Česká zemědělská univerzita v Praze

Provozně ekonomická fakulta

Katedra Informačního Inženýrství

Diplomová práce

Cross-platform vývoj mobilních aplikací pomocí Flutter Framework

Bc. Adam Mader

© 2024 ČZU v Praze

ČESKÁ ZEMĚDĚLSKÁ UNIVERZITA V PRAZE

Provozně ekonomická fakulta

ZADÁNÍ DIPLOMOVÉ PRÁCE

Bc. Adam Mader

Informatika

Název práce

Cross-platform vývoj mobilních aplikací pomocí Flutter Framework

Název anglicky

Cross-platform mobile app development using Flutter Framework

---

Cíle práce

Téma diplomové práce je zaměřené na problematiku vývoje cross-platform mobilních aplikací a implementaci prototypu pro platformy Android a iOS za použití technologie Flutter. Cílem práce je přiblížení problematiky práce s Flutter a následná implementace aplikace s využitím této technologie. Dále se práce zaměřuje na popis nativního vývoje aplikací pro jednotlivé operační systémy, programovací jazyk Dart, cross-platform aplikace a UI/UX design.

Metodika

Teoretická část práce je zaměřena na popis operačního systému iOS a Android, typů vývoje mobilních aplikací, programovacího jazyku Dart, Flutter SDK a UI/UX návrhu. Tato část práce bude vycházet ze studia odborných zdrojů dané problematiky. Praktická část bude zaměřena na návrh uživatelského rozhraní pomocí informační architektury wireframe, které budou vytvořeny pomocí designového nástroje Figma. Implementace proběhne ve vývojovém prostředí Visual Studio Code za použití Flutter SDK.

Doporučený rozsah práce

60-80 stran

Klíčová slova

Flutter, SDK, Mobilní aplikace, Dart, Cross-platform, Wireframe

---

Doporučené zdroje informací

BAILEY, Thomas a Alessandro BIESSEK, 2021. Flutter for Beginners. 2nd ed. Packt Publishing. ISBN ISBN-10 1800565992v.

Flutter architectural overview, b.r. Flutter documentation | Flutter [online]. [cit. 2024-03-12]. Dostupné z: https://docs.flutter.dev/resources/architectural-overview

WALEED, Arshad, 2021. Managing State in Flutter Pragmatically. Packt Publishing. ISBN ISBN-10 1801070776.

Předběžný termín obhajoby

2023/24 LS – PEF

Vedoucí práce

Ing. Martin Pelikán, Ph.D.

Garantující pracoviště

Katedra informačního inženýrství

Elektronicky schváleno dne 26. 11. 2022

Ing. Martin Pelikán, Ph.D.

Vedoucí katedry

Elektronicky schváleno dne 28. 11. 2022

doc. Ing. Tomáš Šubrt, Ph.D.

Děkan

V Praze dne 31. 03. 2024

Čestné prohlášení

Prohlašuji, že svou diplomovou práci Cross-platform vývoj mobilních aplikací pomocí Flutter Framework jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou citovány v práci a uvedeny v seznamu použitých zdrojů na konci práce. Jako autor uvedené diplomové práce dále prohlašuji, že jsem v souvislosti s jejím vytvořením neporušil autorská práva třetích osob.

V Praze dne 31.03.2024 ______________________________

Poděkování

Rád bych touto cestou poděkoval panu Ing. Martinu Pelikánovi, Ph.D. za ochotu a vstřícnost během konzultací.

Cross-platform vývoj mobilních aplikací pomocí Flutter Framework

Abstrakt

Tato diplomová práce se zabývá tématem cross-platform vývoje mobilních aplikací a implementace prototypu aplikace pro platformy Android a iOS za použití technologie Flutter. Teoretická část vychází ze studia odborných zdrojů daných problematik. Tato část umožňuje náhled do jednotlivých možností vývoje mobilních aplikací a klade důraz na konkrétní technologii cross-platform vývoje – Flutter. Výstupem praktické části práce je funkční prototyp mobilní aplikace pro hráče stolních fotbálků. Tato aplikace umožňuje vyhledat podniky na základě značky stolu, zapisovat si skóre jednotlivých zápasů a číst novinky, které jsou publikovány jednotlivými podniky. Jelikož se jedná o prototyp, data jsou pouze simulována a uložena lokálně. V budoucnu by však aplikace mohla být rozšířena o komunikaci se serverem.

Klíčová slova: Aplikace, Flutter, Framework, Cross-platform, Dart, Widget, Desktop, Web, iOS, Android, UI, UX, Wireframe

Cross-platform mobile app development using Flutter Framework

Abstract

This thesis focuses on the topic of cross-platform mobile application development and the implementation of a prototype application for Android and iOS platforms using Flutter framework. The theoretical part is based on the study of expert sources of the given subjects. This part gives an insight into the different possibilities of mobile application development and emphasizes on a particular cross-platform development technology – Flutter. The output of the practical part of the thesis is a functional prototype of mobile application for table football players. This application provides the possibility to search for establishment base on the table brand, to write down the scores of individual matches and to read news published by individual pubs. Since this is a prototype, the data are simulated are stored locally. In the future application could be extended to communicate with the server.

Keywords: Application, Framework, Cross-platform, Dart, Widget, Desktop, Web, iOS, Android, UI, UX, Wireframe

Obsah

1 Úvod **10**

2 Cíl práce a metodika **11**

2.1 Cíl práce 11

2.2 Metodika 11

3 Teoretická východiska **12**

3.1 Operační systémy a vývoj chytrých telefonů 12

3.1.1 Android 12

3.1.2 iOS 14

3.2 Mobilní aplikace 16

3.2.1 Historie mobilních telefonů a mobilních aplikací 16

3.2.2 Druhy vývoje mobilních aplikací 16

3.2.2.1 Hybridní vývoj aplikací 17

3.2.2.2 Nativní vývoj aplikací 18

3.2.2.3 Cross-platform vývoj aplikací 23

3.3 Dart 27

3.3.1 Platformy Dart 28

3.3.2 Objektově orientovaný jazyk 30

3.4 Flutter 31

3.4.1 Výhody Flutter Development 31

3.4.2 Architektura Flutter 32

3.4.3 Platformové kanály 33

3.4.4 Flutter web 34

3.4.5 Widget 35

3.4.6 Flutter state management 40

3.4.7 Flutter doplňky 41

3.4.8 Testování 41

3.5 UI/UX design 42

3.5.1 User Interface (UI) 42

3.5.2 User Experience (UX) 43

3.6 Unified modeling language (UML) 45

4 Praktická část **47**

4.1 Návrh aplikace 48

4.1.1 Informační architektura 48

4.1.2 Wireframe 48

4.1.3 Diagram tříd ... 52
4.1.4 Práce s daty v aplikaci ... 52
4.2 Implementace aplikace ... 53
4.2.1 Flutter balíčky využité pro aplikaci ... 53
4.2.2 Inicializace aplikace ... 55
4.2.3 Funkční část obrazovky Home ... 59
4.2.4 UI část obrazovky Home ... 64
4.2.5 Testování aplikace ... 68

5 Výsledky a diskuze ... 73

6 Závěr ... 76

7 Seznam použitých zdrojů ... 77

8 Seznam obrázků ... 83

Přílohy ... 85

1 Úvod

Mobilní aplikace jsou nedílnou součástí našeho každodenního fungování, kdy v průměru stráví člověk 3 hodiny denně na svém chytrém telefonu. [1] Pouze během let 2016 až 2023 se zvýšil roční počet stažení mobilních aplikací ze 140 miliard na 257 miliard. [2]

V současné době dominují na mobilním trhu dvě platformy – Android od společnosti Google s podílem 71.4 % a iOS od společnosti Apple s 27.8 % podílem z celého trhu. [3] Obě platformy mají vlastní technologie pro vývoj aplikací. Pokud by bylo potřeba vytvořit mobilní aplikaci pro obě platformy, musely by být vytvořeny dvě separátní aplikace, které se chovají a vypadají stejně. Pro nativní vývoj na platformu Android jsou používány programovací jazyky Kotlin nebo Java. Pro platformu iOS byl dříve používán programovací jazyk Objective-C, který v současnosti doplnil programovací jazyk Swift. Vývoj dvou aplikací je však znatelně dražší než pouze aplikace jedné. Tato skutečnost je jeden z důvodu, který stál za vznikem cross-platform vývoje aplikací.

Tato práce je zaměřena na seznámí čtenáře s technologií Flutter a následnou implementací prototypu multiplatformní aplikace pro hráče stolních fotbálků.

Teoretická část práce se zaměřuje na představení dvou nejpopulárnějších operačních systémů Android a iOS, dále jsou představeny jednotlivé druhy vývoje mobilních aplikací na tyto platformy. Hlavním bodem teoretické části je představení možností cross-platform vývoje a především konkrétní technologie Flutter. V detailním rozboru jsou popsány jeho výhody a nevýhody, architektura a programovací jazyk Dart, na kterém Flutter funguje. Na závěr teoretické části je popsán design UI/UX a jazyk UML.

Praktická část práce je zpočátku zaměřena na návrh aplikace specifikací uživatelského rozhraní vytvořením wireframe, diagramu informační architektury a diagramu tříd. Hlavní část praktické části je zaměřena na popis implementace, která přibližuje postup vývoje aplikace při využití technologie Flutter. Na závěr praktické části bylo provedeno testování jednotlivých komponentů použitím různých přístupů k testování.

2 Cíl práce a metodika

2.1 Cíl práce

Hlavním cílem práce je přiblížení problematiky vývoje cross-platform mobilních aplikací a implementaci prototypu aplikace pro operační systémy Android a iOS za použití technologie Flutter. Dílčím cílem této části je seznámení čtenáře s jednotlivými platformami mobilních zařízení, možnostmi vývoje aplikací pro tyto platformy, představení programovacího jazyku Dart a návrhem UI/UX designu.

2.2 Metodika

Diplomová práce je rozdělena do části teoretické a praktické. Teoretická část práce je zaměřena na analýzu a studium odborných zdrojů. Praktická část stojí na poznatcích získaných v teoretické části. První část je zaměřena na návrh uživatelského rozhraní aplikace pomocí informační architektury a wireframe. Pro vytvoření wireframe je využit designový nástroj Figma. Tyto návrhy slouží k popisu struktury a rozložení jednotlivých prvků na obrazovce. Dále je vytvořen diagram tříd, který definuje struktury a vztahy mezi jednotlivými třídami. K Implementaci aplikace je využita technologie k vývoji cross-platformových aplikací Flutter. K vytvoření aplikace je využito integrované vývojové prostředí Visual Studio Code.

3 Teoretická východiska

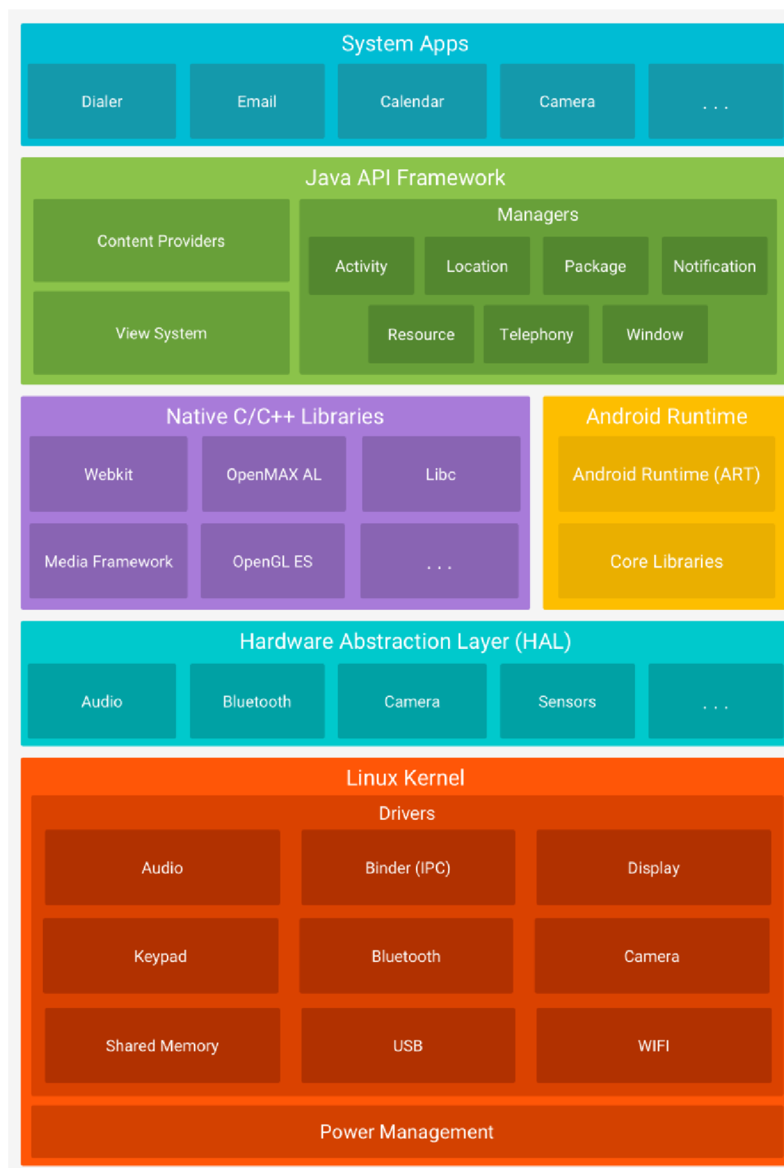
3.1 Operační systémy a vývoj chytrých telefonů

Operační systém pro mobilní zařízení je software, který zajišťuje prostředí, ve kterém mohou uživatelé zařízení spouštět aplikace pohodlným a efektivním způsobem.
Postupem času se návrhy operačního systému pro mobilní zařízení vyvíjely od operačních systémů založených na PC směrem k vestavěnému operačnímu systému až po operační systémy dnešních chytrých telefonů. Architektura mobilních operačních systémů se postupně také měnila od složitějších k méně složitým. Tyto změny byly zapříčiněny různými aspekty – například: technologickým pokrokem, internetem nebo vývojem nových služeb. Z pohledu vývoje hardwarových technologií se mikroprocesory postupně zmenšovaly a byly představeny funkce jako jsou senzory – dotykové obrazovky. Tento pokrok vedl k stále rostoucímu počtu uživatelů. S rozvojem internetu se stávali lidé čím dál tím více angažovaní ve vyhledávání informací, používání aplikací nebo v komunikaci na sociálních sítích. Veškerý pokrok vedl ke vzniku různých konkurenčních operačních systémů. Operační systémy se odlišují v mnoha parametrech, například: uživatelské rozhraní, využití paměti, zabezpečení nebo platformou pro vývoj aplikací. [4]

3.1.1 Android

Android je operační systém vytvořený v roce 2003 společností Android, která byla v roce 2005 odkoupena společností Google. V roce 2007 uvedl Google ve spolupráci s OHA (Open Handset Aliance) verzi Android Open Source Platform. Android je primárně zaměřený na zařízení pro chytré telefony a tablety. Operační systém Android je založený na Linux jádru a dalších open-source software. [5]

Obrázek 1 Architektura Android Platformy



Zdroj: [6]

Linux Kernel – Jak již bylo řečeno, základem platformy Android je Linuxové jádro. Použití Linuxového jádra umožňuje systému Android využívat klíčové bezpečnostní funkce.

Hardwarová Abstrakční vrstva (HAL) – Tato vrstva poskytuje standartní rozhraní, která umožňují hardwarové možnosti zařízení nadřazenému rámci Java API. Skládá se z několika modulů z nichž každý implementuje rozhraní pro daný typ hardwarové komponenty (např. Bluetooth nebo kamera).

Android runtime – Zařízení, které disponují verzi Android 5.0 nebo vyšší probíhá každá aplikace ve vlastním procesu a s vlastní instancí běhového prostředí.

Nativní knihovny C/C++ - Velká část klíčových komponent a služeb systému Android (např. ART a HAL), je vytvořena z nativního kódu, který vyžaduje nativní knihovny napsané v programovacích jazycích C a C++. Platforma Android umožňuje přístup k určitým funkcím těchto nativních komponent skrze Java framework API.

Java API framework – Veškeré funkce operačního systému Android jsou k dispozici prostřednictvím rozhraní API, který je napsaný v programovacím jazyce Java. Tato rozhraní API tvoří základní prvek, který je potřeba k vytváření mobilních aplikací pro operační systém Android.

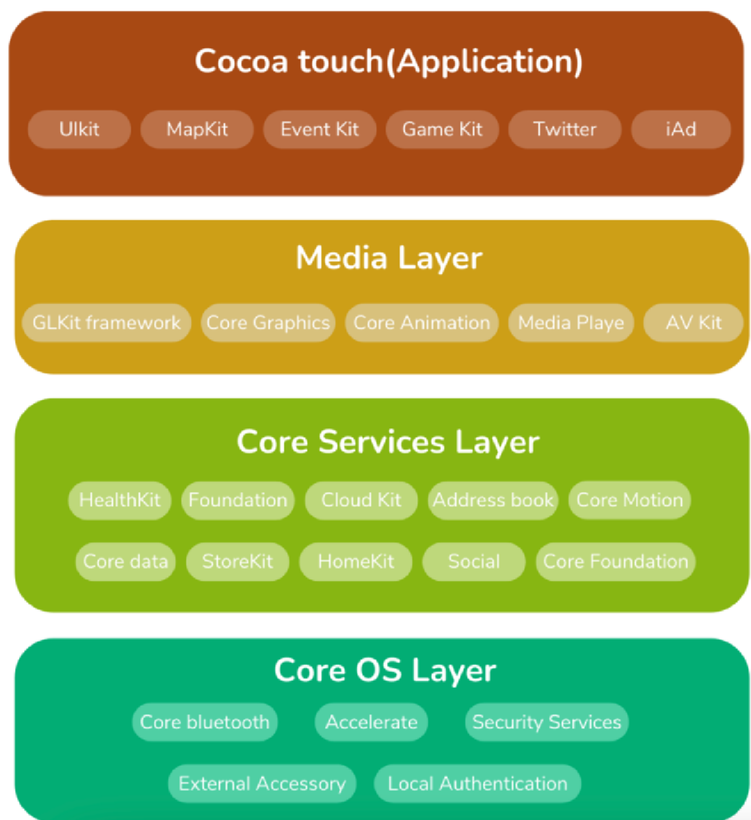
Systémové aplikace – Systém Android obsahuje balíček základních aplikací jakými jsou například: e-mail, kalendář, kontakty a další. [5]

3.1.2 iOS

Operační systém iOS byl vytvořen společností Apple pro jejich přenosná zařízení. Jedná se o druhý nejpoužívanější operační systém pro mobilní telefony po systému Android.

Struktura operačního systému iOS je vrstvená. Komunikace mezi nimi neprobíhá přímo – ke komunikaci napomáhají vrstvy mezi aplikační a hardwarovou vrstvou. Nižší vrstva poskytuje základní operace, na které závisí veškeré aplikace zařízení. Vyšší vrstvy poskytují podporu pro služby související s grafikou a rozhraním. [7]

Obrázek 2 Architektura iOS platformy



Zdroj: [7]

Core OS – Spodní část vrstvené struktury iOS se nachází bezprostředně nad hardwarem zařízení. Funkce na nízké úrovni tvoří základ všech funkcí systému iOS, které jsou zajištěné právě Core OS. Kromě standardních funkcí základního operačního systému, kterými jsou například práce se souborovým systémem a vlákny, správa paměti, zajišťuje také služby přístupu k externímu příslušenství.

Core Services – Služby poskytované vrstvou Core OS jsou abstrahovány vrstvou Core Services. Tato vrstva poskytuje základní služby, které mohou využívat všechny aplikace.

Media Layer – Multimediální služby zajišťuje vrstva Media Layer. Ta umožňuje využívat celou zvukovou, grafickou a video technologii systému. Poskytuje vývojářům možnost pracovat s prvky jakou jsou animace, fotografie, filmy a zvuk.

Cocoa Touch – Vrstva Cocoa Touch představuje abstraktní vrstvu, která poskytuje různé knihovny pro programování na zařízení iPhone a další zařízení s iOS. Součástí této vrstvy je také důležitá skupina frameworků Objective-C, které byly vytvořeny pomocí rozhraní Mac OS X Cocoa API. Tato vrstva podporuje multitasking, oznámení, veškeré systémové služby na vysoké úrovni a další důležité technologie. [8]

3.2 Mobilní aplikace

Jedná se o software/soubor programů, který je spuštěn na mobilním zařízení a provádí pro uživatele určité operace. Mobilní aplikace by měly být snadné pro používání, uživatelsky přívětivé a dostupné pro zařízení všech cenových kategorií. Mobilní aplikace mají široké využití například: volání, zasílání zpráv, komunikace na sociálních sítích a mnoho dalších. Určité aplikace bývají nainstalovány v mobilních zařízeních jich při zakoupení, další aplikace si lze stáhnout z obchodu s aplikacemi dané platformy nebo z internetu. [9]

3.2.1 Historie mobilních telefonů a mobilních aplikací

O prvních mobilních aplikacích jako takových můžeme mluvit od konce 20. století, avšak tyto aplikace byly mnohem jednodušší, než které využíváme dnes. V 80. letech minulého století byl vyvinut první osobní digitální asistent společností Psion, který byl vybaven kalkulačkou, hodinami a dalšími aplikacemi, které v dnešní době považujeme za samozřejmé. V roce 1993 byl představen první chytrý telefon od společnosti IBM, který byl vybaven mnoha jednoduchými aplikacemi. Dalším pokrok v oblasti mobilních aplikací přišel v roce 2002 s chytrým telefonem BlackBerry, který jako první svým uživatelům umožnil odesílání e-mailů. V roce 2010 se dostaly mobilní aplikace do popředí díky společnosti Apple, která představila svůj obchod s aplikacemi s názvem App Store. [10]

3.2.2 Druhy vývoje mobilních aplikací

Při výběru technologie pro tvorbu mobilní aplikace by se měly brát v úvahu tři základní parametry: kvalita, náklady a čas. Ne všechny organizace mají možnost věnovat vývoji aplikace neomezené zdroje. Při rozhodování mezi nativními, hybridním či cross-platform vývojem je třeba zvolit metodu, která nejlépe dokáže pokrýt potřeby uživatelů a vyjde vstříc

obchodním omezením. Důležitou součástí rozhodování je také definováním cílové skupiny uživatelů, zda se například jedná o uživatele určité platformy či uživatele různých platforem. Jakmile je definována cílová skupina je potřeba brát v potaz následující aspekty.

Složitost aplikace – Čím je aplikace složitější, tím spíše přichází v úvahu nativní vývoj. Pomocí nástrojů pro platformy iOS, Android nebo další systémy, včetně rozhraní pro programování a sad pro vývoj softwaru, mohou vývojáři přizpůsobit aplikaci na konkrétní platformu. Výsledkem nativního vývoje je nejlepší možný výkon a uživatelská zkušenost. Pokud se jedná o jednodušší aplikace, tím spíše lze uvažovat o hybridním nebo cross-platformovém vývoji, který je rychlejší a méně nákladný.

Dosah aplikace – Nativní aplikace jsou ke stažení výhradně v obchodech s aplikacemi pro konkrétní platformy. Pro platformu iOS obchod App Store, pro Android Google Play. Hybridní a cross-platformové aplikace lze zpřístupnit na více obchodech s aplikacemi a díky tomu generovat větší viditelnost na trhu. Hybridní aplikace navíc mohou být preferovány uživateli, kteří dávají přednost přístupu z webového prohlížeče před jejím stažením.

Zabezpečení aplikací – Bezpečnostní rizika se stále častěji zaměřují na více mobilních platforem, čímž je ohroženo soukromí a data stále většího počtu podniků a zákazníků. Nativní vývoj aplikací může zajistit kvalitnější ochranu pro uživatele, než hybridní či cross-platform vývoj, díky možnosti využití nativní programovací knihovny, která poskytuje vývojářům nástroje pro vytváření vestavěných bezpečnostních funkcí. [11]

3.2.2.1 Hybridní vývoj aplikací

Vývoj hybridních aplikací zahrnuje využití webových jazyků HTML, CSS a JavaScript ke kódování aplikace, které jsou následně spouštěny v nativním prostředí s vestavěným webovým prohlížečem. Díky tomuto přístupu mohou vývojáři sdílet kód serverové části, což umožňuje rychle vytvářet aplikace s nižšími náklady. Avšak závislost na prohlížeči může vést k nedostatečnému výkonu a nekonzistentním uživatelským rozhraním napříč systémy. [11]

3.2.2.2 Nativní vývoj aplikací

Nativní aplikace je taková, která je vyvinutá pouze pro jeden určitý operační systém (např. iOS nebo Android). Takový přístup vývoje vyžaduje použití programovacích jazyků specifických pro danou platformu a zohlednění vlastností každého systému. Pokud je tedy cílem vytvořit nativní aplikace pro iOS a Android, je nutno tak učinit zvlášť za použití příslušných technologií. Například k vývoji nativní aplikace pro iOS bude nutno použít programovací jazyky Objective-C nebo Swift, zatímco pro Android jazyk Java nebo Kotlin. Tyto aplikace lze stáhnout prostřednictvím obchodů s aplikacemi, jako jsou například App Store společnosti Apple nebo Google Play Store od společnosti Google. [11]

Výhody nativních aplikací

- **Rychlost aplikace** – Jednou z výhod nativních aplikací, oproti hybridním či multiplatformním je rychlost a výkonnost aplikace. Toho je dosaženo především díky tomu, že aplikace komunikuje přímo s nativním rozhraním API a není závislá na middleware. Díky tomu, že nativní aplikace dostávají plnou podporu hardwaru a operačního systému, je zaručena vysoká rychlost a efektivita.

- **Pokročilé přizpůsobení** – Vzhledem k tomu, že nativní aplikace využívají většinu funkcí operačního systému, lze je velmi dobře přizpůsobit. Díky tomu, že vývojáři nemusí kombinovat funkce pro dva operační systémy, mají méně omezení a mohou se zaměřit na vytvoření jediného řešení.

- **User experience** – Každá platforma má své vlastní zásady UI/UX, které by měly být vývojáři dodrženy. Při vývoji nativní aplikace se tyto standardy dobře dodržují, což vede ke vzhledu, který je v souladu s operačním systémem. Konzistence nativních mobilních aplikací také přináší uživatelský zážitek, který je více intuitivní a interaktivnější, jelikož lidé znají rozvržení typické pro jejich operační systém.

- **Větší bezpečnost –** Dalším důvodem, proč zvolit nativní vývoj aplikace je vyšší úroveň zabezpečení. Vzhledem k rostoucím obavám o ochranu dat by softwarová

řešení měla ve všech odvětvích zajistit uživatelům pocit bezpečí při sdílení digitálních informací. Data jsou šifrována pouze v rámci jedné infrastruktury, což výrazně snižuje rizika spojená se zabezpečením.

- **Méně chyb** – Jelikož vytváření nativních aplikací zahrnuje použití specifických a pro každou platformu odlišných nástrojů, má méně technických závislostí a snižuje možnost výskytu chyb. Ve výsledku může být udržování dvou aplikací ve dvou nezávislých codebase méně náročné a náchylné k chybám než implementace dvou aplikací ve stejné codebase. [12]

Nevýhody vývoje nativních aplikací

- **Nulová flexibilita** – Vývojáři nemají žádnou flexibilitu, pokud se jedná o platformu pro vývoj nativních aplikací, jelikož musí kódovat pro každou platformu odděleně. Pokud je tedy zadáním nativní aplikace pro iOS a Android, je třeba dvou týmů vývojářů.

- **Nákladný vývoj** – Programátorů zabývajících se nativním vývojem je málo a proces je poměrně složitý. Vyžaduje tak více práce, což zvyšuje náklady na vývoj a časovou náročnost. Nutno brát také v potaz náklady na údržbu, které tvoří přibližně 15–20 % nákladů na vývoj aplikace.

Přestože mají nativní aplikace velké množství výhod, existuje zde také mnoho problémů. Pokud má projekt menší rozpočet a přísnější časová omezení, je na místě uvažovat o alternativním řešení. [13]

Nativní vývoj pro platformu Android

Kotlin je univerzální programovací jazyk, podobný programovacím jazykům Java, Python, C++ nebo Javascript. Jeho první verze byla vydána v únoru roku 2016. Byla založena společností JetBrains, která Kotlin představila již v roce 2011. Původním záměrem bylo vytvořit nový jazyk zaměřený na virtualní stroj Java (zkráceně JVM). Virtuální stroj Java je

prostředí, které umožňuje spouštět programy na libovolném operačním systému, který JVM podporuje. V roce 2019 společnost Google oznámila Kotlin jako preferovaný jazyk pro vývoj pro platformu Android. [14]

Požadavky na aplikaci od Google play

Pro nahrání aplikace na Google Play musí být dodržena určitá pravidla, která aplikace musejí dodržovat. [15] Mezi tato pravidla patří:

Zakázaný obsah – Aplikace musí být v souladu se zásadami obsahu a zákony. Nesmí obsahovat nelegální činnosti, obsah generovaný umělou inteligencí, hazardní hry či obsah, který není v souladu se zákony v místě využívání. [16]

Napodobování – Aplikace nejsou povoleny, pokud uvádějí uživatele v omyl tím, že se vydávají za někoho jiného (vývojáře, společnost, subjekt) či jinou aplikaci. [17]

Duševní vlastnictví – Pokud jsou porušeny práva duševního vlastnictví jiných osob, nejsou aplikace ani vývojářské účty povoleny. Dále nejsou povoleny aplikace, které navádí nebo podporují k porušování práv duševního vlastnictví. [18]

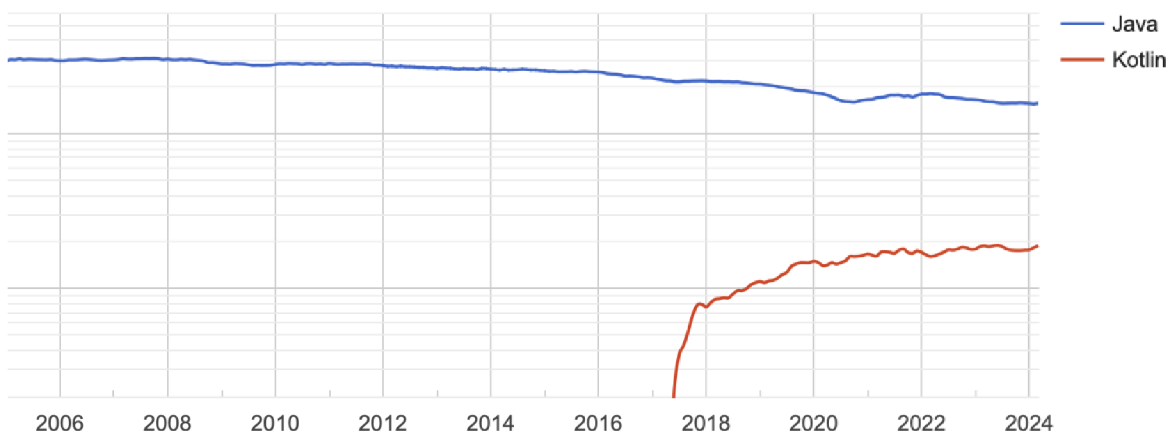
Ochrana soukromí a zneužití zařízení – Aplikace nesmí být škodlivé, podvodné nebo cíleně zneužívat jakoukoli síť či zařízení. [19]

Spam a minimální funkčnost – Aplikace musí nabízet alespoň základní funkčnost a respektující uživatelský zážitek. Pokud aplikace neodpovídá funkčnímu uživatelskému prostředí nebo slouží pouze k zasílání spamu uživatelům, bude aplikace vyřazena. [20]

Malware – Aplikace nesmí obsahovat jakýkoliv kód, který by mohl ohrozit uživatele nebo jeho zařízení. [21]

Nežádoucí software – Pokud aplikace obsahuje software, který porušuje zásady ekosystému Android a Google Play, bude pro ti ni zasaženo. [22]

Obrázek 3 Popularita Kotlin vs Java



Zdroj: [23]

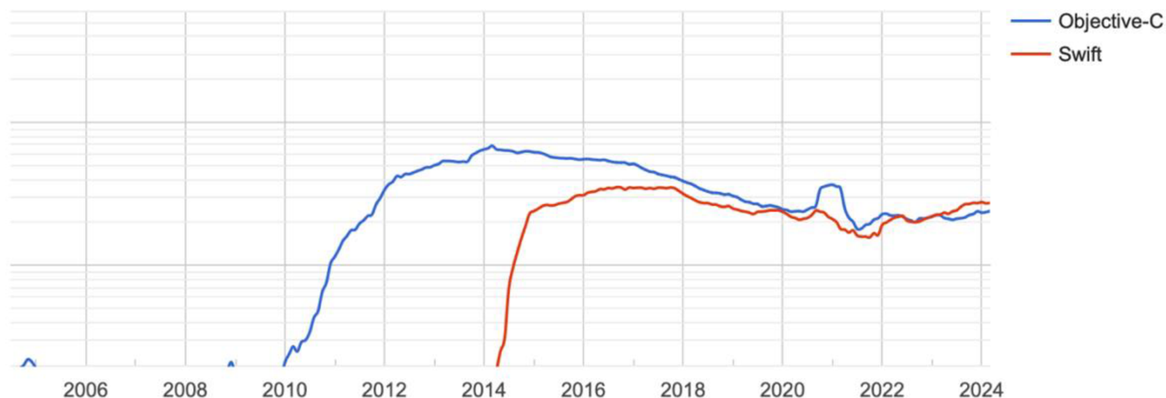
Nativní vývoj pro iOS

Vývoj programovacího jazyka Swift byl zahájen v roce 2010 Chrisem Lattnerem ve společnosti Apple. Společně s ním se na jeho vzniku podílel Doug Gregor, John McCall, Ted Kremenk a Joe Groff. Před uvedením iOS 8 byl stanoveným programovacím jazykem pro vývoj aplikací iOS jazyk Objective-C. V roce 2014 při oznámení iOS 8 byla představena alternativa k jazyku Objective-C v podobě programovacího jazyka Swift.

Díky popularitě systému iOS byl jazyk Objective-C jedním z nejpoužívanějších programovacích jazyků. Nicméně s původem zasazeným ve 40 let starém programovacím jazyce C, zůstal navzdory snahám o modernizaci některých aspektů syntaxe Objective-C projevovat svou zastaralost. [24]

Programovací jazyk Swift byl naproti tomu relativně novým jazykem, který byl navržen s cílem speciálně jak k usnadnění a zrychlení programování, tak i ke snížení náchylnosti k chybám. Syntaxe ovšem nezůstane neznámá těm, kteří ovládají jazyky jako jsou C, C++, Java nebo Kotlin.

Obrázek 4 Popularita jazyku Swift vs Objective-C



Zdroj: [23]

Požadavky na aplikaci od App Store

Bezpečnost pro uživatele – Aplikace nesmí obsahovat rozrušující či urážlivý obsah, nepoškodí jejich zařízení a její používání nezpůsobí fyzickou újmu uživateli.

Obchodní model – Aplikace musí mít jasně popsaný a vysvětlený svůj obchodní model v poznámkách recenze aplikace. Pokud bude popis nejasný nebo neúplný, aplikace může být zamítnuta.

Design – Pro schválení aplikace do obchodu App Store musí být dodrženy určité minimální standardy

- Design aplikace nesmí kopírovat design aplikace jiné
- Platí zákaz zahlcování App Store takzvanými „zbytečnými aplikacemi" které nepřinesou „jedinečný" zážitek

Právní požadavky – Aplikace musí splňovat veškeré zákonné požadavky v místě, kde je zpřístupněna. Veškeré aplikace, které nabádají k trestnému nebo násilnému chování, propagují je budou odmítnuty.

Beta testování – Každá aplikace musí projít beta testováním, než budou uvedeny na trh App Store pro systém iOS. Jedná se o jednu z hlavních fází vývoje pro produkty společnosti

Apple Inc. Součástí tohoto testování uživatelé beta verze kontrolují funkčnost aplikace se snahou nalézt možné chyby či problémy aplikace. Tímto způsobem vývojáři maximálně prověří aplikace a odstraní problémy, než je aplikace umístěna do App Store.

Bezpečnost aplikace – Operační systém iOS a jeho aplikace jsou známé svojí vysokou úrovní zabezpečení. Jednou z hlavních povinností vývojářů je správné šifrování dat, aby se zabránilo jejich ztrátě nebo jejich odcizením. [25]

Kompatibilita aplikace s různými verzemi systému iOS – Společnost Apple každoročně aktualizuje systém iOS, tudíž je nutné, aby zde existovala kompatibilita. Každá verze má svá technická vylepšení a nové funkce, proto musí vývojáři aplikací pro iOS aktualizovat aplikace takovým způsobem, aby odpovídaly modernizovaným standardům nejnovějších verzí. Vývojáři si musí vybrat různá zařízení se systémem iOS, aby zkontrolovali kompatibilitu aplikací, které následně testují a opravují. [26]

3.2.2.3 Cross-platform vývoj aplikací

Vývoj softwarových aplikací, které jsou kompatibilní s několika operačními systémy je označován jako cross-platform. Ačkoliv je nativní vývoj aplikací pro jednotlivé operační systémy finačně i časově náročný, byl často jednodušší, než cross-platform. Důvod byl ten, že kód vyvinutý pro jeden operační systém nebylo možné použít pro jiný operační systém. V současné době existuje již řada frameworků, které tento problém vyřešily a umožňují vývoj WORA (Write Once, Run Anywhere). [27]

Výhody cross-platform aplikací

- **Nižší náklady společně s rychlejším vývojem** – Jelikož pro všechny platformy je zapotřebí vytvořit pouze jednu verzi aplikace, může být proces vývoje dokončen mnohem rychleji. Místo toho, aby se vývojáři musely zabývat vývojem samotných aplikací pro každou platformu, mohou soustředit veškerý svůj čas na vytvoření jedné aplikace, která funguje na více platformách. Díky tomu se zkracuje jak doba vývoje, tak i celkové náklady na vývoj aplikace.

- **Snadná a levná údržba –** Synchronizace změn v aplikaci napříč platformami je díky jedné verzi aplikace časově i finančně výhodná.

- **Znovupoužitelný kód –** Díky použití jedné kódové základy napříč platformami je ušetřen čas vývojářů nad opakovanými úkoly jako je např. ukládání dat, volání API, serializace dat a implementace analytických funkcí. [27]

- **Možnost efektivně oslovit širší publikum –** Díky aplikaci, která funguje napříč systémy iOS, Android a web není nutné si vybírat mezi platformami. Díky kompatibilitě aplikace lze maximalizovat svůj dosah. [28]

Nevýhody cross-platform aplikací

- **Rychlost aplikace –** Díky potížím s interoperabilitou (Schopnost různých systému spolu vzájemně spolupracovat) mezi nativními a nenativními částmi zařízení může vést ke snížení výkonnosti aplikace. Tvůrci cross-platform aplikací musí strávit více času na zajištění jednotnosti aplikace.

- **Omezený přístup k nativním funkcím** – Některé funkce, které jsou specifické pro danou platformu nemusí být dostupné v nástrojích a frameworcích pro cross-platform vývoj, což může vést k omezení funkčnosti aplikace.

- **Obtížnější debugování aplikace** – Během procesu debuggování cross-platform aplikace může dojít k určitým problémům, například:
  - Chyby se objeví pouze na určité platformě
  - Emulátory nebo simulátory nemusí přesně odrážet výkon skutečných zařízení
  - Nástroje pro debuggování nemusí pro některé platformy k dispozici nebo mohou být omezené
  - Cross-platform aplikace bývají často rozsáhlé a mají složitou kódovou základnu

- **Možnost problémů s kompatibilitou** – Mezi nejčastější problémy patří :
    - Cross-platform aplikace nemusí na některých platformách dosahovat stejných kvalit jako nativní aplikace
    - Různé platformy mohou mít odlišné designové zásady a tudíž cross-platform nemusí poskytnout stejnou úroveň uživatelského komfortu jako nativní aplikace.
    - Cross-platform aplikace často využívají knihovny třetích stran, které mohou být kompatibilní pouze s některými platformami. [27]

Cross-platform programovací jazyky

React native

Nejrozšířenějším cross-platform jazykem před nástupem Flutter byl React Native. Stejně jako Flutter je i React Native open-source a obdobně jako u Flutter stojí za ním velká vývojářská společnost – Facebook (v současné době Meta).

Jedná se o oblíbený framework obzvláště proto, že využívá technologie webového frameworku React. Komunita React Native je velice aktivní a posouvá framework neustále dopředu.

React Native využívá programovací jazyk JavaScript pro vzhled aplikace a poté programovací jazyky Java nebo Swift pro psaní nativních modulů. Stejně jako Flutter nabízí React Native funkci hot realod, který umožňuje rychlý vývoj a iterace kódu aplikace. Z hlediska rychlosti má Flutter navrch. Jedním z předních důvodů je, že Flutter je kompilován do nativních knihoven, zatímco React Native má vrstvu JavaScriptu. [29]

Společnosti, které využívají React Native: Facebook, Instagram, Skype nebo Uber Eats. [28]

Přechod z React Native na Flutter

Flutter využívá pohledy reaktivního stylu, přičemž widgety jsou přirovnatelné ke komponentám React. Tato podobnost může pomoci vývojářům React Native přejmout aspekty stavů a funkcí setState, které jsou typické pro Flutter.

Klíčové rozdíly mezi jazykem JavaScript a Flutter:

- **Proměnné** – Dart je na rozdíl od JavaScript typově bezpečný jazyk – proměnné musí být s datovým typem
- **Nedefinované proměnné** – V jazyce Dart neexistuje koncept nedefinovaných proměnných. Proměnná buď hodnotu má nebo je nulová [29]

Kotlin Multiplatform

Kotlin Multiplatform je open-source technologie vytvořená společností JetBrains, která umožňuje vývojářům sdílet kód napříč platformami při zachování výhod nativního programování. Za použití tohoto SDK lze vytvářet aplikace pro Android, iOS, web, desktop a server.

Společnosti, které používají Kotlin Multiplatform: McDonald's, Netflix, Phillips a další. [28]

Cordova

Apache Cordova umožňuje spouštět mobilní aplikace za využití webových technologií HTML, CSS a JavaScript. Cordova je spíše platformou, která umožňuje provozování frameworků, kterým je například Ionic.

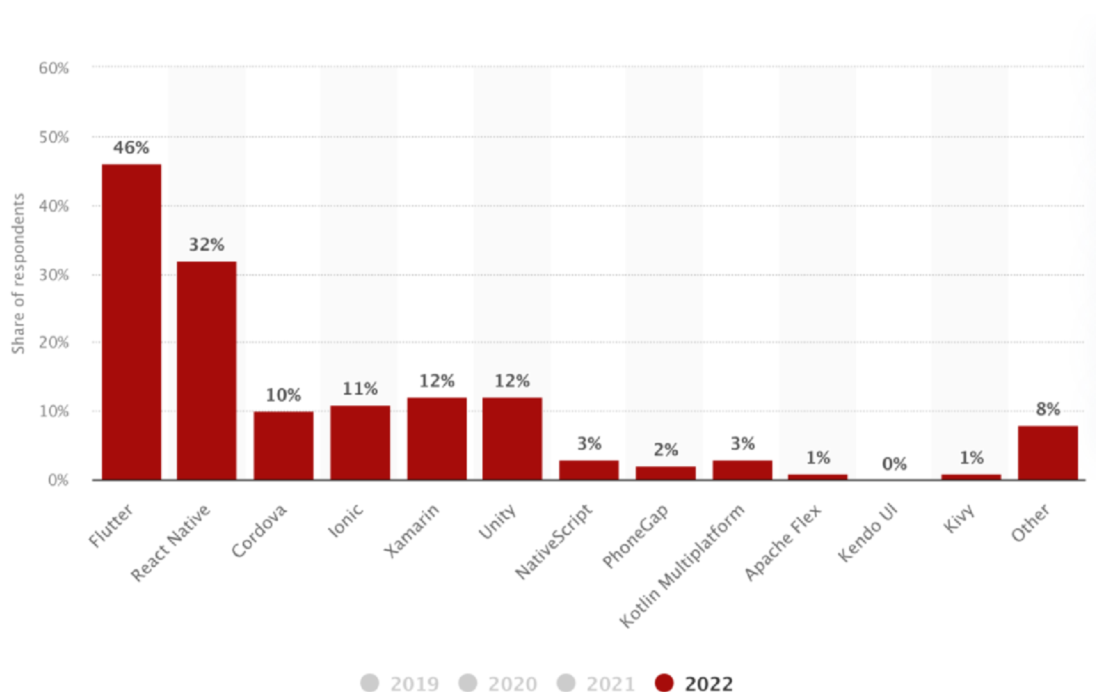
Aplikace Cordova funguje prostřednictvím WebView, který se dá popsat jako vestavěný prohlížeč pro každou specifickou platformu. Na rozdíl tedy od React Native a Xamarin je všechen kód cross-platform. Avšak hlavním problémem Apache Cordova je, že implementace WebView se může mírně lišit pro jednotlivé platformy, což může zapříčinit nekonzistenci a chyby v uživatelském rozhraní. Další nevýhodou jsou rozdílné WebView pro různé verze jedné platformy, tudíž rychlost aplikace a její uživatelské rozhraní mohou být odlišné. [29]

Xamarin

Obdobně jako u Flutter či React Native je Xamarin open-source a stojí za ním velká technologická společnost – Microsoft. Při využívání Xamarin Native mohou vývojáři využít výkonnostní výhody nativních aplikací, avšak bude mít k dispozici kód uživatelského rozhraní, které je specifický pro danou platformu. Ve výsledku je tedy zhruba 75% kódové základny sdílené, tudíž pro vývoj je nutná znalost jak Xamarin, tak i nativních jazyků.

Stejně jako Flutter nebo React Native nabízí Xamarin svým uživatelům funkci hot realod, která napomáhá rychlejšímu vývoji. Na rozdíl od předchozích frameworků je Xamarin placený. Komunita vývojářů není tak silná jako pro Flutter či React Native, což může limitovat podporu komunity a vývojáře jako takové. [29]

Obrázek 5 Nejvyužívanější technologie pro cross-platform vývoj



Zdroj: [30]

3.3 Dart

Programovací jazyk Dart byl představen společností Google koncem roku 2011. Je designovaný především pro vývoj mobilních a webových aplikací, avšak dá se využít k tvorbě serverových a desktopových aplikací.

Syntaxe jazyku Dart je podobná programovacímu jazyku C, tudíž je snadno uchopitelná pro programátory, kteří mají zkušenosti s jazyky, kterými jsou například: C++, C# nebo Java. Byl vyvinut s cílem vyřešit nedostatky programovacího jazyka Javascript, proto vývojáři Dartu přidali navíc generiky, či statické typování, které jazyku JavaScript chybí.

V roce 2013 se stal z Dartu open source programovací jazyk. Tento krok pomohl vývoji kupředu, díky opravení chyb a vytvaření nových knihoven komunitou vývojářů.

Jedním z klíčových okamžiků pro jazyk Dart bylo uvedení SDK Flutter. Úspěch Flutter významně přispěl k nárůstu popularity Dartu a to zejména v oblasti vývoje cross-platform mobilních aplikací. [31]

3.3.1 Platformy Dart

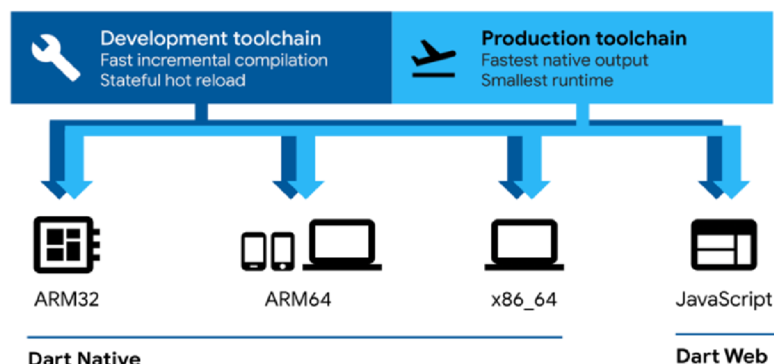
K porozumění flexibilitě, kterou nabízí programovací jazyk Dart je důležité znát možnosti, jakým způsobem lze kód Dart spustit.

- **Nativní platforma** – Pro aplikace, které jsou určené na mobilní nebo stolní zařízení obsahuje programovací Jazyk Dart jak virtuální počítač Dart JIT (Just-in-time), tak překladač kompilátor pro tvorbu strojový kód
- **Webová platforma** – Pro webové aplikace lze kompilovat pro vývojové nebo produkční účely. Kompilátor převádí jazyk Dart do jazyka JavaScript [32]

Kompilace Just-in-time – Při variant kompilace JIT je zdrojový kód kompilován v okamžiku, kdy je potřeba. Virtuální počítač Dart načte a zkompiluje zdrojový kód do nativního strojového kódu za běhu. Tento přístup je využíván ke spouštění kódu v příkazovém řádku nebo během vývoje mobilních aplikací, kdy zajišťuje funkce hot reaload.

Kompilace Ahead-of-time – Při kompilaci AOT je Dart VM a kód předkompilován a funguje jako Dart runtime systém, který poskytuje garbage collector a další nativní metody z Dart SDK. Tento přístup je oproti JIT kompilaci výrazně rychlejší, avšak nenabízí funkce jako debugging a hot reload. [29]

Obrázek 6 Platformy programovacího jazyku Dart



Zdroj: [32]

Virtuální počítač Dart a kompilace do JavaScript

Spuštění kódu Dart může proběhnout pouze v prostředí, které Dart spuštění umožňuje. Takové prostředí poskytuje důležité prvky aplikaci, kterými jsou:

- Runtime systémy
- Knihovny Dart
- Garbage collectors

Spuštění kódu Dart probíhá ve dvou režimech – kompilace **JIT** (Just-in-Time) a kompilace **AOT** (Ahead-of-time).

Běhové prostředí Dart – Bez ohledu na využívanou platformu nebo způsob kompilace kódu, vyžaduje spuštění kódu běhové prostředí Dart.

Běhové prostředí je zodpovědné za:

- **Správa paměti** – Programovací jazyk Dart využívá model spravované paměti, kde nevyužitá paměť je získávána zpět za pomoci Garbage Collectoru
- **Vynucení typového systému Dart** – Ačkoliv většina typových kontrol probíhá v době kompilace (statická), některé typové kontroly probíhají za běhu (dynamické)

Běhové prostředí Dart je na nativních platformách obsaženo uvnitř spustitelných souborů a je součástí virtuálního počítače Dart. [32]

Garbage collector – Jedná se o techniku správy paměti, při které dochází k automatické identifikaci a zpětnému získání paměti, která již programem není využívána. Hlavním cílem je uvolnění paměti obsazenou objekty, které již nejsou dosažitelné nebo již nejsou zapotřebí. Tímto způsobem se zabraňuje únikům paměti a je zajištěno efektivní využití systémových prostředků. [33]

3.3.2 Objektově orientovaný jazyk

Objektově orientovaného programování (dále jen OOP) umožňuje snadné modelování návrhu a vývoje entit. OOP umožňuje programátorům definovat třídy k vytváření objektů a provádět změny na těchto třídách. Tento přístup stojí na třech základních pilířích: dědičnost, polymorfismus a zapouzdření. Díky možnostem, které OOP nabízí patří toto paradigma mezi nejpoužívanější.

Dědičnost – Koncept dědičnosti poukazuje na opětovné použití kódu, bez jeho přepisování nebo opakování. Vytváří třídy z již existujících tříd, které jsou nazývány odvozené třídy nebo podtřídy. Třídy ze které bývají podtřídy odvozeny se označují jako rodičovské třídy či natřídy. Odvozená třída znovu používá členy rodičovské třídy, které má možnost měnit nebo přidávat další. Díky těmto vlastnostem se programátoři mohou vyhnout opětovnému přepisování již existujícího kódu.

Zapouzdření – V okamžiku vytvoření objektu není pro jeho použití nutná znalost implementace. Díky tomu je programátorovi zabráněno v manipulaci s hodnotami, které by měly zůstat neměnné. Datové struktury definované lokálními proměnnými uvnitř objektu jsou skryty. Přístupovaní k lokálním proměnným objektu je umožněno pouze prostřednictvím operací.

Abstrakce – V OOP abstrakce představuje modely z pohledu rozhraní a funkčnosti, bez implementačních detailů. Bývá použita k zjednodušení návrhu a implementace, která je zaměřena na význam chování. [34]

Polymorfismus – Jedná se o schopnost odlišných objektů odpovídat na stejnou zprávu různým způsobem. Objektově orientované programovací jazyky umožňují definovat jednu

nebo vice metod se stejným názvem, které mohou vykonávat různé operace a vracet různé hodnoty. [35]

Třída – Jedná se o soubor objektů obdobného typu. Po vytvoření dané třídy lze vytvořit libovolný počet objektů, které do této třídy patří. Třída seskupuje objekty, které mají společné chování. Objekty (mohou být označovány jako instance) v sobě mohou obsahovat specifické vlastnosti v podobě jedinečných hodnot vlastních lokálních proměnných.

Objekt – jsou samostatné entity, které reagují na zprávy, kde zpráva představuje požadavek na objekt, aby vykonal jednu z jeho operací. Operace objektu představují soubor poskytovaných a použitelných funkcí pro daný objekt. Každý objekt poskytuje zapouždření. [36]

3.4 Flutter

Technologie Flutter představuje přenositelný framework od společnosti Google, který slouží pro vytváření moderních, nativních a reaktivních aplikací pro platformy Android a iOS. Jedná se o open source projekt umístěný na GitHub s příspěvky jak od společnosti Google, tak od komunity vývojářů. Tento framework využívá programovací jazyk Dart, který je kompilován do nativního kódu ARM a kódu JavaScript. Využívá vykreslovací jádro Skida2D, které využívá různých hardwarových a softwarových platforem. [37]

3.4.1 Výhody Flutter Development

Uživatelské rozhraní – Flutter nabízí širokou škálu přizpůsobitelných widgetů, které lze použít k vytvoření uživatelsky přívětivých rozhraní. Tento framework klade velký důraz na design a vizuální stránku, tudíž je vhodnou volbou pro vývoj aplikací, kde je dbáno na vysokou míru vizuální prezentace.
Rychlost – Tento framework je navržen tak, aby plynule běžel i na starších zařízení. Je optimalizován pro výkon, tím pádem je vhodnou volbou i pro náročnější aplikace.
Cross-platform – Jak již bylo zmíněno, Flutter podporuje nejen vývoj mobilních aplikací, ale také desktopových a webových aplikací. To z něj činí univerzální nástroj pro vývoj aplikací, které jsou navrženy pro různé zařízení.

Open-source – Flutter je bezplatný a open source, tudíž je přístupný širokému spektru vývojářů a firem. Rozsáhlá komunita vývojářů a uživatelů, kteří s tímto nástrojem pracují pomáhá zajistit jeho další vývoj. [38]

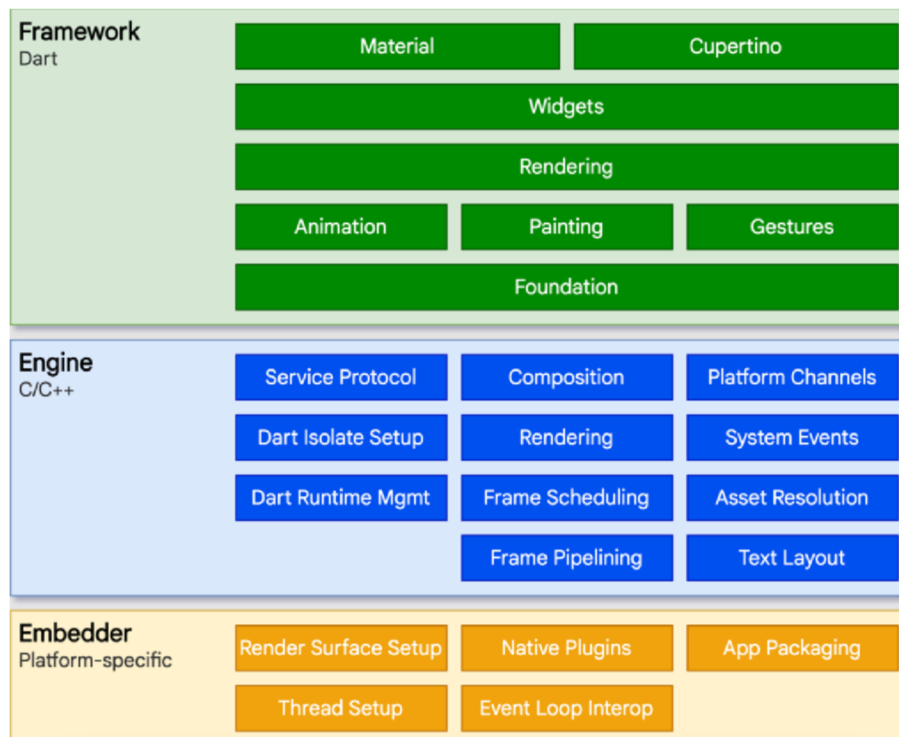
Funkce hot reaload – Jednou z nejpopulárnějších vlastností Flutter je možnost hot reload, která umožňuje vývojáři vidět změny na zařízení ihned po tom, co byly provedeny v kódu. Tato funkce je velmi přínosná, jelikož zkracuje dobu mezi kódem a zpětnou vazbou.

Rychlost vývoje – Vývojový cyklus aplikace umožňuje vidět vývojářům změny v aplikaci v reálném čase. Tato funkce výrazně zvyšuje efektivitu a rychlost vývoje. [29]

3.4.2 Architektura Flutter

Při vývoji fungují aplikace na virtuálním zařízení, které umožňuje funkci „hot reload“. Aplikace Flutter jsou kompilovány přímo do strojového kódu instrukcemi Intel x64 nebo instrukcemi ARM. Pro webové aplikace jsou kompilovány do programovacího jazyku Javascript.

Obrázek 7 Vrstvy Flutter SDK



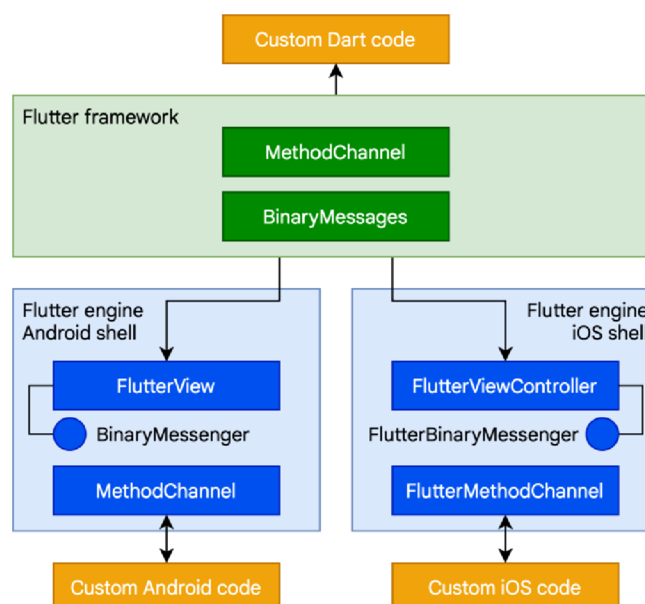
Zdroj: [13]

Flutter aplikace jsou pro operační systém na kterém fungují zabaleny stejným způsobem jako nativní aplikace. Embedder (vkládací modul) pro danou platformu představuje vstupní bod ke službám jako jsou renderování ploch, přístupnost a další. Tento modul je pro operační systém Android napsaný v jazycích Java a C++, pro iOS a MacOS Objective C/Objective C++ a v programovacím jazyku C++ pro Windows a Linux. Pomocí embedderu lze integrovat kód Flutteru do stávající aplikace jak modul nebo může být samostatným obsahem aplikace. Jádrem aplikace Flutter je stejnojmenný engine „Flutter", který je napsán převážně v jazyce C++. Tento engine má na starost rastrování kdykoliv, kdy je potřeba vykreslit novou obrazovku. Poskytuje nízkoúrovňovou implementaci základního API rozhraní včetně grafiky, přístupnosti, rozvžení textu, souborového vstupu/výstupu, architektury pluginů a také běhového/kompilačního řetězce nástrojů Dart. [13]

3.4.3 Platformové kanály

U mobilních a desktopových aplikací Flutter umožňuje volat vlastní kód prostřednictvím takzvaných „platform channels". Vytvořením společného kanálu je umožněno odesílat a přijímat zprávy mezi jazykem Dart a platformovou komponentou vytvořenou v jazyce Swift nebo Kotlin. Data jsou serializována jako mapy a poté deserializovány do ekvivaletní reprezentace v jazyce Kotlin (např. Hashmap) nebo Swift (např. dictionary). [13]

Obrázek 8 Platformové Kanály Flutter



Zdroj: [13]

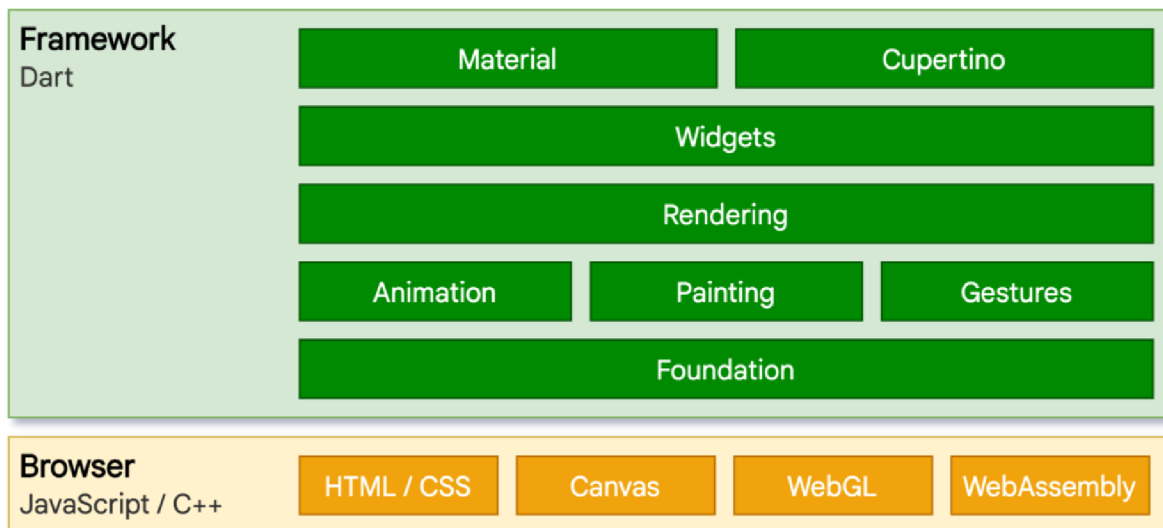
3.4.4 Flutter web

Již od svého vzniku se kompiluje programovací jazyk Dart do JavaScript (viz kapitola 3.3). Mnoho současných aplikací se dnes kompiluje z Dart do Javascript v produkčním provozu – jedním z příkladů může být Google Ads.

Flutter engine je napsaný v jazyce C++ a je navržený především pro komunikaci s operačním systémem než s webovým prohlížečem. Pro komunikaci s web poskytuje Flutter reimplementaci svého enginu nad standardními rozhraními API prohlížeče. V současné době existují dva způsoby vykreslování obsahu Flutter na webovém prohlížeči:

- **HTML** – V prvním případě je využíváno HTML, CSS, Canvas a SVG.
- **WebGL** – V druhém je využívána Flutter verze Skia zkompilovaná do WebAssembly s názvem CanvasKit.

Obrázek 9 Architektura Flutter web



Zdroj: [13]

Oproti jiným platformám, na kterých Flutter funguje, nemusí zde být poskytováno běhové prostředí Dart. Flutter je kompilován do programovacího jazyka JavaScript. [13]

Flutter DevTools – Představuje sadu nástrojů pro vyhledávání chyb a sledování výkonnosti aplikace.

K čemu lze DevTools použít:

- Kontrola rozložení uživatelského rozhraní a stavu aplikace Flutter
- Diagnostikování výkonnostních problémů aplikace
- Profilování procesoru aplikace
- Řešení problémů s pamětí aplikace
- Analýza kódu a velikost aplikace [13]

3.4.5 Widget

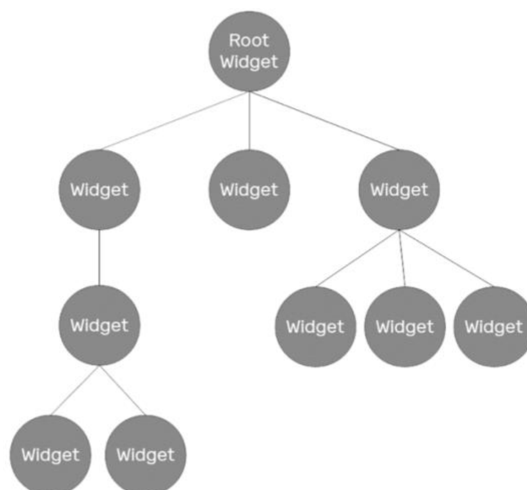
Jednotlivé widgety popisují, jak by mělo vypadat jejich zobrazení vzhledem k jejich aktuální konfiguraci a vztahu. Pokud se stav widgetu změní, znovu vytvoří svůj popis, který je porovnán s předchozím popisem, aby určil změny potřebné v renderovaním stromu pro přechod z jednoho stavu do druhého. Aplikace je tvořena widgety vnořenými mezi sebe. Widget může zobrazovat data, určovat design či zpracovávat interakci s uživatelem. [39]

Stav – Stav aplikace je situace či podmínka, kdy instance odpovídá podmínce dané aplikace v určitý čas.

Důležitost stavů v aplikaci – Stav může být označován za základní stavební kámen aplikace. Definuje chování aplikace od jejího otevření po její ukončení. Každá změna, která je uživateli zobrazena je stav, který má být v danou chvíli zobrazen. [40]

Widget tree (Strom widgetů) – Jedná se o logickou reprezentaci všech widgetů uživatelského rozhraní. Je vypočítán při rozvržení a používá se pro vykreslení. Widgety ve stromu jsou reprezentovány jako uzly a každý z těchto widgetů může mít svůj stav. Pokud dojde ke změně jeho stavu, dojde k přestavbě widgetu a jeho potomků. [29]

Obrázek 10 Strom widgetů



Zdroj: [29]

Stateful Widget

Stavový widget je takový widget, který má proměnlivý stav. Stav představuje informaci, kterou lze číst při sestavení widgetu a která se může měnit v průběhu jeho životnosti. Tento druh widgetů je využíván v případech, kdy se může popisovaná část uživatelského rozhraní měnit. Dané změny lze provést pomocí metody **setState()**. [41] Tento přístup je nazýván deklarativní, což znamená, že uživatelské rozhraní je vytvářeno na základě aktuálního stavu aplikace. [42]

Obrázek 11 Deklarativní přístup Flutter

Zdroj: [42]

Stateless Widget

Bezstavový widget popisuje část uživatelského rozhraní vytvořením konstelace dalších widgetů, které popisují uživatelské rozhraní konkrétněji. Tento proces budování probíhá rekurzivně, dokud není popis uživatelského rozhraní konkrétní. Widgety, které nemají žádné stavy jsou užitečné, pokud popisovaná část uživatelského rozhraní nezávisím na ničem jiném než na konfiguračních informacích v samotném objektu. [43]

V případě, že Flutter potřebuje sestavit určitou část kódu, zavolá metodu build, která vrací podstrom widgetů, které vykreslí uživatelské rozhraní na základě současného stavu aplikace. Během tohoto procesu mohou být představeny nové widgety na základě svého stavu.

Během fáze sestavování widgetů, Flutter převádí kód do daného „stromu elementů“ s jedním elementem pro každý widget. Každý element reprezentuje určitou instanci widgetu s danou pozicí v hierarchii stromů.

Ve Flutter existují dva základní typy elementů:

ComponentElement – hostitel pro další prvky

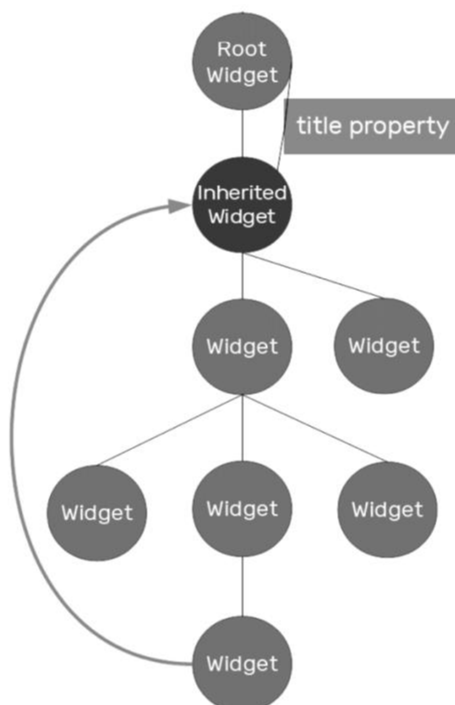
RenderObjectElement – prvek, který se podílí na fázích rozvržení nebo vykreslení. Tyto prvky jsou prostředníkem mezi svou obdobou widgetu a základním objektem **RendeObject**.

Strom prvků je mezi stavy stálý a hraje zásadní výkonnostní roli. Díky tomu může Flutter pracovat s hierarchií widgetů jako by byla jednorázová a ukládat do mezi paměti její původní reprezentaci. Flutter obnovuje pouze ty části stromu prvků, které byly změněny a vyžadují rekonfiguraci. [29]

Inherited Widget

Kromě Stateless a Stateful widgetů existuje v prostředí Flutter ještě typ widgetu s názvem Inherited. Může nastat situace, kdy jeden widget vyžaduje přístup k datům, která se vyskytují hlouběji ve stromu widgetů. V takových případech je jedním z řešení replikovat informace dolů k příslušenému widgetu přes všechny mezilehlé widgety. [29]

Obrázek 12 Inherited widget



Zdroj: [29]

Přidání Inherited widgetu do stromu widgetů zajišťuje možnost každému následujícímu widgetu níže ve stromu přistupovat k datům, která vystavuje. [29]

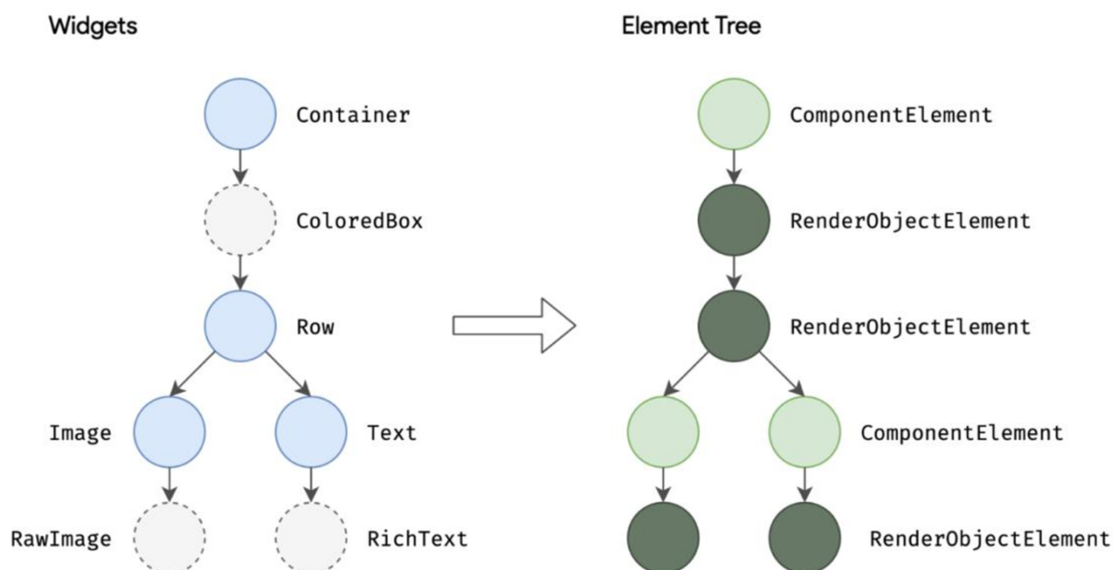
Rozvržení a vykreslování

Pro uživatelské rozhraní každého Frameworku je důležité efektivní rozvržení hierarchie widgetů a schopnost určit pozici a velikost jednotlivých prvků ještě před jejich vykreslením.

Základem pro každý uzel ve vykreslovacím stromu je RenderObject, který definuje abstraktní model pro vykreslování a rozvržení. Každý RenderObject zná svého rodiče, ale o svých potomcích zná pouze jejich omezení a cestu k nim.

Během fáze sestavování widgetů Flutter vytvoří nebo aktualizuje objekt, který dědí od RenderObject, pro každý RenderObjectElement ve stromu prvků (viz. Obrázek níže).

Obrázek 13 Widgety a strom prvků

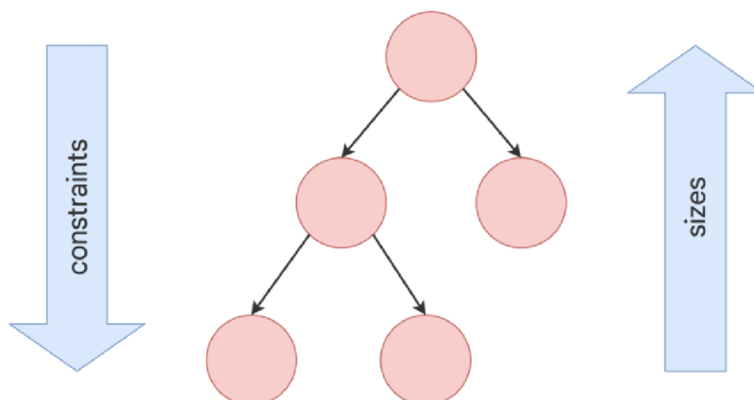


Zdroj: [13]

Za účelem rozvržení prochází Flutter strom vykreslování a předává omezení z rodiče na potomka. Potomek musí dodržovat omezení, která jsou stanovena jeho rodičem. Potomci poté předávají velikost svému rodičovskému objektu v rámci omezení. Stejně jako u velikostí, pozice potomka je také určována rodičem. [29]

Pozice a velikost rodiče závisí také na jeho vlastním rodiči, nelze tedy přesně definovat velikost a pozici žádného widgetu bez zohlednění stromu jako celku. [44]

Obrázek 14 Omezení a velikost widgetů



Zdroj: [13]

3.4.6 Flutter state management

State management (správa stavu) představuje technologii nebo kombinaci technologií, které jsou využívány k manipulaci se změnami v aplikaci. V případě, že aplikace nabere větších rozměrů je potřeba zajistit přístup se kterým lze sledovat změny v aplikaci a patřičně na ně reagovat.

Správa stavu může zahrnovat následující činnosti:

- Reakce na interakci uživatele
- Pozorování dat na různých obrazovkách aplikace
- Změna dat v aplikaci na jednom místě a reakce na patřičné změny na ostatních obrazovkách

Provider – Jedná se o zjednodušení přístupu InheritedWidget, díky kterému lze zajistit stejnou funkcionalitu s menším množstvím kódu. Použitím tohoto přístupu není potřeba vytvářet samostatnou třídu InheritedWidget a mezitřídu, která v sobě obsahuje funkci setState.

Riverpod – Balíček Riverpod představuje vylepšený přístup technologie Provider. Využívá koncepty balíčku Provider, avšak poskytuje lepší flexibilitu a rychlost. Jedním z hlavních problémů, které Riverpod vyřešil je zamezení načítání objektů s nulovou hodnotou ze stavů. Díky tomu je při čtení objektů pomocí funkcí **read** nebo **watch** zajištěno, že vrácená hodnota nebude **null**.

BLoC – Přístup BLoC využívá toky ke sdílení informací o stavu. Jednotlivé widgety naslouchají vybraným tokům stavových informací a při upozornění o změně stavu na ni reagují. Tento přístup zapadá do deklarativního přístupu Flutter. Pokud se změní stav aplikace, je o této změně widget informován a může se rozhodnout o aktualizaci svého vnitřního stavu, pokud je tato změna stavu aplikace relevantní. Pokud dojde k aktualizaci vnitřního stavu, spustí se funkce **builder**, která vykreslí změny uživateli. [40]

3.4.7 Flutter doplňky

Flutter podporuje využívání sdílených doplňků, které pro ekosystém Dart a Flutter vytváří další vývojáři. Díky těmto doplňkům lze vytvářet aplikaci rychleji, aniž by bylo nutné vytvářet vše od začátku. Flutter doplňky se dělí na balíčky a pluginy.

Flutter balíček – Balíček představuje adresář obsahující soubor pubspec.yaml. Mimo jiné může balíček obsahovat závislosti, knihovny Dart, aplikace, testy, obrázky, písma a další. Tyto balíčky lze získat na webových stránkách pub.dev.

Pubspec.yaml – Tento soubor je součástí Flutter a jedná se o soubor, kde jsou definovány jednotlivé balíčky. Mimo jiné obsahuje také sekci, která definuje konfiguraci Flutter. [29]

Flutter plugin – Jedná se o speciální balíček, který aplikaci zpřístupňuje funkce dané platformy. Balíčky pluginů mohou být napsané pro iOS (pomocí jazyka Swift nebo Objective-C), pro Android (pomocí jazyka Java nebo Kotlin), web, macOS, Windows, Linux. Příkladem využití pluginu může být například zpřístupnění fotoaparátu aplikaci. [45]

3.4.8 Testování

S rostoucím počtem funkcí, které aplikace obsahuje je čím dál tím obtížnější jí manuálně otestovat. Automatizované testy pomáhají zajistit správné fungování aplikace a současně při tom zachovat rychlost vývoje funkcí a opravy chyb.

Automatizované testování se dělí do tří kategorií:

Unit testy – Testování jedné funkce, metody nebo třídy. Účelem unit testů je ověření správnosti logické jednotky za různých podmínek. Vnější závislosti testu jsou obvykle simulovány.

Widget testy – Testování jednotlivých widgetů (Pro jiné UI Frameworky označované jako komponentové testy). Cílem widget testů je ověřit, zda widget funguje podle očekávání a zároveň odpovídá uživatelskému rozhraní. Widget testy by měly být schopné reagovat na

akce a události, provádět rozvržení a zpracovávat podřízené widgety. Stejně jako u unit testů je prostředí simulováno – nahrazeno implementací, která je podstatně jednodušší než uživatelské rozhraní celé aplikace.

Integrační testy – Testování celé aplikace nebo její určité části. Hlavním cílem integračního testu je ověření správnosti fungování všech testovaných widgetů a služeb podle očekávání. Integrační testy mohou být také využity k testování výkonu aplikace. Tento druh testování probíhá na skutečném zařízení nebo na emulátoru daného operačního systému.

Obecně platí, že kvalitně otestovaná aplikace má prošla velkým počtem unit testů a widget testů, které jsou monitorovány pomocí code coverage (pokrytí testů) a také integračních testů, které pokrývají všechny důležité případy použití. [46]

3.5 UI/UX design

Při vývoji digitálních produktů hraje uživatelského rozhraní a uživatelského prožitek zásadní roli. Tento design napřímo ovlivňuje způsob, jakým uživatelé s daným produktem komunikují a jak s ním pracují. Dobře navržené uživatelské rozhraní činí digitální produkt vizuálně přitažlivý, intuitivní a snadno ovladatelný. Kvalitní uživatelský prožitek zajišťuje pozitivní zkušenosti uživatele od objevení produktu po jeho používání a vyhledávání podpory. Úspěšný UI/UX návrh může zvýšit spokojenost a loajalitu uživatelů k používání digitálního produktu, což vede k vyšší míře konverze a příjmů podniku. Z těchto důvodů je důležité považovat návrh UI/UX za jeden z kritických aspektů při vývoji digitálního produktu. [47]

3.5.1 User Interface (UI)

Návrhem uživatelského rozhraní se označuje proces navrhování interaktivních a vizuálních prvků digitálního produktu jakými jsou webové stránky nebo mobilní aplikace. Zahrnuje vytvoření takového rozhraní, které je pro uživatele atraktivní po vizuální stránce, nabízí pozitivní uživatelský zážitek a snadno se používá.
Důležitost návrhu uživatelského rozhraní pro úspěšný vývoj digitálního produktu:

- **Uživatelský prožitek** – Správně navržené uživatelské rozhraní zajišťuje snadnou orientaci a plnění úkolů v rámci digitálního produktu, což vede k pozitivní uživatelské zkušenosti.
- **Branding** – Návrh rozhraní je důležitou součástí brandingu. Představuje uživateli charakter produktu a jeho hodnoty, díky kterým je rozpoznatelný a zapamatovatelný.
- **Efektivita** – Efektivní uživatelské rozhraní může zvýšit efektivitu uživatelů snížením potřebného času a úsilí k dokončení úkolů.

Klíčové zásady návrhu uživatelského rozhraní:

- **Jednoduchost** – Důležitým aspektem uživatelského rozhraní je dodržení jednoduchého a přímočarého designu. UI by mělo být jednoduše pochopitelné a přehledné s minimem rušivých prvků.
- **Konzistence** – Konzistentní návrhy vytváří u uživatelů pocit známosti. Prvky konzistetního návrhu mohou být například: barvy, typografie a rozvržení.
- **Zpětná vazba** – Zpětná vazba pomáhá uživatelům porozumět důsledkům jejich akcí. Zpětná vazba může mít reprezentována například: animací, zvukovými efekty nebo vizuálními signály.
- **Návrh zaměřený na uživatele** – Uživatelské rozhraní by mělo klást důraz na potřeby a preference uživatele. Návrh by měl být přizpůsoben cílové skupině a uživatelská zkušenost by měla být přístupná všem uživatelům, včetně těch se zdravotním postižením. [47]

3.5.2 User Experience (UX)

Návrh uživatelského prožitku je proces, který se zaměřuje na preference, chování a potřeby uživatele. Cílem UX je vytvoření bezproblémového a intuitivního prostředí, které odpovídá očekáváním a potřeb uživatele. Tento proces zahrnuje identifikování problémových míst, pochopení motivací a cílů uživatele a vytvoření návrhu, který tyto potíže odstraní.

K čemu přispívá kvalitní UX design:

- **Zlepšení použitelnosti** – Účelem UX designu je vytvoření produktu, který se snadno používá, což snižuje kognitivní zátěž uživatele a usnadňuje dosažení jeho cílů. To může mít vliv na spokojenost a loajalitu zákazníků.
- **Vnímání značky** – Pozitivní zkušenost uživatele s digitálním produktem může zlepšit vnímání značky.
- **Snížení nákladů** – Zaměřením návrhu na uživatele lze identifikovat potenciální problémy dříve v procesu vývoje, čímž se snižují náklady na pozdější opravu těchto problémů.
- **Konkurenční výhoda** – Kvalitní uživatelská zkušenost může odlišit produkt nebo službu od konkurence a získat tím konkurenční výhodu

Uživatelské testování je jedním ze základních stavebních kamenů návrhu UX, jelikož poskytuje důležité informace o chování, preferencích a potřebách uživatelů.

Proč je výzkum a testování potřeb uživatele tak důležitý:

- **Porozumění uživateli** – Výzkum zaměřený na uživatele poskytne důležité informace o uživateli, které pak napomáhají designérům vytvořit návrh, který potřeby uživatele naplňuje. Bez porozumění potřebám uživatele může být vytvořen návrh, který není v souladu s cíli nebo preferencemi uživatele.
- **Identifikace slabých míst** – Průzkum uživatelů může odhalit slabá místa v uživatelské zkušenosti, což umožňuje návrhářům tato místa v návrhu řešit. Odstraněním těchto míst zlepšuje uživatelskou zkušenost a spokojenost s produktem.
- **Omezení předpokladů** – Díky uživatelskému průzkumu lze omezit předpoklady o preferencích a chování uživatelů, což umožňuje vytvoření přesnějšího a efektivnějšího návrhu. Bez tohoto přístupu se musí návrháři spoléhat na předpoklady nebo vlastní intuici, což může vést k návrhu, který není v souladu s potřebami uživatele. [47]

Informační architektura

Informační architektura představuje organizaci a strukturování informací, které mají uživatelům pomoci k pochopení, kde se zrovna nacházejí, co našli a co mohou očekávat. Tento přístup zahrnuje návrh a organizaci aplikací, webových stránek či intranetů tak, aby bylo dosaženo co nejvyšší použitelnosti. [48] Stejně jako u všech aspektů návrhu UX, začíná informační architektura pochopením uživatelů. Jakmile je uživatelské chování pochopeno, lze navrhnout úspěšnou mapu navigaci aplikace nebo uživatelské flow. [49]

Wireframe

Představuje obrázek či prototyp jedné obrazovky či interakční stránky. Jedná se o dvoudimenzionální skeče nebo kresby, které reprezentují rozvržení dané obrazovky. [49]
Typy wireframe:

- **Low-fidelity** – Jedná se o základní wireframe zaměřený na rozvržení, navigaci a informační architekturu
- **Mid-fidelity** – Představuje konečný návrh před přidáním komponentů vizuálního designu
- **High-fidelity** – Definuje konečný prototyp s interaktivními a vizuálními prvky návrhu. V této fázi procesu lze přidat loga, barvy, styly písma apod. Tímto způsobem je možné zachytit konečný dojem a vzhled produktu pro uživatelské testování [50]

3.6 Unified modeling language (UML)

Nedílnou součástí vývoje software je mimo jiné také UML, který je využíván ve fázi analýzy a slouží k identifikaci a specifikaci požadavků. Dále je také využíván k modelování procesů a struktury systému.

Jedná se o specifikační jazyk a může být definován jako jazyk obecného určení, který využívá grafické rozhraní, jehož pomocí lze vytvořit abstraktní model. Tento abstrakční model se nazývá UML. Za definici jazyka je zodpovědné konsorcium Object Management Group (OMG). UML se běžně využívá k vizualizaci a vytváření komplikovaných systémů.

UML celkově obsahuje 14 druhů diagramů, avšak bude popsán pouze jeden z nich z důvodu jeho využití v praktické části. Diagramy se dělí na diagramy struktury a chování pod kterou spadá samostatná skupina diagramů interakce. [51]

Diagram tříd – Tento diagram spadá do skupiny diagramů struktury, kde třída představuje charakteristiku pro množinu objektů, které sdílejí určité atributy či operace. Jednotlivé třídy a objekty jsou přirozeným pojetím světa okolo nás. Koncept objektů a tříd jsou také paradigmaty tvořící základ objektově orientovaného programování. Rozvoj OOP si vyžádal vznik jazyku pro popis objektově orientovaného návrhu, čím vznikl UML. [52]

4 Praktická část

Díky získaným znalostem z teoretické části práce je nyní možné zaměřit se na návrh a implementaci prototypu mobilní aplikace za použití technologie Flutter. Praktická část bude rozdělena na dvě hlavní kapitoly.

V první kapitole bude představen návrh aplikace, který se bude skládat z návrhu uživatelského rozhraní pomocí diagramu informační architektury a wireframe. Poté bude vytvořen diagram tříd, který definuje struktury a vztahy mezi jednotlivými třídami.

Druhá kapitola praktické části bude zaměřena na samotnou implementaci prototypu aplikace. Tato implementace bude popsána detailním způsobem, aby přiblížila problematiku práce s technologií Flutter. Na závěr proběhne testování aplikace pomocí způsobů, které byly představeny v teoretické části (kapitola 3.4.8).

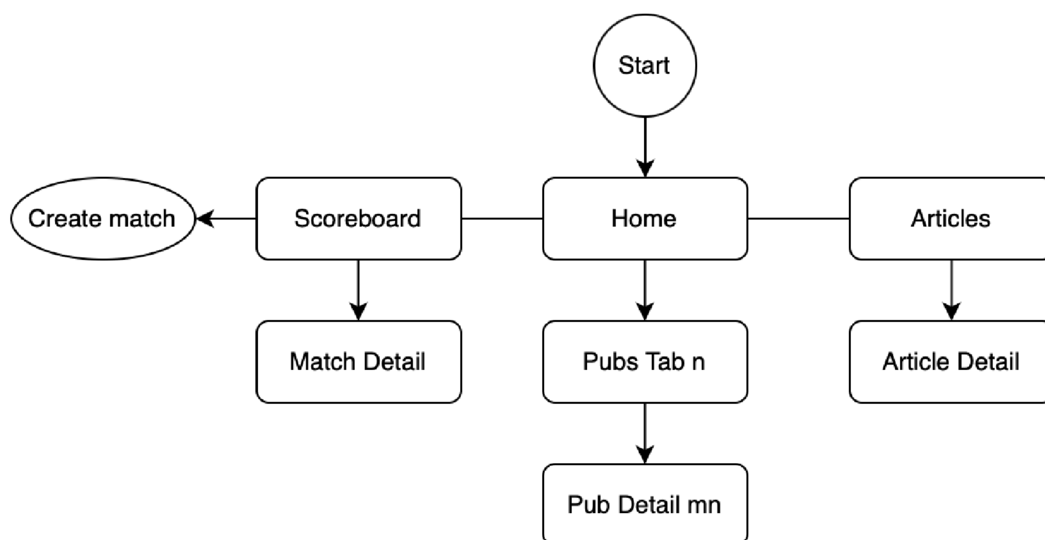
Pro demonstraci vývoje aplikace za použití Flutter byla vymyšlena mobilní aplikace, která má sloužit jako databáze podniků, které disponují stolním fotbálkem. Aplikace má uživateli umožnit zobrazení seznamu hospod s fotbálky. Dále také aplikace nabízí možnost založit si nový zápas. Zápas stolního fotbálku má svůj „stadion“ (podnik, ve kterém zápas proběhl) a je tvořen ze dvou týmů, kde každý tým má svůj název a je tvořen dvěma hráči. Po vytvoření zápasu je uživateli umožněno spravovat zápas – aktualizovat skóre daného zápasu podle výher a zaznamenávat si tak úspěšnost svých her, ke kterým se může zpětně vrátit. Poslední částí aplikace jsou články, kde se může uživatel dočíst novinek ze světa stolních fotbálků, které jsou „publikovány“ jednotlivými podniky.

4.1 Návrh aplikace

Návrh aplikace je zaměřený na vytvoření návrhu uživatelského rozhraní pomocí diagramu informační architektury, vytvoření wireframe pro jednotlivé obrazovky a diagramu tříd.

4.1.1 Informační architektura

Obrázek 15 Informační architektura



Zdroj: Vlastní zpracování

4.1.2 Wireframe

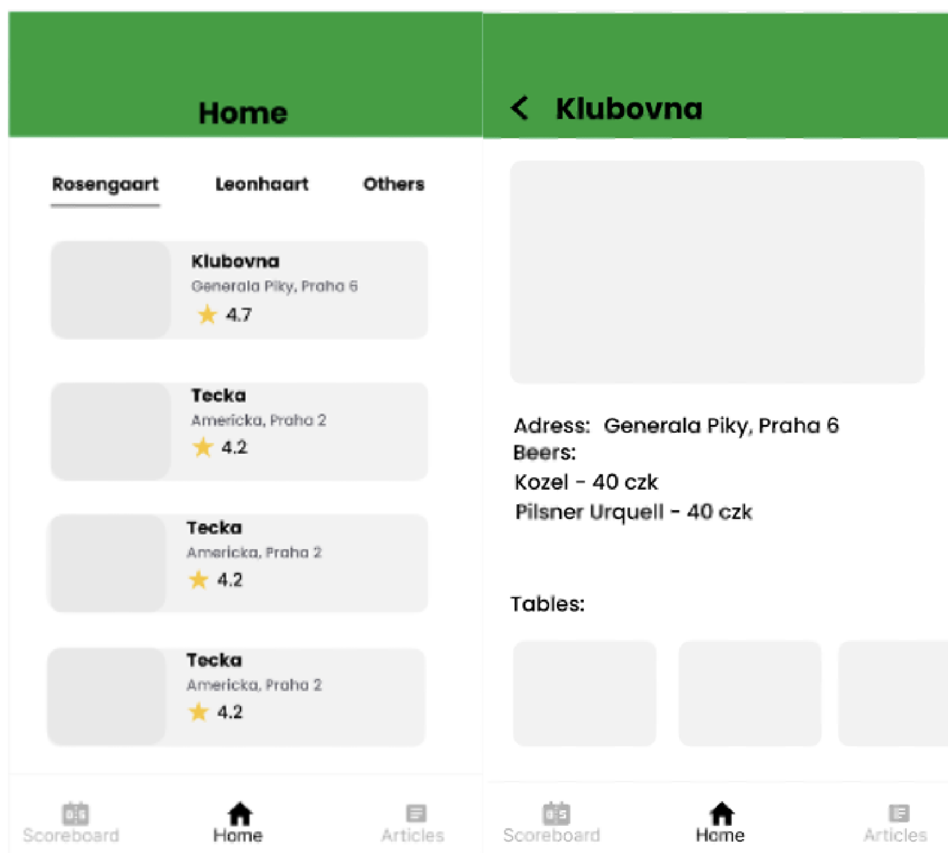
Pro vytvoření wireframe byl použit návrhářský nástroj **Figma**. Pro každou obrazovku aplikace byl vytvořen samostatný design. V celkovém výsledku bylo vytvořeno 7 **wireframe**. Veškeré obrazovky byly vytvořeny na obrazovku zařízení iPhone SE 3. generace.

Po spuštění aplikace je uživatel přesměrován na domovní obrazovku (viz obrázek 14), na které je načten seznam podniků, které disponují stolním fotbálkem. Na spodní části obrazovky je zobrazena navigace, přes kterou se může uživatel přesměrovávat mezi stránkami **Scoreboard**, **Home** a **Articles**. Tato navigace je zobrazena na všech obrazovkách napříč aplikací. Pod nadpisem **Home** budou zobrazeny tři přepínatelné záložky, které filtrují zobrazený seznam podniků podle dvou nejoblíbenějších značek stolů fotbálků a ostatních.

Každá karta tohoto seznamu je klikatelná a skládá se z obrázku daného podniku, názvu, adresy a hodnocení. Po kliknutí na danou kartu bude uživatel přesměrován na detailní popis daného podniku.

Na stránce detailního popisu je v záhlaví zobrazen název podniku, hlavní obrázek podniku, adresa, hodnocení, nabídka piv s cenou a horizontálně posuvný list obrázků stolních fotbálků. V záhlaví této stránky je tlačítko pro vrácení se zpět na stránku **home**.

Obrázek 16 Wireframe - Obrazovka Home, Pub Detail

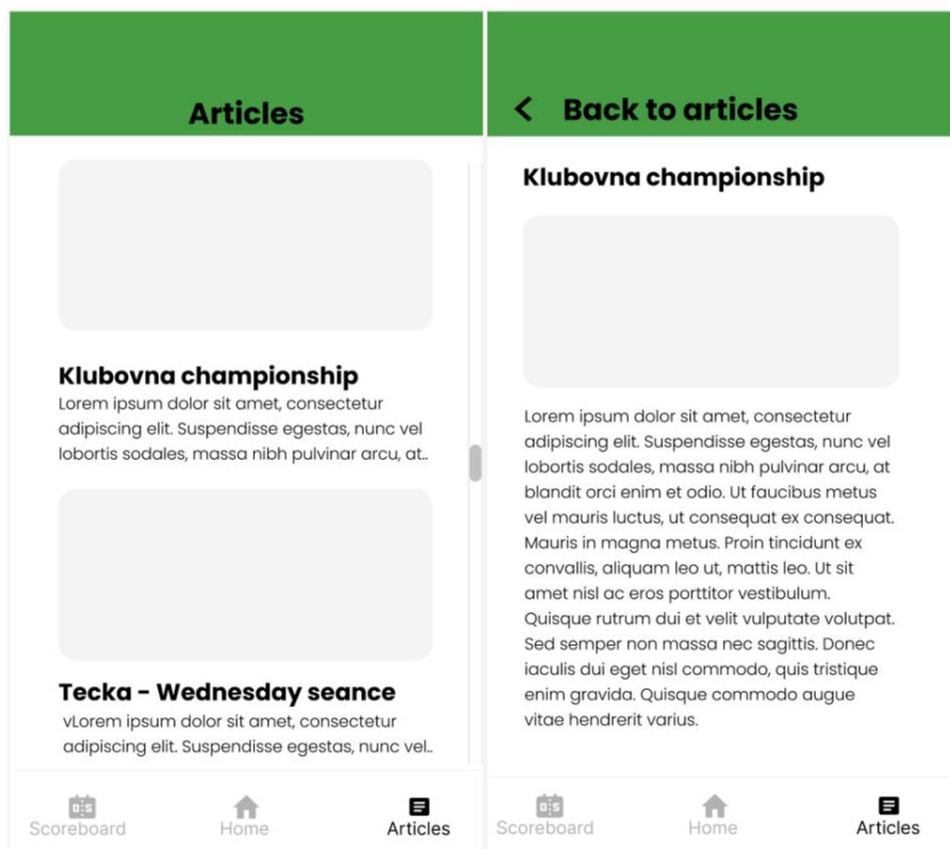


Zdroj: vlastní zpracování

Při otevření záložky **Articles** (viz obrázek 15) bude zobrazen seznam článků z oblasti stolních fotbálků. Tyto články jsou vydávány jednotlivými podniky v návaznosti například na nadcházející turnaje, výslkedků turnajů a podobně. Každý článek je tvořen kartou, která disponuje obrázkem, nadpisem a krátkým textem, který je součástí obsahu článku. Tato karta

je klikatelná a přesměruje uživatele na stránku, která obsahuje celý text daného článku. V záhlaví stránky se lze přesměrovat zpět na výpis všech článků.

Obrázek 17 Wireframe - Obrazovka articles, article detail



Zdroj: vlastní zpracování

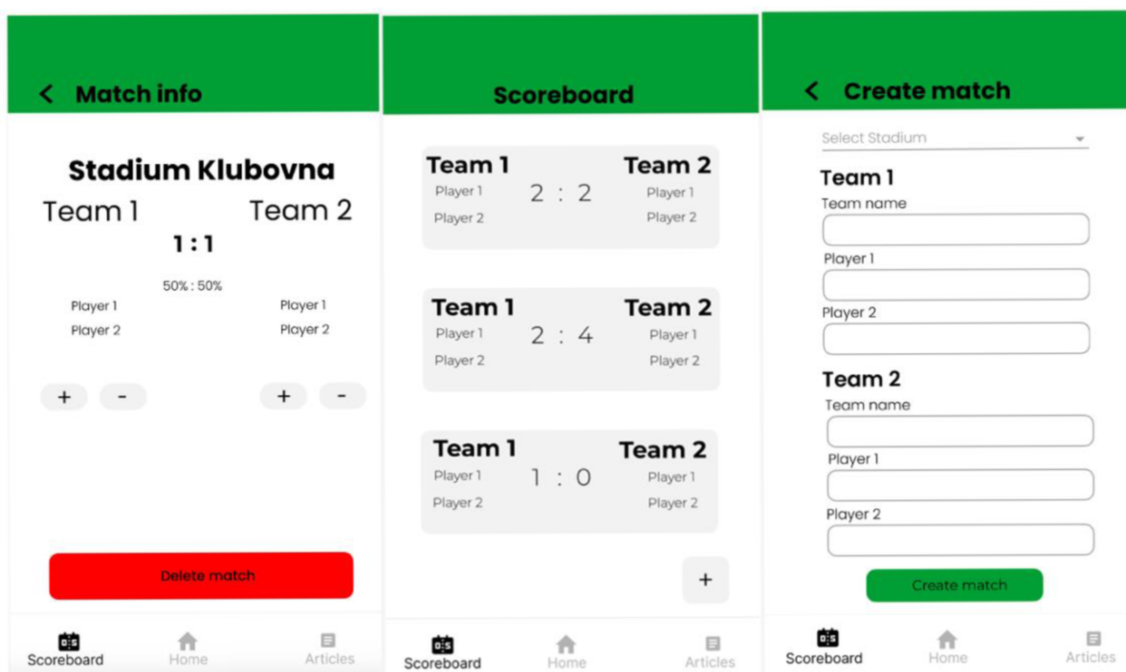
Na stránce **Scoreboard** (viz obrázek 16) je zobrazen seznam uložených utkání, které byly založeny uživatelem aplikace. Každá karta obsahuje názvy obou týmů, jednotlivých hráčů a dosavadní skóre. Každá karta je klikatelná a přesměruje uživatele na stránku **Match Info** (viz obrázek 16 - zleva). V pravém spodním rohu je na fixní pozici tlačítko, které uživatele přesměruje na stránku **Create Match** (viz obrázek 16 - zprava)

Stránka **Create Match** umožňuje založit uživateli dva nové týmy, které se skládají z názvu týmu a jmen obou hráčů. Pod formulářem se vyskytuje tlačítko k vytvoření těchto týmů. Tato pole nemohou být prázdná, pokud má dojít k vytvoření týmu.

Na poslední stránce aplikace **Match Info** jsou vypsány informace o zápasu. Je zde „název stadionu“ který reprezentuje podnik, ve kterém se utkání odehrálo, dále jsou zde

názvy obou týmů, jména hráčů skóre zápasu a procentuální skóre zápasů. Tato stránka umožňuje měnit skóre zápasu pomocí tlačítek (+) a (-).. Ve spodní části této stránky je tlačítko, kterým lze smazat daný zápas.

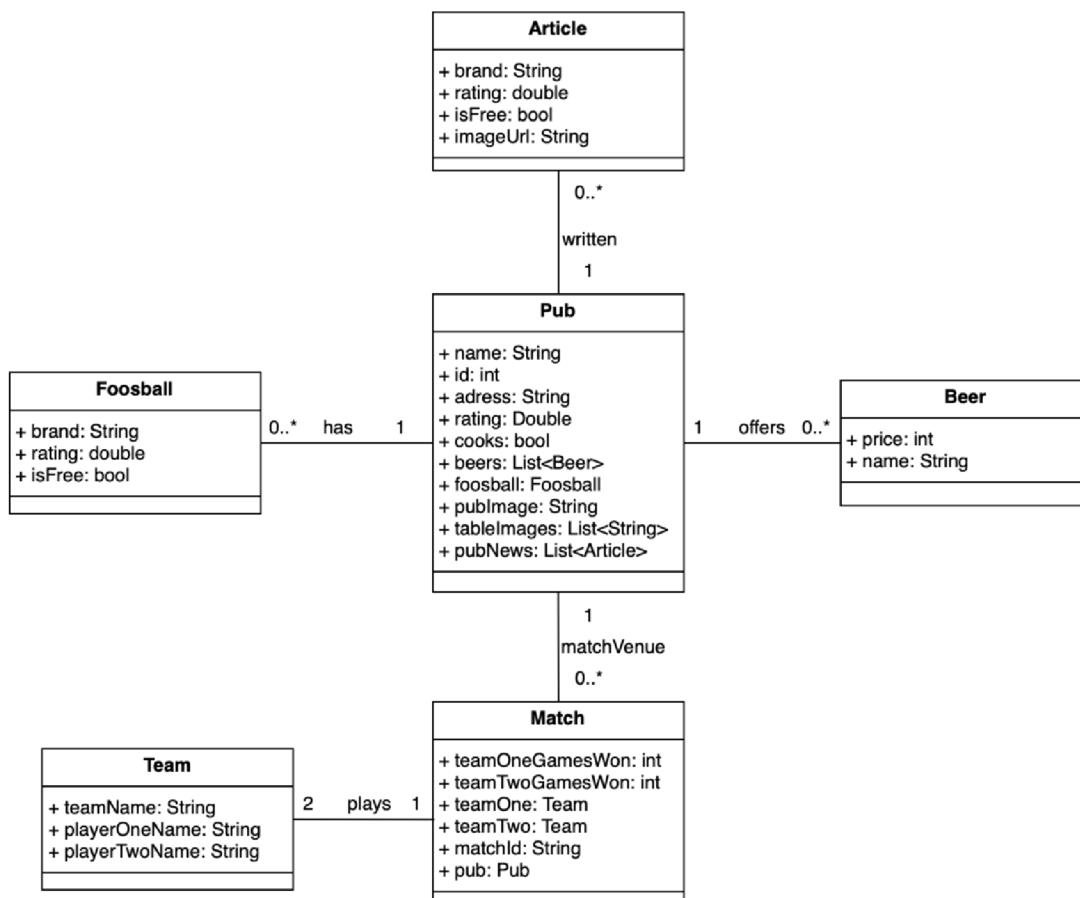
Obrázek 18 Wireframe - Obrazovky - Scoreboard, Match Info, Create Match



Zdroj: Vlastní zpracování

4.1.3 Diagram tříd

Obrázek 19 Diagram tříd



Zdroj: Vlastní zpracování

4.1.4 Práce s daty v aplikaci

Aplikace simuluje práci se serverem, všechna data jsou uložena staticky v simulované databázi přímo v aplikaci jako list daných hodnot.

Pokud by aplikace neměla sloužit pouze jako prototyp a potřebovali bychom mít uložena data v databázi na serveru, bývá využíván pro komunikaci se serverem Dart balíček Dio. Tento balíček umožňuje definovat jednotlivá API volání. Není zvykem tato volání definovat jednotlivě, jelikož většina těchto API volání jsou si implementací velice podobná, tudíž se v praxi využívá generátorů kódu pro urychlení práce. Balíčků s těmito generátory je mnoho, avšak mezi nejpoužívanější patří REST API a Open API.

4.2 Implementace aplikace

4.2.1 Flutter balíčky využité pro aplikaci

Během procesu vývoje mobilních aplikací s technologií Flutter jsou balíčky velmi důležitým prvkem. Tyto balíčky umožňují rychlejší a efektivnější práci. Při implementaci aplikace byly použity následující balíčky:

Balíček getIt
S narůstající velikostí aplikace je důležité, aby její logika byla definována ve třídách, které jsou oddělené od widgetů. Zajištěním, že widgety nemají přímé závislosti udělá kód uspořádaným a snadněji testovatelným.

Balíček getIt umožňuje vývojáři přistupovat k objektům z uživatelského rozhraní. Nejčastěji se využívá s klientem REST API nebo k přístupu BLoC, View či AppModels. Jedná se o lokátor pro objekty, tudíž není potřeba upozornit uživatelské rozhraní na změny v hodnotách. [53]

Způsoby registrování objektů:
Singleton – Záměrem návrhového vzoru singleton je zajištění, aby třída měla pouze jednu instanci. Z tohoto vyplývá, že daná instance singleton by měla být opakovaně používaná. Aby bylo možné uchovat instanci pro její budoucí opětovné využití, je třeba ji uložit do globálního kontextu. [54]

Factory – Jedná se o jeden nejvyužívanějších vzorů v moderních programovacích jazycích. Představuje návrhový vzor pro tvorbu, která představuje rozhraní pro vytváření objektů v rodičovské třídě, ale umožňuje svým potomkům měnit typ objektů, které budou vytvořeny. [55]

Balíček BLoC
Využitím daného balíčku lze jednoduše implementovat návrhového vzoru Business Logic Component. Tento návrhový vzor pomáha oddělit prezentační vrstvu od obchodní logiky.

V balíčku je definovaná třída Cubit, která rozšiřuje BlocBase a může být rozšířena o správu jakéhokoliv stavu. Tato třída vyžaduje počáteční stav, před zavoláním funkce emit. K aktuálnímu stavu cubitu lze přistopit pomocí getteru state a stav cubitu lze aktualizovat voláním funkce emit s novým stavem.

Bloc je třída, která reaguje zejména na události, než na funkce. Tato třída přijímá události a převádí příchozí události na odchozí stavy, namísto volání funkce *Bloc* a volání funkce emit s novým stavem jako u Cubit. [56]

Balíček go_router

Jedná se o deklarativní směrovací balíček, využívající Router API pro přesouvání mezi jednotlivými obrazovkami. Lze definovat vzory URL, pomoci kterých uživatele přesměrovávat mezi obrazovkami.

Mezi užitečné funkce patří:

- Podpora pro přesměrování uživatele na základě stavu aplikace
- Více navigátorů například pro BottomNavigationBar [57]

Balíček uuid

Celým názvem Universally Unique Identifier je balíček, který zjednodušuje vývojáři s vytvořním nového UUID a správu vygenerovaných UUID. Využívá se v různých situacích, kde je zapotřebí vytvořit jedinečný identifikátor. [58]

Balíček bloc_test

Bloc testy jsou využívány pro testování specifických bloc či cubit využívaných v aplikaci. Tyto testy ověřují, zda bloc emituje očekáváné stavy po provedení části kódu definovaného v části **act**.

- **setUp** – Využívá se k nastavení všech závislostí před inicializací testovaného cubitu/blocu. Tato část testu je nepovinná
- **build** – Sestavuje a vrací testovaný **bloc**
- **seed** – Nepovinná funkce, která vrací stav, který bude použit jako počáteční stav před zavolání funkce **act**.
- **act** – Funkce využívaná k interakci s **bloc**. Jedná se o nepovinou funkci
- **skip** – Nepovinná **integer**, který lze využít k vynechání určitého počtu stavů.

- **wait** – Nepovinný parametr **Duration** – využívá se k čekání na asynchronní funkce v rámci testovaného **bloc** či **cubit**.
- **expect** – Nepovinná funkce, která porovnává aktuální stav se stavem předpokládaným
- **verify** – Nepovinná funkce, fungující jako dodatečné potvrzení či ověření po funkci **expect**. [59]

Patrol

Jedná se o open-source testovací framework pro aplikace Flutter od společnost LeanCode. Testování pomocí patrol umožňuje přístup k nativním funkcím platformy, na které je spuštěna Flutter aplikace. Umožňuje komunikaci s oznámením, dialogy, měnit nastavení zařízení či přepínat wi-fi za použití pouze programovacího jazyku Dart. Patrol také rozšiřuje výchozí Flutter vyhledávače pro testování, které jsou kratší a přehlednější. [60]

4.2.2 Inicializace aplikace

Vstupním bodem programu Dart je asynchronní funkce **void main** (řádek 5), která je spuštěna jako první funkce programu a tedy instrukce programu začínají (viz řádek 6). Funkce **main** může přijímat dva dobrovolné parametry, avšak pro většinu aplikací Flutter jsou tyto argumenty zbytečné, proto se zde žádné nepředávají. Funkce **register** má za úkol inicializovat **dependency injection** (řádek 6).

Funkce **runApp** (řádek 7) představuje vstupní bod pro naši aplikaci, podobně jako funkce **main** pro náš program. Správným přístupem je registrace závislostí ještě před spuštěním dané aplikace viz. funkce **register**.

Hlavní widget **MainApp** (řádek 10-21), který ve své funkci **build** vrací knihovní **widget MaterialApp** zkonstruovaný pomocí pojmenovaného konstruktoru **.router**. Tento konstruktor nám dává k dispozici speciální parametr **routerConfig**, který nám umožňuje definovat navigaci napříč obrazovkami aplikace. Parametr **theme** určuje téma naší aplikace.

Obrázek 20 Inicializace aplikace

```
5   void main() async {
6     await register();
7     runApp(const MainApp());
8   }
9
10  class MainApp extends StatelessWidget {
11    const MainApp({super.key});
12
13    @override
14    Widget build(BuildContext context) {
15      return MaterialApp.router(
16        debugShowCheckedModeBanner: false,
17        title: 'FootballApp',
18        routerConfig: AppNavigation.router,
19        theme: ThemeData(fontFamily: 'Montserrat'),
20      ); // MaterialApp.router
21    }
22  }
```

Zdroj: Vlastní zpracování

V kódu (viz obrázek 21) je definována třída **ProjectColors**, která slouží k definování barev, které jsou využívány nejčastěji využívány napříč aplikací.

Mezi tyto barvy patří:

- Barva pro záhlaví **headerColor** (řádek 6)
- Barva pro kartu na domovské stránce **homePageCardColor** (řádek 7)
- Barva pro ikony v tlačítku **createButtonIconcolor** (řádek 8)
- Barva pro pozadí tlačítka pro vytvoření zápasu (řádek 9-10)
- Barva pro kartu zápasu na obrazovce **Scoreboard** (řádek 11)

Tato struktura umožňuje centralizovanou správu barev využívaných v projektu, díky kterému se lze při změnách vyvarovat zadáváním opakovaných hodnot ve více částech kódu.

Obrázek 21 Definování barev pro aplikaci

```
 3  class ProjectColors {
 4    ProjectColors._();
 5
 6    static const Color headerColor = Color.fromARGB(255, 68, 157, 68);
 7    static const Color homePageCardColor = Color.fromARGB(255, 244, 244, 244);
 8    static const Color createButtonIconColor = Color.fromARGB(255, 0, 0, 0);
 9    static const Color createButtonBackgroundColor =
10        Color.fromARGB(255, 219, 219, 219);
11    static const Color scoreboardCardColor = Color.fromARGB(255, 219, 219, 219);
12  }
13
```

Zdroj: Vlastní zpracování

Obdobným způsobem bylo přistoupeno také k textovým stylům, kde byla definována váha písma a velikost (viz obrázek 22).

Obrázek 22 Definování stylů pro aplikaci

```
 3  class CustomTextStyles {
 4    CustomTextStyles._();
 5
 6    static const TextStyle header =
 7        TextStyle(fontSize: 30, fontWeight: FontWeight.bold);
 8    static const TextStyle headlineBold =
 9        TextStyle(fontSize: 24, fontWeight: FontWeight.bold);
10    static const TextStyle headlineRegular = TextStyle(fontSize: 24);
11    static const TextStyle headline2 =
12        TextStyle(fontSize: 20, fontWeight: FontWeight.bold);
13    static const TextStyle regularText = TextStyle(fontSize: 16);
14    static const TextStyle regularTextBold =
15        TextStyle(fontSize: 16, fontWeight: FontWeight.bold);
16    static const TextStyle tab =
17        TextStyle(fontSize: 14, fontWeight: FontWeight.bold);
18    static const TextStyle pubDetailStyle = TextStyle(fontSize: 14);
19  }
```

Zdroj: Vlastní zpracování

V kódu (viz obrázek 23) je definována funkce **register** využívající Flutter balíček **getIt** pro registraci závislostí v projektu**.** Na řádku 19. je vytvořena instance třídy **GetIt**, která je využívána pro registraci a přístup k instancím jednotlivých tříd.

Obrázek 23 Registrace závislostí

```
18  GetIt getIt = GetIt.instance;
19
20  Future<void> register() async {
21    getIt.registerSingleton<PubMockupRepository>(PubMockupRepository());
22
23    getIt.registerSingleton<PubsRepository>(
24      PubsRepositoryImplementation(PubMockupRepository()),
25    );
26    getIt.registerSingleton<PubsRepositoryImplementation>(
27      PubsRepositoryImplementation(PubMockupRepository()),
28    );
29    getIt.registerFactory<HomePageCubit>(
30        () => HomePageCubit(getIt.get<PubsRepository>()));
31    getIt.registerSingleton<ArticlesRepository>(
32        ArticlesRepositoryImplementation());
33    getIt.registerSingleton<MatchesMockupRepository>(
34        MatchesMockupRepository(PubMockupRepository()));
35    getIt.registerSingleton<MatchRepositoryImplementation>(
36        MatchRepositoryImplementation(
37            MatchesMockupRepository(PubMockupRepository()),
38            PubMockupRepository())); // MatchRepositoryImplementation
39    getIt.registerSingleton<MatchesRepository>(MatchRepositoryImplementation(
40        MatchesMockupRepository(PubMockupRepository()), PubMockupRepository()));
41    getIt.registerFactory<CreateMatchCubit>(() => CreateMatchCubit(
42        getIt.get<MatchesRepository>(), getIt.get<MatchesMockupRepository>()));
43    getIt.registerFactory<MatchDetailCubit>(
44        () => MatchDetailCubit(getIt.get<MatchesRepository>()));
45    getIt.registerFactory<MatchCubit>(
46        () => MatchCubit(getIt.get<MatchesRepository>()));
47    getIt.registerFactory<DetailPageCubit>(
48        () => DetailPageCubit(getIt.get<PubsRepository>()));
49    getIt.registerFactory<ArticleDetailPageCubit>(
50        () => ArticleDetailPageCubit(getIt.get<ArticlesRepository>()));
51    getIt.registerFactory<ArticlesPageCubit>(
52        () => ArticlesPageCubit(getIt.get<ArticlesRepository>()));
53  }
```

Zdroj: Vlastní zpracování

Kód (viz obrázek 24) definuje třídu **AppNavigation**, která je zodpovědná za navigaci v rámci aplikace. Na řádku 14. je definována proměnná **initialRoutePath**, která určuje vstupní obrazovku aplikace, na kterou se uživatel přesune při spuštění aplikace. Tato proměnná je předána na řádku 16. třídě **GoRouter**, která spravuje samotnou navigaci v aplikaci.

Obrázek 24 Navigace v aplikaci

```
11  class AppNavigation {
12    AppNavigation._();
13
14    static String initialRouter = '/home';
15    static final GoRouter router = GoRouter(
16      initialLocation: initialRouter,
17      routes: [
18        StatefulShellRoute.indexedStack(
19          builder: (context, state, navigationShell) {
20            return InitialPage(
21              navigationShell: navigationShell,
22            ); // InitialPage
23          },
24          branches: [
```

Zdroj: Vlastní zpracování

4.2.3 Funkční část obrazovky Home

Třída **HomePage** je bezstavový widget, který ve funkci **build** vrací widget **HomePageContentView** (uživatelské rozhraní stránky **Home**), obalený speciálním **widgetem BlocProvider**. **BlocPovider** poskytuje možnost práce s **cubit** ve všech widgetech v podstromu **HomePageContentView**. **Cubit** se kterým se pracuje je předán pomocí lokátoru **getIt** (řádek 13). Ihned po předání je na něm volána funkce **load**, která načte potřebná data pro tuto stránku a vyvolá příslušný stav (viz obrázek 25).

Obrázek 25 Sestavení stránky Home, inicializace BlocProvider

```
7   class HomePage extends StatelessWidget {
8     const HomePage({super.key});
9
10    @override
11    Widget build(BuildContext context) {
12      return BlocProvider(
13        create: (context) => getIt.get<HomePageCubit>()..load(),
14        child: const HomePageContentView(),
15      ); // BlocProvider
16    }
17  }
18
```

Zdroj: Vlastní zpracování

Třídy v kódu (viz obrázek 26) reprezentují stavy domovské stránky aplikace. Třída **HomePageState** představuje obecný stav domovské stránky, která obsahuje seznam podniků.

Třída **HomePageStateLoading** představuje stav, kdy se data na domovské stránce načítají.

Poslední třída **HomePageStateLoaded** představuje stav, kdy jsou data načtena. Na tyto stavy může UI reagovat různými způsoby, například je vhodné pro stav načítání dat zobrazit indikátor načítání namísto prázdné (nenačtené) stránky.

Obrázek 26 Stavy stránky Home

```
4   class HomePageState extends Equatable {
5     const HomePageState({required this.list});
6     final List<PubEntity> list;
7
8     @override
9     List<Object?> get props => [list];
10  }
11
12  class HomePageStateLoading extends HomePageState {
13    HomePageStateLoading() : super(list: []);
14  }
15
16  class HomePageStateLoaded extends HomePageState {
17    const HomePageStateLoaded({required super.list});
18  }
19
```

Zdroj: Vlastní zpracování

V kódu (viz obrázek 27) můžeme vidět třídu **cubit**, která rozšiřuje generickou třídu **Cubit** z balíčku **bloc**, jejíž generickým parametrem je třída obecného stavu **HomePageState**.

Definice funkce **load** (řádek 12.), která zpočátku **emituje** stav stránky **HomePageStateLoading**, poté je využita metoda **getAll**, která je deklarovaná v abstraktní třídě **PubsRepository** a získá list všech hodnot. Na konci funkce je **emitován** stav **HomePageStateLoaded** s načtenými daty.

Obrázek 27 Cubit pro stránku Home

```
8   class HomePageCubit extends Cubit<HomePageState> {
9     HomePageCubit(this.repository) : super(HomePageStateLoading());
10    PubsRepository repository;
11
12    load() {
13      emit(
14        HomePageStateLoading(),
15      );
16
17      List<PubEntity> pubs = repository.getAll();
18
19      emit(
20        HomePageStateLoaded(list: pubs),
21      );
22    }
23  }
```

Zdroj: Vlastní zpracování

V kódu (viz obrázek 28) je definováno rozhraní třídy **PubsRepository**, která deklaruje dvě funkce: **getById**, která vrací hodnoty dané hospody podle jejího identifikátoru a **getAll**, která vrací všechny hospody.

Obrázek 28 Abstraktní třída PubsRepository

```
3  abstract interface class PubsRepository {
4    Pub getById(int id);
5    List<Pub> getAll();
6  }
```

Zdroj: Vlastní zpracování

V kódu (viz obrázek 29) můžeme vidět implementaci rozhraní **PubsRepository**. Proměnná **mockupRepository**, typu **PubMockupRepository** která slouží jako zdroj dat o hospodách. Metoda **getById**, která přijímá **integer „id"** jako parametr, prohledává seznam hospod **listOfPubs**, který je uložen v proměnné **mockupRepository** a vrací hospodu, který má shodné **id.** Metoda **getAll** vrací všechny hodnoty uložené v simulovaném repositáři.

Obrázek 29 Implementace abstraktní třídy PubsRepository

```
4   class PubsRepositoryImplementation implements PubsRepository {
5     PubsRepositoryImplementation(this.repo);
6     final PubMockupRepository repo;
7     @override
8     getById(int id) {
9       final pub = PubMockupRepository.listOfPubs
10          .where((element) => element.id == id)
11          .first;
12      return pub;
13    }
14
15    @override
16    getAll() {
17      final data = PubMockupRepository.listOfPubs;
18      return data;
19    }
20  }
```

Zdroj: Vlastní zpracování

V kódu (viz obrázek 30) je použit **BlocBuilder**, který sleduje stavy vyvolané třídou **HomePageCubit** a z příslušných stavů čte data a zobrazuje je na základě typu stavu, ze kterého byly přečteny (řádek 35. a 39.). Pokud jsou data čtena ze stavu **HomePageStateLoading**, je na obrazovce zobrazen načítací kruhový indikátor. Po úspěšném načtení dat je vyvolán stav **HomePageStateLoaded**, na který stránka reaguje vykreslím konečného stavu domovské stránky s patřičnými daty.

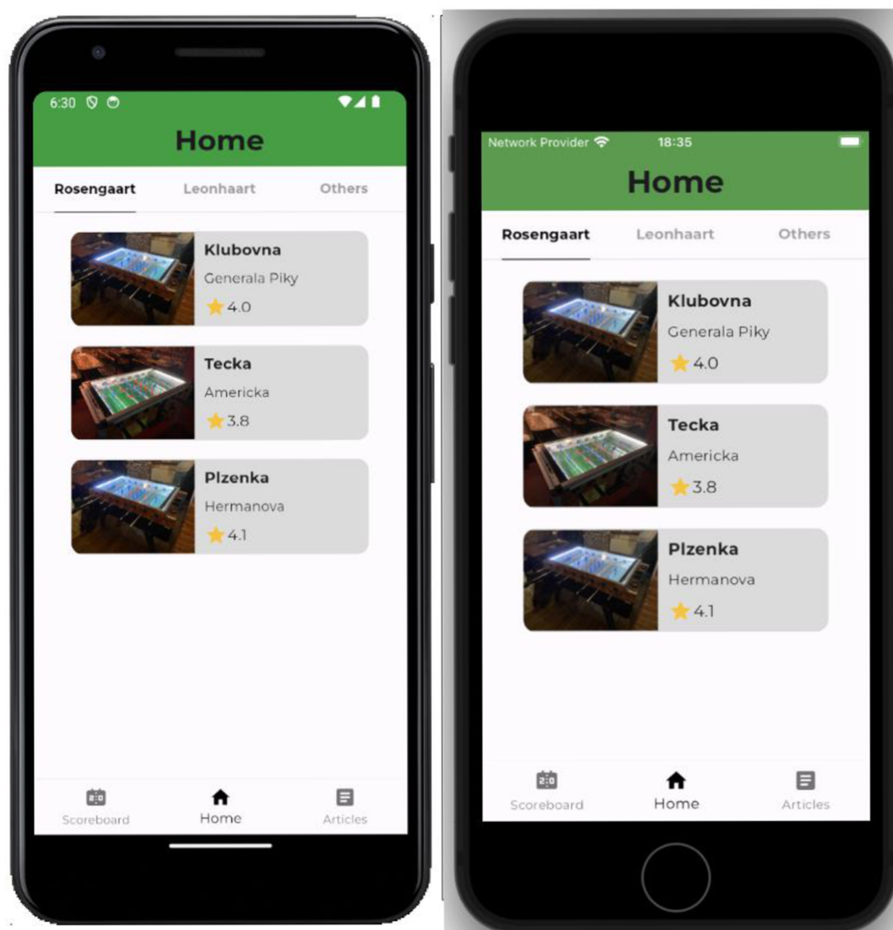
Obrázek 30 Vykreslení stránky Home pomoc BlocBuilder

```
31  @override
32  Widget build(BuildContext context) {
33    return BlocBuilder<HomePageCubit, HomePageState>(
34      builder: (context, state) {
35        if (state is HomePageStateLoading) {
36          const Center(
37            child: CircularProgressIndicator(),
38          ); // Center
39        } else if (state is HomePageStateLoaded) {
40          return Scaffold(
41            appBar: AppBar(
42              backgroundColor: ProjectColors.headerColor,
43              title: const Center(
44                child: Text(
45                  'Home',
46                  style: CustomTextStyles.header,
47                ), // Text
48              ), // Center
49            ), // AppBar
50            body: Column(
```

Zdroj: Vlastní zpracování

Po otevření aplikace je uživatel přesměrován na domovskou stránku, kde jsou vypsány jednotlivé podniky, disponující stolním fotbálkem. Podniky jsou rozděleny pomocí záložek podle značky herních stolů do tří kategorií (viz obrázek 31).

Obrázek 31 Obrazovka Home na zařízení platforem Android a iOS



Zdroj: Vlastní zpracování

4.2.4 UI část obrazovky Home

Scaffold widget poskytuje všem svým potomkům hlavní tému aplikace a rozměrové omezení na velikost obrazovky. V aplikace využíváme dvou parametrů **appBar** a **body** (viz. obrázek 32).

Obrázek 32 Scaffold widget

```
40      return Scaffold(
41 >      appBar: AppBar( // AppBar ⋯
50 >      body: Column( // Column ⋯
186     ); // Scaffold
```

Zdroj: Vlastní zpracování

AppBar (viz obrázek 33) je knihovní implementace často využívaného designového vzoru záhlaví aplikace. Pro jeho správné zobrazení je předáván jako parametr **appBar** do widgetu **Scaffold**. Na řádku 42 lze vidět předání hodnoty **headerColor** z definované třídy **ProjectColors** (viz obrázek 20).

Obrázek 33 AppBar widget

```
41      appBar: AppBar(
42        backgroundColor: ProjectColors.headerColor,
43        title: const Center(
44          child: Text(
45            'Home',
46            style: CustomTextStyles.header,
47          ), // Text
48        ), // Center
49      ), // AppBar
```

Zdroj: Vlastní zpracování

Tělo widgetu **Scaffold** je tvořeno widgetem **Column** (sloupec), který své potomky uspořádává do sloupce. Výchozí uspořádání potomků je od shora na hlavní ose a doprostřed na kolmé ose (viz obrázek 34).

Obrázek 34 Column widget

```
50      body: Column(
51 >      children: [⋯
```

Zdroj: Vlastní zpracování

Widget **TabBar** je knihovní implementací designového vzoru záložek. Controller má za úkol přepínání mezi jednotlivými záložkami (viz obrázek 35).

Obrázek 35 TabBar widget

```
52    TabBar(
53      controller: tabController,
54      labelColor: Colors.black,
55      unselectedLabelColor: ProjectColors.homePageCardColor,
56 >    indicator: const UnderlineTabIndicator( // UnderlineTabIndicator ⋯
61      tabs: [
62 >      Tab( // Tab ⋯
68 >      Tab( // Tab ⋯
74 >      Tab( // Tab ⋯
80      ],
81    ), // TabBar
```

Zdroj: Vlastní zpracování

Widget **Expanded** (řádek 82) roztáhne potomka na maximální velikost, která je k dispozici na hlavní ose rodičovského widgetu **Column** (viz obrázek 36).

Obrázek 36 Expanded widget

```
82    Expanded(
83      child: TabBarView(
84        controller: tabController,
85        children: [
86 >        Padding( // Padding ⋯
```

Zdroj: Vlastní zpracování

Widget **TabBarView** poskytuje implementaci animací a detekci přejetí prstem po obrazovce pro přepínání mezi různými obsahy. Tyto obsahy jsou předány v podobě listu potomků v parametru children. Parametr **controller** přijímá instanci třídy **TabController**, jejíž úkolem je spravovat tyto animace a gesta (viz obrázek 37).

Obrázek 37 TabBarView widget

```
83        child: TabBarView(
84          controller: tabController,
85          children: [
86 >          Padding( // Padding ⋯
115 >         Padding( // Padding ⋯
149 >         Padding( // Padding ⋯
180         ],
181       ), // TabBarView
```

Zdroj: Vlastní zpracování

Podobně jako **Column** uspořádává widget **ListView** své potomky do sloupce s jediným rozdílem, že pokud by se potomci nevešly do vyhrazeného prostoru stane se tento sloupec posuvným.

Pojmenovaný konstruktor **.builder** dává možnost potomky stavit podle daných pravidel a zamezit opakování kódu - symbolizuje smyčku **for** (int index = 0; index < **itemCount**; index++) (viz obrázek 38).

Obrázek 38 ListView widget

```
88        child: ListView.builder(
89          itemCount: state.list.length,
90          itemBuilder: (context, index) {
91            if (state.list[index].fotbalek.brand ==
92                tabBarNames[0]) {
93 >            return PubCardWidget( // PubCardWidget ⋯
108           } else {
109             return const SizedBox.shrink();
110           }
111         },
112       ), // ListView.builder
```

Zdroj: Vlastní zpracování

Pokud je splněna podmínka (řádek 91-92) je předán potomek **PubCardWidget**, jinak je předán widget **SizedBox.shrink**, který reprezentuje prázdného potomka nejmenší možné velikosti (viz obrázek 39).

Obrázek 39 PubCardWidget

```
 93  return PubCardWidget(
 94    name: state.list[index].name,
 95    rating: state.list[index].rating,
 96    adress: state.list[index].adress,
 97    imageUrl: state.list[index].pubImage,
 98    callback: () {
 99      context.pushNamed(
100        DetailPage.routeName,
101        pathParameters: {
102          'id': (state.list[index].id).toString(),
103        },
104      );
105    },
106  ); // PubCardWidget
```

Zdroj: Vlastní zpracování

PubCardWidget představuje kartu, zobrazující nejdůležitější informace o podniku. Zobrazuje název, hodnocení, adresu a obrázek (řádek 94–97), které jsou předané v konstruktoru společně s funkcí, která je volána při doteku karty. Dotekem je uživatel přesměrován na stránku detailu daného podniku (řádek 98-105).

4.2.5 Testování aplikace

Widget test

V kódu (viz obrázek 40) je vypsán test pro ověření správného zobrazení vytvořeného karetního widgetu, který je využíván v aplikaci na stránce **Home**. Test využívá testovací knihovnu **flutter_test** a je testován pomocí asynchronní funkce **testWidgets**, která zajišťuje testování uživatelského rozhraní. Na řádku (9-12) jsou definovány konstanty pro nově vytvořenou instanci. Test je vykreslen v testovacím prostředí funkcí **pumpWidget**. Na řádcích (29-32) probíhá ověření existence textových prvků právě jednou s definovanými hodnotami.

Obrázek 40 Widget test

```
6   testWidgets(
7     'PubCardWidget displays correctly with all of its values',
8     (WidgetTester tester) async {
9       const String name = 'New Pub';
10      const double rating = 4.0;
11      const String address = 'NewPubAdress 1';
12      const String imageUrl = '';
13
14      await tester.pumpWidget(
15        MaterialApp(
16          home: Scaffold(
17            body: PubCardWidget(
18              name: name,
19              rating: rating,
20              adress: address,
21              imageUrl: imageUrl,
22              callback: () {},
23            ), // PubCardWidget
24          ), // Scaffold
25        ), // MaterialApp
26      );
27
28      expect(find.text(name), findsOneWidget);
29      expect(find.text(address), findsOneWidget);
30      expect(find.text(rating.toString()), findsOneWidget);
31      expect(find.byType(Image), findsOneWidget);
32    },
33  );
34 }
```

Zdroj: Vlastní zpracování

Bloc test

V kódu (viz obrázek 41) proběhlo testování funkčnosti **ArticleDetailPageCubit**, která je využívána pro načítání dat jednotlivých článků. Cílem testu je ověření, zda tato třída **cubit** interaguje správným způsobem s rozhraním **ArticlesRepository**, která zajišťuje přístup k datům v aplikaci. Na řádku 9 je definována třída **MockArticleRepository**, která simuluje chování rozhraní **ArticleRepository** pomocí knihovny **mockito**. Samotné testování provádí metoda **blocTest**, která ověřuje funkci chování při volání metody **load**.

Obrázek 41 Bloc test

```
9   class MockArticlesRepository extends Mock implements ArticlesRepository {}
    Run | Debug
10  void main() {
      Run | Debug
11    group('ArticleDetailPageCubit', () {
12      late ArticleDetailPageCubit cubit;
13      late ArticlesRepository mockRepo;
14
15      setUp(() {
16        mockRepo = MockArticlesRepository();
17        cubit = ArticleDetailPageCubit(mockRepo);
18      });
19
20      tearDown(() {
21        cubit.close();
22      });
23
        Run | Debug
24      test('initial state is ArticleDetailPageLoading', () {
25        expect(cubit.state, ArticleDetailPageLoading());
26      });
27
        Run | Debug
28      blocTest<ArticleDetailPageCubit, ArticleDetailPageState>(
29        'emits ArticleDetailPageLoading first and ArticleDetailPageLoaded when load is called successfully',
30        build: () {
31          when(() => mockRepo.getArticleById(any())).thenReturn(ArticleEntity(
32              id: 1,
33              title: 'Test Article',
34              text: 'Test text',
35              imageUrl: 'test.jpg'));
36          return cubit;
37        },
38        act: (cubit) => cubit.load(1),
39        expect: () => [
40          ArticleDetailPageLoading(),
41          ArticleDetailPageLoaded(
42            article: ArticleEntity(
43              id: 1,
44              title: 'Test Article',
45              text: 'Test text',
46              imageUrl: 'test.jpg'), // ArticleEntity
47          ), // ArticleDetailPageLoaded
48        ],
49      );
50    });
51  }
```

Zdroj: Vlastní zpracování

Integrační test

Pro integrační test byl využit balíček patrol, který umožňuje jednoduchým zápisem provádět testování. Tento integrační test byl proveden na zjištění správnosti fungování při postupu od otevření aplikace po vytvoření týmu (viz obrázek 42).

Obrázek 42 Integrační test pomocí Patrol

```
7   void main() {
      Run | Debug
8     group('Integration test', () {
        Run | Debug
9       patrolTest('create match', ($) async {
10        await register();
11        await $.pumpWidgetAndSettle(
12          const MainApp(),
13        );
14        //move to scoreboard page
15        await $('Scoreboard').tap();
16        await $.pumpAndSettle();
17        // tap on create button
18        await $(FloatingActionButton).tap();
19        await $.pumpAndSettle();
20        // filling forms for team 1
21        await $(#teamOneName).enterText('Team1 New');
22        await $.pumpAndSettle();
23        await $(#teamOnePlayer1).enterText('P1');
24        await $.pumpAndSettle();
25        await $(#teamOnePlayer2).enterText('P2');
26        await $.pumpAndSettle();
27        // filling forms for team 2
28        await $(#teamTwoName).enterText('Team2 New');
29        await $.pumpAndSettle();
30        await $(#teamTwoPlayer1).enterText('P1');
31        await $.pumpAndSettle();
32        await $(#teamTwoPlayer2).enterText('P1');
33        await $.pumpAndSettle();
34        // find elevated button
35        await $(ElevatedButton).tap();
36        await $.pumpAndSettle();
37        expect($('Team1 New'), findsOneWidget);
38        await $('Team1 New').tap();
39        await $.pumpAndSettle();
40        expect($('Match info'), findsOneWidget);
41      });
42    });
43  }
```

Zdroj: Vlastní zpracování

Testování za použití nástroje patrol umožňuje náhled na výsledek testování za pomoci vygenerovaného souboru HTML. Pokud test proběhne neúspěšně je vytvořen report s popisem daných chyb, na které test narazil (viz obrázek 43).

Obrázek 43 Report neúspěšného integračního testu Patrol

Class com.example.diplomka.MainActivityTest

all > com.example.diplomka > MainActivityTest

1	1	15.807s	0%
tests	failures	duration	successful

Failed tests Tests

runDartTest[create_match_test.create_match_integration_test Integration test create match]

```
java.lang.AssertionError: Dart test failed: create_match_test.create_match_integration_test Integration test create match
══╡ EXCEPTION CAUGHT BY FLUTTER TEST FRAMEWORK ╞════════════════════
The following WaitUntilVisibleTimeoutException was thrown running
a test:
TimeoutException after 0:00:10.000000: Finder "Found 0 widgets
with type "FloatingActionButton": []" did not find any visible
(i.e. hit-testable) widgets

When the exception was thrown, this was the stack:
#0      PatrolTester.waitUntilVisible.<anonymous closure> (package:patrol_finders/src/custom_finders/patrol_tester.dart:405:11)
<asynchronous suspension>
#1      TestAsyncUtils.guard.<anonymous closure> (package:flutter_test/src/test_async_utils.dart:117:7)
<asynchronous suspension>
#2      PatrolTester.tap.<anonymous closure> (package:patrol_finders/src/custom_finders/patrol_tester.dart:247:30)
<asynchronous suspension>
#3      TestAsyncUtils.guard.<anonymous closure> (package:flutter_test/src/test_async_utils.dart:117:7)
<asynchronous suspension>
#4      PatrolFinder.tap (package:patrol_finders/src/custom_finders/patrol_finder.dart:207:5)
<asynchronous suspension>
#5      main.<anonymous closure>.<anonymous closure> (file:///Users/adam/Desktop/diplomka/diplomka/integration_test/create_match_test/create_match_integration_test.dart:35:7)
<asynchronous suspension>
#6      patrolTest.<anonymous closure> (package:patrol/src/common.dart:134:7)
<asynchronous suspension>
#7      testWidgets.<anonymous closure>.<anonymous closure> (package:flutter_test/src/widget_tester.dart:168:15)
<asynchronous suspension>
#8      TestWidgetsFlutterBinding._runTestBody (package:flutter_test/src/binding.dart:1013:5)
<asynchronous suspension>
#9      TestWidgetsFlutterBinding._createTestCompletionHandler.<anonymous closure> (package:flutter_test/src/binding.dart:804:12)
<asynchronous suspension>

The test description was:
create match
════════════════════════════════════════════════════════════════════

at pl.leancode.patrol.PatrolJUnitRunner.runDartTest(PatrolJUnitRunner.java:135)
at com.example.diplomka.MainActivityTest.runDartTest(MainActivityTest.java:31)
```

Zdroj: Vlastní zpracování

Pokud test dopadl úspěšně je vytvořen report, které vypíše testy, které byly spuštěny, na kterém zařízení a procentuální úspěšnost všech patrol testů (viz obrázek 44).

Obrázek 44 Report úspěšného integračního testu Patrol

Class com.example.diplomka.MainActivityTest

all > com.example.diplomka > MainActivityTest

1	0	6.169s	100%
tests	failures	duration	successful

Tests

Test	Pixel_3a_API_34_extension_level_7_arm64-v8a(AVD) - 14
runDartTest[create_match_test.create_match_integration_test Integration test create match]	passed (6.169s)

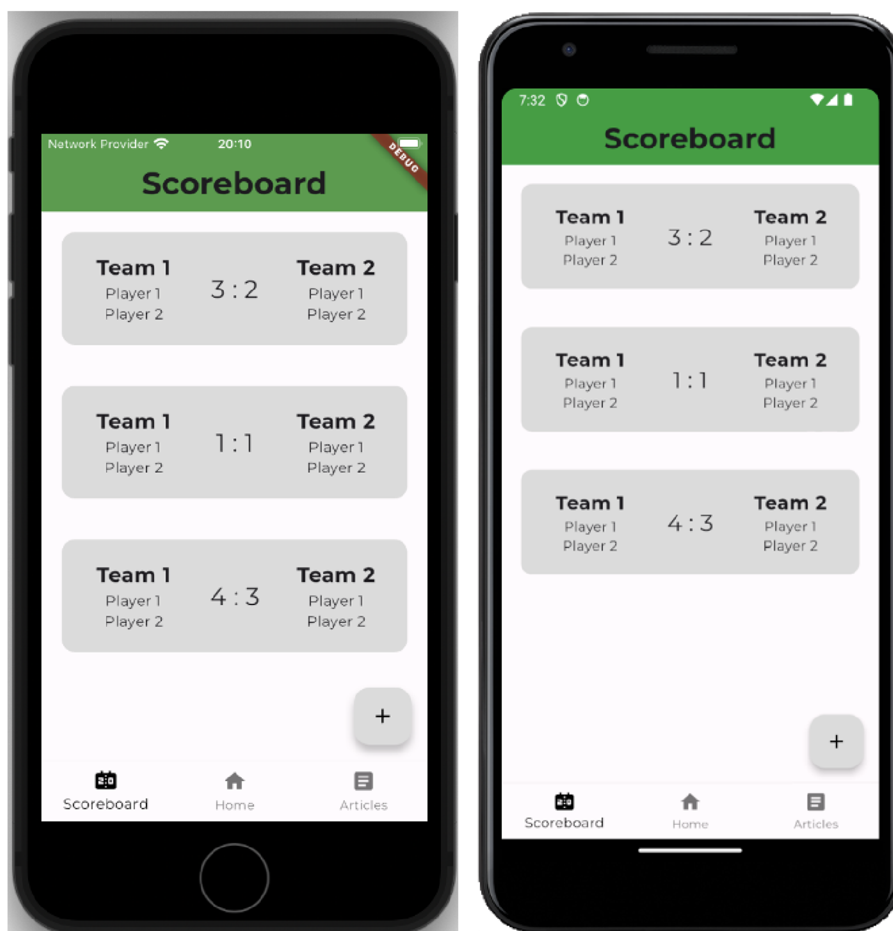
Zdroj: Vlastní zpracování

5 Výsledky a diskuze

V praktické části byl navržen a implementován prototyp aplikace. Tato aplikace má sloužit jako přehled podniků, které disponují stolním fotbálkem. Tato aplikace byla vytvořena cross-platform řešením a je spustitelná na zařízeních s operačními systémy iOS a Android.

Kromě seznamu jednotlivých podniků zde byla vytvořena stránka **Scoreboard**, kde si můžou uživatelé vytvořit s přáteli nový zápas a zapisovat si zde skóre zápasů (viz obrázek 45).

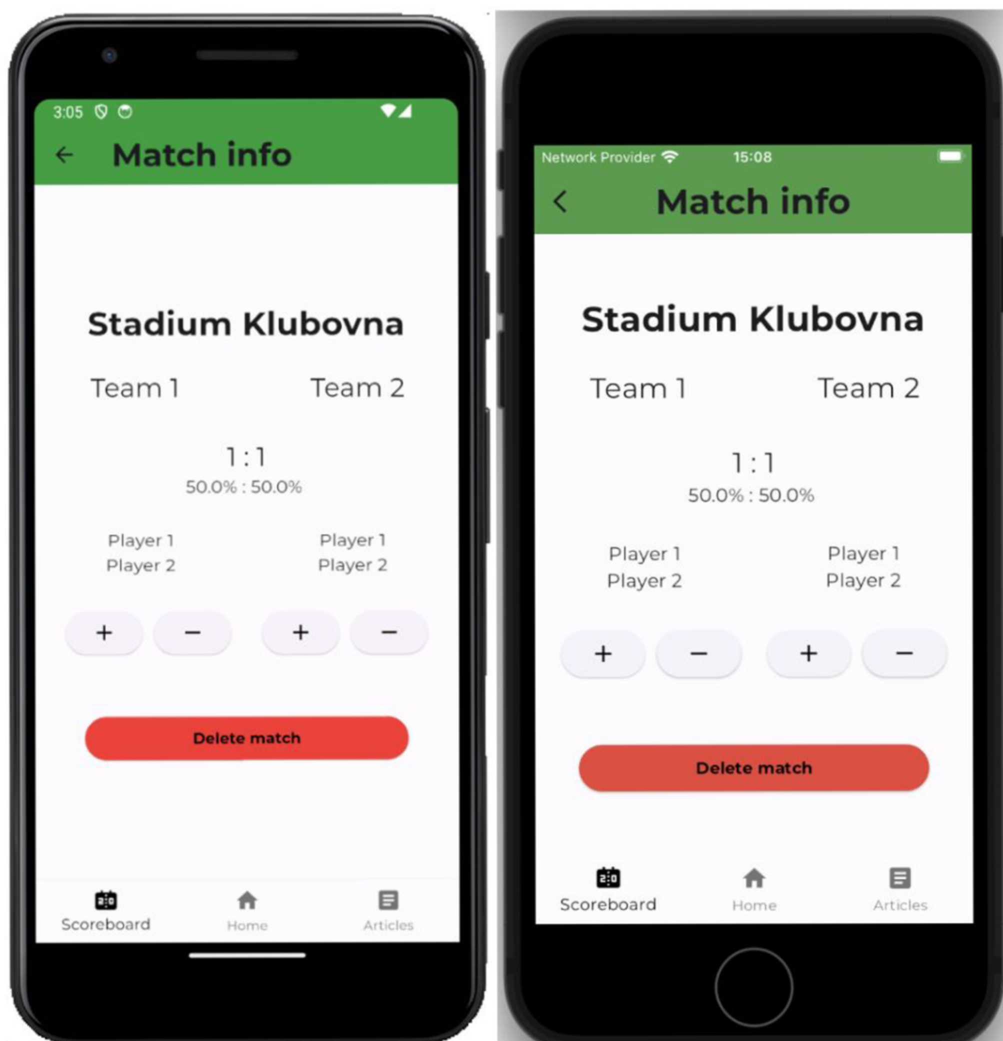
Obrázek 45 Obrazovka Scoreboard na zařízení platforem Android a iOS



Zdroj: Vlastní zpracování

Součástí obrazovky **Scoreboard** jsou jednotlivé zápasy. Zde byla vytvořena stránka **Match Info** pro zobrazení detailů zápasu. Uživatel může upravit skóre jednotlivého zápasu popřípadě záznam zápasu smazat (viz obrázek 46).

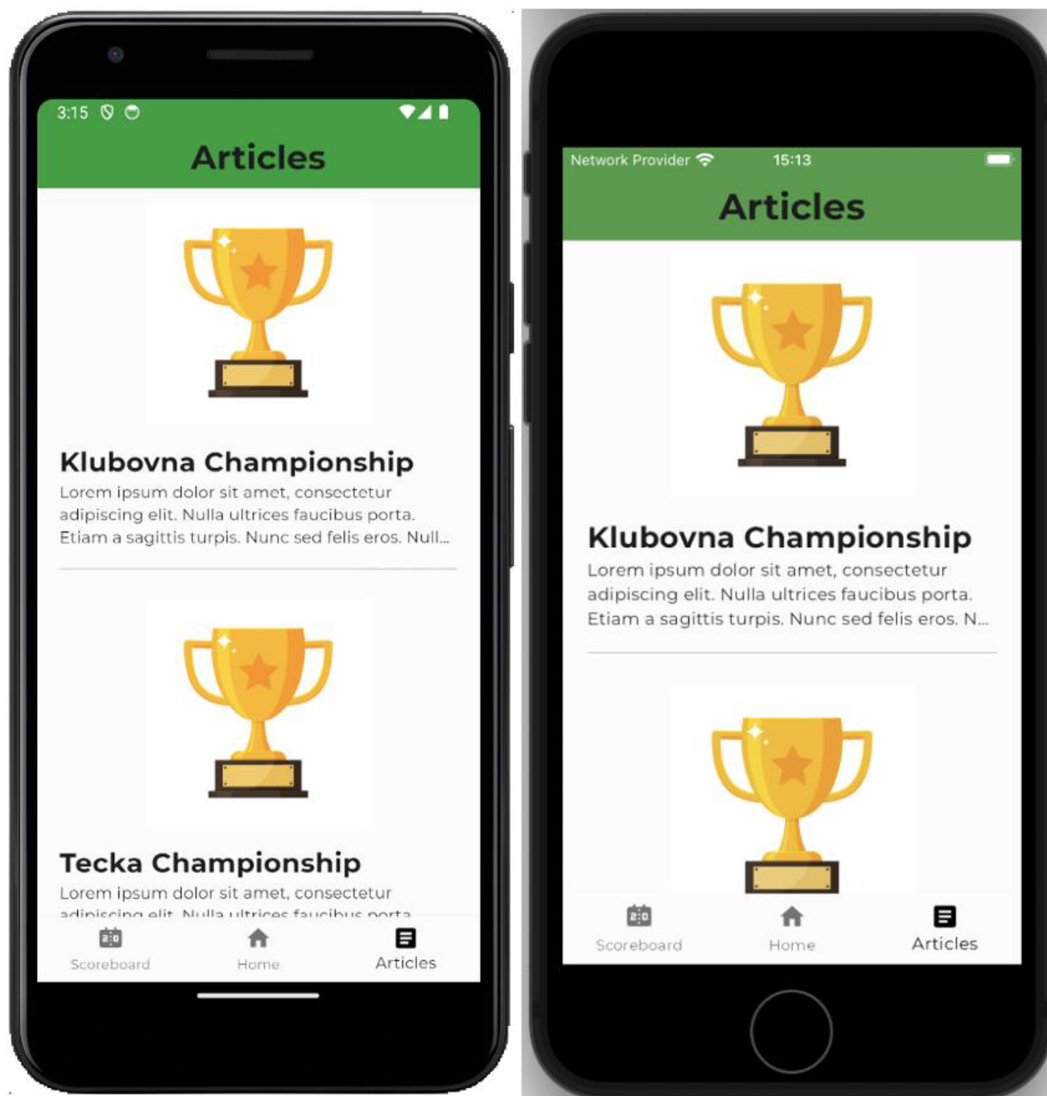
Obrázek 46 Obrazovka Match Info na zařízení platforem Android a iOS



Zdroj: Vlastní zpracování

Dále byla vytvořená obrazovka **Articles**, kde uživatel může nalézt články o událostech spjatých se stolním fotbálkem. Tyto články jsou „publikovány" jednotlivými podniky (viz obrázek 47).

Obrázek 47 Obrazovka Articles na zařízení platforem Android a iOS



Zdroj: Vlastní zpracování

Jak již bylo zmíněno, data v aplikaci jsou uložena pouze lokálně. Pro reálné využití aplikace je potřeba, aby aplikace komunikovala se serverem, odkud by byla data načítána. Aplikace by se také dala rozšířit na sociální síť, kde by uživatelé mohli mezi sebou sdílet jednotlivá utkání, přidávat podniky, udělovat jim recenze.

6 Závěr

Tato práce byla zaměřena na druhy vývoje mobilních aplikací s důrazem na cross-platform vývoj mobilních aplikací s důrazem na konkrétní technologii Flutter.

Na úvod byly představeny jednotlivé platformy a možnosti vývoje mobilních aplikací včetně jejich výhod a nevýhod. Dále bylo představeno objektově orientované programování a programovací jazyk Dart, který je nedílnou součástí této práce, jelikož představuje základní stavební kámen technologie Flutter, na kterou je práce zaměřena. Poté proběhl popis Flutter frameworku, který detailním způsobem přiblížil způsob jeho fungování. Kromě Flutter byly představeny další technologie pro cross-platform vývoj. Na závěr teoretické části byla přiblížena problematika UI/UX design a jazyku UML.

Ke zpracování praktické části práce byly využity poznatky získané z rozboru odborných zdrojů. V první části byl navržen diagram tříd a wireframe pro aplikaci. Následně byl implementován funkční prototyp aplikace s podrobným popisem kódu. Na závěr praktické části proběhlo testování prototypu, které ověřuje funkčnost aplikace. Během implementace byly využity balíčky třetích stran, které umožnily efektivnější vývoj aplikace.

7 Seznam použitých zdrojů

1. Time Spent Using Smartphones (2024 Statistics), 2024. *Exploiding Topics* [online]. [cit. 2024-03-15]. Dostupné z: https://explodingtopics.com/blog/smartphone-usage-stats
2. Annual number of mobile apps downloads worldwide 2023 | Statista, 2024. *Statista* [online]. [cit. 2024-03-14]. Dostupné z: https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads/
3. Mobile Operating System Market Share Worldwide | Statcounter Global Stats, 2024. *GS.Statcounter* [online]. [cit. 2024-03-14]. Dostupné z: https://gs.statcounter.com/os-market-share/mobile/worldwide
4. ALMISREB, Ali Abd a Hana Hadžo MULACIĆ, 2010. *A review on mobile operating systems and application development platforms*. Borås, Sweden. Vědecký článek. University of Borås.
5. UTTARWAR, Pritee S., Rani P. TIDKE, Deepak S. DANDWATE a Umesh J. TUPE, 2021. A Literature Review on Android -A Mobile Operating system. *IRJET* [online]. 6 [cit. 2024-03-29]. ISSN 2395-0056. Dostupné z: https://www.researchgate.net/publication/354576500_A_Literature_Review_on_Android_-A_Mobile_Operating_system
6. Platform architecture | Android Developers, 2023. *Android Mobile App Developer Tools - Android Developers* [online]. [cit. 2024-03-16]. Dostupné z: https://developer.android.com/guide/platform
7. IOS architecture | RedFox Security, 2022. *Penetration Testing Services - Pen Testing Services Company* [online]. [cit. 2024-03-16]. Dostupné z: https://redfoxsec.com/blog/ios-architecture/
8. Architecture of IOS Operating System - GeeksforGeeks, 2023. *Geeksforgeeks.com* [online]. [cit. 2024-03-16]. Dostupné z: https://www.geeksforgeeks.org/architecture-of-ios-operating-system/
9. ISLAM, Md. Rashedul, Md. Rofiqul ISLAM a Tohidul Arafhin MAZUMDER, 2010. Mobile Application and Its Global Impact. *International Journal of*

Engineering & Technology IJET-IJENS [online]. **10**(6), 7 [cit. 2024-03-29]. Dostupné z: https://www.researchgate.net/publication/308022297_Mobile_application_and_its_global_impact

10. CLARK, John. F., b.r. *History of Mobile Applications: MAS 490: Theory and Practice of Mobile Applications* [Online].
11. Native vs. Cross Platform Apps | Microsoft Power Apps, b.r. *Microsoft* [online]. [cit. 2024-03-20]. Dostupné z: https://powerapps.microsoft.com/en-us/native-vs-cross-platform-apps/
12. CARRINGTON, Matthew, 2023. Native Mobile App Development. *Velvetech* [online]. [cit. 2024-03-29]. Dostupné z: https://www.velvetech.com/blog/native-mobile-app-development/
13. Flutter architectural overview | Flutter, b.r. *Flutter documentation* [online]. [cit. 2024-03-17]. Dostupné z: https://docs.flutter.dev/resources/architectural-overview
14. *Beginner's Guide to Kotlin Programming*, 2021. Springer Nature. ISBN 9783030808921.
15. Developer Policy Center, b.r. *Google* [online]. [cit. 2024-03-20]. Dostupné z: https://play.google/developer-content-policy/
16. Restricted Content - Play Console Help, b.r. *Google* [online]. [cit. 2024-03-20]. Dostupné z: https://support.google.com/googleplay/android-developer/topic/9877466
17. Impersonation - Play Console Help, b.r. *Google* [online]. [cit. 2024-03-20]. Dostupné z: https://support.google.com/googleplay/android-developer/topic/9969539
18. Intellectual Property - Play Console Help, b.r. *Google* [online]. [cit. 2024-03-20]. Dostupné z: https://support.google.com/googleplay/android-developer/topic/9876963
19. Privacy, Deception and Device Abuse - Play Console Help, b.r. *Google* [online]. [cit. 2024-03-20]. Dostupné z: https://support.google.com/googleplay/android-developer/topic/9877467

20. Spam and Minimum Functionality - Play Console Help, b.r. *Google* [online]. [cit. 2024-03-26]. Dostupné z: https://support.google.com/googleplay/android-developer/topic/9876964
21. Malware - Play Console Help, b.r. *Google* [online]. [cit. 2024-03-26]. Dostupné z: https://support.google.com/googleplay/android-developer/topic/9975838
22. Mobile Unwanted Software - Play Console Help, b.r. *Google* [online]. [cit. 2024-03-26]. Dostupné z: https://support.google.com/googleplay/android-developer/topic/9969691
23. PYPL PopularitY of Programming Language index, 2024<. *PYPL PopularitY of Programming Language* [online]. [cit. 2024-03-24]. Dostupné z: https://pypl.github.io/PYPL.html
24. History of Swift - Swift Programming for Mobile App Development, b.r. *Educative* [online]. [cit. 2024-03-16]. Dostupné z: https://www.educative.io/courses/swift-programming-mobile-app/history-of-swift
25. App Review Guidelines - Apple Developer, 2024. *Apple Developer* [online]. [cit. 2024-03-16]. Dostupné z: https://developer.apple.com/app-store/review/guidelines/#introduction
26. VERMA, Nishkarsh a Saurabh SAMBHAV, 2020. DEVELOPMENT OF iOS: A REVOLUTIONARY TRANSFORMATION AND THE FUTURE. *IJARET* [online]. **11**(6), 9 [cit. 2024-03-29]. Dostupné z: https://www.researchgate.net/publication/342625092_Development_of_iOS_A_Revolutionary_Transformation_and_the_Future
27. Jayvir, Singh. (2023). Cross Platform Mobile App Development 2023: The Ultimate Guide.
28. What is cross-platform mobile development?, 2024. *Kotlin* [online]. [cit. 2024-03-16]. Dostupné z: https://kotlinlang.org/docs/cross-platform-mobile-development.html
29. BAILEY, Thomas a Alessandro BIESSEK, 2021. *Flutter for Beginners*. 2nd ed. Packt Publishing. ISBN ISBN-10 1800565992v.
30. Cross-platform mobile frameworks used by global developers 2022 | Statista, 2022. *Statista* [online]. [cit. 2024-03-18]. Dostupné z:

https://www.statista.com/statistics/869224/worldwide-software-developer-working-hours/

31. SIKORA, Martin, 2015. *Dart Essentials*. Packt Pub. ISBN 1783989602.
32. Dart Overview | Dart, b.r. *Dart programming language* [online]. [cit. 2024-03-16]. Dostupné z: https://dart.dev/overview
33. OBAID, Alhasan A, 2024. Dart Garbage Collection. *LinkedIn* [online]. [cit. 2024-03-27]. Dostupné z: https://www.linkedin.com/pulse/dart-garbage-collection-alhasan-abo-obaid-vivxf/
34. Yilmaz, Rahime & Sezgin, Anil & Kurnaz, Sefer & Arslan, Yunus Ziya. (2019). Object-Oriented Programming in Computer Science. 10.4018/978-1-5225-7598-6.ch106.
35. Sunday, Agu & Elugwu, Felix. (2022). Object Oriented Programming Approach A Panacea for Effective Software Development. 6. 1-14.
36. Aksit, Mehmet & Bergmans, Lodewijk. (1996). Obstacles in Object-Oriented Software Development. ACM SIGPLAN Notices. 27. 10.1145/141936.141965.
37. NAPOLI, Marco, 2019. *Beginning Flutter: A Hands On Guide to App Development*. Wrox. ISBN 1119550823.
38. Flutter: Advantages, Disadvantages and Future Scopes - GeeksforGeeks, 2023. *GeeksforGeeks | A computer science portal for geeks*[online]. [cit. 2024-03-17]. Dostupné z: https://www.geeksforgeeks.org/flutter-advantages-disadvantages-and-future-scopes/
39. Building user interfaces with Flutter | Flutter, b.r. *Flutter documentation* [online]. [cit. 2024-03-24]. Dostupné z: https://docs.flutter.dev/ui
40. WALEED, Arshad, 2021. *Managing State in Flutter Pragmatically*. Packt Publishing. ISBN ISBN-10 1801070776.
41. StatefulWidget class - widgets library - Dart API, b.r. *Flutter API* [online]. [cit. 2024-03-24]. Dostupné z: https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html
42. Start thinking declaratively | Flutter, b.r. *Flutter documentation* [online]. [cit. 2024-03-24]. Dostupné z: https://docs.flutter.dev/data-and-backend/state-mgmt/declarative

43. StatelessWidget class - widgets library - Dart API, b.r. *Flutter API* [online]. [cit. 2024-03-24]. Dostupné z: https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html
44. Understanding constraints | Flutter, b.r. *Flutter documentation* [online]. [cit. 2024-03-24]. Dostupné z: https://docs.flutter.dev/ui/layout/constraints
45. Using packages | Flutter, b.r. *Flutter documentation* [online]. [cit. 2024-03-24]. Dostupné z: https://docs.flutter.dev/packages-and-plugins/using-packages
46. Testing Flutter apps | Flutter, b.r. *Flutter documentation* [online]. [cit. 2024-03-24]. Dostupné z: https://docs.flutter.dev/testing/overview
47. HAMIDLI, Nasrullah, 2023. *Introduction to UI/UX Design: Key Concepts and Principles*. Baku, Azerbaijan. Vědecký článek. Baku Engineering University, Information Technologies.
48. What is Information Architecture (IA), 2024. *Interaction Design Foundation - UX Feed* [online]. [cit. 2024-03-26]. Dostupné z: https://www.interaction-design.org/literature/topics/information-architecture
49. *The UX Book*, 2019. 2nd. Morgan Kaufmann Publishers. ISBN 978-0-12-805342-3.
50. What is Wireframing? The Complete Guide [Free Checklist]| Figma, b.r. *Figma* [online]. [cit. 2024-03-24]. Dostupné z: https://www.figma.com/resource-library/what-is-wireframing/
51. Waykar, Yashwant. (2013). "A Study of Importance of UML diagrams: With Special Reference to Very Large-sized Projects".
52. Booch, Grady & Rumbaugh, James & Jacobson, Ivar. (1999). Unified Modeling Language User Guide, The (2nd Edition) (Addison-Wesley Object Technology Series). J. Database Manag.. 10
53. Get_it | Dart package, b.r. *The official repository for Dart and Flutter packages* [online]. [cit. 2024-03-24]. Dostupné z: https://pub.dev/packages/get_it
54. Stencel, Krzysztof & Wegrzynowicz, Patrycja. (2008). Implementation Variants of the Singleton Design Pattern. 396-406. 10.1007/978-3-540-88875-8_61.
55. McDonough, J.E. (2017). Factory Design Patterns. In: Object-Oriented Design with ABAP. Apress, Berkeley, CA. https://doi-org.ezproxy.techlib.cz/10.1007/978-1-4842-2838-8_14

56. Bloc | Dart package, b.r. *The official repository for Dart and Flutter packages* [online]. [cit. 2024-03-24]. Dostupné z: https://pub.dev/packages/bloc
57. Go_router | Flutter package, b.r. *The official repository for Dart and Flutter packages* [online]. [cit. 2024-03-24]. Dostupné z: https://pub.dev/packages/go_route
58. What is UUID?, 2021. *Techtarget* [online]. [cit. 2024-03-24]. Dostupné z: https://www.techtarget.com/searchapparchitecture/definition/UUID-Universal-Unique-Identifier
59. Bloc_test | Dart Package, b.r. *The official repository for Dart and Flutter packages* [online]. [cit. 2024-03-24]. Dostupné z: https://pub.dev/packages/bloc_test
60. Main, b.r. *Patrol by LeanCode* [online]. [cit. 2024-03-24]. Dostupné z: https://patrol.leancode.co/

8 Seznam obrázků

Obrázek 1 Architektura Android Platformy 13
Obrázek 2 Architektura iOS platformy 15
Obrázek 3 Popularita Kotlin vs Java 21
Obrázek 4 Popularita jazyku Swift vs Objective-C 22
Obrázek 5 Nejvyužívanější technologie pro cross-platform vývoj 27
Obrázek 6 Platformy programovacího jazyku Dart 29
Obrázek 7 Vrstvy Flutter SDK 32
Obrázek 8 Platformové Kanály Flutter 33
Obrázek 9 Architektura Flutter web 34
Obrázek 10 Strom widgetů 36
Obrázek 11 Deklarativní přístup Flutter 36
Obrázek 12 Inherited widget 38
Obrázek 13 Widgety a strom prvků 39
Obrázek 14 Omezení a velikost widgetů 39
Obrázek 15 Informační architektura 48
Obrázek 16 Wireframe - Obrazovka Home, Pub Detail 49
Obrázek 17 Wireframe - Obrazovka articles, article detail 50
Obrázek 18 Wireframe - Obrazovky - Scoreboard, Match Info, Create Match 51
Obrázek 19 Diagram tříd 52
Obrázek 20 Inicializace aplikace 56
Obrázek 21 Definování barev pro aplikaci 57
Obrázek 22 Definování stylů pro aplikaci 57
Obrázek 23 Registrace závislostí 58
Obrázek 24 Navigace v aplikaci 59
Obrázek 25 Sestavení stránky Home, inicializace BlocProvider 59
Obrázek 26 Stavy stránky Home 60
Obrázek 27 Cubit pro stránku Home 61
Obrázek 28 Abstraktní třída PubsRepository 61
Obrázek 29 Implementace abstraktní třídy PubsRepository 62
Obrázek 30 Vykreslení stránky Home pomoc BlocBuilder 63

Obrázek 31 Obrazovka Home na zařízení platforem Android a iOS 64
Obrázek 32 Scaffold widget 64
Obrázek 33 AppBar widget 65
Obrázek 34 Column widget 65
Obrázek 35 TabBar widget 66
Obrázek 36 Expanded widget 66
Obrázek 37 TabBarView widget 67
Obrázek 38 ListView widget 67
Obrázek 39 PubCardWidget 68
Obrázek 40 Widget test 69
Obrázek 41 Bloc test 70
Obrázek 42 Integrační test pomocí Patrol 71
Obrázek 43 Report neúspěšného integračního testu Patrol 72
Obrázek 44 Report úsěšného integračního testu Patrol 72
Obrázek 45 Obrazovka Scoreboard na zařízení platforem Android a iOS 73
Obrázek 46 Obrazovka Match Info na zařízení platforem Android a iOS 74
Obrázek 47 Obrazovka Articles na zařízení platforem Android a iOS 75

Přílohy

Zdrojový kód prototypu aplikace je dostupný v přiloženém souboru včetně odkazu na GitHub repositář projektu.