KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – Lecture 9

Building and Publishing (EAS) React Native Apps with Expo

Mateusz Pawełkiewicz

1.10.2025

1. Assets: App Icons and Splash Screens

App icons: Every mobile application needs**icons**, visible on the device's home screen and in stores. In Expo, we configure the icon in the configuration file (e.g.app.json). The easiest way is to prepare a 1024x1024 PNG image (square without rounded corners) and add it as a property"icon". Example of icon configuration inapp.json:

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

Expo (EAS Build) uses this single file to generate the required icon sizes for the iOS platform (different resolutions required). For Android, it's worth using the function**Adaptive Icon**– you can define the foreground and background layers separately, so that the icon looks good in various shapes imposed by the system. In the fileapp.jsonthere is a key for this"android":{"adaptiveIcon": ...}allowing you to indicate a separate foreground image (foregroundImage), monochrome (monochromeImage) and background color (backgroundColoror background image). For older Androids (without adaptive icons) you can optionally define"android.icon"– a single icon that combines the background and foreground layers.

On iOS, since Expo SDK 54, the format is supported**Icon Composer**– you can prepare a catalog.iconcontaining a set of icons generated e.g. with an Apple tool*Icon Composer*and insert it into the project, specifying the path inios.iconAlternatively (and in earlier SDKs), a single 1024x1024 PNG file is sufficient – Expo will generate the remaining sizes from it when building the app. Remember that the icon should fill the entire square and not contain any transparent areas or custom rounding (the system will add a mask automatically). You can also provide different icon variants on iOS (dark mode, light mode, tint mode) using an object instead of a path (in the keyios.icon), but most often one universal icon is enough.

Splash Screen: The splash screen is the first thing a user sees when launching an application – it appears while the application is loading. In Expo, the splash screen configuration is handled by the library.**expo-splash-screen**(automatically included). The splash screen configuration is also defined inapp.jsonvia the expo-splash-screen plugin, specifying, among others, the background and logo image. Example of configuration inapp.jsonusing the expo-splash-screen config plugin:

```
{
 "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
   }]
  ]
 }
}
```

In the above example we set the background color (backgroundColor), path to the image displayed on the splash (image), as well as an optional variant**dark mode**with a different image and background (dark.image i dark.backgroundColor). Key imageWidthallows you to adjust the size of the displayed image (in pixels). We can also specify properties for specific platforms separately.**Android** i **iOS**using the fields"android" i "ios"inside the plugin configuration. For example, on Android we can use a different image or a different scaling mode (resizeMode: contain, coverthenative) than on iOS.

Resolutions and splash format: It's best to prepare a splash image/logo in PNG format (Expo only supports PNG for splashes), with a transparent background if you want a separate background color. The recommended minimum image size is 1024x1024 for versatility, but larger background images are often used. Thanks to the config plugin, Expo will automatically set the required native assets in the Android Manifest and iOS LaunchScreen storyBoard during app build (using EAS Build) – in managed mode, there's no need to manually create files in Xcode/Android Studio.**Attention:**While testing the splash screen**we do not use Expo Go or development build**– Expo Go displays its own screen (Expo icon) instead of our splash, and dev-client also has its own splash, which can interfere with testing. Splash testing should be performed on preview or production builds.

Splash display time control: By default, the splash screen disappears automatically when the app is ready, but Expo allows you to control this manually via the API.SplashScreenfrom the libraryexpo-splash-screenWe can, for example,**extend the duration of the splash screen**to wait for important resources (e.g. data from the API) to be loaded. This is done using the methodSplashScreen.preventAutoHideAsync(), called right at the start of the application (before rendering). Then, when we are ready, we callSplashScreen.hideAsync()to hide the splash screen. It's important not to keep the splash screen on longer than necessary – best practice dictates showing the user the actual application interface as quickly as possible. Example (in the context of a React component):

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

// Prevent splash from automatically hiding:
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(); // hide splash when ready
    }
  }, [appReady]);

  if (!appReady) {
return null; // render nothing (splash remains)
  }

  return <MainAppContent />;
}
```

In this way**the splash screen will remain visible longer**untilappReadywill not change to true and we will not hide it manually.

Fade animation: From SDK 52 Expo supports animation options when transitioning from splash to application - you can set**fade**and duration of the animation. This is achieved using the methodSplashScreen.setOptions({ fade: true, duration: 1000 })which we can call before rendering the application.durationis the animation time in milliseconds (e.g. 1000ms = 1s). This option is optional - if we don't set it, the splash will disappear immediately after the application loads.

In summary, proper preparation of assets at Expo includes:**application icon**in the required resolutions (preferably 1024x1024 PNG as source, plus configuration inapp.json), **welcome screen**configured by expo-splash-screen (background and image adjusted to light/dark theme), as well as a possible**Splash display time control**via the expo-splash-screen API (if we need to delay the application start to load resources).

2. Expo Application Services (EAS): OTA builds and updates

Expo Application Services (**EAS**) is a set of Expo cloud services that streamline application development and distribution. EAS includes, among others:**EAS Build**(native IPA/AAB/APK cloud build service) and**EAS Submit**(automatic submission of builds to stores). EAS replaces the existing classic commandsexpo build:ios / expo build:android, offering more flexible and powerful capabilities.

EAS Build vs classic expo build

In the past, Expo offered**expo build**for applications in**managed workflow**, which allowed building apk/ipa without Xcode/Android Studio configuration.**EAS Build**is the successor and extension of this service - it enables the construction*each*React Native applications (both managed Expo and bare React Native) in the cloud. Thanks to EAS Build, we can use native libraries outside the standard Expo set – EAS generates the so-called*development clients*the*custom builds*which contain these native modules. In practice, this means that if we used to have to "eject" from Expo to use some library (e.g. Bluetooth, WebRTC, In-App Payments), we can now stay in Expo and include native modules via**config plugins**and build apk/ipa via EAS. EAS Build generates**smaller binaries**containing only the necessary native modules (which reduces the size of the application).

Key differences between the oldexpo build a eas build:

- **Project type support:** expo buildonly worked for Expo managed projects.**EAS Build works for all RN projects**, including those with their own native code or config plugins.
- **Build profiles:** expo buildhad default settings; EAS Build allows you to define different**building profiles** (development, preview, production) w pliku **eas.json**– more on that in a moment.
- **Internal distribution:**EAS natively supports*internal distribution*– easy sharing of test builds outside the store. On iOS, this is solved by automatically creating a provisioning profile like*To This*(no need to manually add device UDIDs, EAS can generate an ad-hoc profile). On Android, EAS allows convenient file generation**.apk**for testing (instead of just*.aab*).
- **CI/CD integration:**EAS CLI is designed for use in pipelines (JSON commands, webhooks, etc.), making automation easy.expo buildit was rather an interactive tool.
- **Credentials management:**EAS stores and automatically manages keys and certificates. If we've done this before,expo build, we can use the same certificates/keystore in EAS – the tool will automatically detect existing data on Expo servers. EAS can also automatically generate Android keystores or iOS certificates and provisioning profiles during the first build, guiding the developer through an interactive setup.
- **Building on the latest SDK:**Expo has stopped developingexpo buildand announced its expiration in 2023, so new requirements (e.g. architectures, new Xcode/SDK) are supported in EAS. For example, EAS provides up-to-date build images with new Xcode in line with Apple's requirements (something the oldexpo buildwouldn't have gotten it if it wasn't updated).

To sum up: **EAS Build**is a modern CI/CD service from Expo that**simplifies the mobile application compilation process**– from raw JavaScript code to a finished artifact (.apk, .aab, .ipa) in the store or for testing. This allows developers to focus on writing code, while offloading the burden of compilation to the Expo cloud.

EAS Build Requirements and Configuration (eas.json, profile)

To use EAS Build, we need to create an Expo account (free) and install**EAS CLI**(at least v3+). Log in to the CLI with the commandeas login, and then we perform a one-time project configuration with the commandeas build:configureThis command will generate a fileeas.jsonin the main project folder with an example build profile configuration.

By default, Expo creates three build profiles ineas.json: **development**, **preview** i **production**Each profile is a set of settings that determine how the application is to be built. An example of a defaulteas.jsoncreated by Expo:

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

Profil "development":intended for development builds. It has a setting"developmentClient": truewhich means that the application will be included**expo-dev-client**– so our build will work similarly to Expo Go, allowing the project to be loaded during development. Such a build*development*includes developer tools (debug menu etc.) and**Never**It is not shipped to stores. Additionally,"distribution": "internal"means that the build will be prepared for internal distribution (tests) – e.g., on iOS, an Ad Hoc .ipa file will be generated (available for upload to test devices, but not to the App Store). On Androiddistribution: internalwill build APK (instead of AAB) for easy installation directly.

*Attention:*For iOS, you can also do builds on dev-builds**Simulator**(Mac). Settingios.simulator: truein the profile will cause EAS to generate an application in .app format to run in iOS Simulator instead of .ipa. Often a separate profile is created, e.g."development-simulator"if we need both variants (physical device vs simulator).

"Preview" profile:intermediate profile –**test build**resembling production, but not intended for the store. No longer includes expo-dev-client (nodevelopmentClientin configuration), so it's a normal application without debug menu etc., but still withdistribution: "internal"i.e. for testing distribution (APK or iOS Ad Hoc/TestFlight). We use it to share versions with testers, the team, or the client – so they can test the application under conditions**close to production**(e.g. with real performance code, real permissions, but not yet publicly released). Often on iOS, the preview profile uses the distribution**TestFlight**(which is also Apple's internal distribution, although formally it's done through App Store Connect). On Android, the preview profile usually provides an APK, because on Google Play, internal testing can also be done differently.

Profil "production":is a profile for building versions**shop**. Default ineas.jsonIt can be empty, because if we don't define the option, EAS will build the Release variant for the store anyway. Production build**does not contain**development tools and is ready for publication (on iOS signed with a distribution certificate + App Store profile, on Android we usually generate a file**.aab** do Google Play). **Production builds are the only ones that go to public stores**They can also be used for TestFlight (App Store beta testing) or internal Google Play testing, but a preview profile is usually sufficient for testing.

It is worth noting that the profiles ineas.jsonThese are just conventions – you can name them anything you want and have more than three. The Expo documentation suggests this division into three types because it is universally useful, but if we need, for example, a separate profile for end-to-end testing, we can add it. Profiles can use the mechanism**inheritance** (extends), e.g. "preview": { "extends": "production", ... }to avoid duplication of configuration.

When configuring profiles, it is also worth remembering platform-specific settings:eas.jsonyou can lock it up"android"the"ios"within the profile, set e.g."buildType": "apk"(Android – force APK instead of AAB) or"simulator": true(iOS – build for simulator) etc.. Options common to both platforms (e.g.distribution, developmentClient) can be given at the main profile level.

Running builds: With the profiles configured, we trigger the build with the commandeas build --profile <nazwa> --platform <android|ios>(or--platform allfor both at the same time). If we omit--profile, the profile will be used by default**production**The EAS CLI will display live logs and a link to the expo.dev website with a preview of our build. You can also track progress on the website.**Build Dashboard**expo (https://expo.dev/accounts/`{user}`/projects/`{project}`/builds). Once completed, we will receive information on where to download the artifact (.apk/.aab or .ipa file). EAS CLI can also automatically wait for the build to finish (by default) or we can stop and check the status later with the commandeas build:list.

Signing and credenciales: During the first build, EAS will ask us to provide or generate keys:

- **Android:**Keystore (.jks file with private key for signing APK/AAB). EAS can generate a new one or use an existing one (if we used it before)expo build:android, Expo has it in the cloud).
- **iOS:**Distribution Certificate (.p12) + Provisioning Profile. EAS can log in to our Apple Developer account and automatically generate the required certificates and profiles (requires Apple ID, password, or API key). If we have previously usedexpo build:ios, EAS can download the same profiles/certificates.

EAS stores all this sensitive data encrypted on its servers, so we no longer have to enter it manually in subsequent builds.

EAS Submit – sending applications to stores

Once you have built your app (especially a production one), the next step is publishing it to the store.**EAS Submit**is a service/command that automates this process from the command line. It allows you to send a finished .apk/.aab file to Google Play or .ipa file to App Store Connect with a single command:eas submit --platform ios|android --profile <nazwa>(submit profile, different from build). Can be configured ineas.jsonsection"submit"Similarly to builds, e.g., provide sensitive data such as Apple ID, team ID, etc., so as not to enter them every time. EAS Submit will send binaries to Expo servers, and from there directly to the appropriate store, so you can submit even from a non-macOS system (e.g., from Windows or Linux, submit the app to the App Store). This reduces the number of tools to be installed locally and allows integration with CI (you can, for example, initiate submit with GH Actions). In practice, EAS Submit on Android uses**Google Play API**(requires generating a service token in Google Play Console beforehand) and on iOS it uses**Transporter** API Apple.

It is worth noting that EAS Submit assumes that we have previously built the application accordingly: e.g.eas submit --platform ioswe use the .ipa package built with the production profile (signed with a production certificate), and for Google Play .aab. EAS CLI can also automatically take the last production build and send it, simplifying the steps (with the flag--latest).

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

OTA (Over-The-Air) updatesallow**provide users with application updates without having to download a new version from the store**In the case of Expo, we're talking about updating JavaScript code and assets over the internet, using a library**expo-updates**Thanks to this, we can, for example, quickly promote**critical bug fixes or minor UI changes**without going through the review process in the App Store/Google Play.

IN **Managed Workflow**Expo traditionally took place OTA through the commandexpo publishand the so-called*release channels*Currently, in the EAS ecosystem, this mechanism is called**EAS Update**and is based on**channels** and **runtime versions**.

OTA configuration in the project: When we use EAS Build, the libraryexpo-updatesis automatically included, but we need to configure it. The easiest way is to runeas update:configureThis command adds to ourapp.json(orapp.config.js) relevant configuration entries:

- **updates.url**– URL to the update server (Expo servers by default).
- **runtimeVersion**– application environment version, specifying native compatibility.

In addition eas update:configurewill modify the file**eas.json**, assigning profiles*preview* i *production*to the default OTA channels (usually with the same name as the profiles). As a result, each build made with a given profile will "listen" to a specific update channel. For example, the profile**production**may have"channel": "production", the profile**preview** "channel": "staging"(or "preview") – this means that applications built with this profile will receive updates published on the appropriate channels. Example snippeteas.jsonwith channel settings for profiles:

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

Above, production apps will receive updates from the channel**"production"**, and test from the channel**"staging"**(which allows for separation of test updates from those for production users).

OTA update release: To send an OTA update, we use the commandeas update --channel <name>(formerlyexpo publish, but in the new ecosystem it is an alias to EAS Update). For example:eas update --channel productionwill package our current JS code and assets, send them to the Expo server, and mark them as the latest update for the "production" channel. Any app built with a profile assigned to "production" will detect the new package upon launch (if expo-updates is enabled) and download it. The user will receive it**next time you start**(by default, expo-updates checks at startup and applies the update at the next app restart). This can be forced manually in the code via API (methodsUpdates.checkForUpdateAsync(), Updates.fetchUpdateAsync() i Updates.reloadAsync()), but it is usually not necessary – settingupdates.checkAutomatically: "ON_LOAD" i updates.fallbackToCacheTimeout: 0means instant check mode.

Channels and branches: An OTA channel is simply a label (string) identifying the update stream. We can name them whatever we want (e.g., "production," "staging," "beta"). EAS Update also introduces the concept**branchy**, but let's simplify - branch is more for integration with the code repository (you can associate a channel with a git branch). For our purposes, it is enough to know that**each build has a hardcoded update channel** (With eas.json), and we can if necessary*switch channel for a given build*making a new build with a different channel.

Runtime version (runtimeVersion): This parameter is**crucial for OTA update security**It determines the "native environment version" of our application. If we perform an OTA update, which**does not match the native version**application, the application may break or crash (e.g. update refers to a native module that is not present in the old binary). Therefore, it is recommended to**any native change**(e.g. adding a new module, changing the Expo SDK) change the runtimeVersion to a new one (it can be e.g. the application version number or some hash). Expo suggests that each**the new release in the store had a unique runtimeVersion**, which ensures that the OTA only reaches compatible application instances. Technically, runtimeVersion can be set as a string (e.g."1.0.0") or as the same as the application version (something like "42" if versionCode android and buildNumber iOS are 42). If we don't set runtimeVersion, expo-updates can use the fallback of expo SDK version, but EAS Update requires a runtimeVersion definition.

How the app receives OTA: When a user installs an app from the store (i.e., a build made by EAS), the built-in expo-updates library will check our Expo server.updates We have urland check parameters. By default, expo-updates**downloads updates in the background when the application starts**and will apply it on the next startup. You can change the behavior – e.g.updates.fallbackToCacheTimeoutset to >0 will cause it to wait X ms for an update on first run before showing the old version (which prolongs the splash). Most leave fallbackTimeout=0 which means "show the cached version immediately, and the update will happen next time".**Expo OTA is safe**– if the device is offline or the update has not appeared, the application will use the built-in version (contained in the binary).

Dev vs OTA: In developer mode (Expo Go or development build), the expo-updates mechanism is disabled (the app runs in "dev server" mode). Therefore, it is worth testing OTA on preview/production builds. It is also worth remembering that expo-updates**doesn't work in Expo Go**, because Expo Go can run any project (not assigned to a single channel/runtime). For OTA testing, either physical builds or the Expo tool are used**Orbit**(allows you to open an update link in dev-client, which simulates an OTA).

To sum up: **OTA updates w Expo**allow us*react quickly*for bugs and fixes, especially in the JavaScript/resource layer. Thanks**channels**we can separate updates for the test version from the production version. However, it is necessary**be careful about native compatibility**– for every change that requires a new binary (e.g. adding a module, increasing the minimum OS version, changing permissions) we have to release a new version in the store and usually also change the runtimeVersion to prevent old installations from downloading incompatible code.

3. Store publication requirements (Google Play and Apple App Store)

Publishing an application in official stores involves not only providing a binary file, but also fulfilling a number of requirements**formal and technical requirements**Below, I discuss key aspects to consider when preparing your Expo app for release.

Permissions and descriptions in AndroidManifest and Info.plist

Android (AndroidManifest.xml and permissions): In Android, all "dangerous" permissions (camera, location, microphone, etc.) must be declared in the application manifest. In Expo, this is simplified – most of the necessary entries are added automatically by the appropriate Expo libraries during prebuild. For example, if we useexpo-camera, then the config plugin of this library will add<uses-permission android:name="android.permission.CAMERA"/>to AndroidManifest. Basically**we don't have to manually add standard permissions**, unless we need something non-standard. You can enforce additional permissions by enteringapp.jsonunder lock and keyandroid.permissions(a string list of permission names). This is used, for example, when a library requires a permission that is not automatically added – e.g.SCHEDULE_EXACT_ALARMin Android 13+ for accurate alarms.

If we want**remove unnecessary permissions**(because e.g. some lib added and we don't want the application to ask for permission for something we don't use), Expo allows you to block them in the config byandroid.blockedPermissionsThis is equivalent to using the attribute in AndroidManifest.tools:node="remove"– Expo below will apply this. Example:blockedPermissions: ["android.permission.RECORD_AUDIO"]to remove access to the microphone if, for example,expo-cameraadded it by default and we don't record audio.

Please note that**Google Play verifies the validity of permissions**If an app requests "dangerous" permissions (e.g., background location, screen recording, SMS, etc.), an additional declaration may be required upon publication, and in extreme cases (without a clear justification in the app description), it may be rejected. Therefore, make sure that in the store description or section**Privacy**we explain why we need specific permissions and that the application's functionality actually requires them.

iOS (Info.plist and usage keys): On the Apple platform, each request to access sensitive resources (camera, microphone, location, contacts, etc.) requires the so-called**NS*UsageDescription**– that is, a textual justification for the user. For example, to be able to callrequestCameraPermissionsAsync(), there must be a key in

Info.plist**NSCameraUsageDescription**with a value explaining in Polish/English why the app wants to use the camera. Otherwise, Apple will reject the app during verification (or the app may crash when the permission is triggered). Expo automatically adds*default messages*for many popular permissions via the config plugins of a given library. However, these default texts are very general (in English) and**Apple recommends personalizing messages**– otherwise the reviewer may consider them insufficient. In Expo, we can easily set our own descriptions by addingapp.jsonsectionios.infoPlistand there the keys as*NSCameraUsageDescription*with your own string. For example:

```
{
  "expo": {
    "ios": {
      "infoPlist": {
"NSCameraUsageDescription": "The app needs access to the camera to scan QR codes."
      }
    }
  }
}
```

Similarly, we add, for example, NSLocationWhenInUseUsageDescription, NSPhotoLibraryAddUsageDescription, etc., as needed. Many expo modules also have config plugin parameters – e.g.expo-media-libraryallows you to directly set texts for accessing photos viaphotosPermission, savePhotosPermissioninstead of writing keys by hand.

Important: Changes to Info.plist and AndroidManifest**cannot be delivered OTA**– these are native components and must be included in the binary when shipped to the store. Therefore, when planning a new version, check whether we've added a library that requires a new key in the Info.plist. If so, we'll need to perform a new build and release process (OTA won't help here).

In summary, before releasing, let's make sure that:

- Android: Manifest contains only the necessary**uses-permissions**, nothing redundant. Let's remove any unnecessary permissions. When filling out the Google Play form**Data Safety**Let us truthfully indicate what data/permissions are used.
- iOS: In Info.plist there are**all required NS keys...UsageDescription**for the functions we use. The texts are specific and clearly explain the purpose to the user (Apple rejects, for example, "This app needs a camera." as too laconic – it should be, for example, "We use the camera to scan QR codes to enter ticket data faster").

Privacy Policy, User Consent and Regulatory Compliance (GDPR, ATT)

Privacy Policy: Both Apple and Google require that apps that collect user data (even anonymously) have**privacy policy**In practice, this means:

- We need to prepare a document**Privacy Policy**(e.g. hosted on your own website or generated by a generator adapted to our app).

- IN **Google Play Console**In the "App Content" section, you must provide the URL to the privacy policy. For apps requiring sensitive permissions (camera, location, etc.), this is mandatory.
- IN **App Store Connect**We can also (and in some cases must) provide a link to the Privacy Policy. For applications*from individual developer accounts*a link is not always required, but Apple is increasingly enforcing it, especially if the app has any integration with accounts, logins, collects data, etc.

GDPR (REFERENCE): If an app operates in the EU market and processes personal data, we should be GDPR compliant. In the context of a mobile app, this means, for example, obtaining user consent for tracking/analysis (if we use Google Analytics, Amplitude, etc.), providing the option to delete the account/data, and informing what data we collect. Google Play has a section**Data Safety form**, where we declare the data categories and the purpose of their use – this is displayed to users on the application page. This must be completed accurately before publication.

App Tracking Transparency (ATT) on iOS: If our application**tracks the user**as defined by Apple (e.g., uses the IDFA for advertising purposes or shares personal data with third parties for advertising purposes), then we must implement the ATT framework. This means:

- Adding a key to Info.plist**NSUserTrackingUsageDescription**with justification (e.g. "Allow the use of your device identifier to receive personalized ads.").
- API call**TO**(AppTrackingTransparency) to ask the user for consent`requestTrackingPermissionsAsync()`(available at the expo through the library`expo-tracking-transparency`). Apple **requires**this prompt if we use, for example, AdMob, Facebook SDK or other trackers. Without it, consent is denied by default.
- If the user declines, we must limit tracking (e.g., not collect IDFA).

Expo provides the mentioned module`expo-tracking-transparency`for convenience. It requires, as previously mentioned, adding NSUserTrackingUsageDescription. This can be done manually in`ios.infoPlist`, or use the config plugin of this library, which will make the entry for us. For example, in`app.json`:

```
{
  "expo": {
    "plugins": [
      ["expo-tracking-transparency", {
"userTrackingPermission": "Allowing activity tracking will allow us to show you personalized advertising content."
      }]
    ]
  }
}
```

The above will automatically add the key to Info.plist with the text provided.**Apple checks ATT very carefully**– if we use any advertising or analytics library that*they can*use IDFA and we do not implement ATT, the application will be rejected.

In-app user consents: Beyond system permissions and ATT, it's worth considering whether your app needs its own consent screens. For example, if you're collecting data for analytical purposes, it's good practice (and in some jurisdictions, a requirement) to ask the user for consent upon first launch (so-called opt-in consent).**cookie consent**analogous to the web). This mainly applies to the EU. This is not a requirement of the store, but an element of legal compliance.

Preparing graphic resources and application descriptions for publication

In addition to the application package itself,**store website**we need to provide a set of materials:

- **Application name:**Established unique name (up to 50 characters on Google Play, 30 characters on iOS).
- **Application description:**A short description (tagline) and a full description (up to 4,000 characters on Google Play). It's important to clearly describe the app's functions and, if necessary, include keywords. On iOS, instead of explicit keywords in the description, there's a separate "Keywords" field. Remember to consider language localization if you're targeting different markets.
- **Shop icon:**For Google Play you need**icon**512x512 px (PNG). For iOS, the same 1024x1024 app icon is used, which we upload during build preparation in Xcode (EAS does this for us).
- **Screenshots:**This is often the most work. Google Play requires at least 2 screenshots for each supported resolution (phone, 7" tablet, 10" tablet, Android TV, Wear OS - depending on our app). iOS requires screenshots for each supported screen size: typically 5.5" (iPhone 8), 6.5" (large-sized iPhone), 12.9" iPad, etc.,**minimum 3 pieces for each size**In practice, screenshots are most often prepared for iPhones at a resolution of 1242x2688 (6.5" aspect ratio). Apple will also accept them as media for other sizes if you specify them, but it's best to have them for everyone. The screenshots should present the app attractively (it's worth adding descriptions and device frames – although Apple advises against using devices in images).
- **Feature graphic / promo graphic:**Google Play has an optional promotional image (1024x500 px) and optional video (YouTube link). Apple doesn't have such an image, but it does have the option to include a trailer in the App Store (called an App Preview Video).
- **Category, tags:**We choose categories (e.g. Lifestyle, Education, etc.) – both stores have it.
- **Age rating:**In Google Play we fill out a form to assign an age rating (PEGI, etc.), Apple has standard questions (such as whether there is violence, gambling, etc.).
- **Data security form:**Google Play*Data Safety Section*– here we declare what types of data we collect (location, contacts, financial, etc.) and whether it is shared. This must be done in accordance with the facts and the privacy policy.
- **In-App Purchases info:**If the app has in-app payments, you'll need to configure them in the app store (Apple and Google) and add pricing information, SKUs, etc., and provide Apple with screenshots of the purchase screens for review. If the app is fully paid, we'll set the price and currencies.

- **Developer contact:**Both stores require contact information (email, support page). Apple also verifies that the company's developer account is linked to the company's identity (this does not apply to personal accounts).
- **Tests:**Apple offers TestFlight, where it's also worth preparing a description of what to test (the so-called TestFlight beta description) and inviting testers. Google has internal, closed, and open tests, where we also write release notes.

It's worth preparing in advance**graphic asset package**(icons, screenshots) and descriptions in the required languages. This will make the publishing process smooth. Expo EAS does not automate the creation of listings in the store (apart from just uploading binaries viasubmit them), so we do this part manually in the store developer consoles.

Finally, let us add that**good practices** to:

- Make sure you have the following in your app:*somewhere*available privacy information (e.g. link to policy in settings).
- If the application requires logging in, ensure that the registration process meets, for example, Apple's guidelines (logging in via Apple ID is required optionally if we offer logging in, e.g., Google/Facebook – the so-called Sign in with Apple requirement).
- If we use user-generated content, there must be a moderation/reporting mechanism (Apple checks this frequently).
- If the app is aimed at children, special restrictions apply (COPPA etc., e.g. you can't have unlimited tracking, targeted advertising).

4. In-app monitoring and analytics

Once an app is released, it's important to maintain its quality and understand how it's used.**monitoring tools**errors (crash reporting) and**usage analytics**In the Expo environment, we have several options for integrating such tools.

Bug Reporting: Integration with Sentry or Firebase Crashlytics

Sentry: Sentry is a popular platform for tracking bugs in production applications. Expo provides an official SDK.**sentry-expo**, which simplifies the configuration of Sentry in Expo projects. Sentry allows you to log any unhandled JavaScript exceptions (and native crashes as well, if configured appropriately) along with the stacktrace, device information, app version, etc. This way, when a user encounters a crash or error, we receive notification and details, making it easier to quickly identify and fix the problem.

To integrate Sentry:

1. Create an account and project in Sentry (free tier supports ~5k events per month).

2. Generate a DSN (project address) and authentication token for uploading sourcemaps.
3. In the Expo project, installsentry-expoand run**Sentry Wizard**: npx @sentry/wizard -i reactNative -p expo(this command will automatically add the necessary dependencies and configure it).
4. After that, inside the application code we import and initialize Sentry.sentry-expoThis is typically done in the main App.js/tsx file:

```
import * as Sentry from 'sentry-expo';
Sentry.init({
  dsn: 'https://<key>@o<org>.ingest.sentry.io/<project>',
  enableInExpoDevelopment: false,
debug: false // can be set to true when debugging integration
});
```

sentry-expoautomatically integrates with JS errors as well as native errors in the EAS environment (via config plugin adds appropriate native settings).

5. It is crucial to ensure that**source maps**were sent to Sentry. With EAS Build, this is made easier – just set it in the build environment variablesSENTRY_AUTH_TOKENand project/org slug, and EAS Build automatically uploads sourcemaps after compilation. (Sentry Wizard often does this for us, e.g., by adding a reference to the Sentry plugin in eas.json).

After such a setup, eachconsole.errorand the exception in production will appear in the Sentry panel as**issue**There we can see how many users experienced the error, on which devices, follow the stacktrace (decoded thanks to sourcemap), and even see the so-called**breadcrumbs**(a trace of the events leading to the error). Sentry also allows you to log things manually (e.g., catch error -> Sentry.Native.captureException(e)) and add context (e.g., user ID, last action performed) to make debugging easier.

New to the Expo ecosystem is the integration of Sentry with**EAS Insights**– Crash reports and even screen recordings (session replays) from Sentry can be configured to appear in the Expo dashboard, centralizing monitoring. However, this requires additional configuration and is available in newer versions of the Expo SDK/EAS.

Firebase Crashlytics: An alternative to Sentry is**Crashlytics**part of Firebase. Crashlytics is very popular in the native Android/iOS world and offers similar functionalities (crash reports, their aggregation, device information,*breadcrumbs*, key logs). Integrating Crashlytics in Expo requires the use of the library**React Native Firebase**– specifically the package@react-native-firebase/crashlyticsIn Expo (managed) we can use**config plugin**provided by RNFirebase to include Crashlytics in your application.

Crashlytics integration steps:

- Add to package design@react-native-firebase/app and @react-native-firebase/crashlytics(compatible with the RN version of our Expo SDK).
- IN app.jsonadd plugin:"plugins": ["@react-native-firebase/crashlytics"](and possibly additional configurations, e.g. settingcrashlytics_debug_enabledif we want to debug in dev).

- Add google-services.json (Android) and GoogleService-Info.plist (iOS) Firebase configuration files to your project and include them through the appropriate fields inapp.json(Expo has plugins to automatically include these files.)
- Rebuild the app via EAS to include the native Crashlytics SDK.

After that, Crashlytics will automatically log native and JS crashes. In your JS code, you can callcrashlytics().log("...")or force a test crashcrashlytics().crash()to check the operation.**Important:**Crashlytics will only work on*release buildzie*launched outside of Expo Go (Expo Go doesn't have these native modules). Crashlytics may work in dev builds, but it usually doesn't report anything in debug mode.

Both Sentry and Crashlytics can run in parallel, but we usually choose one.**Sentry**has an advantage in integration with JS (better stacktraces from sourcemap, ability to track non-crash errors), while**Crashlytics**is often preferred in companies already using Firebase (easy consolidation with Analytics, Perf Monitoring, etc. in Firebase).

Collecting errors and their context

No matter what tool we choose, the key is to**monitor production errors on an ongoing basis**It's worth setting up alerts (e.g. Sentry can send an email/Slack message when a new type of crash occurs). You should also remember about**describing the application version**– Sentry and Crashlytics group errors per version, so it is worth setting the version number accordingly when submitting the build (Sentry-expo does it automatically, Crashlytics relies on versionCode/CFBundleVersion).

It is also a good practice**logging important events as breadcrumbs**In Sentry, breadcrumbs automatically include e.g. screen changes (if we integrate with e.g. React Navigation), http requests, etc., but we can also manually add e.g.Sentry.Breadcrumbwhen a user clicks an important button. Crashlytics, on the other hand, hascrashlytics().log()This means that when we get a crash, we have some "user action context" right before it.

Expo itself doesn't have a built-in error logging system for its own service (outside of Metro bundler during development). Therefore, integration with an external service is recommended for production applications.

User Event Analytics (Expo Analytics, Amplitude, Firebase Analytics)

Why analytics?We want to know how users use our app: which screens they visit most often, where they abandon the process, how much time they spend, etc. This allows us to improve UX and verify business hypotheses.

Expo no longer offers its own analytics module (the former was expo-analytics-segment, now recommending third-party solutions). Options include:

- **Firebase Analytics:**If we use Firebase anyway (e.g. Crashlytics), we can also add Analytics. The integration is similar – using@react-native-firebase/analyticsand plugin config. Firebase Analytics automatically collects a lot of events (app opening, update, turn on/off, number of daily users, etc.), and we send our own events viaanalytics().logEvent("event_name", {param: value})The advantage is easy integration with Crashlytics (common dashboard).

- **Segment**: Segment is an analytics aggregation platform – it has an SDK (React Native Segment) that allows you to send events, and then Segment forwards them to various services (Amplitude, Mixpanel, Google Analytics, etc.). Expo used to have a built-in Segment, but now you can install a package to use it.@segment/analytics-react-nativeand use the appropriate plugin. Segment itself does not provide a UI for analytics (it is an intermediary).
- **Amplitude**: popular product analytics, with a rich interface for analyzing funnels, retention, and more. Amplitude provides an RN SDK (which runs in Expo dev-build). You can also send events to Amplitude via Segment (Segment has a ready-made integration with Amplitude).
- **Expo compatible lightweight SDK**: there are also newer, lightweight analytical services that run purely on the JS side and are Expo-friendly. For example:**Aptabase**, **Astrolytics**, **PostHog**– according to the Expo documentation, they even work in Expo Go (because they only use fetch, without native code). Their advantage is that there's no need to configure a native module, but their disadvantage is that they sometimes have less analytical power than giants like Firebase or Amplitude.

Analytics integration – sample scenario: Let's say we choose Firebase Analytics. What do we do?

- We add inapp.json plugin @react-native-firebase/analytics.
- We add google-services files (as with Crashlytics).
- Once built, in the code we can use:

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

- Then in the Firebase console we will see custom events and we can create reports.

If we choose Amplitude:

- We can use the packageexpo-analytics-amplitude(if it exists for the new SDK - Expo used to provide it, now you can use the official one@amplitude/analytics-react-native).
- We initialize e.g. in App.js:Amplitude.init(API_KEY).
- We send the event:Amplitude.logEventAsync('event_name').

Expo Router and analytics: In Expo applications with Expo Router, you can integrate navigation with analytics, e.g. log an event when a new screen loads.

Note regarding Expo Go: Most advanced analytics SDKs require native modules (Firebase, Segment, Amplitude RN). This means they only work in**development build or production application**– not in Expo Go (Expo Go doesn't have these native dependencies). The exception are pure JS services (like the aforementioned Aptabase/PostHog) – these can even be tested in Expo Go.

Expo's Simple Analytics System: If we don't want to integrate an external platform, we can send events to Google Analytics, for example, via simple HTTP requests (GA4 has the Measurement Protocol). However, this requires some work and API knowledge.

In summary, monitoring and analytics are**second stage**after the app release:

- **Crash reporting**: we make sure we know when the application crashes on user devices (Sentry, Crashlytics).
- **Error tracking**: we also catch errors that do not terminate the application but are important (e.g. failed API calls – they can also be reported to Sentry as handled exceptions).
- **Analytics**: we collect key events for our product to analyze e.g. user paths, feature popularity, onboarding effectiveness, etc.

5. Technical requirements of stores (Android and iOS)

Mobile stores also impose certain**technical requirements**regarding the applications themselves – primarily the minimum and target SDK versions and compatibility with new platforms. In 2025, we are focusing on the following issues:

Android

- **Support for 16 KB page size (Android 15)**: Google announced that from**November 1, 2025**all new apps and updates**they have to**be compiled with support for 16KB memory pages. Android 15 introduces devices with a larger memory page size (16KB instead of the traditional 4KB) to improve performance on devices with more RAM. Practically, for the developer, this means the need to use an appropriately new version of the NDK/compiler to build native libraries. In the context of Expo – it is necessary**update Expo SDK and all native libraries to versions that support 16KB**Many vendors have already done this: React Native from version 0.72+ has support, popular SDKs (Unity, Flutter) as well. If we don't make sure of this, Google Play**will reject publication**with a message about the lack of 16k support (this can be detected in the package scan). Fortunately, Expo SDK 50+ is based on RN supporting 16k, but you need to be careful, for example, about using older libraries or older versions of Expo. In case of a problem, the solution is**upgrade expo / native libraries**to versions released after this change (i.e. mid-2024 and later) or recompiling your native modules with the new NDK. The benefit is that 16k compatibility automatically gives us some performance boost (faster startup, better battery life on new devices).
- **targetSdkVersion:**Google Play raises the required level every year**target API level**(target API level). In 2025, the minimum target for new applications will probably be**API 34 (Android 14)**or higher, and for updates perhaps API 33 (Android 13) – the specifics depend on Google's announcements, but as a rule:*from August 2023, target 33 required for new entrants, from 2024 target 34*, etc. Expo SDK 52 defaults to targetSdkVersion 34, which meets the requirements. In Expo, you can override targetSdk in`app.json`(plugin expo-build-properties), but it's better to stick to the default Expo, because they try to update it to meet Play Store requirements.**Lack**

of fulfillment – Google Play odrzuci upload z komunikatem "Your targetSdkVersion is X, which is lower than required Y".

- **mySdkVersion:**Google Play doesn't impose a very strict minimum version (minSdk), but apps with a minSdk that's too low may not be available for newer devices or may not be shipped at all if it's dramatically low. Expo currently requires at least Android 7.0 (SDK 24) for SDK 50+, but in fact, Expo SDK 54 raised the compileSdk to 36 (Android 14), and the minSdk probably holds ~21 or 24. The Expo table shows that the minimum Android version is 7+ (i.e., SDK 24) for Expo 52/53/54. This meets practically all realities (currently <1% of users have Android <7). If we needed anything lower, we'd have to modify and build it ourselves – rather pointless.**More important** Is **compileSdkVersion**– expo sets compileSdk equal to targetSdk (e.g. 34), which is ok.
- **Application size**: In the context of the store, it may limit us*package size*– Google Play has a 150MB limit for APK/AAB. AABs are dynamic, so we rarely exceed this limit (unless it's a game with many assets). In case of such a limit, we would have to use**Expansion Files**, but this probably doesn't apply to a typical Expo app.
- **64-bit architectures:**Since 2019, apps have been required to include native 64-bit libraries (arm64-v8a). Expo has been generating 64-bit for a long time, so we're meeting that requirement.
- **Google Play App Bundle:**As of 2021, new apps must be published as .aab (Android App Bundle), not .apk. EAS defaults to .aab for production Android, so we agree on that. Test versions (preview profiles) can be .apk, as those don't make it to app stores.
- **Other:**If we are building an application on**Android 13/14**, it is worth handling new permissions as*POST_NOTIFICATIONS*(for notifications – otherwise the app cannot send notifications without consent) and*Bluetooth*(now there is a BLUETOOTH_CONNECT permission). If this applies to us, we need to add it to the manifest and handle the permission request in the code.

iOS

- **Latest Apple SDK Requirement:**Apple introduces a requirement every year that, from a certain date, all apps must be built using at least a certain version of Xcode/iOS SDK. In 2025, they announced that from**April 2025**all updates must be**compiled with iOS 18 SDK (Xcode 16)**In practice, if we use EAS Build, Expo has taken care of updating the build images from Xcode 16.1+ to meet this requirement. For us, this means:**make sure you are using the Expo SDK compatible with Xcode 16**(at least Expo SDK 51 or 52). In the case of an older version, EAS Build will probably compile in the latest Xcode, but there may be warnings or the need to change the minimum deployment target. Apple's rule applies*toolchain*, does not force a change of iOS min, but in practice new Xcodes may force some boosting (e.g. Xcode 16 requires min iOS 12 or 13).
- **Minimum iOS version (deployment target):**Apple does not officially have a policy of minimum supported version – theoretically you can support very old iOS, but in practice, as mentioned, Expo sets its own minimums. Expo SDK 50/51 had minimum iOS ~13.0/13.4, while from Expo SDK 52 the minimum iOS version is**15.1**This is a significant leap, dictated by new features in the RN and Apple's policy (fewer and

fewer devices are running iOS <15, and Apple required new apps to support at least iOS 12+ from 2023, but this is a loose requirement). It's worth checking the Expo documentation for a specific SDK, but if, for example, you build an app in Expo 52, it won't run on iOS 14 and earlier. For most, this isn't a problem, as users update anyway (iOS 15 and above already accounts for >90% of the market).**Do not artificially lower the deployment target**, because Expo libraries can use APIs not available in older systems. Generally, we stick to the minimum version recommended by Expo for a given SDK.

- **App Store assets:**On the technical side, iOS requires certain things to be provided:
    - **Icons in Asset Catalog:**Expo generates all the necessary icon sizes from our 1024x1024 and packs them into .ipa.
    - **Launch Storyboard (Splash):**Since iOS 13, a storyboard is required as a LaunchScreen (instead of a static .png). Expo-splash-screen generates such a storyboard with our logo and background color, so we meet the requirement (otherwise Apple would reject it for the lack of an adaptive launch screen).
    - **Bitcode:**Apple removed the bitcode requirement in 2022, so we don't have to do anything (Expo is building without bitcode anyway).
    - **IPv6 networking:**Apple verifies that the app works on an IPv6-only network. It's worth making sure, for example, that the URLs you're using are domains with AAAA records, or supporting that. (This is more of a backend issue.)
    - **App Thinning:**.ipa contains Asset Catalogs, Expo supports it - images are generated in 3x/2x etc. Here it's probably ok.
- **TestFlight:**Before public release, we often submit the app to TestFlight. It's important to remember that, from a technical perspective, the TestFlight build must be production-quality (the same build that will be submitted to the App Store). Apple also reviews test builds (although somewhat faster and less rigorously, but issues such as permissions or crashes can result in rejection even at the TestFlight Beta Review stage).
- **Application size and limits:**Apple has a size limit for apps downloaded over a cellular network (150MB, previously 200MB – but that's less important because users can download over WiFi). Try to keep the .ipa file small – EAS Build tries to trim unused code (tree-shaking expo-modules) anyway. For VR/AR apps, there are also ARKit requirements (Info.plist usageDescription required an ARKit entry if ARKit is used). Apps using HealthKit or other specific apps also require additional attributes and often tests.
- **Sdk Capabilities:**If we use certain Apple services (e.g. Sign in with Apple, Apple Pay, Push Notifications, Background Modes) – we need to enable them in**Capabilities**applications (Expo usually has plugins for this, e.g., expo-notifications automatically adds push entitlement). Before submitting, check that, for example, we're not sending push messages without having Push Notification entitlement – because Apple would reject it (Expo, when it detects expo-notifications, will add it). Similarly, if our app plays audio in the background, we need to have background audio enabled in the permissions.

To summarize the technical requirements section:

- Android: **target SDK**compliant with the requirement (preferably the newest),**16k page support**– i.e. using the current toolchain (Expo 50+ ensures this),**arch. 64-bit**, appropriate **mySdk**(Expo default). This usually runs EAS in the background.
- iOS: **built with the latest Xcode**(EAS dba o to),**my iOS**at an acceptable level (Expo 50+ currently iOS 13/15, which is ok), plus meeting all requirements such as ATT, usage descriptions, etc., which we discussed earlier.

6. Demo: From Configuration to Publishing – Sample Process

In this section, we'll combine the theory we've discussed with a practical example. Let's say we have an Expo application that we want to prepare for release 1.0. We'll show you some configuration and release steps:

Configuring the eas.json file with profiles

For the sake of example, let's configure`eas.json`as follows:

```
{
  "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"
      }
    }
  }
}
```

What we did here:

- We have a profile**development**: for fast iteration. We added`ios.simulator: true`to generate a build that works in Simulator and**developmentClient: true**to be able to load projects in it like in Expo Go. Distribution "internal" - i.e. not to the store.

- Profile**preview**: test build for our team/client. Distribution "internal" (we will distribute via link/file or TestFlight manually), without dev-client (i.e. full release build). We've added"channel": "staging"– i.e. all preview builds will receive any OTAs from the "staging" channel (e.g. for testing).
- Profile**production**: build for store. Channel "production" – production apps will listen for official OTAs. Here, distribution is "store" by default (because we didn't specify internal).
- Section**submit.production**: EAS Submit configuration for production. We have provided the data needed for automatic submission:
    - iOS: appleId(our Apple login) andascAppId(App Store Connect app identifier – such a string of numbers assigned to the application can be found in App Store Connect; alternatively, you can enterappName i bundleIdentifierand EAS will try to search for it itself). We would also addappleTeamIdif it's a business account.
    - Android: path to a JSON file with the Google Play service key (this file must be downloaded from the Play Console – it contains the client's email, private key, etc.). Instead of path, you can provide the contents of the env.

With such a file we can build and submit immediately.

Build & release process simulation

1. **Pre-build project preparation:**We make sure that inapp.jsonwe have set it correctly:
    - "version"(human app version, e.g. "1.0.0") and accordingly"ios.buildNumber" and "android.versionCode"(we increase these with each published version).
    - Icon (icon) and splash (as in point 1).
    - Info.plist permissions (e.g. NSCameraUsageDescription if camera) and Android permissions (if something custom).
    - runtimeVersionin the updates section (if we want to manage it manually; if not, expo will generate it from the string in eas.json).
    - If you are using config plugins (e.g. Sentry, Firebase), make sure they are inpluginsand configured.
2. **Test build:**During development, we often build a dev version:

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

This will generate a build for the iOS Simulator (because that's how we configured it) from expo-dev-client. Open it and test it. Then:

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

This will give us an app.apk file. We can send it to testers or install it locally on the device, or use**EAS CLI**to share (e.g.eas build:listgives a link to a file that can be uploaded.) On iOS, profile preview:

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

we will get .ipa (Ad Hoc). We can upload this .ipa to TestFlight:

- Manually: download .ipa, in Xcode -> Organizer -> Distribute App -> upload.
- Or useeas submit --profile production --platform ios --latestto send the latest build

A possibly better approach: we build**production iOS**(what ipa will do for app store):

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

but then

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

it will send it to TestFlight (because the ascAppId is provided, Apple will automatically upload it to TestFlight). In App Store Connect, we mark this build as available in TestFlight for testers.

3. **Internal tests:**Testers use the app. This allows us to:
    - Collect feedback, crash logs (if we have integrated Sentry/Crashlytics, we will already see any errors appearing in them).
    - Make fixes. We can even deploy critical JS fixes OTA to the staging channel (our preview channel) without the need for a new build:eas update --channel stagingAfter running the application twice, the tester will receive the update.
    - When everything is in order, we prepare the final release.
4. **Production build:**We assume that after testing we have increased the version number (e.g. from 1.0.0(1) to 1.0.0(2) or 1.0.1). Now we execute:

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

(you--autocauses automatic confirmation of steps, e.g. selecting certificates). After some time, we have two packages:.*aab*for Android and.*violence*for iOS (in the expo cloud).

EAS CLI will display links to files and the so-called**build details page**On the expo.dev website we can see the details for each build: logs, sizes, SDK versions used, as well as**compilation time**and possible warnings.

When it comes to **signing**:

- Android: EAS generated a keystore or used ours. After build, it's worth doingeas credentialsand download a copy of the keystore (keep it in a safe place).
- iOS: EAS created (or used) a certificate and profile. These profiles are valid for 1 year (for distribution). EAS will renew them as needed in the next build.

5. **Publication to stores:**
    - **Android (Google Play):**We can now do:

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

EAS will use the configuration from eas.json (it will take our filegoogle-play-credentials.json) and upload the AAB to our Google Play Console. If the application is new, it will be posted as a draft version. We need to log in there, complete*App Content*(privacy, rating, etc.), and submit for review (e.g., on the "Production" track immediately or first on "Closed testing"). Often, the first release is made in a gradual mode (staged rollout, e.g., 10% of users).

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

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

will upload the .ipa to App Store Connect (ASC). Since we providedascAppId, EAS will associate it with the correct application. The build will appear in TestFlight (immediately or within a few minutes). Note: Apple*Always*carries out**Beta App Review**for the first version of the app before making it available in TestFlight to external testers – this may take a day or two. When we are ready for publication, we create a test in App Store Connect**New App Version**(e.g. 1.0.1), select our uploaded build, add a description of what's new, screenshots, answer questions (does it use cryptography, etc.) and send it to**App Review**.

6. **Approval and release:**Google Play automatically scans the app (it can take several hours), and the app will typically be available within a day (unless it goes to manual review, which takes longer). Apple App Review takes anywhere from a few hours to a few days. Once approved, the app appears in the App Store.
7. **Post-release support (OTA and errors):**If we find a minor bug after publication, and it is a JavaScript bug, we may consider releasing it.**OTA update**to the production channel, instead of immediately creating a new build in the store. For example, we discovered a typo in the text or we want to change the color of a button – this is the perfect task for an OTA. We perform:

```
eas update --channel production --message "Hotfix button color"
```

(the description is just meta for us). Users will get this fix the next time they launch the app (but remember**runtimeVersion**– The OTA will only be available for those installations whose runtimeVersion matches the build we tested on.) To be sure, you can first do`eas update --channel staging`and test it on our staging version, and then push the same to production.

If the error is serious or concerns the native layer (e.g. a native module crashes) – we will not avoid releasing it**new version in stores**Then we make, for example, version 1.0.1, improve the code, increment the numbers, build via EAS, and publish as above.

8. **Monitoring:**After the release, we monitor Sentry for new bugs. We can configure Slack integration with Sentry to receive crash notifications. We also monitor**user feedback**in the store – this is often a source of information about problems that we have not detected (e.g. "the app does not work on tablets with Android 13" – which may indicate a specific bug).

9. **Analytics:**We collect analytical data—for example, we look at Firebase Analytics for active users, whether they're using a new feature, and what percentage of users complete registration. Based on this, we plan future iterations.

Build structure, OTA channels – what goes to the store vs. what we update remotely

Finally, let us explain,**what is included in the shop build**, and what can we change OTA:

- **Contents of the binary package (IPA/AAB):**contains all native codes (including built-in expo modules),**bundle JavaScript**(compiled JS code of our application) and assets (images, fonts) needed to run. This bundle is considered the base version of the application.
- **OTA update:**This is a new JS bundle + possibly new assets (e.g. we added an image – it will also be sent in the update). OTA**cannot add/change native code**Therefore, we can't add support for a new device sensor over the air (OTA)—this requires a native library and a new build. However, we can change the logic, appearance, text, etc. over the air (OTA), as long as we use existing native capabilities. If we try, for example, to call a method of a native module that was added only in the next version of the app (and the old one doesn't have it), nothing will happen or we'll get an error—hence the runtimeVersion requirement to isolate such cases.
- **OTA channels:**as described, they allow you to have e.g. a separate update stream for**development**(possibly called "development", although Expo Go and dev build don't use OTA anyway), for**preview/staging**for testing (so that testers can release experimental changes quickly), and**production**for users.
- **How this relates to profiles:**In oureas.jsonThe demo above, production and preview profiles were pinned to specific channels. This allows us to release, for example, a new feature first on*staging channel*– testers (who have the build preview) will receive it OTA – and production users will not (because of a different channel).
- **OTA project structure on Expo server:**(optional to understand) – EAS Update organizes publications in*branchach*, each release has a number (revision ID) and is assigned to a channel and runtimeVersion. In the expo-updates app, it natively saves the OTA manifest and files to the device's memory and decides whether to load them.
- **What goes into the store:**We send the .aab file to Google Play – it contains our binary and individual resource sets (Google Play will generate an APK for various devices from it). We send the .ipa file to the App Store – it's signed and already contains everything (Apple will convert it into an .app file in its infrastructure).
- **Store updates:**When we do**major update**(e.g., version 2.0 with new native features), we release a new build through the app stores. The user must update the app (manually or automatically, if enabled). After updating, the app can also immediately download a newer OTA if available, but usually after a fresh install, there's no need to do so—the embedded bundle is likely the latest.

End-to-end testing / automation: As a point of interest, EAS also allows for end-to-end testing to be run in the cloud after build (e.g., integration with Maestro or Detox), as well as other workflows (like automatic OTA sending on every push to main, etc.). However, this is beyond the scope of our talk.

Literature:

1. https://docs.expo.dev/develop/user-interface/splash-screen-and-app-icon/(Accessed: 1/10/2025) – The official Expo guide to app icon configuration, explaining the principles of creating Adaptive Icons for Android and the requirements for iOS.

2. https://docs.expo.dev/build/introduction/(Accessed: 1/10/2025) – A complete guide to EAS Build, covering the process of building production binaries (.ipa, .aab) in the cloud and file configurationeas.json.

3. https://docs.expo.dev/eas-update/introduction/(Access date: 1/10/2025) – EAS Update service documentation, explaining the Over-The-Air (OTA) update mechanism, channel management and runtime versioning.

4. https://docs.expo.dev/build/setup/(Access date: 1/10/2025) – Instructions for configuring the EAS CLI environment, necessary for authorizing projects, managing Apple/Google certificates and automating the publication process.