Univerzita Hradec Králové

Fakulta informatiky a managementu

Katedra informačních technologií

Implementace multiplatformní aplikace za použití frameworku Flutter

Diplomová práce

Autor práce: Bc. Miroslav Brandýský

Studijní obor: Aplikovaná Informatika

Vedoucí práce: Mgr. Daniela Ponce, Ph.D.

Hradec Králové 2022

Prohlášení

Prohlašuji, že jsem diplomovou práci zpracoval samostatně a s použitím uvedené literatury.

Miroslav Brandýský v.r.

28. dubna 2022

Poděkování

Děkuji vedoucí diplomové práce Mgr. Daniele Ponce, Ph.D. za metodické vedení práce.

Anotace

Tato diplomová práce se zabývá představením problematiky při tvorbě multiplatformních aplikací a demonstrací vývoje za použití frameworku Flutter. Práce se skládá ze třech částí. První z nich nejprve popisuje způsoby vývoje multiplatformních aplikací a následně porovnává nejvíce používané frameworky. Druhá část se zabývá popisem samotného frameworku Flutter, kterému se v poslední době výrazně zvýšila popularita mezi vývojáři. Tato část přibližuje jeho strukturu a základní principy při tvorbě aplikací. Poslední část se zaměřuje na způsoby, které Flutter využívá při tvorbě multiplatformních aplikací. Ty jsou současně prezentovány na demonstrační aplikaci, kterou je možné spustit na zařízeních s operačním systémem Android a Windows a ve webovém prohlížeči.

Anotation

Title: Implementation of a multiplatform application using the Flutter framework

This diploma thesis deals with the introduction of issues in the creation of multiplatform applications and demonstration of development using the Flutter framework. The work consists of three parts. The first of them describes the ways of developing multiplatform applications and then compares the most used frameworks. The second part deals with the description of the Flutter framework itself, which has recently significantly increased its popularity among developers. This part introduces its structure and basic principles in creating applications. The last part focuses on the ways that Flutter uses to create multiplatform applications. These are also presented on a demo application that can be run on Android and Windows devices and in a web browser.

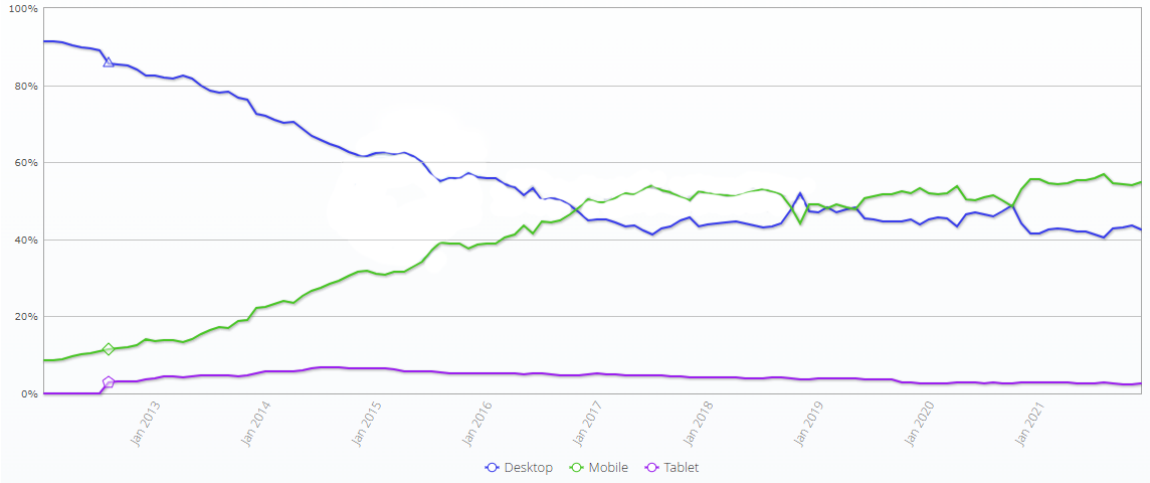
Obsah

Seznam obrázků

Seznam zdrojových kódů

1 Úvod

V posledních letech stoupá popularita mobilních zařízení, která mohou mít lidé neustále u sebe, díky tomu, že poskytují rychlý přístup k potřebným informacím a službám. Tento trend popisuje Obrázek 1, který současně ukazuje, že podíl mobilních telefonů v roce 2016 překonal klasické počítače a prvenství si drží do dnes. Situace je velmi odlišná podle regionů, v Evropě je tržní podíl mobilů necelých 46 procent, v Africe víc než 60 procent. Mobilní telefony si lidé oblíbili nejen díky jejich přenositelnosti, ale také díky jejich chytrým senzorům, které zdokonalují uživatelský zážitek a rozšiřují jejich možnosti. Současně s tímto trendem se také zvyšuje výpočetní výkon, který nosíme v kapsách, a proto nic nenasvědčuje tomu, že by lidé v budoucnu začali ztrácet zájem o mobilní telefony.



Obrázek 1: Vývoj zastoupení platform desktop, mobile a tablet [1]

Co se týče trhu s mobilními telefony, tak v současné době dominují dva operační systémy - Google Android a Apple iOS. Konkrétně je to 72.4% pro Android a 26.7% pro iOS. Zbytek zastupují KaiOS, Windws a další, ale jejich podíl je zanedbatelný. [14] Při vývoji aplikací pro mobilní telefony musí sice vývojáři brát ohled jen na dva operační systémy, ale protože každý operační systém si šel vlastní cestou vývoje a vykazuje technologické rozdíly, díky kterým se systémy liší, vyžadují oba speciální přístup při vývoji aplikací. Proto při vývoji dochází často k tomu, že jsou vyvíjeny dvě stejné aplikace paralelně akorát pro rozdílné platformy. To však může vést následně k tomu, že sice existuje pro každý operační systém daná aplikace, ale kvůli rozdílnosti

systémů nemusí obsahovat stejnou funkcionalitu. Mobilní jsou vyvíjeny šesti různými přístupy: nativní, webové, hybridní, interpretované, modelem řízené a aplikace založené na kompilaci [15]. Nativní aplikace běží v operačním systému zařízení a musí být přizpůsobeny pro různá zařízení. Webové aplikace vyžadují webový prohlížeč na mobilním zařízení. Hybridní aplikace jsou webové aplikace zabalené do nativního režimu. Interpretované aplikace interpretují značkovací jazyk a vracejí komponenty, které jsou specifické pro danou platformu. Modelem řízené aplikace mají generátory kódu, které transformují model nezávislý na platformě na zdrojový kód specifický pro platformu. Aplikace založené na kompilaci mapují vstupní aplikace na cílovou reprezentaci.

Vývojáři se zajímají hlavně o vytváření nativních aplikací, protože mohou využívat nativní funkce zařízení (např. Fotoaparát, senzory, akcelerometr, geolokace). [16] Ty je ale v dnešní době možné používat i v aplikacích, vyvíjených některým z ostatních pěti přístupů. Proto se tato práce zabývá porovnáním aktuálních přístupů při vývoji mobilních aplikací. Následně rozebere principy a metodiky potřebné pro multiplatformní vývoj za použití frameworku Flutter. Ty budou prezentovány na demonstrační aplikaci. Tento vývoj bude ohodnocen, jestli je přínosný pro multiplatformní vývoj.

2 Cíl práce

Cílem této diplomové práce je porovnat současné technologie a přístupy pro vývoj multiplatformních aplikací a vytvořit multiplatformní aplikaci za použití frameworku Flutter, která bude spustitelná na mobilních telefonech, chytrých hodinkách, počítačích s operačním systémem Windows a ve webovém prohlížeči. Práce bude zkoumat metody a principy Flutteru při tvorbě multiplatformních aplikací. Dále bude cílem sledovat množství kódu, který je sdílené mezi jednotlivými platformami, protože tento údaj je klíčový v efektivitě psaní multiplatformních aplikací.

Výsledkem práce bude demonstrační aplikace bude sloužit k prezentaci konkrétních principů používaných ve Flutteru při tvorbě multiplatformních aplikací.

3 Metodika zpracování

Nejprve budou popsány způsoby, kterými je možné vyvíjet multiplatformní aplikace. Dále práce seznámí s aktuálně nejvíce používanými frameworky, kde stručně popíše jejich principy fungování, výhody a nevýhody. Následně práce podrobněji rozebere samotný frameworku Flutter. Ten bude použit v praktické části práce, kde budou sledovány jeho přístupy při tvorbě multiplatformních aplikací. Konkrétní ukázky budou prezentovány na demonstrační aplikaci, na které se bude vyhodnocovat efektivita tohoto frameworku.

4 Multiplatformní vývoj

Při vytváření aplikací pro více platforem se vývojáři mohou rozhodnout mezi několika přístupy, které ovlivní celý vývoj. Rozlišují se v časové náročnosti napsání aplikace nebo například v množství funkcionalit, které výsledná aplikace bude umět vykonávat. Toto rozhodnutí je klíčové, a proto hraje velkou roli v samotném vývoji.

4.1 Typy multiplatformního vývoje

Vývoj mobilních aplikací nabízí nativní, webový, progresivní, hybridní, interpretovaný, kompilovaný a modelem řízený přístup vývoje. Každý z nich má své klady i zápory a je na vývojáři, který z nich upřednostní. Následující část stručně popisuje každý z nich.

4.1.1 Nativní aplikace

Nativní přístup při vývoji znamená, že pro každou platformu probíhá separátní vývoj a výsledný kód není multiplatformní a žádné jeho části se nedají znovu použít na jiné platformě. Proto je tato cesta nejnákladnější z hlediska času na vývoj i na potřebné znalosti, které se pro jednotlivé platformy liší. Na druhou stranu nativní přístup zajišťuje nejlepší přístup ke zdrojům samotného zařízení, jako je práce se soubory, internetovým připojením nebo například jeho senzory. Další výhodou je nejlepší výkon aplikace, protože má přímý přístup k hardwaru, a lepší uživatelský zážitek díky lepší optimalizaci. Na druhou stranu přinášejí nevýhodu v podobě komplikovaného dodávání aktualizací oproti třeba webovému přístupu. Díky instalaci aplikace přímo do zařízení je sice možné ji využívat i bez přístupu k Internetu, ale při vydání nové verze se musí vývojáři vypořádat s faktem, že ne všichni uživatelé provedou aktualizaci okamžitě. Zde musí dojít k vynucení provedení aktualizace (např. varovný text při startu aplikace) nebo zajištění zpětné kompatibility. [17]

4.1.2 Webové a progresivní aplikace

Mobilní webová aplikace je v podstatě webová aplikace vyvinutá pomocí webových technologií jako klasické webové stránky. Jsou zde použity technologie HTML, CSS a JavaScript. Tato aplikace je optimalizována pro rozlišení obrazovky mobilního telefonu a tabletu. Díky tomu není třeba aplikaci nainstalovat do zařízení, ale je spuštěna v příslušném prohlížeči dané platformy. Typicky se jedná o Google Chrome pro

Android a Operu pro iOS. S rostoucí standardizací a podporou různých API v prostředích mobilních prohlížečů v posledních letech je možné přistupovat k nativním funkcím a senzorům zařízení, jako je senzor GPS a ukládání dat. [18]

Výhodou tohoto řešení je udržování aktuální verze na straně klienta, protože po každém spuštění aplikace dojde k volání vzdáleného serveru, který vrací aktuální verzi aplikace. Na koncovém zařízení (na klientovi) nemusí být nainstalovaná samotná aplikace, stačí webový prohlížeč, přes který se aplikace načte. Dále toto řešení umožňuje vyvinutí jedné aplikace pro více platforem současně. Oproti tomu je velkou nevýhodou závislost na online připojení, bez kterého není možné aplikaci používat. Dále mohou nastat problémy, při nestabilním nebo pomalém připojení, kdy aplikace nemusí fungovat správně nebo minimálně nebude uživatelsky příjemná na používání.

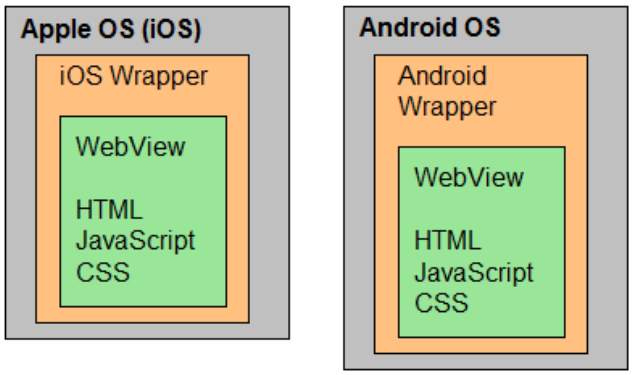
Aby se kompenzoval vzhled a dojem z webových stránek, Google nedávno představil nový přístup - Progresivní webové aplikace (dále PWA). Ty vylepšují tradiční webové aplikace pomocí takzvaných servisních pracovníků (umožňujících spouštění kódu ve vlákně na pozadí), manifestu webové aplikace (pro poskytování metadat), off-line funkcí a uživatelského prostředí [19]. To vedlo k možnostem, které dříve nebyly k dispozici pro klasické webové aplikace spouštěné přes webový prohlížeč. I když má PWA lepší přístup k funkcím zařízení než klasické webové aplikace, jsou stále omezené. PWA nemůže přistupovat k funkcím zařízení, které nejsou přístupné prostřednictvím webového prohlížeče. [18]

4.1.3 Hybridní aplikace

Hybridní přístup kombinuje prvky mobilního i webového vývoje. Umožňuje vyvíjet mobilní aplikace s využitím stejných znalostí, které je potřeba pro vývoj webových stránek. Tento přístup umožňuje zabalení souborů HTML, CSS a JavaScript do instalovatelné aplikace. Tyto soubory jsou následně vykresleny pomocí WebView. Jedná se o kontejner, který umožňuje zobrazit webový obsah jako součást aplikace, ale postrádá některé funkcionality, kterými disponují plnohodnotné prohlížeče [20]. Oproti klasickému webovému prohlížeči skrývá typické ovládací prvky jako jsou například: adresní řádek, záložky nebo nastavení. Je zde možné využívat více funkcí zařízení oproti klasickým webovým aplikacím. Jedná se například o přístup k seznamu kontaktů nebo Bluetooth a síťového připojení, které jsou poskytovány prostřednictvím API jazyka JavaScript, z nichž každé funguje jako rozhraní FFI (Foreign Function Interface) mezi komponentou WebView a základním nativním kódem. Hybridní apli-

kaci je možné stáhnout z obchodů s aplikacemi, nainstalovat do telefonu a používat offline stejně jako nativní aplikace.

Kombinace přístupu k funkcím zařízení, snadné implementace uživatelského rozhraní prostřednictvím známých webových jazyků a nativního chování jsou hlavní výhody tohoto přístupu. Možnou nevýhodou tohoto přístupu jsou uživatelská rozhraní založená na HTML, protože se mohou chovat odlišně od nativních prvků uživatelského rozhraní. [18]

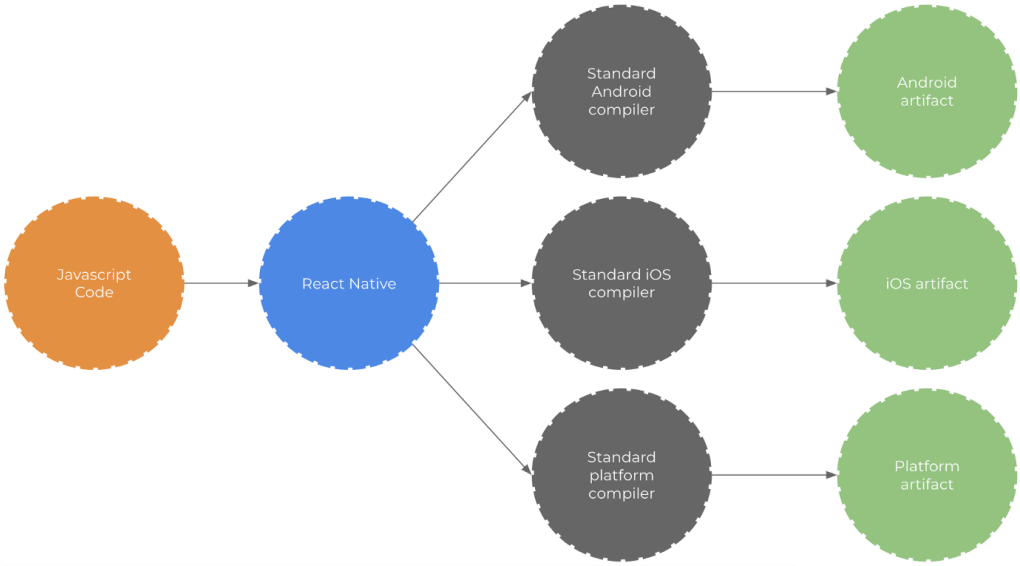


Obrázek 2: Hybridní aplikace. Zdroj [2]

4.1.4 Interpretované aplikace

Na rozdíl od hybridního přístupu, který používá webový prohlížeč skrze modul WebView, aplikace vytvořené s tímto přístupem se dodávají se samostatnou komponentou modulu runtime [21]. Tento přístup se obvykle označuje jako interpretovaný přístup nebo JavaScript-to-Native pro aplikace založené na JavaScriptu. Dodavatel architektury musí vyvinout běhovou vrstvu pro všechny cílové platformy. Vývojáři aplikací pak mohou použít společné rozhraní API pro přístup k základním funkcím dané platformy. Kód aplikace je obvykle napsán pomocí programovacího jazyka, jako je JavaScript (např. v React Native a NativeScript) nebo C# (Xamarin). Také místo poskytování přístupu k nativním funkcím prostřednictvím komponenty WebView, frameworky tohoto přístupu obvykle vystavují proprietární přemosťovací systémy založené na pluginech, které umožňují vyvolání FFI v nativním kódu. Používají kombinaci nativně vykreslených uživatelských rozhraní v kombinaci s modulem runtime pro byznys logiku založenou na JavaScriptu. To je možné pomocí interpretů jazyka

(např. JavaScriptCore nebo V8), které interpretují značkovací jazyk a vracejí komponenty, které jsou specifické pro danou platformu. Princip fungování je takový, že se pomocí JavaScriptu definuje prvek, který se má vykreslit. Daný framework podle konkrétní platformy zvolí, co se má vykreslit. [18]



Obrázek 3: Princip fungování React Native. Zdroj [3]

4.1.5 Modelem řízené aplikace

Modelem řízený přístup se zaměřuje na model jako abstraktní reprezentaci systému, ze kterého je odvozen skutečný softwarový artefakt [22]. V kontextu mobilních zařízení tento přístup umožňuje vývoj multiplatformních aplikací s využitím vyšší úrovně abstrakce než zdrojový kód, často prostřednictvím použití textových nebo grafických jazyků specifických pro doménu (DSL) nebo modelování pro obecné účely, jako je například UML. Následně generátory kódu (jeden na cílovou platformu) transformují model nezávislý na platformě na zdrojový kód specifický pro platformu, který pak může být zkompilován a sestaven do každé mobilní platformy podporované rozhraním. Výsledné aplikace tak mohou plně využít potenciál platformy, protože jsou, v ideálním případě, k nerozeznání od nativních aplikací. [23]

4.1.6 Aplikace založené na kompilaci

Přístupy založené na kompilaci (tzv. transkompilátory) mají za cíl opětovné použití nativní aplikace mapováním vstupní aplikace na cílovou reprezentaci. To se může stát na úrovni byte kódu. Vzhledem ke složitosti na nízké úrovni abstrakce a rozdílům mezi příslušnými platformami se přístupy založené na kompilaci obvykle zaměřují na konkrétní aspekty aplikace, jako je byznys logika, a potřebují ruční dodatky k replikaci všech funkcí aplikace. Mezi příklady patří Google Flutter framework, který je nejnovějším frameworkem s tímto přístupem. Hlavním rozdílem mezi Flutterem a interpretovaným přístupem je to, že nevykresluje nativní komponenty uživatelského rozhraní. Místo toho Flutter ponechává veškeré vykreslování na Skia Graphics Engine, který je schopen znovu vytvořit vzhled nativních uživatelských rozhraní prostřednictvím Skia Canvas. Při sestavování v režimu ladění aplikace Flutter navíc obsahuje virtuální počítač Dart potřebný pro vylepšené prostředí pro vývojáře, včetně funkcí, jako je například opětovné načtení za provozu. Byznys logika v jazyce Dart spolu se sadou Flutter SDK jsou předem kompilovány do nativních knihoven (ARM/x86), díky čemuž nemusí dojít k použití interpretů. [18] [24]

4.2 Multiplatformní frameworky

Operační systémy, které se v současné době používají na mobilních zařízeních, tj. Android, Apple a další, nesdílejí žádné API nebo programovací jazyk. Mobilnímu vývoji to tak ukládá kritické rozhodnutí. Vzhledem k tomu, že vývojáři usilují o nejvyšší počet uživatelů, může volba podporovat pouze jeden konkrétní mobilní operační systém a ostatní nepodporovat dramaticky snížit celkový počet uživatelů aplikace. Na druhou stranu podpora všech dostupných operačních systémů výrazně zvyšuje náklady na vývoj, pokud jde o požadovaný čas, lidi pracující na aplikaci a potřebné programovací dovednosti. Vývojáři mobilních aplikací usilují o co nejširší okruh cílových uživatelů, tedy zákazníků. [25]

K vyřešení tohoto problému byla v posledních několika letech věnována zvláštní pozornost frameworkům, které nabízejí vyvíjet aplikace napříč platformami. Tyto frameworky se v mnohém liší a před samotným vývojem musí vývojář důkladně zvážit všechna pro a proti daných alternativ. Sdílejí ale jednu myšlenku - umožnit rychlejší vývoj aplikace na více operačních systémech díky sdílenému kódu pro všechny operační systémy podporované frameworkem. Komplexnější části mohou být ovšem

rozděleny pro jednotlivé distribuce, protože operační systémy nabízejí jiné možnosti, takže je daná logika napsána víckrát nebo pro nějaký systémy úplně vypuštěna. [25]

Heterogenita předních mobilních platforem, pokud jde o uživatelská rozhraní, uživatelskou zkušenost, programovací jazyk a ekosystém, učinila freameworky pro různé platformy populární. Ty pomáhají vytvářet mobilní aplikace, které lze spouštět napříč cílovými platformami (obvykle Android a iOS) s minimálním až žádným kódem specifickým pro platformu. Vzhledem k možnostem úspor nákladů a času, které byly s příchodem takových frameworků zavedeny, se multiplatformní vývoj v posledních letech těší čím dál tím větší popularitě. Při zkoumání souboru znalostí se však často setkáváme s diskusemi o nevýhodách těchto frameworků, zejména s ohledem na výkon aplikací, které generují. [18]

Následující část práce porovnává aktuální technologická řešení v multiplatformním vývoji aplikací. Porovnají se primárně aktuální popularita a použitelnost. Popularita je ve světě technologií klíčový ukazatel, který poměrně jasně indikuje, jak se daná technologie v posledních letech používá a jestli je tento trend rostoucí, klesající nebo stagnuje. Díky tomu se vývojáři mohou vyhnout technologii, která ztrácí popularitu v komunitě a tím pádem i podporu na veřejných fórech, kde si vývojáři navzájem radí. Tato data se primárně získávají ze stránky Google Trends, která vizualizuje četnost vyhledávání. Další neméně důležitý faktor je použitelnost - jinými slovy, co všechno daná technologie může nabídnout. Zde je potřeba zvážit jaké jsou aktuální potřeby daného projektu, ale také i případné možnosti do budoucna, které by při zvolení nevhodné technologie mohly přinést komplikace v podobě nutné změny frameworku.

4.2.1 Flutter

Flutter je multiplatformní framework, jehož cílem je poskytnout nástroje pro vývoj vysoce výkonných mobilních aplikací. Jeho první veřejné vydání bylo v roce 2016 společností Google a od té doby získává pravidelné aktualizace. Aplikace vyvinuté ve Flutteru mohou běžet nejen na Androidu a iOS, ale také jako webové, desktopové a vestavěné aplikace. To přináší velikou flexibilitu, kde jeden kód může být znovu použit i pro jiné platformy [26]. Oproti ostatním frameworkům se odlišuje tím, že implementuje vlastní vykreslovací vrstvu, díky které má plnou kontrolu nad tím, co a kde se vykreslí.

Flutter přistupuje k vývoji multiplatformních aplikací tím nejradikálnějším způsobem. Poskytuje vlastní sadu objektů rozhraní, vykreslování a modul, který imple-

mentuje animace, grafiku, soubory a síťové vstupy a výstupy mezi mnoha dalšími základními knihovnami [27].

Projekt Flutter je napsán v programovacím jazyce Dart a AOT (Ahead-of-time) kompilován do nativní architektury platformy, čímž se dosahuje vysoké rychlosti. Na nejvyšší úrovni Flutter poskytuje widgety, ze kterých se skládají vetší celky, aby vytvořily nejběžnější objekty rozhraní, na které je uživatel zvyklý na platformách iOS a Android. Vzhledem k tomu, že Flutter jde cestou otevřené a vrstvené architektury, mohou vývojáři vytvářet vlastní widgety skládající jiné widgety na libovolné úrovni vrstvené architektury. Ty lze pak sdílet mezi ostatní vývojáře v oficiálním úložišti balíčků. Ve skutečnosti je to způsob, jakým tým Flutter vytvořil stávající widgety na vysoké úrovni a neexistuje žádná překážka z frameworku pro vývojáře, aby udělali totéž. [28][29]

4.2.2 React Native

React Native pochází od společnosti Meta Platforms, která se rozhodla rozšířit pole působnosti po velkém úspěchu jejich frameworku React, který je zaměřený na tvorbu webových stránek. React Native začal jako interní projekt uvnitř společnosti a jeho původním cílem bylo sjednotit vývojový proces pro iOS a Android. Nicméně, jak framework výrazně roste, tak React Native může být již nasazen i na jiné platformy, jako jsou Windows, Web a Tizen. Tento framework je velmi populární a přispívají k tomu nejen jednotliví vývojáři a Meta Platforms, ale také řada technologických gigantů, jako jsou Microsoft a Samsung, kteří hrají důležitou roli při vývoji. Jednou z jeho největších předností je, že přináší moderní webové techniky do vývoje mobilních aplikací, aniž by to mělo vliv na funkce a výkon. I když jsou aplikace React Native většinou napsány a pracují na jádru JavaScriptu, neznamená to, že aplikace React Native jsou hybridní nebo HTML5 aplikace. Použití základního nativního rozhraní umožňuje aplikaci vykreslovat zobrazení a přistupovat k nativnímu hardwaru, jako je kamera a úložiště. [26]

4.2.3 Xamarin

Xamarin je řešení pro vývoj mobilních aplikací napříč platformami, které vytváří jednotné prostředí pro vývojáře. Místo psaní kódu v Objective-C a Swift pro iOS a Javě pro Android umožňuje vývojářům psát kód v jazyce C#. Každá implementace aplikace je vyvíjena nezávisle na cílové platformě a současně se zachovávají nativní

uživatelská rozhraní. V rámci každé aplikace je přímý přístup k nativním rozhraním API, což umožňuje vývojářům přístup ke všem nativním hardwarovým funkcím (např. senzorům). Sdílený kód pro platformy zahrnuje položky, jako jsou datové modely, byznys logiky, cloudové integrace a přístup k databázím. Pokud projekt vyžaduje použití knihovny napsané nativně, může vývojář vytvořit nativní vazbu, která k ní umožňuje přístup prostřednictvím volání jazyka C#. Výkon, který nabízí Xamarin, se blíží nativní rychlosti a v určitých situacích může fungovat lépe. Při použití tohoto frameworku je možné znovu použít až 75% kódu. Po dokončení vývoje lze aplikaci zkompilovat a nasadit do příslušného obchodu s aplikacemi podle cílové platformy. [30]

4.2.4 NativeScript

NativeScript je nedávný open source projekt, který umožňuje generování nativních aplikací pomocí JavaScriptu. Kromě toho může být aplikace vyvinuta pomocí TypeScript, což je bezplatný, open source jazyk vyvinutý společností Microsoft, který se rozšiřuje na JavaScript, v podstatě přidává statické psaní a objekty založené na třídách. V tomto smyslu je při kompilaci aplikace kód TypeScript přeložen do kódu jazyka JavaScript. [31]

NativeScript poskytuje multiplatformní modul, který umožňuje získat nativní aplikace z kódu JavaScript. Tento modul umožňuje konzistentní přístup k funkcím nabízeným zařízením a jeho základní platformou z kódu JavaScript. Podobně lze uživatelská rozhraní definovat pomocí kódu JavaScript, HTML dokumentů a souborů CSS nezávisle na skutečných nativních komponentách. Při kompilaci aplikace je část multiplatformního kódu přeložena do nativního kódu, zatímco zbývající kód je interpretován za běhu. Obr. 2 ukazuje reprezentaci vnitřní architektury NativeScriptu.

NativeScript je jedním z framewroků, které umožňují nativní vývoj pro mnoho různých platforem. V současné době podporuje operační systémy iOS a Android a umožňuje konstrukci progresivní webové aplikace. Aplikaci můžeme vyvíjet v rámci běžného kódu JavaScript/TypeScript nebo použít další technologie jako Angular nebo Vue.js. V podstatě vám NativeScript pomáhá pracovat na vašem zařízení s nástroji pro spouštění kódu (Android Runtime a iOS Runtime), které fungují jako kompilátory mezi kódem napsaným JavaScriptem a operačním systémem.

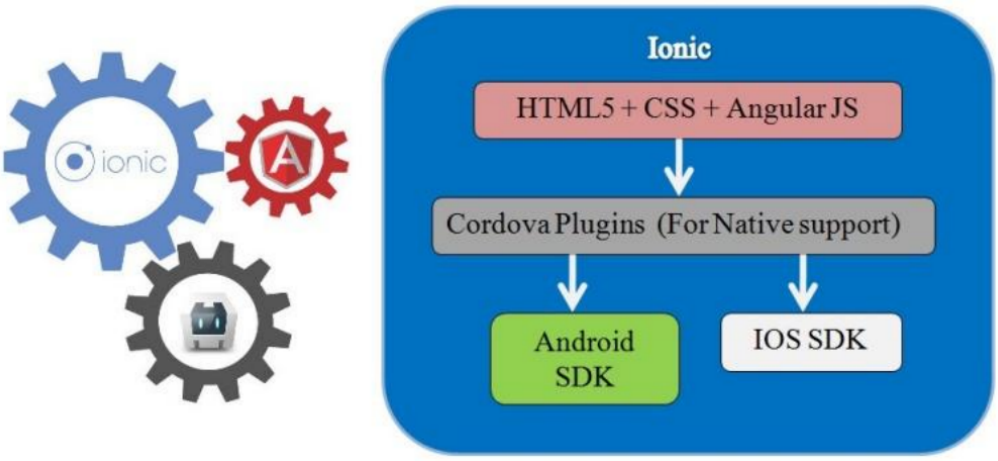
4.2.5 Cordova

Cordova je open source framework pro vývoj mobilních aplikací využívajících webové technologie (HTML, CSS, JavaScript). Webová aplikace na mobilním zařízení se zobrazuje pomocí nativní komponenty WebView. Když se spustí, Cordova vyhledá počáteční soubor (obvykle index.html) a spustí jej uvnitř WebView. Poté ponechá řízení komponentě WebView, aby uživatel mohl komunikovat s aplikací. Provoz na dvou různých zařízeních se proto může lišit, protože verze a funkčnost této komponenty závisí na operačním systému. Před použitím určitých prvků HTML nebo vlastností souboru CSS musíme ověřit, zda komponenta v zařízení podporuje jejich použití. Aplikace funguje stejným způsobem, jako by se webová aplikace zobrazila v prohlížeči, a proto nemá přímou komunikaci se zařízením. Ve vývoji Cordova nabízí JavaScript API ve formě zástrček, které zajišťují komunikaci mezi aplikací a zařízením. Projekt vždy obsahuje konfigurační .xml soubor, pomocí kterého nastavujeme parametry aplikace [10], jako je název a ID balíčku. [32]

4.2.6 Ionic

Ionic je open source framework, který používá HTML5, CSS a Javascript. Proces vývoje je poměrně rychlý a vývojáři mají přímý přístup k API s Cordovou. Ionic využívá vlastnost CSS nazvanou Syntactically Awesome Style Sheets (SASS). Ionic poskytuje bohaté komponenty uživatelského rozhraní, usnadňuje vývoj složitých mobilních aplikací a používá frameworky JavaScript MVVM a Angular JS k vylepšení prostředí. Angular JS je mobilní JS framework společnosti Google, jeho hlavními rysy jsou soulad s návrhem MVC, modularita a skládání závislostí. Ionic framework je založený na kompilační platformě Cordova a může být kompilován do různých platforem. Doba vývoje je navíc mnohem kratší a jeho kód je relativně snadno udržovatelný. [33]

Tento framework nabízí Ionic Creator. Jedná se platformu používanou při budování uživatelského rozhraní přetažením hotových komponent. Dále vývojáři mohou využít Ionic View usnadňující uživatelům spouštět a sdílet vyvinuté aplikace na více zařízeních před jejich finálním spuštěním na oficiálním obchodu s aplikacemi. Ionic Market se používá pro sdílení mobilních aplikací a může účtovat poplatky za jednotlivá stažení. Kromě toho má Ionic důvěryhodnou uživatelskou podporu spolu s dokumentací. Vývojáři mohou také využít Laboratoř Ionic pro testování nově vyvinuté aplikace na Androidu a IOS. [4]

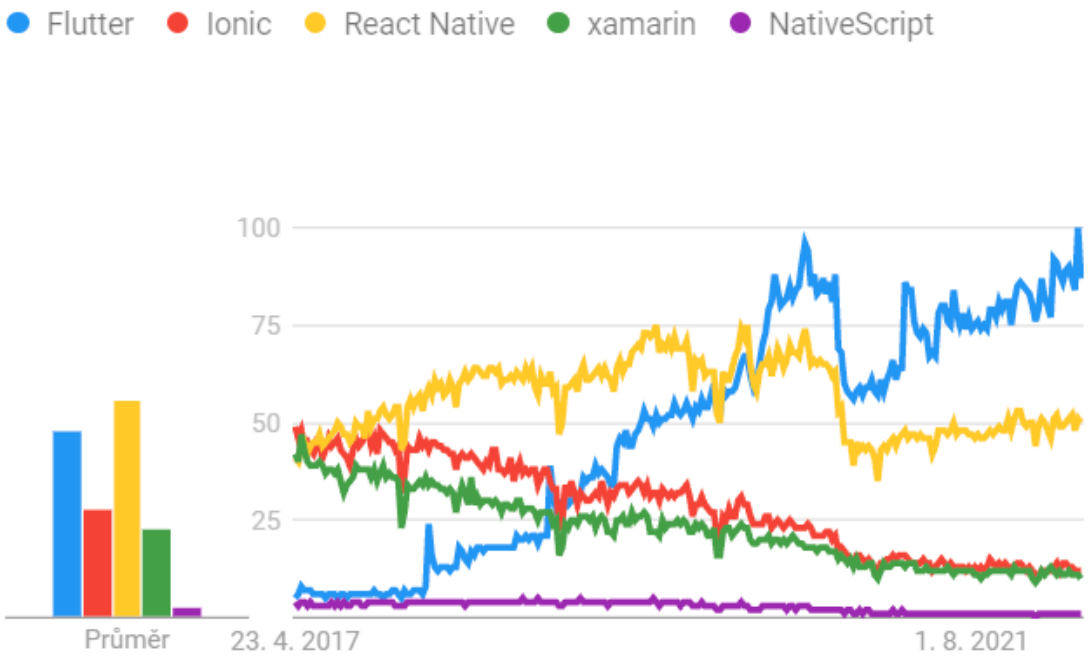


Obrázek 4: Architektura frameworku Ionic. Zdroj [4]

4.2.7 Porovnání

Výše byly popsány jednotlivé frameworky, které je možné použít při tvorbě multiplatformních aplikací. Každý z nich nabízí nějaké výhody i nevýhody. Jelikož se jedná o nejvíce používané frameworky, jako primární parametr pro porovnání byla zvolena četnost vyhledávání na webu. Detailně to popisuje Obrázek 5, na kterém je možné pozorovat několik úkazů. Prvně je možné si všimnout, že NativeScript je dlouhodobě z těchto pěti frameworků nejméně vyhledávaný. Dále je možné vidět, že Ionic spolu s Xamarinem postupně ztrácejí svou popularitu, zatímco React Native se po celou dobu drží na podobných hodnotách. Modrá křivka popisuje, jak se v posledních pěti letech Flutter zvedal na popularitě a v současné době si drží prvenství. Je dobré ale zmínit, že toto je celosvětový průměr a třeba ve Spojených Státech Amerických stále vede React Native.

Díky těmto informacím je patrné, že Flutter se těší mezi vývojáři oblibě. Proto se tato práce zaměří právě na jeho popis a zhodnocení jeho přístupu při tvorbě multiplatformní aplikace.

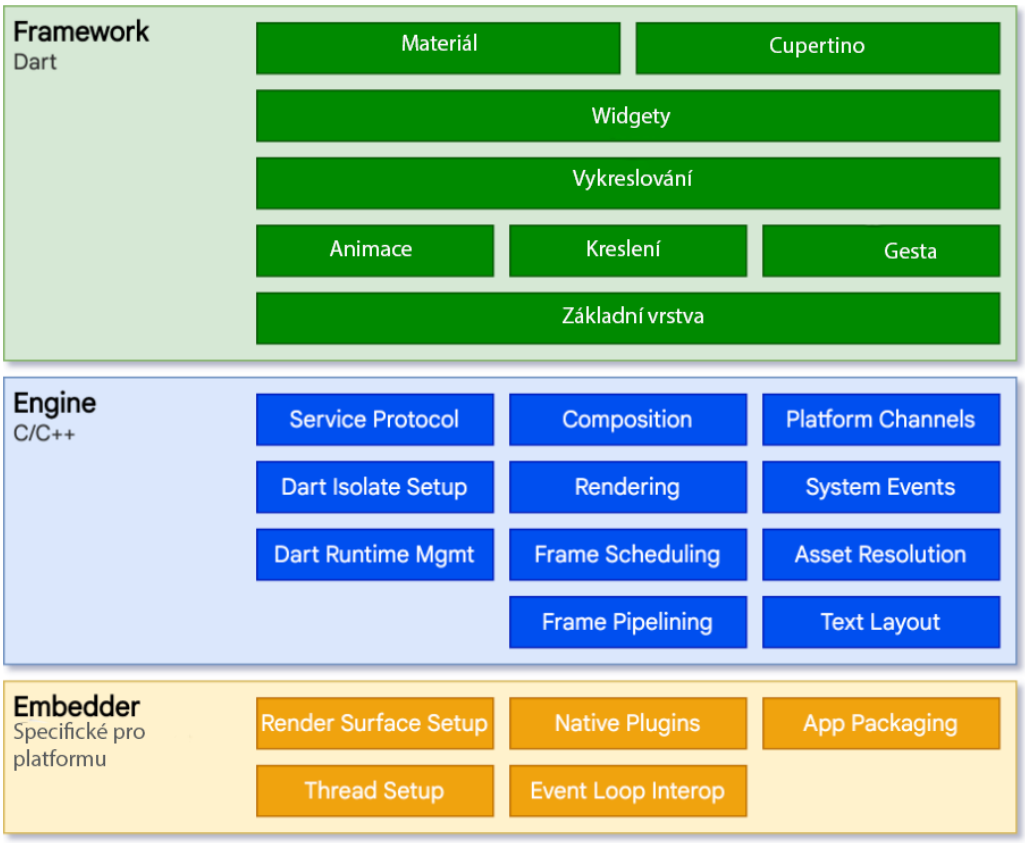


Obrázek 5: Četnost vyhledávání jednotlivých frameworků podle Google Trends za posledních 5 let. Zdroj [5]

5 Charakteristika Flutteru

Google definuje Flutter jako přenosnou sadu nástrojů uživatelského rozhraní pro vytváření nativně kompilovaných aplikací pro mobilní, webové a počítačové aplikace [34]. Je navržen tak, aby umožňoval opakované použití kódu napříč operačními systémy a zároveň umožňoval aplikacím přímé spojení se základními službami konkrétní platformy. Cílem je umožnit vývojářům vytvářet vysoce výkonné aplikace, které se na různých platformách chovají jako nativní, umožňují vytvářet specifické úpravy tam, kde se to pro danou platformu nabízí, a zároveň pracovat s co nejvíce sdíleným kódem, který by se například při psaní dvou nativních aplikací psal zbytečně dvakrát. Během vývoje běží aplikace na virtuálním počítači, který nabízí opětovné načtení změn za běhu aplikace bez nutnosti opětovné kompilace. Pro vystavení hotové aplikace se Flutter kompiluje přímo do strojového kódu konkrétní cílové platformy, ať už jsou Intel x64 nebo ARM, nebo např. do JavaScriptu pro web. [6]

Flutter je navržen jako rozšiřitelný, vrstvený systém. Existuje jako řada nezávislých knihoven, z nichž každá závisí na podkladové vrstvě (viz Obrázek 6). Žádná vrstva nemá privilegovaný přístup k vrstvě níže, a každá část úrovně frameworku je navržena tak, aby byla volitelná a nahraditelná. [6]



Obrázek 6: Architektura frameworku Flutter. Zdroj [6]

Pro operační systém jsou aplikace Flutter zabaleny stejným způsobem jako jakákoli jiná nativní aplikace daného systému. Embedder specifický pro platformu poskytuje vstupní bod aplikace, koordinuje s operačním systémem přístup ke službám, jako jsou vykreslovací plochy, přístupy, řízení vstupů a správa smyčky událostí zpráv. Embedder je napsán v jazyce, který je vhodný pro platformu. V současné době to je Java a C++ pro Android, Objective-C/Objective-C++ pro iOS a macOS a C++ pro Windows a Linux. Pomocí embedderu může být kód Flutter integrován do existující aplikace jako modul nebo samostatná aplikace. [6]

Jádrem Flutteru je Flutter engine, který je většinou napsán v C++ a podporuje prvky nezbytné pro podporu všech flutterových aplikací. Engine je zodpovědný za rastrování scén vždy, když je třeba namalovat nový snímek aplikace. Poskytuje nízkoúrovňovou implementaci základního API Flutteru, včetně grafického řešení prostřednictvím Skia.

Skia je open source 2D grafická knihovna, která poskytuje společné API, které funguje napříč různými hardwarovými a softwarovými platformami. Slouží jako grafický engine pro Google Chrome, Chrome OS, Android, Flutter a mnoho dalších produktů. [35].

Engine se dále stará o rozložení textu, souborové a síťové I/O operace, podporu přístupnosti, architekturu pluginů a sadu nástrojů Dart runtime a kompilaci. Modul je vystaven rozhraní Flutter prostřednictvím dart:ui, které zabalí základní kód jazyka C++ do tříd Dart. Tato knihovna zpřístupňuje prvky nejnižší úrovně, jako jsou třídy pro řízení vstupních, grafických a textových podsystémů. Architektura Flutteru je napsaná v jazyce Dart a sestavena z knihoven, které se skládají z řady vrstev. Ty pracují zdola nahoru [6]:

- Základní třídy a služby stavebních bloků, jako jsou animace, kreslení a gesta, které nabízejí běžně používané abstrakce nad základní třídou.
- Vykreslovací vrstva poskytuje abstrakci pro práci s rozvržením. Pomocí této vrstvy je možné vytvořit strom vykreslitelných objektů (widgetů). S těmito objekty je možné manipulovat dynamicky, přičemž se strom automaticky aktualizuje.
- Vrstva widgetů je abstrakce kompozice. Každý renderovaný objekt ve vykreslovací vrstvě má odpovídající třídu ve vrstvě widgetů. Vrstva widgetů navíc umožňuje definovat kombinace tříd, které je možné znovu použít. V této vrstvě je zaveden model reaktivního programování.
- Knihovny Material a Cupertino nabízejí komplexní sady ovládacích prvků, které používají komponenty pro vrstvu widgetů k implementaci konkrétních sad stylů Material (Android) a Cupertino (iOS).

Framework Flutter je relativně malý. Mnoho funkcí vyšší úrovně, které mohou vývojáři používat, je implementováno formou balíčků, včetně pluginů dané platformy, jako jsou kamera a webview, stejně jako funkce nezávislé na platformě, jako jsou http volání a animace, které staví na základních knihovnách Dart a Flutter. Některé z těchto balíčků pocházejí z širšího ekosystému a pokrývají služby, jako jsou platby v aplikaci, ověřování od Apple a animace. [6]

5.1 Dart

Tým Flutter nebral výběr programovacího jazyka pro jejich framework na lehkou váhu. Podle Erica Seidela (vedoucího týmu Flutter v Googlu) byl JavaScript první volbou a desítky dalších programovacích jazyků byly zvažovány, ale vyhrál Dart, programovací jazyk původně navržený pro vývoj webových aplikací a vyvíjen a udržován společností Google. Hlavními hodnotícími kritérii při výběru programovacího jazyka byly zejména potřeby vývojářů a koncového uživatele. Flutter má funkci hot reload, která umožňuje vývojářům zobrazit změny na emulátorech bez opětovného spuštění aplikace, což velmi usnadňuje vývoj, protože u větších aplikací může být spouštění samotné aplikace zdlouhavé a pro ladění UI aplikace je to nezbytné. Výsledkem je Dart VM (Virtual Machine) a jeho různých režimů pro ladění a sestavení aplikace. Dart VM v režimu ladění je schopen pracovat v režimu kompilace JIT (Just-in-time). Díky tomu je schopen dynamicky načítat a kompilovat zdrojový kód pro usnadnění opětovného načítání, ladění a dalších funkcí pro zrychlení vývoje [36]. V režimu sestavování aplikace slouží Dart VM jako běhová knihovna namísto virtuálního stroje. Zdrojový kód Dart je zkompilován AOT (Ahead of Time) a Dart VM se používá ke spuštění předkompilovaného strojového kódu s rychlým uvolňováním paměti, dynamickým vyhledáváním metod a dalšími podporami modulu runtime. [36] [29]

Silné stránky Dartu [29]:

- Je typově založený
- Podporuje kompilaci AOT a JIT, proto je ideální pro funkci hot-reload
- Je rychlejší než JavaScript

Dart byl původně vyvinut jako náhrada a nástupce JavaScriptu. Implementuje tedy většinu důležitých charakteristik standardu JavaScriptu (ES7), jako jsou klíčová slova "async"a "await". Nicméně, aby přilákal vývojáře, kteří nejsou obeznámeni s JavaScriptem, Dart má syntaxi podobnou Javě. Podobně jako u jiných systémů, které využívají reaktivní pohledy, aplikace Flutter aktualizuje strom zobrazení na každém novém vykreslovacím snímku. Nevýhodou tohoto chování je, že pro každý nový snímek budou vytvořeny nové objekty, které budou mít životnost pouze jeden snímek. Naštěstí Dart, jako moderní programovací jazyk, je optimalizován pro řízení tohoto chování na úrovni paměti pomocí generačního garbage collectoru. [26]

5.2 Generační garbage collector

Dart obsahuje garbage collector, nezbytnou součást pro přidělování a odebírání paměti. Bezstavové widgety se vytvářejí při vykreslování každého nového snímku na obrazovce, ničí se a znovu sestavují, když se změní stav aplikace nebo když už nejsou viditelné. Většinou mají velmi krátkou životnost. Pro ilustraci u aplikace s přiměřeně složitým uživatelským rozhraním lze spustit na tisíce widgetů. Protože Flutter má reaktivní přístup, kde jsou objekty nahrazovány novými při každé změně stavu, musí umět pracovat s velkým množstvím objektů, pro které je nutné umět přidělovat a odebírat paměť. Garbage collector v Dartu se skládá se ze dvou fází: young space scavenger a parallel marking and concurrent sweeping. Umí rychle alokovat paměť pro nové objekty a současně se rychle zbavovat objektů, které již nejsou potřeba, prostřednictvím integrovaného generačního garbage collectoru. [7]

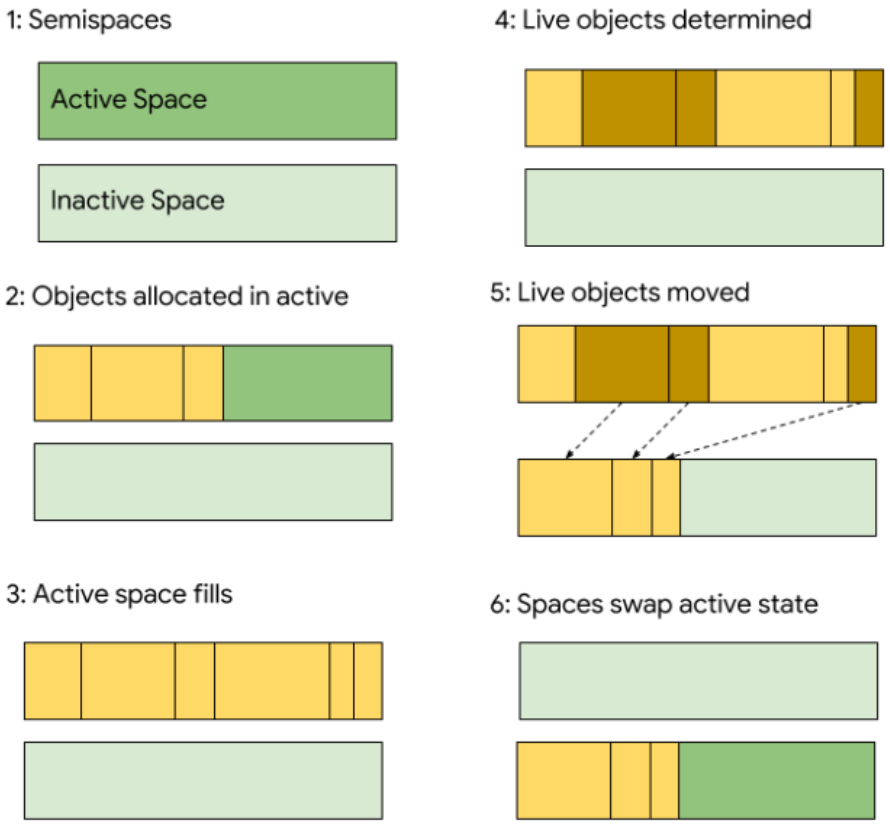
5.2.1 Plánování

Aby se minimalizovaly dopady uvolňování paměti na výkon samotné aplikace a plynulost uživatelského rozhraní, systém uvolňování paměti poskytuje spouštěče, které ho upozorní, když modul zjistí, že aplikace je nečinná a nedochází k žádné interakci uživatele. To dává systému uvolňování paměti časová okna, kdy je dobré spustit fázi garbage collectoru bez dopadu na výkon. Systém uvolňování paměti může během těchto intervalů nečinnosti také spouštět posuvnou komprimaci, což minimalizuje režii paměti snížením fragmentace paměti a následného rychlejšího čtení z paměti. [7]

5.2.2 Young Space Scavenger

Tato fáze garbage collectoru je určena k vyčištění paměti od dočasných objektů, které mají krátkou životnost, jako jsou například bezstavové widgety. I když blokuje hlavní vlákno aplikace, je mnohem rychlejší než druhá generace mark/sweep a v kombinaci s plánováním prakticky eliminuje vnímané pauzy v aplikaci při spuštění, protože se operace spouští v čase, kdy není aplikace využívána uživatelem. Princip je takový, že objekty jsou přiděleny do souvislého prostoru v paměti a při vytváření nových objektů jim je přiděleno další dostupné místo, dokud není přidělená paměť zaplněna. Dart používá přidělení ukazatele nárazu, které slouží k rychlému přidělení nového prostoru, což činí proces velmi rychlým.

Popis chování popisuje Obrázek 7. Prostor, který je přidělován objektům, se skládá ze dvou polovin, známých jako poloprostory. V jednom okamžiku se používá pouze jedna polovina, kde jedna je aktivní a druhá neaktivní. Novým objektům je přidělen prostor z aktivní poloviny a jakmile je zaplněn, aktivní objekty jsou zkopírovány z aktivní poloviny do neaktivní. Toto kopírování ignoruje mrtvé objekty, které nechá v původní polovině. Neaktivní polovina se pak aktivuje (a aktivní deaktivuje) a proces se opakuje. Pokud je potřeba určit, které objekty jsou živé nebo mrtvé, kolektor začíná kořenovými objekty, jako jsou proměnné zásobníku, a zkoumá, na co odkazují. Poté přesune odkazované objekty. Odtud prověřuje, na co tyto přesunuté objekty ukazují, a přesouvá tyto odkazované objekty. To pokračuje, dokud nejsou přesunuty všechny živé objekty. Mrtvé objekty nemají žádné odkazy, a proto zůstávají v původní polovině prostoru, která se v následujícím kroku přepíše novými objekty. [7]



Obrázek 7: Popis algoritmu Young Space Scavenger. Zdroj [7]

5.2.3 Parallel Marking and Concurrent Sweeping

Tato fáze by se dala volně přeložit jako paralelní značení a souběžné zametání. Když objekty dosáhnou určité životnosti, jsou povýšeny do nového paměťového prostoru, spravovaného kolektorem druhé generace: mark-sweep. Tato technika uvolňování paměti má dvě fáze: nejprve jsou procházeny objekty a systém si označí, které jsou stále používány. Během druhé fáze je naskenována celá paměť a všechny objekty, které nejsou označeny, jsou recyklovány. Všechny příznaky jsou poté vymazány. Tato forma garbage collectoru se může přinést zablokování hlavního UI vlákna, když při označovací fázi nedojde ke změně paměti. Tento typ je méně používaný, protože objekty s krátkou životností jsou spravovány Young Space Scavenger algoritmem. Ovšem stále existují případy, kdy se běh musí pozastavit, aby bylo možné spustit tuto formu uvolňování paměti. Vzhledem k tomu, že Flutter má schopnost naplánovat spuštění garbage collectoru, měl by být dopad na běh systému minimální. Dále třeba poznamenat, že pokud aplikace nedodržuje slabou generační hypotézu (která uvádí, že většina objektů má krátký životní cyklus), pak se tato forma bude vyskytovat častěji. [7]

5.2.4 Izolátory

Flutter podporuje funkci izolátorů - vláken, která nepřetržitě zpracovávají události ve vlastním paměťovém prostoru, takže nemohou přistupovat k objektům z cizího izolátu. Kód běží ve vlastní smyčce událostí a každá událost může spouštět menší úlohy ve vnořené frontě mikroúkolů. Dart umožňuje spuštění více vláken tím, že vytvoří další izoláty, kde každý má vlastní paměťový prostor. je možné komunikovat mezi jednotlivými izoláty prostřednictvím portů, které navíc mohou umožnit jiným izolátům řídit vlastní smyčku událostí a kontrolovat izolát, například pozastavením. [37]

5.3 Widget

Aplikace Flutter se skládá z dílčích částí, které se nazývají widgety. Jedná se o nejdůležitější prvky aplikace, ze kterých je tvořeno uživatelské prostředí. Kvůli tomu musí být atraktivní a intuitivní, aby používání aplikace bylo pro uživatele příjemné. Widgety nejenže řídí a ovlivňují zobrazení, ale také zpracovávají a reagují na akce uživatele. Proto je důležité, aby fungovaly rychle, včetně vykreslování a animací. Namísto opětovného využití widgetů z dané platformy, stejně jako to, co dělá React

Native, se tým Flutter rozhodl poskytnout své vlastní widgety To znamená, že Flutter má plnou kontrolu nad tím, kdy a jak budou widgety vykresleny. Svým způsobem tento přístup přesouvá widgety a vykreslovač z úrovně systému do samotné aplikace, což umožňuje více přizpůsobitelný vývoj. Na druhou stranu widgety a vykreslovač v aplikaci zvětšují velikost výsledné aplikace. [26] [38]

5.3.1 Stav widgetu

Widgety se rozdělují na dva typy: stateful (stavové) a stateless (bezstavové).

5.3.2 Bezstavový widget

Jedná se o Widget, kterému se nemění jeho vnitřní stav v čase. Jeho struktura je velmi jednoduchá. To znamená, že se jeho stav od doby vytvoření až po jeho zánik nezmění. Většinou se jedná o widgety, které uvnitř sebe obsahují další stavové widgety [39]. Bezstavový widget je užitečný, pokud část uživatelského rozhraní, kterou popisuje, nezávisí na ničem jiném než na konfiguračních informacích v samotném objektu a BuildContextu. Typicky se může tedy jednat o prostý text, který se časem nemění a jako vnitřní konfiguraci má nastavený obsah textového pole a například nějaké styly zobrazení. Jeho základní podoba je ukázána ve Zdrojovém kódu 1. [40]

```
class StatelessExample extends StatelessWidget {
  const StatelessExample({ Key? key }) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Container(color: const Color(0xFF2DBD3A));
  }
}
```

Zdrojový kód 1: Struktura bezstavového widgetu

5.3.3 Stavový widget

Druhým typem je stavový widget (Stateful). Jedná se o speciální widget, který má v čase se měnící vnitřní stav (objekt State). Jeho základní struktura je popsána ve Zdrojovém kódu 2. Typicky se jedná o dynamická pole, která se mění podle vstupů

od uživatele, nebo například textová pole, která načítají svou hodnotu z API. Instance stavového widgetu jsou samy o sobě neměnné a ukládají svůj proměnlivý stav buď do samostatných objektů State, které jsou vytvořeny metodou createState, nebo do objektů, ke kterým se tento stav přihlásí, například objekty Stream nebo ChangeNotifier, na které jsou odkazy uloženy v polích na samotném widgetu. Framework volá metodu createState vždy, když vytvoří nový, což znamená, že více objektů State může být přidruženo ke stejnému stavovému widgetu, pokud byl tento widget vložen do stromu na více místech. Podobně, pokud je stavový widget odebrán ze stromu a později znovu vložen, framework znovu zavolá createState a vytvoří pro něj nový objekt State, což zjednoduší životní cyklus. Je zde i možnost zachování stejného objektu State při přesunu umístění a to je použití globálního klíče (GlobalKey). Jedná se o unikátní identifikaci objektů napříč aplikací, která umožňuje přístup k jejich stavům i z cizích widgetů [41]. [42]

```
class StatefulExample extends StatefulWidget {
  const StatefulExample({ Key? key }) : super(key: key);

  @override
  State<StatefulExample> createState() => _StatefulExampleState();
}

class _StatefulExampleState extends State<StatefulExample> {
  @override
  Widget build(BuildContext context) {
    return Container(color: const Color(0xFFFFE306));
  }
}
```

Zdrojový kód 2: Struktura stavového widgetu

5.3.4 Skládání widgetů

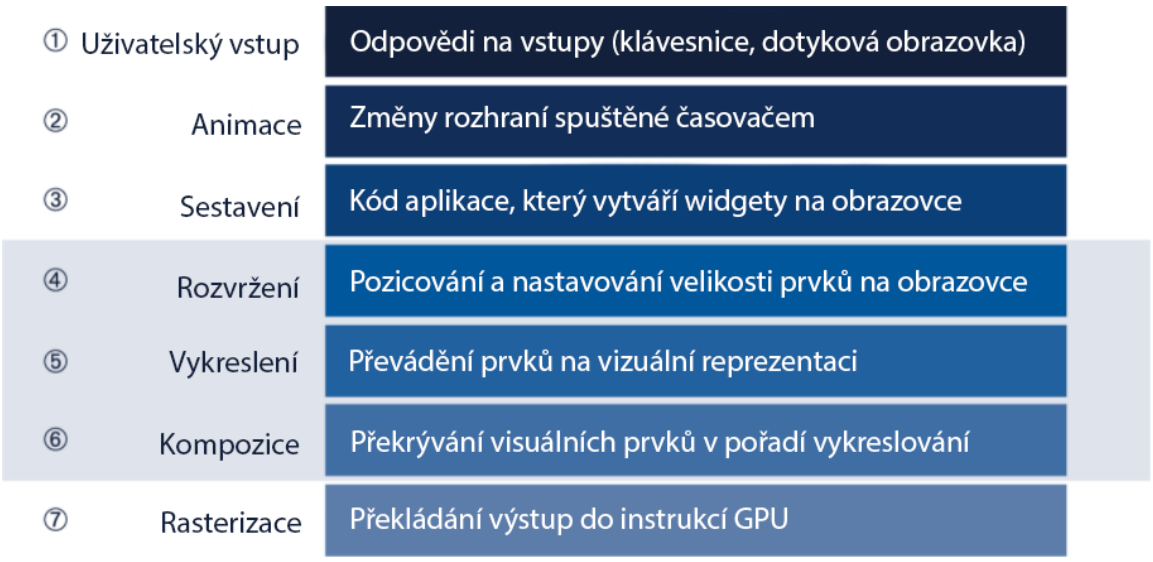
Myšlenkou Flutteru je, aby se systém skládal z mnoha dalších malých jednoúčelových podčástí, které se kombinují a společně vytvářejí výkonné celky. Tam, kde je to možné, se omezuje počet designových konceptů na minimum, aby se zachovala vysoká zno-

vupoužitelnost. Tento přístup Flutter aplikuje i ve vrstvě widgetů, kde používá stejný základní koncept (widget) k vykreslování obsahu na obrazovce, rozložení elementů, interaktivitě s uživatelem, správě stavu, motivů, animací a navigace. Ve vrstvě animace pokrývá zase většinu návrhového prostoru dvojice widgetů Animation a Tween. Ve vrstvě vykreslování se používá RenderObject k popisu rozvržení a malování. Existují stovky widgetů a objektů vykreslení a desítky typů animací. Jejich hierarchie je záměrně mělká a široká, aby se maximalizoval možný počet kombinací, se zaměřením na malé, skládatelné widgety, z nichž každý dělá jednu věc. Základní funkce jsou abstraktní, dokonce i základní funkce, jako je odsazení a zarovnání, jsou implementovány jako samostatné komponenty. Zde je vidět rozdíl oproti tradičnějšími rozhraními, kde jsou tyto funkce integrovány do společného jádra každé zobrazující se komponenty. Ve Flutteru je nutné pro nastavení určité vlastnosti rozložení, zabalit daný objekt do widgetu, který danou vlastnost nastaví.Tyto widgety rozložení nemají vlastní vizuální znázornění. Místo toho je jejich jediným účelem ovládat některé aspekty rozvržení jiného widgetu. [6]

5.3.5 Vykreslování widgetů

Každý widget, který má být vykreslen musí implementovat funkci build (sestavení), která definuje, co se má následně vykreslit do stromu prvků. Tento strom představuje úplný seznam prvků uživatelského rozhraní. Například panel nástrojů může mít funkci build, která vrací vodorovné rozložení některých textů a různých tlačítek. Podle potřeby Flutter rekurzivně požádá každý widget o sestavení, dokud nebude strom zcela popsán konkrétními vykreslitelnými objekty. Následně dojde ke spojení vykreslených objektů do stromu objektů. Funkce sestavení widgetu by neměla obsahovat logiku, která má vedlejší efekty. To znamená, že sem nepatří například volání API, protože kdykoli je funkce vyzvána k sestavení widgetu, měla by vrátit nový strom widgetů bez ohledu na to, co widget dříve vrátil. Nehledě na to, že tato funce se může volat i několikrát za sekundu a provádět zde nějakou komplexnější byznys logiku by bylo časově náročné. [6]

Flutter analyzuje strom prvků, aby určil, které funkce sestavení konkrétních widgetů je třeba volat. Pro každý vykreslovaný snímek Flutter umí znovu vytvořit pouze ty části uživatelského rozhraní, kde se změnil stav a díky tomu šetří výpočetní výkon. Proto je důležité, aby sestavení widgetů bylo co možná nejrychlejší a složitější logika byla prováděna paralelně a poté uložena jako součást stavu. [6]

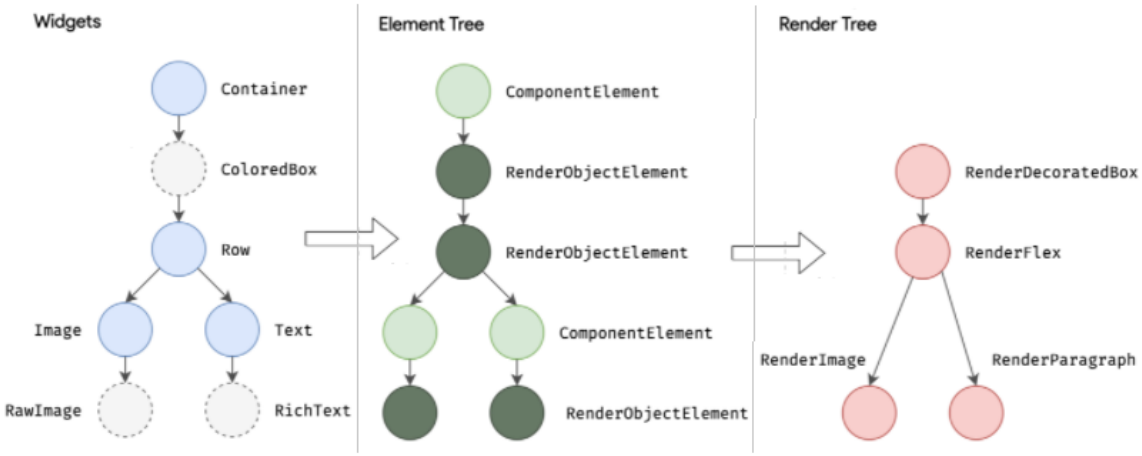


Obrázek 8: Postup při vykreslování prvků ve Flutteru. Zdroj [6]

Když Flutter potřebuje vykreslit widget, zavolá metodu build, která vrátí podstrom widgetů, který vykreslí uživatelské rozhraní na základě aktuálního stavu aplikace. Během tohoto procesu může metoda build podle potřeby zavést nové widgety na základě svého aktuálního stavu. Například při vykreslování prvku Container, který má přiřazené vnořené elementy, se z pohledu zdrojového kódu pro Container můžete manuálně kontrolovat, pokud barva není null, tak vloží ColoredBox představující výchozí barvu. Výsledný strom hierarchie widgetů proto může být hlubší a dynamicky se měnící.

Během fáze sestavení Flutter převádí widgety vyjádřené v kódu do odpovídajícího stromu prvků s jedním prvkem pro každý widget. Každý prvek představuje konkrétní instanci widgetu pro dané umístění stromové hierarchie. Ta je znázorněna na Obrázku 9. Existují dva základní typy prvků [6]:

- ComponentElement- obsahuje další prvky.
- RenderObjectElement- prvek, který se účastní fáze rozvržení nebo samotného vykreslení



Obrázek 9: Postup při převádění widgetů do stromu prvků. Zdroj [6]

RenderObjectElement je prostředníkem mezi jeho widgetem a podkladovým RenderObjectem. Element pro libovolný widget lze odkazovat prostřednictvím jeho BuildContextu, což je popisovač umístění widgetu ve stromu. Vzhledem k tomu, že widgety jsou neměnné, včetně vztahu nadřazený/podřízený mezi uzly, jakákoli změna stromu widgetů způsobí, že bude vrácena nová sada objektů. Strom prvků se ukládá do mezi paměti pro každý nový snímek obrazovky, a proto je důležité, aby aktualizace tohoto stromu byla rychlá. Flutter naštěstí umí detekovat, kde byla provedena změna a aktualizuje pouze danou část stromu.

Klíčem správného fungování každé Flutter aplikace je efektivní rozložení hierarchie widgetů a určení velikosti a umístění každého prvku před jeho vykreslením na obrazovce. Základní třída pro každý uzel ve stromu vykreslení je RenderObject. Ten definuje abstraktní model pro rozvržení a vykreslení na obrazovku. Je však velmi obecný, protože se nezavazuje se k pevnému počtu dimenzí nebo ke kartézskému souřadnicovému systému. Každý RenderObject zná svého rodiče, ale ví jen málo o svých dětech. To poskytuje RenderObject dostatečnou abstrakci, aby bylo možné zpracovat s různými typy použití. Během fáze sestavení Flutter vytvoří nebo aktualizuje objekt, který dědí z RenderObject pro každý RenderObjectElement ve stromu elementů. RenderObjects jsou primitiva: RenderParagraph vykresluje text, RenderImage vykresluje obrázek a RenderTransform aplikuje transformaci před malováním svého potomka. [6]

Většina widgetů Flutter je vykreslena objektem, který dědí z podtřídy RenderBox. Ten představuje RenderObject pevné velikosti a je umístěn ve 2D kartézském

prostoru. RenderBox poskytuje základní omezení elementu a vytváří minimální a maximální šířku a výšku pro každý widget, který má být vykreslen. Pro vytvoření rozvržení prvků, Flutter prochází stromem vykreslení a předává tato omezení velikosti z nadřazeného na podřízený prvek. Při určování své velikosti musí potomek respektovat omezení, která mu dal jeho rodič. Děti reagují předáním velikosti nadřazenému objektu v rámci omezení, která rodič stanovil. Na konci každého takového průchodu stromem má každý objekt definovanou velikost v rámci svých rodičovských omezení a je připraven k vykreslení voláním metody paint. [6]

5.3.6 Sestavování rozložení komponent

S velkým počtem widgetů a vykreslovacích objektů jsou klíčem k dobrému výkonu efektivní algoritmy, které je vykreslují. Hlavní vliv na výkon má samotné rozvržení komponent. O to se stará algoritmus, který určuje geometrii (například velikost a polohu) objektů. Flutter se zaměřuje na lineární výkon pro počáteční rozvržení a sublineární výkon při aktualizaci stávajícího rozvržení. Čas strávený na výpočet rozvržení by měl růst pomaleji pomaleji než počet vykreslených objektů. [43]

Flutter provádí jeden výpočet rozvržení na nový snímek obrazovky a algoritmus rozvržení funguje v jednoprůchodově. Vazby jsou předávány stromem z nadřazených objektů, které volají metodu rozložení na každém ze svých potomků. Ty rekurzivně provedou své vlastní rozvržení a pak vrátí geometrii nahoru stromem. Důležité je, že jakmile se objekt jednou vrátí ze své metody rozvržení, tak již ji nebude znovu pouštět až do rozvržení pro další snímek. V důsledku toho je každý objekt vykreslení navštíven maximálně dvakrát během rozvržení: jednou na cestě dolů stromem a jednou na cestě nahoru. [43]

Obecněji řečeno, během rozvržení jsou jedinými informacemi, které proudí z nadřazeného na podřízený objekt, omezení a opačně, tedy jediné informace, které proudí z podřízeného na nadřazený, je geometrie. Tyto faktory mohou snížit množství práce potřebné během vytváření rozvržení [43]:

- Pokud potomek neoznačí své vlastní rozvržení jako neplatné a rodič mu vrátí stejná omezení, jaká obdrželo během předchozího rozvržení, může se okamžitě vrátit z rozvržení a neprovádět nový průchod sestavováním nového.
- Kdykoli rodič zavolá metodu rozložení potomka, rodič označí, jestli používá informace o velikosti vrácené svého potomka. Pokud rodič nepoužije tuto infor-

mace o velikosti, nemusí přepočítat své rozložení pokaždé, když potomek změní svou velikost.

- Při použití těsných vazeb, kdy jsou jednoznačně definovaná omezení. Dochází k tomu například tehdy, kdy minimální a maximální výška (nebo šířka) jsou stejné. Jediná velikost, která tedy splňuje tato omezení, je právě jedna. Pokud rodič poskytuje takto přísná omezení, nemusí přepočítávat své rozložení vždy, když potomek změní své rozložení, i když rodič používá ve svém rozložení velikost potomka, protože podřízený objekt nemůže změnit velikost bez nových omezení od svého nadřazeného objektu.

- Objekt může deklarovat, že používá vazby poskytnuté nadřazeným objektem pouze k určení své geometrie. Taková deklarace informuje framework, že nadřazený objekt vykreslení nemusí přepočítávat své rozložení, když podřízený objekt přepočítá své rozložení, i když omezení nejsou těsná a dokonce i když rozložení nadřazeného objektu závisí na velikosti podřízeného objektu, protože podřízený objekt nemůže změnit velikost bez nových omezení od svého nadřazeného objektu.

6 Multiplatformní vývoj za použití Flutteru

Následující část práce popisuje jednotlivé principy a metodiky, jak postupovat při psaní multiplatformní aplikace za využití frameworku Flutter. Konkrétní příklady jsou přímo implementovány a popsány nad demonstrační aplikací.

6.1 Popis demonstrační aplikace

Demonstrační aplikace si klade za cíl být být spustitelná na mobilních telefonech, tabletech, chytrých hodinkách, počítačích s operačním systémem Windows a ve webovém prohlížeči. To vše za použití jednoho zdrojového kódu. Aplikace demonstruje systém, který slouží pro organizaci času, vytváření úkolů a jejich správu. Aplikace obsahuje možnost vytvoření účtu, pomocí kterého je možné se přihlásit na všech, výše uvedených, platformách. Uživatel si může na hlavní stránce vytvořit sloupce, do kterých vytváří své úkoly. Sloupce znázorňují jednotlivé skupiny úkolů, mezi kterými je možné své úkoly libovolně přesouvat. Dále se zde nachází dashboard pro reprezentaci statistických údajů.

6.1.1 Server

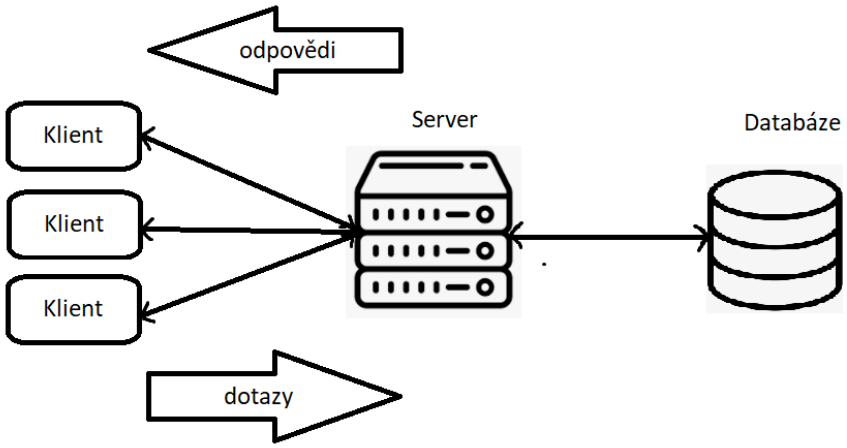
Na straně serveru, který se stará o požadavky klientů a vykonává byznys logiku, stojí Spring Boot aplikace, která vystavuje svým klientům REST API, přes které spolu komunikují. Server se současně stará o správu dat v databázi.

6.1.2 Komunikace

Pro implementaci aplikace byl použit model klient-server. Tento přístup získal svou popularitu při nástupu webových stránek, nicméně v dnešní době nalézá své uplatnění i při mobilním vývoji. Systém klient-server stále více minimalizuje dobu vývoje aplikací rozdělením funkcí mezi server a jeho klienty (koncová zařízení). Uživatel komunikuje s klientem skrze svoje uživatelské rozhraní, a server se stará o byznys logiku aplikace jako jsou různé výpočty, komunikace s databází nebo získávání informací z třetí strany. [8]

V dnešní době se tento model stal tak populárním, protože se používá prakticky každý den pro nejrůznější aplikace. Mezi standardizované protokoly, které klienti a servery používají ke komunikaci se sebou samými, patří: FTP (File Transfer Protocol), SMTP (Simple Mail Transfer Protocol) a HTTP (Hypertext Transfer Proto-

col). Model klient-server lze tedy definovat jako softwarovou architekturu tvořenou klientem i serverem, kdy klienti vždy odesílají požadavky, zatímco server odpovídá na odeslané požadavky, jak je popsáno na Obrázku 10. Klient-server poskytuje meziprocesovou komunikaci, protože zahrnuje výměnu dat od klienta i serveru, přičemž každý z nich vykonává různé funkce. [8]



Obrázek 10: Model klient-server. Zdroj [8]

Při použití tohoto modelu je potřeba se vypořádat s těmito oblastmi [8]:

- **Odborné znalosti:** Mnoho sítí klient-server není dobře postaveno a spravováno. Nastavení sítě klientského serveru je složité a vyžaduje kvalifikované techniky, kteří ji zvládnou.
- **Servery jsou poměrně drahé:** Servery jsou navrženy tak, aby splňovaly vysoký standard, aby byly spolehlivé a měly lepší výkon.
- **Bezpečnost:** Ze všeho nejvíce kritická je bezpečnost. Výměna zpráv mezi klientem a serverem vede k mnoha bezpečnostním výzvám. Je potřeba zajistit, aby přístupové body serveru byly zabezpečené a útočník přes ně nemohl například vytvářet nové záznamy v databázi.

Protože je aplikace rozdělena na dvě části (klient a server), je potřeba mezi nimi zajistit komunikaci, díky které bude možné tyto dvě části propojit. Pro tento účel bylo zvoleno REST API. To popisuje sadu přístupových bodů a sadu operací, které lze přes tyto body volat. Volání lze provést z libovolného klienta, který umí poslat požadavek pomocí HTTP, včetně kódu JavaScript na straně klienta, který běží ve webovém

prohlížeči [44]. REST API má základní cestu, která je podobná kořenovému adresáři kontextu. Všechny prostředky v REST API jsou definovány relativně k jeho základní cestě. Základní cestu lze použít k zajištění izolace mezi různými rozhraními REST API a také izolace mezi různými verzemi stejného rozhraní REST API. Cesty k jednotlivým přístupovým bodům mohou být hierarchické a dobře navržená struktura cesty může uživateli pomoci porozumět, jaká operace je pomocí něho vykonávána. [44]

Data skrze REST API mohou proudit několika způsoby [44]:

- **Path parameter:** Ty lze použít k identifikaci konkrétního zdroje. Hodnota parametru je předána do operace klientem HTTP jako proměnná část adresy URL a hodnota parametru je extrahována z cesty pro použití v operaci. Výsledná cesta může tedy vypadat například takto */api/users/{userId}*.
- **Query parameter:** Hodnota parametru dotazu je předána do operace klientem HTTP jako pár (klíč a hodnota) v řetězci dotazu na konci adresy URL. Například parametry dotazu lze použít k předání omezujících faktorů při výběru prvků z databáze: */items?priceMin=5&priceMax=20*
- **Header parameter:** Klient HTTP může předat parametry hlaviček operaci tak, že je přidá jako header HTTP v požadavku HTTP. Parametry hlavičky mohou být například použity k předání jedinečného identifikátoru nebo určení typu odesílaných dat v body.
- **Body:** Tento způsob požadavku se používá k odesílání a přijímání dat pokud se využívá metoda POST nebo PUT. Tímto způsobem se posílají větší objemy dat. Typicky se jedná o datové objekty v požadovaném formátu (např. JSON) nebo například celé soubory.

6.1.3 Zabezpečení komunikace

Pro zvýšení bezpečnosti komunikace mezi klientem a serverem byl použit JSON Web Token (dále JWT). Jedná se o standard RFC 7519, který definuje kompaktní a samostatný způsob bezpečného přenosu informací mezi stranami prostřednictvím objektu JSON. Platnost těchto informací lze ověřit, protože jsou digitálně podepsány , a proto jim je možné důvěřovat. JWT lze podepsat pomocí tajného klíče (algoritmem HMAC) nebo párem veřejného a soukromého klíče pomocí RSA nebo ECDSA. Pokud jsou to-

keny podepsány pomocí párů veřejného a soukromého klíče, podpis také potvrzuje, že pouze strana, která drží soukromý klíč, je ta, která jej podepsala. [45]

JWT je nejvíce využívaný na autorizaci a stejně tomu bude i v této aplikaci. Jakmile se uživatel přihlásí, dostane svůj jedinečný JWT, díky kterému bude jasné, který uživatel volá REST API na serveru. To umožní přístup k adresám, službám a prostředkům, které jsou povoleny s tímto tokenem. Jednotné přihlašování je funkce, která v dnešní době široce používá JWT, protože má malou režii a schopnost snadného použití v různých doménách. JWT se dále může používat na výměnu běžných informací mezi stranami. Vzhledem k tomu, že JWT lze podepsat, například pomocí párů veřejného a soukromého klíče, systém si může být jist, že odesílatelé jsou tím, za koho se vydávají. Navíc při výpočtu podpisu pomocí záhlaví a datové části je možné ověřit, zda s obsah nebyl upraven. [45]

JWT token se skládá ze tří částí: hlavičky, těla a podpisu. Tyto části jsou za sebou v tomto pořadí oddělené tečkami. Hlavička se obvykle skládá ze dvou částí: typu tokenu, což je JWT, a použitého algoritmu podepisování, jako je HMAC SHA256 nebo RSA. Druhou částí tokenu je jeho tělo, které obsahuje nároky (claims). To jsou informace o entitě (obvykle uživateli) a další data. Existují tři typy nároků:

- **Registrované nároky:** Jedná se o soubor předdefinovaných nároků, které nejsou povinné, ale doporučené, aby poskytly soubor užitečných nároků. Některé z nich jsou: **iss** (emitent), **exp** (doba vypršení platnosti), **sub** (předmět), **aud** (publikum) a další.

- **Veřejné nároky:** Ty mohou být definovány podle libosti vývojářů, kteří používají JWT. Aby se však předešlo kolizím, měly by být definovány v registru webových tokenů IANA JSON nebo definovány jako identifikátor URI.

- **Soukromé nároky:** Jedná se o vlastní deklarace vytvořené za účelem sdílení informací mezi stranami, které se dohodnou na jejich použití, a nejedná se o registrované ani veřejné nároky.

Třetí a zároveň poslední částí je podpis. Pro jeho vytvoření je nutné vzít hlavičku a tělo, které se zakódují pomocí Base64Url, a podpisový klíč. Tyto atributy se vloží do algoritmu, který je specifikován v hlavičce. Podpis se používá k ověření, že zpráva nebyla po cestě změněna, a v případě tokenů podepsaných soukromým klíčem je možné také ověřit, že odesílatel JWT je tím, za koho se vydává. Výstupem jsou tři řetězce Base64URL oddělené tečkami, které lze snadno předat v prostředích HTTP

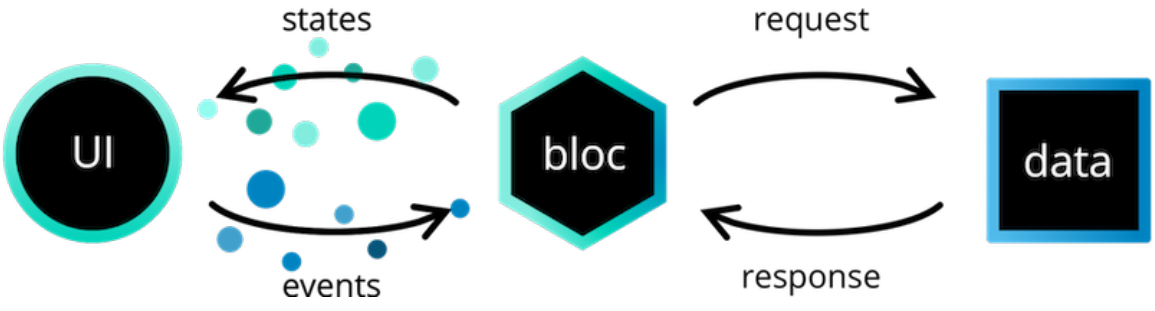
a zároveň jsou kompaktnější ve srovnání se standardy založenými na XML. Ukázka výsledného JWT je na Obrázku 11. [45]

```
eyJhbGciOiJIUzUxMiJ9.eyJzdWIiOjEwLCJpYX
QiOjE2NDg5MTc5MjUsImV4cCI6MTY0OTA5MDcyN
X0.X79HoBDJ30SV1oOwoxUb27sMStx27xVOzpJq
tiswloO63s_7CcnE80sNpOwpZEpYyHUjVMK2jBj
-2mIKo2vMDQ
```

Obrázek 11: Výsledný JWT. Zdroj autor

6.2 BLoC

BLoC je návrhový vzor vytvořený společností Google, který pomáhá oddělit byznys logiku od prezentační vrstvy, a proto umožňuje vývojářům efektivněji používat jednotlivé komponenty. Pomáhá spravovat stavy aplikace pomocí streamů, které jsou součástí aplikace a nevyžadují externí knihovnu. BLoC je zkratka pro Business Logic Component. Hlavní myšlenka je, že BLoC přijímá události, které způsobují změny stavu, které jsou pak emitovány z BLoC zpět do aplikace. BLoC má tedy oddělené vstupy a výstupy. Výstupy lze sledovat pomocí widgetů uživatelského rozhraní, které reagují na změny stavu. Na Obrázku12 je znázorněno, jak celý cyklus probíhá. Prezenční vrstva posílá eventy do BLoC, který následně mění data, a naopak prezenční vrstva získává data pomocí dotazů na aktuální stav (state). [46] [10]



Obrázek 12: Architektura návrhového vzoru BloC. zdroj [9]

V demonstrační aplikaci byla použita knihovna flutter_bloc, která pomáhá s integrací návrhového vzoru BloC do Flutter aplikace. Pro vytvoření funkčního modelu je nutné vytvořit třídu state, event a BloC.

Třída state representuje obecný stav. Můžou zde být definované atributy, které pak daný stav s sebou nese. Z této třídy následně dědí již konkrétní stavy, které mají své vlastní třídy spolu s jejich atributy. Příklad z demonstrační aplikace je znázorněn na Zdrojovém kódu 3. Zde je vidět, že stav BoardLoaded nese s sebou list načtených objektů, stav BoardErrorState obsahuje pouze chybovou hlášku a zbylé stavy jsou pouze informativní - bez dat.

```
abstract class BoardState {}

class BoardInitial extends BoardState {}
class BoardLoading extends BoardState {}
class BoardLoaded extends BoardState {
  final List<BoardDto> boards;
  BoardLoaded(this.boards);
}

class BoardErrorState extends BoardState {
  final String errorMessage;
  BoardErrorState(this.errorMessage);
}
```

Zdrojový kód 3: Definice stavových třídy návrhového vzoru BLoC

Třídy event jsou obdobně jako state definovány pomocí jednoho obecné třídy, ze které dědí již konkrétní akce. Tyto třídy by se daly popsat jako abstraktní předpisy funkcí. Mohou obsahovat data potřebná pro vykonání dané operace nebo mohou zůstat i prázdná. Níže ve Zdrojovém kódu 4 je uvedena ukázka třídy pro načtení všech tabulí s úkoly a třídy pro změnu pořadí tabulí.

```
abstract class BoardEvent {}

class BoardLoadEvent extends BoardEvent{}

class ReorderBoardsEvent extends BoardEvent{
  final int oldListIndex;
  final int newListIndex;
  ReorderBoardsEvent(this.oldListIndex, this.newListIndex);
}
```

Zdrojový kód 4: Definice tříd pro akce v návrhovém vzoru Bloc

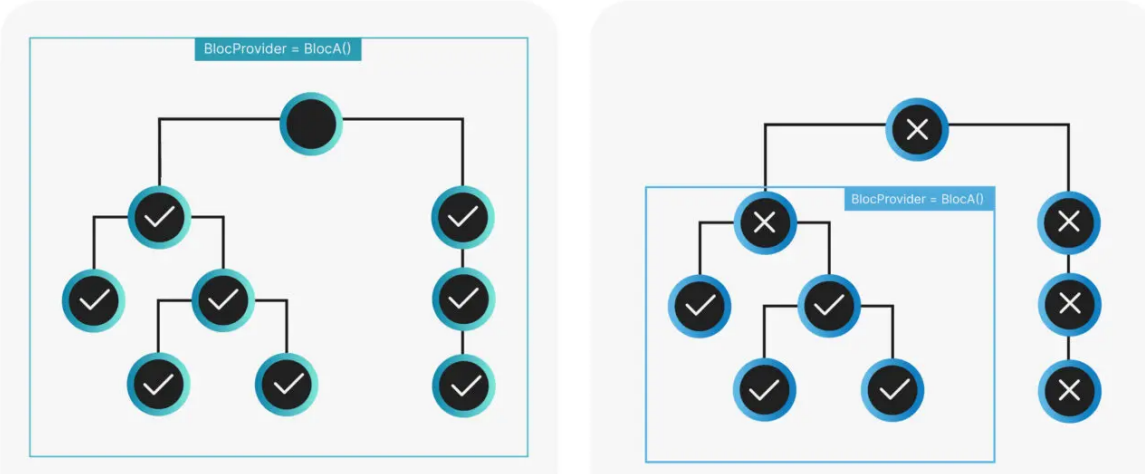
Třetí částí konfigurace je samotná třída BloC. Zde definováno, co se má vykonat pro každou akci a jak se v průběhu vykonávání akce mají měnit konkrétní stavy daného BloCu. Na ukázce zdrojového kódu 5 je možné pozorovat, že pokud BloC obdrží BoardLoadEvent, tak dojde k vykonání funkce loadBoards. Zde se nejdříve přepne stav BloC do BoardLoading, díky čemuž vizuální komponenty, které reagují na změnu stavu, mohou zobrazit načítací animaci. Dále dojde provolání API v TaskRepository a podle návratové hodnoty se nastaví stav na BoardLoaded, který obsahuje i načtené tabule, nebo dojde k nastavení chybného stavu BoardErrorState spolu s chybovou hláškou. Díky tomuto přepínání stavů je možné efektivně měnit konkrétní části uživatelského rozhraní. Tyto principy jsou popsány v následující části.

```
class BoardBloc extends Bloc<BoardEvent, BoardState> {
  final TaskRepository taskRepository;

  BoardBloc(this.taskRepository) : super(BoardInitial());

  @override
  Stream<BoardState> mapEventToState(
    BoardEvent event,
  ) async* {
    if (event is BoardLoadEvent) {
      yield* loadBoards();
    }
  }
  Stream<BoardState> loadBoards() async* {
    yield BoardLoading();
    try {
      List<BoardDto> boards = await taskRepository.loadBoards();
      yield BoardLoaded(boards);
    } catch (e) {
      String errorMessage = DEFAULT_ERROR_MESSAGE;
      if (e is OcValidationException) {
        errorMessage = e.getFirstMessageOrDefault;
      } else if (e is RuntimeException) {
        errorMessage = e.message;
      }
      yield BoardErrorState(errorMessage);
    }
  }
}
```

Zdrojový kód 5: Definice třídy BloC a realizace akcí

Klíčovým komponentem pro práci s návrhovým vzorem BloC ve Flutteru je BlocProvider. Jedná se o widget, který vytváří a poskytuje přístup k BloC všem svým dětem. Toto je známo jako widget pro vkládání závislostí, kde jedna instance Blocu

může být poskytnuta více widgetům v rámci podstromu. Jinými slovy, celý podstrom widgetů bude moci poslouchat změny stavů, které byly vyvolány jedinou událostí. Obrázek 13 znázorňuje výhodu, kterou má použití BlocProvideru již v hlavním widgetu aplikace, který je nejvýše v hierarchii widgetů. Díky tomu se nemůže stát, že by část aplikace nepracovala s aktuálními stavy. [10]



Obrázek 13: Výhoda použití BlocProvideru pro celou aplikaci. Zdroj [10]

Dalším widgetem, který pracuje s BloCem je BlocBuilder. Je to widget, který pomáhá znovu sestavit uživatelské rozhraní na základě změn stavu BloCu. Tato komponenta vyvolá změnu uživatelské rozhraní pokaždé, když dojde ke změně stavu. Sestavení velké části uživatelského rozhraní uvnitř aplikace může vyžadovat mnoho výpočetního výkonu. Proto je dobrým zvykem zabalit menší část uživatelského rozhraní, která má reagovat na změny stavu, uvnitř BlocBuilderu. Jedná se například o textové komponenty, které se aktualizují ze sekvence generovaných stavů. Bylo by zbytečné kvůli změně jednoho textového pole přepočítávat rozložení celé stránky. Místo toho je mnohem efektivnější použít BlocBuilder, který změní pouze malou část uživatelského rozhraní. V demonstrační aplikaci byla tato funkcionalita využita například u vypisování průběhu vytváření úkolu (viz Zdrojový kód 6). [10]

```
BlocBuilder<CreateTaskBloc, CreateTaskState>(builder: (context, state) {
    String message = "";
    bool error = false;
    if (state is CreateTaskSaving) {
        message = "Saving...";
    } else if (state is CreateTaskSaved) {
        message = "Saved";
    } else if (state is CreateTaskError) {
        message = state.errorMessage;
        error = true;
    }
    return FormInfoLabel(message, error);
})
```

Zdrojový kód 6: Definice třídy BloC a realizace akcí

Pokud je potřeba pouze reagovat na změny stavu, ale neměnit při tom nějaké prvky ze stromu widgetů, je možné použít BlocListener. Jak název napovídá, bude poslouchat jakoukoli změnu stavu, jak to dělá BlocBuilder. Ale místo vytváření widgetu, provede danou funkci. Ta se může lišit podle aktuálního stavu. Naslouchá pouze změnám stavu a provádí některé operace. Využití nalezne například při navigaci na jinou obrazovku, zobrazení dialogového okna nebo změny lokální proměnné. [47]

6.3 Princip vykreslování

Jak již bylo řečeno, Flutter podporuje vývoj pro platformy Androidu a iOS, ale také webové či desktopové aplikace. To z něj tvoří silný nástroj pro vývojáře, kteří díky němu mohou ušetřit mnoho času při psaní aplikací na více platforem současně. Jak bylo popsáno předešlé kapitole, spíše než aby Flutter překládal svoje komponenty do ekvivalentních widgetů jednotlivých operačních systémů, uživatelská rozhraní Flutter jsou postavena, rozložena, složena a vykreslena samotným frameworkem Flutter. Mechanismus pro získání textur a účast na životním cyklu aplikace se ale nevyhnutelně liší v závislosti na jedinečných přístupech každé platformy. Engine frameworku je nezávislý na platformě a představuje stabilní ABI (Application Binary Interface), které

poskytuje embedder každé platformy s konkrétním způsobem, jak nastavit a používat Flutter aplikace. Embedder platformy je nativní aplikace operačního systému, která hostuje veškerý obsah a funguje jako spojovací část mezi hostitelským operačním systémem a samotnou aplikací. Když dojde ke spuštění, embedder poskytne vstupní bod, inicializuje modul Flutter, získá vlákna pro uživatelské rozhraní a rastrování a vytvoří texturu, do které může Flutter vykreslovat svůj obsah. Embedder je také zodpovědný za životní cyklus aplikace, včetně vstupních gest jako je myš, klávesnice a dotykové ovládání. Dále má na starost velikosti oken a správy vláken. Flutter obsahuje embeddery pro Android, iOS, Windows, macOS a Linux. Podporuje také možnost si vytvořit vlastní embedder například pro propojení s Raspberry Pi. [11]

Každá platforma má svou vlastní sadu rozhraní API spolu s několika omezeními [11]:

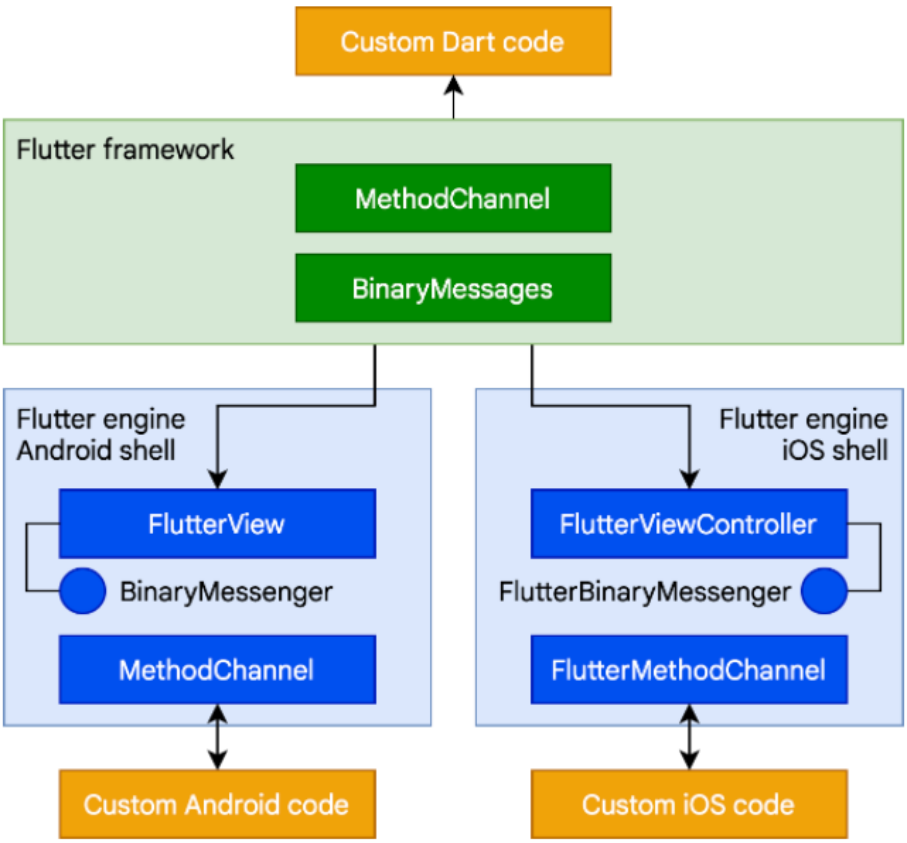
- Na zařízeních s operačním systémem iOS a macOS se Flutter načte do embedderu jako UIViewController nebo NSViewController. Embeder následně vytvoří FlutterEngine, který slouží jako hostitel virtuálního počítače Dart a běhového prostředí aplikace, se vykreslí pomocí OpenGL nebo Metal. FlutterViewController se připojí k FlutterEngine pro předávání vstupních událostí UIKit nebo Kakaa do Flutteru a pro zobrazení snímků vykreslených FlutterEngine pomocí Metal (grafický framework pro s podporou 3D vykreslování [48]) nebo OpenGL (nejrozšířenější 2D a 3D grafické API [49]).

- Flutter se pro Android zařízení ve výchozím nastavení načte do embedderu jako aktivita. Zobrazení je řízeno komponentou FlutterView, která vykresluje svůj obsah buď jako pohled nebo jako texturu.

- V systému Windows je Flutter hostován v tradiční aplikaci Win32 a obsah je vykreslován pomocí knihovny ANGLE , která překládá volání rozhraní Api OpenGL na ekvivalenty DirectX 11 [50]. V současné době probíhají snahy nabídnout také windows embedder pomocí modelu aplikace UWP a také nahradit ANGLE.

Flutter u mobilních a desktopových aplikací umožňuje volat vlastní kód prostřednictvím kanálu dané platformy, což je mechanismus pro komunikaci mezi kódem Dart a kódem hostitelské aplikace specifické pro platformu. Vytvořením společného kanálu je možné odesílat a přijímat zprávy mezi Dart a komponentou napsanou v jazyce, jako

je Kotlin nebo Swift. Data jsou serializována do mapy objektů a poté deserializována do ekvivalentní reprezentace.



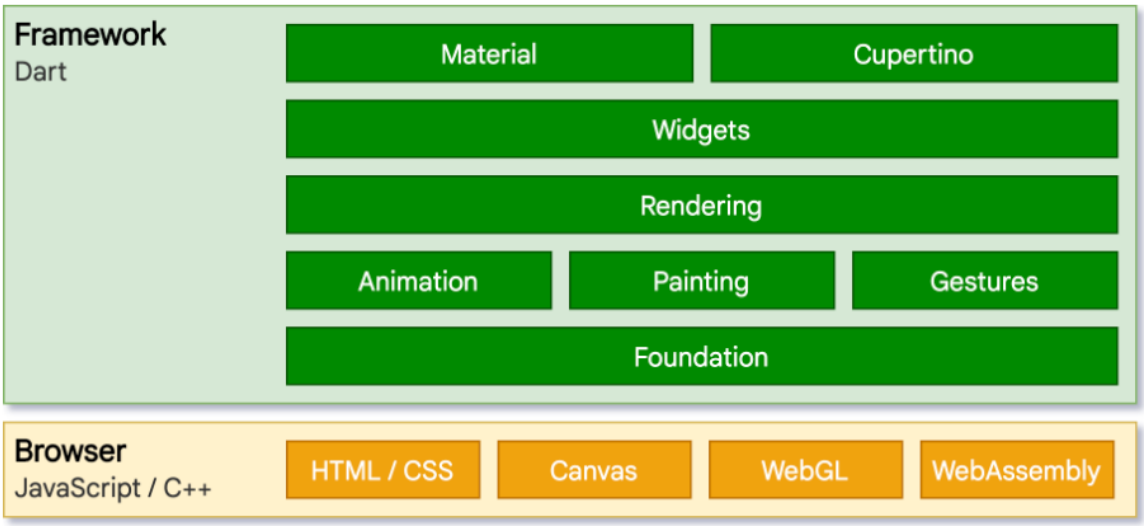
Obrázek 14: Kanály pro volání specifického kódu dané platformy. Zdroj [11]

6.4 Podpora webového rozhraní

Zatímco obecné architektonické koncepty platí pro všechny platformy, které Flutter podporuje, pro webovou podporu existuje pár jedinečných vlastností, které ji odlišují. Dart je možné zkompilovat do JavaScriptu od té doby, co existuje. Vzhledem k tomu, že Flutter framework je napsán v Dartu, který je Javascriptu velmi podobný, jeho kompilace do JavaScriptu je relativně jednoduchá. Engine Flutteru, který je napsaný v jazyce C++, je však navržen tak, aby komunikoval se základním operačním systémem spíše než s webovým prohlížečem. Proto je nutný jiný přístup. Pro webové aplikace Flutter poskytuje reimplementaci enginu nad standardními rozhraními API prohlížeče, jak je znázorněno na Obrázku 15, který popisuje achitekturu Flutteru pro podporu webu. V současné době existují sva přístupy při vykreslování

obsahu Flutter na webu: HTML (standardní značkovací jazyk pro vytváření webových stránek.) a WebGL (multiplatformní rozhraní API, které se používá k vytváření 3D grafiky pro webové prohlížeče [51]). V režimu HTML používá Flutter HTML, CSS, Canvas a SVG. Pro vykreslení do WebGL používá Flutter verzi Skia zkompilovanou do WebAssembly s názvem CanvasKit. Zatímco režim HTML nabízí psát aplikace s menší velikostí kódu, CanvasKit poskytuje nejrychlejší cestu ke grafickému zásobníku prohlížeče a nabízí poněkud vyšší grafickou věrnost a podobnost s nativní aplikací. [12]

Největší rozdíl ve srovnání s jinými platformami, na kterých Flutter běží, je, že není třeba, aby Flutter poskytoval běhové prostředí. Místo toho je rozhraní Flutter zkompilováno do jazyka JavaScript. Během vývoje používá Flutter dartdevc, kompilátor, který umožňuje spouštět a ladit webovou aplikaci Dart v prohlížeči Chrome a podporuje přírůstkovou kompilaci, a proto umožňuje horký restart [52]. Pro vytvoření produkční webové aplikace se používá dart2js. Jedná se o vysoce optimalizovaný produkční kompilátor Dartu do JavaScriptuu, který zabalí jádro a architekturu Flutter spolu s aplikací do zmenšeného zdrojového souboru, který lze nasadit na libovolný webový server. Kód může být nabízen v jednom souboru nebo rozdělen do více souborů prostřednictvím odložených importů. [12]



Obrázek 15: Architektura Flutter aplikace při podpoře webového rozhraní. Zdroj [12]

6.5 Specifické požadavky platforem

Kromě správného zobrazení uživatelského rozhraní na různě velkých obrazovkách je neméně důležité dobře zvážit, jaké jsou silné a slabé stránky každé cílové platformy. Není vždy ideální, aby multiplatformní aplikace nabízela na všech platformách stejnou funkcionalitu. Proto je potřeba se před samotným vývojem zaměřit na konkrétní funkcionality a pokud je to možné, odstranit určité funkce u některých kategorií zařízení. Například mobilní zařízení jsou přenosná a mají fotoaparáty a různé senzory, ale nejsou vhodná pro detailní kreativní práci. S ohledem na tuto skutečnost je možné se u nich více zaměřit na zachycení obsahu a jeho označení údaji o poloze nebo na ovládání uživatelského rozhraní pomocí již zmíněných senzorů jako jsou gyroskop a akcelerometr. Oproti tomu je vhodné umožnit detailnější práci s daty pro zařízení s větší obrazovkou jako jsou tablety a stolní počítače. Přemýšlet o tom, co jaká platforma dělá nejlépe, a zjistit, zda existují jedinečné schopnosti každé z nich, které je možné využít, je ve vývoji multiplatformní aplikace klíčové.

Dalším faktorem, který hraje velkou roli, je vytvoření komfortního dotykového uživatelského rozhraní, které může být často na vytvoření obtížnější než tradiční uživatelské rozhraní stolního počítače, částečně kvůli nedostatku vstupních akcelerátorů, jako je kliknutí pravým tlačítkem myši, rolovací kolečko nebo klávesové zkratky. Jedním ze způsobů, jak se k tomuto problému postavit, je zaměřit se primárně na uživatelské rozhraní orientované na dotykové ovládání. Důležité je zvážit, co uživatel očekává při používání konkrétního vstupního prvku, a pracovat na tom, aby se to implementovalo i v multiplatformní aplikaci.

6.5.1 Uživatelské vstupy

Pokud je mezi cílovými platformami platforma, která umí vypracovat s fyzickou klávesnicí, jako je většina desktopových a webových aplikací, je dobré se zamyslet nad zpracováním vstupů a mapování klávesových zkratek, které jsou všeobecně používané, protože to přináší uživatelskou přívětivost. Flutter přichází se zaměřeným systémem, který směruje vstupy z klávesnice do konkrétní části aplikace. Díky tomu uživatel může zaměří svůj vstup na určitou část aplikace klepnutím nebo kliknutím na požadovaný prvek uživatelského rozhraní. Jakmile k tomu dojde, text zadaný pomocí klávesnice "přeteče"do této části aplikace, dokud se zaměření nepřesune do jiné části aplikace. Zaměření na vstupní prvek lze také přesunout stisknutím určité klávesové

zkratky, která je obvykle vázána na klávesu Tab, které se někdy nazývá procházením pomocí tabulátoru. [53]

6.5.1.1 Kontrola zaměření na vstupní komponentu

Konkrétně ve Flutteru se pro kontrolu zaměření využívají objekty FocusNode a FocusScopeNode. Ty implementují mechaniku zaměření systému. Jedná se o objekty s dlouhou životností, které udržují stav fokusu a atributy tak, aby byly trvalé mezi jednotlivými sestaveními stromu widgetů (nevytváří se pro každé volání metody build). Společně tvoří datovou strukturu stromu zaměření. Původně byly zamýšleny jako objekty zaměřené na vývojáře, které se používají k ovládání některých aspektů zaměřovacího systému, ale v průběhu času se vyvinuly tak, aby implementovaly detaily zaměřovacího systému. Obecně platí, že se nejvíce využívají jako popisovač, předávaný podřízenému widgetu, který umí volat metodu requestFocus na widgetu rodiče. [53]

Pro odstranění zaměření na konkrétní komponentu se používá metoda FocusNode.unfocus(). Zde ale nedochází k úplnému odstranění zaměření, ale pouze k přesunutí na jiný prvek aplikace, protože vždy existuje jedno primární zaměření. Když prvek obdrží požadavek unfocus, přesune své zaměření buď na nejbližší FocusScopeNode, nebo na dříve zaměřený uzel v tomto oboru, v závislosti na volitelném argumentu v požadavku unfocus. Flutter nabízí také vlastní kontrolu na přesměrovávání zaměření, kde místo volání metody unfocus je možné explicitně určit na jaký jiný uzel se zaměření přesune. To je možné využít i na mobilních zařízeních. Tato funkcionalita je implementována v demonstrační aplikaci, kde slouží pro přesměrování zaměření při kliknutí na tlačítko, které přidává podúkoly. Po kliknutí na tlačítko dojde k vytvoření nového textového pole, kam se automaticky odkáže ukazatel a uživatel tak může rovnou vkládat text bez nutnosti klikání na ono nové textové pole (viz ukázka Zdrojového kódu 7). [53]

```
/*pridani noveho podukolu, ktery obsahuje svuj FocusNode,
na ktery je rovnou zavolan requestFocus()*/
 void addSubTask() {
   FocusNode newFocusNode = FocusNode();
   setState(() {
     taskDto.items.add(TaskItemDto(done: false,
                     focusNode: newFocusNode,key: UniqueKey()));
     newFocusNode.requestFocus();
   });
 }
/*vypis svech widgetu podukolu*/
 createSubTaskList() {
   List<TaskItemWidget> defaultList = [];
   if (taskDto != null || taskDto.items != null) {
     defaultList = [
       for (var item in taskDto.items) new TaskItemWidget(item,
       removeTaskItem, UniqueKey(),item.focusNode),
     ];
   }
   return defaultList;
 }
 /*ukazka casti widgetu, ktery obsahuje TextField spolu s FocusNode*/
 TextField(
       focusNode: widget.focusNode,
       controller: bodyController,
       /*rest of TextField*/
```

Zdrojový kód 7: Přesměrování zaměření na nové textové pole po kliknutí na tlačítko přidat podúkol

6.5.1.2 Události vyvolané klávesnicí

Pro poslouchání událostí vyvolaných klávesnicí na konkrétním widgetu je potřeba daný widget obalit widgetem Focus. Ten přijímá jako atribut onKey, pomocí kterého je možné jak poslouchat, tak i odchytávat libovolné události. Odchytávání událostí

z klávesnice začíná na uzlu zaměření (FocusNode), který má primární zaměření (je zrovna vybrán). Pokud tento uzel nevrátí KeyEventResult.handled ze své obslužné rutiny onKey, pak se tato událost předá jeho nadřazenému uzlu zaměření. Pokud takto žádný uzel nezachytí danou událost a ona dojde až do kořene stromu zaměřní, tak dojde k vrácení a řízení se předá dalšímu nativnímu ovládacímu prvku v aplikaci. Události, které jsou zpracovány, nejsou šířeny již do jiných widgetů a současně nejsou šířeny ani do nativních prvků. [54]

Aby aplikace mohla být ovládána, musí mít akce, díky kterým uživatele určí, co chtějí zrovna provést. Akce jsou často jednoduché funkce, které přímo provádějí například nastavení hodnoty do textového pole. Ve větších aplikacích jsou může být logika akcí složitější, a proto kód pro vyvolání akce a kód pro samotnou akci mohou být na různých místech. Klávesové zkratky mohou vyžadovat definici na úrovni, která neví nic o akcích, které samy vyvolávají. Flutter obsahuje vlastní systém pro obsluhu klávesových zkratek a jim přidělených akcí. Umožňuje vývojářům definovat akce, které splňují konkrétní záměry. Zde je záměrem myšlena obecná akce, kterou si uživatel přeje provést. Pro tento účel existuje třída Intent, která představuje záměry uživatele. Záměr může být obecný účel, splněný různými akcemi v různých kontextech. Akce může být jednoduché zpětné volání. [54]

Akce jako takové se dají ve Flutteru využít i samostatně, nicméně největší využití nalézají právě ve spojení s klávesovými zkratkami. Klávesové zkratky se aktivují stisknutím klávesy nebo kombinací kláves. Tyto kombinace jsou umístěny v tabulce spolu se záměry, které jsou k nim vázány. Když je widget Shortcuts vyvolá, odešle jejich odpovídající záměr do podsystému akcí k jeho uskutečnění. Widget Shortcuts se vkládá přímo do hierarchie ostatních zobrazovacích widgetů a definuje kombinace kláves, které při stisknutí vyvolají určitou akci. Pro převedení záměru kláves na konkrétní akci se používá widget Actions, který slouží k mapování záměrů na samotné akce. [54]

```
/*metoda, ktera vraci objekt formulare */
  Widget _buildForm() {
    return Shortcuts(
      shortcuts: {
        LogicalKeySet(LogicalKeyboardKey.delete): DeleteTaskIntent(),
      },
      child: Actions(
        actions: <Type, Action<Intent>>{
          DeleteTaskIntent: DeleteTaskAction(taskDto.id, context),
        },
        child: Builder(
            builder: (BuildContext context) => Focus(
                  autofocus: true,
                  child: _buildFormContent(),
                ),),);
  }
  /*akce, ktera po zavolani otevre dialogove okno pro potvrzeni smazani*/
 class DeleteTaskAction extends Action<DeleteTaskIntent> {
  DeleteTaskAction(this.taskId, this.context);
  final BuildContext context;
  final int taskId;

  @override
  void invoke(covariant DeleteTaskIntent intent) => {
        showDialog<String>(
          context: context,
          builder: showConfirmDialog(taskId),
        )};
  }
```

Zdrojový kód 8: Zachycení klávesové zkratky delete a následné vyvolání akce smazání.

V demonstrační aplikaci se tato funkcionalita využila při mazání úkolu na obrazovce jeho detailu viz Zdrojový kód 8. Zde buď klient musí kliknout na tlačítko

"Delete"nebo pokud aplikaci používá na počítači, tak stačí pouze zmáčknout klávesu Delete, která vykoná stejnou funkci.

Je dobré tyto klávesové zkratky implementovat do multiplatformních aplikací, protože velmi usnadňují používání a zároveň pomáhají navodit pocit, že se jedná o nativní aplikaci.

6.5.2 Specifické přístupy k platformám ovládaných myší

Některé platformy občas vyžadují specifický přístup, aby se docílilo pocitu, že se jedná o nativní aplikaci.

6.5.2.1 Změna kurzoru

Na počítači a webu je běžné, že dojde ke změně kurzoru myši podle funkce obsahu, na který myš najíždí. Typicky se jedná o změnu při najetí na tlačítko, kde se změní z klasického na výběrový. Obdobně tomu je i u textových polí, kde se mění na výběr textu. Tento aspekt při vývoji multiplatformní aplikace by neměl být opomíjen, protože přispívá k lepšímu ovládání a pocitu, že se jedná o nativní aplikaci stvořenou přímo pro počítač. V demonstrační aplikaci to bylo využito u tabulí s úkoly, kde se po najetí změní kurzor na možnost přesunout, která je znázorněna šipkami do všech směrů. Díky tomu by měl uživatel intuitivně vědět, že je možné s jednotlivými tabulemi pohybovat. Ve Flutteru je docílení tohoto efektu docíleno pomocí widgetu MouseRegion, který kromě změny kurzoru po najetí na jeho potomka, umí i odchytávat akce kurzoru jako jsou najetí, opuštění a pohyb po jeho potomkovi. Názorně je to ukázáno na Zdrojovém kódu 9.

```
MouseRegion(
    cursor: SystemMouseCursors.move,
    child: Container(
        /*rest of components*/
    )
)
```

Zdrojový kód 9: Změna kurzoru po najetí na widget

6.5.2.2 Popis funkčních prvků

Dalším specifickým prvkem zejména pro počítače jsou popisky funkčních prvků. Díky nim je uživatel informován o funkcionalitě dané komponenty. Typicky se jedná o text, který se snaží co nejstručnější a nejvýstižnější. Ve frameworku Flutter na to slouží widget Tooltip, díky kterému je možné zobrazit informační popisek při najetí na jakýkoli widget, kterého má jako potomka. Zdrojový kód 10 demonstruje zobrazení popisku po 300 milisekundách od najetí na box úkolu. Je zde mimo jiné možné nastavit i pozici, kde se daný text zobrazí.

```
Tooltip(
     message: "Show detail",
     waitDuration: Duration(milliseconds: 300),
     verticalOffset: 10,
     child: Container(
         /*rest of components*/
     )
 )
```

Zdrojový kód 10: Zobrazení informačního popisku

6.5.3 Hustota zobrazení

Různá vstupní zařízení nabízejí různé úrovně přesnosti, se kterými uživatel pracuje při ovládání. Typicky se jedná o velikosti tlačítek, které na mobilních zařízeních vyžadují větší velikost kvůli ovládání prsty, zatím aplikace na Windows mohou mít menší tlačítka díky ovládání myší, která je výrazně přesnější než prst. Ve Flutteru existuje třída VisualDensity, která usnadňuje úpravu hustoty zobrazení v celé aplikaci. Když dojde ke změně VisualDensity pro aplikaci, komponenty, které ji podporují, změní své velikosti tak, aby odpovídaly. Flutter nabízí nastavení vertikální a horizontální hodnoty, které mohou být kladné i záporné. Ve výchozím nastavení jsou obě hustoty nastaveny na hodnotu 0. Díky přepínáním mezi různými hustotami lze snadno upravit uživatelské rozhraní. Komponenty automaticky reagují na změny hustoty, a také animují změnu velikosti, když dojde ke změně. [55]

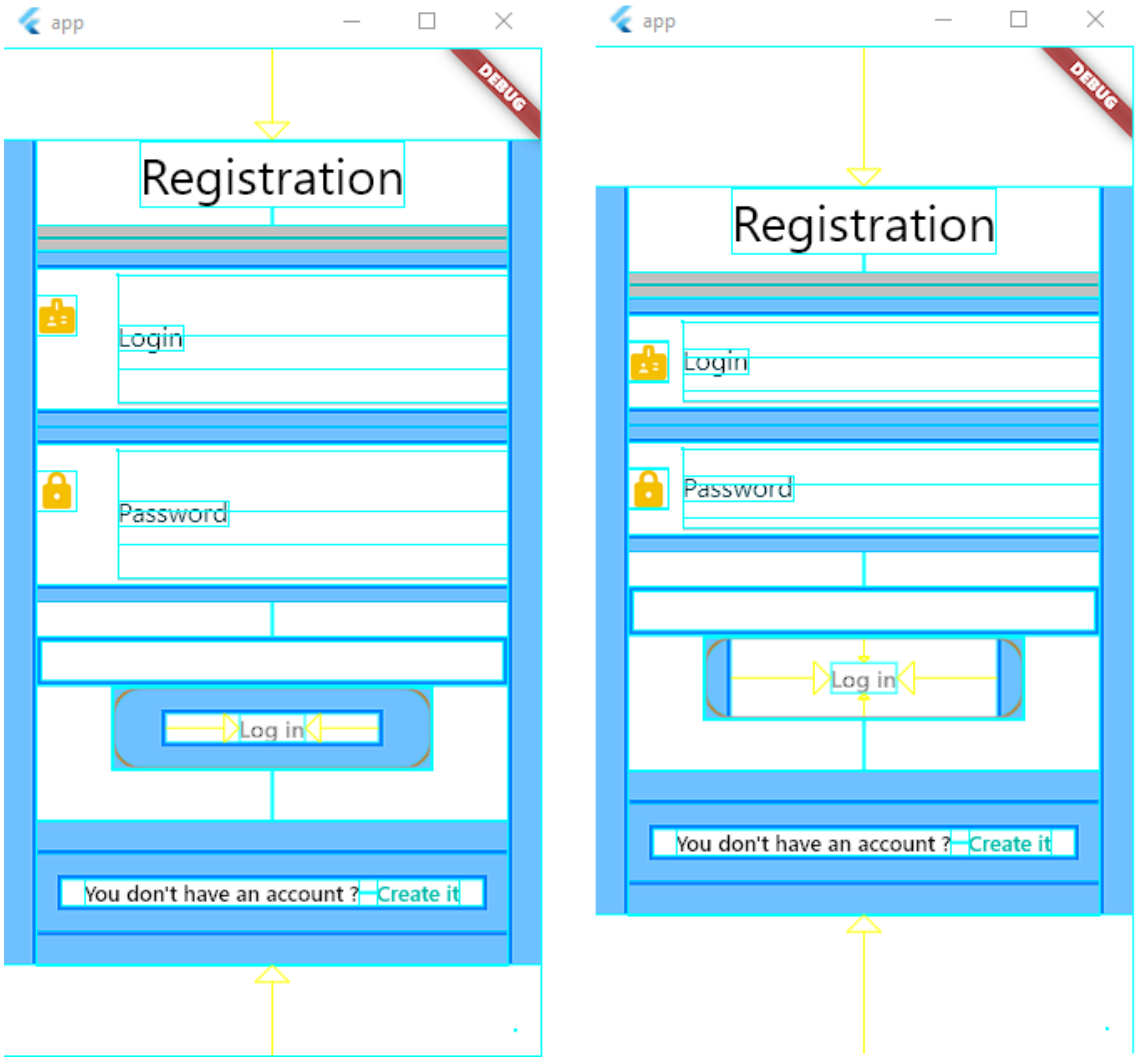
VisualDensity je nastavována bez konkrétních vizuálních jednotek, takže může pro různá zařízení představuje různou hodnotu. Komponenty obecně používají hodnotu

přibližně 4 logické pixely pro každou jednotku vizuální hustoty. Logické pixely jsou jednotkou délky zhruba podobné velikosti na různých zařízeních. Tyto pixely jsou nejprve převedeny na vykreslené obrazové body a poté zmenšeny na skutečné pixely zařízení, což pak mohou být skutečné obrazové body se třemi různými subpixelovými body [56].

```
static const VisualDensity compact = VisualDensity(horizontal: -2.0,
vertical: -2.0);
static const VisualDensity comfortable = VisualDensity(horizontal: 0,
vertical:0);

static VisualDensity get adaptivePlatformDensity {
   switch (defaultTargetPlatform) {
     case TargetPlatform.android:
     case TargetPlatform.iOS:
     case TargetPlatform.fuchsia:
       break;
     case TargetPlatform.linux:
     case TargetPlatform.macOS:
     case TargetPlatform.windows:
       return compact;
   }
   return VisualDensity.standard;
 }
```

Zdrojový kód 11: Adaptivní výběr VisualDensity podle aktuální platformy

Výše uvedený Zdrojový kód 11 ukazuje, jak funguje vnitřní logika Flutteru při použití adaptivního nastavení hustoty komponent. Pro Platformy Linux, MacOS a Windows nastaví kompaktní hustotu, která má vertikální i horizontální hodnotu nastavenou na -2. Pro zbylé platformy nastaví standardní hodnotu 0. Je možné ovšem vytvořit i vlastní objekt VisualDensity s libovolnými hodnotami, který se následně vloží do globálního vizuálního tématu aplikace.



Obrázek 16: Porovnání největší (vlevo) a nejmenší (vpravo) hustoty zobrazení. Zdroj autor

Obrázek16 znázorňuje rozdíl v nejvyšší hustotě zobrazení, která je vhodná pro zařízení ovládána prsty, a nejmenší hustotě zobrazení, která se naopak využívá na platformy, kde je hlavním ovládacím prvkem kurzor myši. Pro lepší vizualizaci byl zapnut vývojářský mód, který ohraničuje zobrazované komponenty. Je zřejmé, že prvky jako jsou textová pole a tlačítka změnila svou velikost. Ostatní části, jako je například nadpis stránky, zůstaly neměnné.

6.5.4 Rozložení podle kontextu

Pro rozsáhlejší změny oproti upravení hustoty rozložení nabízí Flutter specifické rozložení podle aktuálního kontextu. Jedná se o procedurálnější přístup k úpravě parametrů, výpočtu velikostí, záměně widgetů nebo úplné restrukturalizaci uživatelského rozhraní tak, aby vyhovovalo konkrétním požadavkům. Nejjednodušší princip využívá zarážek založených na obrazovce. Ve Flutteru to lze provést pomocí rozhraní MediaQuery API. Neexistují zde žádná striktní pravidla pro velikosti, které se zde mají použít, ale jedná se o obecné hodnoty. [13]

```
enum ScreenSize { Small, Normal, Large, ExtraLarge }

ScreenSize getSize(BuildContext context) {
  double deviceWidth = MediaQuery.of(context).size.shortestSide;
  if (deviceWidth > 900) return ScreenSize.ExtraLarge;
  if (deviceWidth > 600) return ScreenSize.Large;
  if (deviceWidth > 300) return ScreenSize.Normal;
  return ScreenSize.Small;
}
```

Zdrojový kód 12: Ukázka rozdělení velikosti aplikace za pomocí MediaQuery. Zdroj [13]

Kontrola celkové velikosti obrazovky je skvělá pro rozhodování o globálním rozvržení celých obrazovek. Často to ale není ideální pro vnořené komponenty, které mají často své vlastní vnitřní rozhodovací systém a starají se pouze o prostor, který mají k dispozici pro vykreslení přidělený od svého rodiče. Nejjednodušší způsob, jak to zvládnout ve Flutteru, je použití třídy LayoutBuilder. Ten umožňuje widgetům reagovat na příchozí omezení velikosti, což může učinit widgety univerzálnějším, než kdyby závisely na globální hodnotě. LayoutBuilder se v demonstrační aplikaci využívá pro nastavení globální proměnné, ke které následně jednoduše přistupují všechny widgety (viz Zdrojový kód 13). Díky volání builderu v LayoutBuilderu po každé změně velikosti (například při rotaci mobilního telefonu) je docíleno jednotné změny rozložení v celé aplikaci. [13]

```
MaterialApp(
    builder: (context, child) {
      return Scaffold(
        key: scaffoldKey,
        body: LayoutBuilder(
          builder: (BuildContext context, BoxConstraints constraints) {
            if (constraints.maxWidth <= 320) {
              globals.screenWidth = globals.ScreenWidth.SMALL;
            } else if (constraints.maxWidth <= 600) {
              globals.screenWidth = globals.ScreenWidth.MEDIUM;
            } else if (constraints.maxWidth <= 1200) {
              globals.screenWidth = globals.ScreenWidth.LARGE;
            } else {
              globals.screenWidth = globals.ScreenWidth.X_LARGE;
            }
            return child;
          },
        ),
        resizeToAvoidBottomInset: false,
      );
    },
    /*rest of configuration of MaterialApp*/
```

Zdrojový kód 13: Ukázka demonstrační aplikace, kde se ve hlavním widgetu pomocí LayoutBuilderu do globální proměnné nastavuje aktuální informace o velikosti.

Flutter také umí měnit rozvržení na základě aktuální platformy, na které aplikace běží, bez ohledu na velikost. V demonstrační aplikaci je tato metoda využita například při kontrole, jestli má dané zařízení přístup k internetu. Pro tento účel se využívá knihovna Connectivity plus [57], která nepodporuje web, a proto je zde potřeba provádět kontrolu (viz Zdrojový kód 14) a v případě webové platformy nevolat knihovnu Connectivity plus, která by jinak způsobila chybu.

```
static Future<bool> isInternet() async {
  var connectivityResult =
                await (Connectivity().checkConnectivity());

  if (kIsWeb) {
    return true;
  } else {
    if (connectivityResult == ConnectivityResult.mobile) {
      if (await DataConnectionChecker().hasConnection) {
        return true;
      } else {
        return false;
      }
      /*the rest of the method*/
```

Zdrojový kód 14: Kontrola aktuální platformy a omezení funkcionality pro web

6.6 Struktura aplikace

klíčovým faktorem při vývoji aplikace je správné (přehledné a systematické) rozdělení do hierarchie adresářů. Toto zásadně ovlivňuje udržovatelnost a škálovatelnost aplikace. Pro demonstrační aplikaci byla použita následující struktura:

- **/android:** Složka android obsahuje soubory a složky potřebné pro spuštění aplikace v operačním systému Android. Tyto soubory a složky jsou automaticky generovány během vytváření projektu Flutter. Doporučuje se, aby tyto složky a soubory zůstaly tak, jak jsou [58]. Primární podsložky složky android jsou složka res a soubor AndroidManifest.xml. Složka res obsahuje neprogramovatelné prostředky potřebné pro aplikaci, jako jsou ikony a obrázky v různých rozlišeních nebo písma, zatímco soubor AndroidManifest.xml obsahuje informace potřebné pro sadu SDK aplikace. Pro Flutter aplikace se zde může například konfigurovat nastavení pro notifikace.
- **/assets:** V této složce se nachází veškerá grafika, která se v aplikaci používá. Jedná se o obrázky nebo například vlastní ikony ikon ve formátu svg. Je dobré dále rozčlenit tyto soubory podle účelu nebo místa použití.

- **/ios:** Stejně jako složka android obsahuje tato složka soubory vyžadované ke spuštění Flutter aplikace na platformách iOS. Hlavní soubory ve složce ios jsou složka Assets.xcassests a soubor info.plist. Složka Assets.xcassests je jako složka res v systému Android. Obsahuje neprogramovatelné prostředky potřebné pro aplikaci. Soubor info.plist je podobný souboru "AndroidManifest.xml". Ukládá informace vyžadované sadou SDK aplikace. Pokud je třeba provést nějakou úpravu, aby byla aplikace specifická pro platformy IOS, měla by být provedena v této složce. [58]
- **/lib:** Toto je nejdůležitější složka v projektu, která se používá k napsání většiny z Flutter aplikace. Ve výchozím nastavení obsahuje složka lib soubor main.dart, který je vstupním bodem aplikace. Tuto konfiguraci však lze změnit. Dále tato složka obsahuje následující položky:
  - **/config:** V této složce se nachází několik typů konfiguračních částí aplikace. Konkrétně se zde nachází konfigurace a logika pro SecureStorage, který se stará o ukládání dat do lokálního úložiště zařízení, správa výjimek, globální proměnné nebo obsluha HTTP klienta, díky kterému je možné posílat a získávat data přes API.
  - **/data:** Do této složky je zahrnuto vše, co se týká správy dat. To znamená definice objektů (tříd), funkce pro volání jednotlivých API a konfigurace BloC, který se stará o stavy jednotlivých částí dat.
  - **/generated:** Automaticky generovaná složka ze souborů ve složce asset. Díky tomu je každý obrázek (jeho adresa) adresovatelný pomocí globální proměnné.
  - **/presentation:** Zde se nachází vše, co vidí uživatel. To jsou jednotlivé obrazovky a widgety na nich zobrazené. Nachází se zde složka screens, ve které jsou v jednotlivých podsložkách rozděleny obrazovky aplikace. Pro každou obrazovku existuje widget screen, který obsahuje adresu stránky, díky které je možné na ní přesměrovat, a základní nastavení jako je navigační panel, a widget body, na který se umísťují už samotné vizuální komponenty. Dále podle rozsahu každé obrazovky se zde můžou nacházet definice dalších widgetů, které se na dané stránce zobrazují.

Kromě definice obrazovek se zde také nachází složka s obecnými widgety, které naleznou využití napříč aplikací. Konkrétně to jsou různá tlačítka nebo okna s chybovou hláškou.

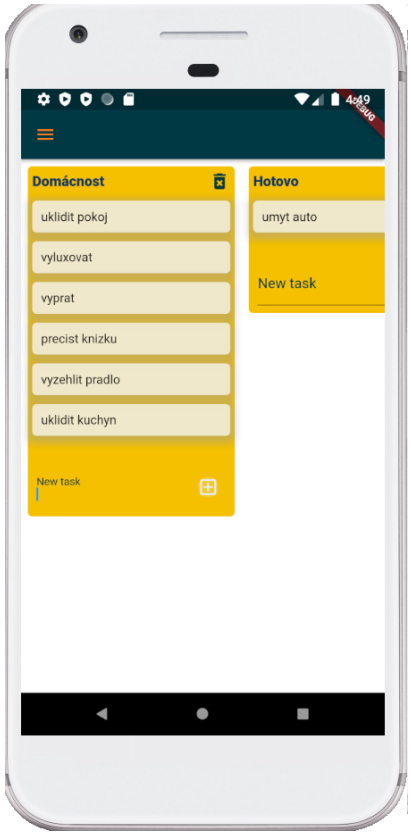
- **/web:** Složka, která obsahuje výchozí soubor pro spuštění webové aplikace index.html, ve kterém je definováno, že má jako tělo aplikace použít obsah ze souboru maint.dart.js, který vznikne při sestavování pomocí dart2js. Dále v této složce jsou specifické soubory pro webovou platformu.

- **pubspec.yaml:** Každý projekt Flutter obsahuje soubor pubspec.yaml, který je generován při vytváření nového projektu. Je umístěn v kořenovém adresáři a obsahuje metadata o projektu, které nástroje Dart a Flutter potřebují. Pubspec je napsán v YAML, který je lehce čitelný pro člověka. Tento soubor dále určuje závislosti, které projekt vyžaduje, jako jsou konkrétní balíčky s jejich verzemi, písma nebo soubory obrázků. Určuje také další požadavky, jako jsou závislosti na vývojářských balíčcích nebo konkrétní omezení verze sady Flutter SDK. [59]

7 Shrnutí výsledků

Při použití frameworku Flutter byla úspěšně vytvořena demonstrační aplikace, kterou je možné spustit na zařízeních s operačním systémem Android jako jsou mobilní telefony, tablety, chytré hodinky ale i třeba chytré televize. Dále aplikace může běžet na stolních počítačích s operačním systémem Windows a ve webovém prohlížeči. Na všech těchto platformách je samozřejmě možné spustit tuto aplikaci až po sestavení aplikace pro konkrétní cílovou platformu.

Pro zařízení s operačním systémem Android vznikne po sestavení buď spustitelná aplikace s koncovkou apk, který je sestaven zvlášť pro každé cílové binární rozhraní aplikace (ABI), nebo balíček s koncovkou aab, který ve výchozím nastavení obsahuje sestavený modul Flutter pro armeabi-v7a (ARM 32bitový), arm64-v8a (ARM 64bitový) a x86-64 (x86 64bitový) [60]. Výsledná aplikace je demonstrována na Obrázku 17, kde je spuštěna na mobilním telefonu, a na Obrázku 18, na kterém běží aplikace na chytrých hodinkách v kompaktnějším uživatelském rozhraní.



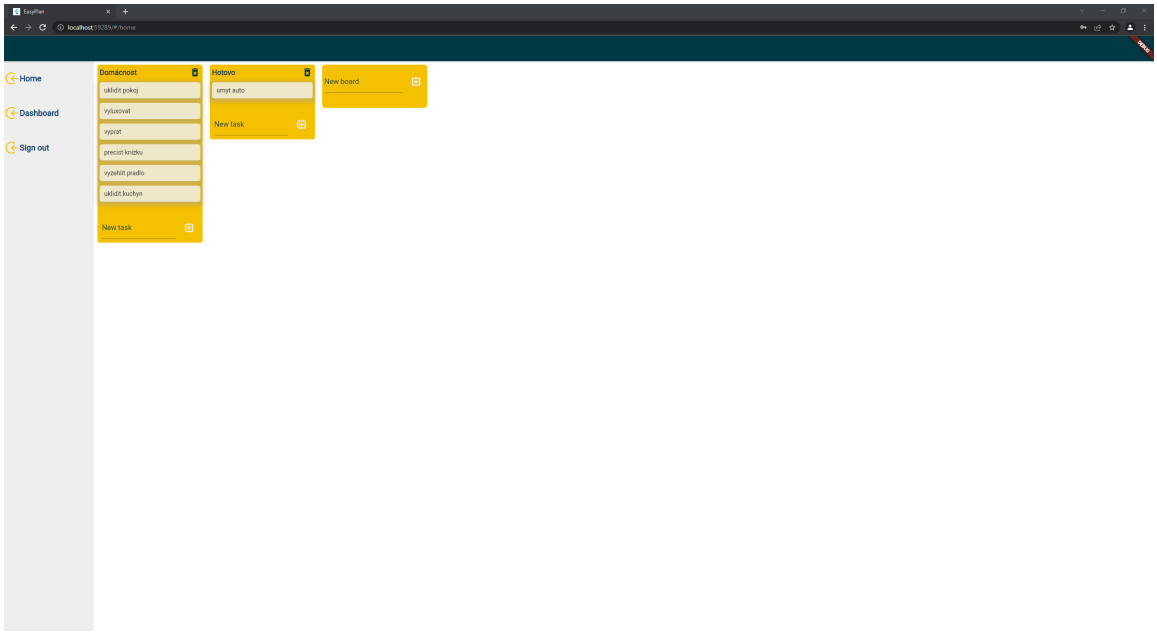
Obrázek 17: Ukázka demonstrační aplikace na mobilním telefonu. Zdroj autor

Obrázek 18: Ukázka demonstrační aplikace na chytrých hodinkách. Zdroj autor

Další platformou, na které může aplikace běžet je web. Po sestavení aplikace pro tuto platformu vznikne adresář s následující strukturou:

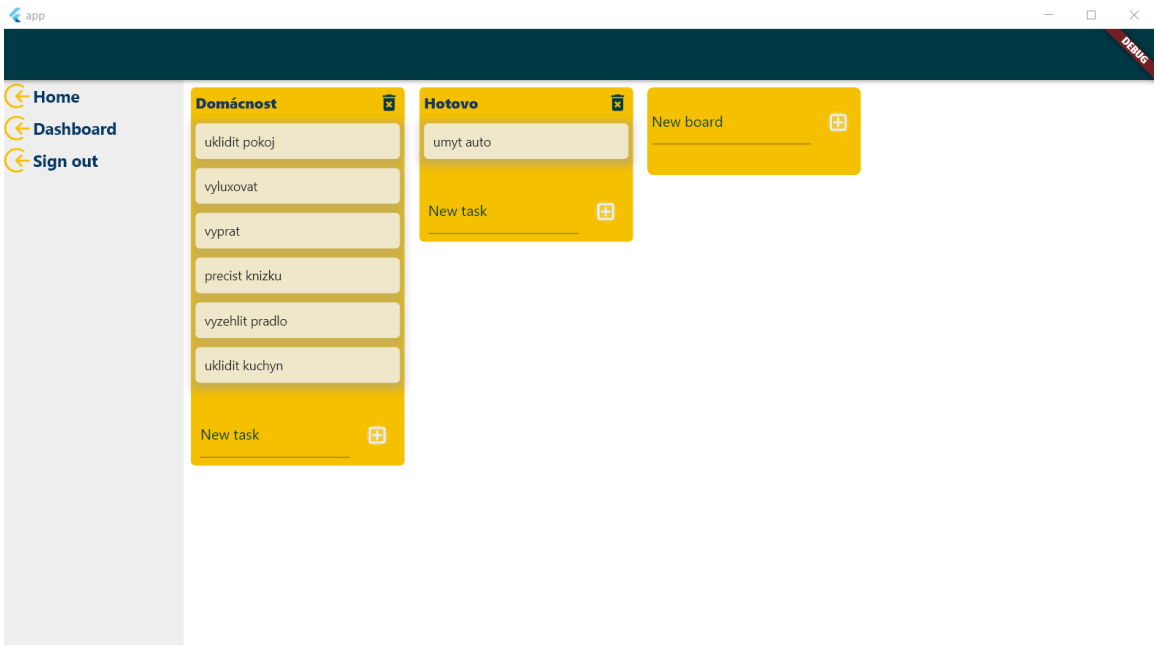
- /assets
- /canvaskit
- /icons
- favicon.png
- flutter_service_worker.js
- index.html
- main.dart.js
- manifest.json
- version.json

Výchozím souborem pro spuštění webové aplikace je soubor index.html. V něm je definováno, že má jako tělo aplikace použít obsah ze souboru main.dart.js, který zobrazí Flutter aplikaci. Rozložení webové aplikace je ukázáno na Obrázku 19. Zde je vidět, že oproti mobilním zařízením nabízí o mnoho více místa pro vykreslování komponent.



Obrázek 19: Ukázka demonstrační aplikace ve webovém prohlížeči. Zdroj autor

Poslední platformou, na které aplikace může běžet jsou počítače s operačním systémem Windows. Po sestavení aplikace vznikne spustitelný soubor s příponou exe. Není jej tedy nutné instaloval. Ukázka aplikace z aplikace je na Obrázku 20. Svým uživatelským rozhraním se podobá webové variantě aplikace.



Obrázek 20: Ukázka demonstrační aplikace v operačním systému Windows 10. Zdroj autor

Výsledná aplikace splnila požadavky - být spustitelná na výše uvedených platformách za použití co nejvíce sdíleného kódu pro všechny platformy. Samotný vývoj aplikace byl urychlen díky funkci hot reload, která umožňovala propsání změn do již běžící aplikace v emulátoru. Dalším kladem bylo oficiální úložiště balíčků pub.dev, které obsahuje velké množství již vytvořených komponent, které je možné použít ve své aplikaci. Ty často bývají hezky popsané v jejich dokumentaci, takže použití je většinou velmi snadné.

Oproti tomu tvoření uživatelského prostředí pomocí tak velkého množství widgetů může být pro někoho nepříjemné. Navzdory tomu se však Flutter jeví jako vhodná volba pro tvorbu multiplatformních aplikací.

8 Závěr

Cílem této diplomové práce bylo vytvořit multiplatformní aplikaci za použití frameworku Flutter a současně prozkoumat technologie, které se v současné době používají při tvorbě multiplatformních aplikací. V první části byly vysvětleny rozdíly v přístupech při tvorbě multiplatformních aplikací. Dále byly stručně popsány a porovnány nejpoužívanější technologie, ze kterých byl vybrán framework Flutter, který byl dále podrobněji popsán. Práce se dále zaměřila na metodiky, které využívá Flutter při tvorbě multiplatformních aplikací a ty byly prezentovány na demonstrační aplikaci.

Během práce bylo zjištěno, že Flutter nabízí sadu nástrojů, díky které je možné vytvářet plnohodnotné multiplatformní aplikace, které se snaží vypadat co nejvíce jako nativní. Způsob vytváření uživatelského prostředí pomocí skládání velkého množství komponentů (widgetů) může být pro některé vývojáře zbytečně obsáhlé, protože některé widgety nic nezobrazují, ale pouze nastavují specifické vlastnosti svému potomkovi, jako je například zarovnání na střed nebo uspořádání potomků do jednoho řádku.

Tato práce popsala základní principy při tvorbě multiplatformní aplikace za použití frameworku Flutter. Existují ovšem i další, díky kterým je možné docílit ještě ještě kvalitnějších multiplatformních aplikací.

Literatura

[1] Desktop vs Mobile vs Tablet Market Share Worldwide. Available from: https://gs.statcounter.com/platform-market-share/desktop-mobile-tablet/worldwide/

[2] Definition of hybrid mobile app. Available from: https://www.pcmag.com/encyclopedia/term/hybrid-mobile-app

[3] Frachet, M. Understanding the React Native bridge concept. Apr. 2020. Available from: https://medium.com/hackernoon/understanding-react-native-bridge-concept-e9526066ddb8

[4] Waranashiwar, J.; Ukey, M. Ionic Framework with Angular for Hybrid App Development: p. 2.

[5] Google Trends. Available from: https://trends.google.com/trends/explore

[6] Flutter architectural overview. Available from: https://docs.flutter.dev/resources/architectural-overview

[7] Sullivan, M. Flutter: Don't Fear the Garbage Collector. Jan. 2019. Available from: https://medium.com/flutter/flutter-dont-fear-the-garbage-collector-d69b3ff1ca30

[8] Sulyman, S. Client-Server Model. *IOSR Journal of Computer Engineering*, volume 16, Jan. 2014: pp. 57–71, doi:10.9790/0661-16195771.

[9] Bloc State Management Library. Available from: https://bloclibrary.dev/#/coreconcepts?id=bloc

[10] Paramitha, A. P. Getting Started_with_Flutter_Bloc Pattern | Mitrais Blog. Oct. 2021. Available from: https://www.mitrais.com/news-updates/getting-started-with-flutter-bloc-pattern/

[11] Flutter_Platform_embedding. Mar. 2022. Available from: https://docs.flutter.dev/resources/architectural-overview#platform-embedding

[12] Flutter_web_support. Mar. 2022. Available from: https://docs.flutter.dev/resources/architectural-overview#flutter-web-support

[13] Contextual layout. Available from: https://docs.flutter.dev/development/ui/layout/building-adaptive-apps#building-adaptive-layouts

[14] Mobile Operating System Market Share Worldwide. Available from: https://gs.statcounter.com/os-market-share/mobile/worldwide/

[15] Masi, E.; Cantone, G.; Calavaro, G.; et al. Mobile Apps Development: A Framework for Technology Decision Making. 2012, ISBN 978-3-642-36631-4, doi:10.1007/978-3-642-36632-1_4.

[16] Joorabchi, M. E.; Mesbah, A.; Kruchten, P. Real Challenges in Mobile App Development. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, Baltimore, Maryland: IEEE, Oct. 2013, ISBN 978-0-7695-5056-5, pp. 15–24, doi:10.1109/ESEM.2013.9. Available from: http://ieeexplore.ieee.org/document/6681334/

[17] What is the Difference Between Web Apps, Native Apps, Hybrid Apps and Progressive Web Apps? | HackerNoon. Available from: https://hackernoon.com/what-is-the-difference-between-web-apps-native-apps-hybrid-apps-and-progressive-web-apps-py19n2gdi

[18] Biørn-Hansen, A.; Rieger, C.; Grønli, T.-M.; et al. An empirical investigation of performance overhead in cross-platform mobile development frameworks. *Empirical Software Engineering*, volume 25, no. 4, July 2020: pp. 2997–3040, ISSN 1382-3256, 1573-7616, doi:10.1007/s10664-020-09827-6. Available from: https://link.springer.com/10.1007/s10664-020-09827-6

[19] Haldar, M. What is a PWA and why should you care? Mar. 2019. Available from: https://blog.bitsrc.io/what-is-a-pwa-and-why-should-you-care-388afb6c0bad

[20] WebView. Available from: https://developer.android.com/reference/android/webkit/WebView

[21] Corbalan, L.; Fernandez, J.; Cuitiño, A.; et al. Development frameworks for mobile devices: a comparative study about energy consumption. In *Proceedings*

of the 5th International Conference on Mobile Software Engineering and Systems, MOBILESoft '18, New York, NY, USA: Association for Computing Machinery, May 2018, ISBN 978-1-4503-5712-8, pp. 191–201, doi:10.1145/3197231.3197242. Available from: https://doi.org/10.1145/3197231.3197242

[22] Stahl, T.; Völter, M.; Czarnecki, K. *Model-driven software development: technology, engineering, management*. John wiley \& sons, inc. edition, 2006.

[23] Völter, M.; Stahl, T.; Bettin, J.; et al. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, June 2013, ISBN 978-1-118-72576-4, google-Books-ID: 9ww_D9fAKncC.

[24] Rasmusson Wright, Y.; Hedlund, S. *Cross-platform Frameworks Comparison : Android Applications in a Cross-platform Environment, Xamarin Vs Flutter*. 2021. Available from: http://urn.kb.se/resolve?urn=urn:nbn:se:bth-21696

[25] Ciman, M.; Gaggi, O. An empirical analysis of energy consumption of cross-platform frameworks for mobile development. *Pervasive and Mobile Computing*, volume 39, Aug. 2017: pp. 214–230, ISSN 15741192, doi:10.1016/j.pmcj.2016.10.004. Available from: https://linkinghub.elsevier.com/retrieve/pii/S1574119216303170

[26] Wu, W. React Native vs Flutter, cross-platform mobile application frameworks: p. 34.

[27] The Engine architecture · flutter/flutter Wiki. Available from: https://github.com/flutter/flutter

[28] Flutter Developers. 2019. Available from: https://docs.flutter.dev/resources/faq

[29] Dagne, L. Flutter for cross-platform App and SDK development: p. 37.

[30] Dickson, J. Xamarin Mobile Development: p. 18.

[31] Delia, L.; Galdamez, N.; Corbalan, L.; et al. Approaches to mobile application development: Comparative performance analysis. In *2017 Computing Conference*, London: IEEE, July 2017, ISBN 978-1-5090-5443-5, pp. 652–659,

doi:10.1109/SAI.2017.8252165. Available from:
http://ieeexplore.ieee.org/document/8252165/

[32] Oblak, M. Razvoj mobilne aplikacije v ogrodjih Ionic in NativeScript. 2021. Available from:
https://repozitorij.uni-lj.si/IzpisGradiva.php?id=124364

[33] Yang, Y.; Zhang, Y.; Xia, P.; et al. Mobile Terminal Development Plan of Cross-Platform Mobile Application Service Platform Based on Ionic and Cordova. In *2017 International Conference on Industrial Informatics - Computing Technology, Intelligent Technology, Industrial Information Integration (ICIICII)*, Wuhan: IEEE, Dec. 2017, ISBN 978-1-5386-2434-0, pp. 100–103, doi:10.1109/ICIICII.2017.28. Available from:
http://ieeexplore.ieee.org/document/8328596/

[34] Flutter - Build apps for any screen. Mar. 2022. Available from: //flutter.dev/

[35] Skia. Available from: https://skia.org/

[36] Dart VM. Feb. 2022. Available from: https://mrale.ph/dartvm/

[37] Isolate class - dart:isolate library - Dart API. Available from:
https://api.flutter.dev/flutter/dart-isolate/Isolate-class.html

[38] Introduction to widgets. Available from:
https://docs.flutter.dev/development/ui/widgets-intro

[39] Pokorný, M. A Multiplatform Mobile Application using Flutter Technology. May 2019. Available from: http://hdl.handle.net/10563/44468

[40] StatelessWidget class - widgets library - Dart API. Available from:
https://api.flutter.dev/flutter/widgets/StatelessWidget-class.html

[41] GlobalKey class - widgets library - Dart API. Available from:
https://api.flutter.dev/flutter/widgets/GlobalKey-class.html

[42] StatefulWidget class - widgets library - Dart API. Available from:
https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html

[43] Sublinear Layout. Available from:
https://docs.flutter.dev/resources/inside-flutter#sublinear-layout

[44] Petychakis, M.; Lampathaki, F.; Askounis, D. Adding Rules on Existing Hypermedia APIs. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, New York, NY, USA: Association for Computing Machinery, May 2015, ISBN 978-1-4503-3473-0, pp. 1515–1517, doi:10.1145/2740908.2743041. Available from: https://doi.org/10.1145/2740908.2743041

[45] auth0.com. JWT.IO - JSON Web Tokens Introduction. Mar. 2022. Available from: http://jwt.io/introduction

[46] Slepnev, D. State management approaches in Flutter. 2020, accepted: 2020-12-21T06:29:40Z. Available from: http://www.theseus.fi/handle/10024/355086

[47] Flutter_Bloc_A_Complete_Guide. Available from: https://dhruvnakum.xyz/flutter-bloc-a-complete-guide

[48] About Metal and This Guide. Mar. 2022. Available from: https://developer.apple.com/library/archive/documentation/Miscellaneous/Conceptual/MetalProgrammingGuide/Introduction/Introduction.html#//apple_ref/doc/uid/TP40014221

[49] OpenGL - The Industry's Foundation for High Performance Graphics. July 2011, section: API. Available from: https://www.khronos.org//

[50] ANGLE - Almost Native Graphics Layer Engine. Available from: https://chromium.googlesource.com/angle/angle/+/master/README.md

[51] Getting Started - WebGL Public Wiki. Available from: https://www.khronos.org/webgl/wiki/Getting_Started

[52] dartdevc: The Dart dev compiler. Available from: https://dart.dev/tools/dartdevc/

[53] Understanding Flutter_focus system. Available from: https://docs.flutter.dev/development/ui/advanced/focus

[54] Using_Actions_and_Shortcuts. Available from: https://docs.flutter.dev/development/ui/advanced/actions_and_shortcuts

[55] VisualDensity. Available from: https://docs.flutter.dev/development/ui/layout/building-adaptive-apps#building-adaptive-layouts

[56] Does Flutter use logical pixels? – QuickAdviser. Available from: https://quick-adviser.com/does-flutter-use-logical-pixels#What_are_logical_pixels

[57] connectivity_plus | Flutter Package. Available from: https://pub.dev/packages/connectivity_plus

[58] Flutter_Folder_Organization. Available from: https://www.section.io/engineering-education/flutter-folder-organization/

[59] Flutter_pubspec. Available from: https://docs.flutter.dev/development/tools/pubspec

[60] Build_and_release an Android app. Available from: https://docs.flutter.dev/deployment/android

Univerzita Hradec Králové
Fakulta informatiky a managementu

Zadání diplomové práce

Autor: **Bc. Miroslav Brandýský**

Studium: I2000030

Studijní program: N1802 Aplikovaná informatika

Studijní obor: Aplikovaná informatika

Název diplomové práce: **Implementace multiplatformní aplikace za použití frameworku Flutter**

Název diplomové práce AJ: Implementation of a multiplatform application using the Flutter framework

Cíl, metody, literatura, předpoklady:

Cíl: Implementovat adaptivní aplikaci a demonstrovat multiplatformitu za využití frameworku Flutter Osnova: 1. Úvod 2. Cíl práce 3. Multiplatformní vývoj 4. Charakteristika Fultteru 5. Multiplatformní vývoj za použití Flutteru 6. Souhrn výsledků 7. Závěr 8. Shrnutí a doporučení

FAYZULLAEV, Jakhongir. Native-like Cross-Platform Mobile Development: Multi-OS Engine & Kotlin Native vs Flutter. 2018.
DELIA, Lisandro, et al. Multi-platform mobile application development analysis. In: *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*. IEEE, 2015. p. 181-186.
WU, Wenhao. React Native vs Flutter, Cross-platforms mobile application frameworks. 2018.

Garantující pracoviště: Katedra informačních technologií,
Fakulta informatiky a managementu

Vedoucí práce: Mgr. Daniela Ponce, Ph.D.

Datum zadání závěrečné práce: 15.10.2021