POLSKO-JAPOŃSKA AKADEMIA TECHNIK KOMPUTEROWYCH

Wydział Informatyki

Katedra Inżynierii Oprogramowania

Inżynieria Oprogramowania i Baz Danych

Martyna Tołoczko
Nr albumu 17108

Porównanie progresywnych aplikacji webowych, hybrydowych oraz natywnych

Praca magisterska napisana
pod kierunkiem:

dr Mariusz Trzaska

Warszawa, wrzesień 2019

Streszczenie

Niniejsza praca porównuje trzy typy aplikacji mobilnych – aplikację natywną, hybrydową oraz progresywną aplikację webową. W tym celu przeprowadzono przegląd obecnie stosowanych rozwiązań do budowy aplikacji. Następnie zbudowane i przetestowane zostały trzy aplikacje typu social media o takich samych założeniach funkcjonalnych. Przedstawione zostały różnice w implementacji, debugowaniu, wyniki testów wydajnościowych oraz ogólne doświadczenie użytkownika. Rezultaty porównania zostały opisane w podsumowaniu pracy.

Słowa kluczowe: aplikacja natywna, aplikacja hybrydowa, technologie webowe, progresywna aplikacja webowa.

Podziękowania

Serdecznie dziękuję mojemu promotorowi, dr Mariuszowi Trzasce, za wsparcie przy pisaniu niniejszej pracy, zaangażowanie oraz poświęcony czas.

Spis treści

1. Wstęp

W dzisiejszych czasach świat aplikacji, zarówno desktopowych jak i mobilnych, jest ogromny i wciąż rośnie. Niestety dla przeciętnego użytkownika uciążliwy może być fakt, że często aplikacje, szczególnie te spoza pierwszych kilkunastu pozycji na liście popularności, choć działają idealnie na jednym systemie operacyjnym, mogą być zaniedbane lub zupełnie niedostępne na innym. Wraz z rozwojem technologicznym w naszych domach pojawia się coraz więcej urządzeń, na których jesteśmy w stanie instalować ulubione aplikacje – smartfony, tablety, smartwatche, telewizory, konsole, lodówki. Wiele z nich obsługuje już nawet technologie takie jak rozszerzona czy wirtualna rzeczywistość. Zakładając, że trend ten będzie się utrzymywał, przydatne byłoby rozwiązanie pozwalające na tworzenie aplikacji działających na wszystkich urządzeniach wymienionych powyżej (a także tych, które zagoszczą w naszych domach w przyszłości) oraz na wszystkich systemach operacyjnych. Koszty napisania takiej aplikacji w kilku językach programowania czy technologiach tak, aby spełnić to założenie, mogą okazać się bardzo dużę i nie do przyjęcia dla firm z mniejszym budżetem. Z pomocą zdają się przychodzić działające w przeglądarkach internetowych aplikacje webowe, czyli takie, które są pełni obsługiwane przez przeglądarki internetowe. Choć koncept ten istnieje od dawna, jeszcze nigdy nie były one tak blisko aplikacji natywnych czy też hybrydowych, a to za sprawą idei progresywnych aplikacji webowych. Praca ta ma na celu porównanie nowoczesnych aplikacji webowych z natywnymi oraz hybrydami, a także próbę odpowiedzi na pytanie – czy aplikacje webowe mają szansę zastąpić aplikacje hybrydowe i wyprzeć aplikacje natywne?

1.1. Cel pracy

Praca ta ma na celu zgłębienie zagadnień związanych z progresywnymi aplikacjami webowymi, porównanie ich z aplikacjami natywnymi oraz hybrydowymi, a także próbę określenia pod jakimi aspektami aplikacje natywne i hybrydowe różnią się od aplikacji webowych i jak kwestie te mają szansę rozwinąć się w przyszłości.

1.2. Rozwiązanie przyjęte w pracy

Aby osiągnąć zamierzony cel pracy zbudowane zostały trzy aplikacje. Pierwsza z nich została przygotowana dla systemu Android z użyciem języka Java. Druga, progresywna aplikacja webowa, została napisana przy pomocy języka JavaScript i biblioteki React.js. Trzecia aplikacja to hybryda

stworzona przy pomocy frameworka React Native. Wszystkie trzy aplikacje mają takie same założenia funkcjonalne, aby łatwo można było porównać implementację każdego z nich.

1.3. Rezultaty pracy

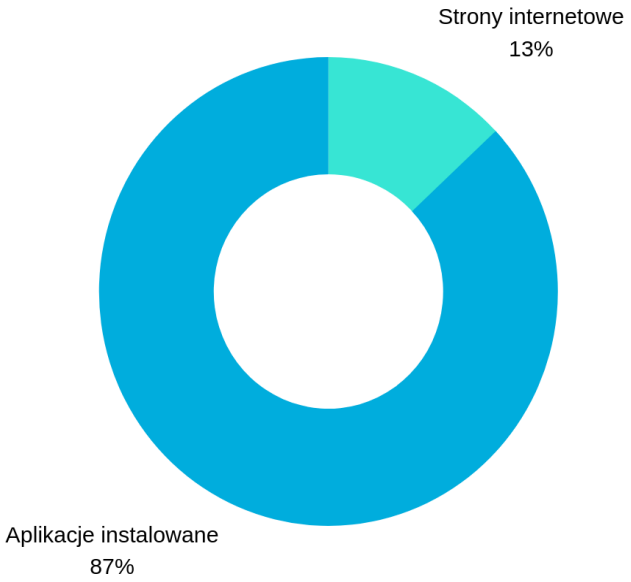
Rezultatem pracy jest porównanie wad i zalet trzech podejść do budowania aplikacji mobilnych – aplikacji natywnej, hybrydowej i progresywnej aplikacji webowej, na podstawie zbudowanych trzech aplikacji reprezentujących każde z tych podejść i przeprowadzonych testów wydajnościowych oraz użytecznościowych.

1.4. Organizacja pracy

Pierwsza część pracy to wstęp, określenie celu pracy i opisanie przyjętych rozwiązań. W drugiej części pracy opisane zostały najpopularniejsze rozwiązania przy budowaniu aplikacji natywnych – ich zalety oraz wady. Trzeci rozdział skupia się na przeglądzie technologii związanych z aplikacjami hybrydowymi, różnicami pomiędzy konwencjonalnymi hybrydami bazującymi na WebView i bardziej zaawansowanymi, kompilowanymi. Czwarta część pracy to opis aplikacji webowych – ich historia rozwoju, przegląd najpopularniejszych technologii wykorzystywanych przy ich tworzeniu. Piąta część pracy zgłębia szczegółowo działanie progresywnych aplikacji webowych. Opisane zostały standardy oraz narzędzia, które ułatwiają ich budowanie, stopień ich rozwoju na dziś, w tym kompatybilność z różnymi przeglądarkami, ich zalety oraz wady. Szósty rozdział pracy przedstawia zbudowane wcześniej aplikacje – natywną, hybrydową oraz webową, opisuje ich wymagania funkcjonalne i wykorzystywane w ich budowaniu narzędzia, programy, frameworki, rozszerzenia. W kolejnej części pracy dokonano porównania trzech powyższych aplikacji, opisano wyniki przeprowadzonych wcześniej testów, ich podobieństwa oraz różnice pod takimi względami jak szybkość działania, obciążenie procesora, user experience („doświadczenie użytkownika”), zużycie baterii, pamięci itp. Ostatni rozdział pracy opisuje wnioski wynikające z porównania tych aplikacji.

2. Aplikacje natywne

Zdecydowanie najpopularniejszym podejściem przy budowaniu aplikacji mobilnych jest tworzenie aplikacji natywnych. Są one pobierane przez użytkownika zazwyczaj z Google Store w przypadku aplikacji na Androida, Apple App Store w przypadku platformy iOS, lub też z Microsoft Store w przypadku aplikacji dla Windowsa[1], a następnie instalowane na danym urządzeniu. Aplikacja natywna działa tylko na jednej platformie dla której została stworzona i w określonym języku programowania dla danego systemu operacyjnego, dlatego też zazwyczaj konieczne jest tworzenie kilku osobnych aplikacji. W kontekście tej pracy aplikacje tworzone w technologiach webowych a następnie opakowywane w tzw. „wrappery" natywne nie są zaliczane do grupy aplikacji natywnych – ze względu na możliwość wykorzystania tego samego kodu w kilku aplikacjach na różne platformy traktowane są jako aplikacje hybrydowe i opisane zostały w rozdziale 3. Aplikacja natywna jest częstym wyborem dla wielu firm ze względu na swoje możliwości i szybkość działania. Ponieważ aplikacje te są lepiej wspierane i znacznie szybciej rozwijane, korzystanie z nich jest znacznie przyjemniejsze dla wielu użytkowników smartfonów, którzy znacznie więcej czasu spędzają właśnie na korzystaniu z aplikacji instalowanych na urządzeniu niż na stronach internetowych (Wykres 1).



Wykres 1. Czas spędzony na urządzeniach mobilnych: aplikacje instalowane a strony internetowe. Źródło: [2]

[1] Microsoft zdecydował się na wycofanie platformy Windows Phone – jej wsparcie będzie wciąż podtrzymywane, ale nie będą już raczej powstawać nowe aplikacje na tę platformę [10].

Dochód ze sprzedaży aplikacji natywnych i wyświetlania zawartych w nich reklam w 2020 roku szacowany jest na prawie 189 miliardów dolarów, a w 2022 roku pobranych zostanie ponad 258 miliardów aplikacji, co obrazuje wielkość potencjału tego rynku [8][9].

2.1. Przegląd najpopularniejszych technologii

Poniższe podrozdziały przedstawiają opis najpopularniejszych rozwiązań do tworzenia aplikacji natywnych dla trzech największych platform mobilnych – Android, iOS oraz Windows. W każdym z tych przypadków używane są inne języki programowania, narzędzia oraz IDE, czyli zintegrowane środowisko programistyczne.

2.1.1 Android

Android to najpopularniejsza obecnie platforma mobilna na rynku [30]. Aplikacje natywne na tę platformę pisane są w języku Java lub Kotlin, korzystając z dedykowanego IDE - Android Studio. Obsługa kompilacji oparta jest na narzędziu Gradle. Daje możliwość korzystania z wbudowanych kreatorów zawierających szablony tworzące podstawowe komponenty zgodne z Material Design oraz edycję układów na zasadzie drag-and-drop, dzięki czemu można przeciągać i upuszczać komponenty interfejsu użytkownika oraz podglądać ich widok na bieżąco. Ma także wbudowaną obsługę Google Cloud Platform umożliwiającą integrację np. z Firebase. Do testowania aplikacji deweloperzy mogą używać własnego urządzenia mobilnego lub też wbudowanego emulatora Android Virtual Device.

Android to projekt open-source, co daje programistom szerokie możliwości w zakresie dostosowywania aplikacji do potrzeb. Niestety, brak ścisłej kontroli w sklepie Google Play wiąże się z zagrożeniem ukrycia w aplikacji wirusów oraz niebezpiecznego oprogramowania.

2.1.2 iOS

Do zaprogramowania aplikacji na urządzenia Apple konieczna jest znajomość języka Objective-C lub Swift oraz urządzenie Mac, na którym można uruchomić IDE do budowania aplikacji na iOS, czyli Xcode. Podobnie jak Android Studio, oferuje on budowę podstawowych układów za pomocą drag-and-drop oraz podgląd aplikacji na żywo.

Istnieją badania [17], które potwierdzają, że budowanie aplikacji na platformę iOS jest szybsze i mniej kosztowne niż w przypadku aplikacji na Androida. Dzieje się tak, ponieważ na ogół Java wymaga napisania większej ilości kodu niż np. Swift, oraz dlatego, że Android to projekt open-source (w przeciwieństwie do zamkniętego ekosystemu Apple), co oznacza większą ilość urządzeń, komponentów oraz fragmentację oprogramowania do uwzględnienia. Z drugiej strony aplikacje na iOS muszą być zgodne ze znacznie bardziej restrykcyjnymi standardami Apple, co oznacza dłuższy

czas akceptacji jej w sklepie oraz możliwość jej odrzucenia [18]. Daje to jednak znacznie większą pewność niż w przypadku Google Play Store, że aplikacja jest bezpieczna do pobrania.

2.1.3 Universal Windows Platform

Ciekawym rozwiązaniem jest zaprezentowany przez Microsoft interfejs API nazwany Universal Windows Platform. Jego głównym założeniem jest pisanie aplikacji uniwersalnych, działających na wszystkich urządzeniach bazujących na Windows 10, w tym na komputerach, smartfonach, konsolach Xbox oraz okularach HoloLens bez potrzeby przepisywania ich dla każdego z nich. Aplikacja nie musi mieć zdefiniowanego typu urządzenia, które je obsługuje – program sam dostosowuje swoje funkcje do możliwości danego urządzenia. Ponadto, po podłączeniu aplikacji do stacji dokującej, istnieje możliwość uruchamiania aplikacji na telefonie w pełnej wersji komputerowej. Aplikacje te mogą być tworzone przy wykorzystaniu języków C#, C++, Visual Basic oraz JavaScript, natomiast sam ich wygląd kodowany jest w XAML, HTML lub DirectX [19].

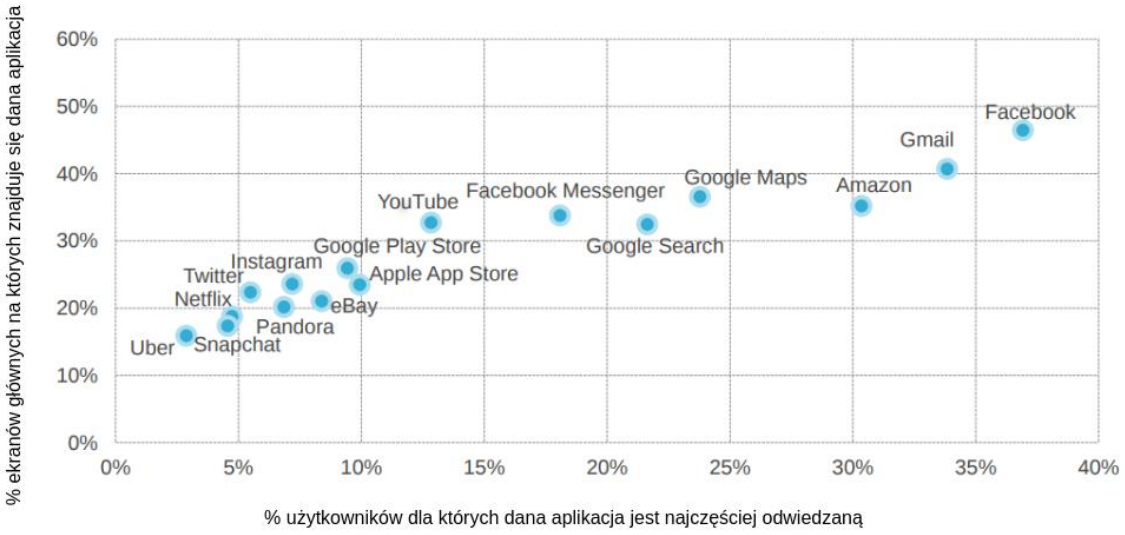
2.2. Zalety aplikacji natywnych

Aplikacje natywne oferują przede wszystkim szybkość działania zarówno po stronie klienta, jak i pod względem interakcji logiki aplikacji z interfejsem użytkownika dzięki bezpośredniej możliwości komunikowania się z API systemu operacyjnego. Poza wydajnością aplikacje natywne zapewniają także płynność przy przeglądaniu aplikacji, minimalny czas ładowania się (nawet przy pierwszym uruchomieniu) oraz przyjemny, dostosowany do urządzeń mobilny design, najczęściej oparty na wzorcach typu Material Design. Ma to największe znaczenie przy tworzeniu gier mobilnych, które są najbardziej obciążającym urządzenie typem aplikacji i w których szczególnie istotny jest brak zacinania się, przeskakiwania lub opóźnionej reakcji aplikacji.

Drugą największą ich zaletą jest pełny dostęp do natywnych funkcji i podzespołów urządzenia, takich jak kamera (przede wszystkich używana w aplikacjach typu social media), geolokalizacja, logowanie odciskiem palca, czy NFC (szczególnie istotne dla aplikacji bankowych). Ponadto działają offline – nawet jeśli pewne dane nie będą w tym trybie aktualizowane, aplikacja otworzy się i zazwyczaj pozwoli na podstawowe interakcje. Aplikacje te działają także w trybie pełnoekranowym.

Aplikacje natywne mają także kilka zalet pod względem ponownego angażowania użytkownika, do niedawna nieosiągalnych np. dla aplikacji webowych. Jedną z nich jest wykorzystywanie powiadomień push - aplikacja komunikuje się z użytkownikiem nawet gdy jest zamknięta, a telefon schowany jest w kieszeni. Z danych comScore wynika, że aż 63% osób w wieku 18-34 otwiera aplikację od razu po otrzymaniu powiadomienia. Bardzo istotna jest także ikona aplikacji na ekranie

głównym urządzenia. Widać bardzo silną korelację pomiędzy obecnością ikony aplikacji na ekranie głównym a częstotliwością otwierania jej (Wykres 2). Użytkownicy często otwierają aplikację nie w konkretnym celu, a dla zabicia czasu – gdy podnoszą telefon i widzą ikonę ulubionej aplikacji automatycznie w nią klikają. O tym, jak ważne są ikony na ekranie głównym świadczyć może również fakt, że dla grupy wiekowej 18-34 aż 21% użytkowników smartfonów przyznało, że zdarzyło im się usunąć aplikację, jeśli jej logo nie przypadło im do gustu [2].

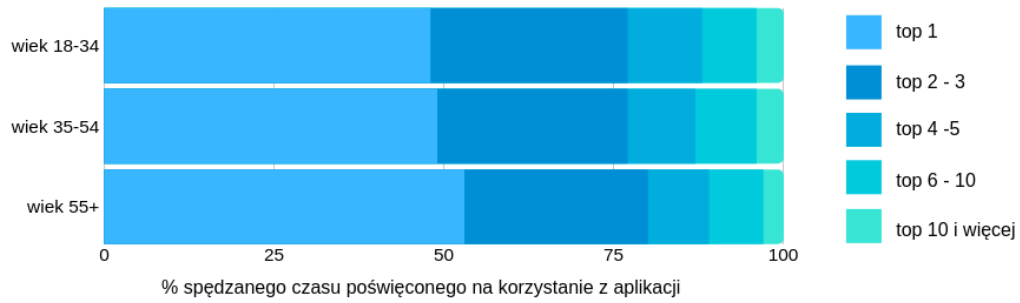


Wykres 2. Umiejscowienie ikony aplikacji na ekranie głównym a najczęściej odwiedzane aplikacje. Źródło: [2]

2.3. Wady aplikacji natywnych

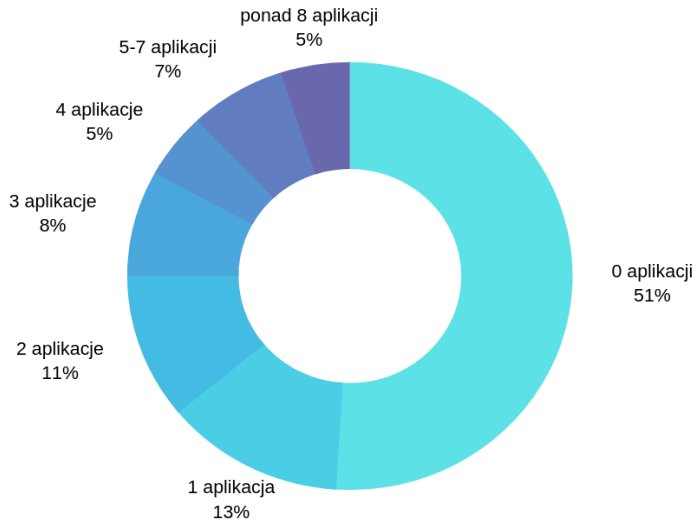
Największą wadą aplikacji natywnych jest fakt, że powinno się napisać ją dla co najmniej dwóch platform – Android i iOS (inne platformy, włączając w to Windows Phone, zostały wyparte już z rynku). To wiąże się z dodatkowymi, często bardzo wysokimi kosztami zbudowania i utrzymywania dwóch aplikacji napisanych w dwóch różnych językach programowania, najczęściej przez dwa różne zespoły programistów. Istnieje również ryzyko, że platforma, na którą stworzona została aplikacja zostanie niedługo wyparta z rynku, choć na chwilę obecną nic na to nie wskazuje, a czas i pieniądze poświęcone na budowanie aplikacji pójdą na marne. Przykładem może być Black Berry OS mająca w latach 2009 – 2010 aż 20% rynku, która już 5 lat później została całkowicie zastąpiona przez inne platformy [16]. Biorąc to pod uwagę można stwierdzić, że sukces tego rodzaju aplikacji może być uzależniony od sukcesu platformy.

Choć aplikacje natywne mają niepodważalnie bardzo wiele zalet, dość ciężko jest przekonać użytkownika do ich pobrania i powracania do nich. Użytkownicy spędzają aż połowę czasu poświęconego na korzystanie z aplikacji w tej z pierwszego miejsca w rankingu popularności, a praktycznie cały ten czas korzystając tylko z dziesięciu aplikacji (Wykres 3). Wynika z tego, że mniejsze firmy nie mają zbyt wielu szans przebicia się.



Wykres 3. Procent czasu poświęconego na korzystanie z aplikacji a pozycja w rankingu popularności. Źródło: [2]

Ponadto okazuje się, że średnio użytkownik nie pobiera miesięcznie ani jednej aplikacji (Wykres 4) – wiąże się to z tym, że zazwyczaj przy kupnie nowego telefonu pobiera on wszystkie potrzebne mu aplikacje i rzadko zdarza się, że potrzebuje kolejnej.



Wykres 4. Średnia ilość pobranych aplikacji miesięcznie na użytkownika. Źródło: [2]

Użytkownicy często niechętnie podchodzą do samego procesu instalacji aplikacji na swoje urządzenia. Obawiają się, czy aplikacja jest w pełni bezpieczna - czy nie pobiorą wraz z nią wirusa, czy aplikacja nie będzie łączyła się bez ich wiedzy z kamerą lub mikrofonem, czy nie będzie miała dostępu do wiadomości lub plików. Pomimo starań Google w sklepie z aplikacjami wciąż pojawiają się aplikacje oferujące np. czyszczenie pamięci telefonu, zarządzanie baterią lub filtry postarzające osoby na zdjęciach, które z pozoru nie wyglądają na podejrzane, jednak po zainstalowaniu mogą wyrządzić wiele szkód na urządzaniu oraz w kwestii naruszenia prywatności. W wielu przypadkach użytkownika zniechęca także fakt, że aplikacja może zająć zbyt wiele miejsca na telefonie – jest to także najczęstszy powód odinstalowywania aplikacji. Czasem użytkownik może uważać, że aplikacja nie będzie warta czasu instalacji i pieniędzy w przypadku aplikacji płatnych. Ten problem może być jednak wkrótce zminimalizowany dzięki wprowadzonemu niedawno przez Google rozwiązaniu Android Instant Apps [14]. Jest to możliwość otworzenia okrojonej wersji aplikacji bez konieczności instalowania jej, dzięki czemu użytkownik może zdecydować, czy aplikacja odpowiada jego oczekiwaniom i dopiero później podjąć decyzję o ewentualnej instalacji na urządzeniu.

Pewną wadą aplikacji natywnych jest też ryzyko, że ogromna ilość zainwestowanych pieniędzy i czasu nie przyniesie oczekiwanych zysków, ponieważ początkowe założenia były błędne, lub np. użytkownicy wcale nie potrzebują takiej aplikacji, bądź wybiorą inną. Przebudowanie projektu może okazać się nieopłacalne, a deweloperzy aplikacji natywnych nie mają możliwości korzystania z takich narzędzi jak Google Optimize i testy A/B mówiące o zachowaniu użytkowników na stronie, wedle których mogliby modyfikować design, kontent i procesy biznesowe. Z drugiej strony może pojawić się także niebezpieczeństwo polegające na zbyt dużym zainteresowaniem aplikacją, na które firma nie była przygotowana. Taka sytuacja wymaga dużego wsparcia ze strony zespołu deweloperskiego – jeśli np. nie nadążałby z rozwiązywaniem zgłaszanych przez użytkowników problemów, aplikacja szybko otrzymałaby dużą ilość negatywnych opinii w sklepie z aplikacjami, które mogłyby nawet zablokować dalszy sukces aplikacji [15].

3. Aplikacje hybrydowe

Aplikacje hybrydowe to pewnego rodzaju kompromis pomiędzy aplikacjami natywnymi i webowymi. Rozwiązują główny problem związany z aplikacjami natywnymi – pisane najczęściej w technologiach webowych z użyciem odpowiednich frameworków mogą być uruchamiane na różnych popularnych platformach (najczęściej Android i iOS), a nie tylko na jednej. Mają dostęp do wybranych funkcji urządzeń.

3.1. Przegląd najpopularniejszych technologii

W przypadku aplikacji hybrydowych bardzo dużo zależy od wybranego frameworka. Do hybryd możemy zaliczyć zarówno aplikacje wykorzystujące WebView (widoki mobilne, jak aplikacja webowa, uruchamiane w aplikacji natywnej) oraz aplikacje łączące podejście webowe z natywnym (np. łączące kod JavaScript z kodem Java lub Objective-C czy Swift). Najpopularniejsze technologie na dziś [25] zostały przedstawione poniżej.

3.1.1 React Native

React Native to częsty wybór osób, które już wcześniej miały styczność z React.js[2] – jest to framework także stworzony przez Facebooka i używany przez największe firmy na świecie takie jak Facebook, Instagram czy Uber. Technologia oparta jest na JavaScripcie. Główne zalety to świetna wydajność i szybkość, relatywnie małe zużycie pamięci, wiele dostępnych komponentów i bibliotek i dość mały czas potrzebny do stworzenia aplikacji [12]. Wymaga jednak dostosowywania kodu do konkretnej platformy. W ramach potrzeby daje możliwość pisania modułów także w językach Objective-C, Java, Kotlin i Swift.

Głównym założeniem React Native jest zdanie „learn once, write everywhere”, co oznacza, że pomimo konieczności dostosowywania kodu do każdej platformy osobno zawsze wykorzystywana jest do tego ta sama technologia – React (Native).

React Native to nowoczesna wersja aplikacji hybrydowej. Choć wykorzystuje technologie webowe, aplikacja napisana z jego użyciem tak naprawdę jest prawdziwą aplikacją natywną, w odróżnieniu od typowo hybrydowych aplikacji opartych na Ionicu czy Cordovie, które przypominają aplikacje natywne, ale uruchamiają je na komponentach typu WebView, czyli są

[2] React.js to najpopularniejsza obecnie biblioteka JavaScript do tworzenia aplikacji webowych, stworzona przez Facebooka.

aplikacjami webowymi umieszczonymi w kontenerach, które umożliwiają ich działanie jako aplikacje natywne. W przypadku React Native cały interfejs użytkownika jest kompilowany na kod natywny dla iOS lub Androida, co zapewnia świetną wydajność i szybkość oraz zaawansowany dostęp do systemu operacyjnego [12].

Choć React Native bardzo szybko się rozwija przez co zdarzają się błędy oraz jest bardzo zależny od zewnętrznych bibliotek, ma ogromny potencjał, o czym może świadczyć fakt, że jest używany przez jedne z największych korporacji oferujących serwisy internetowe jak np. Netflix.

3.1.2 Flutter

Flutter to dość świeże, ale zyskujące dużą popularność rozwiązanie open-source przedstawione w 2018 przez Google. Można w nim tworzyć zaawansowane aplikacje działające na Androidzie i iOS. Minusem może być konieczność programowania w jeszcze nie tak popularnym języku Dart co skutkuje niewielką ilością specjalistów dostępnych na rynku pracy. Cechuje się dobrą wydajnością i responsywnością, ale niestety także dużym rozmiarem zbudowanej aplikacji w porównaniu do np. React Native [13].

3.1.3 PhoneGap

PhoneGap to zaawansowany framework, który umożliwia tworzenie aplikacji nie tylko na platformy Android i iOS, ale też na mniej popularne jak BlackBerry czy Symbian – wszystko to bazując na tym samym kodzie [25]. To właśnie z PhoneGap wykształciła się technologia Apache Cordova, która umożliwia opakowywanie aplikacji webowych w natywne kontenery na zasadzie pewnego rodzaju mobilnej przeglądarki. Udostępnia ona także most pomiędzy kodem JavaScript a natywnymi API [6]. Niestety ma to negatywny wpływ na szybkość działania aplikacji oraz możliwe gorsze działanie interfejsu użytkownika jeśli chodzi o animacje czy płynność działania.

3.1.4 Ionic

Kolejnym bardzo popularnym frameworkiem jest Ionic, czyli framework open-source oparty na CSS, HTML i AngularJS (lub JavaScript). Może być testowany w przeglądarkach. Także opiera się na technologii Cordova i WebView, przez co ma znacznie mniejsze możliwości niż np. React Native.

W przeciwieństwie do React Native temu frameworkowi przyświeca sentencja „write once, use everywhere”, czyli raz napisany kod będzie działał od razu na wszystkich platformach bez konieczności dostosowywania go. Ma od razu wbudowane style i komponenty zgodne np. ze standardami Material Design [26].

3.1.5 Framework 7

Framework 7 jest - podobnie jak Ionic i PhoneGap - oparty na technologii Apache Cordova i tak jak one wyświetla aplikacje pisane w technologiach webowych jako WebView. Ma wbudowane komponenty dostosowane do wyglądu iOS i Material Design, umożliwia pisanie aplikacji w React czy też Vue [6].

3.1.6 Xamarin

Dość popularnym rozwiązaniem jest także tworzenie aplikacji hybrydowych korzystając z platformy Xamarin. Pozwala ona budować aplikacje na platformy Android, iOS oraz Windows w języku C#. W 2016 roku została przejęta przez Microsoft i udostępniona jako open-source. Dzieli się na dwa podejścia: Xamarin.Native oraz Xamarin.Forms. Pierwsze z nich stworzone zostało z myślą o aplikacjach wymagających zaawansowanego interfejsu, wykorzystujących wiele natywnych API, drugie natomiast pozwala szybko budować proste aplikacje [53]. W przeciwieństwie do aplikacji pisanych w technologiach webowych, Xamarin pozwala na tworzenie aplikacje praktycznie nie różniących się od aplikacji natywnych. Minusem jest dość mała popularność w porównaniu do technologii webowych i natywnych (według danych z ankiety StackOverflow w 2018 roku z Xamarin'a korzystało jedynie 7,4% programistów, w porównaniu do 28,3% korzystających z React'a [52]), co może się wiązać z trudnością znalezienia wykwalifikowanych deweloperów [53].

3.2. Zalety aplikacji hybrydowych

Główną zaletą aplikacji hybrydowych jest dużo niższy koszt budowy niż w przypadku aplikacji natywnych, dzięki temu, że działają zarówno na Androidzie jak i na iOS. Programista znający tylko jeden język programowania jest w stanie tworzyć aplikacje na obie te platformy [12].

Ze względu na dużą różnorodność frameworków do budowania aplikacji hybrydowych możliwe jest dopasowanie technologii do preferencji i umiejętności zespołu deweloperskiego. Każde z tych narzędzi posiada inne zalety i ma różne zastosowania. Wiele zależy od tego, jaki aspekt będzie najistotniejszy dla danego projektu – może być to prędkość działania i kompilacji, niski koszt, user experience, dobre wsparcie, skalowalność czy też łatwość debugowania i testowania.

Teoretycznie aplikacje hybrydowe oferują możliwość korzystania z natywnych komponentów działających znacznie szybciej i płynniej niż w aplikacjach webowych, a także z funkcji zupełnie niedostępnych dla przeglądarek.

3.3. Wady aplikacji hybrydowych

Niestety aplikacje hybrydowe nie są tak wydajne jak aplikacje natywne, gdy w grę wchodzi wielowątkowa logika biznesowa (szczególnie jeśli chodzi o aplikacje oparte na WebView). Mogą wolniej działać i przycinać się w przypadku bardziej zaawansowanych aplikacji, co sprawia, że nie sprawdzają się zbyt dobrze np. w grach.

Minusem może być także fakt, że nieznajomość Javy lub Swifta / Objective-C może znacznie utrudnić debugowanie lub rozszerzanie o nowe wtyczki kompilowanej aplikacji hybrydowej. Często zdarza się, że zmiany trzeba nanieść właśnie w tych językach.

Podobnie jak w przypadku zalet, wady hybryd zależą od wybranej technologii, w której są napisane. Najczęstszym problemem jest wydajność i skalowalność takiej aplikacji, a także możliwości debugowania i zależność od zewnętrznych modułów i wtyczek.

Dość wysoki może okazać się także koszt utrzymania i rozwoju aplikacji ze względu na szybki rozwój danego frameworka, szczególnie jeśli chodzi o React Native [12].

4. Aplikacje webowe

„Jeśli daną aplikację możesz zbudować w oparciu o HTML, CSS i JavaScript, najprawdopodobniej tak właśnie powinieneś zrobić.” [24]

Stark J., Jepson B.

Aplikacje webowe różnią się od aplikacji natywnych przede wszystkim tym, że trzymane są na zewnętrznym serwerze, pobierane na żądanie i interpretowane w językach HTML, CSS i JavaScript przez przeglądarkę internetową – dzięki temu aplikacja webowa jest niezależna od platformy i może działać na każdym urządzeniu, które ma możliwość zainstalowania przeglądarki (m.in. na komputerach stacjonarnych, laptopach, smartfonach, tabletach, telewizorach, smartwatchach i konsolach). W przeciwieństwie do zwykłej strony internetowej ma charakter nie tylko informacyjny, ale także interaktywny.

Aplikacje webowe wykorzystują model klient-serwer, który określa relacje między dostawcą (serwer) i odbiorcą (klient – przeglądarka internetowa) usługi. W przypadku aplikacji internetowej komunikacja między klientem a serwerem odbywa się za pośrednictwem protokołu komunikacyjnego HTTP (ang. Hypertext Transfer Protocol). Komunikacja po stronie serwera zazwyczaj odbywa się przy wykorzystaniu języków takich jak PHP, Node.js a po stronie klienta – w JavaScript.

4.1. Przegląd technologii tworzenia nowoczesnych aplikacji webowych

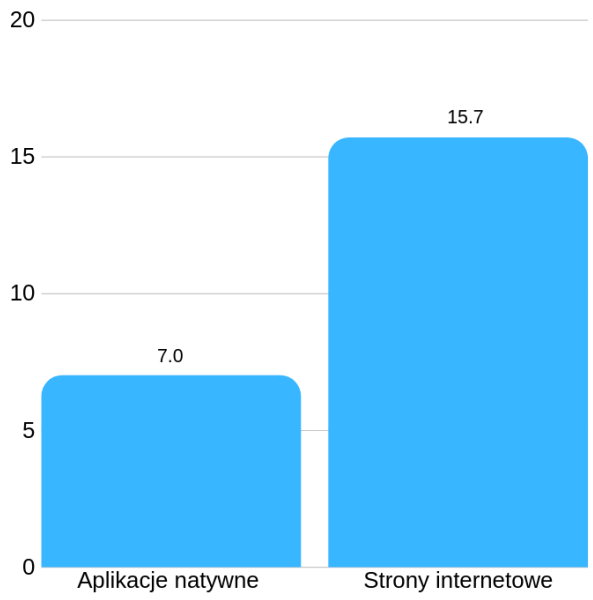
Aplikacje webowe bardzo rozwinęły się w przeciągu ostatnich kilku lat. Poniżej zaprezentowano trzy najpopularniejsze obecnie podejścia do ich tworzenia: Single Page Applications (SPA), przyspieszone strony mobilne (AMP) oraz progresywne aplikacje webowe (PWA).

4.1.1 Single Page Applications (SPA)

W ostatnich latach dużą popularność zyskały tzw. Single Page Apps, tworzone przy pomocy technologii webowych, czyli HTML, CSS oraz JavaScript. Za aplikację webową uważa się program uruchamiany w przeglądarce jak zwykła strona internetowa, ale która niekoniecznie ma osobny adres URL do każdej podstrony i która oprócz statycznych treści ma pewne interaktywne funkcje z których użytkownik może korzystać. Kod takiej aplikacji i DOM pobierane są raz (przy wejściu na stronę) i modyfikowane na bieżąco wraz z interakcjami użytkownika i wprowadzanymi przez niego danymi, tak więc użytkownik nie doświadcza przeładowania czy też odświeżania strony. W większości

przypadków aplikacja taka pisana jest w JavaScripcie przy użyciu technologii takich jak React.js, Vue czy też Angular. Dane zamiast z serwera najczęściej pobierane są w ramach komunikacji z różnymi API. Niestety, renderowanie aplikacji po stronie klienta stwarza problemy z pozycjonowaniem strony, jednak istnieje coraz więcej rozwiązań, które pomagają robotom Google czytać zawartość strony, jak np. Server Side Rendering – pre-renderowanie kodu JavaScript po stronie serwera.

Główną zaletą takich aplikacji jest ich zasięg – może być uruchomiona na każdym urządzeniu mającym przeglądarkę internetową, nie ma konieczności tworzenia kilku wersji na różne platformy. Dodatkowo, przy założeniu, że zastosowane zostały techniki wspomagające SEO, aplikacje te znajdowane są przez wyszukiwarki internetowe jak zwykłe strony internetowe. To oznacza, że użytkownik może na nie trafić znacznie łatwiej niż w przypadku aplikacji natywnych – według statystyk strony internetowe mają aż 2.2 razy większy zasięg niż aplikacje natywne (Wykres 5).



Wykres 5. Nowi użytkownicy miesięcznie (w milionach). Źródło: [2]

Zaletą aplikacji webowych jest niewątpliwie także brak konieczności ich instalowania i fakt, że są darmowe. Nie wymagają także aktualizacji – zawsze mamy pewność, że wszyscy użytkownicy korzystają z najnowszej wersji. Dodatkowo koszty stworzenia takiej aplikacji są na ogół niższe niż w przypadku aplikacji natywnej.

Niestety zwykłe aplikacje webowe mają także kilka wad. Przede wszystkim do ich działania konieczne jest stabilne połączenie z Internetem. W miejscach ze słabym zasięgiem lub za granicą,

gdzie użytkownik nie korzysta akurat z Wi-Fi, aplikacje te stają się bezużyteczne. Mogą też działać wolniej lub częściej się zacinać niż aplikacji natywne. Na ogół nie mają także zbyt łatwego dostępu do natywnych funkcji urządzenia, nie mogą wykorzystywać ich pełnych możliwości.

4.1.2 Przyspieszone strony mobilne

W przypadku stron oferujących głównie treści statyczne lub proste treści dynamiczne istotne dla SEO (takie jak blog) dobrym rozwiązaniem okazać się mogą tzw. Accelerated Mobile Pages, czyli przyspieszone strony mobilne zaprezentowane przez Google i Twittera. Nie są one dostosowywane do wielkości i możliwości każdego urządzenia jak ma to miejsce w przypadku SPA – AMP są stworzone specjalnie dla urządzeń mobilnych i budowane tylko z myślą o nich. Ich głównym założeniem jest błyskawiczne działanie i możliwość szybkiego przechodzenia między podstronami. Standardy AMP są z tego powodu bardzo restrykcyjne, np. wiele elementów potencjalnie obciążających przeglądarkę jest całkowicie zabronionych, a arkusze CSS i kod JavaScript powinny być jak najbardziej zoptymalizowane, w przeciwnym wypadku strona nie zostanie zatwierdzona przez Google jako AMP. Dzięki specjalnemu skryptowi aplikacji AMP wszystkie zdjęcia ładują się na zasadzie lazy-loading, czyli w momencie przewinięcia strony do momentu wyświetlania grafiki, jednocześnie nie powodując wrażenia ‘skakania’ strony. Istotny jest także fakt, że aplikacje te przechowywane są bezpośrednio na serwerach Google.

Do zalet AMP należy zaliczyć także czytelność (wszystkie są w miarę podobne przez ograniczenia standardów) oraz oszczędność baterii i transferu, ponieważ są lekkie i nie mają niestandardowych skryptów JavaScript. Dodatkowo są znacznie lepiej indeksowane przez wyszukiwarkę Google i specjalnie przez nią oznaczane.

Prostota AMP wiąże się także z kilkoma ograniczeniami – brak możliwości dodawania niestandardowych skryptów, koniecznych np. do analityki i testów A/B, możliwa utrata spójności z wyglądem oryginalnej strony internetowej firmy ze względów na ograniczone możliwości stylowania, oraz widoczny adres google.com [23].

4.1.3 Progresywne aplikacje webowe

Progresywne aplikacje webowe są największym przełomem w świecie technologii dla urządzeń mobilnych od czasu wprowadzenia SPA. Ten zaprezentowany i silnie promowany przez Google termin oznacza zestaw dobrych praktyk i standardów które przekształcająca zwykłą aplikację webową lub stronę internetową w aplikację działająca prawie tak samo, jak aplikacja natywna. Ich cechy oraz wady i zalety omówione zostały w następnym rozdziale.

5. Progresywne aplikacje webowe

„Aplikacje te nie są pakowane i publikowane w sklepach - to po prostu strony internetowe, które wzięły wszystkie odpowiednie witaminy.” [54]

Russel A.

Progresywne aplikacje webowe to termin odnoszący się do kilku funkcji, które można dodać do każdej istniejącej już aplikacji webowej lub strony internetowej, aby ich działanie przypominało bardziej aplikacje natywne, czyli responsywny design, działanie offline, możliwość dodania aplikacji do ekranu głównego, dostęp do kamery, geolokalizacji, synchronizacja danych w tle powiadomienia push etc. Zaletą takiego rozwiązanie jest to, że nie trzeba implementować wszystkich tych funkcji naraz, a jedynie te, które są potrzebne. Co więcej, jest możliwość dodania ich w taki sposób, że będą działały nie tylko na wszystkich systemach operacyjnych, ale także na wszystkich przeglądarkach, a nawet ich starszych wersjach. Jeśli nawet jest to z jakiegoś powodu niemożliwe, można po prostu wyłączyć te funkcje dla danego przypadku, ale użytkownik wciąż może korzystać z innych funkcjonalności aplikacji, co dodatkowo zwiększa zasięg [3]. Czasem też nie warto wprowadzać wszystkich funkcji PWA ze względu na user experience – przypadkiem może być strona banku jako strona czysto produktowa i informacyjna. Szybkość ładowania strony jaką zapewniłoby przekształcenie jej w PWA mogłaby znacznie poprawić konwersję i sprzedaż, jednak nie powinna proponować dodawania jej do ekranu głównego (aby nie myliła się z natywną aplikacją transakcyjną, z której użytkownicy korzystają znacznie częściej), oraz nie powinna przechowywać zbyt wielu treści dla dostępu offline – strona taka nie jest raczej odwiedzana aż tak często.

5.1. Standardy progresywnych aplikacji webowych

Standardy progresywnych aplikacji webowych są szczegółowo opisane w tzw. PWA Checklist przygotowanym przez Google [7], z podziałem na wersje podstawową i rozszerzoną. W wersji podstawowej PWA powinna mieć następujące cechy:

- serwowanie przez protokół HTTPS, co zapewnia bezpieczne połączenie,
- responsywny design na urządzeniach mobilnych,

- dostępność wszystkich adresów URL aplikacji w trybie offline (nawet jeśli część z nich przekierowuje do wewnętrznej strony typu 404 lub strony z informacją o niedostępności w trybie offline),
- poprawnie zbudowany Web App Manifest, pozwalający na dodawanie aplikacji do ekranu głównego,
- szybkie ładowanie się aplikacji nawet przy użyciu sieci 3G oraz przy pierwszym wejściu na stronę, gdy nie jest jeszcze zapisana w cache,
- działanie na różnych przeglądarkach (m.in. Chrome, Edge, Firefox, Safari),
- niezauważalne przechodzenie między stronami – strona nie ładuje się od nowa przy każdej zmianie adresu, DOM modyfikowany jest na bieżąco,
- unikalny adres URL dla każdej podstrony.

Istnieje także możliwość rozszerzenia aplikacji PWA o dodatkowe opcje, które zapewniają użytkownikom znacznie lepsze doświadczenia offline, są jeszcze bardziej interaktywne i jeszcze bardziej przypominają aplikacje natywne:

- w kwestii **SEO** – aplikacja powinna być dobrze indeksowana przez Google, korzystać ze znaczników Schema.org oraz meta danych Open Graph dla mediów społecznościowych, co pozwala na pokazywanie grafiki i krótkiego opisu strony w momencie przesyłania jej linka np. na Facebooku czy Twitterze, powinna posiadać także kanoniczne adresy URL oraz korzystać z History Api, dzięki czemu użytkownik może poruszać się po aplikacji za pomocą przycisków ‘wstecz’ i ‘przejdź dalej’ w swojej przeglądarce lub odnaleźć daną podstronę w historii przeglądania,
- w kwestii **user experience** – aplikacja nie powinna ‘skakać’ przy ładowaniu się (ma to miejsce np. na wielu stronach internetowych, gdzie najpierw ładuje się sam tekst a dopiero potem grafiki przesuwające kontent strony w dół), przejście wstecz powinno przenieść użytkownika do miejsca do którego poprzednio przewinął, pola tekstowe nie powinny być zasłaniane przez natywną klawiaturę urządzeń mobilnych w momencie pisania, użytkownik powinien mieć łatwy dostęp do podzielenia się treścią aplikacji w mediach społecznościowych nawet w trybie standalone czy pełnoekranowym, gdzie domyślny pasek przeglądarki jest schowany i nie jest wyświetlany link strony,

- w kwestii **cache** – aplikacja powinna korzystać z podejścia ‘cache first’, czyli zaciągać treść najpierw z pamięci przeglądarki, a dopiero potem z sieci oraz powinna informować użytkownika, jeśli znajduje się w trybie offline,
- W kwestii **powiadomień push** - użytkownik powinien być zapytany o zgodę na wyświetlanie powiadomień push i być informowany, jeśli takie będą do niego wysyłane, prośby o włączenie powiadomień nie powinny być natarczywe, aplikacja powinna mieć także opcję wyłączenia powiadomień push,
- dodatkowo - użytkownik powinien być zalogowany na różnych urządzeniach za pomocą Credential Management API[3] oraz (w przypadku e-commerce) powinien mieć możliwość łatwego dokonywania płatności za pomocą Payment Request API[4].

5.1.1 Web App Manifest

Dodanie do ekranu głównego ma ogromne znaczenie dla powracania klienta do danej aplikacji. W dzisiejszych czasach jest to opcja niezbędna np. dla każdego większego portalu informacyjnego, gdzie konkurencja jest dość wysoka, a gdzie często użytkownicy wchodzą w wolnej chwili, nie podnosząc telefonu specjalnie w tym celu, tylko automatycznie po zobaczeniu ikony, aby sprawdzić ‘co nowego’. Rzadko natomiast szukaliby aplikacji do kilku portali informacyjnych w sklepie aplikacji. W tej branży bardzo istotny jest także tryb pełnoekranowy, który ułatwia czytanie i przeglądanie artykułów. Przykładem efektywnego wprowadzenia PWA może pochwalić się Forbes, gdzie czas ładowania skrócił się do 1 sekundy, a długość sesji użytkownika wzrosła o 43%.

W przypadku wyskakującego okna informującego o możliwości instalacji aplikacji istotne jest, aby nie był wyświetlany zbyt często i nachalnie, a także w momentach gdy np. wyświetlany jest inny popup lub gdy użytkownik nie powinien być w danym momencie rozpraszany.

5.1.2 Bezpieczeństwo

Wszystkie aplikacje progresywne powinny być obsługiwane przez protokół HTTPS ze względów bezpieczeństwa. Związane jest to głównie z wykorzystywaniem Service Worker’a, który umożliwia przejmowanie, wytwarzanie oraz filtrowanie odpowiedzi. Protokół HTTPS daje gwarancję, że dany Service Worker nie został zmodyfikowany w niepożądany sposób.

---

[3] API przeglądarki, które zapewnia bezproblemowe logowanie się na różnych urządzeniach oraz zapamiętywanie użytkownika.

[4] standard mający na celu ułatwienie przeprowadzania transakcji, oferujący ujednolicony wygląd oraz bezpieczeństwo

Ponadto, tak jak w przypadku każdej aplikacji webowej, dane wprowadzane przez użytkownika zapisywane są na zewnętrznym, zabezpieczonym serwerze, co zapewnia ich bezpieczeństwo w przypadku awarii urządzenia.

5.1.3 Service Worker

Aplikacje webowe operujące na protokole HTTPS mogą wysłać prośbę do danej przeglądarki o zainstalowanie oddzielnego skryptu zwanego Service Worker. Działa on w tle niezależnie od aplikacji, nasłuchuje konkretnych zdarzeń i może na nie reagować nawet po zamknięciu przeglądarki i przy braku połączenia z Internetem, dzięki czemu PWA może działać offline. Jest niezbędny do zaimplementowania np. powiadomień push czy też synchronizacji danych w tle, a w przyszłości może także geofencing. Ponieważ działa na osobnym wątku niż kod JavaScript aplikacji, nie ma dostępu do DOM'u. Jest za to dostępny na każdej stronie aplikacji, w przeciwieństwie do zwykłego kodu JavaScript, który przypisywany jest do każdej podstrony osobno.

Cykl życia Service Worker'a rozpoczyna się po wysłaniu prośby o rejestrację go do przeglądarki, następnie zostaje zainstalowany. Jeśli w przeglądarce nie działa w danym momencie jego stara wersja (różniąca się przynajmniej o jeden bajt od nowej), zostaje on aktywowany przez przeglądarkę, w przeciwnym wypadku zostanie automatycznie aktywowany dopiero po zamknięciu wszystkich kart i otwarciu przeglądarki na nowo. W przypadku dłuższego czasu bezczynności (braku nadchodzących wydarzeń, na które mogły reagować) Service Worker przechodzi w stan uśpienia i zostaje aktywowany ponownie dopiero, gdy będzie taka potrzeba.

5.1.4 Responsywność

Interfejs progresywnej aplikacji webowej powinien być responsywny i intuicyjny niezależnie od rozmiaru ekranu, na którym odtwarzana jest aplikacja. Bardzo istotny jest także dobrze zaprojektowany UX, czyli „doświadczenie użytkownika". Często aplikacje PWA są inspirowane standardami określającymi wygląd aplikacji natywnych, np. Material Design dla Androida.

5.2. Narzędzia deweloperskie dla progresywnych aplikacji webowych

Budowanie aplikacji PWA wiąże się z koniecznością przestrzegania wielu zasad. Aby na bieżąco sprawdzać zgodność ze standardami danej aplikacji oraz jej stan firma Google wydała kilka pomocnych narzędzi do przeprowadzania audytów aplikacji webowych. Zostały one przedstawione w poniższych podrozdziałach.

5.2.1 Google Lighthouse

Podstawowym narzędziem do testowania standardów progresywnych aplikacji webowych jest rozszerzenie Lighthouse dostępne w Chrome Web Store. Jest w stanie m.in. przetestować aplikację w warunkach offline i przy słabym połączeniu internetowym, szybkość ładowania się aplikacji, czy jest serwowana z bezpiecznego źródła i czy stosuje dobre praktyki budowania PWA. Po wykonaniu audytu Lighthouse oblicza wynik dla danej strony w skali 0 – 100 w każdej z następujących kategorii:

- prędkość,
- PWA,
- dostępność,
- dobre praktyki,
- SEO.

5.2.2 Application – Google Chrome Dev Tools

Aktualny stan progresywnej aplikacji webowej można śledzić w zakładce Application w narzędziach deweloperskich Chrome. Znajdują się tam informacje o zainstalowanym Web App Manifest i Service Worker (także opcja zainstalowania / odinstalowania / odświeżenia go), dane przechowywane w Local i Session Storage oraz w IndexedDB, ciasteczka, pamięć podręczna oraz możliwość dodania aplikacji do ekranu głównego.

5.3. Stopień rozwoju progresywnych aplikacji webowych

Ze względu na to, że brak PWA w sklepach z aplikacjami może generować stratę pewnej części klientów, a same PWA nie są jeszcze traktowane przez użytkowników jako pełnoprawne aplikacje, Google ogłosił pod koniec stycznia 2019, ze strony zgodne z PWA będą pojawiały się w Play Store. W przypadku iOS natomiast problem ten jest rozwiązywany za pomocą wrapperów, gdzie natywne jest tylko opakowanie aplikacji, ale wszystkie funkcje zaciągane są z PWA. To powoduje dodatkowe zwiększenie i tak już ogromnego zasięgu tych aplikacji webowych.

Ciekawą kwestią jest także istotność PWA w świetle wojen handlowych, np. niedawnego konfliktu prezydenta Stanów Zjednoczonych Donalda Trumpa z Chinami, który poskutkował zakazem udostępniania technologii Google (czyli m.in. systemów Android) firmie Huawei. Biorąc pod uwagę 2,5 miliarda urządzeń tej marki na całym świecie, globalne skutki tej decyzji mogą być znaczące. W przypadku aplikacji natywnych już niedługo użytkownicy korzystający z tych urządzeń mogą mieć

problem z aktualizacjami swoich ulubionych aplikacji czy pobieraniem nowych, tu alternatywą okazać mogą się PWA, do instalacji których nie jest potrzebny sklep i które działają niezależnie od platformy [4].

PWA wciąż zyskuje na popularności. Taką wersję aplikacji mają już największe firmy, takie jak Google Maps, Tinder, Facebook, Alibaba itd., a do 2020 roku PWA mają zastąpić aż 50% aplikacji [11].

5.4. Zalety progresywnych aplikacji webowych

Progresywne aplikacje webowe łączą w sobie zalety zarówno aplikacji natywnych jak i webowych. Głównym założeniem tego podejścia do tworzenia aplikacji jest fakt, że jego elementy możemy zaimplementować praktycznie w każdej aplikacji webowej, a nawet stronie internetowe, dzięki czemu w przypadku niskiego budżetu lub braku czasu nie trzeba budować aplikacji od nowa a koszty projektu zostają zminimalizowane. Użytkownik nie musi także aktualizować aplikacji – zawsze otwiera najnowszą wersję [5].

Tak jak zwykłe aplikacje webowe działają w zwykłej przeglądarce internetowej, z tym, że dzięki zaawansowanemu wykorzystaniu pamięci podręcznej powinny działać znacznie szybciej i oferować dostępność przynajmniej części swoich funkcji bez konieczności podłączenia do Internetu. Dobrze zbudowana aplikacja PWA ładuje się natychmiastowo (nawet przy pierwszym uruchomieniu) oraz szybko reaguje na dane wejściowe użytkownika, dzięki czemu użytkownik nie zauważa przejść między stronami. Wraz z intuicyjnym designem jest to najważniejszy atut istotny przede wszystkim w branży e-commerce, gdzie nawet pół sekundy może zaważyć o wysokości tzw. pre-bounce rate, czyli odsetku klientów opuszczających stronę przed jej załadowaniem, co bardzo mocno wpływa na konwersję i realny zysk ze sprzedaży. Dzięki temu, że aplikacja działa w przeglądarce, użytkownik może zapisywać i udostępniać linki do konkretnej podstrony. Bardzo ważny w tej branży jest także fakt, że aplikacje PWA są indeksowane przez Google i prezentowane w wynikach wyszukiwania [22].

Progresywna aplikacja webowa, w przeciwieństwie do aplikacji natywnych, nie wymaga instalacji, co sprawia, że użytkownicy mogą chętniej z niej korzystać. Przykładem może być aplikacja firmy ubezpieczeniowej, która oferuje funkcje takie jak wycenę polisy i likwidacji szkód oraz obsługę wniosków – użytkownik będzie szukał takich narzędzi przede wszystkim w wyszukiwarce, na stronie ubezpieczyciela, a ponieważ najpewniej planuje skorzystać z nich raz lub sporadycznie, nie będzie zbyt chętny do instalacji jej [1]. Z pewnością jednak istotne będzie dla niego szybkie i płynne działanie takiego kalkulatora, intuicyjny wygląd na urządzeniu mobilnym, oraz działanie w miejscach

z ograniczonym lub słabym dostępem do Internetu (np. w miejscu zdarzenia drogowego odległego od miasta).

PWA oferuje także kilka zalet aplikacji natywnych. Główną z nich jest przede wszystkim dostęp do natywnych funkcji urządzenia. Jest to możliwe dzięki dostępnym Web APIs (Tabela 1). Choć nie wszystkie funkcje są jeszcze w pełni obsługiwane, a część z nich z pewnością może działać płynniej w aplikacjach natywnych, w większości przypadków wystarczają, aby zbudować zaawansowaną aplikację. Najczęściej wykorzystywane funkcje to te związane z działaniem aplikacji offline, czy też synchronizacją danych w tle.

Tabela 1. Przegląd dostępnych dla przeglądarek funkcji natywnych przez API. Źródło: [20][21]

Kategoria	**Funkcja**	**API / technologia**	**Wsparcie (mobile)**	**Wsparcie (desktop)**
Kamera i mikrofon	Przechwytywanie obrazu wideo i audio	getUserMedia/Stream API	93,88	91.04
	Zaawansowane elementy kamery	ImageCapture API	-	-
	Nagrywanie audio i wideo	MediaRecorder API	67,91	80,24
	Komunikacja w czasie rzeczywistym	RTCPeerConnection API	88,52	90,77
Otoczenie	Bluetooth	Web Bluetooth API	67,1	68,32
	USB	WebUSB API	61,28	67,75
	NFC	Web NFC API	0[5]	0
	Czujnik oświetlenia otoczenia	Ambient Light API	1,11	6,16
Urządzenie	Typ i prędkość połączenia z siecią	Network Information API	70,03	67,75
	Status połączenia z siecią	Online / offline status	97,22	99,77
	Wibracje	Vibration API	76,14	81,3
	Stan baterii	Battery Status API	75,41	71,33
	Pamięć urządzenia (całkowita pamięć RAM)	Device Memory API	-	-
Funkcje natywne	Powiadomienia	Notifications API	63,91	91,67
	Wiadomości push	Push API	75,67	84,3
	Instalacja na ekranie głównym	Web Manifest	93,34	68,52
	Widoczność aplikacji na wierzchu	Page Visibility API	96,53	96,75
	Uprawnienia	Permissions API	69,32	80,22
System operacyjny	Pamięć podręczna / działanie offline	Web Storage API	97,29	99,93
		IndexedDB	96,51	96,83
		Cache API	97,29	99,12
		Storage API	97,29	99,93

[5] Dostępne tylko w Chrome dla Androida po włączeniu w zakładce chrome://flags

	Dostęp do plików	File API	97,18	99,06
	Kontakty	Contacts API	0[6]	0
	SMS	Messaging API	0	0
	Dostępna pamięć	Quota Estimation API	-	-
	Zarządzanie alarmami / przypomnieniami	Task Scheduler API	0	0
Dane wejściowe	Sterowanie gestami	Touch Events API	96,54	80,95
		Pointer Events API	67,29	86,91
	Sterowanie głosem	Speech Recognition API	68,47	68,9
	Schowek (kopiowanie i wklejanie)	Synchronous Clipboard API	90,31	99,57
Płynne działanie	Działanie offline	Service Worker	93,34	89,44
	Synchronizacja danych w tle	Background Sync API	73,53	69,83
	Udostępnianie między aplikacjami	Web Share API	75,31	2,91
	Płatności	Payment Request API	85,58	85,05
	Dane logowania	Credential Management API	73,03	68,66
Lokalizacja i położenie urządzenia	Geolokalizacja	Generation API	97,29	99,58
	Geofencing	Geofencing API	0	0
	Pozycja urządzenia	DeviceOrientation API	97,2	91,18
		Orientation Sensor	61,28	65,73
	Ruch urządzenia	Accelerometer API	61,28	65,73
		Gyroscope API	61,28	65,73
		Magnetometer API	0	65,73
	Czujnik zbliżeniowy	Proximity API	1,11	10,56
Ekran i prezentacja	Wirtualna / rozszerzona rzeczywistość	WebXR API	68,21[7]	13,61
	Pełny ekran	Fullscreen API	93,21	98,66
	Orientacja ekranu i blokada ekranu	Screen Orientation API	74,54	90,26
	Wake lock[8]	Wake Lock API	-	-
	Funkcje prezentacji	Presentation API	33,72	32,62

Poza zaletami związanymi z użytkowaniem aplikacji progresywnych istotne są także funkcjonalności, które skutecznie przywracają klientów. Pierwsza z nich to możliwość wysyłania powiadomień push, nawet gdy przeglądarka jest zamknięta, druga natomiast to możliwość dodawania aplikacji do ekranu głównego.

[6] Dostępne tylko w Firefox Mobile 18 and Firefox OS 1.1 dla uprzywilejowanych aplikacji, nie jest standardem W3C

[7] Dostępne w Chrome po włączeniu w zakładce chrome://flags

[8] działanie w tle pomimo zablokowanego urządzenia / włączonego trybu oszczędzania baterii

5.5. Wady progresywnych aplikacji webowych

Choć progresywne aplikacje webowe mają bardzo wiele zalet, wiążą się także z pewnymi wyzwaniami i niedogodnościami.

Pierwsza wada to fakt, że wciąż nie wszystkie natywne funkcje dostępne są z poziomu przeglądarki i możliwe, że jeszcze przez dłuższy czas to się nie zmieni. Nie obsługują np. logowania odciskiem palca ani nie wspiera w pełni NFC, przez co nie znajdą zastosowania w bankowych aplikacjach transakcyjnych [1]. Nie ma dostępu także do książki kontaktowej czy też geofencing, a także do czujników światła lub zbliżeniowego.

Dodatkowo, aplikacje progresywne mimo szybkiego działania wciąż mogą wypadać słabiej w porównaniu do aplikacji natywnych, jeśli chodzi o wydajność, obciążenie procesora i płynność działania takich elementów jak mapy czy obsługa kamery. Może to utrudniać budowanie zaawansowanych aplikacji takich jak gry czy też aplikacji potrzebujących dużej mocy obliczeniowej.

6. Przykładowa aplikacja

Aby dokonać porównania opisywanych w pracy podejść zbudowano trzy aplikacje. Pierwsza z nich to aplikacja natywna napisana w języku Java dla platformy Android. Druga to aplikacja hybrydowa stworzona przy użyciu frameworka React Native. Trzecia aplikacja to progresywna aplikacja webowa napisana przy użyciu technologii webowych i biblioteki React.js. Wszystkie trzy to aplikacje typu social media, przypominające popularną aplikację Instagram, a każda z nich miała takie same założenia funkcjonalne:

1. Możliwość **rejestracji / zalogowania się** – dane użytkowników zapisywane powinny być w Firebase i zapisywane w pamięci tak, aby użytkownik nie musiał za każdym razem logować się ponownie,
2. Możliwość **robienia zdjęć** – aplikacja powinna posiadać obsługę kamery urządzenia i przesyłać zdjęcia do Firebase,
3. Możliwość korzystania z **geolokalizacji** – aplikacja powinna być w stanie automatycznie pobrać nazwę miasta, w którym znajduje się użytkownik,
4. Możliwość **dodawania postów** – posty powinny być zapisywane w Firebase, każdy post powinien mieć zdjęcie, nazwę miasta (podpowiadaną z geolokalizacji), krótki opis i informacje o autorze, a także informacje o polubieniach,
5. Wyświetlanie **listy wszystkich postów** na stronie głównej,
6. Wyświetlanie **listy postów danego użytkownika i jego polubionych postów** po wejściu na swoje konto,
7. Możliwość **polubienia zdjęcia** przez innych użytkowników – informacje o osobach lubiących dane zdjęcie także powinny być przechowywane w Firebase, a ilość polubień powinna pokazywać się pod każdym zdjęciem,
8. **Powiadomienia push** – możliwość włączenia / wyłączenia ich w widoku konta użytkownika, jeśli opcja ta jest włączona, użytkownik powinien otrzymać powiadomienie push za każdym razem, gdy zostanie dodany nowy post,
9. **Działanie offline** – aplikacja powinna być w stanie otworzyć się i wyświetlać wszystkie zapisane wcześniej zdjęcia nawet bez dostępu do Internetu, powinny działać także wszystkie podstrony aplikacji,

10. Możliwość **dodania do ekranu głównego / instalacji**,
11. Wygląd aplikacji zbliżony do wytycznych **Material Design**.

6.1. Wybrane technologie

Przykładowa aplikacja z założenia miała być aplikacją typowo użytkową, do której w rzeczywistości dana firma mogłaby rozważać użycie każdego opisywanego w pracy podejścia. Wybrano jedne z popularniejszych technologii na podstawie badań rynku oraz ankiet przeprowadzanych wśród programistów przez m.in. serwis StackOverflow [27][52].

Jako back-end wykorzystano usługę **Firebase** utrzymywane przez Google. Oferuje ono m.in. bazę danych, przechowywanie zdjęć, możliwość pisania funkcji w Node.js i wywoływania ich automatycznie podczas zmian w bazie, wysyłanie powiadomień oraz autoryzację użytkowników. Został wybrany ze względu na popularność, stabilność oraz łatwą integrację ze wszystkimi rodzajami aplikacji opisywanymi w pracy bez konieczności posługiwania się innymi językami programowania niż JavaScript (Node.js) oraz Java (w przypadku aplikacji natywnej).

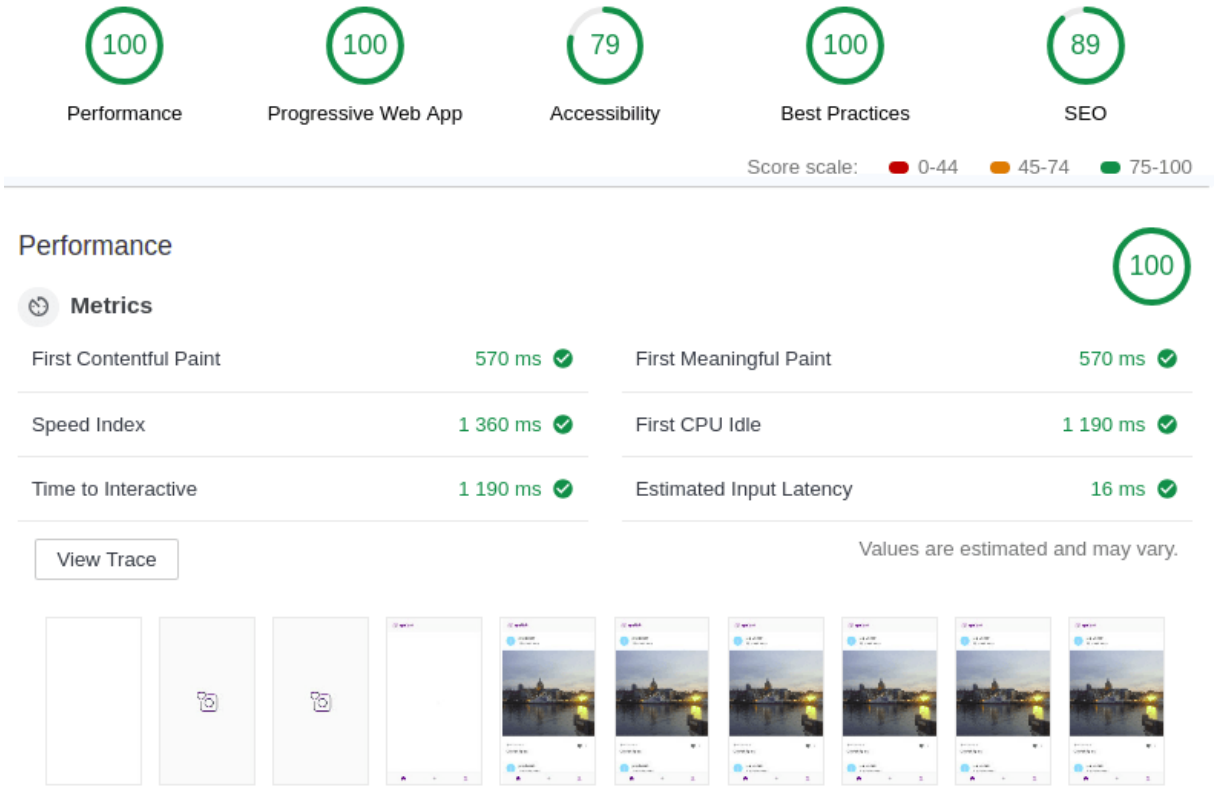
Aplikacja natywna została przygotowana na platformę **Android** ze względu na największą popularność (76% rynku [30]), oraz stworzona w dedykowanym dla Androida IDE – **Android Studio**. Poza wtyczką do doładowywania zdjęć oraz Firebase nie było konieczności instalacji dodatkowych modułów, ponieważ prawie wszystkie potrzebne funkcjonalności można było zaimplementować korzystając z wbudowanych funkcji i motywów.

Aplikacja hybrydowa została napisana w technologiach webowych (HTML / CSS / JavaScript) z wykorzystaniem frameworka **React Native**, ponieważ zajmuje on 2 miejsce na liście najpopularniejszych frameworków/ bibliotek webowych [27] oraz ze względu na możliwość reużywalności kodu pomiędzy aplikacją hybrydową i webową. Choć jest to wciąż rozwijająca się technologia, jest ona bardzo popularna wśród osób znających już React.js, cechująca się świetną wydajnością. Dużym ułatwieniem jest możliwość Live Update – natychmiastowego podglądania zmian w kodzie na urządzeniu lub emulatorze [27]. Przy budowaniu aplikacji hybrydowej wykorzystano także bibliotekę **Redux** (zajmującą 1 miejsce na liście najpopularniejszych technologii do zarządzania danymi i stanem aplikacji webowej [27]). Technologia ta okazała się konieczna przy React Native do utrzymania przejrzystego kodu oraz kontrolowania stanu aplikacji i każdego jej komponentu w jednym miejscu. Jest odpowiedzialna także za działanie offline aplikacji dzięki rozszerzeniu **redux-persist** [41]. Konieczne okazało się instalowanie wielu dodatkowych modułów,

m.in. obsługę kamery i wyboru zdjęć z galerii, nawigację i ikony. Aplikację napisano w Visual Studio Code.

Aplikacja webowa (PWA) została stworzona wykorzystując technologie webowe (HTML / CSS / JavaScript) oraz bibliotekę **React.js**, zajmującą 1 miejsce na liście najpopularniejszych frameworków webowych. Jest to technologia najczęściej wybierana i polecana przy tworzeniu aplikacji webowych. Ma dobrze opisaną dokumentację, jest wspierana przez Facebooka, oferuje świetną wydajność oraz łatwość debugowania aplikacji [27]. Do stylowania aplikacji wykorzystano bibliotekę **Material-UI**. Umożliwia ona tworzenie interfejsów zgodnych z Material Design, czyli wytycznymi określającymi wygląd większości aplikacji na Androida. Stylowanie komponentów opiera się na rozwiązaniu CSS-in-JS, jednym z najnowszych w zakresie CSS[9]. Rozwiązuje ono m.in. problem nadpisywania styli w kilku miejscach aplikacji dla np. takiej samej klasy dzięki haszowaniu nazw klas i kompilowaniu razem z kodem JavaScript. Umożliwia też stylowanie warunkowe oraz korzystanie ze zmiennych i obliczeń [28]. Ten rodzaj aplikacji wymagał zainstalowania największej ilości modułów. Do cache'owania aplikacji według określonych strategii i tym samym jej działania offline wykorzystano polecaną przez Google bibliotekę **Workbox** [29] Dodatkowo, do przechowywania danych w przeglądarce i automatycznego ich odświeżania wykorzystano API **IndexedDB**. Korzysta ono z indeksów i umożliwia szybkie przeszukiwanie dużych ilości informacji [34] Aby móc nasłuchiwać wiadomości push oraz wyświetlać powiadomienia aplikacja korzysta z **Web Push** API, które działa razem z Service Worker'em [36]. Do nawigacji pomiędzy widokami i zapewnienia każdemu z nich osobnego linku zainstalowano bibliotekę **React Router** [37] Pobieranie nazwy miasta na podstawie lokalizacji zaciąganej z przeglądarki było możliwe dzięki modułowi **React Geocode** [38]. Aby zwiększyć szybkość ładowania aplikacji zaimplementowano także **React Lazyload** - bibliotekę zapewniającą doładowywanie komponentów i zdjęć na bieżąco [39]. Aplikacja została napisana w Visual Studio Code i opublikowana korzystając z Github Pages co zapewniło serwowanie jej przez HTTPS [40]. Dzięki spełnieniu wszystkich wymogów związanych ze standardami PWA budowana aplikacja otrzymała maksymalną ilość punktów w tej kategorii w wykonanym audycie Lighthouse (Rysunek 1).

[9] Cascading Style Sheets – język używany do opisu wyglądu wyświetlanego kodu HTML.

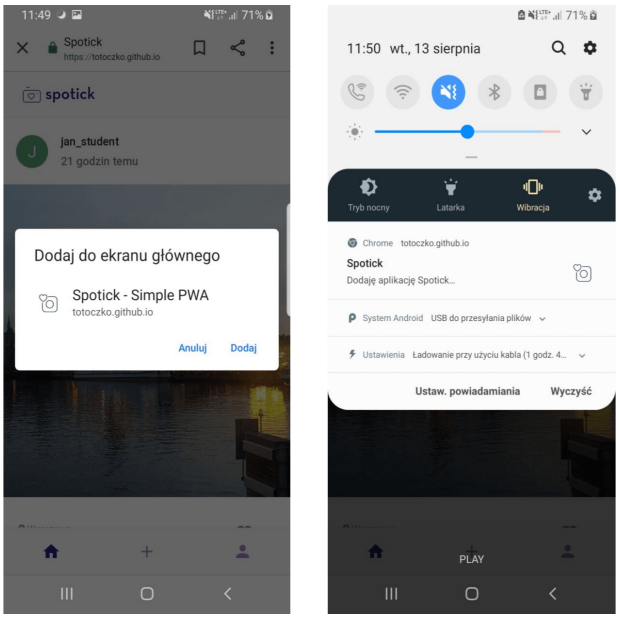


Rysunek 1. Wyniki wykonanego audytu Lighthouse dla aplikacji webowej. Źródło: opracowanie własne.

6.2. Instalacja[10]

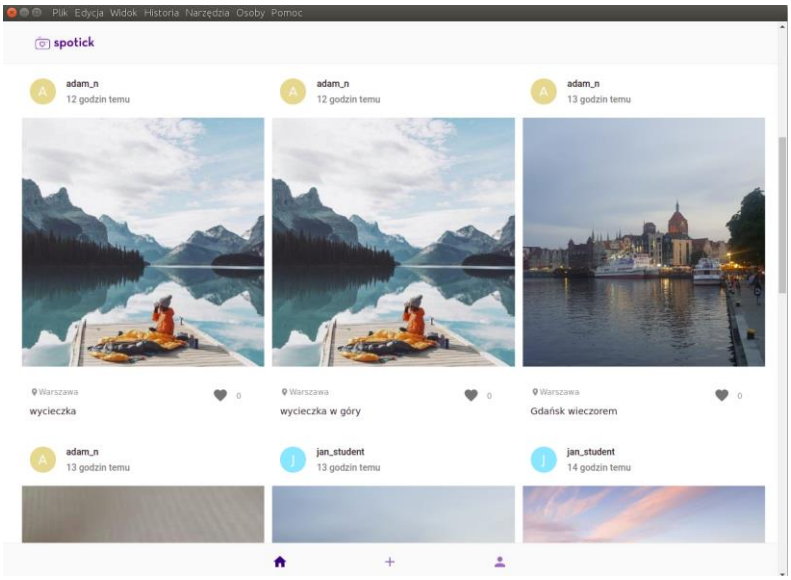
Zarówno aplikacja natywna jak i hybrydowa powinny być zainstalowane przez sklep Google Play, jednak wymagałoby to wcześniejszej autoryzacji aplikacji, dlatego też na potrzeby pracy zostały one zainstalowane ręcznie przez Android Studio / terminal. W przypadku aplikacji webowej przeglądarka umożliwiła instalowanie aplikacji po dodaniu Web App Manifest'u oraz spełnieniu podstawowych wymagań PWA opisywanych w poprzednim rozdziale (Rysunek 2).

[10] Wszystkie aplikacje testowane były na urządzeniu Samsung Galaxy S8 z systemem Android 9.



Rysunek 2. Instalacja aplikacji PWA. Źródło: opracowanie własne.

Instalacja powiodła się nie tylko na urządzeniu mobilnym, ale także na laptopie (system Ubuntu 16.04 – Linux) – w tym przypadku na pulpicie także pojawiła się ikona, a sama aplikacja uruchamia się poza standardowym oknem przeglądarki (Rysunek 3).

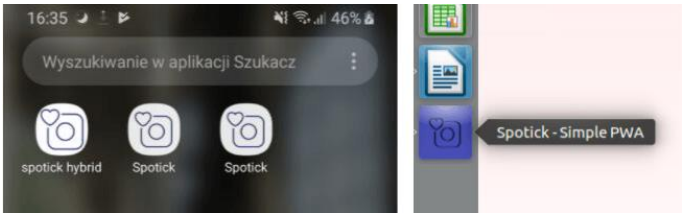


Rysunek 3. Zainstalowana aplikacja PWA na systemie Ubuntu. Źródło: opracowanie własne.

Aplikacja webowa podczas uruchomienia pokazuje tzw. splash screen, czyli ekran z ikoną aplikacji, co jest wbudowaną cechą PWA i zapewnia lepsze doświadczenia użytkownika. Nie jest

widoczny pasek przeglądarki, natomiast w trakcie korzystania z aplikacji w pasku powiadomień przeglądarka wyświetla komunikat z możliwością skopiowania lub udostępnienia jej adresu.

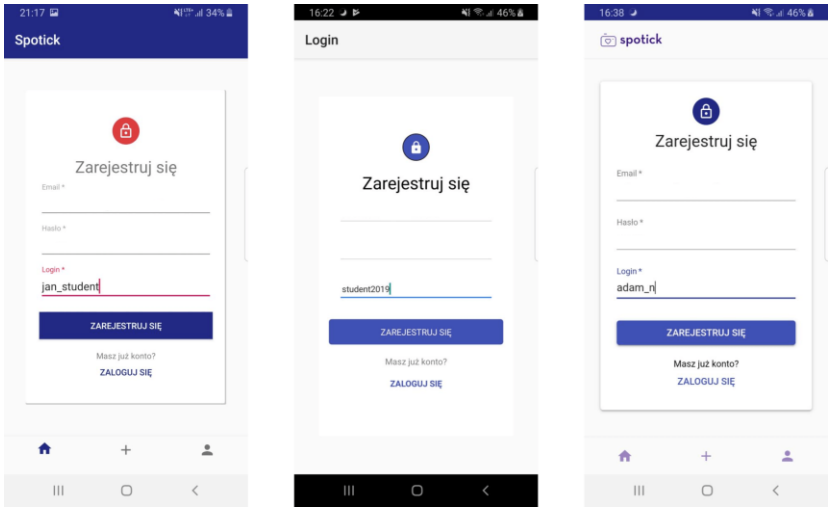
Ikony dodane do ekranu głównego dla każdej aplikacji wyglądają podobnie, przy czym ikona aplikacji webowej nie ma nakładki przeglądarki jak w przypadku skrótów (Rysunek 4).



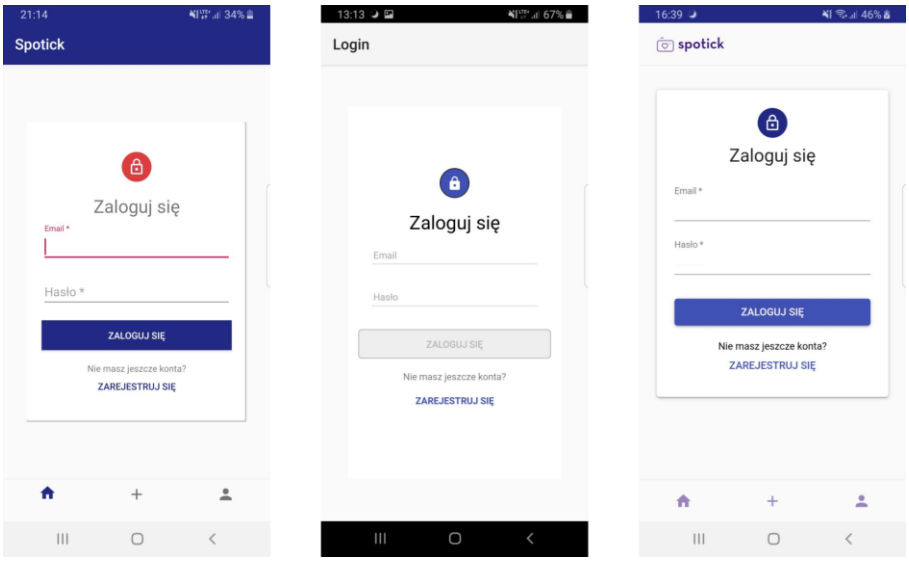
Rysunek 4. Ikony aplikacji: hybrydowej, PWA, natywnej, PWA na systemie Ubuntu (od lewej). Źródło: opracowanie własne.

6.3. Rejestracja i logowanie się

Rejestracja i logowanie użytkowników zostały zaimplementowane korzystając z Firebase Authentication. We wszystkich trzech przypadkach integracja z Firebase nie stanowiła problemu dzięki dobrej dokumentacji. Zastosowano metodę logowania przy użyciu adresu e-mail oraz hasła. Dodatkowo w momencie rejestracji użytkownicy zapisywani są w bazie danych, aby móc pobrać kolor ich profilu oraz polubione zdjęcia.

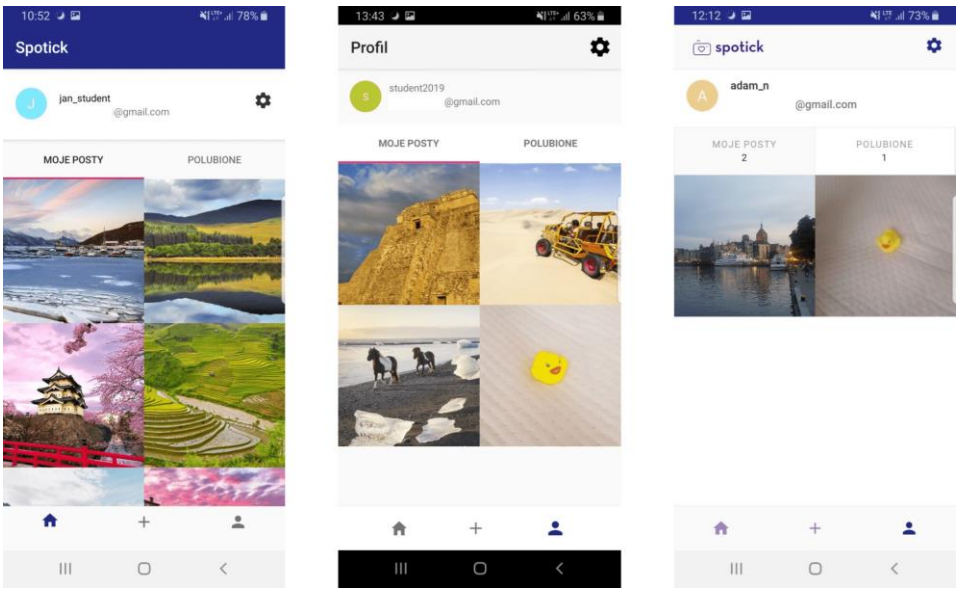


Rysunek 5. Widok rejestracji w aplikacjach: natywnej, hybrydowej i PWA (od lewej). Źródło: opracowanie własne.

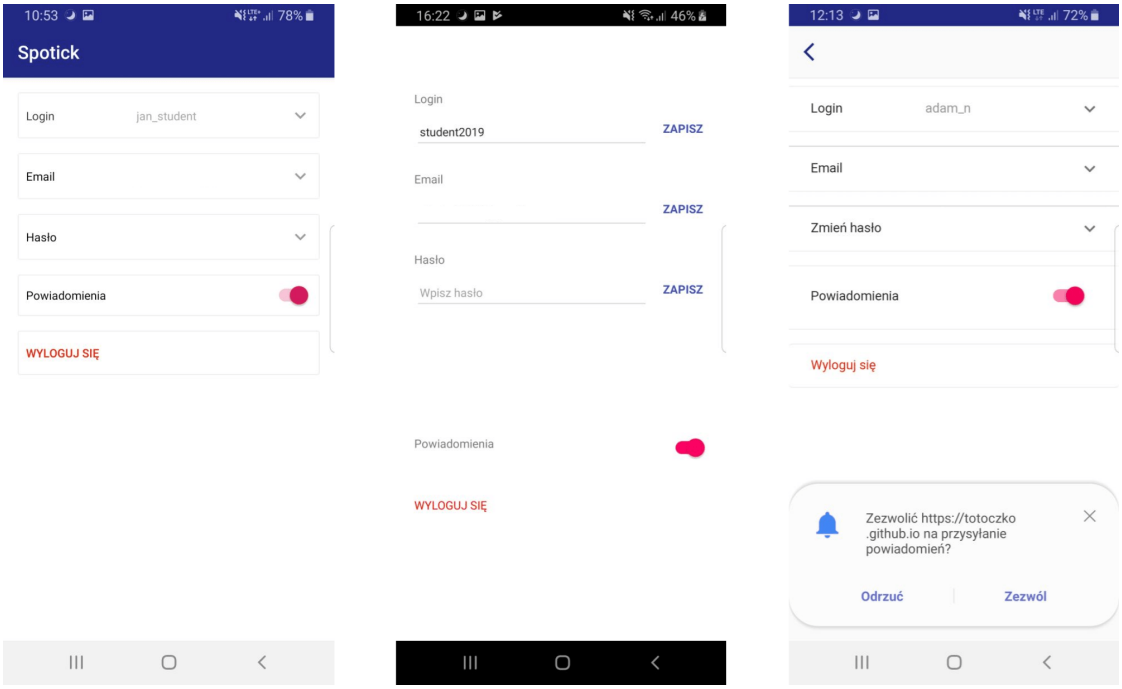


Rysunek 6. Widok logowania w aplikacjach: natywnej, hybrydowej i PWA (od lewej). Źródło: opracowanie własne.

Użytkownik ma możliwość wejścia na swój profil (Rysunek 7), gdzie może zobaczyć dodane przez siebie posty, zobaczyć polubione posty oraz ma możliwość zmiany swoich danych (nazwa użytkownika, email, hasło), włączyć / wyłączyć powiadomienia oraz wylogować się (Rysunek 8).



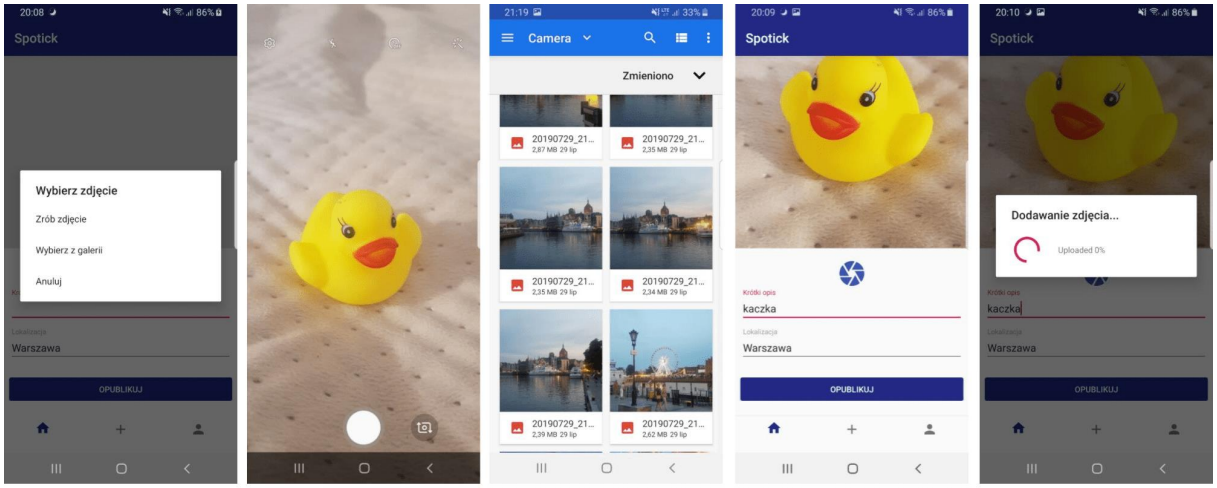
Rysunek 7. Widok profilu użytkownika w aplikacjach: natywnej, hybrydowej i PWA (od lewej). Źródło: opracowanie własne.



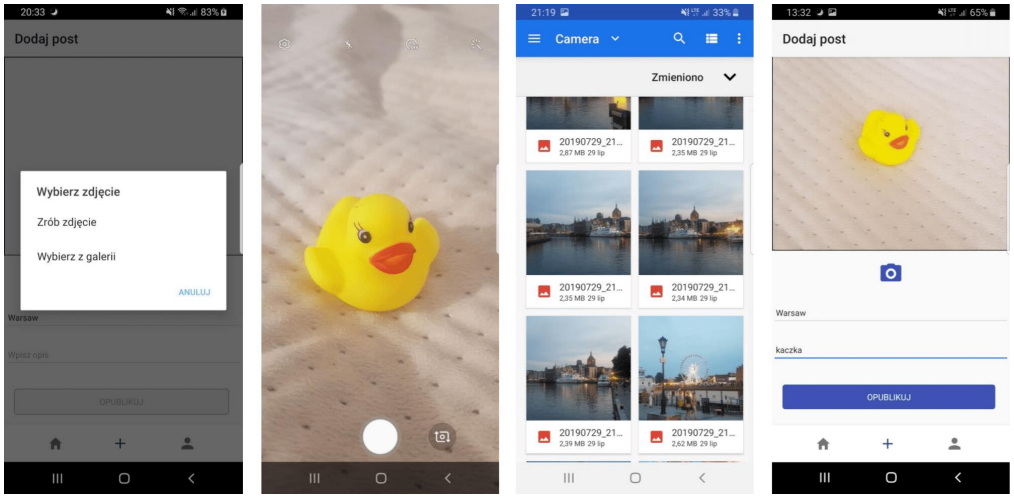
Rysunek 8. Widok ustawień użytkownika w aplikacjach: natywnej, hybrydowej i PWA (od lewej). Źródło: opracowanie własne.

6.4. Dodawanie postów

Dodawanie postów składa się z kilku części: możliwości zrobienia zdjęcia lub wybrania go z galerii urządzenia, pobranie nazwy miasta, w której znajduje się użytkownik oraz wysłanie danych do Firebase. Obsługa kamery i zdjęć okazała się najprostsza w przypadku aplikacji hybrydowej, dzięki zainstalowanemu pakietowi ImagePicker [42]. W przypadku aplikacji natywnej otrzymanie podobnego efektu wymagało napisania znacznie dłuższego kodu, z własnym rozwiązaniem problemów takich jak obsługa pozwoleń, generowanie URI zdjęcia, dekodowanie go z bitmapy, zapisywanie go w odpowiedniej rozdzielczości, przekręcenie go itp.

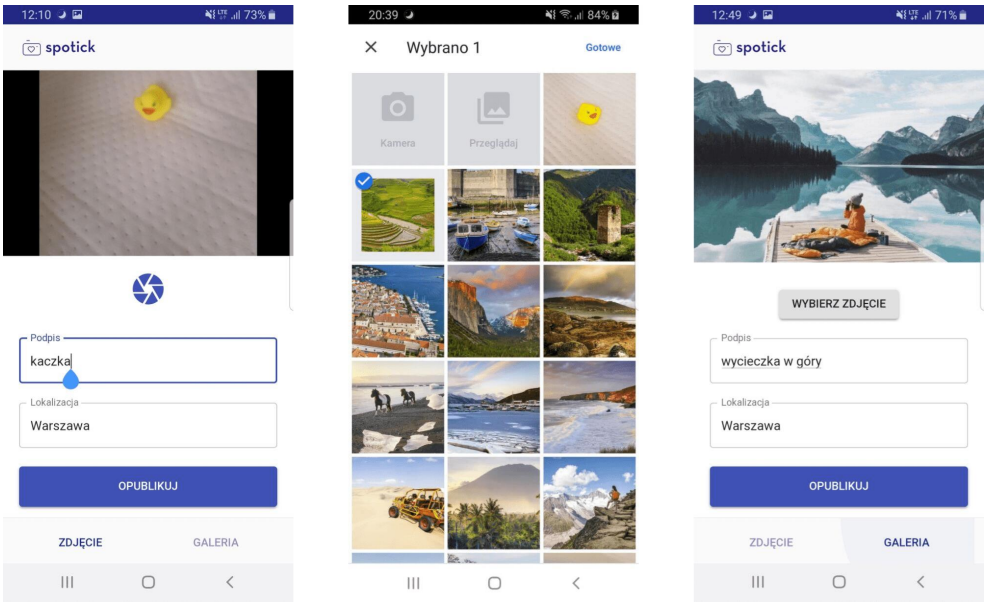


Rysunek 9. Proces dodawania wpisu w aplikacji natywnej. Źródło: opracowanie własne.



Rysunek 10. Proces dodawania wpisu w aplikacji hybrydowej. Źródło: opracowanie własne.

Obsługa kamery i zdjęć najbardziej kłopotliwa okazała się w przypadku aplikacji webowej. Sam dostęp do aparatu był możliwy dzięki funkcji getUserMedia, która automatycznie obsługuje pozwolenia na korzystanie z np. kamery. Zwraca ona strumień wideo, dzięki czemu możliwe było dodanie wewnątrz aplikacji podglądu z kamery (bez przejścia do pełnoekranowej wersji aparatu) oraz na kliknięcie ikony aparatu zatrzymanie go i zamianę na zdjęcie, które następnie zostaje przesłane. Niestety jakość wykonanych w ten sposób zdjęć jest widocznie słabsza, a obsługa elementu wyświetlającego podgląd wideo i zapisane zdjęcie znacznie cięższa niż w przypadku aplikacji natywnej i hybrydowej.



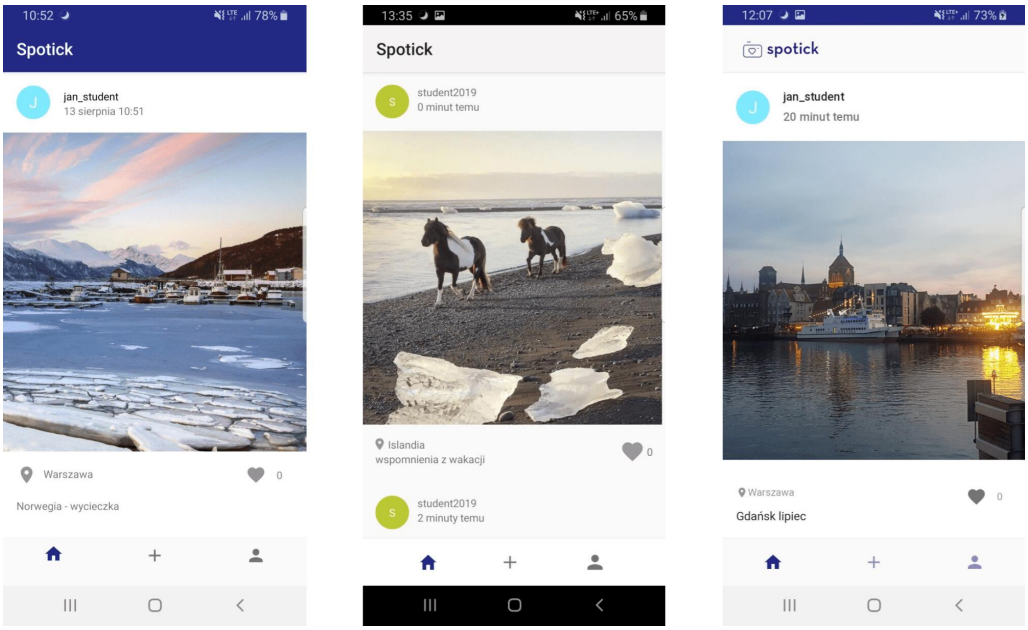
Rysunek 11. Proces dodawania wpisu w aplikacji PWA. Źródło: opracowanie własne.

6.5. Wyświetlanie postów

W aplikacji natywnej możliwe było wyświetlanie postów w kontenerze RecyclerView, który zapewnia dobrą wydajność i mniejsze zużycie pamięci dzięki temu, że nie tylko doładowuje posty na bieżąco wraz z przewijaniem strony, ale wykorzystuje do tego elementy listy z góry, które nie są już widoczne dla użytkownika podmieniając ich dane na nowe [43]. Niestety powoduje to nieprzyjemne skoki w aplikacji w momencie, gdy użytkownik chce jak najszybciej przewinąć na górę strony.

React Native oferuje interfejs FlatList, który działa na podobnej zasadzie jak RecyclerView, jednak znacznie lepiej radzi sobie z szybkim przewijaniem strony, ma także wbudowaną obsługę ‘pull to refresh’ - pociągnięcia ekranu, aby odświeżyć dane, do czego przyzwyczajeni są użytkownicy.

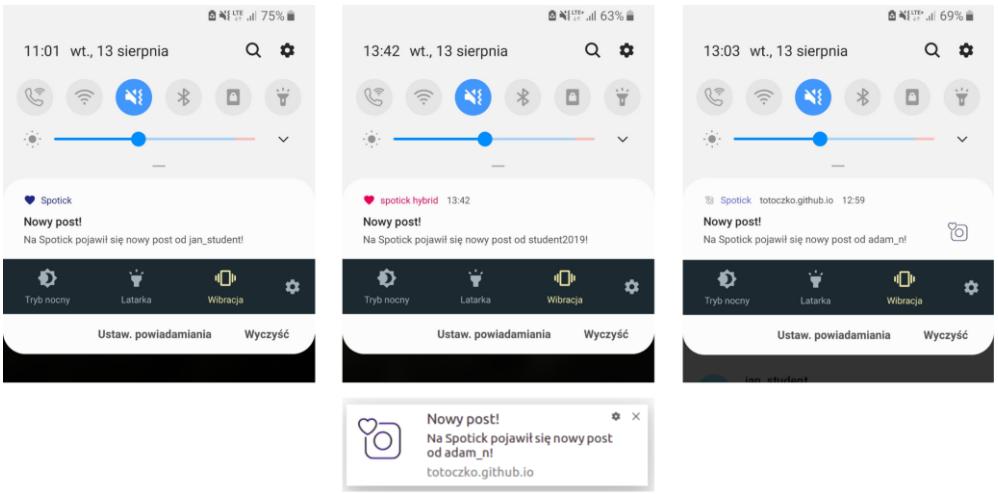
W przypadku aplikacji webowej wykorzystano zwykłą listę oraz zainstalowano moduł zapewniający doładowywanie postów na bieżąco, dzięki czemu posty i zdjęcia ładują się bardzo szybko, a przy przewijaniu w górę nie muszą się na nowo doładowywać. Fakt, że aplikacja wyświetla się wciąż w przeglądarce (nawet pomimo instalacji) zapewnia także wbudowaną obsługę odświeżania przy pociągnięciu ekranu w dół, dokładnie tak, jak w przypadku stron internetowych.



Rysunek 12. Widok ekranu głównego w aplikacjach: natywnej, hybrydowej i PWA (od lewej). Źródło: opracowanie własne.

6.6. Powiadomienia Push

Aby umożliwić wysyłanie i odbieranie powiadomień konieczne było napisanie funkcji w Node.js, która wykonuje się po dodaniu nowego posta do bazy danych i opublikowanie jej na serwerze Firebase.



Rysunek 13. Powiadomienia push w aplikacjach mobilnych: natywnej, hybrydowej, PWA (od lewej), oraz na systemie Ubuntu (na dole). Źródło: opracowanie własne.

W przypadku aplikacji natywnej wysyłanie powiadomień odbywa się na zasadzie zapisywania w bazie tokenów urządzeń, które mają włączone otrzymywanie powiadomień w zainstalowanej aplikacji a następnie wysyłanie do każdego z urządzeń wiadomości i nasłuchiwanie ich poprzez FirebaseMessagingService.

W aplikacji hybrydowej zastosowano rozwiązanie subskrypcji do danego tematu wiadomości, wiązało się to jednak z koniecznością napisania własnej obsługi zapisywania i uaktualniania tokenów.

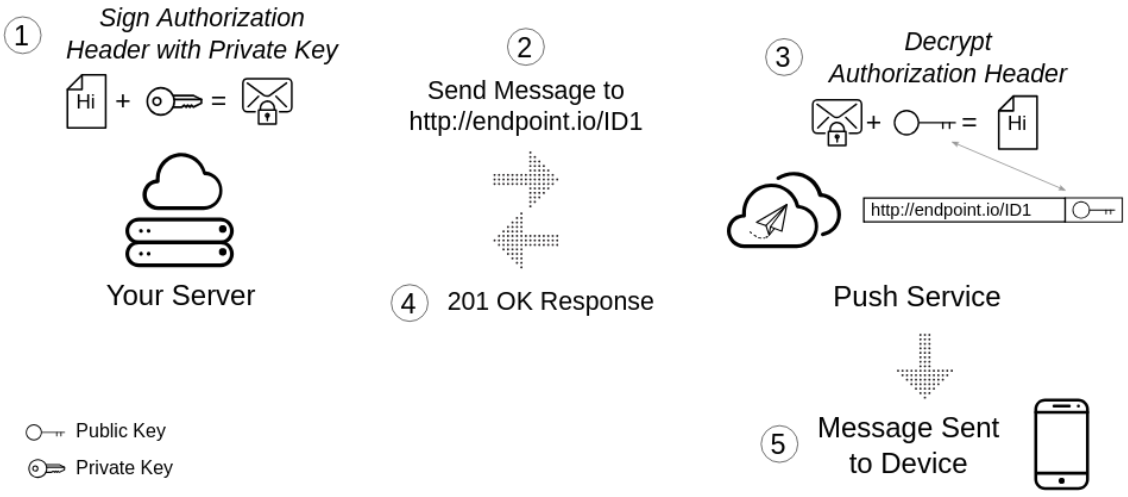
Aplikacja webowa korzysta natomiast z Web Push API. Proces uwierzytelnienia aplikacji i zapisywania subskrypcji jest tu bardziej skomplikowany jeśli chodzi o implementację – wysyłane dane w formacie JSON podpisywane są prywatnym kluczem serwera znajdującym się w nagłówku żądania POST (w tym przypadku wysyłanym z funkcji Firebase) i przekazywane do usługi push (Listing 1).

Listing 1. Wysyłanie powiadomień push przy użyciu Web Push API w funkcji Firebase.

```
exports.sendNotifications = functions.https.onRequest((request, response) => {
  cors(request, response, () => {
    webpush.setVapidDetails(
      'mailto:test@test.pl',
      'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx',
      'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx');
    const from = request.body.from;
    const subscriptions = request.body.subscriptions;

    subscriptions.forEach((sub) => {
      const pushConfig = {
        endpoint: sub.endpoint,
        keys: {
          auth: sub.keys.auth,
          p256dh: sub.keys.p256dh
        }
      };
      webpush.sendNotification(
        pushConfig,
        JSON.stringify({
          title: "Nowy post!",
          content: "Na Spotick pojawił się nowy post od " + from.name + "!",
          openUrl: "/"
        })
      ).catch((err) => {
        return response.status(500).json({ error: err });
      });
    });
    return response.status(200).json({ message: "Data stored" });
  }).catch((err) => {
    return response.status(500).json({ error: err });
  });
});
```

Następnie usługa push sprawdza czy odebrane informacje podpisane są kluczem prywatnym zgodnym z publicznym kluczem każdej zapisanej aplikacji. Jeżeli klucze zgadzają się, wysyłany jest komunikat push (Rysunek 14). Taka autoryzacja jest niezbędna, ponieważ sam endpoint do którego wysyłane są wiadomości byłby dość prosty do przejęcia przez osoby trzecie.



Rysunek 14. Działanie protokołu Web Push. Źródło: [44].

Nasłuchiwanie powiadomień push odbywa się w Service Worker, dlatego też uruchomienie aplikacji na tym samym urządzeniu, ale w różnych przeglądarkach będzie wymagało nowej subskrypcji (Listing 2). Samo wyświetlanie powiadomień jest możliwe dzięki Notification API, co działa zarówno na urządzeniach mobilnych jak i komputerach (Rysunek 13).

Listing 2. Zapisywanie subskrypcji przy użyciu Web Push API.

```
configurePushSubscription = () => {
  // check if this SW handled through given browser have suscription for this device:
  if (!('serviceWorker' in navigator)) {
    return;
  } else {
    let reg;
    navigator.serviceWorker.ready
      .then((swreg) => {
        reg = swreg;
        return swreg.pushManager.getSubscription();
      })
      .then((sub) => {
         if (sub === null) {
          const vapidPublicKey = 'xxxxxxxxxxxxxxxxxxxx';
          const convertedVapidPublicKey = urlBase64ToUint8Array(vapidPublicKey);
          reg.pushManager.subscribe({
            userVisibleOnly: true,
            applicationServerKey: convertedVapidPublicKey
        });
      }
      return sub;
    })
    .then((newSub) => {
       this.setState({
       subscription: newSub
      }, () => {
        this.handleUpdateFirebase() // save an endpoint, auth and p256dh key
      })
    return true
    })
    .catch((err) => {
       console.log(err)
    })
  }
}
askForNotificationPermission = () => {
  Notification.requestPermission((result) => {
    if (result === 'granted') {
      this.configurePushSubscription();
    }
  })
}
```

6.7. Działanie offline

W przypadku aplikacji natywnej zapisywanie danych na urządzeniu zostało umożliwione dzięki włączeniu opcji setPersistenceEnabled podczas tworzenia instancji bazy Firebase. Wszystkie zapisane dane dostępne są po zamknięciu i ponownym uruchomieniu aplikacji nawet przy późniejszym braku łączności z siecią.

Listing 3. Włączenie zapisywania danych z Firebase w aplikacji natywnej.

```
public class FirebaseInstance extends android.app.Application{
    @Override
    public void onCreate() {
        super.onCreate();
        FirebaseDatabase.getInstance().setPersistenceEnabled(true);
    }
}
```

Dość proste okazało się także dodanie działania offline dla aplikacji hybrydowej – dzięki zainstalowanemu pakietowi redux-persist w momencie zamknięcia aplikacji wszystkie dane aplikacji znajdujące się w Redux (posty, dane użytkownika) zapisywane są w AsyncStorage, czyli pamięci podręcznej [41].

Działanie offline aplikacji webowych jest bardziej skomplikowane, ponieważ wszystkie dane i pliki jakie potrzebne są do wyświetlania aplikacji muszą być przechowywane i automatycznie odświeżane w pamięci cache przeglądarki. Jest to jeden z głównych powodów, dla których instalowany jest Service Worker – to w nim zakodowane jest cache'owanie aplikacji korzystając z Cache API. Polega ono na zapisywaniu w przeglądarce konkretnych danych, jeśli choć raz udało się je pobrać przy połączeniu z siecią – następnym razem, gdy aplikacja wyśle żądanie pod ten sam adres dane nie będą musiały być pobierane na nowo. Dane z pamięci cache mogą być pobierane nie tylko z poziomu Service Worker'a, ale także zwykłego kodu JavaScript. Każdy plik (np. szkielet aplikacji) lub zestaw danych (np. posty) można zapisywać korzystając z innej strategii [3][45]:

- '**Cache with network fallback**' – Service Worker 'przechwytuje' wysłane przez aplikację żądanie do sieci i sprawdza, czy w pamięci cache nie znajdują się już dane z adresu nagłówka. Jeśli dane zostaną znalezione, zostają zwracane jako odpowiedź do aplikacji, natomiast jeśli nie – Service Worker przekazuje żądanie do sieci i stamtąd aplikacja uzyskuje odpowiedź. Daje to świetną wydajność aplikacji – nawet przy połączeniu z Internetem wszelkie dane pobierane są znacznie szybciej. Z drugiej strony nie jest to dobra strategia dla danych które często się zmieniają – aplikacja otrzymywałaby wciąż nieaktualne dane pobierane z cache zamiast z sieci.
- '**Cache only**' – Service Worker 'przechwytuje' wysłane przez aplikację żądanie do sieci i wysyła odpowiedź do aplikacji z danymi znalezionymi w pamięci cache – nie przekazuje dalej żądania, jeśli dane nie zostały wcześniej zapisane.
- '**Network with cache fallback**' - Service Worker 'przechwytuje' wysłane przez aplikację żądanie i przekazuje je do sieci. Tylko w momencie, gdy połączenie nie powiedzie się, Service Worker sprawdza, czy dane są dostępne w cache i przekazuje je do aplikacji. Dzięki takiemu

rozwiązaniu dane w aplikacji są zawsze aktualne, aplikacja może działać offline, jednak nie wykorzystywana jest możliwość przyspieszenia działania aplikacji. Ma to znaczenie szczególnie w momencie, gdy połączenie z siecią jest, ale słabe – odrzucenie żądania przez sieć nie dzieje się natychmiastowo, ale nawet po kilkunastu sekundach i dopiero po takim czasie pobierane byłyby dane z cache.

- '**Cache, then network**' - aplikacja bezpośrednio pobiera dane z cache (bez ingerencji Service Worker'a) i jednocześnie wysyła żądanie do sieci które jest przechwytywane przez Service Worker'a. Service Worker przekazuje żądanie do sieci i czeka na odpowiedź. Po otrzymaniu jej zapisuje lub uaktualnia otrzymane dane w cache, a następnie przekazuje dane do aplikacji. Pozwala to na szybkie odczytanie danych z pamięci przeglądarki i odświeżanie ich, jeśli pojawiła się w sieci nowa wersja.

Do zaimplementowania odpowiednich strategii cache'owania wykorzystano bibliotekę Workbox (Listing 4). W przypadku styli czcionek oraz plików statycznych, czyli szkieletu aplikacji, zastosowano strategię 'cache, then network', aby mieć pewność, że szybko się ładują i zawsze są aktualne dla każdego użytkownika. Do samych plików czcionek oraz zdjęć postów użyto strategii 'cache with network fallback', ponieważ pliki te nie zmieniają się w czasie, więc nie ma potrzeby ich częstego uaktualniania [29].

Listing 4. Przykład zapisywania plików w cache przy użyciu biblioteki Workbox.

```
// Cache static app files with a 'cache, then network' strategy.
workbox.routing.registerRoute( /\.(?:js|css|html)$/,
  new workbox.strategies.StaleWhileRevalidate({
    cacheName: 'static-resources',
  })
);

// Cache images with a 'cache with network fallback' strategy.
workbox.routing.registerRoute(
  /^https:\/\/firebasestorage\.googleapis\.com/,
  new workbox.strategies.CacheFirst({
    cacheName: 'post-images',
    plugins: [
      new workbox.expiration.Plugin({
        maxEntries: 60,
        maxAgeSeconds: 30 * 24 * 60 * 60, // 30 Days
      }),
    ]
  })
);
```

Cache'owanie w aplikacji podzielone jest na trzy części. Pierwsza część to początkowe cache'owanie plików aplikacji (szkieletu) w momencie uruchomienia jej. Druga to dynamiczne cache'owanie – za każdym razem, gdy aplikacja otrzyma pliki (np. czcionki, zdjęcia) zapisuje je w pamięci przeglądarki. Trzecia część to cache'owanie dynamicznych danych. Różni się ona od

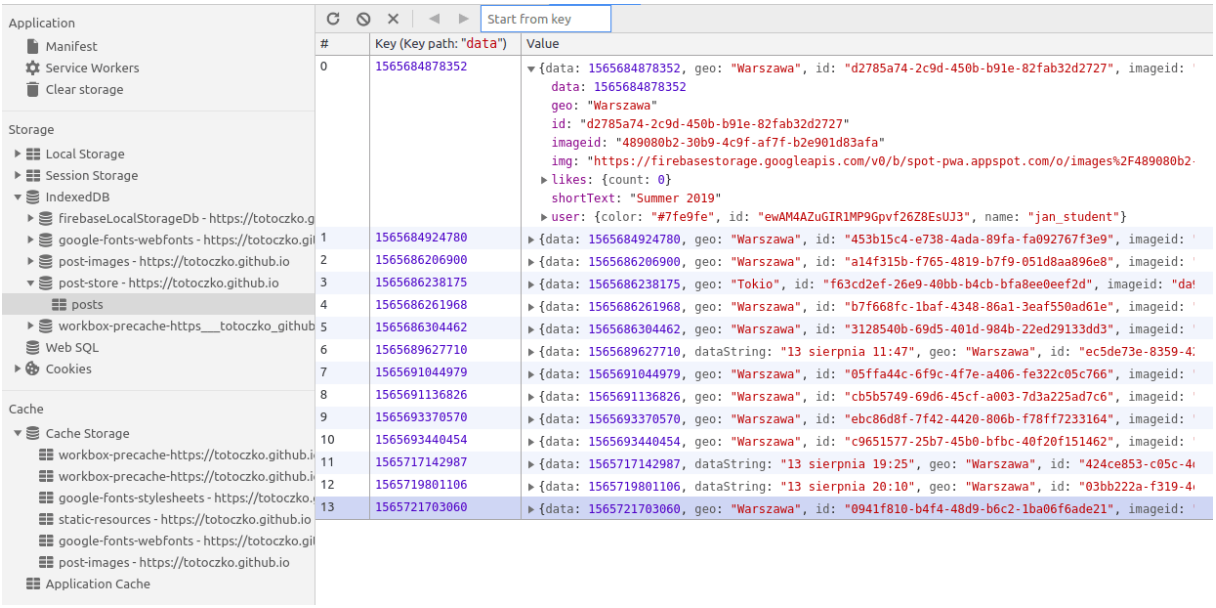
dynamicznego cache'owania formatem pobieranych i zapisywanych danych – w tym przypadku cache'owane dane w formacie JSON zamiast całych plików. Zapisywane są one nie w Cache, a w IndexedDB, czyli transakcyjnej bazie danych trzymanej w pamięci przeglądarki (Listing 5). Transakcyjność IndexedDB polega na tym, że jeśli choć jedna zmiana z wysłanych kilku żądań nie powiedzie się, wszystkie zmiany zostaną anulowane, co daje pewność, że nie zostanie zaktualizowana tylko część informacji.

Listing 5. Zapisywanie danych dynamicznych w IndexedDB.

```
// Create database for dynamic data in IndexedDB
let dbPromise = idb.open('post-store', 1, (db) => {
  if (!db.objectStoreNames.contains('posts')) {
    db.createObjectStore('posts', { keyPath: 'data' })
  }
});
// Save dynamic data in IndexedDB on fetch
self.addEventListener('fetch', (event) => {
  const url = 'https://spot-pwa.firebaseio.com/posts';
  if (event.request.url.indexOf(url) > -1) {
    event.respondWith(
      fetch(event.request)
        .then((res) => {
          const cloned = res.clone();
          if (event.request.url.indexOf('posts.json') > -1) {
            deletePostsFromIDB('posts', dbPromise).then(() => {
              return cloned.json()
            })
              .then((data) => {
                for (let key in data) {
                  savePostsIntoIDB('posts', data[key], dbPromise)
                }
              })
          }
          return res
        })
        .catch((err) => console.log(err))
    )
  }
})
const savePostsIntoIDB = (st, data, dbPromise) => {
  return dbPromise.then((db) => {
    let tx = db.transaction(st, 'readwrite');
    let store = tx.objectStore(st);
    store.put(data);
    return tx.complete;
  })
}
const deletePostsFromIDB = (st, dbPromise) => {
  return dbPromise.then((db) => {
    let tx = db.transaction(st, 'readwrite');
    let store = tx.objectStore(st);
    store.clear();
    return tx.complete;
  })
}
```

Dane w IndexedDB zapisywane są na zasadzie klucz-wartość (Rysunek 15), dzięki czemu przeszukiwanie danych i operacje przeprowadzane na niej odbywają się ze świetną wydajnością nawet przy dużych ilościach informacji. Możliwe jest także automatyczne zmienianie formatu pobieranych

z sieci danych i zapisywanie ich w formie spełniającej potrzeby danej aplikacji. Dane z IndexedDB pobierane są asynchronicznie, co sprawia, że w przeciwieństwie do synchronicznych Local i Session Storage[11] przeglądarki dobrze nadają się do pracy z Service Worker'em, który działa na zasadzie nasłuchiwania zdarzeń [3].



Rysunek 15. IndexedDB aplikacji webowej i zapisane w niej posty. Źródło: opracowanie własne.

[11] Obiekty Web Storage, czyli magazynu danych w przeglądarce przechowywanych przez określony czas, działające na podobnej zasadzie do ciasteczek (cookies) przeglądarki.

7. Porównanie aplikacji natywnej, hybrydowej i PWA

Zbudowane i opisane w poprzednim rozdziale aplikacje różnią się pod wieloma względami – trudnościami przy implementacji konkretnej funkcjonalności, poświęconą im ilością czasu ich budowania, wykorzystywanymi narzędziami, dostępnym wsparciem twórców danej technologii i społeczności, a także w pewnym stopniu wyglądem, wydajnością i płynnością działania. Każde z rozwiązań okazało się mieć konkretne wady i zalety.

7.1. Różnice w implementacji poszczególnych aplikacji

Pomijając różnice wynikające z używania innych języków i narzędzi przy budowaniu każdej z aplikacji, tworzenie ich różniło się także pod względem subiektywnych odczuć i łatwość pisania kodu.

Możliwość szybkiego debugowania przy użyciu odpowiednich narzędzi jest jedną z najważniejszych kwestii dla dewelopera oraz pośrednio dla zleceniodawcy, ponieważ szybka możliwość znalezienia błędów to duża oszczędność czasu i pieniędzy. Według zasady Pareto zazwyczaj 80% czasu pisania aplikacji poświęcone jest tylko na 20% najbardziej problematycznej części kodu [46], co tym bardziej pokazuje jak ważne jest odpowiednie radzenie sobie z błędami.

Budując aplikację w Android Studio najwięcej problemów sprawiło stworzenie zaplanowanego wyglądu interfejsu. Choć dostępne są gotowe, ostylowane już elementy, tworzenie własnych nie jest tak intuicyjne jak w przypadku technologii webowych. Przykładem może być np. ustawianie własnych kształtów kolorowego tła w awatarze użytkownika. Często pojawiały się także problemy z niezgodnymi wersjami SDK przy instalowaniu modułów, jednak dość łatwo można było je rozwiązać. Sporadycznie zdarzało się jednak, że błędy w konsoli nie wskazywały precyzyjnie w czym leży problem, co wydłużało czas pracy nad aplikacją.

Aplikacja hybrydowa okazała się być zdecydowanie najcięższa w debugowaniu. O ile sama budowa aplikacji nie różni się mocno od budowania aplikacji webowej – kod jest prawie identyczny jak w przypadku React.js - samo znajdowanie błędów zajęło zdecydowaną większość poświęconego na nią czasu. Choć React Native umożliwia korzystanie z Hot i Live Reloading, czyli automatycznego odświeżania aplikacji uruchamianego na każdą zmianę w kodzie, najczęściej po każdej większej edycji kodu konieczne było zupełne odinstalowanie i ponowne zainstalowanie aplikacji. Problem często stwarzało też włączone debugowanie JavaScript w przeglądarce – aplikacja nie pokazywała

wtedy żadnych treści. Niestety w wielu takich przypadkach nie pojawiały się informacje o błędach – ani w konsoli przeglądarki, ani w Android Studio - natomiast aplikacja przestawała działać, co powodowało konieczność wyszukiwania potencjalnych błędów na forach internetowych ‘w ciemno’. Niestety dokumentacja często okazywała się niewystarczająca do rozwiązywania problemów. Bardzo dużą przeszkodę stanowiło także instalowanie modułów – są one potrzebne, aby aplikacja napisana w React Native mogła działać, ponieważ framework ten nie oferuje domyślnie wielu koniecznych funkcjonalności takich jak np. nawigacja między widokami. Sam proces ich instalacji jest dość skomplikowany ze względu na konieczność wprowadzania zmian bezpośrednio w wielu plikach osobno dla Androida i iOS. Praktycznie za każdym razem pojawiały się także błędy związane z niekompatybilnością modułów – konieczne było instalowanie np. ich starszych wersji lub implementowanie zupełnie innych rozwiązań. Ze względu na bardzo szybki rozwój React Native bardzo ciężko dobrać odpowiednie wersje wtyczek tak, aby aplikacja odpowiednio działała. Wiele z nich budowana jest przez osoby z małym doświadczeniem (ponieważ sam React Native jest dość świeżym rozwiązaniem), przez co ilość błędów pojawiająca się na Androidzie i iOS jest naprawdę duża i ciężka do rozwiązywania na własną rękę. Jest to jeden z największych problemów React Native – na tyle poważny, że nawet najbardziej popularna korzystająca z niego firma (zaraz po jego twórcach), czyli AirBnb, ogłosiła, iż odchodzą od budowania nowych rozwiązań w React Native i wracają do rozwiązań natywnych [47]. Wbrew początkowym założeniom, w rzeczywistości wymagana jest znajomość platform Android i iOS oraz ich natywnych języków. Bez technicznej wiedzy o możliwościach tych konkretnych platform oraz rozumienia działania kodu natywnego, np. gradle czy application manifest w przypadku Androida, stworzenie dużej aplikacji w React Native może okazać się naprawdę ciężkie i nieopłacalne dla zleceniodawcy. Budowa interfejsu okazała się trudniejsza niż początkowo zakładano – choć stylowanie komponentów przypomina pisanie styli w aplikacji webowej i oparte jest na popularnym dla nich flexboxie, ilość atrybutów jakie można wykorzystać jest mocno ograniczona, natomiast stylowanie w narzędziach deweloperskich najczęściej nie działało razem z automatycznym odświeżaniem podglądu, co skutkowało koniecznością stylowania na zasadzie prób i błędów.

Debugowanie w aplikacji webowej sprawiło najmniej problemów – Chrome Developer Tools razem z rozszerzeniem React jest bardzo zaawansowanym i wciąż rozwijanym narzędziem, które w prosty sposób (w przeciwieństwie do długich logów w Javie) od razu wskazuje na problem i w którym miejscu (często nawet w jaki sposób) można go naprawić. Dokumentacje wszystkich potrzebnych modułów okazały się dość rozbudowane, a wszystkie problemy z jakimi spotkano się w trakcie budowy aplikacji opisane były już przez innych programistów na forach i blogach.

Implementacja wielu funkcjonalności przedstawiona jest także w formie poradników na konkretnych przykładach. W sieci znajduje się też bardzo duża ilość blogów prezentujących różne rozwiązania przy budowaniu aplikacji webowej, pisana przez pracowników dużych, kompetentnych firm. Wszystko to spowodowało, że choć pisanie aplikacji webowej wraz ze wszystkimi funkcjonalnościami PWA mogło wydawać się najbardziej skomplikowane i wymagało napisania największej ilości kodu oraz pracy z największą ilością wtyczek, nie sprawiało ono zbyt wielu problemów i zajęło mniej czasu niż np. budowanie aplikacji hybrydowej. W tym przypadku najprostsze okazało się także tworzenie wyglądu komponentów i interfejsu, ponieważ stylowanie w CSS daje prawie nieograniczone możliwości.

7.2. Przeprowadzone testy

Na potrzeby przetestowania i porównania wszystkich trzech aplikacji do bazy danych wgrano 100 postów, każde z innym zdjęciem. Każda z aplikacji została przebadana pod kątem czasu wykonywania przykładowych funkcji, obciążenia CPU (ang. *central processing unit*, procesor operacyjny) i GPU (ang. *graphic processing unit*, procesor graficzny), zużycia pamięci i baterii, rozmiaru aplikacji oraz ogólnego doświadczenia użytkownika. Aplikacje przetestowano na urządzeniu Samsung Galaxy S8 z systemem Android 9.

Obciążenie CPU, zużycie baterii i pamięci RAM dla aplikacji hybrydowej i natywnej zostały zmierzone korzystając z narzędzia Android Profiler w Android Studio. Możliwości testowania aplikacji PWA są niestety bardziej ograniczone – do zmierzenia obciążenia CPU wykorzystano narzędzie JavaScript Profiler wbudowanego w Chrome Developer Tools, nie ma natomiast możliwości sprawdzenia zużycia baterii oraz pamięci RAM dla tego typu aplikacji. Obciążenie GPU zostało zmierzone dla wszystkich trzech aplikacji za pomocą wbudowanego w urządzenie narzędzia Profile GPU Rendering. Informacje o zużytej pamięci i zajętym przez aplikację miejscu pochodzą z informacji o aplikacjach dostępnych w ustawieniach urządzenia.

7.3. Czas uruchomienia

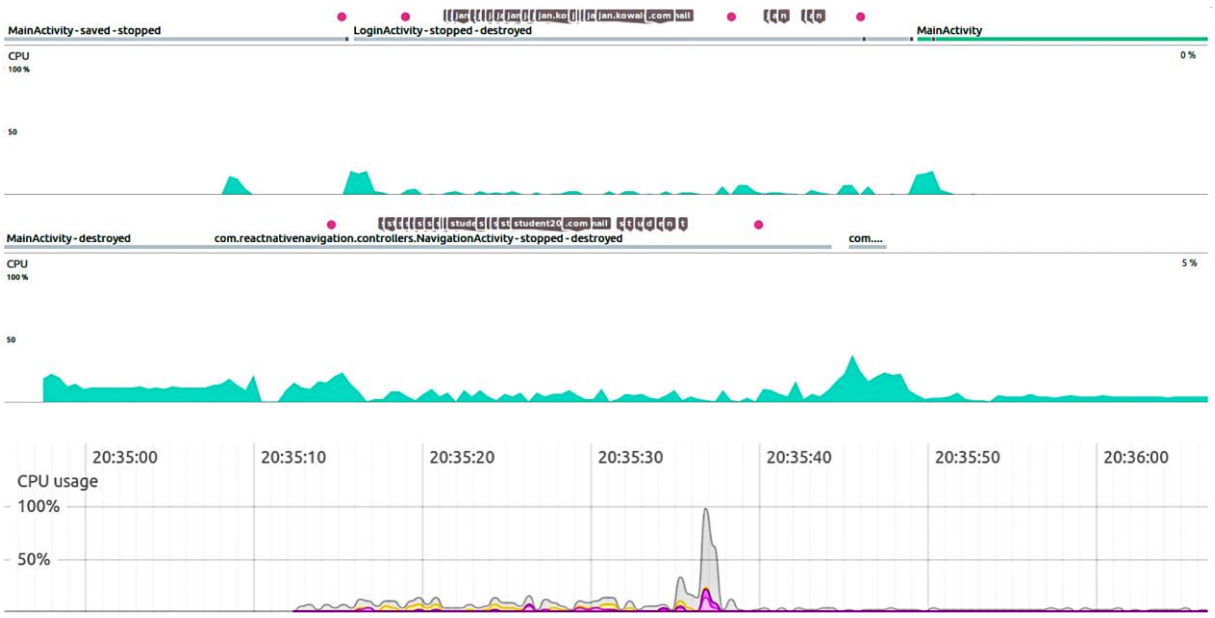
Dla każdej aplikacji trzykrotnie zmierzony został czas uruchomienia się, a następnie wyniki te zostały zsumowane i podzielone przez ilość prób, dając w rezultacie średni czas uruchomienia się aplikacji. Dla aplikacji natywnej średnia ta wyniosła 1,908 sekundy, dla aplikacji hybrydowej 3,383 sekundy, natomiast dla aplikacji webowej 4,245 sekundy. Najszybciej uruchamiającą się aplikacją

była aplikacja natywna. Choć różnice kilku sekund mogą wydawać się niewielkie, w praktyce mają wpływ na doświadczenie użytkownika.

7.4. Obciążenie CPU

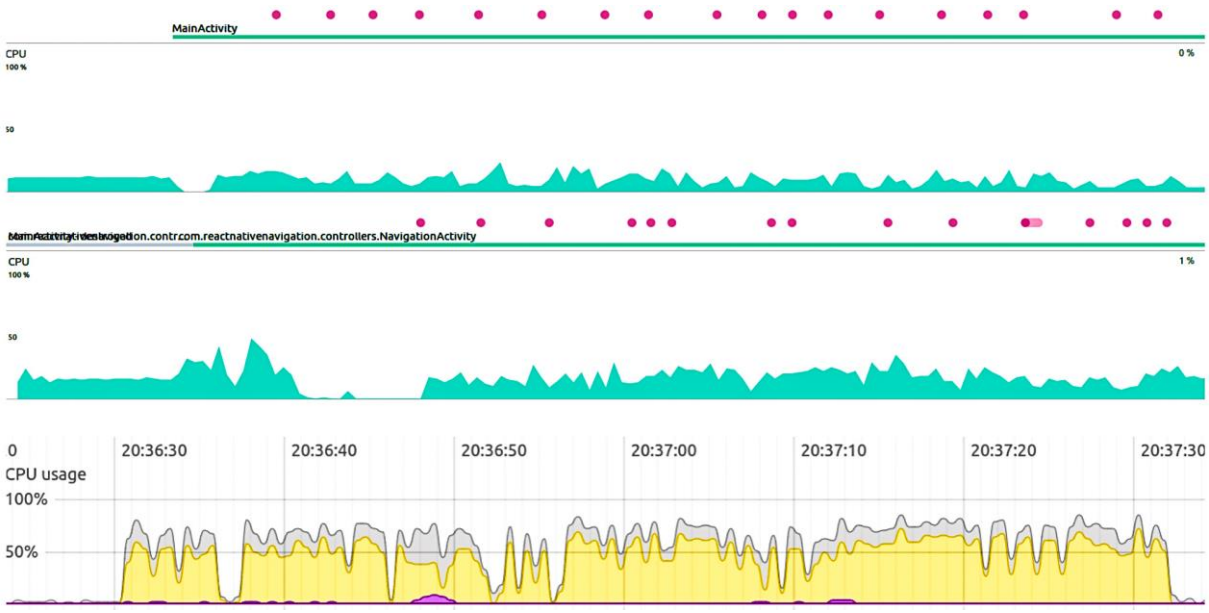
Jedną z najważniejszych miar wydajności aplikacji jest obciążenie przez nią procesora. Wpływa to m.in. na szybkość i płynność działania oraz oszczędność baterii [48]. Dla każdej z badanych aplikacji prześledzono wykres obciążenia CPU przy wykonywaniu standardowych procesów takich tak zalogowanie się, przewijanie strony głównej i doładowywanie postów, dodanie posta korzystając z kamery oraz wybierając zdjęcie z galerii a także przejścia do widoku użytkownika i usunięcia posta.

Logowanie się użytkownika obejmuje wyświetlenie widoku logowania, wpisanie adresu e-mail i hasła, kliknięcie w przycisk logowania i przekierowanie do strony głównej. Zdecydowanie największe skoki w użyciu CPU widać w aplikacji PWA, sięgają one aż około 75% przy zmianie widoku. Sam proces wypełnienia formularza nie jest obciążający dla praktycznie żadnej aplikacji, gdzie poziom obciążenia utrzymuje się poniżej kilkunastu procent. Wyraźny wzrost widać natomiast przy zmianie widoku i powrocie do strony głównej – w przypadku aplikacji hybrydowej sięga 36%, w przypadku aplikacji natywnej 21% (Wykres 6).



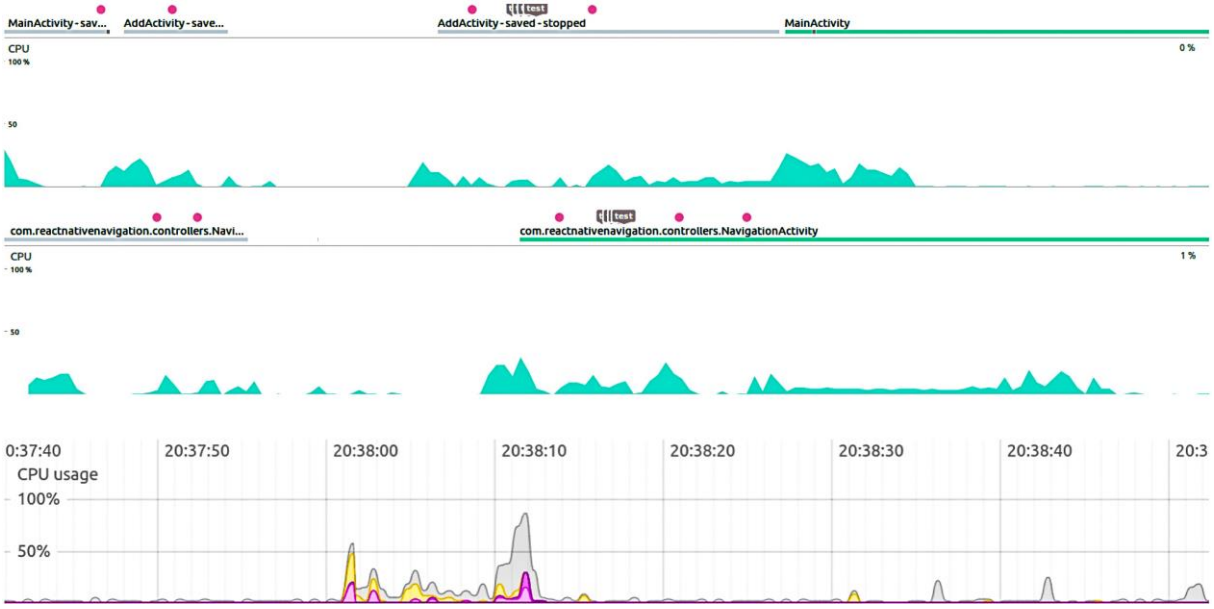
Wykres 6. Obciążenie CPU podczas logowania się użytkownika
(od góry: aplikacja natywna, hybrydowa, PWA). Źródło: opracowanie własne.

Wyświetlenie strony głównej i przewijanie postów to przede wszystkim doładowywanie zdjęć i obsługa lazy-loading, czyli generowania widoków na bieżąco wraz z przewijaniem strony. Najmniej obciążające okazało się to być dla aplikacji natywnej, w tym przypadku wartości nie przekraczały raczej 20%. Aplikacja hybrydowa ładowała kolejne posty z obciążeniem między 20 a 30%. Zdecydowanie najbardziej obciążające było to w przypadku aplikacji webowej – wartości te sięgały nawet 70% (Wykres 7).

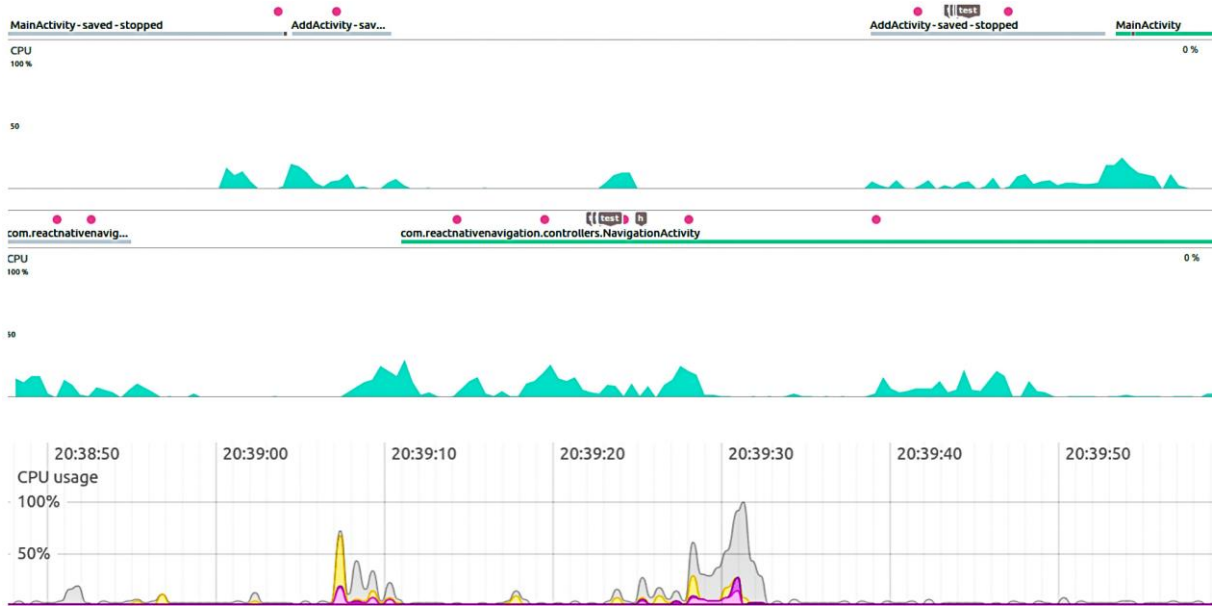


Wykres 7. Obciążenie CPU podczas przewijania postów
(od góry: aplikacja natywna, hybrydowa, PWA). Źródło: opracowanie własne.

Dodawanie postów podzielone zostało na dwa osobne procesy – z dodawaniem zdjęcia z kamery oraz z galerii urządzenia. W obu przypadkach zaobserwować można, że proces ten nie jest zbytnio obciążający dla żadnej z aplikacji, szczególnie biorąc pod uwagę fakt, że tylko aplikacja webowa ma wbudowaną w widoku obsługę kamery bez konieczności przejścia do osobnego widoku.



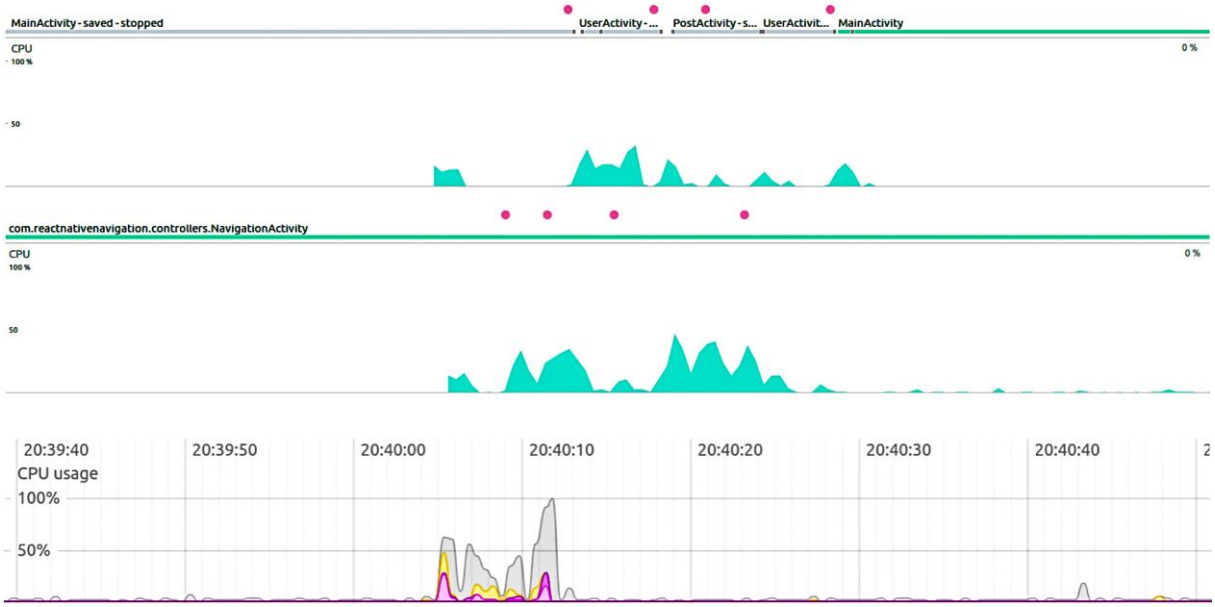
Wykres 8. Obciążenie CPU podczas dodawania zdjęcia z kamery urządzenia (od góry: aplikacja natywna, hybrydowa, PWA). Źródło: opracowanie własne.



Wykres 9. Obciążenie CPU podczas dodawania zdjęcia z galerii urządzenia (od góry: aplikacja natywna, hybrydowa, PWA). Źródło: opracowanie własne.

Usunięcie posta składa się z następujących kroków: przejście do profilu użytkownika i załadowanie kafelków z jego postami, kliknięcie na konkretny post i przejście do widoku danego posta, kliknięcie w ikonę opcji i wyświetlenie nakładki z dostępnymi opcjami, wybranie opcji ‘Usuń’

i przekierowanie do strony głównej. Duża ilość przejść między widokami w tym procesie okazała się najcięższa dla aplikacji hybrydowej – każde przejście oraz proces usunięcia zdjęcia powodowało skok obciążenia CPU od 36% do nawet 50%. W przypadku aplikacji webowej największy skok można było zauważyć podczas wyświetlenia listy z postami użytkownika oraz podczas procesu usuwania posta – wartość skoczyła aż do 100%. Ponownie, najmniej obciążająca dla procesora okazała się być aplikacja natywna (Wykres 10).



Wykres 10. Obciążenie CPU podczas usuwania posta (od góry: aplikacja natywna, hybrydowa, PWA). Źródło: opracowanie własne.

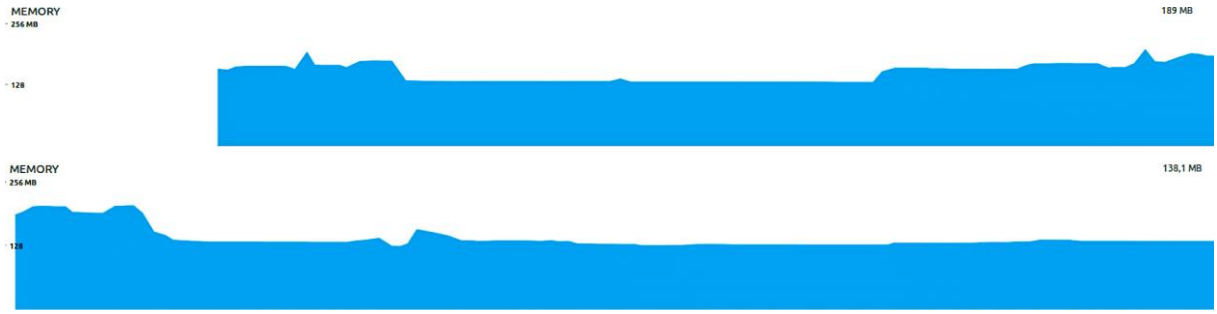
7.5. Zużycie pamięci

Podczas działania aplikacji urządzenie na bieżąco dostosowuje ilość pamięci przeznaczonej na jej procesy i dane. W momencie, gdy aplikacja potrzebuje więcej miejsca niż urządzenie jest jej w stanie zapewnić, zawiesza się ona lub wyłącza, dlatego też ważne jest testowanie ilości zajmowanego przez nią miejsca i sprawdzanie, czy nie występują tzw. *memory leaks*, czyli sytuacja, gdy niepotrzebna pamięć jest niewłaściwie zwalniana do puli systemowej. [49].

Ze względu na brak dostępu odpowiednich narzędzi zbadano profile pamięci jedynie dla aplikacji natywnej i hybrydowej. Można zauważyć, że aplikacje te wykorzystują mniej więcej podobną ilość miejsca podczas wykonywania opisanych wyżej procesów, np. przewijania

i doładowywania postów (Wykres 11), z lekką przewagą dla aplikacji hybrydowej w przypadku korzystania z galerii zdjęć (Wykres 12) i dla natywnej w przypadku dodawania zdjęcia z kamery (Wykres 13).

Wykres 11. Wykorzystanie pamięci podczas przewijania postów
(od góry: aplikacja natywna, hybrydowa). Źródło: opracowanie własne.



Wykres 12. Wykorzystanie pamięci podczas dodawania zdjęcia z galerii urządzenia
(od góry: aplikacja natywna, hybrydowa). Źródło: opracowanie własne.

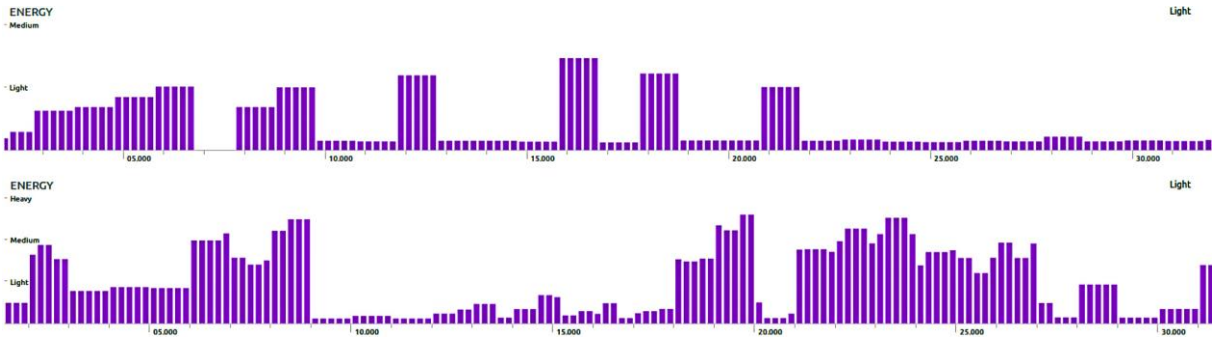
Wykres 13. Wykorzystanie pamięci podczas dodawania zdjęcia z kamery
(od góry: aplikacja natywna, hybrydowa). Źródło: opracowanie własne.

Według informacji o aplikacjach znajdujących się w ustawieniach urządzenia, po wykonaniu opisywanych wcześniej procesów średnie użycie pamięci RAM wynosiło 59 MB dla aplikacji natywnej oraz 73 MB dla aplikacji hybrydowej, natomiast maksymalne użycie to odpowiednio 277 MB oraz 355 MB. Wskazuje to na fakt, że pomimo podobnych wyników na powyższych wykresach aplikacja hybrydowa jest bardziej obciążająca system niż aplikacja natywna.

7.6. Zużycie baterii

Wykorzystując Android Profiler przeprowadzono także testy dla aplikacji natywnej oraz hybrydowej w zakresie zużycia baterii. Niestety nie było to możliwe dla aplikacji webowej.

Najbardziej wyraźną różnicę pomiędzy aplikacjami można było zauważyć podczas przewijania i doładowywania nowych postów, czyli obsługi odpowiednio RecyclerView dla aplikacji natywnej i FlatList dla aplikacji hybrydowej. Aplikacja hybrydowa zużywała znacznie więcej energii za każdym razem, gdy doładowywała nowy post (prawie każde doładowanie oznaczone zostało jako '*heavy*'), w porównaniu z aplikacją natywną (gdzie każde doładowanie posta znajdowało się na granicy oznaczenia '*light'* oraz '*medium'*) (Wykres 14).



Wykres 14. Zużycie baterii podczas przewijania postów (od góry: aplikacja natywna, hybrydowa). Źródło: opracowanie własne.

Według informacji o aplikacjach w urządzeniu po wykonaniu wszystkich wyżej opisanych czynności aplikacja natywna oraz webowa użyły 0% baterii, natomiast aplikacja hybrydowa aż 2%.

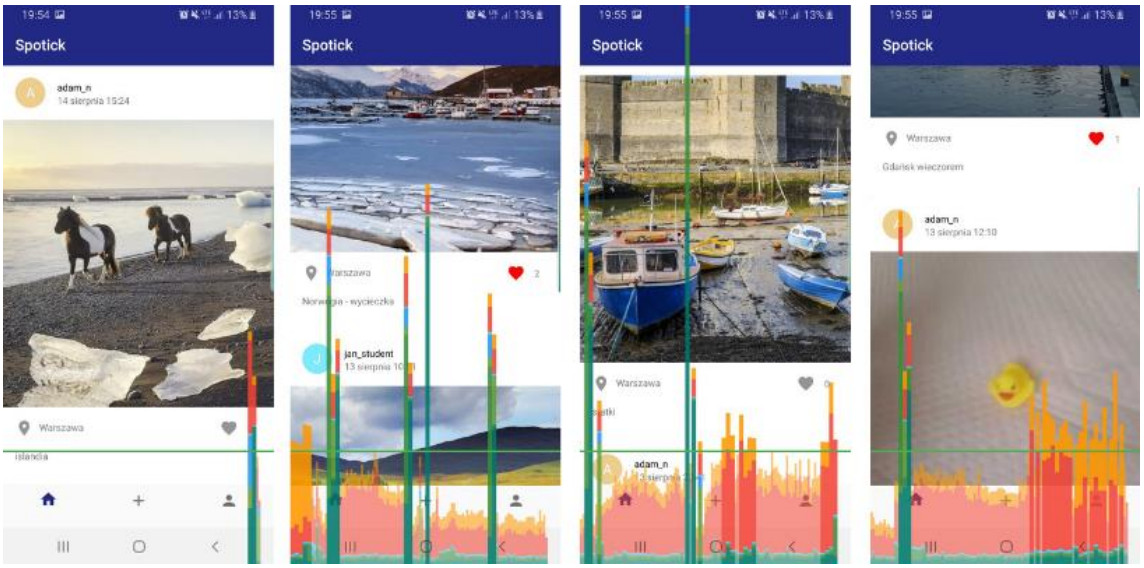
7.7. Obciążenie GPU

Procesor graficzny (GPU) jest odpowiedzialny za renderowanie interfejsów, wyświetlanie wideo oraz animacji. Do zbadania jego obciążenia podczas działania aplikacji wykorzystano wbudowane w urządzenie narzędzie do renderowania profilu GPU. Przedstawia on czas potrzebny do wyrenderowania poprzedniej klatki jako kolorowe, pionowe paski o różnej wysokości – im wyższy

jest pasek, tym dłuższy czas renderowania. Za dobrą wydajność uważa się renderowanie klatek na poziomie 60 klatek na sekundę, co oznacza, że paski powinny być niższe niż zaznaczona zielona linia przedstawiająca 16 milisekund [50]. Im bardziej paski wykraczają ponad ten poziom, tym większe prawdopodobieństwo, że np. animacje nie będą działać płynnie. Każdy kolor przedstawiony na pasku oznacza inną część procesu renderowania klatki (Ilustracja 1).

Ilustracja 1. Legenda kolorów pojawiających się na paskach przy profilowaniu GPU. Źródło: [51]

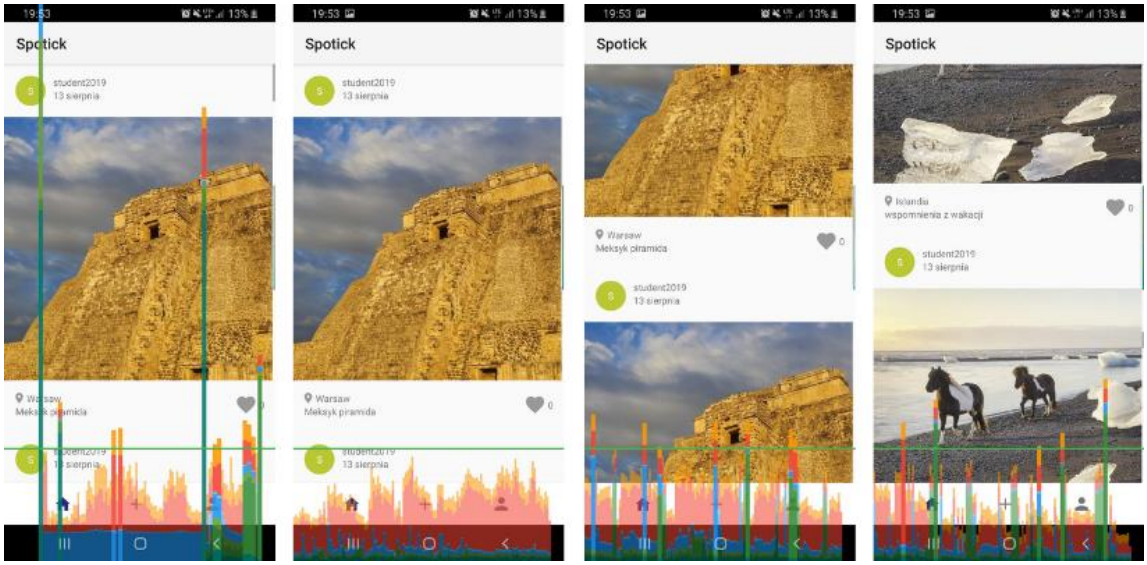
Obciążenie GPU przetestowano na przykładzie procesu przewijania i doładowywania zdjęć, ze względu na największą ilość animacji występujących w tym procesie i istotność płynnego działania. Podczas renderowania aplikacji natywnej można zauważyć, że każde doładowanie nowego posta mocno wykracza poza zieloną linię 16 milisekund. Podczas przewijania dość dużo czasu poświęcane jest na zadania typu '*Issue*' i '*Swap*', co oznacza, że jest wykonywane zbyt dużo poleceń na GPU, szczególnie przy wolniejszym przewijaniu (Ilustracja 2).



Ilustracja 2. Profil GPU dla aplikacji natywnej. Źródło: opracowanie własne.

W przypadku aplikacji hybrydowej profil GPU wskazuje na znacznie wyższy skok podczas początkowego załadowania widoku oraz na delikatne, znacznie niższe niż w przypadku aplikacji natywnej, wzrosty czasu podczas doładowywania nowych postów. Podczas zarówno szybkiego

przewijania jak i wolniejszego GPU zdaje się być mniej obciążone niż w przypadku aplikacji natywnej (Ilustracja 3).



Ilustracja 3. Profil GPU dla aplikacji hybrydowej. Źródło: opracowanie własne.

Profil GPU dla PWA wskazał natomiast silne obciążenie przy początkowym załadowaniu się strony, natomiast zdecydowanie niższe wartości dla późniejszego ładowania się postów i przewijania ekranu w porównaniu z aplikacją natywną i hybrydową. Bardzo sporadycznie paski przekraczały linię 16 milisekund, a nawet wtedy były to minimalne odchylenia. Zgodnie z tymi wynikami przewijanie postów wydawało się być najbardziej płynne w aplikacji webowej (Ilustracja 4).

Ilustracja 4. Profil GPU dla aplikacji webowej. Źródło: opracowanie własne.

7.8. Rozmiar aplikacji

Po przeprowadzonych i opisanych powyżej testach sprawdzono jak dużo miejsca każda z aplikacji zajmuje na urządzeniu. Dane te znajdowały się w informacjach o każdej z aplikacji w ustawieniach urządzenia. Z zebranych danych wynikało, że zdecydowanie najwięcej miejsca zajmuje aplikacja hybrydowa, zarówno pod względem rozmiaru samej aplikacji, przechowywanych danych jak i pamięci cache. Łącznie było to aż 61,81 MB. Aplikacja natywna zajmowała trzykrotnie mniej miejsca – 20,5 MB. Najlżejsza okazała się aplikacja webowa, która zajęła jedynie 258 kB (Tabela 2). Biorąc pod uwagę fakt, że wielu użytkowników zwraca dużą uwagę na rozmiar aplikacji instalowanych na swoich urządzeniach, tak wielka różnica między tymi wynikami daje bardzo dużą przewagę progresywnym aplikacjom webowym.

Tabela 2. Pamięć zajmowana na urządzeniu przez testowane aplikacje. Źródło: opracowanie własne.

	aplikacja natywna	aplikacja hybrydowa	PWA
aplikacje	4,01 MB	41,04 MB	217 kB
dane	9,03 MB	12,10 MB	24,58 kB
pamięć cache	7,47 MB	8,67 MB	16,38 kB
łącznie	**20,5 MB**	**61,81 MB**	**258 kB**

7.9. Porównanie User Experience

Wszystkie trzy aplikacje działają w zadowalający sposób pod względem doświadczenia użytkownika (*User Experience*). Aplikacja natywna wyróżnia się wbudowanymi animacjami i zachowaniem wielu komponentów takich jak np. przyciski, pola formularzy czy kart do jakich przyzwyczajeni są użytkownicy Androida. Posty i zdjęcia doładowują się szybko, a każde przejście pomiędzy widokiem jest animowane, łącznie z przejściami po kliknięciu przycisku ‘wstecz’. Jedynym problemem zdaje się być zauważalne przycinanie się animacji podczas szybkiego doładowywania postów na stronie głównej, co wyjaśnione zostało podczas omawiania profilu GPU.

Aplikacja hybrydowa przypomina pod względem UX w prawie wszystkich kwestiach aplikację natywną, łącznie z animacjami, efektami przycisków. Nie obsługuje jedynie w sposób domyślny przycisku ‘wstecz’ jako nawigacji po widokach. Znacznie lepiej natomiast w tym przypadku wygląda animacja doładowywania postów – nie widać przycinania się nawet przy szybkich ruchach.

Pod względem doświadczenia użytkownika PWA daje największe możliwości ze względu na łatwość stylowania, jednak zaprojektowanie takiej aplikacji wymaga od dewelopera wiedzy na temat UX i dostosowywania każdego elementu do potrzeb użytkowników na własną rękę. Nawet przy korzystaniu z zewnętrznych bibliotek ważna jest konsekwencja w budowaniu widoków. Komponenty pojawiające się w aplikacji wyraźnie różnią się od aplikacji hybrydowej i natywnej, nie ma tu wbudowanych animacji pomiędzy przejściami, natomiast z pewnością jest możliwość dodania ich. Jeśli chodzi o formularze czy też nawigację nie widać tu wyraźnych różnic porównując do pozostałych aplikacji, a posty doładowują się bardzo szybko i bez zacinania się. Jedynym elementem ładującym się odrobinę wolniej jest zdjęcie każdego posta, natomiast ze względu na wcześniej wyświetloną całą resztę elementu (podpisy, polubienia itp.) nie ma to wpływu na doświadczenie użytkownika.

W ramach testu doświadczeń użytkownika zmierzono czas procesu usuwania posta. Wybrano go ze względu na kilkukrotną konieczność kliknięcia elementu i przejścia do innego widoku co dobrze obrazuje UX, oraz brak konieczności wpisywania danych z klawiatury urządzenia, co mogłoby nie dawać obiektywnych rezultatów. W przypadku aplikacji natywnej proces ten trwał 11,37 sekund, w przypadku aplikacji hybrydowej 10,82 sekundy, natomiast najszybciej proces przebiegł na aplikacji webowej – jedynie 6,12 sekundy. Związane było to ze znacznie krótszym czasem ładowania się kolejnych widoków i przejść.

Zmierzono także czas otrzymywania powiadomienia w każdej aplikacji po dodaniu nowego posta. Dla aplikacji natywnej czas ten wyniósł 2,29 sekund, dla aplikacji hybrydowej 4,72 sekundy, a dla aplikacji webowej aż 8,3 sekundy.

7.10. Podsumowanie rezultatów

Biorąc pod uwagę wyniki wszystkich przeprowadzonych testów, aplikacja hybrydowa jest zdecydowanie najcięższa – zajmuje najwięcej miejsca na dysku, korzysta z największych zasobów pamięci i zużywa najwięcej baterii. Z drugiej strony obciąża ona procesor graficzny w mniejszym stopniu niż aplikacja natywna, a oferowane przez nią doświadczenia użytkownika są nieco lepsze niż w przypadku innych aplikacji. Niestety problemy związane z debugowaniem jej znacznie osłabiają jej pozycję w ogólnym porównaniu.

Aplikacja natywna jest wciąż najlepsza, jeśli chodzi o wykorzystanie procesora CPU, zużywa też mniej pamięci RAM niż aplikacja hybrydowa, zajmuje też zdecydowanie mniej miejsca niż ona. Jest jednak aplikacją najbardziej obciążającą procesor graficzny. Wyróżnia się szybkością uruchomienia i natychmiastową obsługą powiadomień.

Progresywna aplikacja webowa wypadła najlepiej, jeśli chodzi o swój rozmiar – w porównaniu do aplikacji natywnej i hybrydowej jest prawie niezauważalna na dysku. Choć uruchamia się najdłużej, działa bardzo szybko i płynnie, obciążając procesor graficzny w najmniejszym stopniu i nie nadwyrężając baterii. To sprawia, że idealnie nadaje się do wyświetlania i przeglądania dużych ilości np. zdjęć czy produktów w sklepie internetowym.

8. Podsumowanie

Celem pracy było porównanie trzech podejść do budowania aplikacji mobilnych – aplikacji natywnej, hybrydowej oraz webowej (PWA). Każde z nich zostało szczegółowo opisane porównując ich wady i zalety na podstawie dostępnej literatury, a następnie przetestowane na przykładzie aplikacji typu social media. Opisane zostały trudności przy ich implementacji, wyniki testów wydajnościowych oraz testów użyteczności.

Podsumowując otrzymane rezultaty można stwierdzić, że każde z wyżej wymienionych podejść ma swoje mocne i słabe strony, oraz że każde z nich może nadawać się odpowiednio do innego przypadku. W momencie, gdy istotna jest niezawodność aplikacji, szybki dostęp do wszelkich funkcji natywnych, obsługa bardzo dużej ilości użytkowników i małe obciążenie urządzenia, najlepszym rozwiązaniem będzie wciąż aplikacja natywna, przygotowana specjalnie dla konkretnej platformy. Z drugiej strony, progresywna aplikacja webowa ma szansę świetnie sprawdzić się w przypadkach, gdy najważniejsze jest zyskanie szerokiego zasięgu użytkowników i przekonanie ich do zainstalowania aplikacji np. znikomym obciążeniem pamięci. Może okazać się także najtańszym podejściem, ze względu na obsługę wszystkich platform, łatwość debugowania (czyli oszczędność czasu) i największą popularność technologii webowych w porównaniu do technologii natywnych czy hybrydowych (co oznacza m.in. większą ilość specjalistów na rynku pracy). Aplikacje tego typu bardzo dobrze dają sobie radę z podstawowymi funkcjami natywnymi, np. geolokalizacją, obsługą powiadomień, działaniem offline czy robieniem zdjęć. Ponadto każda ze stron ma osobny adres URL, a projektowanie widoków nie jest praktycznie niczym ograniczane. Biorąc to wszystko pod uwagę, aplikacje PWA wydają się być idealnym rozwiązaniem dla sklepów internetowych i mniejszych firm, które nie posiadają zasobów finansowych porównywalnych z wielkimi korporacjami. Trzecie podejście, czyli aplikacje hybrydowe, wypadły najsłabiej zarówno pod względem trudności implementacji jak i wyników testów. Stworzona aplikacja była najcięższa, zużywała najwięcej pamięci i baterii. Choć idea tworzenia prawdziwie natywnych aplikacji w technologiach webowych rozwiązywałaby wiele problemów, stworzenie aplikacji hybrydowej gotowej do wykorzystywania w produktach dla użytkowników byłoby na dzień dzisiejszy najbardziej uciążliwe, możliwe natomiast, że w niedalekiej przyszłości sytuacja ta ulegnie zmianie. Na tę chwilę, biorąc pod uwagę wszystkie wady i zalety, można uznać, że podejście to może zostać wyparte przez progresywne aplikacje webowe.

Bibliografia

[1]. M. Waszkiewicz. Komu PWA, czyli jakie biznesy skorzystają z Progressive Web Apps. *https://www.e-point.pl/blog/komu-pwa-czyli-jakie-biznesy-skorzystaja-z-progressive-web-apps*, luty 2019. Dostęp 14 czerwca 2019.

[2]. comScore. The 2017 U.S. Mobile App Report, lipiec 2017.

[3]. M. Schwarzmuller. Progressive Web Apps (PWA) – The Complete Guide. *https://www.udemy.com/progressive-web-app-pwa-the-complete-guide/learn/lecture/7874018#overview*, maj 2019. Dostęp 24 września 2018.

[4]. M. Waszkiewicz. Coraz więcej ofiar wojny handlowej. Niech żyje PWA! *https://www.e-point.pl/blog/coraz-wiecej-ofiar-wojny-handlowej-niech-zyje-pwa*, maj 2019. Dostęp 14 czerwca 2019.

[5]. F. Makowiecki. PWA jako wyzwanie UX. O czym mówiliśmy na Mobile Conference 2019? *https://www.e-point.pl/blog/pwa-jako-wyzwanie-ux*, marzec 2019. Dostęp 14 czerwca 2019.

[6]. A. Zinchuk. Cordova Frameworks: Ionic vs. Framework7. *https://www.toptal.com/apache-cordova/frameworks-ionic-framework7*, 2019. Dostęp 4 sierpnia 2019.

[7]. Google, Progressive Web App Checklist. *https://developers.google.com/web/progressive-web-apps/checklist*, maj 2019. Dostęp 14 czerwca 2019.

[8]. J. Clement. Worldwide mobile app revenues in 2014 to 2023. *https://www.statista.com/statistics/269025/worldwide-mobile-app-revenue-forecast*, czerwiec 2019. Dostęp 14 czerwca 2019.

[9]. J. Clement. Number of mobile app downloads worldwide in 2017, 2018 ans 2022. *https://www.statista.com/statistics/271644/worldwide-free-and-paid-mobile-app-store-downloads*, czerwiec 2018. Dostęp 14 czerwca 2019.

[10]. Bankier.pl, Microsoft uśmierca Windows Phone 8. *https://www.bankier.pl/wiadomosc/Microsoft-usmierca-Windows-Phone-8-7532266.html*, lipiec 2017. Dostęp 15 czerwca 2019.

[11]. R. Kwiatkowski. PWA: progresywne aplikacje webowe to przyszłość mobilnego handlu. *https://matsuu.com/pl/progresywne-aplikacje-webowe-pwa-to-przyszlosc-mobilnego-handlu*, luty 2019. Dostęp 3 sierpnia 2019.

[12]. Matsuu. Czy React Native może zastąpić natywne aplikacje mobile. *https://medium.com/@matsuustudio/czy-react-native-mo%C5%Bce-zast%C4%85pi%C4%87-natywne-aplikacje-mobilne-2ae946895b09*, sierpień 2018. Dostęp 15 czerwca 2019.

[13]. Diophant. Flutter vs React Native. *https://diophant.com/blog/flutter-vs-react-native*, listopad 2018. Dostęp 15 czerwca 2019.

[14]. V. Saratchandran. A guide to Android Instant Apps. *https://medium.com/swlh/https-medium-com-vinodsfingent-a-guide-to-android-instant-apps-c05d905c0098,* listopad 2018. Dostęp 15 czerwca 2019.

[15]. J. Stangarone. Seven hidden risks of native mobile app development. *https://www.mrc-productivity.com/blog/2013/11/7-hiddens-risks-of-native-mobile-app-development,* listopad 2013. Dostęp 15 czerwca 2019.

[16]. F. Richter. The terminal decline of BlackBerry. *https://www.statista.com/chart/8180/blackberrys-smartphone-market-share*, czerwiec 2017. Dostęp 16 czerwca 2019.

[17]. T. Car. Android development is 30% more expensive than iOS. *https://infinum.co/the-capsized-eight/android-development-is-30-percent-more-expensive-than-ios*, październik 2015. Dostęp 16 czerwca 2019.

[18]. The Manifest. Android vs iOS: Which Platform to Build Your App for First? *https://medium.com/@the_manifest/android-vs-ios-which-platform-to-build-your-app-for-first-22ea8996abe1*, luty 2018. Dostęp 16 czerwca 2019.

[19]. Microsoft. What's a Universal Windows Platform (UWP) app? *https://docs.microsoft.com/en-us/windows/uwp/get-started/universal-application-platform-guide,* lipiec 2018. Dostęp 16 czerwca 2019.

[20]. A. Bar. What Web Can Do Today. *https://whatwebcando.today,* czerwiec 2019. Dostęp 20 czerwca 2019.

[21]. *https://caniuse.com*. Dostęp 20 czerwca 2019.

[22]. Greenparrot.pl, Plusy I minusy Progressive Web App w punktach i case study sukcesów PWA, *https://greenparrot.pl/blog/plusy-i-minusy-progressive-web-apps-w-punktach-i-case-study-sukcesow-pwa,* listopad 2018. Dostęp 20 czerwca 2019.

[23]. P. Mansfeld. Projektowanie stron w standardzie AMP. *https://mansfeld.pl/webdesign/projektowanie-stron-w-standardzie-amp*, lipiec 2017. Dostęp 21 czerwca 2019.

[24]. J. Stark, B. Jepson, Android. Tworzenie aplikacji w oparciu o HTML, CSS i JavaScript, wyd. Helion 2013.

[25]. R. Trivedi. Top 5 hybrid Mobile App Frameworks in 2019 – Choose the best one for you. *https://www.websoptimization.com/blog/hybrid-mobile-app-frameworks*, kwiecień 2018. Dostęp 26 czerwca 2019.

[26]. Ionic Docs. *https://ionicframework.com/docs/building/webview,* lipiec 2019. Dostęp 4 sierpnia 2019.

[27]. State of Javascript 2018. *https://2018.stateofjs.com/front-end-frameworks/overview*. Dostęp 12 sierpnia 2019.

[28]. Material-UI Documentation. *https://material-ui.com/styles/basics*. Dostęp 12 sierpnia 2019.

[29]. Workbox Documentation. *https://developers.google.com/web/tools/workbox,* maj 2019. Dostęp 14 sierpnia 2019.

[30]. Mobile Operating System Market Share Worldwide - July 2019. *https://gs.statcounter.com/os-market-share/mobile/worldwide*. Dostęp 12 sierpnia 2019.

[31]. Firebase Docs. *https://firebase.google.com/docs*. Dostęp 14 sierpnia 2019.

[32]. React Native Docs. *https://facebook.github.io/react-native*. Dostęp 12 sierpnia 2019.

[33]. Redux Documentation. *https://redux.js.org*. Dostęp 15 sierpnia 2019.

[34]. IndexedDB API, MDN web docs. *https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API*. Dostęp 12 sierpnia 2019.

[35]. Web Push Notifications: Timely, Relevant, and Precise. *https://developers.google.com/web/fundamentals/push-notifications*, maj 2019. Dostęp 12 sierpnia 2019.

[36]. Repozytorium web-push. *https://github.com/web-push-libs/web-push*. Dostęp 12 sierpnia 2019.

[37]. Repozytorium react-router. *https://github.com/ReactTraining/react-router*. Dostęp 12 sierpnia 2019.

[38]. Repozytorium react-geocode. *https://github.com/shukerullah/react-geocode*. Dostęp 12 sierpnia 2019.

[39]. Repozytorium react-lazyload. *https://github.com/twobin/react-lazyload*. Dostęp 12 sierpnia 2019.

[40]. Github Pages. *https://pages.github.com.* Dostęp 12 sierpnia 2019.

[41]. Repozytorium redux-persist. *https://github.com/rt2zz/redux-persist*. Dostęp 12 sierpnia 2019.

[42]. Repozytorium react-native-image-picker. *https://github.com/react-native-community/react-native-image-picker*. Dostęp 12 sierpnia 2019.

[43]. Create a List with RecyclerView, Android Docs. *https://developer.android.com/guide/topics/ui/layout/recyclerview*. Dostęp 13 sierpnia 2019.

[44]. The Web Push Protocol, Web Fundamentals. *https://developers.google.com/web/fundamentals/push-notifications/web-push-protocol*. Dostęp 13 sierpnia 2019.

[45]. The Offline Cookbook, Web Fundamentals. *https://developers.google.com/web/fundamentals/instant-and-offline/offline-cookbook.* Dostęp 14 sierpnia 2019.

[46]. Pareto Principle for Software, Embedded Artistry. *https://embeddedartistry.com/blog/2017/6/14/pareto-principle-for-software*, czerwiec 2017. Dostęp 14 sierpnia 2019.

[47]. React Native at Airbnb: The Technology. *https://medium.com/airbnb-engineering/react-native-at-airbnb-the-technology-dafd0b43838*, czerwiec 2018. Dostęp 12 sierpnia 2019.

[48]. Inspect CPU activity with CPU Profiler, Android Studio. *https://developer.android.com/studio/profile/cpu-profiler*. Dostęp 16 sierpnia 2019.

[49]. Memory Leak Patterns in Android, Medium. *https://android.jlelse.eu/memory-leak-patterns-in-android-4741a7fcb570*, marzec 2017. Dostęp 16 sierpnia 2019.

[50]. Inspect GPU rendering speed and overdraw, Android Studio. *https://developer.android.com/studio/profile/inspect-gpu-rendering*. Dostęp 16 sierpnia 2019.

[51]. A. Spallino. *Native versus hybrid mobile application development for professional membership services*, czerwiec 2018.

[52]. StackOverflow Developer Survey Results 2018. *https://insights.stackoverflow.com/survey/2018.* Dostęp 16 sierpnia 2019.

[53]. The Good and The Bas of Xamarin Mmobile Development, Altexsoft. *https://www.altexsoft.com/blog/mobile/pros-and-cons-of-xamarin-vs-native*, kwiecień 2019. Dostęp 16 sierpnia 2019.

[54]. Alex Russel, Progressive Web Apps: Escaping Tabs Without Losing Our Soul. *https://medium.com/@slightlylate/progressive-apps-escaping-tabs-without-losing-our-soul-3b93a8561955*, sierpień 2015. Dostęp 17 sierpnia 2019.

Spis wykresów

Spis listingów

Spis rysunków

Spis ilustracji

Spis tabel