivých zařízení v síti lze vidět na obrázku 2.1.

V dřívější době tento model fungoval připojováním „hloupých“ terminálů, které uměli pouze zobrazit data, k mainframu – centrálnímu počítači. S vývojem počítačů začala být výpočetní síla i na straně klientů, což vedlo k jejímu rovnoměrnějšímu rozložení a snížení náročnosti na straně serveru [3].

Klientská část modelu se nachází na běžných zařízeních jako jsou osobní počítače, notebooky a mobilní zařízení, na kterých běží aplikace, typicky s uživatelským prostředím, která zpracovává informace poskytované serverem. V síti zahajuje komunikaci klient zasláním dotazu na předem známý server. Zároveň však klient nemůže přímo komunikovat s klientem jiným, toto je umožněno v architektuře Peer-to-peer.

Serverová část modelu se nachází na výpočetně silnějších zařízeních v síti, která jsou neustále zapnutá, aby s nimi klient mohl kdykoliv zahájit komunikaci. Server čeká na požadavek od klienta a odpovídá mu nalezenými daty. Architektura modelu dovoluje mít více klientů, kteří se snaží kontaktovat jeden server. Z tohoto důvodu existují datacentra, která obsahují stovky až tisíce zařízení, které dokáží dohromady tvořit jeden virtuální server a řeší tak problém zahlcení požadavky [31].



Obrázek 2.1: Ukázka hierarchie různých zařízení v síti. Klientem může být stolní počítač, notebook nebo mobilní telefon. Server bývá zpravidla počítač. Klient i server jsou připojeni k internetu.

2.2 Základní typy architektur

Při využití modelu se funkcionalita systému musí rozdělit mezi jeho jednotlivé části. Separace zajistí větší přehlednost při vývoji, modularitu a zefektivňuje výpočetní sílu.

2-vrstvá architektura

Tato architektura, jak její název napovídá, se skládá ze 2 základních částí.

Klientská aplikace má za úkol komunikovat s uživatelem a poskytnout mu prostředí s možností dotázat se na data. Zároveň však obsahuje aplikační logiku, která realizuje propojení aplikace s databází a formování dotazů pro zisk korektních dat.

Databáze slouží jako úložiště dat, kterých se klienti mohou dožadovat. Nachází se zpravidla na jiném zařízení připojeném k síti.

Hlavní výhoda architektury spočívá v menší náročnosti návrhu a implementace aplikace. Oproti tomu je však složitější udržovat klientskou část, jelikož obsahuje aplikační logiku a je uložena lokálně u více klientů.

3-vrstvá architektura

Ukázalo se, že aplikační logika a funkcionalita na straně klientské aplikace je nepraktická, proto vznikla architektura 3-vrstvá. Dělí se na následující celky.

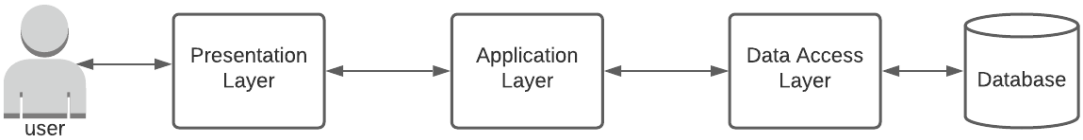
Prezentační vrstva zajišťuje interakci s uživatelem. Poskytuje mu informace ze serveru v čitelné formě a zpracovává jeho vstupy, které potom přeposílá serveru a obdrženou odpověď zobrazuje.

Aplikační vrstva leží uprostřed logického uspořádání a nese v sobě logiku celého modelu. Zpracovává požadavky, validuje příchozí data, rozhoduje o odpovědích a dalších úkonech. na rozdíl od 2-vrstvé architektury se logika této vrstvy nachází na samotném serveru. Zpracovává požadavky z prezentační i datové vrstvy a vyhodnocuje je. Nemá

v sobě uložená žádná konkrétní data a pokud tato data potřebuje zjistit, kontaktuje databázový server.

Datová vrstva obsahuje systém správy samotné databáze a všechna uložená data.

Každá vrstva se nachází na samostatném zařízení. Tato skutečnost nám přináší dle článku [29] několik výhod. Údržba aplikace se stala snesitelnější kvůli její modulárnosti. Izolace aplikační vrstvy mezi vrstvu prezentační a datovou zvýšilo integritu přenášených dat. Zároveň může prezentační vrstva při velkém množství požadavků lépe distribuovat zátěž mezi známé servery. Vzájemnou komunikaci mezi jednotlivými vrstvami lze vidět na obrázku 2.2.



Obrázek 2.2: Komunikační tok v 3-vrstvé architektuře.

2.3 Komunikace klienta a serveru

Celý model jsme tedy rozdělen na několik vrstev, přičemž každá je zodpovědná za příslušnou část architektury. Nyní je uveden způsob, jakým mezi sebou klient a server mohou komunikovat, jaké principy je při komunikaci vhodné dodržovat a budou popsány nejznámější přístupy. pro lepší pochopení je na začátku uvedena definice webové služby.

Webová služba (anglicky `web service`) je podle Mezinárodního webového konsorcia [34] softwarový systém navržený pro podporu vzájemné interakce zařízení přes síť.

2.3.1 API

Aplikační programové prostředí neboli API[1] je způsob propojení částí aplikace či různých dvou aplikací, který slouží k výměně informací. Hlavním cílem je sdílení části funkcionality pro urychlení a zjednodušení tvorby aplikace vývojářům. Tato prostředí existují pro různé typy systémů, přes operační systémy až po knihovny a samotný web. Webové API je speciálním typem prostředí, kde je komunikace po internetu zajištěna pomocí specifických webových protokolů. Prostředí jako takové poskytuje soubor endpointů (koncových bodů), formáty požadavků a odpovědí. pro komunikaci přes aplikační programové prostředí dnes využíváme hypertextový protokol HTTP[2].

2.3.2 Simple Object Access Protocol

Simple Object Access Protocol, zkráceně SOAP, je standardizovaný protokol určený k výměně dat v distribuovaném prostředí [17]. Dokáže pracovat společně s protokoly z aplikační

[1] Application Programming Interface
[2] Hypertext Transfer Protocol

vrstvy, jako je HTTP, SMTP[3] aj. Obsahuje vestavěnou autorizaci a řešení chyb, čímž zajišťuje určitou míru bezpečnosti. Dle článku [33] se protokol bude v následujících letech využívat pro služby, které vyžadují vysokou bezpečnost přenosu dat, jako třeba bankovní transakce nebo telekomunikace. Přenosovým formátem dat je zde XML, ke kterému jsou přidány speciální řídicí prvky specifické pro SOAP. Každá odeslaná zpráva se skládá z následujících částí:

- **Obálka** – Údaj vymezující začátek a konec zprávy.
- **Hlavička** – Část zprávy, která může obsahovat rozšiřující informace o zprávě, například její prioritu nebo čas expirace. Není povinnou součástí zprávy.
- **Tělo** – Samotná data ve formátu XML přenášená serverem k příjemci.
- **Chybové informace** – Volitelná část zprávy, která může obsahovat informace o chybách při jejím zpracování.

2.3.3 REST

REST (Representational State Transfer) je architektonický styl určený pro distribuované mediální systémy, který definuje chování API. Je postavený na klient-server modelu a byl navržen Royem Fieldingem v roce 2000, jenž ho poprvé zmínil ve své disertační práci [24]. REST není na rozdíl od SOAP protokolem, komunikace tudíž probíhá přes HTTP. Správně navržený systém (angl. označovaný jako restful) musí splňovat následující požadavky [35]:

- **Client-Server** – Prostředí musí mít logicky oddělené uživatelskou část systému a místo uložení dat, aby byla zajištěna přenositelnost uživatelské části systému pro různé platformy a snazší práce s komponentami serveru.
- **Stateless** – Klient musí komunikovat se serverem bezstavově, tzn. každá žádost poslaná na server musí obsahovat veškerá data potřebná pro její interpretaci, není možné využít uložení kontextu na serverové straně.
- **Cacheable** – Uživatel musí mít možnost uložit si získaná data od serveru pro jejich pozdější využití při odeslání stejné žádosti.
- **Uniform Interface** – Tato podmínka odlišuje `restful` rozhraní od jiných architektur užitím principu generality komponent systému.
- **Layered System** – Přidání vrstev rozčlenní systém na více částí, každá komponenta systému vidí pouze do vrstvy, ve které se nachází.
- **Code on demand** – Dobrovolnou poslední podmínkou je možnost rozšíření funkcionality klienta stažením a spuštěním kódu ve formě skriptů či appletů.

Základní jednotkou informací v RESTu je *zdroj*. Zdrojem je jakákoliv informace, která se dá v systému pojmenovat, např. dokument, obrázek či klidně skupina jiných zdrojů. Každý zdroj je adresovatelný pomocí URI[4]. při komunikaci vzniká problém aktuálnosti zdroje. Uživatel například požádá internetový blog o zaslání nejnovějšího příspěvku, ten je

[3] Simple Mail Transfer Protocol – protokol pro odchozí e-mailovou poštu
[4] Uniform Resource Identifier – řetězec sloužící pro identifikaci zdrojů, viz RFC 3986 [16].

ale každý den jiný, protože editor blogu průběžně přidává nové příspěvky. Nemůžeme tedy spojit zdroj s konkrétním jménem, protože bychom dostávali pořád stejnou odpověď.

Zde se dostáváme k druhému důležitému pojmu, *reprezentaci*. Reprezentace je aktuálním stavem nějakého zdroje, která je zasílána mezi klienty a serverem. Formát reprezentace může být různý, od textového souboru přes obrázek nebo například JSON[5] či XML[6] notaci. Zároveň platí, že více reprezentací může ukazovat na jeden zdroj.

Komunikace nám umožňuje uplatňovat na zdroje tzv. CRUD akce (zkratka pro: Create, Read, Update, Delete). Komunikačním protokolem je HTTP, můžeme tak tyto akce jezdnoznačně přiřadit k protokolovým metodám.

Akce	**HTTP Metoda**	**Popis**
CREATE	POST, PUT	Vytvoření zdroje
READ	GET	Přečtení zdroje
UPDATE	PUT, PATCH	Upravení zdroje
DELETE	DELETE	Vymazání zdroje

Tabulka 2.1: Vysvětlení a přiřazení CRUD akcí k HTTP metodám.

Při porovnání RESTu a protokolu SOAP zjistíme, že ačkoliv je SOAP mohutnější a poskytuje vyšší bezpečnost, je pomalejší. Nutností použití je také omezení se na jediný přenosový formát, XML. pro veřejně dostupné API je lépe využitelný REST. Není v něm omezení přenosového formátu, spotřebuje menší šířku pásma a je rychlejší, viz článek [6].

[5] Javascript Object Notation – jazykový formát pro výměnu dat, viz RFC 8259 [18].

[6] Extensible Markup Language – jazyk pro strukturování dat, viz specifikace Mezinárodního webového konsorcia [19].

Kapitola 3

Mobilní platforma iOS

Následující kapitola obsahuje informace k operačnímu systému iOS běžícím na mobilních zařízeních vytvořených společností *Apple Inc.* na úvod kapitoly je řečeno několik slov o samotné společnosti. Hned poté je stručně popsána historie platformy a průběh jejího vývoje až do dnešních let. Vzápětí je vysvětlena architektura této platformy až k samotnému jádru operačního systému a ke konci jsou uvedeny prostředky pro vývoj nativních aplikací platformy a jejich klíčové vlastnosti. Jak bylo uvedeno v úvodu, iOS, dříve iPhone OS, je mobilní operační systém vyvíjený a produkovaný společností *Apple Inc.* pro její mobilní zařízení iPhone.

V dubnu roku 1976 byla společnost *Apple Inc.*, tehdy pod názvem Apple Computer, Inc., založena Stevem Jobsem a Stevem Wozniakem. Klíčovým nápadem byla myšlenka vytvoření počítače velikostně použitelného v běžné domácnosti či kanceláři. Tehdejší první počítač (Apple I) byl vyvíjen v garáži Steva Jobse a byl prodáván bez jakéhokoliv periferního zařízení [37]. Dnes má Kalifornská společnost *Apple Inc.* biliónovou tržní hodnotu [36] a prodává mobilní telefony, tablety, chytré hodinky, počítače, notebooky a další zařízení.

3.1 Vývoj systému a důležité milníky

První iOS byl představen společně s prvním mobilním zařízením společnosti v roce 2007 na konferenci Macworld Keynote zakladatelem Stevem Jobsem [28]. na konferenci nazýval Steve Jobs systémem OS X[1], protože byl založený na stejném UNIXovém jádře a zabíral celou obrazovku zařízení. Dle slov Steva Jobse tento operační systém sjednotil dohromady tři základní aspekty: multimédia, telekomunikaci a přístup k internetu. Verze této platformy obsahovala pouze několik vestavěných aplikací, jako byly Mail, Calendar, Photos a další. iOS jako první také podporoval technologii multi-touch, která umožnila uživatelům ovládat prostředí pomocí specifických gest. Tehdy ještě systém neumožňoval aplikace třetích stran a Steve Jobs nabádal vývojáře, aby navrhovali webové aplikace, které budou mít podobné chování jako aplikace nativní [26].

O rok později změnila společnost názor a prezentovala první iOS SDK[2], balíček nástrojů pro tvorbu nativních aplikací [27], a aplikaci App Store, která aplikace vývojářů třetí strany umožnila poskytnout ke stažení.

V průběhu následujících let se do systému přidali další prvky, například geolokace, notifikace či vlastní mapy.

[1] Původní název operačního systému pro počítače společnosti *Apple Inc.*

[2] Software development kit – soubor nástrojů pro tvorbu nativních aplikací.

Další velká změna nastala v sedmé verzi iOS, kde bylo představen zbrusu nový design operačního systému [32]. Až do této verze byl využíván designový styl originálně nazývaný Skeumorphic, který se vyznačuje podobností objektů v aplikaci s objekty reálného světa. Charakterizují ho stíny, 3D struktury, textury a další prvky. Aplikace na poznámky má vypadat jako reálný papír s nakreslenými linkami, kalendářová aplikace měla zase vypadat jako reálný papírový visací kalendář. Od verze iOS 7 začal operační systém využívat tzv. Flat design. Cílem tohoto vzhledu je poskytnout uživateli možnost maximálně se soustředit na kontext informací na obrazovce, nikoliv na jednotlivé detaily designu. Styl se vyznačuje svojí jednoduchostí ve využití barev a tvarů. Porovnání lze vidět na obrázku 3.1.

Během dalších let se iOS dostal až do své aktuální verze 13. Klíčové novinky systému jsou předělané vestavěné aplikace, nové mapy nebo dark mode, který umožňuje používat iPhone v tmavém režimu [9].

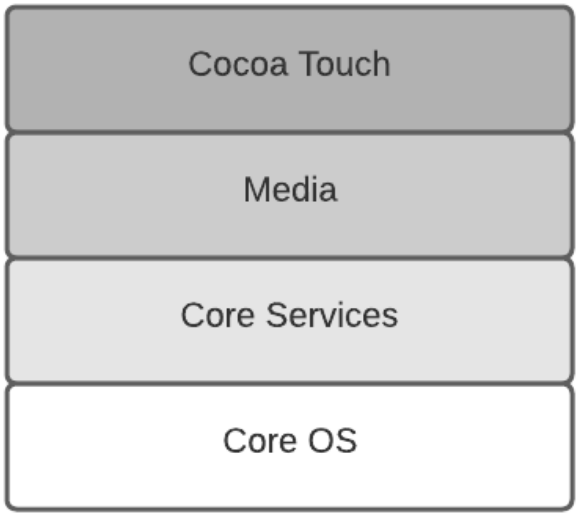
Obrázek 3.1: Porovnání Skeumorphic designu (na levé straně) a Flat designu (na pravé straně). Obrázek je dostupný na adrese https://inkbotdesign.com/wp-content/uploads/2014/05/apple-ui-design.jpg.

3.2 Architektura

Nyní jsou uvedeny čtyři základní vrstvy celé architektury, které zahrnují všechny frameworky a technologie implementované v iOS [22]. Vrstvy představují určitou abstrakci systému, kdy nejvyšší vrstva obsahuje frameworky, které jsou často objektově-orientovanou abstrakcí frameworků z vrstev nižších. Každá z nich tedy komunikuje pouze s vrstvami sousedními (viz obrázek 3.2).

Cocoa Touch Layer je nejvyšší vrstva celé architektury. Nachází se v ní sada objektově-orientovaných frameworků, které poskytují prostředí pro běh aplikací na iOS.

Media Layer se v architektuře stará o zpracování audia a videa, vykreslování grafických elementů, zobrazování animací, textu a další. Je úzce závislá s vrstvou Core Services, se kterou kooperuje.



Obrázek 3.2: Jednotlivé vrstvy iOS architektury seřazené sestupně podle míry abstrakce.

Core Services Layer obsahuje služby zajišťující jednotlivé datové typy, síťovou komunikaci, persistenci dat a částečný přístup k hardwarové složce zařízení, jako GPS, kompasu a dalších.

Core OS Layer je nejnižší vrstva, která komunikuje přímo s hardwarem v zařízení. Stará se o správu paměti, její alokování a uvolňování pro aplikace, využití BSD socketů pro komunikaci po internetu, vypínání aktuálně nevyužívaných části hardwaru a další.

Pro každou Cocoa aplikaci jsou dle dokumentace [7] nepostradatelné tyto konkrétní frameworky: `Foundation` a `UIKit`, dnes nahraditelný za `SwiftUI`. `Foundation` [8] poskytuje objekty pro jednotlivé primitivní datové typy, kolekce a další. Zároveň se stará o jejich základní chování a správu. Nachází se ve vrstvě `Core Services`. `UIKit` a `SwiftUI` vývojáři poskytují prostředí pro tvorbu uživatelských rozhraní. Více o těchto frameworcích pojednávám v podkapitole 3.5.

3.3 Programovací jazyk Swift

Na vytváření aplikací a programů pro platformu iOS se dlouhá léta používal programovací jazyk Objective-C, který je nadstavbou jazyka C. Zdědil po něm syntaxi, primitivní datové typy, řízení toku aplikace a je rozšířen o možnost programování v objektově-orientovaném stylu [11]. V roce 2014 představila společnost *Apple Inc.* na WWDC nový programovací jazyk zvaný Swift [12].

Cílem Swiftu je moderní přístup ke kritickým oblastem programování. Vývojářům přináší bezpečnější formu tvoření. Proměnné musí být inicializované před prvním použitím a správa paměti je automatická. Nedovolí žádnému objektu, aby byl prázdný, místo toho přichází s novinkou v podobě `optionals`, způsobu, který rozšiřuje obor hodnot každého datového typu o `nil` neboli možnost být prázdný a definuje ochranné prvky, jak k takovému datovému typu můžeme bezpečně přistoupit.

3.4 Vývojářské prostředí Xcode

S příchodem možnosti tvorby iOS aplikací třetích stran bylo nezbytné, aby měli vývojáři dostupné prostředky, které k tomu budou moci využít. Překladače na zmíněné programovací jazyky existovaly, ale vytvářet aplikace s uživatelským rozhraním v textovém editoru se jevilo velmi obtížné. Apple tudíž přišel s vlastním vývojářským prostředím, zvaném Xcode [15].

Xcode je sada vývojářských prostředků pro vytváření, testování, optimalizování a v neposlední řadě také publikování aplikací pro Apple zařízení. Základní šablony umožňují rychlý start psaní kódu. Emulátor iOS vypomáhá při vyzkoušení funkčnosti a odhalování chyb v uživatelském prostředí. Dokáže simulovat všechny Apple zařízení, které jsou momentálně v prodeji a zajišťuje tím testování kompatibility aplikace směrem k většinové části běžných uživatelů.

Pro vytváření uživatelského prostředí aplikace slouží grafický editor. Vývojáři zde ručně přidávají prvky, které následně mohou upravovat a propojovat mezi sebou. Xcode následně vytvoří zdrojový kód tohoto návrhu pro zpracování překladačem.

3.5 UIKit a SwiftUI

Pro tvorbu uživatelských rozhraní existují dvě základní nativní knihovny, které se od sebe však diametrálně liší. Pomocí obou jsme schopni implementovat grafické uživatelské prostředí, každá z nich však využívá zcela odlišného přístupu při návrhu.

Knihovna `UIkit` [14] vznikla jako první. Poskytuje vývojářům základní sadu grafických elementů použitelných pro tvorbu interaktivního grafické prostředí poháněného událostmi, možností zpracování uživatelského vstupu a hlavní smyčku udržující aplikaci v chodu. Knihovna stojí na návrhovém stylu MVC[3], jehož diagram je možné vidět na obrázku 3.3. Pro zobrazení konkrétní obrazovky, neboli `View` je zapotřebí vytvořit tzv. `ViewController`, který se stará o její inicializaci, vykreslování a zpracování událostí. Důležitým prvkem knihovny je však možnost netvořit obrazovky programově, ale vizuálně. `Storyboards` dává programátorovi možnost navrhovat obrazovky aplikace pomocí grafického editoru. V jednom souboru dokáže vytvořit více různých obrazovek a zajistit přechody mezi nimi. Každou takto vytvořenou obrazovku poté vývojář propojí s `viewcontrollerem`, který se bude starat o její chování.

`SwiftUI` [13] si dal za cíl zjednodušit tvorbu uživatelského prostředí na nezbytné minimum a přichází s novým, deklarativním návrhem obrazovek. Aplikaci v podstatě řekneme, jaké prvky chceme na obrazovce, jak se mají uspořádat a jaké mají mít vlastnosti. Jednotlivé grafické prvky jsou implementované jako struktury, které se navzájem mohou libovolně zanořovat. Tím vzniká hierarchické uspořádání, ze kterého se vykreslí finální obrazovka. Všechny grafické elementy, neboli `views` jsou datového typu `struct`. na každé `view` lze aplikovat specifické modifikátory, které element upraví podle našich představ. Základními modifikátory mohou být například barva, velikost, pozice či stanovené rozměry.

Zatímco v `UIkitu` se změna na obrazovce musela implementovat úpravou modelu a následným zavoláním překreslení obrazovky, ve `SwiftUI` je přístup odlišný. Každé `view` může obsahovat tzv. stavovou proměnnou. Když nastane změna hodnoty této proměnné, obrazovka se automaticky překreslí a zahrne všechny změny v datové části aplikace. Stav

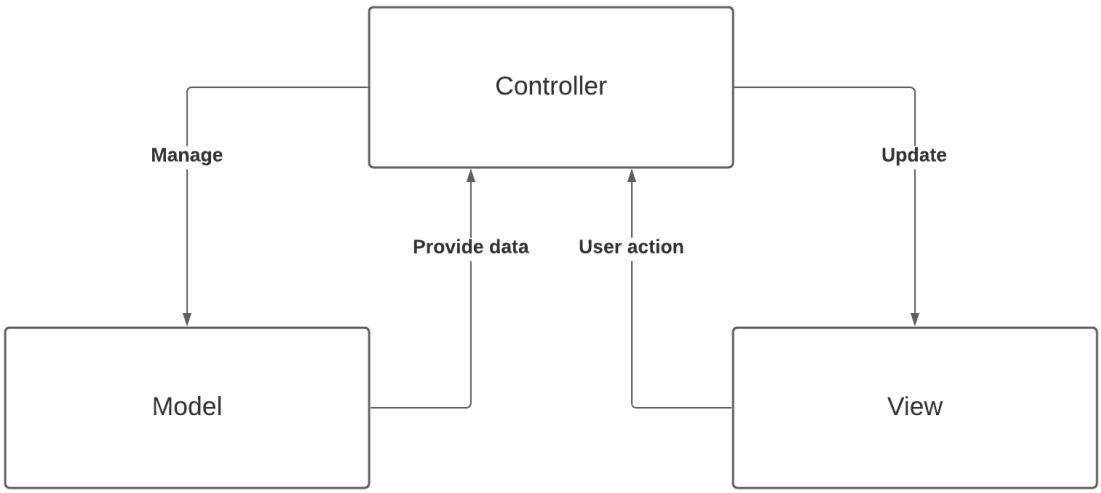
[3] Model-View-Controller – návrhový styl uplatňovaný při implementaci aplikací.

můžeme sdílet do dalších obrazovek pomocí `bindingu` a zefektivnit tak tok důležitých dat pro vykreslování aplikace.

Co se týče rozložení jednotlivých grafických prvků na obrazovce, nabízí `SwiftUI` speciální `views`, které seřadí vnořené grafické prvky podle daného způsobu. Prvky lze rozložit vertikálně, horizontálně či skládat na sebe, můžeme je dát do automaticky se tvořícího seznamu nebo tabulky.

Největší nevýhodou `SwiftUI` zůstává fakt, že byla knihovna představena na podzim roku 2019 v rámci konference WWDC[4] [5] a stále se nachází na počátku vývoje. Vývojové prostředí občas hlásí chyby v syntaxi kódu, přičemž je kód bezchybný a je potřeba program restartovat, a spousta chybových hlášení neodpovídá reálnému pochybení v kódu aplikace.

Přestože jsou knihovny svým přístupem pro tvorbu uživatelských rozhraní odlišné, mají vývojáři možnost je zkombinovat. `SwiftUI` některé grafické elementy z `UIkitu` neobsahuje, dovolí je však implementovat pomocí struktury typu `UIViewRepresentable`, která poskytne prostředí pro tvorbu `ViewControlleru` a `View` pomocí `UIkitu`.



Obrázek 3.3: Diagram návrhového stylu Model-View-Controller.

[4] Apple Worldwide Developers Conference – Konference, na které společnost *Apple Inc.* představuje nové produkty a technologie.

Kapitola 4

Tvorba uživatelského rozhraní aplikace

Kapitola se zaměřuje na proces tvorby uživatelského rozhraní vzhledem k jejímu hlavnímu účelu. na úvod specifikuje základní požadavky aplikace. V dalším kroku analyzuje podobná existující řešení. na závěr kapitoly je prezentován vlastní návrh uživatelského rozhraní, který bere v potaz základní požadavky a informace získané analýzou existujících aplikací.

4.1 Základní požadavky

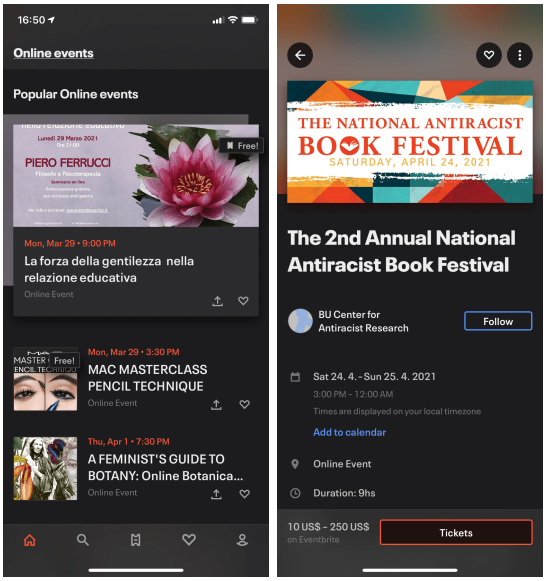
Hlavním účelem aplikace je zobrazovat informace o festivalech a sjednotit je na jedno místo. Cílový uživatel tak nebude muset pro každý festival zjišťovat informace v různých aplikacích či na internetu, ale budou mu dostupné na jednom místě. Hlavním požadavkem na aplikaci tudíž bude jednotnost uživatelského rozhraní. Velkou nevýhodou konkrétních aplikací vytvořených pro účely konkrétních festivalů, např. Majálesu (viz podkapitola 4.2), vnímám roztříštěnost jednotlivých rozhraní. Každá taková aplikace používá jiná barevná schémata, uspořádání ovládacích prvků a uspořádání obrazovek, což je přirozené, jelikož jsou tyto aplikace vytvořené na míru. Dalším požadavkem je rychlá dostupnost informací. Uživatel by měl při použití aplikace zobrazit požadované informace v co nejkratším čase. Důležité proto bude rozvržení struktury obrazovek a přehlednost prostředí.

4.2 Ukázka existujících řešení

V této sekci představím podobná existující řešení jak aplikací pro jednotlivé festivaly, tak aplikací zaměřujících se na sjednocení informací o událostech podobného či stejného druhu. Každou aplikaci krátce popíšu a zhodnotím její použitelnost. na konci všechny poznatky využiji k vylepšení požadavků na aplikaci.

Eventbrite

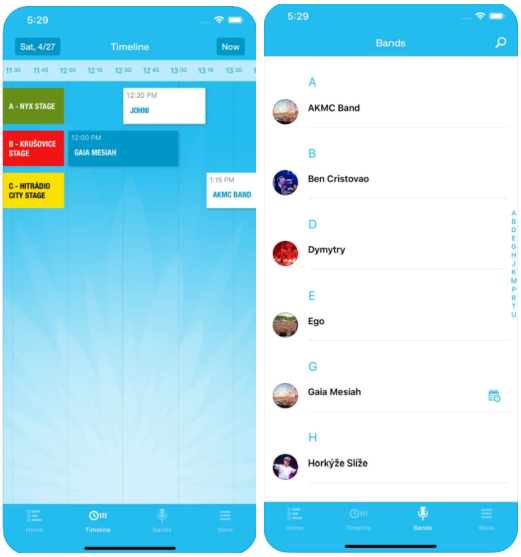
Hlavním účelem této aplikace (viz obrázek 4.1) je sdílení událostí různých druhů, od přednášek, seminářů až po workshopy. za velkou výhodu považuji přehlednost v seznamu událostí, který má jednotný vzhled, až na první událost, která je vyobrazena ve větším provedení. Detail události obsahuje stručně a přehledně základní informace. Nevýhodou zde může být absence podpory vytvořit komplexnější událost, která by obsahovala více menších akcí.



Obrázek 4.1: Uživatelské rozhraní aplikace Eventbrite.

Majáles

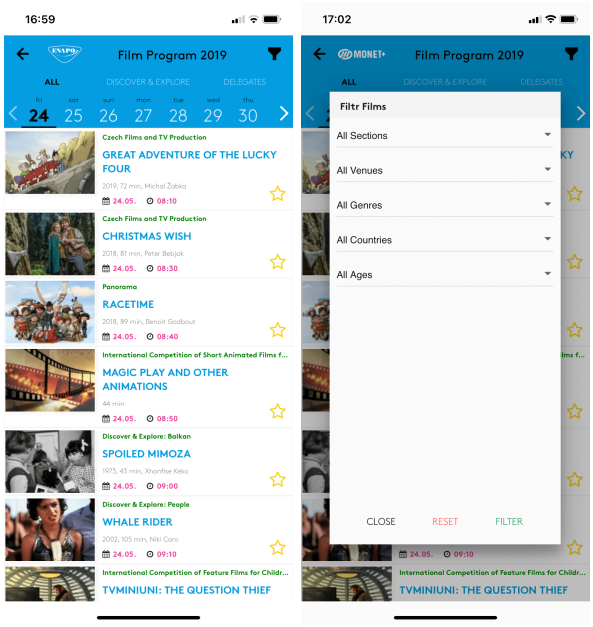
Velmi známý český hudební festival, určený zejména studentům, má také svoji mobilní aplikaci (viz obrázek 4.2) pro lepší dostupnost informací svým příznivcům. Nalezneme v ní kromě abecedně seřazeného seznamu interpretů také časovou osu, která graficky zobrazuje, kdy se která vystoupení konají a na jakém místě. Tuto možnost vnímám velice pozitivně, jelikož grafické zpracování informací značně pomáhá uživatelům v orientaci a je přehlednější než pouhý seznam. Mohou z něj jednoduše zjistit, která vystoupení se časově překrývají a upravit podle toho své plány. V seznamu jde navíc interprety vyhledávat, což je další pozitivum vzhledem k jejich celkovému počtu.



Obrázek 4.2: Uživatelské rozhraní aplikace Majáles.

Zlinfest

Tato aplikace (viz obrázek 4.3) byla vytvořena pro filmový festival konající se ve Zlíně. Kromě obecných informací ukazuje uživatelům seznam filmů a akcí, seřazených podle jejich času konání, což pokládám za přehledné. Vše je kategorizováno do jednotlivých dní, mezi kterými si uživatel může přepínat. Lze zde filmy i vyhledávat. Kromě toho je v ní přidané filtrování, ve kterém se události dají ještě třídit podle sekcí, žánrů a dalších kritérií. Tato vlastnost zde působí velice vhodně, jelikož si uživatel může zvolit přesně, co chce najít mezi seznamem desítek filmů či akcí. Drobnou nevýhodou tohoto filtru je však nemožnost zvolit více možností z jedné kategorie. Nachází se zde i možnost tvořit si vlastní program, který se zobrazuje v samostatné záložce. Naneštěstí tato vlastnost v aplikaci nefunguje, což vedlo mimo jiné k jejímu nízkému hodnocení.



Obrázek 4.3: Uživatelské rozhraní aplikace Zlinfest.

Závěrečný souhrn poznatků

Po prozkoumání aplikací jsem dostal zcela nový pohled na určité aspekty funkcionality, které úzce souvisí s návrhem uživatelského rozhraní. Všechny aplikace měly na obrazovce, kde vykreslovaly seznam událostí, většinu prostoru využitou pro tento účel, aby tak uživatel mohl vidět co nejvíce relevantních informací. Ovládací prvky byly často zahrnuté v rámci `navigation baru`[1], nerušily tak hlavní zobrazované informace. Dále chci do aplikace implementovat filtrování mezi festivaly, které však bude mít možnost zvolit více možností pro finální výběr. Časová osa se při analýze jevila jako přehledný způsob prezentovat jednotlivé události ve stejný moment.

[1] Vrchní část obrazovky poskytující uživateli informaci, kde v hierarchii obrazovek se nachází.

4.3 Návrh obrazovek aplikace

V této sekci uvádím návrh obrazovek aplikace společně s jejich vzájemným uspořádáním. Obrazovky detailněji popisuji a vysvětluji výhody uspořádání ovládacích prvků. Vycházím zde z poznatků analyzovaných u podobných existujících aplikací a z rad pro tvorbu uživatelských rozhraní uvedených v příručce [10].

Uspořádání obrazovek

Mojí prvotní myšlenkou je mít jednu hlavní obrazovku, která bude zobrazovat seznam festivalů a z ní tvořit hierarchické uspořádání. Problém nastává při řazení festivalů. Rozdělil jsem ji tudíž na dva samostatné seznamy, které budou mít stejné rozhraní. První z nich bude zobrazovat festivaly, které se teprve budou konat, druhá poté festivaly, které již proběhly. Festivaly chci avšak také filtrovat podle žánrů. Filtr začleněný do každé obrazovky zbytečně duplikuje jeho funkcionalitu, tudíž jsem přidal třetí hlavní obrazovku, sloužící pro filtrování a vyhledávání mezi festivaly. Struktura obrazovek nově kombinuje ploché a hierarchické uspořádání. Všechny tři hlavní obrazovky odkazují na detail festivalu.

Detail festivalu má zobrazovat obecné informace o festivalu, časovou osu a seznam festivalů. Rozhodl jsem se jí rozdělit do tří záložek, na které bude odkazovat `tab bar`, který uživatele informuje, ve které z nich se právě nachází. Ze seznamu festivalů se uživatel dostane na poslední obrazovku zobrazující detailní informace o událostech konaných v rámci festivalu.

Obrazovka *Seznam festivalů*

Jakmile uživatel zapne aplikaci, dostane se na tuto vstupní obrazovku (viz obrázek 4.4). Zde je zobrazeny festivaly, které se teprve budou konat. Seřazené jsou sestupně podle data konání. Každý festival je samostatná buňka zobrazující základní informace, jako název, datum konání, obrázek a část popisu. Buňku jsem se rozhodl vytvořit spíše většího rozměru, aby v ní byl zřetelně čitelný popis a vynikl v ní obrázek, který pořadatel festivalu může do systému přidat. Buňku jsem nejdříve vytvořil obdélníkovou a sahala do samotných krajů obrazovky. Bílá barva v jejím pozadí však dostatečně nerozlišila, kde se nachází samotná buňka a kde pozadí, na kterém se buňky vykreslují. Po přidání efektu stínu bylo možné buňky vizuálně rozlišit. Stín se však zobrazoval pouze na vertikálních okrajích, což působilo dojmem, že uživateli není buňka zobrazená v celé její šířce. Zúžením a zaoblením jejích rohů jsem docílil zobrazení stínu na všech jejích hranách. Díky stínování navíc buňky na uživatele působí, jako kdyby „plavaly“ na jejich pozadí.

V rámci požadavku na jednotnost uživatelského rozhraní aplikace jsem názvu festivalu v buňce nastavil hlavní barvu aplikace, oranžovou. Obrázek festivalu zabírá téměř polovinu celé buňky. Zaoblení v rozích vzhledově doplňuje vnější tvar buňky. Vedle obrázku se nachází datum konání festivalu a počáteční část jeho popisu. pro tyto informace jsem zvolil šedou barvu, aby nenarušily celkový vzhled buňky.

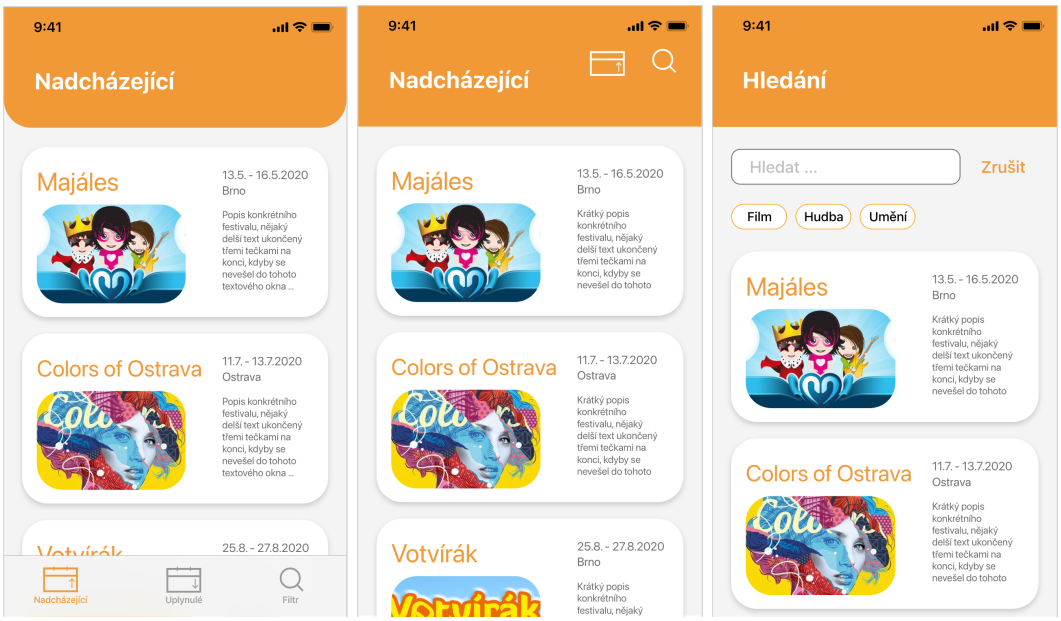
Nejprve jsem chtěl mít pro přepínání mezi nadcházejícími a uplynulými festivaly společně s obrazovkou *Hledání festivalů* `tab bar`. Jeho nevýhodou však bylo zmenšení plochy obrazovky pro vykreslování festivalů a fakt, že oba seznamy vypadají stejně, akorát zobrazují jiné festivaly. Prvky pro přepínání mezi těmito obrazovkami jsem tudíž zařadil do `navigation baru`. Levé tlačítko přepíná mezi seznamem nadcházejících a uplynulých festivalů, což indikuje šipka. Tlačítko lupy přepne uživatele do obrazovky *Hledání festivalů*.

Změna zaoblení rohů `navigation baru` je detailněji popsána u tvorby záložky *Základní informace* (4.3).

Obrazovka *Hledání festivalů*

Tato obrazovka (viz obrázek 4.4) umožňuje uživateli filtrovat festivaly podle žánrů a vyhledávat je podle jejich názvů. Dokáže zároveň tyto dvě možnost zkombinovat a předložit velmi specifický výběr.

Ve vrchní části se nachází `search bar`, do kterého může uživatel psát vyhledávací frázi. Vedle něj se nachází tlačítko *Zrušit*, kterým se dostane zpět na předchozí obrazovku. Níže v záhlaví je umístěn filtr s žánry. Každý žánr je zobrazen jako malá buňka s oranžovým okrajem. při zvolení žánru se buňka vybarví celá oranžovou barvou. Je zde i možnost vybrat více žánrů a použít tak pro vyhledání jejich kombinaci. při zadávání vyhledávací fráze se s každým zadaným znakem nebo zvoleným žánrem aktualizuje seznam festivalů. Využívá stejných buněk pro zobrazení seznamu festivalů jako obrazovka *Seznam festivalů*. Pokud uživatel nezvolí žádný žánr nebo nezadá vyhledávací frázi, žádný festival se mu nezobrazí.



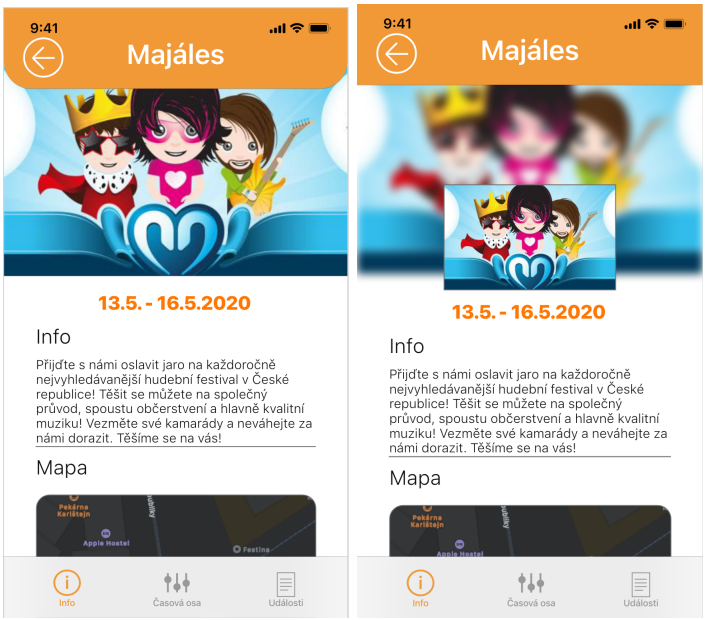
Obrázek 4.4: Původní (vlevo) a nový (uprostřed) návrh obrazovky *Seznam festivalů*, návrh obrazovky *Hledání festivalů* (vpravo).

Obrazovka *Detail festivalu*

Po zvolení konkrétního festivalu z výše popsaných obrazovek se uživatel kliknutím na jeho buňku dostane na tuto obrazovku. Ta má za účel zobrazit podat uživateli detailní informace o festivalu, jako seznam jednotlivých událostí, kompletní popis, místo konání a případný odkaz na webovou stránku. pro lepší přehlednost jsem se rozhodl rozdělit obrazovku do tří záložek, mezi kterými bude uživatel přepínat pomocí `tab baru`.

Záložka *Základní informace*

Po vstupu na obrazovku se uživateli zobrazí tato záložka (viz obrázek 4.5). ve vrchní části se nachází obrázek či logo festivalu, který zabírá celou šířku obrazovky. Část obrázku je překrytá `navigation barem`, aby u jeho zaoblených rohů vyplňovala místo. Obrázek ale nebyl zobrazen celý, proto jsem jeho zobrazení pozměnil. Nechal jsem se inspirovat uživatelským rozhraním v aplikaci *Eventbrite* (viz obrázek 4.1). Nyní je vykreslen v menší velikosti a v jeho pozadí se nachází jeho rozmazaná verze. Odpadla tedy nutnost u navigační lišty vyplňovat volné místo, to vedlo k následnému oddělání oblých rohů ve všech obrazovkách aplikace. Pod obrázkem je vypsáno datum počátku a konce festivalu. Následuje jeho popis a mapa zobrazující lokaci konání, která uživatele po kliknutí přenese do nativní aplikace *Mapy*, která mu dovolí nastavit si do místa konání navigaci.



Obrázek 4.5: Původní (vlevo) a nový (vpravo) návrh obrazovky záložky *Základní informace*.

Záložka *Časová osa*

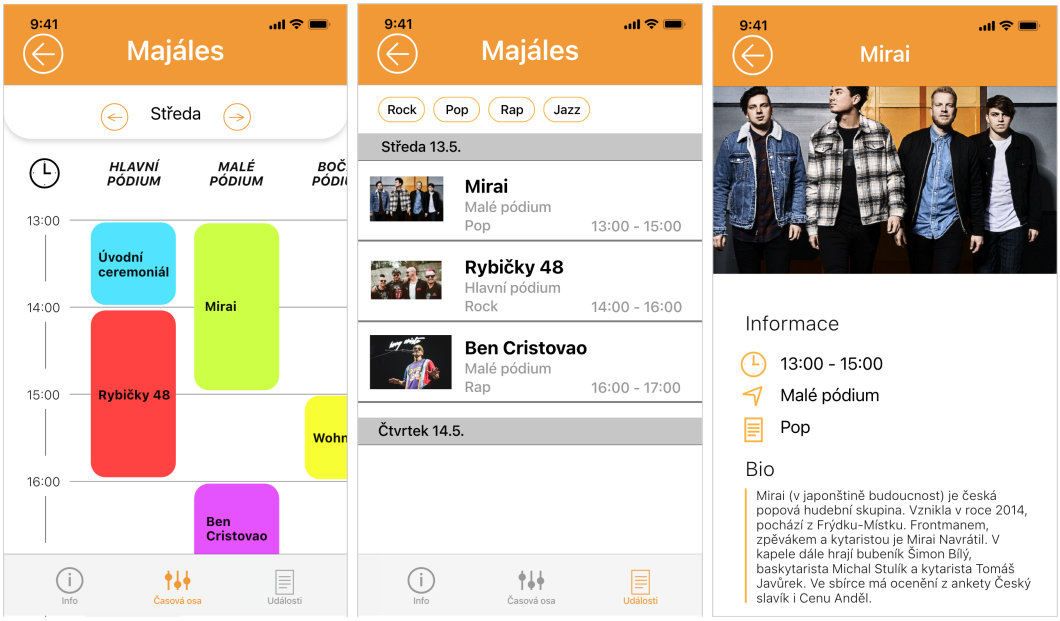
Druhá záložka (viz obrázek 4.6) má za úkol zobrazit uživateli grafický náhled na konané události v jednom dni. Vrchní část obsahuje název aktuálně zvoleného dne. K tomu jsem pro lepší přehlednost později přidal i datum. Z obou stran kolem tohoto popisu jsou tlačítka pro přepínání mezi jednotlivými dny festivalu. při dosažení jednoho z krajních dnů vždy příslušné tlačítko zmizí, uživatel tak zjistí, že se nachází na konci, popřípadě začátku. ve zbylé části obrazovky, vyjma `tab baru`, se nachází samotná časová osa. Vykresluje se vždy pro zvolený den a zobrazuje všechny jeho události. Vertikální osa reprezentuje čas a horizontální osa místa, ve kterých se události konají. Každá událost je samostatná jednoduchá buňka, na které je napsán její název. Barvy buněk jsem zpočátku chtěl mít různorodé. Po přidání žánrů k událostem jsem se rozhodl tuto informaci zobrazit také v rámci této buňky, konkrétně v její barvě.

Záložka *Seznam událostí*

Poslední záložka (viz obrázek 4.6) uživateli vypisuje seznam událostí v rámci festivalu. V záhlaví obrazovky se nachází filtr, ve kterém je možné zvolit více žánrů událostí. Filtr má stejnou podobu buněk s oranžovým okrajem jako na obrazovce *Seznam festivalů*, funkčně se však lehce liší. Pokud zde není zvolen žádný žánr, aplikace zobrazuje namísto prázdného seznamu všechny události nehledě na jejich žánrové zařazení. Pod filtrem jsou zobrazené události, které jsou členěné do dnů a seřazené podle času začátku konání. Každý řádek události obsahuje obrázek, její název, místo a čas konání a její žánr. Název je napsán tučným černým fontem, aby v řádku vynikl, pro zbytek informací jsem zvolil barvu šedou.

Obrazovka *Detail události*

Poslední obrazovka aplikace (viz obrázek 4.6) poskytuje bližší informace o konkrétní události konající se v rámci festivalu. Zpočátku jsem byl rozhodnutý ji nechat co možná nejjednodušší a poskytnout v ní pouze obrázek, na který je zde více místa, proto může být větší, a popis události. Vzhledem k tomu, že jsou ale další informace o událostech vypsány pouze v *Seznamu událostí*, kde jsou navíc šedým, menším fontem, rozhodl jsem se je do této obrazovky taktéž začlenit. Vrchní část tvoří obrázek ve větší velikosti. Pod ním se nachází informace o čase, místě a žánru události. K těmto informacím jsem přiřadil oranžové ikony symbolizující jejich charakteristiku. ve spodní části je popis události, který je uvozen vertikální čarou v oranžové barvě.



Obrázek 4.6: Návrh záložky *Časová osa* (vlevo), záložky *Seznam událostí* (uprostřed) a obrazovky *Detail události* (vpravo).

Kapitola 5

Návrh architektury systému

V následující kapitole pojednávám o procesu a myšlenkových pochodech při tvorbě návrhu systémové architektury. Nejprve se zabývám účelem aplikace. Navazuji rozdělením úloh mezi klientskou a serverovou část a na závěr přikládám návrh množiny entit, které budou v systému figurovat.

Kapitola pojednává o procesu a myšlenkových pochodech při tvorbě návrhu systémové architektury. Nejprve se zabývá účelem aplikace. Navazuje rozdělením úloh mezi klientskou a serverovou část systému a navržením jednotlivých entit. na závěr popisuje výběr implementačních prostředků.

5.1 Účel aplikace

Festivalů, které se v dnešní době konají, je nesmírné množství. S jejich narůstajícím počtem je stále těžší o nich dohledávat informace. Ty z nich, které vznikají nově, vstupují do velice konkurenčního prostředí, ve kterém je obtížné sehnat svoje příznivce. Jedním z účelů aplikace je pomoci těmto menším festivalům se dostat do povědomí veřejnosti. Hlavním účelem je však poskytnout o daném festivalu hlavní informace, jako např. název, místo konání, seznam událostí festivalu a další, aby se na základě nich mohl uživatel rozhodnout, zda by o něj jevil zájem. Výhodou bude dostupnost těchto informací o více festivalech ve strukturované, jednotné formě na jednom místě. Uživatel tak bude moci porovnat festivaly mezi sebou a rozhodnout se, kterého by se chtěl zúčastnit.

V systému budou figurovat dva typy uživatelů, jejichž úlohy se liší. Pořadatel bude využít systém k registrování vlastního festivalu. do něj následně přidá detailnější informace o konaných událostech. Festivalový návštěvník bude používat systém k zobrazení informací ve strukturované formě prostřednictvím mobilní aplikace.

5.2 Úlohy serveru a klienta

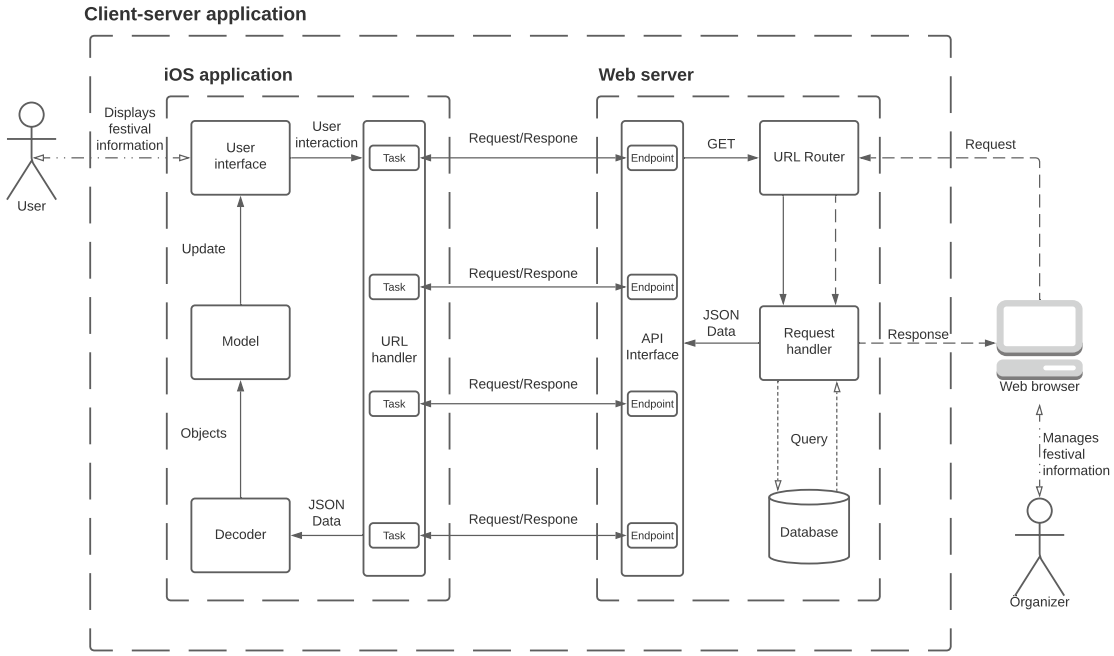
Rozdělení funkcionality mezi serverovou a klientskou část bylo důležité z hlediska celkového fungování aplikace.

V prvotním návrhu jsem přemýšlel nad variantou fungování serveru pouze jako vzdáleného úložiště dat. Tvorba festivalů měla fungovat přímo na klientské straně aplikace společně se zobrazováním strukturovaných dat. Mobilní aplikace by tak sloužila pro organizátory festivalů a uživatele současně. Tuto variantu jsem nakonec zavrhl, jelikož měla několik nevýhod. První nevýhodou bylo nerovnoměrné rozložení úloh mezi dvě zmíněné strany. Aplikace by

se starala o většinové fungování, na druhou stranu by server nenesl žádnou aplikační logiku. Druhou, pro mě rozhodující, nevýhodou byla tvorba nových festivalů přímo v aplikaci. Organizátoři, kteří by si zde chtěli zaregistrovat svoji událost, by si aplikaci museli stáhnout a využít ji. při zadávání objemného množství informací však není mobilní aplikace podle mě nejvhodnější prostředek. Mobilní zařízení mají poměrně malou obrazovku i klávesnici a proto by byl proces vytváření festivalů zdlouhavý. Kromě toho chtějí organizátoři zadávat festivaly, které jsou již plně navržené, tudíž potřebují svoje informace do aplikace pouze přepsat. Předpokládám, že v dnešní době mají tyto informace uložené v elektronické podobě a pro jejich předání aplikaci by je museli hledat buď přímo v mobilním telefonu, ale na jiném místě, nebo na jiném elektronickém zařízení.

Tato myšlenka mě přivedla na rozdělení úloh organizátorů a uživatelů mezi klientskou aplikaci a server jiným způsobem. Mobilní aplikace zůstane pouze pro uživatele a bude jim poskytovat informace o festivalech a serverová část poslouží kromě ukládání dat zároveň k jejich zadávání do systému a spravování prostřednictvím webové aplikace. To rozšíří organizátorům možnosti, jak informace vložit a nebudou na samotnou mobilní aplikaci vázáni. Návrh fungování celého systému lze vidět na obrázku 5.1.

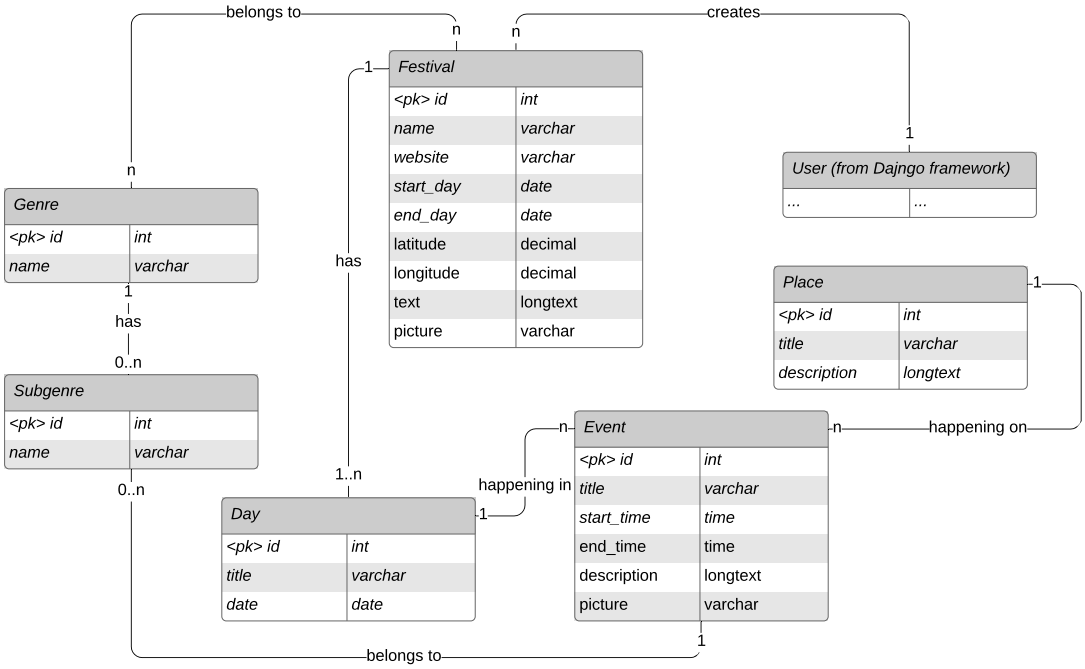
K zajištění komunikace jsem se rozhodl využít REST API rozhraní používající protokol HTTP. Mobilní aplikace bude o informace žádat zasíláním požadavků `GET` na koncové body (tzv. endpointy) API v případě její potřeby. Odpověď, kterou server zašle zpět, bude v serializačním formátu JSON. Odpověď následně bude v aplikaci dekódována do původních objektů a graficky zobrazena uživateli.



Obrázek 5.1: Abstraktní diagram architektury systému.

5.3 Model systému

Před samotnou implementací aplikace je potřeba navrhnout, jaké informace o festivalech se budou ukládat a jaké vztahy tyto informace budou mít mezi sebou. K tomuto jsem využil jazyk `UML`[1] a konceptuální model `ERD`[2]. Diagram zobrazuje jednotlivé entity, které se v systémů nacházejí, vztahy, které mají navzájem mezi sebou a atributy neboli vlastnosti, které dané entity obsahují.



Obrázek 5.2: ER diagram systému.

Na každém festivalu se konají různé akce, rozprostřené na předem daných místech a s odlišnými časy konání. Cílem je tedy vytvořit model, který bude schopen tyto informace navzájem spojovat. Základní entitou systému je `Festival`. Ukládám jeho název, datum začátku a konce, informace o poloze, kde se koná, volitelný odkaz na jeho webovou stránku, popis a obrázek. při návrhu jsem narazil na dvě možnosti modelace událostí. První možností bylo u každé události uložit den jejího konání jako atribut, druhou potom vymodelovat den jako samostatnou entitu. Zvolil jsem druhou možnost, protože den uložený v atributu události by znamenal složitější filtrování v rámci aplikace. Den jako samostatná entita mi umožní hierarchické uspořádání festivalů, dnů a událostí mezi sebou. Další entitou je `Day`. Atributy dne (název a datum) bude systém generovat automaticky, aby se zachovalo jednotné pojmenování dnů v rámci celé aplikace a nedošlo k chybě či překlepu u zadávání data. Systém sám ohlídá, aby nesl daný den datum, které patří do časového horizontu konání festivalu. Třetí základní entitou je `Event`, ten reprezentuje vystoupení, koncert či jinou akci konající se v rámci festivalu. Každá akce bude mít svůj název, čas začátku a konce, krátký popis a obrázek. Akce bude přiřazena ke dnu, ten bude přiřazený ke konkrétnímu festivalu. Tímto způsobem dodržím zmíněné hierarchické uspořádání. Akce se však také musí konat na ně-

[1] Unified Modeling Language – modelovací jazyk určený pro tvorbu vizuálních návrhů systémů.
[2] Entity Relationship Diagram – diagram entit.

jakém místě v rámci festivalu. na hudebních festivalech to mohou být pódia, na filmovém zase kinosály. Kvůli lepšímu lepšímu filtrování jsem zvolil možnost modelovat místo jako samostatnou entitu. Entita `Place` bude mít svůj název a krátký popis, který organizátor využije např. pro zlepšení orientace. Věděl jsem, že nebudu chtít poskytovat uživateli pouze seznam festivalů, ve kterém by se při jejich větším počtu zorientoval velmi těžko. Členěním pro festivaly jsou v první řadě jejich žánry. Zatímco Majáles (4.2) se věnuje hudbě, Zlinfest (4.2) je naopak festivalem filmovým. pro případ, že by chtěl organizátor přidat festival, který by se týkal více žánrů, jsem se rozhodl z něj udělal entitu. `Genre` bude entitou velmi jednoduchou a bude mít pouze svůj název. Jeho hlavní účel však bude zlepšit členění festivalů a možnost jejich filtrování na základě požadavků uživatele. V systému však může být nejen hodně festivalů, ale v rámci nich může být i hodně událostí. Rozhodl jsem se pro zakomponování možnosti filtrování tudíž i pro akce. Zde jsem vytvořil entitu shodnou s `Genre`, která nese akorát odlišný název, `Subgenre`. Výše popsané entity a jejich vzájemné vztahy lze vidět na obrázku 5.2.

5.4 Výběr implementačních prostředků

Posledním krokem před samotnou realizací systému je volba vhodných prostředků k implementaci, kterých je k implementaci mobilní aplikace i webového serveru celá řada.

K vývoji aplikace jsem se rozhodl použít novější knihovnu s novým přístupem k programování, `SwiftUI`. Hlavním důvodem je vyzkoušení tvorby uživatelského rozhraní pomocí deklarativní syntaxe, což je u vývoje mobilních aplikací zcela nový přístup. Chci také zjistit, zda framework poskytuje, přestože je nově vytvořený, stejné možnosti jako `UIKit` a jestli si s ním vystačím na celou realizaci aplikace.

U webového serveru existuje možností značně více. Frameworky pro vývoj informačních systémů existují již desítky let. při výběru jsem dbal na splnitelnost několika požadavků. Cílový framework by měl umět pracovat celým rozložením 3-vrstvé architektury popsané v kapitole 2.2, měl by být napsaný v programovacím jazyku *Python* nebo *PHP*, se kterými mám největší zkušenosti a měl by podporovat návrhový styl MVC. Z výsledků dotazníku nejpoužívanějších webových frameworků [4] požadavkům vyhověly tyto tři: *Django* [23], *Flask* [1] a *Laravel* [2]. Po prostudování jejich oficiálních dokumentací společně se studií v bakalářské práci [25] jsem se rozhodl použít *Django*. Jako jeho hlavní výhody vnímám důraz na pokrytí bezpečnostních rizik, čitelnost společně se členěním zdrojového kódu a přehlednost dokumentace.

Kapitola 6

Implementace

Tato kapitola popisuje implementaci celého systému podle dříve vytvořeného návrhu. Kapitola se dělí na dvě hlavní části: serverovou a klientskou. Každá uvádí, které prostředky byly využity pro jejich implementaci, za využití jakých prvků byla aplikace realizována a další nezbytné součásti souhrnné tvorby. Seznam použitých externích knihoven je dostupný v příloze C.

6.1 Webová aplikace

Tato podkapitola popisuje implementaci serverové části aplikace. Úvodem pojednává o zvoleném webovém frameworku *Django*. Zbylá část je rozdělená na jednotlivé celky, které bylo na serveru potřeba realizovat k jeho funkčnosti. Text vychází zejména z dokumentace frameworku *Django* [23] ke konci doplněnému o využití externího modulu *Django REST Framework* [21]. K účelům testování mi můj vedoucí práce poskytl školní server. Ukázka stránek webového systému je dostupná v příloze B.

6.1.1 Django

Django je webový framework vytvořený v programovacím jazyce *Python*. Má vysokou míru abstrakce a podporuje rychlý vývoj webových aplikací s čistým designem za použití méně zdrojového kódu. Jedná se o tzv. *full-stack* framework, což znamená, že obsahuje rozhraní starající se o veškeré části implementace včetně webového uživatelského rozhraní a správy uložených dat. Využívá návrhový styl MVC, ve kterém ale část*view* není zodpovědná za to, jak se data zobrazí uživateli, ale která se mu zobrazí. za část *controller*, která se v běžném případě stará o aplikační logiku, tvůrci považují framework jako takový. Tento styl nazývají MTV (model, template, view), ve kterém způsob zobrazení dat zastupuje *template*. Význam části *model* zůstává stejný. Příkladem základních filozofií, na kterých Django staví, jsou:

- **Loose coupling** – Přestože je Django *full-stack* frameworkem, jednotlivé jeho části jsou na sobě nezávislé.
- **Less code** – Vyvinuté webové aplikace by měly obsahovat co nejméně zdrojového kódu, za využití dynamiky Pythonu.
- **Quick development** – Cílem frameworku je maximálně zrychlit proces vývoje.
- **Dont repeat yourself (DRY)** – Každá část informace by se měla v aplikace nacházet pouze na jednom místě.

- **Consistency** – Framework by měl být konzistentní ve všech úrovních, od stylu psaní jazykem Python po samotný „zážitek“ z využití Djanga.

6.1.2 Správa dat

Jak je stanoveno v podkapitole 5.2, jednou z hlavních úloh serveru je ukládání informací o jednotlivých festivalech. K tomuto účelu nabízí *Django* abstraktní vrstvu *Models*. Framework vývojářům poskytuje aplikační rozhraní pro přístup do databáze. Modely, které vývojář v aplikaci vytvoří, *Django* zpracuje a vytvoří příslušné SQL příkazy nutné pro vytvoření či změnu tabulek do databáze. Tuto skutečnost nazýváme objektově-relační mapování (zkr. ORM). Každý model se tvoří jako samostatná třída, která je potomkem třídy `django.db.models.Model`. Její atributy slouží k reprezentaci jednotlivých polí tabulky. Každý takový atribut musí být instance třídy `Field`, aby tak mohl být jednoznačně určen typ záznamu. Příkladem typů polí mohou být:

- **CharField** – Pole pro uložení textového řetězce.
- **DateField** – Pole pro uložení data.
- **FileField** – Pole pro uložení souboru.
- **IntegerField** – Pole pro uložení celého čísla.

Každé takové pole navíc může obsahovat určité vlastnosti. Ty mohou být buď specifické pro určité pole (např. vlastnost `max_length`, která udává maximální délku textového pole a je navíc povinná) nebo jsou obecné a jdou aplikovat ke všem druhům polí (např. vlastnost `null`, která může dovolit uživatelům uložit prázdnou hodnotu do databáze jako `null`).

Pro přístup k již uloženým informacím se v každém *modelu* nachází rozhraní zvané `Manager`. Nad ním můžeme volat jeho metody, které nám vrátí uložené záznamy v datovém typu `QuerySet`. Příkladem metod volání jsou *all()*, která vrací všechny položky tabulky nebo *filter()*, která podle zadaného parametru dokáže mezi záznamy vybrat pouze ty, které splňují dané podmínky. V modelu se metody také dají vytvořit. Zatímco metody volané nad rozhraním `Manager` mají za úkol pracovat s celou tabulkou, metody modelu slouží k přidání funkcionality jednoho záznamu. Poskytují tak možnost udržet logiku práce s modelem na místě jeho definice.

6.1.3 Zpracování požadavků

Nejdůležitější částí webové aplikace je schopnost zpracovávat příchozí požadavky od uživatelů a rozhodnout, jak na ně odpovědět. Dříve než bude zmíněno, jak *Django* v tomto případě postupuje, je potřeba popsat jednotlivé moduly, které se na zpracovávání podílí.

URL Dispatcher

Jedná se o modul, který se stará o rozpoznávání URL adres. pro tvorbu adres do aplikace je nutno vytvořit modul s názvem `URLConf`. Jeho úkolem je mapovat jednotlivé adresy k funkcím, které se starají o poskytování příslušných odpovědí. Cesty se zapisují do proměnné *urlpatterns* jako funkce `django.urls.path()` nebo `django.urls.re_path()`. Obě obsahují parametry *route*, *view*, *kwargs* a *name*. První parametr je využit k přidání textového tvaru adresy. *Django* zde dovolí si část adresy zapamatovat a poslat ji do namapované

funkce jako argument s konkrétním názvem. pro tuto skutečnost je potřeba zvolenou část adresy vložit mezi špičaté závorky a pro vlastní potřeby lze specifikovat i typ dat, které se předávají. Rozdílem mezi dvěma zmíněnými funkcemi je forma adresy. Zatímco *path()* očekává zadanou adresu jako textový řetězec, *re-path()* očekává regulární výraz. Parametr *view* odkazuje na funkci, která zaslanou žádost na server obslouží. Může být zapsána jako funkce nebo třída. pro lepší členění kódu naší aplikace zde *Django* dovolí zavolat funcki *include()*, která zde dosadí jinde vytvořený `URLConf` modul. Tvary adres tak mohou být rozdělené na logické celky. Parametr *kwargs* přidává vývojáři možnost předat další argumenty do volání obslužné metody a parametr *name* slouží jako identifikace konkrétní URL adresy pro možnost se na ní odkazovat. Pokud existují dvě mapování se stejnou adresou pro zpracování, ale rozdílnými funkcemi pro zpracování, *Django* vezme tu, na kterou při hledání narazí první. Jejich pořadí tudíž v aplikaci hraje roli.

Obslužné funkce

Každý požadavek, který projde `URL Dispatcherem`, se dostane do funkce, která ho vyhodnotí, zvané `View`. ve funkci je dostupný v proměnné *request*, která obsahuje všechny data, která byla s požadavkem zaslána, jako například položky hlavičky, `GET` nebo `POST` parametry, kódování a další. Tyto informace funkce vyhodnotí a rozhodne, jakou odpověď poslat. Odpověď může být různého typu, od HTML dokumentu, přesměrování až po obrázek, XML dokument, obrázek či další. V rámci těchto funkcí lze přistupovat k databázi způsobem popsaným v 6.1.2, číst z ní záznamy, upravovat je, mazat či vytvářet nové.

Alternativou k obslužným funkcím jsou třídy. Jejich úkolem není funkce nahradit, ale zefektivnit zpracovávání požadavků a udělat obsluhu modulární. Řeší hlavní nedostatky, kterými jsou organizace kódu, kdy ve třídách lze například požadavek `GET` a `POST` zpracovat pomocí separátních metod místo podmíněného větvení, a přidává možnost využití objektově-orientovaného paradigmatu. Třídy od sebe mohou dědit a stávají se znovupoužitelnými.

6.1.4 Šablony

Důležitou součástí frameworku je způsob tvorby dynamických HTML stránek. *Django* k tomuto účelu nabízí více možností. První variantou je vestavěný šablonovací systém využívající jazyk zvaný *Django template language*. Alternativou k této možnosti je využití systémů třetích stran, popřípadě tvorba vlastního backendu. Jeho implementace spočívá ve vytvoření třídy dědící z `django.template.backends.base.BaseEngine`, která musí povinně obsahovat metodu *get_template()*. Její úkol je správný výběr šablony.

Šablonou ve vestavěném systému je HTML dokument sloučený s řetězci psanými v jazyce *Django template language*, vycházejícího z jazyka *Python*, které jsou označené řídícími znaky. Tyto části šablony se před jejich vykreslením zpracují šablonovacím enginem, který je nahradí za příslušné hodnoty, popřípadě vyhodnotí vepsané řídicí struktury. Po zpracování se výsledný dokument odešle jako odpověď na požadavek. do šablony lze zaslat různé informace potřebné k zobrazení pomocí proměnné *context*, do které se uloží data v párech klíč-hodnota. Šablonovací jazyk rozeznává dohromady čtyři druhy zápisů.

Proměnné slouží k vypsání jejich hodnot. do šablony jsou zasílány pomocí *contextu* a jsou obklopeny řídícími znaky `{{` a `}}`. při zaslání složitějších datových typů, jako jsou slovníky, objekty či seznamy, se k jednotlivým jejich položkám či atributům přistupuje pomocí tečkové notace.

Tagy poskytují možnost zapsání řídících struktur do samotné šablony k pozdějšímu vykreslování. Každá konstrukce je obklopena řídicími znaky `{%` a `%}`. Samozřejmostí jsou základní řídicí prvky jako podmíněné větvení `if-else` nebo cyklus `for`. Dále *Django* poskytuje sadu vestavěných tagů pro různé další použití, jako například kombinování více šablon, vytváření hypertextových odkazů a další. pro potřeby vývojáře je přidána možnost tvorby vlastních tagů.

Filtry přidávají schopnost úpravy proměnných vložených v šabloně. Připisují se přímo za samotnou proměnnou a jsou od ní odděleny znakem `|`. *Django* zde opět poskytuje základní sadu předpřipravených možností, které zvládnou naformátovat datum a čas, upravovat hodnoty proměnných v *contextu* a mnohé další. Stejně jako u tagů, i zde mají vývojáři možnost vlastní tvorby.

Komentáře slouží pro zapsání textu či poznámek, které se ve výsledném dokumentu neobjeví.

6.1.5 Formuláře

Formuláře jsou potřebnou součástí komunikace uživatelů se serverem. Práce s nimi obsahuje několik samostatných částí, od tvorby formuláře po jeho zpracování. *Django* má přesně pro tento případ dostupný modul `Forms`, který slouží k automatizaci procesu jejich použití. Stará se o tři samostatné části, kterými jsou příprava dat k vykreslování, vytváření HTML formulářů pro tyto data a na závěr jejich zpracovávání.

Na začátek se formuláře, které v rámci webového serveru budou používány, musí vytvořit. Každý formulář reprezentuje samostatně vytvořená třída, která dědí z `forms.Form`. Jeho pole zastupují jednotlivé třídní atributy, které svoje datové typy berou z `forms`. Velká část typů polí je shodná s těmi v modelech, popsaných v podkapitole 6.1.2. Ke každému z nich lze využít různé druhy validátorů, které jsou schopny omezovat vložené hodnoty a kontrolovat tak jejich rozsah, délku a další vlastnosti. Formuláře mohou být však vytvořeny za použití modelů. Takový formulář dědí z `forms.ModelForm`. V jeho vnořené třídě `Meta` se specifikuje do proměnné *model*, který model má být pro něj předlohou a do proměnné *fields* se zadají konkrétní pole modelu, které se mají ve formuláři použít. *Django* automaticky zajistí, aby HTML prvky odpovídali příslušným polím.

Formulář se vytváří v rámci `Views` instanciací příslušné třídy. Dále se v *contextu* pošle do šablony, aby mohl být vyrenderován. V šabloně dostává vývojář více možností, jak bude formulář vypadat. Pole se mohou vypsat jednotlivě za použití tečkové notace. Dalším způsobem je iterace přes jednotlivé atributy formuláře. Třetí možností je využití vestavěných metod, které umí formuláře převést na tabulku, samostatné odstavce nebo seznam. Zápis formuláře v šabloně do tabulky vypadá následovně: `{{ form.as_table }}`, kde *form* je název proměnné s formulářem. *Django* zde přidává možnost použít ochranu proti CSRF[1] útokům. V takovém případě stačí do šablony na začátek formuláře přidat tag `{% csrf_token %}`, který slouží jako podpis formuláře.

Zpracování probíhá, stejně jako jeho tvorba, ve `Views`. V obslužné funkci se vytvoří nová instance formuláře s tím rozdílem, že jako jeho jediný parametr použijeme zaslaný požadavek `POST`. Proběhne kontrola, zda v něm není žádná chyba, pomocí funkce *is_valid()*. při úspěchu jde přistoupit k datům a zpracovat je. Výhodu zde dostávají modelové formuláře, ze kterých se data nemusí vytahovat, ale pomocí funkce *save()* lze celý formulář uložit

[1] Cross Site Request Forgery – podvržení požadavku na webové stránce pomocí cizích přihlašovacích údajů.

jako nový řádek do tabulky v databázi, která odpovídá danému modelu. při neúspěchu se k jednotlivým atributům uloží popis vzniklých chyb a je možno formulář opět zaslat do šablony, která chyby může vypsat uživateli. Níže je přiložena ukázka kódu pro uložení nového místa v rámci festivalu.

```
def add_place(request, festival_id):

    if request.method == 'GET':
        form = AddPlaceForm()
        return render(request, 'festivaly/addplace.html',
                      context={'form':form, 'add':True})
    else:
        form = AddPlaceForm(request.POST)
        if form.is_valid():
            place = form.save(commit=False)
            place.festival_id = festival_id
            place.save()
            return redirect('/festival/' + str(festival_id) + '/')
        else:
            return render(request, 'festivaly/addplace.html',
                          context={'form':form, 'add':True})
```

Výpis 6.1: Implementace zpracování formuláře pro přidání místa k festivalu.

6.1.6 Autentizace

Pro zajištění registrace a přihlašování uživatelů má *Django* vlastní vestavěný systém. Ten se stará o uživatelské účty, skupiny, práva a další. Hesla uživatelů ukládá v *hashované* formě v databázi, chrání je tak proti odcizení. Systém se nachází v modulu `django.contrib.auth`. pro autentizaci uživatele se použije funkce *authenticate()*, která vezme do parametrů uživatelské jméno a heslo, porovná ho se systémem a v případě úspěchu vrátí objekt `User`, ve kterém jsou dostupné ostatní informace o účtu. Přihlášení uživatele zajistí funkce *login()*. Po dobu přihlášení lze informace o účtu nalézt v `request.user`. Odhlášení uživatele se provede funkcí *logout()*.

6.1.7 API endpointy

Komunikace mezi klientem se provádí zpravidla pomocí aplikačně programového prostředí. Samotné *Django* však modul k tomuto určený neobsahuje. pro tyto účely jsem se rozhodl využít *Django REST framework*. Jedná se o samostatně vyvinutý modul určený pro implementaci serializace dat a zprostředkování komunikace. Vývojářům umožňuje využití spousty modulů, od autentizace při požadavku na data po zajištění filtrace a stránkování. Velkou výhodou je možnost implementace obslužných funkcí v rámci samotných `Views` s použitím dekorátoru *@api_view()*, který změní chování obslužné funkce a dovolí ji využít výhod tohoto frameworku. Jedinou povinnou součástí využití této funkce je volba způsobu serializace dat.

Implementace serializátoru probíhá obdobně jako u formulářů. při využití třídy `serializers.Serializer` je potřeba v rámci atributů specifikovat každou informaci včetně typu pole zvlášť. Alternativou a zkratkou k tomuto přístupu je třída

`serializers.ModelSerializer`, která sama od příslušného modelu odvodí a vytvoří všechna pole. Model a pole, která chce vývojář serializovat se uvedou v podtřídě *Meta* v rámci proměnných *model* a *fields*.

Pomocí výše zmíněných obslužných funkcí bylo dosáhnuto velmi rychlého způsobu pro sdílení serializovaných dat. Limitem však zůstává nemožnost využití ostatní výhod frameworku. Druhým způsobem, který již není omezený, je implementace tříd namísto funkcí. Ty musí dědit z jedné ze tříd obsažených v `rest_framework.generics`. Každá z nich je určena pro komunikaci za pomocí různých typů požadavků. Příkladem může být třída pracující s požadavkem `GET`, která pracuje s více instancemi modelu, `ListAPIView`. Seznam následujících atributů ovládá chování třídy:

- **queryset** – Všechny instance modelu, které mají být v odpovědi.
- **serializer_class** – Třída určující způsob serializace.
- **lookup_field** – Volba pole, které se použije pro hledání jednotlivých instancí modelu.
- **lookup_url_kwarg** – URL argument určený k hledání.
- **pagintaion_class** – Třída, podle které bude uplatněno stránkování.
- **filter_backends** – Seznam filtračních tříd pro úpravu *querysetu*.

Cesta	Popis
api/festival_list_upcoming/	Nadcházející festivaly seřazené od nejbližšího
api/festival_list_past/	Uplynulé festivaly seřazené od posledně konaného
api/festival/search/	Vyhledávání podle názvu a filtrace podle žánrů
api/festival/<festival_id>/	Základní informace o festivalu
api/festival/<festival_id>/days/	Seznam dnů a jejich událostí festivalu
api/festival/<festival_id>/places/	Seznam míst festivalu
api/festival/<festival_id>/subgenres/	Seznam žánrů událostí festivalu
api/genre/<genre_id>/	Seznam festivalů žánru
api/genre_list/	Seznam všech žánrů pro festivaly

Tabulka 6.1: Seznam koncových bodů API. Výraz *<festival_id>* a *<genre_id>* vyznačují identifikátor festivalu a žánru.

Všechny koncové body ve webové aplikaci komunikují prostřednictvím požadavku `GET` a slouží k poskytnutí informací ve formátu JSON pro stažení do klientské aplikace. Vyhledávání je jediné, které obsahuje dodatečně dva parametry, *query* pro zadání řetězce k vyhledání dle názvu a *genre* pro specifikaci jednoho či více festivalových žánrů.

6.2 Mobilní aplikace

V této podkapitole je popsán vývoj klientské aplikace na platformu iOS. pro implementaci byl vybrán framework `SwiftUI`. Hlavním důvodem této volby byla možnost vyzkoušet si nový, deklarativní styl tvorby uživatelského rozhraní. Tvorba aplikace probíhala v nativním vývojářském prostředí Xcode (3.4) a průběžné testování umožnil vestavěný iOS emulátor. Podkapitola vychází převážně z oficiální dokumentace `SwiftUI` [13] a `UIKitu` [14]. Rozdělena je na dílčí implementační části potřebné k celkové realizaci. Obrazovky implementované aplikace jsou dostupné v příloze A.

6.2.1 Ovládací a zobrazovací prvky

K vytvoření základních vizuálních bloků uživatelského rozhraní aplikace poskytuje `SwiftUI` sadu prvků. Každý z nich je datového typu `struct` (struktura) a implementuje protokol `View`. Zobrazovací prvky mají za úkol vyplnit obrazovku relevantními informacemi. Slouží pro zobrazení textu, obrázků, různých tvarů a dalších elementů. Ovládací prvky přidávají uživateli možnost s aplikací interagovat. Pomocí nich je aplikace schopna zpracovat vstup a vyrenderovat odpovídající výstup. do obrazovky se každý prvek přidá vytvořením jeho instance s možnými vstupními parametry. Příklady prvků jsou následující:

- **Text** – Napsaný text pro čtení.
- **TextField** – Textové pole pro uživatelský vstup.
- **Image** – Zobrazovací prvek pro obrázek.
- **Menu** – Ovládací prvek pro zobrazení volby akcí.
- **Toggle** – Přepínač mezi stavy zapnuto a vypnuto.
- **Picker** – Prvek pro výběr hodnoty z různých možností.

Všechny elementy, které na obrazovce reálně uvidíme, patří do této sekce. při tvorbě aplikace je žádoucí, abychom mohli prvky modifikovat. Každý text v aplikaci nebude mít stejnou velikost a barvu, nadpisy může vývojář chtít zobrazit třeba tučně a obrázky můžou mít různou velikost a tvar. K aplikování těchto a spousty dalších vlastností slouží *ViewModifiers*. Tyto modifikátory mohou být užity k více účelům:

- úprava vzhledu jak zobrazovacích, tak ovládacích prvků;
- reakce na události, například po zobrazení prvku na obrazovce;
- podmíněné zobrazování speciální části obrazovek;
- konfigurace hierarchických elementů.

Aplikování modifikátorů je následující. za konkrétní prvek v naší obrazovce pomocí tečkové notace se napíše daný modifikátor, stejně jako by se volala třídní metoda na její instanci. do závorek přijdou případné parametry. Jako příklad je uveden zápis kódu pro aplikování zelené barvy na text.

```
Text("Custom text")
    .foregroundColor(Color.green)
```

Výpis 6.2: Způsob aplikace modifikátoru pro zelené zabarvení textu.

ViewModifiers mohou navzájem na sebe navazovat. na každý prvek je tedy možno uplatnit více úprav. Záleží však na pořadí, ve kterém jsou zapsány. Vyhodnocení každého modifikátoru probíhá zvlášť a vždy se vrátí nový, upravený prvek, na který se poté aplikuje změna další. Tímto způsobem lze dosáhnout velmi pestrého zobrazení i jednoduchých elementů. Ne všechny však jdou uplatnit kdekoliv. Některé z nich jsou společné pro všechny prvky, jako například *.frame(width:height:alignment:)*, který určuje velikost a zarovnání. Ostatní však mohou být použity pouze pro konkrétní sadu či dokonce jenom jeden element.

6.2.2 Rozložení prvků na obrazovce a hierarchie

Přidávání prvků do obrazovky je první část tvorby uživatelského rozhraní. Nadále je potřeba zařídit, aby byl každý prvek na svém místě, což zařizuje rozložení. Jelikož `SwiftUI` využívá deklarativní syntaxi, aplikuje se rozložení opět pomocí elementů. Ty ovlivňují pořadí, způsob rozložení a další vlastnosti. Obrazovku lze tvořit i dynamických způsobem z přiložených dat. V kombinaci se statickými prvky vzniká výsledné uživatelské rozhraní. Jako příklad je uvedeno pár z nich:

- **HStack** – Horizontální rozložení vložených prvků.
- **VStack** – Vertikální rozložení vložených prvků.
- **ZStack** – Rozložení, které vložené prvky skládá na sebe.
- **ScrollView** – Vlastnost obrazovky posouvat její obsah nahoru a dolů.
- **Spacer** – Prvek, který vytlačí sousední prvky v rozložení o maximální možnou šířku či výšku.
- **ForEach** – Implementovaná forma cyklu, která dynamicky vytváří další elementy z připravených dat.

Fungování rozložení spočívá v zanořování všech rozkládacích, zobrazovacích i ovládacích prvků do sebe. Výsledkem je ucelená obrazovka. Pokud má být obrazovek v aplikaci více, musí se krom jejich tvorby zajistit také jejich návaznost a způsob přepínání, neboli hierarchie. Základem pro tyto vztahy obrazovek je *NavigationView*, které vytváří efekt pohybu mezi obrazovkami směrem dopředu a dozadu. Uživatelům platformy iOS je velice známé a přirozené, jelikož je použito ve většině systémových aplikací. Vizuálně se nachází ve vrchní části obrazovky s nadpisem udávajícím, v jaké části aplikace se uživatel právě nachází. Přechod na další obrazovku je spojen s animací jejího přejetí zprava doleva a je implementován pomocí ovládacího prvku *NavigationLink*. Návrat na obrazovku zpět je zpravidla pomocí tlačítka umístěného v levé části navigační lišty. Animace návratu je přejezd rodičovské obrazovky z druhé strany. Dalším typickým příkladem je *TabView*, rozložení umožňující rozdělit obrazovku do více záložek a přepínání mezi nimi pomocí spodní lišty.

Prvkům rozložení se dají vlastnosti měnit též. I zde mají některé z nich své vlastní specifické modifikátory, avšak když zde použijeme takovou modifikaci, která nemá na daný element přímý vliv (např. změna fontu na `VStacku`), vlastnost se nasdílí do všech vnořených *views*. Všechny prvky, které leží uvnitř a mají zobrazovat text, budou mít tuto vlastnost. Lokálně ji pak můžeme u libovolného textu přepsat použitím stejného modifikátoru.

6.2.3 Vazba proměnných na uživatelské rozhraní

Další vlastností frameworku `SwiftUI` je schopnost automaticky upravovat uživatelské rozhraní. Děje se tak za pomocí proměnných zabalených v *property wrapperech*. Změna jejich hodnot způsobí instantní překreslení ovlivněných částí obrazovky. Datům také umožňuje sdílení napříč více obrazovkami, což vede redukci redundance informací napříč aplikací. Vývojář má možnost vybrat si z několika možností podle potřeby implementace.

State variable (stavová proměnná) je základní možností vazby dat na uživatelské rozhraní. Používá se jako lokální proměnná, proto by se k ní mělo přistupovat pouze ze

samotné struktury reprezentující obrazovku. Zapisuje se uvozením *property wrapperu* `@State` před deklaraci dané proměnné. Lze sdílet do dalších obrazovek pomocí prefixu *$*, kdy slouží jako *Binding*.

ObservedObject se užívá v případě, když jsou data uložená na externím místě v aplikaci. Sledovaná třída či struktura musí implementovat protokol `ObservableObject`. Konkrétní proměnné ovlivňující změnu uživatelského rozhraní musí být zabalené v *property wrapperu* `@Published`. `ObservedObject` se využije u proměnné přímo uložené ve struktuře konkrétního *view*.

Binding propojuje dvoucestně data napříč obrazovkami. Takto sdílené data lze upravovat na více místech a ovlivní vizuální změnu všude, kde jsou propojená s prvky uživatelského rozhraní. Před název proměnné se zapíše `@Binding` a do nové instance tohoto *view* se v parametru pošlou data reprezentující stav.

Environment se využije při nastavování globálních vlastností pro určité *view*. Přistupuje se k němu pomocí zápisu `@Environment` před deklaraci proměnné. za *property wrapper* se do jako parametr zadá klíč, pod kterým je hodnota v prostředí uložená.

Pracovat s tokem dat v aplikaci jde i dalšími způsoby, které zde nejsou uvedeny. Zahrnují možnost posílat hodnoty v hierarchii obrazovek směrem nahoru (`PreferenceKey`) nebo spravovat persistentně uložená data pomocí frameworku `CoreData` (`FetchRequest`). Pomocí tohoto způsobu je implementována aplikační logika ovlivňující zobrazení informací koncovému uživateli.

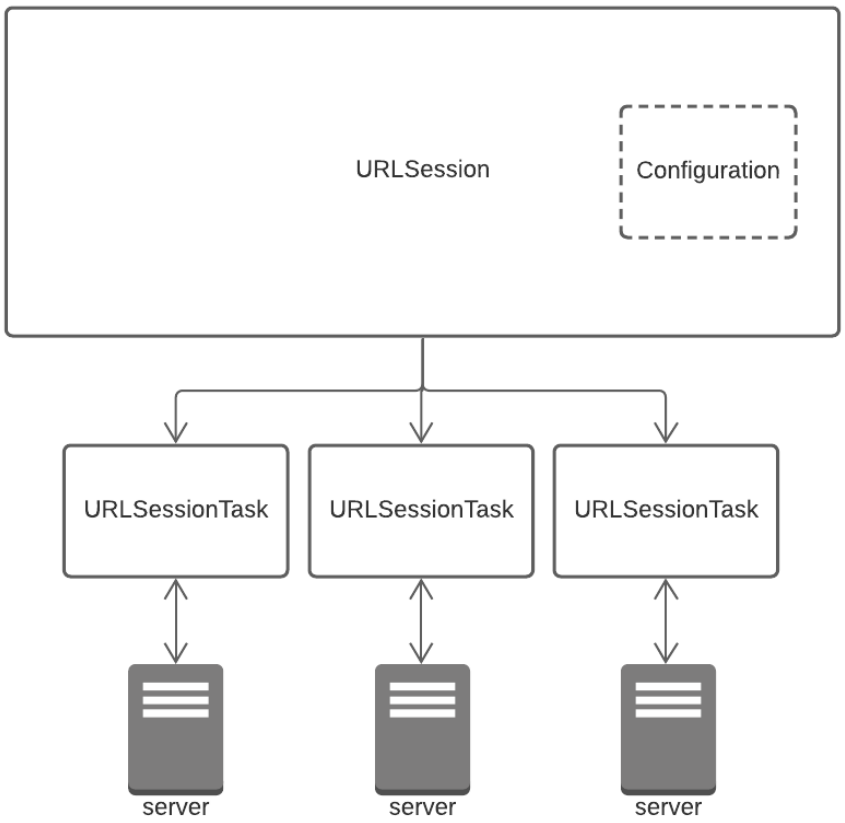
6.2.4 Stahování dat

Mobilní aplikace jako taková je závislá na datech uložených na serveru, proto je potřeba zajistit, aby mezi nimi probíhala komunikace. K těmto účelům je určená třída `URLSession`. Poskytuje rozhraní pro stahování a nahrávání dat na koncové body serverů. Umožňuje také provádět komunikaci v případě, že je aplikace na pozadí či dokonce není spuštěná. na její instanci lze vytvořit více požadavků reprezentovanými třídou `URLSessionTask`, které přímo komunikují se serverem. Existují čtyři typy:

- **Data tasks** – Vytváření jednoduchých, krátkých požadavků.
- **Upload tasks** – Nahrávání obyčejných dat či souboru a možnost práce na pozadí aplikace.
- **Download tasks** – Stejné jako *Upload tasks*, ale slouží ke stahování.
- **WebSocket** – Výměna zpráv pomocí protokolů *TCP* a *TLS*.

Každá *session* sdílí pro svoje *tasky* delegáta, který se stará o správu informací při specifických událostech, jako například selhání autentizace či dokončení stahování dat ze serveru. Přidělení delegáta není povinné a můžeme ho při tvorbě sezení vynechat poskytnutím hodnoty `nil`. Pokud je potřeba použití složitější konfigurace, využije se objekt `URLSessionConfiguration`, který může ovlivnit například práci s *cookies* nebo ukládáním dat pomocí *cache*. Diagram komunikace je vidět v obrázku 6.1. Pokud není potřeba vytvořit spojení se specifickou konfigurací, poslouží pro základní účely objekt `shared`, který se automaticky inicializuje ve vytvořené instanci `URLSession`. Použití tohoto rozhraní je

plně asynchronní, předchází se tak problémům typu zaseknutí aplikace při špatném internetovém připojení či při načítání velkého objemu dat. Stahování i nahrávání probíhá na pozadí aplikace a poskytuje prostředky ke zjištění, jestli se činnost již ukončila či nikoliv.



Obrázek 6.1: Diagram zprostředkování síťové komunikace v iOS.

Po oznámení stažení příslušných dat je na řadě jejich zpracování a zobrazení uživateli. Vždy, když přijde odpověď na zaslaný požadavek, jsou dvě možnosti, jak ji zpracovat. První je zavolání metod delegáta, které se postarají o přesun dat na správné místo v aplikaci, druhou je implementace tzv. *completion handleru*, bloku kódu, který se provede ihned po dokončení přenosu dat. Nehledě na volbě způsobu se tak musí dít v rámci hlavního vlákna, které jako jediné umožňuje provádění změn v uživatelském rozhraní. Data se asynchronně podle potřeby nejprve dekódují či jinak interpretují a následně uloží do příslušných proměnných pracujících se stavem aplikace, které při změně uživatelské rozhraní překreslí.

Serializační formát JSON

Při komunikaci v architektuře klient-server se data odesílají v serializovaném formátu, který usnadňuje proces tvorby dat zpět do příslušných datových struktur, zvaný deserializace. při stažení dat v JSON formátu se nejprve musí dekódovat a teprve následně s nimi jde pracovat. K tomuto účelu slouží třída `JSONDecoder`, která deserializuje data do připravených datových struktur. Každá taková struktura musí přesně odpovídat zaslanému JSON objektu, co se týče názvů proměnných a jejich datových typů, v opačném případě skončí

dekódování chybou, zároveň musí implementovat protokol `Codable`. Deserializace se spustí zavoláním funkce *decode()*, které se do parametrů uvedou data, ze kterých bude dekódování provedeno a formát struktury, který jim odpovídá.

6.2.5 Propojení UIKitu do SwiftUI

U implementace záložky *Časová osa* (4.3) nastal problém. Nejdůležitější vlastností této části obrazovky je schopnost posouvat ji horizontálně i vertikálně zároveň se zachováním pevného prvního řádku a sloupce, které informují uživatele, v jaký čas a na jakém místě se událost koná. První sloupec, obsahující časové údaje, tak musí být posouvatelný pouze ve vertikálním směru, aby údaje nemizely z obrazu. Stejně to platí o prvním řádku ve směru horizontálním. `SwiftUI` dovoluje vytvořit *ScrollView*, které zajistí posun do obou směrů, ale nelze v něm zajistit ukotvení jeho částí. Z tohoto důvodu jsem se rozhodl využít externí framework *SpreadsheetView*, který je napsán v jazyku *Swift* a využívá `UIKit`. Samotná implementace vychází z *UICollectionView*, které slouží pro zobrazení sady dat a jejich prezentaci pomocí nastavitelného rozložení.

Propojení prvků `UIKitu` do `SwiftUI` se provádí přes *UIViewRepresentable*. Jedná se o *wrapper*, do kterého se vloží instance *UIView*, což je třída, ze které dědí všechny zobrazovací prvky v `UIKitu`. Nahrazuje tak *ViewController*, který se normálně *view* spravuje. Zajišťuje vytvoření, reagování na změny a destrukci. Metoda *makeUIView()* zajišťuje jeho inicializaci a základní nastavení. Jelikož v `UIKitu` není možnost využití stavu pro automatické provedení změn na obrazovce, musí se dále implementovat *Coordinator*, který se stará o obousměrnou komunikaci mezi těmito frameworky.

Kapitola 7

Testování klientské aplikace

Kapitola popisuje testování, které bylo provedeno po dokončení implementace aplikace v rozsahu podle předešlého návrhu. Úvodem pojednává o účelu a způsobu samotného testování, po kterém popisuje jeho průběh. Nadále předkládá analýzu výsledků, ohlasů vybraných uživatelů a na závěr je testování shrnuto a je předložen návrh na další pokračování ve vývoji aplikace.

7.1 Účel a způsob testování

Závěrečné testování mělo za úkol zjistit stav přehlednosti aplikace při jejím prvotním použití. Mělo tak zhodnotit, jak intuitivní je uživatelské rozhraní, jestli na uživatele působí vzhledově dobrým dojmem a zdali lze porozumět všem ovládacím prvkům a možnostem aplikace. Testovaná skupina uživatelů byla zvolena následovně. Každý jedinec musel vlastnit mobilní zařízení iPhone alespoň jeden rok a tudíž chápat obecné principy ovládání aplikací v systému iOS, musel se účastnit více než tří festivalů za posledních deset let a používal mobilní aplikace na denní bázi. Testovaných osob bylo dohromady deset, přičemž všechny byly ve věkovém rozmezí 20 až 40 let, průměrně okolo 26 let.

7.2 Průběh testování

Každá osoba podstupovala testování samostatně, aby nedošlo k vzájemnému ovlivňování výsledků. na začátku bylo jedinci sděleno, k čemu aplikace slouží společně s informacemi o průběhu testu. Každý dostal jednu minutu na první průchod aplikací. Součástí testování bylo plnění úkolů dle testujícího, odpovídání na předem připravené dotazy a závěrečná diskuze, prostor pro připomínky, nápady a kladné či záporné hodnocení jednotlivých částí aplikace. Sada úkolů a otázek je dostupná v příloze D. Zadání každého úkolu mohlo být testované osobě zopakováno na základě její žádosti, ale kromě toho nebyly poskytovány žádné jiné nápovědy. K dalšímu úkolu se přecházelo až po dokončení úkolu aktuálního sdělením odpovědi, která byla pro všechny z nich jednoznačná. při plnění úkolu byl testovaný jedinec sledován s cílem zjistit, jak rychle se v aplikaci orientuje, jakým způsobem přišel na výsledek (některé úkoly měli více způsobů řešení) a které části aplikace pro něj byly problematické v orientaci. Po dobu plnění úkolů byl měřen čas a po dokončení byl zaznamenán. Před odpovídání na dotazy byl každý vyzván, aby se všechny odpovědi ideálně snažil rozvést do více vět a požádán, aby jakékoliv myšlenky sdělil klidně během připravených dotazů,

i když se jich přímo netýkaly. Diskuze na závěr byla dobrovolnou částí testování, kterou ale nakonec podstoupili všichni uživatelé.

7.3 Výsledky testování

Všechny testované osoby dokázaly splnit všech sedm úkolů. Celkový čas jejich plnění byl v rozmezí 60 a 150 vteřin, průměrný čas se pohyboval okolo 90 vteřin.

Hledání festivalu konajícího se v nejbližším termínu nedělalo žádnému uživateli větší problém. Sedm z nich uvedlo, že pochopili jejich řazení na obrazovce ihned po nahlédnutí. Dva zbývající porovnávaly data konání více festivalů a následně porozuměli řazení taktéž. Jeden účastník přecházel do obrazovky *Detail festivalu* a data hledal až tam. V závěrečné diskuzi pak uvedl, že si na první pohled nevšiml dat uvedených u jednotlivých festivalů už na vstupní obrazovce, proto je tam následně nehledal. Zobrazení festivalů, které se již konaly, nedělalo účastníkům mnoho potíží. Šest uživatelů přešlo na jejich obrazovku ihned, ostatní čtyři nejprve nahlédli do obrazovky *Hledání festivalů*, kde následně zjistili, že nemá vlastnost řadit festivaly podle data. Následně ihned využili druhé tlačítko nacházející se v navigační liště a dospěli tak k odpovědi. Hledání filtrování proběhlo rychle. Nejdelším časem splnění úkolu bylo zhruba 5 vteřin, průměrně potom těsně pod 3 vteřiny. Všichni testovaní využili navigačního tlačítka s lupou a následně ihned označili specifikovaný žánr. Nutno podotknout, že někteří z nich měli výhodu z předchozího úkolu, jelikož do této obrazovky již nahlédli předem. Hledání počtu událostí konkrétní den festivalu rozdělilo účastníky na poloviny. První polovina, která počet zjistila ze záložky *Časová osa*, v diskuzi uvedla, že záložkami postupovala postupně, tudíž se dostala nejprve k této sekci, ze které již počet zjistila. Druhá polovina se nejprve podívala na popisky nacházející se ve spodním přepínači záložek a zvolila záložku *Seznam událostí*, ve které očekávala, že odpověď nalezne. Při úkolu na časově se překrývající události zůstala polovina, která vyčítala předchozí počet událostí ze záložky *Časová osa* na stejném místě a úkol splnila. Pouze dvě osoby z druhé poloviny přepnuli aktivní záložku na časovou osu a údaje zjišťovali tam. Ostatní tři zůstali v záložce *Seznam událostí* a porovnávali časy jednotlivě. V diskuzi uvedli, že si v ten moment neuvědomili, že lze k tomuto úkolu, k jeho rychlejšímu splnění, využít grafického zpracování záložky *Časová osa*. Při nastavování navigace byl účelně vybrán stejný festival jako v minulém úkolu. Zkoumáno bylo, zdali uživatelé využijí k cestě do dané záložky přímo `tab bar` nebo použijí tlačítko zpět, které je vrátí na úvodní obrazovku a uvědomí si, že se musí k festivalu zpět vrátit. Osm osob zvládlo přepnutí přímo, dvě využili zpětného tlačítka. U filtrování událostí se devět osob správně přemístilo na záložku *Seznam událostí* a filtr použilo. Šest z nich v závěrečné diskuzi uvedlo, že si z úkolu na filtrování festivalů pamatovali, jak filtr vypadá a tudíž tušili, v jaké části obrazovky ho mají hledat. Jeden uživatel se na delší dobu zastavil v záložce *Časová osa* a snažil se najít filtrování tam. Následně se přepnul do správné záložky. Bližší informace o události našlo osm účastníků na první pokus. Dva z nich se nejprve pokusili hledat v záložce *Časová osa*, až následně informace našli v záložce sousední.

Z následných přichystaných dotazů vyplynulo najevo, že aplikace byla pro všechny zúčastněné spíše jednoduchá k použití. Nejoblíbenější prvkem se stala záložka *Časová osa* díky možnosti grafické interpretace konání událostí. Většina uživatelů si všimla, že barvy událostí nejsou v této záložce náhodné, ale že korespondují s žánry událostí. Dále byla kladně hodnocena úvodní obrazovka aplikace, konkrétně její vizuální zpracování. Tři účastníci uvedli, že jim v aplikaci chybí informace o vstupenkách, konkrétně jejich cena. Jeden uživatel navrhl, že by v aplikaci uvítal mapu areálu korespondující s pojmenováním míst

konání událostí. Všechny osoby uvedly, že se ještě nikdy s podobnou aplikací v mobilu nesetkaly, jedna z nich zmínila, že podobnou službu již viděla na internetu. Souhlasily též s opětovným použitím aplikace pro vyhledání informací o festivalech. na závěr v připomínkách dva uživatelé zmínili, že by jim přišlo přívětivější, kdyby přepínání záložek v obrazovce *Detail festivalu* bylo ve vrchní části namísto spodní a jeden další by rozdělil vyhledávání a filtrování festivalů na samostatné části.

7.4 Zhodnocení a možnost pokračování

Testování dopadlo dle mého názoru poměrně úspěšně. Většina testovaných osob při každém úkolu volila správnou cestu k dosažení odpovědi. S kontrolními prvky a tlačítky neměl nikdo větší problém. Návrh na přesun přepínání mezi záložkami do vrchní části obrazovky jsem původně vůbec neočekával, jelikož systémové aplikace iOS tuto funkcionalitu řeší zpravidla pomocí `tab baru`.

V rámci možnosti pokračování jsem uvážil svoje nápady společně s připomínkami testovaných uživatelů. Informace k festivalu by se daly rozšířit o ceny vstupného a možné přiložení mapy areálu pro zlepšení orientace společně s detailnějšími údaji o jednotlivých místech konání. Ty mají v systému již vyčleněné místo, v aplikaci však prozatím nejsou zobrazeny. Hodnocení festivalů a recenze by mohly uživatelům detailněji povědět o jeho průběhu v minulých letech a zajistit tak hlavně menším festivalům pravidelnou návštěvnost.

Kapitola 8

Závěr

Výsledkem mojí bakalářské práce je klient-server aplikace sdružující informace o festivalech na jedno místo. Serverová část se stará o uložení údajů a nad rámec zadání poskytuje webové rozhraní organizátorům, kteří si zde mohou svůj festival zaregistrovat. Mobilní aplikace vyvinuta na platformu iOS zobrazuje strukturované informace o jednotlivých festivalech. Uživatelé mohou festivaly též vyhledávat či filtrovat podle názvu. Časová osa v aplikaci graficky zobrazuje v kolik hodin a na jakém místě se události na festivalu konají, uživatelé tak mají přehled o časových kolizích a jsou schopni dopředu plánovat svoji účast. do místa konání festivalu umožňuje aplikace nastavit navigaci pomocí systémové aplikace *Mapy*. Aplikace je vhodná pro všechny druhy festivalů, jako třeba hudební, filmové a další.

Práce na začátku popisuje teorii potřebnou k vytvoření návrhu celé aplikace. Následuje porovnání a analýza podobných existujících aplikací společně s tvorbou uživatelského rozhraní klientské aplikace. Hned poté práce uvádí proces návrhu architektury celého systému včetně vzájemné komunikace. na závěr práce předkládá způsob implementace a závěrečné testování funkcionality a uživatelského rozhraní.

Hlavním podnětem k výběru tohoto tématu byla absence aplikace na trhu. Motivací pak bylo podpoření menších festivalů, které si nemohou dovolit vytvořit vlastní aplikaci, sjednocení informací na jedno místo společně s eliminací roztříštěnosti uživatelských rozhraní aplikací vytvořených na míru pro větší festivaly.

Možnosti pokračování ve vývoji aplikace by zahrnovaly rozšíření počtu informací o každém festivalu, začlenění mapy areálu místa konání nebo například systém hodnocení a recenzí. V blízké budoucnosti však neplánuji v jejím vývoji pokračovat dále.

Literatura

[1] *Flask: web development, one drop at a time.* Dostupné z: https://flask.palletsprojects.com/, [Online; navštíveno 26.5.2020].

[2] *Laravel: The PHP Framework for Web Artisans.* Dostupné z: https://laravel.com/, [Online; navštíveno 26.5.2020].

[3] *A Bit History of Internet: Client-Server*. Dostupné z: https://en.wikibooks.org/wiki/A_Bit_History_of_Internet/Chapter_5_:_Client-Server, 2003, [Online; navštíveno 15.01.2020].

[4] *Developer Survey Results 2019*. Dostupné z: https://insights.stackoverflow.com/survey/2019, 2019, [Online; navštíveno 23.5.2020].

[5] *WWDC 2019 Keynote*. Dostupné z: https://developer.apple.com/videos/play/wwdc2019/101/, 2019, [Online; navštíveno 15.3.2020].

[6] Altvater, A.: *SOAP vs. REST: The Differences and Benefits Between the Two Widely-Used Web Service Communication Protocols*. Dostupné z: https://stackify.com/soap-vs-rest/, Březen 2017, [Online; navštíveno 13.01.2020].

[7] Apple Inc.: *Cocoa Fundamentals Guide*. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/CocoaFundamentals/WhatIsCocoa/WhatIsCocoa.html, [Online; navštíveno 19.1.2020].

[8] Apple Inc.: *Foundation framework*. Dostupné z: https://developer.apple.com/documentation/foundation, [Online: navštíveno 19.1.2020].

[9] Apple Inc.: *iOS 13: A whole new look. On a whole new level.* Dostupné z: https://www.apple.com/ios/ios-13/, [Online; navštíveno 19.01.2020].

[10] Apple Inc.: *iOS Human Interface Guidelines*. Dostupné z: https://developer.apple.com/design/human-interface-guidelines/ios/overview/themes/, [Online; navštíveno 3.2.2021].

[11] Apple Inc.: *Programming with Objective-C*. Dostupné z: https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html, [Online; navštíveno 1.2.2021].

[12] Apple Inc.: *Swift*. Dostupné z: https://swift.org/, [Online; navštíveno 2.2.2021].

[13] Apple Inc.: *SwiftUI framework*. Dostupné z: https://developer.apple.com/documentation/swiftui, [Online; navštíveno 19.1.2020].

[14] Apple Inc.: *UIKit framework*. Dostupné z: https://developer.apple.com/documentation/uikit, [Online; navštíveno 19.1.2020].

[15] Apple Inc.: *Xcode*. Dostupné z: https://developer.apple.com/documentation/xcode/, [Online; navštíveno 3.2.2021].

[16] Berners-Lee, T.; Fielding, R. T.; Masinter, L.: *Uniform Resource Identifier (URI): Generic Syntax*. Dostupné z: https://www.rfc-editor.org/rfc/rfc3986.txt, Leden 2005.

[17] Box, D.; Ehnebuske, D.; Kakivaya, G.; aj.: *Simple Object Access Protocol (SOAP) 1.1*. Dostupné z: https://www.w3.org/TR/2000/NOTE-SOAP-20000508/, Květen 2000, [Online; navštíveno 17.1.2020].

[18] Bray, T.: *The JavaScript Object Notation (JSON) Data Interchange Format*. Dostupné z: https://www.rfc-editor.org/rfc/rfc8259.txt, Prosinec 2017.

[19] Bray, T.; Paoli, J.; Sperberg-McQueen, C. M.; aj.: *Extensible Markup Language (XML) 1.0 (Fifth Edition)*. Dostupné z: https://www.w3.org/TR/xml/, Listopad 2008, [Online; navštíveno 17.1.2020].

[20] Christensson, P.: *Client-Server Model Definition*. Dostupné z: https://techterms.com/definition/client-server_model, Červen 2016, [Online; navštíveno 15.01.2020].

[21] Christie, T.: *Django REST framework is a powerful and flexible toolkit for building Web APIs*. Dostupné z: https://www.django-rest-framework.org/, [Online; navštíveno 12.9.2020].

[22] Daniel, S. F.: *Xcode 4 iOS Development Beginner's Guide*. Packt Publishing, Srpen 2011, ISBN 1849691304.

[23] Django Software Foundation: *Django: The web framework for perfectionists with deadlines*. Dostupné z: https://www.djangoproject.com/, [Online; navštíveno 26.5.2020].

[24] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce, University of California, Irvine, 2000.

[25] Ghimire, D.: *Comparative study on Python web frameworks: Flask and Django*. Bakalářská práce, Metropolia University of Applied Sciences, 2020.

[26] Hein, B.: *The evolution of iOS: From iPhone OS to iOS 11*. Dostupné z: https://www.cultofmac.com/488454/ios-evolution-iphone-os/, [Online; navštíveno 18.01.2020].

[27] Jacobs, B.: *Exploring the iOS SDK*. Dostupné z: https://code.tutsplus.com/tutorials/exploring-the-ios-sdk--mobile-13959, Prosinec 2012, [Online; navštíveno 19.01.2020].

[28] Jobs, S.: *Macworld Keyonte*. 2007, Leden 8.-12.

[29] Kanaiza, F.: *Two-tier & Three-tier Architecture*. Dostupné z: https://medium.com/@fleviankanaiza/two-tier-three-tier-architecture-8b02536d3482, [Online; navštíveno 15.01.2020].

[30] Kinnunen, M.; Honkanen, A.; Luonila, M.: *Frequent music festival attendance: festival fandom and career development*. Dostupné z: https://doi.org/10.1108/IJEFM-08-2020-0050, [Online; navštíveno 15.3.2021], ISSN 1758-2954.

[31] Kurose, J. F.; Ross, K. W.: *Computer Networking: A Top-Down Approach*. Pearson Education, Inc., sedmé vydání, 2016, ISBN 0-13-359414-9.

[32] Luboslav Lacko: *Vývoj aplikací pro iOS*. Computer Press, 2018, ISBN 978-80-251-4942-3.

[33] Monus, A.: *SOAP vs REST vs JSON comparsion [2019]*. Dostupné z: https://raygun.com/blog/soap-vs-rest-vs-json/, Srpen 2018, [Online; navštíveno 17.01.2020].

[34] Orchard, D.; McCabe, F.; Ferris, C.; aj.: *Web Services Architecture*. Dostupné z: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, Únor 2004, [Online; navštíveno 16.1.2020].

[35] Purushothaman, J.: *RESTfu Java Web Services*. Packt Publishing, druhé vydání, 2015, ISBN 1784399094.

[36] Strauss, D.: *Apple hits record high, extends market value above $1 trillion (AAPL)*. Dostupné z: https://markets.businessinsider.com/news/stocks/apple-stock-price-hits-record-market-value-exceeds-1-trillion-2019-10-1028594012, [Online; navštíveno 17.01.2020].

[37] Terrell, E.; Richardson, A.: *Apple Computer, Inc.* Dostupné z: https://www.loc.gov/rr/business/businesshistory/April/apple.htm, [Online; navštíveno 17.01.2020].

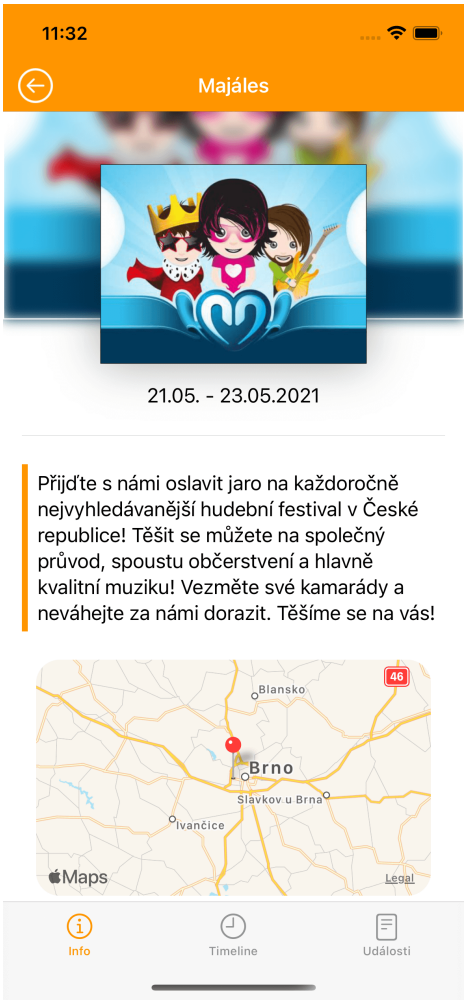
Příloha A

Obrazovky aplikace

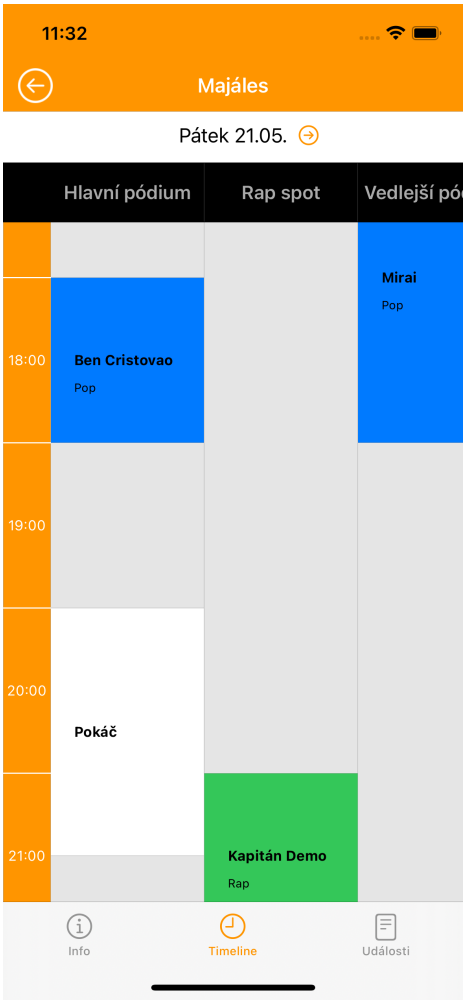
Obrázek A.1: Obrazovka *Seznam festivalů* se seznamem nadcházejících festivalů.

Obrázek A.2: Obrazovka *Seznam festivalů* se seznamem uplynulých festivalů.

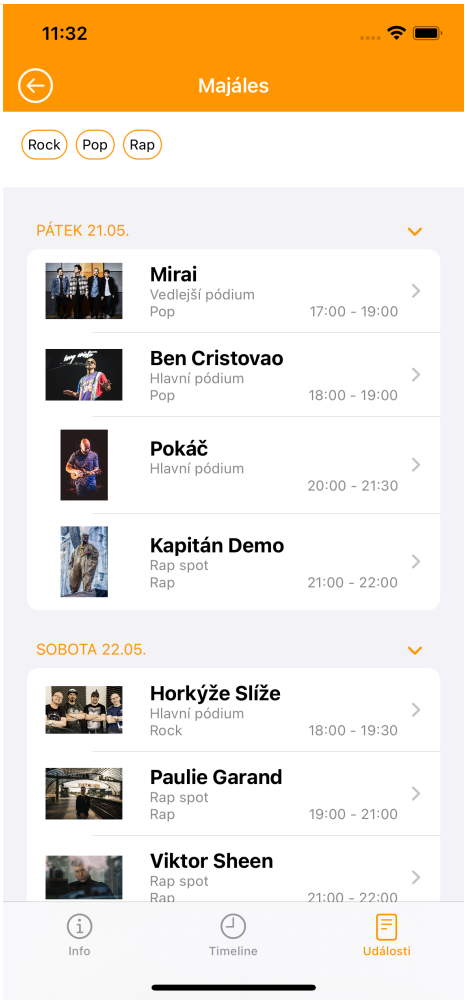
Obrázek A.3: Obrazovka *Hledání festivalů*.



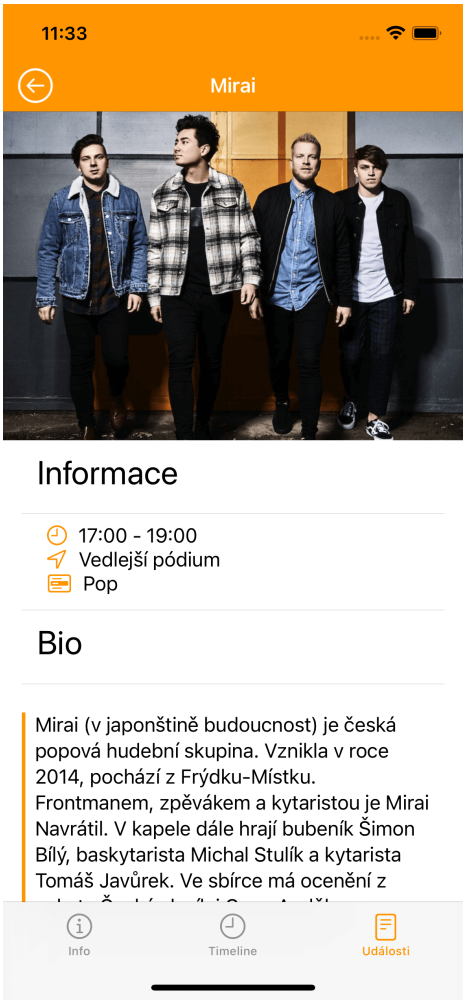
Obrázek A.4: Záložka *Základní informace* obrazovky *Detail festivalu*.



Obrázek A.5: Záložka *Časová osa* obrazovky *Detail festivalu.*



Obrázek A.6: Záložka *Seznam událostí* obrazovky *Detail festivalu.*

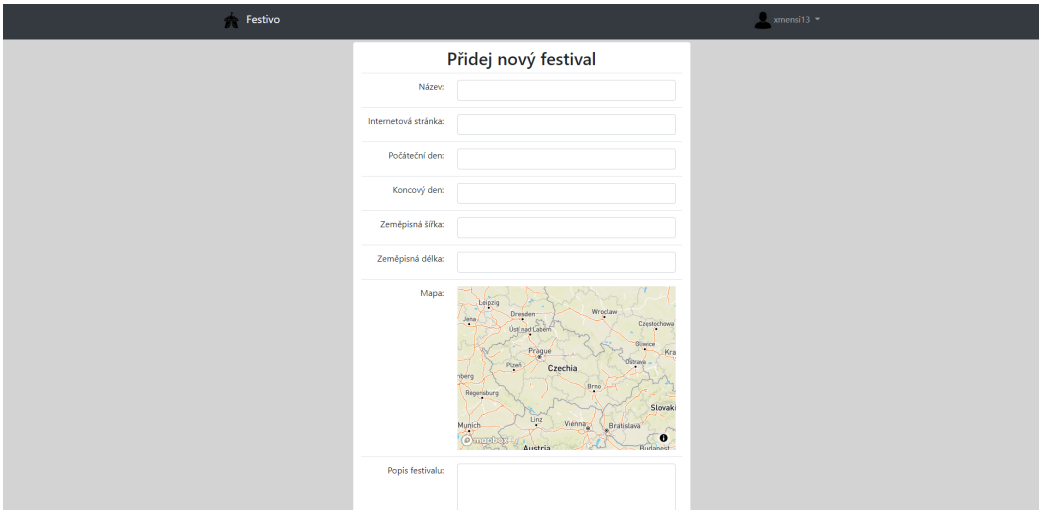


Obrázek A.7: Obrazovka *Detail události*.

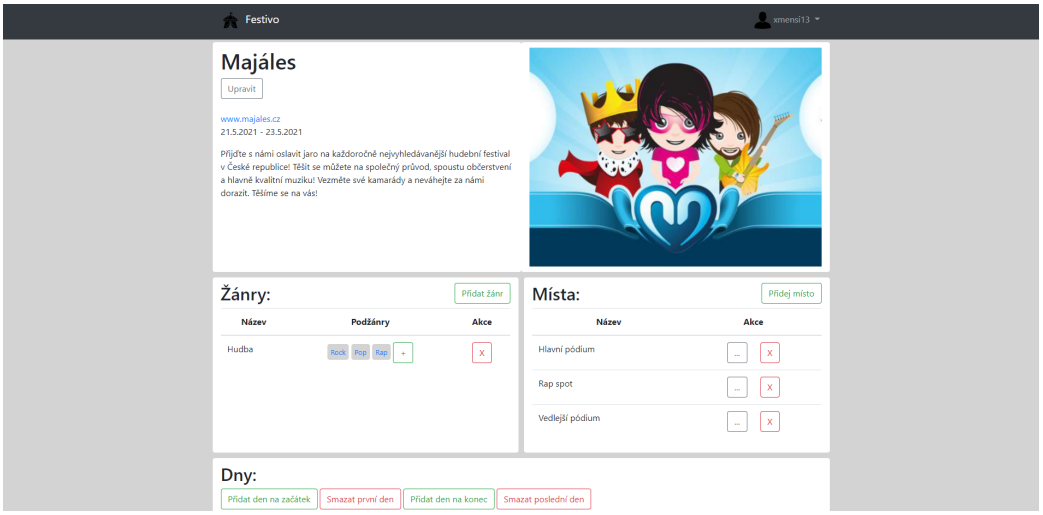
Příloha B

Ukázka webového rozhraní

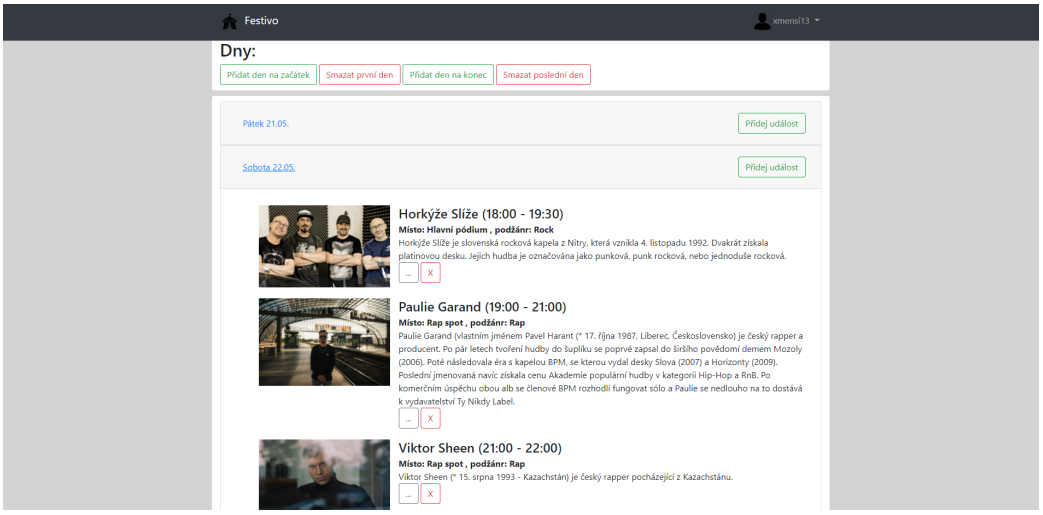
Obrázek B.1: Hlavní stránka webového systému.



Obrázek B.2: Vrchní část formuláře pro přidání festivalu do systému.



Obrázek B.3: Vrchní část stránky s detailními informacemi o festivalu.



Obrázek B.4: Spodní část stránky s detailními informacemi o festivalu.

Příloha C

Využité knihovny

- *Bootstrap – MIT License*
  https://github.com/twbs/bootstrap
- *Datepicker – MIT License*
  https://github.com/fengyuanchen/datepicker
- *MapBox GL JS – proprietární software (omezená licence zdarma)*
  https://github.com/mapbox/mapbox-gl-js
- *Django Rest Framework – BSD License*
  https://github.com/encode/django-rest-framework
- *Spreadsheetview – MIT License*
  https://github.com/bannzai/SpreadsheetView

Příloha D

Testovací úkoly a otázky

Úkoly

1. Zjistěte, který festival se koná v nejbližším termínu.
2. Zobrazte si festivaly, které proběhly v minulosti.
3. Zobrazte si pouze festivaly, které jsou žánru .....
4. Nalezněte, kolik událostí se koná druhý den festivalu .....
5. Zjistěte, zdali se časově některé z nich překrývají, pokud ano, uveďte které.
6. Nastavte si do místa konání festivalu ..... navigaci.
7. Zobrazte si pouze události festivalu ....., které jsou žánru .....
8. Zjistěte bližší informace o poslední konané události na festivalu .....

Otázky

1. V jakých případech byste aplikaci použil? Uveďte alespoň dva.
2. Přišlo Vám její použití jednoduché nebo složité? Proč?
3. Která vlastnost či vizuální prvek se Vám na aplikaci líbil nejvíce?
4. Chybí Vám v aplikaci nějaká informace nebo vlastnost? Popřípadě která a proč?
5. Už jste se setkal/a s nějakou podobnou aplikací?
6. Použil/a byste aplikaci znovu pro vyhledání informací o festivalech?
7. Setkal jste se v aplikaci s něčím, čemu jste nerozuměl/a nebo nepochopil/a, proč se to v ní nachází? Popřípadě s čím a proč?

Příloha E

Plakát