POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 9

Budowanie i publikacja (EAS) aplikacji React Native z Expo

Mateusz Pawełkiewicz

1.10.2025

1. Assets: Ikony aplikacji i ekrany powitalne (Splash)

Ikony aplikacji: Każda aplikacja mobilna potrzebuje **ikony**, widocznej na ekranie głównym urządzenia i w sklepach. W Expo konfigurujemy ikonę w pliku konfiguracyjnym (np. app.json). Najprościej jest przygotować obraz PNG 1024x1024 (kwadrat bez zaokrąglonych rogów) i dodać go jako właściwość "icon". Przykład konfiguracji ikony w app.json:

```
{
  "expo": {
    "icon": "./assets/images/icon.png"
  }
}
```

Expo (EAS Build) na podstawie tego jednego pliku generuje wymagane rozmiary ikon dla platformy iOS (wymagane różne rozdzielczości). Dla Androida warto skorzystać z funkcji **Adaptive Icon** – można zdefiniować osobno warstwę pierwszoplanową (foreground) i tło (background), by ikona dobrze wyglądała w różnych kształtach narzucanych przez system. W pliku app.json służy do tego klucz "android":{"adaptiveIcon": ...} pozwalający wskazać osobny obraz pierwszoplanowy (foregroundImage), monochromatyczny (monochromeImage) oraz kolor tła (backgroundColor lub obraz tła). Dla starszych Androidów (bez adaptive icons) można opcjonalnie zdefiniować "android.icon" – pojedynczą ikonę, która łączy warstwę tła i pierwszoplanową.

Na iOS od Expo SDK 54 obsługiwany jest format **Icon Composer** – można przygotować katalog .icon zawierający zestaw ikon wygenerowany np. narzędziem Apple *Icon Composer* i wstawić go do projektu, podając ścieżkę w ios.icon. Alternatywnie (i we wcześniejszych SDK) wystarczy pojedynczy plik PNG 1024x1024 – Expo wygeneruje z niego pozostałe rozmiary podczas budowania aplikacji. Pamiętaj, aby ikona wypełniała cały kwadrat i nie zawierała przezroczystych obszarów ani własnych zaokrągleń (system sam doda maskę). Można również przewidzieć różne warianty ikony na iOS (ciemny, jasny tryb, tryb tint) za pomocą obiektu zamiast ścieżki (w kluczu ios.icon), jednak najczęściej wystarcza jedna uniwersalna ikona.

Ekran powitalny (Splash Screen): Ekran powitalny to pierwsze, co widzi użytkownik uruchamiając aplikację – wyświetla się podczas ładowania aplikacji. W Expo za konfigurację splash screenu odpowiada biblioteka **expo-splash-screen** (automatycznie dołączana). Konfigurację splash screenu definiujemy również w app.json poprzez plugin expo-splash-screen, wskazując m.in. tło i obraz logo. Przykład konfiguracji w app.json z użyciem wtyczki (config plugin) expo-splash-screen:

```
{
  "expo": {
    "plugins": [
      ["expo-splash-screen", {
        "backgroundColor": "#232323",
        "image": "./assets/images/splash-icon.png",
        "dark": {
          "image": "./assets/images/splash-icon-dark.png",
          "backgroundColor": "#000000"
        },
```

```
      "imageWidth": 200
    }]
   ]
  }
}
```

W powyższym przykładzie ustawiliśmy kolor tła (backgroundColor), ścieżkę do obrazu wyświetlanego na splash (image), a także opcjonalny wariant **dark mode** z innym obrazem i tłem (dark.image i dark.backgroundColor). Klucz imageWidth pozwala dostosować rozmiar wyświetlanego obrazu (w pikselach). Możemy również osobno określić właściwości dla platform **Android** i **iOS** korzystając z pól "android" i "ios" wewnątrz konfiguracji pluginu. Przykładowo, możemy na Androidzie użyć innego obrazu lub innego trybu skalowania (resizeMode: contain, cover lub native) niż na iOS.

Rozdzielczości i format splash: Najlepiej przygotować obraz/logo splash w formacie PNG (Expo wspiera tylko PNG dla splash), z przeźroczystym tłem jeśli chcemy by kolor tła był oddzielnie definiowany. Zalecana minimalna wielkość grafiki to 1024x1024 dla uniwersalności, jednak często wykorzystuje się też większe grafiki tła. Dzięki config plugin Expo automatycznie ustawi wymagane zasoby natywne w Android Manifest i iOS LaunchScreen storyBoard podczas budowy aplikacji (przy użyciu EAS Build) – w trybie zarządzanym nie musimy ręcznie tworzyć plików w Xcode/Android Studio. **Uwaga:** Podczas testowania splash screenu **nie używamy Expo Go ani development build** – Expo Go wyświetla własny ekran (ikonę Expo) zamiast naszego splash, a dev-client też ma własny splash, co może zaburzać test. Należy testować splash na buildach typu preview lub production.

Kontrola czasu wyświetlania splash: Domyślnie splash screen znika automatycznie, gdy aplikacja jest gotowa, ale Expo pozwala to kontrolować manualnie poprzez API SplashScreen z biblioteki expo-splash-screen. Możemy np. **wydłużyć czas trwania splash screenu**, aby poczekać na załadowanie ważnych zasobów (np. danych z API). Służy do tego metoda SplashScreen.preventAutoHideAsync(), wywoływana zaraz na starcie aplikacji (przed renderowaniem). Następnie, gdy jesteśmy gotowi, wywołujemy SplashScreen.hideAsync() aby ukryć ekran powitalny. Ważne, by nie trzymać splash screenu dłużej niż to konieczne – dobre praktyki nakazują jak najszybciej pokazać użytkownikowi interfejs właściwej aplikacji. Przykład (w kontekście komponentu React):

```
import * as SplashScreen from 'expo-splash-screen';
import { useEffect, useState } from 'react';

// Zapobiegamy automatycznemu schowaniu splash:
SplashScreen.preventAutoHideAsync();

export default function App() {
 const [appReady, setAppReady] = useState(false);

 useEffect(() => {
  async function prepareResources() {
   // tutaj np. load fonts, fetch data etc.
   await loadImportantData();
   setAppReady(true);
  }
```

```
    prepareResources();
  }, []);

  useEffect(() => {
    if (appReady) {
      SplashScreen.hideAsync(); // ukryj splash, gdy gotowe
    }
  }, [appReady]);

  if (!appReady) {
    return null; // nie renderuj nic (pozostaje splash)
  }

  return <MainAppContent />;
}
```

W ten sposób **splash screen pozostanie widoczny dłużej**, dopóki appReady nie zmieni się na true i nie ukryjemy go ręcznie.

Animacja zanikania: Od SDK 52 Expo wspiera opcje animacji przy przejściu ze splash do aplikacji – można ustawić **zanikanie (fade)** i czas trwania animacji. Służy do tego metoda SplashScreen.setOptions({ fade: true, duration: 1000 }), którą możemy wywołać przed renderowaniem aplikacji. duration to czas animacji w milisekundach (np. 1000ms = 1s). Ta opcja jest opcjonalna – jeśli jej nie ustawimy, splash zniknie natychmiast po załadowaniu aplikacji.

Podsumowując, właściwe przygotowanie assets w Expo obejmuje: **ikonę aplikacji** w wymaganych rozdzielczościach (najlepiej 1024x1024 PNG jako źródło, plus konfiguracja w app.json), **ekran powitalny** skonfigurowany przez expo-splash-screen (tło i obraz dostosowane do jasnego/ciemnego motywu), a także ewentualną **kontrolę czasu wyświetlania splash** poprzez API expo-splash-screen (jeśli potrzebujemy opóźnić start aplikacji do załadowania zasobów).

2. Expo Application Services (EAS): budowanie i aktualizacje OTA

Expo Application Services (**EAS**) to zestaw usług chmurowych Expo usprawniających budowanie i dystrybucję aplikacji. W skład EAS wchodzą m.in. **EAS Build** (usługa budowania natywnych plików IPA/AAB/APK w chmurze) oraz **EAS Submit** (automatyczne zgłaszanie buildów do sklepów). EAS zastępuje dotychczasowe klasyczne komendy expo build:ios / expo build:android, oferując bardziej elastyczne i potężne możliwości.

EAS Build vs klasyczne expo build

Dawniej Expo oferowało **expo build** dla aplikacji w **managed workflow**, co pozwalało zbudować apk/ipa bez konfiguracji Xcode/Android Studio. **EAS Build** jest następcą i rozwinięciem tej usługi – umożliwia budowanie *każdej* aplikacji React Native (zarówno managed Expo, jak i bare React Native) w chmurze. Dzięki EAS Build możemy korzystać z bibliotek natywnych spoza standardowego zestawu Expo – EAS generuje tzw. *development*

clients lub *custom builds*, które zawierają te natywne moduły. W praktyce oznacza to, że jeśli kiedyś musieliśmy "ejectować" z Expo by użyć jakiejś biblioteki (np. Bluetooth, WebRTC, In-App Payments), to teraz możemy pozostać w Expo i dołączyć natywne moduły poprzez **config plugins** oraz zbudować apk/ipa przez EAS. EAS Build generuje **mniejsze binaria** zawierające tylko potrzebne natywne moduły (co zmniejsza rozmiar aplikacji).

Kluczowe różnice między starym `expo build` a `eas build`:

- **Obsługa typu projektu:** `expo build` działał tylko dla projektów Expo zarządzanych. **EAS Build działa dla wszystkich projektów RN**, także tych z własnym kodem natywnym czy config pluginami.
- **Profile budowania:** `expo build` miał domyślne ustawienia; EAS Build pozwala na definicję różnych **profilów budowania** (development, preview, production) w pliku **eas.json** – o tym więcej za chwilę.
- **Dystrybucja wewnętrzna:** EAS natywnie wspiera *internal distribution* – łatwe udostępnianie buildów testowych poza sklepem. Na iOS jest to rozwiązane przez automatyczne tworzenie provisioning profile typu *Ad Hoc* (nie trzeba ręcznie dodawać UDID urządzeń, EAS może wygenerować profil ad-hoc). Na Androidzie EAS umożliwia wygodne generowanie plików **.apk** do testów (zamiast tylko *.aab*).
- **Integracja z CI/CD:** EAS CLI jest przystosowane do użytku w pipeline (polecenia JSON, webhooki, etc.), co ułatwia automatyzację. `expo build` było raczej narzędziem interaktywnym.
- **Zarządzanie credenciales:** EAS przechowuje i automatycznie zarządza kluczami i certyfikatami. Jeśli wcześniej robiliśmy `expo build`, możemy użyć tych samych certyfikatów/keystore w EAS – narzędzie samo wykryje istniejące dane na serwerach Expo. EAS potrafi też samo wygenerować keystore Android czy certyfikaty i provisioning profile iOS przy pierwszym buildzie, prowadząc dewelopera przez interaktywny setup.
- **Budowanie na najnowszych SDK:** Expo przestało rozwijać `expo build` i zapowiedziało jego wygaszenie w 2023, więc nowe wymagania (np. architektury, nowe Xcode/SDK) są wspierane w EAS. Przykładowo EAS zapewnia aktualne obrazy build z nowymi Xcode zgodnie z wymaganiami Apple (coś, czego stary `expo build` by nie dostał, gdyby nie był aktualizowany).

Podsumowując: **EAS Build** to nowoczesna usługa CI/CD od Expo, która **upraszcza proces kompilacji aplikacji mobilnej** – od momentu surowego kodu JavaScript aż do gotowego artefaktu (.apk, .aab, .ipa) w sklepie czy do testów. Pozwala to deweloperom skupić się na pisaniu kodu, a ciężar kompilacji przenosi na chmurę Expo.

Wymagania i konfiguracja EAS Build (eas.json, profile)

Aby skorzystać z EAS Build, potrzebujemy założyć konto Expo (darmowe) i zainstalować **EAS CLI** (co najmniej v3+). Logujemy się w CLI komendą `eas login`, a następnie przeprowadzamy jednorazową konfigurację projektu komendą `eas build:configure`. Ta komenda wygeneruje plik `eas.json` w głównym folderze projektu z przykładową konfiguracją profili budowania.

Domyślnie Expo tworzy trzy profile buildów w eas.json: **development**, **preview** i **production**. Każdy profil to zbiór ustawień określających jak ma zostać zbudowana aplikacja. Przykład domyślnego eas.json utworzonego przez Expo:

```
{
  "build": {
    "development": {
      "developmentClient": true,
      "distribution": "internal"
    },
    "preview": {
      "distribution": "internal"
    },
    "production": {}
  }
}
```

Profil "development": przeznaczony do buildów deweloperskich. Ma ustawienie "developmentClient": true co oznacza, że do aplikacji zostanie dołączony **expo-dev-client** – czyli nasz build będzie działał podobnie do Expo Go, umożliwiając wczytywanie projektu w trakcie developmentu. Taki build *development* zawiera narzędzia developerskie (debug menu itp.) i **nigdy** nie jest wysyłany do sklepów. Dodatkowo "distribution": "internal" oznacza, że build będzie przygotowany do dystrybucji wewnętrznej (testy) – np. na iOS wygeneruje się plik .ipa typu Ad Hoc (możliwy do wgrania na urządzenia testowe, ale nie do App Store). Na Androidzie distribution: internal spowoduje zbudowanie APK (zamiast AAB) dla łatwej instalacji bezpośrednio.

Uwaga: Dla iOS można w ramach dev-buildów robić też buildy na **Simulator** (Mac). Ustawienie ios.simulator: true w profilu spowoduje, że EAS wygeneruje aplikację w formacie .app do uruchomienia w iOS Simulator zamiast .ipa. Często tworzy się osobny profil np. "development-simulator" jeśli potrzebujemy obu wariantów (urządzenie fizyczne vs symulator).

Profil "preview": profil pośredni – **build testowy** przypominający produkcyjny, ale nie przeznaczony do sklepu. Nie zawiera już expo-dev-client (brak developmentClient w konfiguracji), więc jest to normalna aplikacja bez menu debug etc., ale nadal z distribution: "internal" czyli do dystrybucji testowej (APK lub iOS Ad Hoc/TestFlight). Używamy go do udostępniania wersji testerom, zespołowi czy klientowi – aby mogli sprawdzić aplikację w warunkach **zbliżonych do produkcji** (np. z realnym wydajnościowo kodem, realnymi uprawnieniami, ale jeszcze nie opublikowaną publicznie). Często na iOS profil preview korzysta z dystrybucji **TestFlight** (co też jest wewnętrzną dystrybucją Apple, choć formalnie odbywa się przez App Store Connect). Na Androidzie profil preview zwykle daje APK, bo na Google Play wewnętrzne testy można też robić inaczej.

Profil "production": to profil do budowania wersji **sklepowej**. Domyślnie w eas.json może być pusty, bo jeśli nie zdefiniujemy opcji, EAS i tak zbuduje wariant Release dla sklepu. Build produkcyjny **nie zawiera** narzędzi deweloperskich i jest gotowy do publikacji (na iOS podpisany certyfikatem dystrybucyjnym + profilem App Store, na Androidzie generujemy zwykle plik **.aab** do Google Play). **Production buildy są jedynymi, które trafiają do publicznych sklepów**. Mogą też służyć do TestFlight (beta testów w App Store) lub wewnętrznych testów Google Play, ale zazwyczaj do testów wystarczy profil preview.

Warto zauważyć, że profile w eas.json to tylko umowne nazwy – można je nazwać dowolnie i mieć więcej niż trzy. W dokumentacji Expo sugeruje ten podział na trzy typy, bo jest on powszechnie użyteczny, ale jeśli potrzebujemy np. osobnego profilu dla testów end-to-end, możemy go dodać. Profile mogą korzystać z mechanizmu **dziedziczenia** (extends), np. "preview": { "extends": "production", ... } żeby uniknąć duplikacji konfiguracji.

Przy konfiguracji profili warto też pamiętać o ustawieniach specyficznych dla platform: w eas.json można pod kluczem "android" lub "ios" w obrębie profilu ustawić np. "buildType": "apk" (Android – wymuszenie APK zamiast AAB) czy "simulator": true (iOS – build na symulator) itp.. Opcje wspólne dla obu platform (np. distribution, developmentClient) można dać na poziomie głównym profilu.

Uruchamianie buildów: Mając skonfigurowane profile, wywołujemy budowanie poleceniem eas build --profile <nazwa> --platform <android|ios> (lub --platform all dla obu jednocześnie). Jeśli pominiemy --profile, domyślnie użyty zostanie profil **production**. EAS CLI wyświetli logi na bieżąco i link do strony expo.dev z podglądem naszego builda. Można równocześnie śledzić postęp na stronie **Build Dashboard** expo (https://expo.dev/accounts/`{user}`/projects/`{project}`/builds). Po zakończeniu, otrzymamy informacje gdzie pobrać artefakt (plik .apk/.aab lub .ipa). EAS CLI potrafi też automatycznie poczekać na zakończenie builda (domyślnie) lub możemy przerwać i sprawdzić status później komendą eas build:list.

Podpisywanie i credenciales: Przy pierwszym buildzie EAS poprosi nas o dostarczenie lub wygenerowanie kluczy:

- **Android:** Keystore (plik .jks z kluczem prywatnym do podpisywania APK/AAB). EAS może wygenerować nowy lub użyć istniejącego (jeśli wcześniej korzystaliśmy z expo build:android, Expo ma go w chmurze).
- **iOS:** Certyfikat dystrybucyjny (.p12) + Provisioning Profile. EAS może zalogować się do naszego Apple Developer i automatycznie wygenerować wymagane certyfikaty i profile (wymagane podanie Apple ID, hasła lub klucza API). Jeśli wcześniej używaliśmy expo build:ios, EAS może pobrać te same profile/certyfikaty.

Wszystkie te wrażliwe dane EAS przechowuje zaszyfrowane na swoich serwerach, więc przy kolejnych buildach nie musimy już ich podawać ręcznie.

EAS Submit – wysyłanie aplikacji do sklepów

Po zbudowaniu aplikacji (zwłaszcza produkcyjnej) następnym krokiem jest publikacja w sklepie. **EAS Submit** to usługa/komenda która automatyzuje ten proces z linii poleceń. Pozwala wysłać gotowy plik .apk/.aab na Google Play lub .ipa na App Store Connect jednym poleceniem: eas submit --platform ios|android --profile <nazwa> (profil submit, inny niż build). Można skonfigurować w eas.json sekcję "submit" analogicznie do buildów, np. podać wrażliwe dane jak Apple ID, team ID, itp., aby nie wpisywać ich za każdym razem. EAS Submit wyśle binaria na serwery Expo, a stamtąd bezpośrednio do odpowiedniego sklepu, dzięki czemu możemy dokonać submitu nawet z systemu nie-macOS (np. z Windows czy Linux wysłać aplikację na App Store). To zmniejsza liczbę narzędzi do instalacji lokalnie i pozwala

integrację z CI (można np. z GH Actions zainicjować submit). W praktyce EAS Submit na Androidzie korzysta z **Google Play API** (wymaga wcześniej wygenerowania tokenu serwisowego w Google Play Console), a na iOS używa **Transporter** API Apple.

Warto zaznaczyć, że EAS Submit zakłada, iż wcześniej zbudowaliśmy aplikację odpowiednio: np. do `eas submit --platform ios` używamy paczki .ipa zbudowanej profilem produkcyjnym (podpisanej certyfikatem produkcyjnym), a do Google Play .aab. EAS CLI potrafi też automatycznie wziąć ostatni build produkcyjny i go wysłać, upraszczając kroki (flagą `--latest`).

Over-The-Air updates (OTA) z expo-updates i EAS Update

OTA (Over-The-Air) updates pozwalają **dostarczać użytkownikom aktualizacje aplikacji bez konieczności pobierania nowej wersji ze sklepu**. W przypadku Expo mówimy o aktualizacji kodu JavaScript i assetów przez internet, wykorzystując bibliotekę **expo-updates**. Dzięki temu możemy np. szybko rozpropagować **poprawki krytycznych bugów lub drobne zmiany UI** bez przechodzenia przez proces review w App Store/Google Play.

W **Managed Workflow** Expo tradycyjnie OTA odbywały się poprzez komendę `expo publish` i tzw. *release channels*. Obecnie, w ekosystemie EAS, mechanizm ten nazywa się **EAS Update** i opiera się o **kanały (channels)** oraz **wersje środowiska (runtime versions)**.

Konfiguracja OTA w projekcie: Gdy używamy EAS Build, biblioteka `expo-updates` jest automatycznie uwzględniona, ale musimy ją skonfigurować. Najprostszym sposobem jest uruchomienie `eas update:configure`. Ta komenda dodaje do naszego `app.json` (lub `app.config.js`) odpowiednie wpisy konfiguracyjne:

- **updates.url** – URL do serwera z aktualizacjami (domyślnie serwery Expo).
- **runtimeVersion** – wersja środowiska aplikacji, określająca kompatybilność natywną.

Ponadto `eas update:configure` zmodyfikuje plik **eas.json**, przypisując profile *preview* i *production* do domyślnych kanałów OTA (zwykle o tej samej nazwie co profile). W efekcie, każdy build wykonany z danym profilem będzie "nasłuchiwał" na konkretnym kanale aktualizacji. Np. profil **production** może mieć `"channel": "production"`, a profil **preview** `"channel": "staging"` (lub "preview") – oznacza to, że aplikacje zbudowane tym profilem będą otrzymywać aktualizacje publikowane na odpowiednich kanałach. Przykład fragmentu `eas.json` z ustawieniem kanałów dla profili:

```
{
  "build": {
    "production": {
      "channel": "production"
    },
    "preview": {
      "channel": "staging",
      "distribution": "internal"
    }
  }
}
```

Powyżej aplikacje produkcyjne odbiorą update'y z kanału **"production"**, a testowe z kanału **"staging"** (co umożliwia odseparowanie testowych aktualizacji od tych dla użytkowników produkcyjnych).

Publikacja OTA update: Aby wysłać aktualizację OTA, używamy komendy eas update --channel <nazwa> (dawniej expo publish, ale w nowym ekosystemie to jest alias do EAS Update). Przykładowo: eas update --channel production spakuje nasz aktualny kod JS i assety, wyśle je na serwer Expo i oznaczy jako najnowszą aktualizację dla kanału "production". Każda aplikacja, która została zbudowana z profilem przypisanym do "production", po uruchomieniu wykryje (o ile expo-updates jest włączone) nową paczkę i ją pobierze. Użytkownik dostanie ją **przy następnym uruchomieniu** (domyślnie expo-updates sprawdza na starcie i stosuje update przy kolejnym restarcie apki). Można to wymusić ręcznie w kodzie poprzez API (metody Updates.checkForUpdateAsync(), Updates.fetchUpdateAsync() i Updates.reloadAsync()), ale zazwyczaj nie jest to konieczne – ustawienie updates.checkAutomatically: "ON_LOAD" i updates.fallbackToCacheTimeout: 0 oznacza tryb natychmiastowego sprawdzania.

Kanały i branch'e: Kanał OTA to po prostu etykieta (string) identyfikująca strumień aktualizacji. Możemy dowolnie je nazywać (np. "production", "staging", "beta"). EAS Update wprowadza też pojęcie **branchy**, ale uprośćmy – branch to bardziej dla integracji z repozytorium kodu (można powiązać kanał z gałęzią git). Dla naszych potrzeb wystarczy wiedzieć, że **każdy build ma wpisany na sztywno kanał aktualizacji** (z eas.json), i możemy w razie potrzeby *przełączyć kanał dla danego builda* robiąc nowy build z innym kanałem.

Wersja środowiska (runtimeVersion): Ten parametr jest **kluczowy dla bezpieczeństwa aktualizacji OTA**. Określa on "wersję natywnego środowiska" naszej aplikacji. Jeśli zrobimy update OTA, który **nie pasuje do wersji natywnej** aplikacji, to aplikacja może się zepsuć lub crashować (np. update odwołuje się do natywnego modułu, którego nie ma w starej binarce). Dlatego zaleca się przy **każdej zmianie natywnej** (np. dołożeniu nowego modułu, zmianie SDK Expo) zmienić runtimeVersion na nowy (może to być np. numer wersji aplikacji lub jakiś hash). Expo sugeruje, by każdy **nowy release w sklepie miał unikalny runtimeVersion**, co gwarantuje, że OTA trafi tylko do kompatybilnych instancji aplikacji. Technicznie, runtimeVersion można ustawić jako string (np. "1.0.0") albo jako tożsamą z wersją aplikacji (coś jak "42" jeśli versionCode android i buildNumber iOS to 42). Jeśli nie ustawimy runtimeVersion, expo-updates może użyć fallbacku w postaci expo SDK version, ale przy EAS Update wymaga się definicji runtimeVersion.

Jak aplikacja odbiera OTA: Gdy użytkownik zainstaluje aplikację ze sklepu (czyli build zrobiony przez EAS), wbudowana biblioteka expo-updates będzie sprawdzać nasz serwer Expo. W konfiguracji updates mamy url oraz parametry check. Domyślnie expo-updates **pobiera update w tle przy starcie aplikacji** i zastosuje go przy kolejnym uruchomieniu. Można zmienić zachowanie – np. updates.fallbackToCacheTimeout ustawione na >0 spowoduje czekanie X ms na update przy pierwszym uruchomieniu zanim pokaże starą wersję (co wydłuża splash). Większość zostawia fallbackTimeout=0 co oznacza "pokaż od razu zcache'owaną wersję, a update dojdzie następnym razem". **Expo OTA jest bezpieczne** – jeśli urządzenie jest offline lub update nie pojawił się, aplikacja użyje wbudowanej wersji (zawartej w binarce).

Dev vs OTA: W trybie deweloperskim (Expo Go lub development build) mechanizm expo-updates jest wyłączony (app działa w trybie "dev server"). Dlatego warto testować OTA na buildach preview/production. Warto też pamiętać, że expo-updates **nie działa w Expo Go**, bo Expo Go może uruchomić dowolny projekt (nie przypisany do jednego kanału/runtime). Do testów OTA służą albo fizyczne buildy, albo narzędzie Expo **Orbit** (pozwala otworzyć link do update w dev-client, co symuluje OTA).

Podsumowując: **OTA updates w Expo** pozwalają nam *szybko reagować* na błędy i poprawki, szczególnie w warstwie JavaScript/zasobów. Dzięki **kanałom** możemy oddzielić aktualizacje dla testerskiej wersji od produkcyjnej. Należy jednak **uważać na kompatybilność natywną** – przy każdej zmianie wymagającej nowej binarki (np. dodanie modułu, zwiększenie minimalnej wersji OS, zmiana uprawnień) musimy wydać nową wersję w sklepie i zwykle także zmienić runtimeVersion, by stare instalacje nie pobrały niezgodnego kodu.

3. Wymogi publikacji w sklepach (Google Play i Apple App Store)

Publikacja aplikacji w oficjalnych sklepach wiąże się nie tylko z dostarczeniem pliku binarnego, ale też spełnieniem szeregu **wymagań formalnych i technicznych**. Poniżej omawiam kluczowe aspekty, na które trzeba zwrócić uwagę przygotowując aplikację Expo do wydania.

Uprawnienia i opisy w AndroidManifest oraz Info.plist

Android (AndroidManifest.xml i uprawnienia): W systemie Android wszystkie "niebezpieczne" uprawnienia (kamera, lokalizacja, mikrofon, itd.) muszą być zadeklarowane w manifeście aplikacji. W Expo jest to uproszczone – większość potrzebnych wpisów jest dodawana automatycznie przez odpowiednie biblioteki Expo podczas prebuilda. Np. jeśli używamy expo-camera, to config plugin tej biblioteki doda <uses-permission android:name="android.permission.CAMERA"/> do AndroidManifest. Zasadniczo **nie musimy ręcznie dodawać standardowych uprawnień**, chyba że potrzebujemy czegoś niestandardowego. Można wymusić dodatkowe uprawnienia przez wpis w app.json pod kluczem android.permissions (lista stringów nazw uprawnień). Tego używamy np. gdy jakaś biblioteka wymaga uprawnienia, które nie jest automatycznie dodawane – np. SCHEDULE_EXACT_ALARM w Androidzie 13+ dla dokładnych alarmów.

Jeśli chcemy **usunąć niepotrzebne uprawnienia** (bo np. jakaś lib dodała, a my nie chcemy by aplikacja prosiła o zgodę na coś, czego nie używamy), Expo pozwala w configu zablokować je poprzez android.blockedPermissions. To odpowiednik użycia w AndroidManifest atrybutu tools:node="remove" – Expo pod spodem to zastosuje. Przykład: blockedPermissions: ["android.permission.RECORD_AUDIO"] by usunąć dostęp do mikrofonu, jeśli np. expo-camera domyślnie go dodało, a my nie nagrywamy audio.

Należy pamiętać, że **Google Play weryfikuje zasadność uprawnień**. Jeśli aplikacja prosi o "niebezpieczne" permission (np. lokalizacja w tle, nagrywanie ekranu, SMSy itp.), może być wymagana dodatkowa deklaracja przy publikacji, a w skrajnych przypadkach (bez wyraźnego

uzasadnienia w opisie aplikacji) może zostać odrzucona. Dlatego upewnijmy się, że w sklepie w opisie lub sekcji **Privacy** wyjaśniamy dlaczego potrzebujemy określonych uprawnień i że faktycznie funkcjonalność aplikacji tego wymaga.

iOS (Info.plist i klucze użycia): Na platformie Apple, każda prośba o dostęp do wrażliwych zasobów (kamera, mikrofon, lokalizacja, kontakty, itp.) wymaga podania w pliku Info.plist tzw. **NS*UsageDescription** – czyli tekstowego uzasadnienia dla użytkownika. Przykładowo, by móc w ogóle wywołać `requestCameraPermissionsAsync()`, w Info.plist musi istnieć klucz **NSCameraUsageDescription** z wartością tłumaczącą po polsku/angielsku po co aplikacja chce użyć kamery. W przeciwnym razie Apple odrzuci aplikację podczas weryfikacji (lub aplikacja może się crashować przy odpaleniu uprawnienia). Expo automatycznie dodaje *domyślne komunikaty* dla wielu popularnych uprawnień poprzez config plugins danej biblioteki. Jednak te domyślne teksty są bardzo ogólne (po angielsku) i **Apple zaleca spersonalizować komunikaty** – inaczej reviewer może uznać je za niewystarczające. W Expo możemy z łatwością ustawić własne opisy dodając do `app.json` sekcję `ios.infoPlist` i tam klucze jak *NSCameraUsageDescription* z własnym stringiem. Np.:

```
{
  "expo": {
    "ios": {
      "infoPlist": {
        "NSCameraUsageDescription": "Aplikacja potrzebuje dostępu do aparatu, aby umożliwić skanowanie
kodów QR."
      }
    }
  }
}
```

Analogicznie dodajemy np. NSLocationWhenInUseUsageDescription, NSPhotoLibraryAddUsageDescription itd., zależnie od potrzeb. Dla wielu modułów expo istnieją też parametry pluginu config – np. `expo-media-library` pozwala wprost ustawić teksty dla dostępu do zdjęć poprzez `photosPermission`, `savePhotosPermission` zamiast ręcznie pisać klucze.

Ważne: Zmiany w Info.plist i AndroidManifest **nie mogą być dostarczane OTA** – to elementy natywne, muszą być zawarte w binarce przy wysyłce do sklepu. Dlatego planując nową wersję, sprawdźmy czy nie dodaliśmy biblioteki wymagającej nowego klucza w Info.plist – jeśli tak, to musimy zrobić nowy build i proces publikacji (OTA tu nie pomoże).

Podsumowując, przed wydaniem upewnijmy się, że:

- Android: manifest zawiera tylko niezbędne **uses-permissions**, nic nadmiarowego. Usuńmy ewentualne zbędne uprawnienia. Przy wypełnianiu formularza Google Play **Data Safety** wskażmy zgodnie z prawdą jakie dane/uprawnienia są wykorzystywane.
- iOS: w Info.plist są **wszystkie wymagane klucze NS...UsageDescription** dla funkcji, z których korzystamy. Teksty są konkretne i jasno tłumaczą użytkownikowi cel (Apple odrzuca np. "This app needs camera." jako zbyt lakoniczne – trzeba np. "Używamy kamery do skanowania kodów QR aby szybciej wprowadzić dane biletu").

Polityka prywatności, zgody użytkownika i zgodność z regulacjami (GDPR, ATT)

Polityka prywatności: Zarówno Apple, jak i Google wymagają, aby aplikacje, które gromadzą dane użytkowników (nawet anonimowo), posiadały **politykę prywatności**. W praktyce oznacza to:

- Musimy przygotować dokument **Privacy Policy** (np. hostowany na własnej stronie lub generowany przez generator, dostosowany do naszej apki).
- W **Google Play Console** w sekcji "App Content" trzeba podać URL do polityki prywatności. Dla aplikacji wymagających uprawnień wrażliwych (kamera, lokalizacja itp.) jest to obowiązkowe.
- W **App Store Connect** również możemy (a w niektórych przypadkach musimy) podać link do Privacy Policy. Dla aplikacji *z kont deweloperskich indywidualnych* nie zawsze jest wymagany link, ale Apple coraz bardziej to egzekwuje, zwłaszcza jeśli apka ma jakąkolwiek integrację z kontami, logowaniem, zbiera dane itp.

GDPR (RODO): Jeśli aplikacja operuje na rynku UE i przetwarza dane osobowe, powinniśmy być zgodni z RODO. W kontekście aplikacji mobilnej oznacza to np. uzyskanie zgody użytkownika na śledzenie/analizę (jeśli np. używamy Google Analytics, Amplitude itp.), udostępnienie opcji usunięcia konta/danych, poinformowanie jakie dane zbieramy. Google Play ma sekcję **Data Safety form**, gdzie deklarujemy kategorie danych i cel ich użycia – to jest pokazywane użytkownikom na stronie aplikacji. Trzeba to rzetelnie wypełnić przed publikacją.

App Tracking Transparency (ATT) na iOS: Jeżeli nasza aplikacja **śledzi użytkownika** w rozumieniu Apple (np. używa identyfikatora IDFA do celów reklamowych lub udostępnia dane osobowe firmom trzecim w celach reklamowych), to musimy implementować ramy ATT. Oznacza to:

- Dodanie do Info.plist klucza **NSUserTrackingUsageDescription** z uzasadnieniem (np. "Pozwól na użycie identyfikatora urządzenia aby otrzymywać spersonalizowane reklamy.").
- Wywołanie API **ATT** (AppTrackingTransparency) by poprosić użytkownika o zgodę requestTrackingPermissionsAsync() (dostępne w expo przez bibliotkę expo-tracking-transparency). Apple **wymaga** tego promptu, jeśli wykorzystujemy np. AdMob, Facebook SDK lub inne trackery. Bez tego zgoda jest domyślnie odmówiona.
- Jeśli użytkownik odmówi, musimy ograniczyć śledzenie (np. nie pobierać IDFA).

Expo udostępnia wspomniany moduł expo-tracking-transparency dla wygody. Wymaga on, jak wcześniej wspomniano, dodania NSUserTrackingUsageDescription. Można to zrobić ręcznie w ios.infoPlist, albo użyć config plugin tej biblioteki, który dokona wpisu za nas. Np. w app.json:

```
{
  "expo": {
    "plugins": [
      ["expo-tracking-transparency", {
        "userTrackingPermission": "Umożliwienie śledzenia aktywności pozwoli na wyświetlanie Ci
spersonalizowanych treści reklamowych."
      }]
```

```
    ]
  }
}
```

Powyższe spowoduje automatyczne dodanie klucza do Info.plist z podanym tekstem. **Apple bardzo skrupulatnie sprawdza ATT** – jeżeli korzystamy z jakiejkolwiek biblioteki reklam lub analiz, które *mogą* używać IDFA, a nie zaimplementujemy ATT, aplikacja zostanie odrzucona.

Zgody użytkownika w aplikacji: Poza systemowymi uprawnieniami i ATT, warto rozważyć czy nasza aplikacja nie potrzebuje własnych ekranów zgód. Np. jeśli zbieramy dane do celów analitycznych, dobrym zwyczajem (i w niektórych jurysdykcjach wymogiem) jest zapytać użytkownika przy pierwszym uruchomieniu czy wyraża na to zgodę (tzw. **cookie consent** analogicznie do web). Dotyczy to głównie UE. Nie jest to wymóg sklepu, ale element zgodności z prawem.

Przygotowanie zasobów graficznych i opisów aplikacji do publikacji

Oprócz samej paczki aplikacji, na **stronie sklepu** musimy dostarczyć zestaw materiałów:

- **Nazwa aplikacji:** Ustalona unikalna nazwa (do 50 znaków na Google Play, 30 znaków na iOS).
- **Opis aplikacji:** Krótki opis (tagline) i pełny opis (na Google Play do 4000 znaków). Ważne by jasno opisać funkcje appki i ewentualnie zawrzeć słowa kluczowe. Na iOS zamiast słów kluczowych jawnie w opisie, jest osobne pole "Keywords". Pamiętajmy o lokalizacjach językowych jeśli kierujemy do różnych rynków.
- **Ikona sklepu:** Dla Google Play potrzebny jest **icon** 512x512 px (PNG). Dla iOS – wykorzystywana jest ta sama ikona aplikacji 1024x1024, którą wgrywamy podczas przygotowania builda w Xcode (EAS robi to za nas).
- **Zrzuty ekranu (screenshots):** To często najwięcej pracy. Google Play wymaga co najmniej 2 screenshoty dla każdej obsługiwanej rozdzielczości (telefon, tablet 7", tablet 10", Android TV, Wear OS – zależnie od naszej aplikacji). iOS wymaga screenów dla każdej obsługiwanej wielkości ekranu: typowo 5.5" (iPhone 8), 6.5" (iPhone dużych rozmiarów), 12.9" iPad, itd., po **minimum 3 sztuki na każdy rozmiar**. W praktyce dla iPhone'ów najczęściej przygotowuje się zrzuty w rozdzielczości 1242x2688 (proporcje 6.5") – Apple zaakceptuje je też jako media dla innych rozmiarów jeśli się oznaczy, ale najlepiej mieć dla każdego. Zrzuty powinny atrakcyjnie prezentować aplikację (warto dodać opisy, ramki urządzeń – choć Apple odradza używanie urządzeń w obrazkach).
- **Feature graphic / promo graphic:** Google Play ma opcjonalną grafikę promocyjną (1024x500 px) oraz ewentualnie video (YouTube link). Apple nie ma takiej grafiki, ale ma opcję dołączenia traileru w App Store (tzw. App Preview Video).
- **Kategoria, tagi:** Wybieramy kategorie (np. Lifestyle, Education itp.) – obydwa sklepy to mają.
- **Rating wiekowy:** W Google Play wypełniamy formularz do przyznania kategorie wiekowej (PEGI itp.), Apple ma standardowe pytania (jak np. czy jest przemoc, hazard, itp.).
- **Formularz bezpieczeństwa danych:** Na Google Play *Data Safety Section* – tu deklarujemy jakie typy danych zbieramy (lokalizacja, kontakty, finansowe itd.) i czy są

one udostępniane. Trzeba to zrobić zgodnie ze stanem faktycznym i polityką prywatności.
- **In-App Purchases info:** Jeśli aplikacja ma płatności wewnątrz, trzeba je skonfigurować w sklepie (Apple i Google) i dodać informacje cenowe, SKU itd., oraz dostarczyć na Apple zrzuty ekranu ekranów zakupów do review. Jeśli appka jest płatna w całości, ustalamy cenę, waluty.
- **Kontakt dewelopera:** Oba sklepy wymagają podania kontaktu (email, strona wsparcia). Apple dodatkowo weryfikuje czy konto deweloperskie firmy jest powiązane z tożsamością firmy (dla kont osobistych nie dotyczy).
- **Testy:** Apple umożliwia TestFlight – do którego też warto przygotować opis co testować (tzw. TestFlight beta description) i zaprosić testerów. Google ma wewnętrzne, zamknięte i otwarte testy – tam też piszemy release notes.

Warto przygotować sobie wcześniej **pakiet assetów graficznych** (ikony, screenshoty) i opisy w wymaganych językach. Dzięki temu proces publikacji będzie płynny. Expo EAS nie automatyzuje tworzenia listingów w sklepie (poza samym wrzuceniem binares przez `eas submit`), więc tę część robimy ręcznie w konsolach deweloperskich sklepów.

Na koniec dodajmy, że **dobre praktyki** to:

- Upewnić się, że w aplikacji są *gdzieś* dostępne informacje o prywatności (np. link do polityki w ustawieniach).
- Jeśli aplikacja wymaga logowania, zadbać by proces rejestracji spełniał np. wytyczne Apple (logowanie poprzez Apple ID jest wymagane opcjonalnie, jeśli oferujemy logowanie np. Google/Facebook – tzw. Sign in with Apple requirement).
- Jeżeli korzystamy z kontentu generowanego przez użytkowników, musi być mechanizm moderacji/zgłaszania (Apple to często sprawdza).
- Jeśli aplikacja jest skierowana do dzieci, obowiązują specjalne restrykcje (COPPA etc., np. nie można mieć nieograniczonego trackingu, reklam targetowanych).

4. Monitorowanie i analityka w aplikacji

Po wypuszczeniu aplikacji ważne jest utrzymanie jej jakości i zrozumienie jak jest używana. Służą do tego **narzędzia monitorujące** błędy (crash reporting) oraz **analityka użytkowania**. W środowisku Expo mamy kilka opcji integracji takich narzędzi.

Raportowanie błędów: integracja z Sentry lub Firebase Crashlytics

Sentry: Sentry to popularna platforma do śledzenia błędów aplikacji produkcyjnych. Expo udostępnia oficjalny SDK **sentry-expo**, który ułatwia konfigurację Sentry w projektach Expo. Sentry pozwala rejestrować wszelkie nieobsłużone wyjątki JavaScript (i natywne crash'e również, przy odpowiedniej konfiguracji) wraz ze stacktrace, informacjami o urządzeniu, wersji appki itp.. Dzięki temu, gdy użytkownik napotka crash lub błąd, my otrzymamy powiadomienie i szczegóły, co ułatwia szybkie zidentyfikowanie i naprawę problemu.

Aby zintegrować Sentry:

1. Założyć konto i projekt w Sentry (free tier obsługuje ~5k zdarzeń miesięcznie).
2. Wygenerować DSN (adres projektu) i token uwierzytelniający dla uploadu sourcemapów.
3. W projekcie Expo zainstalować sentry-expo oraz uruchomić **Sentry Wizard**: npx @sentry/wizard -i reactNative -p expo (to polecenie automatycznie doda potrzebne zależności i dokona konfiguracji).
4. Po tym, wewnątrz kodu aplikacji importujemy i inicjalizujemy Sentry. Z sentry-expo zazwyczaj robi się to w pliku głównym App.js/tsx:

```
import * as Sentry from 'sentry-expo';
Sentry.init({
  dsn: 'https://<key>@o<org>.ingest.sentry.io/<project>',
  enableInExpoDevelopment: false,
  debug: false // można dać true podczas debugowania integracji
});
```

   sentry-expo automatycznie integruje się z błędami JS, a także natywnymi błędami w środowisku EAS (poprzez config plugin dodaje odpowiednie natywne ustawienia).

5. Kluczowe jest zapewnienie, by **mapy źródłowe (source maps)** były wysyłane do Sentry. Przy EAS Build jest to ułatwione – wystarczy ustawić w zmiennych środowiskowych builda SENTRY_AUTH_TOKEN oraz projekt/org slug, a EAS Build automatycznie uploaduje sourcemapy po kompilacji. (Sentry Wizard często robi to za nas, dodając np. w eas.json odniesienie do pluginu Sentry).

Po takim setupie, każde console.error i wyjątek w produkcji pojawi się w panelu Sentry jako **issue**. Możemy tam zobaczyć ile użytkowników doświadczyło błędu, na jakich urządzeniach, prześledzić stacktrace (zdekodowany dzięki sourcemap), a nawet zobaczyć tzw. **breadcrumbs** (ślad zdarzeń prowadzących do błędu). Sentry pozwala również logować ręcznie pewne rzeczy (np. catch error -> Sentry.Native.captureException(e)) i dodawać kontekst (np. identyfikator użytkownika, ostatnia wykonana akcja), by łatwiej debugować.

Nowością w ekosystemie Expo jest integracja Sentry z **EAS Insights** – można tak skonfigurować, aby crash reporty i nawet nagrania ekranu (session replay) z Sentry były widoczne w panelu Expo, co centralizuje monitorowanie. To wymaga jednak dodatkowej konfiguracji i znajduje się w nowszych wersjach expo SDK/EAS.

Firebase Crashlytics: Alternatywą do Sentry jest **Crashlytics** wchodzące w skład Firebase. Crashlytics jest bardzo popularne w świecie natywnym Android/iOS i oferuje podobne funkcjonalności (raporty crashy, ich agregacja, informacje o urządzeniach, *breadcrumbs*, kluczowe logi). Integracja Crashlytics w Expo wymaga użycia biblioteki **React Native Firebase** – konkretnie pakietu @react-native-firebase/crashlytics. W Expo (managed) możemy skorzystać z **config plugin** dostarczanego przez RNFirebase, aby dołączyć Crashlytics do aplikacji.

Kroki integracji Crashlytics:

- Dodać do projektu paczki @react-native-firebase/app oraz @react-native-firebase/crashlytics (zgodne z wersją RN naszego Expo SDK).
- W app.json dopisać plugin: "plugins": ["@react-native-firebase/crashlytics"] (oraz ewentualnie konfiguracje dodatkowe, np. ustawienie crashlytics_debug_enabled jeśli chcemy debugować w dev).
- Dodać google-services.json (Android) i GoogleService-Info.plist (iOS) pliki konfiguracyjne Firebase do projektu i uwzględnić je przez odpowiednie pola w app.json (Expo ma pluginy do automatycznego włączenia tych plików).
- Przebudować aplikację przez EAS, aby uwzględnić natywne SDK Crashlytics.

Po tym Crashlytics będzie automatycznie rejestrował natywne i JS crashy. W kodzie JS można wywołać crashlytics().log("...") lub wymusić testowy crash crashlytics().crash() by sprawdzić działanie. **Ważne:** Crashlytics zadziała dopiero na *release buildzie* odpalonym poza Expo Go (Expo Go nie ma tych natywnych modułów). W dev-buildach Crashlytics może działać, ale zwykle w trybie debug nie raportuje.

Zarówno Sentry jak i Crashlytics mogą działać równolegle, ale zazwyczaj wybieramy jedno. **Sentry** ma przewagę w integracji z JS (lepsze stacktrace'y z sourcemap, możliwość trackowania nie-crashowych błędów), natomiast **Crashlytics** jest często preferowany w firmach już używających Firebase (łatwa konsolidacja z Analytics, Perf Monitoring itp. w Firebase).

Zbieranie błędów i ich kontekstu

Nieważne jakie narzędzie wybierzemy, kluczowe jest, aby **monitorować błędy produkcyjne na bieżąco**. Warto skonfigurować alerty (np. Sentry może wysyłać maila/slacka gdy pojawi się nowy rodzaj crasha). Trzeba też pamiętać o **opisywaniu wersji aplikacji** – Sentry i Crashlytics grupują błędy per wersja, więc warto wysyłając build ustawić odpowiednio numer wersji (Sentry-expo robi to automatycznie, Crashlytics bazuje na versionCode/CFBundleVersion).

Dobrym zwyczajem jest również **logowanie ważnych zdarzeń jako breadcrumbs**. W Sentry breadcrumbs automatycznie obejmują np. zmiany ekranów (jeśli integrujemy z np. React Navigation), http requesty itp., ale możemy też manualnie dodać np. Sentry.Breadcrumb gdy użytkownik kliknie ważny przycisk. Crashlytics ma z kolei crashlytics().log(). To sprawia, że jak dostaniemy crash, to mamy pewien "kontekst akcji użytkownika" tuż przed.

Expo jako takie nie ma wbudowanego systemu logowania błędów do własnego serwisu (poza Metro bundler podczas dev). Dlatego integracja z zewnętrznym serwisem jest wskazana w aplikacjach produkcyjnych.

Analityka zdarzeń użytkownika (Expo Analytics, Amplitude, Firebase Analytics)

Dlaczego analityka? Chcemy wiedzieć jak użytkownicy korzystają z naszej aplikacji: które ekrany odwiedzają najczęściej, gdzie porzucają proces, ile czasu spędzają itp. To pozwala ulepszać UX i weryfikować hipotezy biznesowe.

Expo nie oferuje już własnego modułu analityki (dawniej był moduł expo-analytics-segment, teraz zalecane jest użycie rozwiązań firm trzecich). Do wyboru mamy m.in.:

- **Firebase Analytics:** jeśli i tak korzystamy z Firebase (np. Crashlytics), możemy dołożyć również Analytics. Integracja podobna – za pomocą `@react-native-firebase/analytics` i config pluginów. Firebase Analytics zbiera automatycznie sporo zdarzeń (otwarcie app, update, włącz/wyłącz, liczba użytkowników dziennych itp.), a własne zdarzenia wysyłamy przez `analytics().logEvent("event_name", {param: value})`. Plusem jest łatwa integracja z Crashlytics (dashboard wspólny).
- **Segment**: Segment to platforma agregująca analitykę – ma SDK (React Native Segment) które pozwala wysyłać zdarzenia, a dalej Segment przekazuje je do różnych usług (Amplitude, Mixpanel, Google Analytics etc.). Expo kiedyś miało wbudowany Segment, obecnie aby go użyć, można zainstalować paczkę `@segment/analytics-react-native` i użyć odpowiedniego pluginu. Segment sam w sobie nie daje UI do analityki (jest pośrednikiem).
- **Amplitude**: popularna analityka produktowa, z bogatym interfejsem do analizy lejków, retencji itp. Amplitude udostępnia SDK RN (które działa w Expo dev-build). Można też wysyłać eventy do Amplitude za pośrednictwem Segment (Segment miewa gotową integrację z Amplitude).
- **Expo kompatybilne lekkie SDK**: istnieją też nowsze, lekkie usługi analityczne które działają czysto po stronie JS i są przyjazne Expo. Np. **Aptabase**, **Astrolytics**, **PostHog** – według dokumentacji Expo działają nawet w Expo Go (bo używają tylko fetch, bez natywnego kodu). Ich zaletą jest brak konieczności konfigurowania natywnego modułu, wadą – czasem mniejsza moc analityczna niż giganty jak Firebase czy Amplitude.

Integracja analityki – przykładowy scenariusz: Załóżmy, że wybieramy Firebase Analytics. Co robimy?

- Dodajemy w `app.json` plugin `@react-native-firebase/analytics`.
- Dodajemy pliki google-services (jak przy Crashlytics).
- Po zbudowaniu, w kodzie możemy użyć:

```
import analytics from '@react-native-firebase/analytics';
analytics().logEvent('OpenedProductPage', { productId: '123' });
```

- Następnie w konsoli Firebase będziemy widzieć zdarzenia niestandardowe, możemy tworzyć raporty.

Jeśli wybierzemy Amplitude:

- Możemy użyć paczki expo-analytics-amplitude (jeśli istnieje dla nowego SDK – Expo kiedyś udostępniało, teraz można użyć oficjalnego @amplitude/analytics-react-native).
- Inicjalizujemy np. w App.js: Amplitude.init(API_KEY).
- Wysyłamy event: Amplitude.logEventAsync('event_name').

Expo Router a analityka: W aplikacjach Expo z Expo Router można zintegrować nawigację z analityką, np. logować event przy wczytaniu nowego ekranu.

Uwaga dot. Expo Go: Większość zaawansowanych SDK analitycznych wymaga natywnych modułów (Firebase, Segment, Amplitude RN). To oznacza, że działają one dopiero w **development build albo produkcyjnej aplikacji** – w Expo Go nie (Expo Go nie ma tych natywnych zależności). Wyjątkiem są usługi czysto JS (jak wspomniane Aptabase/PostHog) – te można nawet testować w Expo Go.

Prosty system analityki Expo: Jeśli nie chcemy integrować zewnętrznej platformy, możemy na własne potrzeby wysyłać zdarzenia do np. Google Analytics poprzez proste zapytania HTTP (GA4 ma Measurement Protocol). Jednak to wymaga trochę pracy i znajomości API.

Podsumowując, monitorowanie i analityka to **drugi etap** po wydaniu appki:

- **Crash reporting**: upewniamy się, że wiemy kiedy aplikacja się wysypuje na urządzeniach użytkowników (Sentry, Crashlytics).
- **Error tracking**: wyłapujemy także błędy nie kończące aplikacji ale istotne (np. nieudane wywołania API – można je też raportować do Sentry jako handled exceptions).
- **Analytics**: zbieramy zdarzenia (eventy) kluczowe dla naszego produktu, by móc analizować np. ścieżki użytkownika, popularność funkcji, skuteczność onboarding itp.

5. Wymagania techniczne sklepów (Android i iOS)

Sklepy mobilne narzucają również pewne **wymogi techniczne** dotyczące samych aplikacji – głównie minimalnych i docelowych wersji SDK oraz kompatybilności z nowymi platformami. W 2025 roku zwracamy uwagę na następujące kwestie:

Android

- **Wsparcie dla 16 KB page size (Android 15)**: Google ogłosiło, że od **1 listopada 2025** wszystkie nowe aplikacje i aktualizacje **muszą** być skompilowane z obsługą stron pamięci 16 KB. Android 15 wprowadza urządzenia z większym rozmiarem strony pamięci (16kB zamiast tradycyjnych 4kB) w celu poprawy wydajności na urządzeniach z większą ilością RAM. Praktycznie, dla dewelopera oznacza to konieczność użycia odpowiednio nowej wersji NDK/kompilatora do budowy natywnych bibliotek. W kontekście Expo – należy **zaktualizować Expo SDK i wszystkie biblioteki natywne do wersji wspierających 16KB**. Wielu dostawców już to zrobiło: React Native od wersji 0.72+ ma wsparcie, popularne SDK (Unity, Flutter) również. Jeśli tego nie

dopilnujemy, Google Play **odrzuci publikację** z komunikatem o braku wsparcia 16k (może to wykryć w skanie pakietu). Na szczęście Expo SDK 50+ bazuje na RN obsługującym 16k, ale trzeba uważać np. na użycie starszych bibliotek czy starszych wersji Expo. W razie problemu rozwiązaniem jest **upgrade expo / bibliotek natywnych** do wersji wydanych po wprowadzeniu tej zmiany (tj. połowa 2024 i później) lub ponowna kompilacja własnych modułów natywnych nowym NDK. Benefit jest taki, że kompatybilność 16k daje nam automatycznie pewien wzrost wydajności (szybsze uruchamianie, lepsza bateria na nowych urządzeniach)

- **targetSdkVersion:** Google Play co roku podnosi wymagany **docelowy poziom API** (target API level). W 2025 minimalny target dla nowych aplikacji to zapewne **API 34 (Android 14)** lub wyżej, a dla aktualizacji być może API 33 (Android 13) – konkrety zależą od komunikatów Google, ale z reguły: *od sierpnia 2023 wymagany target 33 dla nowych, od 2024 target 34*, itd. Expo SDK 52 domyślnie ustawia targetSdkVersion 34, co spełnia wymagania. W Expo można nadpisać targetSdk w app.json (plugin expo-build-properties), ale lepiej trzymać się defaultu Expo, bo starają się go aktualizować pod wymagania Play Store. **Brak spełnienia** – Google Play odrzuci upload z komunikatem "Your targetSdkVersion is X, which is lower than required Y".
- **minSdkVersion:** Google Play nie narzuca bardzo restrykcyjnie minimalnej wersji (minSdk), ale aplikacje z minSdk zbyt niskim mogą nie być dostępne dla nowszych urządzeń lub w ogóle nie można wysłać gdy jest dramatycznie niski. Expo obecnie wymaga min. Android 7.0 (SDK 24) dla SDK 50+, a tak naprawdę expo SDK 54 podniósł compileSdk do 36 (Android 14) i prawdopodobnie minSdk trzyma ~21 lub 24. Z tabelki expo wynika, że min Android to 7+ (czyli SDK 24) dla expo 52/53/54. To spełnia praktycznie wszystkie realia (obecnie <1% userów ma Android <7). Jeśli byśmy potrzebowali niżej, trzeba by modyfikować i budować samemu – raczej bez sensu. **Ważniejsze** jest **compileSdkVersion** – expo ustawia compileSdk równy targetSdk (np. 34), co jest ok.
- **Rozmiar aplikacji**: W kontekście sklepu może nas ograniczać *rozmiar paczki* – Google Play ma limit 150MB na APK/AAB. AAB mają mechanizm dynamiczny, więc rzadko to przekroczymy (chyba że gra z wieloma assetami). W razie czego, musielibyśmy używać **Expansion Files**, ale to raczej nie dotyczy typowej appki Expo.
- **Architektury 64-bit:** Od 2019 wymóg aby apki zawierały natywne biblioteki 64-bit (arm64-v8a). Expo od dawna generuje 64-bit, więc to spełniamy.
- **Google Play App Bundle:** Od 2021 nowe aplikacje muszą być publikowane jako .aab (Android App Bundle), nie .apk. EAS domyślnie dla production Android robi .aab, więc tu jesteśmy zgodni. Wersje testowe (profil preview) mogą być .apk, bo te nie trafiają do sklepów.
- **Inne:** Jeśli budujemy aplikację na **Android 13/14**, warto obsłużyć nowe uprawnienia jak *POST_NOTIFICATIONS* (dla powiadomień – inaczej app nie może wysyłać powiadomień bez zgody) i *Bluetooth* (teraz jest uprawnienie BLUETOOTH_CONNECT). Jeśli nas to dotyczy, trzeba dodać do manifestu i ogarnąć prośbę o uprawnienie w kodzie.

iOS

- **Wymóg najnowszego SDK Apple:** Apple wprowadza co roku wymóg, by od pewnej daty wszystkie appki były zbudowane przy użyciu minimum określonej wersji

Xcode/iOS SDK. W 2025 ogłosili, że od **kwietnia 2025** wszystkie aktualizacje muszą być **skompilowane z iOS 18 SDK (Xcode 16)**. W praktyce jeśli używamy EAS Build, Expo zadbało o aktualizację obrazów buildowych z Xcode 16.1+ żeby spełnić ten wymóg. Dla nas to znaczy: **upewnij się, że używasz Expo SDK kompatybilnego z Xcode 16** (co najmniej Expo SDK 51 lub 52). W przypadku starszej wersji, EAS Build i tak skompiluje pewnie w najnowszym Xcode, ale mogą być np. ostrzeżenia lub potrzeba zmiany minimalnego deployment targetu. Reguła Apple dotyczy *toolchain*, nie wymusza zmiany min iOS, ale w praktyce nowe Xcode mogą wymuszać pewne podbicie (np. Xcode 16 wymaga min iOS 12 lub 13).

- **Minimalna wersja iOS (deployment target):** Apple nie ma oficjalnie polityki minimalnej wspieranej wersji – można teoretycznie wspierać bardzo stare iOS, ale w praktyce Expo jak wspomniano ustala swoje minima. Expo SDK 50/51 miały min iOS ~13.0/13.4, natomiast od Expo SDK 52 minimalna wersja iOS to **15.1**. Jest to dość znaczący skok, podyktowany nowościami w RN i polityką Apple (coraz mniej urządzeń na iOS <15, a Apple wymagał od 2023, żeby nowe appki wspierały minimum iOS 12+, ale to luźny wymóg). Warto sprawdzić dokumentację expo dla konkretnej SDK, ale jeśli np. zbudujemy aplikację w Expo 52, to nie uruchomi się ona na iOS 14 i starszych. Dla większości to nie problem, bo użytkownicy i tak aktualizują (iOS 15 i wyżej to już >90% rynku). **Nie należy sztucznie obniżać deployment target**, bo biblioteki expo mogą korzystać z API niedostępnych w starszych systemach. Ogólnie, trzymamy się minimalnej wersji rekomendowanej przez Expo dla danej SDK.
- **App Store assets:** Po stronie technicznej iOS wymaga dostarczenia pewnych rzeczy:
    - **Ikony w Asset Catalogu:** Expo generuje z naszego 1024x1024 wszystkie potrzebne rozmiary ikon i pakuje w .ipa.
    - **Launch Storyboard (Splash):** od iOS 13 wymagany jest storyboard jako LaunchScreen (zamiast statycznego .png). Expo-splash-screen generuje taki storyboard z naszym logo i kolorem tła, więc spełniamy wymóg (inaczej Apple by odrzuciło za brak adaptacyjnego launch screen).
    - **Bitcode:** Apple zniosło wymóg bitcode w 2022, więc nie musimy nic robić (Expo i tak buduje bez bitcode).
    - **IPv6 networking:** Apple sprawdza, czy aplikacja działa w sieci IPv6 only. Warto upewnić się, że np. używane URL-e to domeny z AAAA rekordami lub obsłużyć to. (To bardziej dot. backendu).
    - **App Thinning:** .ipa zawiera Asset Catalogi, Expo to wspiera - obrazy są generowane w 3x/2x etc. Tutaj raczej ok.
- **TestFlight:** Przed publicznym release często wysyłamy appkę na TestFlight. Pamiętajmy, że od strony technicznej build na TestFlight musi być jakości produkcyjnej (ten sam build co potem do App Store). Apple recenzuje również buildy testowe (choć trochę szybciej i mniej rygorystycznie, ale np. kwestie uprawnień, crashy mogą spowodować odrzucenie nawet na etapie TestFlight Beta Review).
- **Rozmiar aplikacji i limity:** Apple ma limit rozmiaru appki pobieranej przez sieć komórkową (150MB, dawniej 200MB – ale to akurat mniej istotne bo user może na WiFi pobrać). Starajmy się by .ipa nie była ogromna – EAS Build i tak stara się wycinać nieużywany kod (tree shaking expo-modułów). Dla VR/AR aplikacji jeszcze dochodzą wymagania ARKit (w Info.plist usageDescription musiał być ARKit wpis jeśli ARKit jest używany). Dla appek korzystających z HealthKit czy innych specyficznych – też wymagane dodatkowe atrybuty i często testy.

- **Sdk Capabilities:** Jeżeli używamy pewnych usług Apple (np. Sign in with Apple, Apple Pay, Push Notifications, Background Modes) – musimy włączyć je w **Capabilities** aplikacji (Expo zazwyczaj ma pluginy do tego, np. expo-notifications dodaje automatycznie push entitlement). Przed submission sprawdźmy, czy np. nie wysyłamy pushy bez posiadania Push Notification entitlement – bo Apple by odrzuciło (Expo gdy wykryje expo-notifications, to doda go). Podobnie, jeśli nasza apka odtwarza audio w tle – musimy mieć background audio włączone w uprawnieniach.

Podsumowując sekcję wymogów technicznych:

- Android: **target SDK** zgodny z wymogiem (najlepiej najnowszy), **obsługa 16k page** – czyli używanie aktualnego toolchain (Expo 50+ to zapewnia), **arch. 64-bit**, odpowiednie **minSdk** (Expo default). To zwykle ogarnia EAS w tle.
- iOS: **zbudowana najnowszym Xcode** (EAS dba o to), **min iOS** na poziomie akceptowalnym (Expo 50+ aktualnie iOS 13/15, co jest ok), plus spełnienie wszystkich wymogów typu ATT, usage descriptions, itd., co omówiliśmy wcześniej.

6. Demo: od konfiguracji do publikacji – przykładowy proces

W tej części połączymy omówioną teorię w praktyczny przykład. Załóżmy, że mamy aplikację Expo, którą chcemy przygotować do wydania wersji 1.0. Pokażemy fragmenty konfiguracji i kroki wypuszczenia:

Konfiguracja pliku eas.json z profilami

Na potrzeby przykładu, skonfigurujmy eas.json następująco:

```
{
 "build": {
  "development": {
   "developmentClient": true,
   "distribution": "internal",
   "ios": {
    "simulator": true
   }
  },
  "preview": {
   "distribution": "internal",
   "channel": "staging"
  },
  "production": {
   "channel": "production"
  }
 },
 "submit": {
  "production": {
   "ios": {
    "appleId": "<nasz.apple@id.com>",
    "ascAppId": "<App Store Connect App ID>"
   },
```

```
    "android": {
      "serviceAccountKeyPath": "./google-play-credentials.json"
    }
   }
  }
}
```

Co tu zrobiliśmy:

- Mamy profil **development**: do szybkiego iterowania. Dodaliśmy ios.simulator: true aby wygenerować build działający w Simulatorze i **developmentClient: true** by móc w nim wczytywać projekty jak w Expo Go. Distribution "internal" – czyli nie do sklepu.
- Profil **preview**: build testowy dla naszego zespołu/klienta. Distribution "internal" (będziemy dystrybuować przez link/plik lub TestFlight ręcznie), bez dev-client (czyli pełen release build). Dodaliśmy "channel": "staging" – czyli wszystkie buildy preview będą odbierać ewentualne OTA z kanału "staging" (np. do testowania).
- Profil **production**: build na sklep. Channel "production" – production appki będą nasłuchiwać oficjalnych OTA. Tu distribution domyślnie jest "store" (bo nie podaliśmy internal).
- Sekcja **submit.production**: konfiguracja EAS Submit dla produkcji. Podaliśmy dane potrzebne do automatycznego submitu:
    - iOS: appleId (nasz Apple login) i ascAppId (App Store Connect app identifier – taki ciąg cyfr przypisany aplikacji, można znaleźć w App Store Connect; alternatywnie można podać appName i bundleIdentifier a EAS sam spróbuje wyszukać). Dodalibyśmy też appleTeamId jeśli to konto firmowe.
    - Android: ścieżka do pliku JSON z kluczem serwisowym do Google Play (ten plik trzeba pobrać z Play Console – zawiera email klienta, private key itd.). Zamiast path można dać zawartość w env.

Z takim plikiem możemy budować i od razu submitować.

Symulacja procesu build & release

1. **Przygotowanie projektu przed build:** Upewniamy się, że w app.json mamy poprawnie ustawione:
    - "version" (wersja aplikacji dla ludzi, np. "1.0.0") i odpowiednio "ios.buildNumber" oraz "android.versionCode" (te zwiększamy przy każdej publikowanej wersji).
    - Ikona (icon) i splash (jak w punkcie 1).
    - Uprawnienia Info.plist (np. NSCameraUsageDescription jeśli kamera) i Android permissions (jeśli coś custom).
    - runtimeVersion w sekcji updates (jeśli chcemy ręcznie nim zarządzać; jak nie, expo wygeneruje ze stringa w eas.json).
    - Jeśli korzystamy z config plugins (np. Sentry, Firebase), upewnić się że są w plugins i skonfigurowane.
2. **Build wersji testowej:** W trakcie developmentu budujemy często wersję dev:

```
eas build --profile development --platform ios
```

To wygeneruje nam build na iOS Simulator (bo tak ustawiliśmy) z expo-dev-client. Otwieramy go, testujemy. Potem:

```
eas build --profile preview --platform android
```

to da nam plik app.apk. Możemy go wysłać testerom lub zainstalować na urządzeniu lokalnie, albo użyć **EAS CLI** do udostępnienia (np. `eas build:list` daje link do pliku, który można przekazać). Na iOS, profil preview:

```
eas build --profile preview --platform ios
```

otrzymamy .ipa (Ad Hoc). Tę .ipa możemy wrzucić na TestFlight:

- Ręcznie: pobieramy .ipa, w Xcode -> Organizer -> Distribute App -> upload.
- Lub skorzystać z `eas submit --profile production --platform ios --latest` aby wysłać najświeższy build

Ewentualnie lepsze podejście: budujemy **production iOS** (co zrobi ipa dla App Store):

```
eas build --profile production --platform ios
```

a potem

```
eas submit --profile production --platform ios --latest
```

to wyśle do TestFlight (bo ascAppId podany, Apple automatycznie wrzuci to do TestFlight). W App Store Connect oznaczamy ten build jako dostępny w TestFlight dla testerów.

3. **Testy wewnętrzne:** Testerzy używają aplikacji. Dzięki temu możemy:
   - Zbierać feedback, crash logi (jeśli zintegrowaliśmy Sentry/Crashlytics, już zobaczymy ewentualne błędy pojawiające się u nich).
   - Wprowadzać poprawki. Poprawki krytyczne w JS możemy nawet deployować OTA na kanał "staging" (nasz preview channel) bez potrzeby nowego builda: `eas update --channel staging`. Tester po dwukrotnym uruchomieniu aplikacji dostanie update.
   - Gdy wszystko gra, przygotowujemy finalny release.
4. **Build produkcyjny:** Zakładamy, że po testach zwiększyliśmy numer wersji (np. z 1.0.0(1) na 1.0.0(2) albo 1.0.1). Teraz wykonujemy:

```
eas build --profile production --platform all --auto
```

(tu `--auto` powoduje automatyczne potwierdzanie kroków, np. wyboru certyfikatów). Po pewnym czasie mamy dwie paczki: *.aab* dla Androida i *.ipa* dla iOS (w chmurze expo).

EAS CLI wyświetli nam linki do plików oraz tzw. **build details page**. Na stronie expo.dev dla każdego builda możemy zobaczyć szczegóły: logi, rozmiary, użyte wersje SDK, jak również **czas kompilacji** i ewentualne warningi.

Jeśli chodzi o **podpisywanie**:

- Android: EAS wygenerował keystore lub użył naszego. Po buildzie warto zrobić eas credentials i pobrać kopię keystore (trzymać w bezpiecznym miejscu).
- iOS: EAS utworzył (lub użył) certyfikat i profil. Te profile mają ważność 1 rok (dla dystrybucji). EAS je odnowi jak będzie potrzeba przy kolejnym buildzie.

5. **Publikacja do sklepów:**
   - **Android (Google Play):** Możemy teraz wykonać:

```
eas submit --profile production --platform android --latest
```

     EAS użyje konfiguracji z eas.json (weźmie nasz plik google-play-credentials.json) i wrzuci AAB na nasz Google Play Console. Jeśli aplikacja jest nowa, trafi jako wersja robocza. Musimy tam zalogować się, uzupełnić *App Content* (privacy, rating etc.), i wystawić do review (np. na track "Production" od razu lub najpierw "Closed testing"). Często pierwszy release robi się w trybie stopniowym (staged rollout, np. 10% użytkowników).

   - **iOS (App Store):** Podobnie:

```
eas submit --profile production --platform ios --latest
```

     spowoduje upload .ipa do App Store Connect (ASC). Ponieważ podaliśmy ascAppId, EAS powiąże to z właściwą aplikacją. Build pojawi się w TestFlight (od razu lub w ciągu kilkunastu minut). Tutaj uwaga: Apple *zawsze* przeprowadza **Beta App Review** dla pierwszej wersji aplikacji zanim udostępni ją w TestFlight testerom zewnętrznym – może to potrwać dzień lub dwa. Gdy jesteśmy gotowi do publikacji, w App Store Connect tworzymy **New App Version** (np. 1.0.1), wybieramy nasz uploadnięty build, dodajemy opis co nowego, screenshoty, odpowiadamy na pytania (czy używa kryptografii, itd.) i wysyłamy do **App Review**.

6. **Zatwierdzenie i release:** Google Play automatycznie skanuje aplikację (może kilka godzin trwać) i zwykle aplikacja będzie dostępna w ciągu dnia (chyba że trafi do ręcznej weryfikacji, wtedy dłużej). Apple App Review trwa od kilku godzin do kilku dni. Po zatwierdzeniu, aplikacja pojawia się w App Store.
7. **Obsługa po publikacji (OTA i błędy):** Jeśli znajdziemy drobny błąd już po publikacji, i jest to błąd w kodzie JavaScript, możemy rozważyć wypuszczenie **OTA update** na kanał produkcyjny, zamiast robić od razu nowy build w sklepie. Przykład: odkryliśmy literówkę w tekście lub chcemy zmienić kolor przycisku – to idealne zadanie dla OTA. Wykonujemy:

```
eas update --channel production --message "Hotfix kolor przycisku"
```

   (opis to tylko meta dla nas). Użytkownicy dostaną tę poprawkę przy następnym uruchomieniu aplikacji (pamiętajmy jednak o **runtimeVersion** – OTA trafi tylko do tych instalacji, których runtimeVersion zgadza się z buildem na którym testowaliśmy).

Dla pewności można zrobić najpierw `eas update --channel staging` i przetestować na naszej wersji staging, a potem wypchnąć to samo na production.

Jeśli błąd jest poważny lub dotyczy warstwy natywnej (np. crashuje natywny moduł) – nie unikniemy wypuszczenia **nowej wersji w sklepach**. Wtedy robimy np. wersję 1.0.1, poprawiamy kod, zwiększamy numerki, budujemy przez EAS, i publikujemy tak jak wyżej.

8. **Monitorowanie:** Po wydaniu, obserwujemy w Sentry czy pojawiają się nowe błędy. Możemy skonfigurować Slack integration z Sentry, by dostawać powiadomienia o crashach. Również patrzymy na **feedback użytkowników** w sklepie – to często źródło informacji o problemach, których nie wykryliśmy (np. "apka nie działa na tabletach z Android 13" – co może wskazywać na specyficzny bug).
9. **Analityka:** Zbieramy dane analityczne – np. patrzymy w Firebase Analytics aktywnych użytkowników, czy używają nowej funkcji, jaki % kończy rejestrację. Na podstawie tego planujemy kolejne iteracje.

Struktura buildów, kanały OTA – co trafia do sklepu vs co aktualizujemy zdalnie

Na koniec wyjaśnijmy jeszcze, **co zawiera build sklepowy**, a co możemy zmieniać OTA:

- **Zawartość paczki binarnej (IPA/AAB):** zawiera wszystkie natywne kody (w tym wbudowane moduły expo), **bundel JavaScript** (skompilowany kod JS naszej aplikacji) oraz assety (obrazy, czcionki) potrzebne do działania. Ten bundle jest traktowany jako wersja bazowa aplikacji.
- **OTA update:** to nowy bundle JS + ewentualnie nowe assety (np. dodaliśmy obrazek – w update też się wyśle). OTA **nie może dodawać/zmieniać natywnego kodu**. Dlatego nie możemy OTA dodać np. obsługi nowego sensora urządzenia – to wymaga natywnej biblioteki i nowego builda. Możemy natomiast OTA zmieniać logikę, wygląd, teksty, itp., dopóki korzystamy z istniejących natywnych możliwości. Jeśli spróbujemy np. wywołać metodę natywnego modułu, który doszedł dopiero w kolejnej wersji aplikacji (i starą go nie ma), to nic się nie stanie albo dostaniemy błąd – stąd ten wymóg runtimeVersion, by odseparować takie przypadki.
- **Kanały OTA:** jak opisano, pozwalają mieć np. oddzielny strumień aktualizacji dla **development** (ew. nazywany "development", choć Expo Go i dev build i tak OTA nie używają), dla **preview/staging** do testów (żeby testerom móc wypuszczać eksperymentalne zmiany szybko), i **production** dla użytkowników.
- **Jak to się wiąże z profilami:** W naszym `eas.json` demo powyżej, profile production i preview były przypięte do konkretnych kanałów. Dzięki temu możemy wypuścić np. nową funkcję najpierw na *staging channel* – testerzy (mający build preview) dostaną ją OTA – a użytkownicy produkcyjni nie (bo inny kanał).
- **Struktura projektu OTA na serwerze Expo:** (opcjonalnie, by zrozumieć) – EAS Update organizuje publikacje w *branchach*, każda publikacja ma numerki (revisionID) i jest przypięta do jakiegoś channel i runtimeVersion. W aplikacji expo-updates natywnie zapisuje manifest OTA i pliki w pamięci urządzenia i decyduje czy je załadować.
- **Co trafia do sklepu:** do Google Play wysyłamy plik .aab – on zawiera naszą binarkę i poszczególne zestawy zasobów (Google Play sam wygeneruje z niego APK dla różnych

urządzeń). Do App Store wysyłamy .ipa – on jest podpisany i zawiera już wszystko (Apple zrobi z niego plik .app w swojej infrastrukturze).

- **Aktualizacje w sklepie:** Gdy robimy **większą aktualizację** (np. wersja 2.0 z nowymi natywnymi funkcjami), wypuszczamy nowy build przez sklepy. Użytkownik musi zaktualizować aplikację (ręcznie lub automatycznie jeśli ma włączone). Po zaktualizowaniu, aplikacja może też od razu pobrać nowszy OTA jeśli taki jest, ale zwykle po świeżej instalacji nie ma potrzeby – bo embedowany bundle pewnie jest najnowszy.

Testy end-to-end / automatyzacja: Jako ciekawostka, EAS umożliwia również uruchamianie testów end-to-end w chmurze po zbudowaniu (np. integracja z Maestro lub Detox), a także inne workflow (jak automatyczne wysyłanie OTA na każdy push do main, itp.). To jednak wykracza poza nasz zakres wykładu.

Literatura:

1. **https://docs.expo.dev/develop/user-interface/splash-screen-and-app-icon/** (Data dostępu: 1.10.2025) – Oficjalny przewodnik Expo dotyczący konfiguracji ikon aplikacji, wyjaśniający zasady tworzenia ikon adaptacyjnych (Adaptive Icons) dla systemu Android oraz wymagań dla iOS.
2. **https://docs.expo.dev/build/introduction/** (Data dostępu: 1.10.2025) – Kompletny przewodnik po usłudze EAS Build, omawiający proces tworzenia binarek produkcyjnych (.ipa, .aab) w chmurze oraz konfigurację pliku eas.json.
3. **https://docs.expo.dev/eas-update/introduction/** (Data dostępu: 1.10.2025) – Dokumentacja usługi EAS Update, wyjaśniająca mechanizm aktualizacji Over-The-Air (OTA), zarządzanie kanałami (channels) oraz wersjonowanie runtime.
4. **https://docs.expo.dev/build/setup/** (Data dostępu: 1.10.2025) – Instrukcja konfiguracji środowiska EAS CLI, niezbędna do autoryzacji projektów, zarządzania certyfikatami Apple/Google oraz automatyzacji procesu publikacji.