KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – lecture 7

Device functions and permissions in Expo/React Native

Mateusz Pawełkiewicz

1.10.2025

1. Expo Permissions

Mobile applications often need to be**user permissions**to access sensitive device functions (e.g., camera, location, photo gallery). In Expo and React Native, permissions are obtained using appropriate Expo modules. Previously, Expo offered a universal moduleexpo-permissions, but since SDK 41 it has been**marked as obsolete**in favor of methods in individual modules. This means that instead of using e.g.Permissions.askAsync(Permissions.CAMERA)we are currently callingCamera.requestCameraPermissionsAsync()or similar methods in the module to which the permission applies. Each Expo module (camera, location, media library, notifications, etc.) provides its own functions for checking and requesting required consents.

Permission request: Most Expo modules provide an asynchronous methodrequest...PermissionsAsync()This displays a native system popup asking for permission. For example, to access the camera, we use:

```
import { Camera } from 'expo-camera';

const { status } = await Camera.requestCameraPermissionsAsync();
if (status !== 'granted') {
alert('No access to camera!');
  return;
}
// Permission granted - you can use the camera
```

It works similarly, e.g.Location.requestForegroundPermissionsAsync()for location orMediaLibrary.requestPermissionsAsync()for the photo library (gallery). When the application calls such a function for the first time, the system (Android or iOS) will display a window asking the user for consent.

Checking status (getPermissionsAsync): Each module also has a methodget...PermissionsAsync(), which allows you to check the current permission status without displaying a window. The returned status is usually one of the following values:"granted"(granted),"denied"(refused) or"undetermined"(not asked yet). Example usage:

```
const perm = await Camera.getCameraPermissionsAsync();
console.log(perm.status); // np. 'granted', 'denied' lub 'undetermined'
```

Often you can call it firstgetPermissionsAsync()– if it returns that the status is"undetermined", then callrequestPermissionsAsync(). However, the callrequest...immediately is also correct - if the permission was already granted earlier, the function will immediately return the status"granted"without asking the user again.

Dedicated modules and permissions: The key Expo modules and their corresponding permissions are listed below:

- **Camera (expo-camera)**– requires permission to use the camera (Camera); when recording video with sound, you also need access to a microphone. Methods:Camera.requestCameraPermissionsAsync(), Camera.requestMicrophonePermissionsAsync().

- **Media Library (expo-media-library)**– Access to the device's photo gallery. Permission to read/write photos (and videos) on the device. Methods:MediaLibrary.requestPermissionsAsync()(on iOS it asks to read by default*i*notation; can be separated by parameterwriteOnly).
- **Selecting images (expo-image-picker)**– This module internally uses camera or media library permissions. It has methods:ImagePicker.requestCameraPermissionsAsync() and ImagePicker.requestMediaLibraryPermissionsAsync()to collect consents from the user.
- **Location (expo-location)**– requires a location permit:*foreground*(while using the app) and optionally*background*(in the background) – more on that below. Methods:Location.requestForegroundPermissionsAsync(), Location.requestBackgroundPermissionsAsync().
- **Notifications (expo-notifications)**– on iOS, permission to display notifications is required. On Android up to and including Android 12, permission was granted automatically upon installation, but since Android 13, runtime permission for notifications has also been introduced. Method:Notifications.requestPermissionsAsync()(you can pass options like alert/sound/badge icon to display).

Platform Differences (Android vs iOS)

Permissions work slightly differently on**Android and iOS**:

- **iOS:**Each permission request must be accompanied by an explanation in the app's Info.plist file explaining why we need that permission. In Expo, this is configured in app.json or via plugins—for example, for a camera key.NSCameraUsageDescription, for locationNSLocationWhenInUseUsageDescriptionetc. Default messages are added automatically by Expo, but they should be adapted to the application context. On iOS, the user can grant (Allow) or deny (Don't Allow) the permission the first time – if they deny, subsequent callsrequestPermissionsAsync() **they won't show it again**dialogue and the status will remain"denied"(iOS assumes the user does not want to be asked again.) In such a situation, the only option is to suggest the user to change the settings manually. Some permissions on iOS have*different levels*: e.g. location has*When In Use*(only when using the app) vs*Always*(also in the background). Request for**background location**requires first consent to "while in use" and then an additional dialogue about "Always". iOS may also offer the option to share approximate location instead of precise location (from iOS 14) - the app has no influence on this, other than specifying the requirement*Accuracy*(the system will still allow the user to decide whether they want to share accurate GPS data).
- **Android:**Permissions are divided into standard and "dangerous" - the latter requires runtime approval. Android allows the user to select "Don't ask again" when denying permissions - in this case, the methodrequest...will return status"denied"and the fieldcanAskAgain= false, which means that the dialog must no longer be shown (subsequent requests will be automatically rejected). In this case, similarly to iOS, the user must be guided to the application settings. Android, starting from version 11, introduced more detailed permissions for files and media: e.g.**READ_MEDIA_IMAGES** i **READ_MEDIA_VIDEO**(instead of the general READ_EXTERNAL_STORAGE), and since Android 13, the native system image picker window has appeared as the preferred

method. Expo ImagePicker in the latest versions has adapted to these changes - if we use the system picker, we can limit the scope of media access instead of asking for full permission to read all files (this helps to meet Google Play's requirements for accessing images). In practice, callingImagePicker.launchImageLibraryAsyncon Android 13+, an app may not need the READ_MEDIA permission at all if the system picker is used – Expo can automatically handle this in newer SDKs. Another example of changes on Android 13 is the permission**POST_NOTIFICATIONS**– now the app must ask for permission to display push notifications (previously, the user agreed by installing the app). ExpoNotifications.requestPermissionsAsync()takes this into account and will ask for this permission on Android 13+.

- **Automatic configuration:**Expo tries to automatically add most of the necessary permission declarations to the native configuration files. When we add a module likeexpo-camera Whether expo-locationand we will build the application, it is required<uses-permission>in AndroidManifest.xml or Info.plist keys are usually added by the so-called config plugins Expo. For example, the AndroidManifest will be enteredandroid.permission.CAMERAusing expo-camera and default text will be added to Info.plist*"Allow $(PRODUCT_NAME) to access your camera"*The programmer may additionally**to remove**unwanted permissions (if the package adds something unnecessary) withandroid.blockedPermissionsin app.json – e.g. blockingRECORD_AUDIOif we use the camera only for photos, so as not to ask for a microphone unnecessarily.

UX Best Practices When Requesting Permissions

Requesting permissions is the moment when the user can decide whether to trust our app in a given area. Here are some tips for doing so while respecting user experience:

- **Ask only when necessary:**Don't display a series of dialogs immediately after launching the app. Instead, request a permission just before the function that requires it. For example, show a camera request dialog before taking a photo, and a file access dialog before saving a file. Users better understand the context of a request when it's linked to their action (e.g., they tapped "Take a photo," so it makes sense that they'd be asked about the camera).
- **Communicate clearly and concisely:**The system window will display the text from the Info.plist/Manifest anyway, so make sure this message is clear (e.g., "Allow app XYZ to access your camera to take photos"). It's also helpful to include a message within the app interface explaining why the request is being made. For example, you could have a dedicated screen that says, "To add a photo, we need access to your camera. A system request will be displayed shortly." This isn't always necessary, but it can be helpful, especially with more sensitive data.
- **Respect the user's decision and offer an alternative:**If a user refuses permission, don't repeatedly force them to accept it. Instead, adapt. For example, if GPS location is not allowed, allow the user to manually enter the address or simply show a static map of the default location. If camera access is denied, allow them to select a photo from the gallery (as they may be more willing to grant this permission), or provide a message that the photo-taking function will be unavailable. It's important that the app remains usable even with limited permissions.

- **Retries and settings:**If the user declined and checked "don't ask again" (Android) or simply declined on iOS (where it defaults to "don't ask again"), the only option is to ask them to change their mind in the system settings. You can display a message like: "Feature X is disabled because permissions have not been granted. You can enable them in the app settings." and, for example, a button "Open Settings". Expo doesn't have a special API for opening settings, but you can use`Linking.openSettings()`with React Native, which will redirect to the app settings. However, don't bombard the user with such prompts – only show them if the feature is truly crucial at the moment.
- **Errors and exceptions:**Always handle Promises from functions that ask for permissions. The user can not only choose "Allow" or "Deny," but sometimes the dialog might be interrupted or some other error might occur. The code should anticipate that`requestPermissionsAsync()`may throw an exception or return an object with a status that is not`"granted"`. Therefore, insert conditions`if (status !== 'granted')`and react to them (e.g., interrupt the action as shown above in the camera example). This will prevent situations in which the application tries to use a resource without permission and, for example, receives an error or (worse still) freezes.

By applying the above rules, we increase the chance that the user**grant permissions**(because it understands why they are needed), and even if not – the application will still work sensibly, instead of frustrating with error messages.

2. Camera, gallery, files – multimedia at Expo

In this section we will discuss how to use Expo/React Native**device camera**, how to get photos from the gallery, how to modify and store files, and how to share them. We will use the following Expo libraries for this purpose:`expo-camera`, `expo-image-picker`, `expo-image-manipulator`, `expo-file-system`, `expo-media-library` and `expo-sharing`All of these libraries are part of the Expo SDK ecosystem (versions current as of 2025).

2.1 Using the camera (`expo-camera`)

Module**expo-camera**allows you to control the camera (both front and rear) directly within the app. It allows you to display a live preview of the camera as a React component and take a photo or record a video. You can also adjust camera parameters (zoom, focus, white balance, flash, etc.) and scan barcodes/QR codes in real time.

Installation: To use it, install the package with the command`npx expo install expo-camera`Then we import the necessary elements, e.g. a component`Camera`the hook`useCameraPermissions`etc. Since the camera is a feature that requires privacy permissions,**We must obtain user consent before use**for access to the camera (and microphone if we plan to record audio along with the video).

Camera preview: `expo-camera`provides a React component`<Camera>`to be inserted into the render tree. Typical use is to place it on part (or all) of the screen along with the shutter button. Example of a simplified component using a camera:

```
import React, { useRef, useState, useEffect } from 'react';
import { Text, View, TouchableOpacity, Image, StyleSheet } from 'react-native';
import { Camera } from 'expo-camera';

export default function CameraExample() {
  const cameraRef = useRef(null);
  const [hasPermission, setHasPermission] = useState(null);
const [photo, setPhoto] = useState(null); // URI of the captured photo to preview

  useEffect(() => {
    (async () => {
      const { status } = await Camera.requestCameraPermissionsAsync();
      setHasPermission(status === 'granted');
    })();
  }, []);

  if (hasPermission === null) {
return <Text>Please wait...</Text>; // while asking for permission
  }
  if (hasPermission === false) {
return <Text>Camera access denied.</Text>; // permission denied
  }

  const takePhoto = async () => {
    try {
      const result = await cameraRef.current.takePictureAsync({
quality: 0.8, // photo quality (0-1)
base64: false, // can be true if we want base64 (e.g. for thumbnail preview)
skipProcessing: false // defaults to false, if true it skips post-processing (e.g. rotation)
      });
setPhoto(result.uri); // save the photo URI to the state
    } catch (e) {
console.error('Error taking photo', e);
    }
  };

  return (
    <View style={styles.container}>
      {photo ? (
// If a photo was taken, we show a preview of it
        <>
          <Image source={{ uri: photo }} style={styles.preview} />
          <View style={styles.buttons}>
            <TouchableOpacity onPress={() => setPhoto(null)} style={styles.button}>
              <Text>Retake</Text>
            </TouchableOpacity>
            <TouchableOpacity onPress={() => alert('Upload zdjęcia...')} style={styles.button}>
<Text>Send</Text>
            </TouchableOpacity>
          </View>
        </>
      ) : (
// If no photo has been taken yet, we display the camera preview and a button
        <>
          <Camera style={styles.camera} ref={cameraRef} type={Camera.Constants.Type.back} />
          <TouchableOpacity onPress={takePhoto} style={styles.captureButton}>
<Text style={styles.captureText}>Take a photo</Text>
```

```
        </TouchableOpacity>
       </>
     )}
   </View>
  );
}

const styles = StyleSheet.create({
  container: { flex: 1 },
  camera: { flex: 1 },
  captureButton: {
    position: 'absolute', bottom: 20, alignSelf: 'center',
    padding: 15, backgroundColor: '#fff', borderRadius: 5
  },
  captureText: { fontWeight: 'bold' },
  preview: { flex: 1 },
  buttons: {
    position: 'absolute', bottom: 20, width: '100%', flexDirection: 'row',
    justifyContent: 'space-around'
  },
  button: { padding: 10, backgroundColor: '#ddd', borderRadius: 5 }
});
```

The code above illustrates the basics: we ask for camera permissions (inuseEffectat the start of the component), then if there are no permissions – we inform about it. When we have permission, we display<Camera ref={...}>We use referencescameraRefto refer to the methodtakePictureAsync()camera component. After pressing the "Take a photo" button, we take a photo and receive an object containing, among others,typephotos. We use this URI to show a preview (<Image source={{ uri: photo }} />). We've also added two buttons: "Retake" (removes the preview and returns to camera mode) and "Send" (here we just display an alert simulating sending).

Comments: takePictureAsyncallows you to specify options such as quality (0-1), whether to return the image as base64, or skip processing. On iOS, e.g., expo-camera automatically rotates the photo according to the device orientation;skipProcessing: truecan skip these steps (useful, for example, to take a photo faster at the expense of no rotation). The resulting photo is automatically saved in the application's cache.result.uriwe get the path to the file (e.g.file:///data/user/0/.../somefilename.jpg). **If we want to keep this photo for longer**, we should move them to a permanent place (e.g. toFileSystem.documentDirectoryor to the device gallery – more on this in section 2.4).

We can also change**camera type**(front/back) dynamically – in our example we usedCamera.Constants.Type.backFor example, you can add a "Switch camera" button that setstypethatCamera.Constants.Type.fronttheback. Expo-camera also supports flash (flashMode), zoom (zoom), autofocus etc., via props passed to the component<Camera>.

2.2 Selecting a photo from the gallery or taking a quick photo (expo-image-picker)

Not every app needs a full camera preview and its own interface for taking photos. Sometimes it's more convenient to use a native media picker—one that lets the user select an existing photo from the gallery, or launches the default camera app and then returns to your app after taking a photo. This is what it's for.**expo-image-picker**.

Installation: npx expo install expo-image-pickerThis module contains both functions for opening**gallery**device and to run the native application**camera**In both cases, after performing an action (selecting a photo or taking a new one), the result is passed to our application.

Right: In the case of expo-image-picker, we need to ensure appropriate permissions:

- If we want to select files from the photo library, we need permission to access media (Photo/Media Library). On Android (up to 12) it was READ/WRITE_EXTERNAL_STORAGE, on Android 13 – the aforementioned READ_MEDIA_IMAGES, and on iOS – access to Photos (with a description in Info.plist). Expo-image-picker provides a methodrequestMediaLibraryPermissionsAsync()to be called before opening the picker. You can passwriteOnly: trueif we plan to only save (e.g. save a photo to the gallery) without reading existing ones - then on iOS, for example, it will only ask for limited write permission.
- If we want to use the camera via image-picker (i.e. open the native camera app), we need the Camera permission. Here we userequestCameraPermissionsAsync()with expo-image-picker (internally, it works similarly to expo-camera). Additionally, on older Androids and iOS 10, access to the "camera roll" (library) was also required, but on newer systems, the native camera app typically saves the photo to its location and returns it to our app without any additional requirements – expo-image-picker manages this abstractly.

Use:expo-image-picker offers two main features:

- ImagePicker.launchImageLibraryAsync(options)– opens the system gallery/files, allows you to select a photo or video.
- ImagePicker.launchCameraAsync(options)– opens the camera (native UI to take a photo).

Both functions return a Promise, which returns a result object upon completion. In the latest versions of expo-image-picker, the result has a structure containing a fieldcanceled(boolean) andassets(array of media objects). For a single photo, this will be a single-element array, with an object having, among other things,type(path to the photo file),width, height, type(media type), possiblybase64(if requested) andexif(if metadata was requested). For simplicity, we can treat it as if we getresult.assets[0].urias a path to the selected photo.

Example: user wants to select an avatar from the gallery:

```
import * as ImagePicker from 'expo-image-picker';

async function pickImageFromGallery() {
// First, let's make sure we have permission to read media
  const perm = await ImagePicker.requestMediaLibraryPermissionsAsync();
  if (!perm.granted) {
alert("The app needs access to your photos to select an image.");
    return;
  }
// Open the gallery and let the user select an image
  const result = await ImagePicker.launchImageLibraryAsync({
mediaTypes: ImagePicker.MediaTypeOptions.Images, // only images (not videos)
allowsEditing: false, // true would allow cropping the image in the system interface
quality: 1 // scale 0-1, 1 = original quality
  });
  if (result.canceled) {
console.log("User canceled image selection");
    return;
  }
  const selectedAsset = result.assets[0];
console.log("Photo selected:", selectedAsset.uri,
            "wymiary:", selectedAsset.width, "x", selectedAsset.height);
// here you can e.g. set a state with this photo to display a preview
}
```

In the above fragment, we first request consent (we show how we handle the situation when the user refuses – we display an alert and abort). If consent is granted, we call`launchImageLibraryAsync`In the options, we specify that we are interested in images (not, for example, videos) and whether to allow editing. Option`allowsEditing: true`would cause that after selecting a photo, the user would be able to crop it (to a square) in the built-in editor. This function uses native mechanisms and, for example, on iOS it will always return a JPEG image even if the source was HEIC, while on Android it has some limitations (e.g. when`allowsEditing:true` i `quality<1`animated GIFs will be reduced to a static image.) In our example, we turn off editing and download full quality.

Similarly, we can do**photo from the camera**without embedding a camera component:

```
async function takePhotoViaSystemCamera() {
// Make sure you have camera permission:
  const permCam = await ImagePicker.requestCameraPermissionsAsync();
  if (!permCam.granted) {
alert("No access to camera.");
    return;
  }
// (Optional: on Android 10 - request for MediaLibrary may be needed
// but expo-image-picker will take care of it itself if necessary)
// Launch the native camera app:
  const result = await ImagePicker.launchCameraAsync({
    allowsEditing: true,
quality: 0.5, // reduce quality to 50% for a smaller file
base64: false // can be true if we also want to get the base64 of the image
  });
  if (!result.canceled) {
    const photo = result.assets[0];
console.log("Photo taken:", photo.uri);
// e.g. set the image to state to show preview in UI:
    setPhotoUri(photo.uri);
  }
}
```

Here after obtaining permission to the camera, we call`launchCameraAsync`The native camera will open (outside of our app, as a system view). After taking a photo, the user usually has the option**accept**the**redo**(this is provided by the system). Once accepted, it returns to our application, andlaunchCameraAsyncresolves the Promise by returning a photo object. In this example, we have includedallowsEditing:true, which means that after taking a photo, the user will be able to, for example, crop or confirm the photo (on iOS, a window with the option to move/scale will appear). With the quality set to 0.5, we will get a compressed photo (smaller file size, at the expense of quality).

What next with the selected photo?In both cases (gallery or camera) we get*type*local file. We can display the image immediately (e.g. in<Image source={{uri: ...}} />), because this file is locally available to our application. If this photo needs to be sent to the server, it can be passed on (e.g., in a form or via upload via fetch). If we want to save it to the application's persistent storage or to the gallery, see section 2.4 below.

Note for iOS (limited access to photos): On iOS from version 14, when we request access to the library, the user can select*limited access*– that is, allow only selected photos. Expo-image-picker tries to handle this – for example, if the user gives limited access, when tryinglaunchImageLibraryAsynca native window will appear for selecting these allowed photos. For the developer, this means thatrequestMediaLibraryPermissionsAsync()can return status"granted"even if access is limited and the fieldaccessPrivilegesmay indicate limited. In the case of limited access, the user may not be able to select any photo, only the previously selected ones. If necessary, it can be detectedperm.accessPrivileges === "limited"and, for example, display a message encouraging you to grant full access in settings if a given functionality requires it.

2.3 Cropping and Compressing Images (expo-image-manipulator)

Often, after taking or selecting a photo, we want to**transform**– for example, reduce the resolution (to save data transfer when sending), rotate, crop to square, or change the format/compression. Expo offers a library**expo-image-manipulator**, which allows you to perform these operations on the device.

Installation: npx expo install expo-image-manipulator.

This library allows you to manipulate an image stored in a local (or base64) file. The key function (in the older API style) isImageManipulator.manipulateAsync(uri, actions, saveOptions). It assumes:

- type– source file location (e.g.photo.uritaken with a camera or selected from a picker).
- actions– an array of objects describing actions (e.g., resize, rotate, flip, crop). Each of these objects has a key specifying the operation and values.
- saveOptions– object with options for saving the result (file format, compression, whether to include base64).

The function returns a Promise with an object containing, among others:typenewly created file (with modified image),width, heightand optionalbase64(if requested).

Example: let's say we want to take a photo of the user (e.g. taken with a camera) and**reduce and compress them**before sending to the server to limit the file size. We can do this like this:

```
import * as ImageManipulator from 'expo-image-manipulator';

// ... let's assume we have a photo.uri from a camera with a resolution of e.g. 4000x3000

const manipResult = await ImageManipulator.manipulateAsync(
  photo.uri,
[{ resize: { width: 1000 } }], // actions: resize to width 1000px, height proportionally
{ compress: 0.7, format: ImageManipulator.SaveFormat.JPEG }
);
console.log("New file:", manipResult.uri, "size:",
      manipResult.width, "x", manipResult.height);
```

Above we pass one action:resizeup to 1000 pixels wide (the height will be adjusted automatically, maintaining the proportions).saveOptionswe setcompress: 0.7(70% quality, i.e. moderate JPEG compression) andformat: JPEG(PNG or WEBP is also possible). The result will be a new JPEG file with a lower resolution, the URI of which we get inmanipResult.uriWe can now, for example, send such a file to a server – it will be significantly smaller than the original photo.

Other actions available in ImageManipulator:

- rotate: degrees– rotates the image by the given angle (90, 180, 270, ...).
- flip: FlipType.HorizontaltheFlipType.Vertical– mirror image horizontally or vertically.
- crop: { originX, originY, width, height }– crops a rectangle from the image with the given starting coordinates and dimensions. For example, to crop the center of the image to a 1000x1000 square, we would need to calculate the appropriate originX/Y.
- *(From SDK 49+)* extend(canvas extension) – less typical, allows you to e.g. add a margin around the image.

It should be noted thatmanipulateAsync **creates a new file**on each call (it does not overwrite the original, because iOS/Android have mechanisms for caching images by path). If we manipulate it repeatedly, it may make sense to delete temporary files after use (e.g. FileSystem.deleteAsync of old files).

New API: The Expo documentation mentions that for some timemanipulateAsyncis marked as*deprecated*, and it is recommended to use the new context-based API (hookuseImageManipulatorand image objects). The new API allows chained manipulations and has a slightly different style (more object-oriented). However, for simple applications (as above), you can still safely useImageManipulator.manipulateAsync, because it is simple and effective. In the context of this lecture, we will focus on this simpler form.

2.4 Saving files locally (expo-file-system i expo-media-library)

When we have a file (e.g. a photo) in the Expo application, we may want to**save permanently**or make it available to the user outside the app. We have two main paths:

- save the file in the application's internal file system (so-called sandbox, not directly accessible from other applications),

- save the file to the device's public library (e.g. a photo to the gallery, a file to the public directory) so that the user can see it outside of our application.

Expo FileSystem: Library expo-file-systemallows access to the file system**inside the application sandbox**By default, expo provides paths such as:

- FileSystem.documentDirectory– persistent directory for applications (data remains here until the user uninstalls the application or deletes it).
- FileSystem.cacheDirectory– a temporary directory (cache) that the system can clean up when necessary.

Features such asFileSystem.moveAsync, copyAsync, writeAsStringAsync, readAsStringAsync, deleteAsyncetc., we can manipulate files.

Example: we took a photo with a camera and received e.g.photo.uri = "file:///.../Camera/abc.jpg"(location in the Expo cache). If we want to save them, for example, in our documents folder under the name "profile.jpg", we can do this:

```
import * as FileSystem from 'expo-file-system';

const sourceUri = photo.uri; // source path (cached file)
const destUri = FileSystem.documentDirectory + "profile.jpg";
await FileSystem.copyAsync({ from: sourceUri, to: destUri });
console.log("File copied to documents:", destUri);
```

The above will copy the file. You can usemoveAsync instead copyAsync, if we want to move it (remove it from the source). This way, the file will be available in the future at a known address (e.g., we can load and display it later, even after restarting the application). The internal memory of the application is isolated - e.g. on Android it is usually/data/data/<package>/files/..., on iOSDocuments/...for the application – and it will not automatically appear in the user's gallery or file manager.

Expo MediaLibrary: If we want a file (e.g. a photo) to be sent to**user photo gallery**, we should useexpo-media-libraryThis library integrates with the system's multimedia library (application**Photos**on iOS or the media library on Android). It allows, among other things, reading photos/albums, but also saving files to the album. To use it, you need permission to read/write media (as discussed in the permissions section). We assume that the user has given consent.

To save a photo to the gallery, we use the functionMediaLibrary.createAssetAsync(uri). E.g.:

```
import * as MediaLibrary from 'expo-media-library';

const asset = await MediaLibrary.createAssetAsync(photo.uri);
await MediaLibrary.createAlbumAsync("MojaAplikacja", asset, false);
```

The first line creates*asset*from the specified URI file – that is, it adds a photo to the system media database (in the default album, usually*Camera Roll*). Returns an asset object (containing, among others, a unique ID, type, URI, etc.). The second line creates an album called "MyApp" and moves this asset to it (parameterfalsemeans that if the album already exists, do not duplicate the file). You can skipcreateAlbumAsync, if we just want to throw it to the default location.

Expo-media-library will automatically take care of saving the image to Photos on iOS (which the user will see in the Photos app), and on Android it will place the file in DCIM/ or Pictures/.

Attention: From Android 10+ the concept was introduced**Media Store**, so expo-media-library uses this API to save files. This requires declaring permissions like READ/WRITE_MEDIA_IMAGES (which Expo does automatically, although due to Google's policy, if your usage is sporadic, you might consider using the system picker instead of full permissions). However, if you need to save files programmatically, expo-media-library is the right tool.

To sum up: expo-file-systemis used to manage files within the application (cache, documents), andexpo-media-libraryallows you to interact with the user's gallery. In a typical scenario:

- we use expo-camera or image-picker -> we get the file in the cache,
- if the user saves a photo – we ask for consent (if it was not given) and use MediaLibrary to save it to the gallery,
- Additionally or instead, we can keep the photo in the application's sandbox (if it is, for example, private data only for our application).

2.5 File sharing (system share sheet withexpo-sharing)

We often want to enable the user**sharing**a photo or file through other applications – e.g. send a photo by e-mail, via instant messenger, save to Drive, etc. Mobile systems provide the so-called*share sheet*– the standard “Share” window, which shows a list of applications and actions possible for a given file. In Expo, we can invoke this window through the library.**expo-sharing**.

Installation: npx expo install expo-sharing.

Expo-sharing has a very simple interface: first, it is worth checking whether sharing is available viaSharing.isAvailableAsync()It will always be available on native mobile platforms; it may not be available on the web (the Web Share API is limited). Then we useSharing.shareAsync(url, options) - Where url to **Local file URI**that we want to share.

Let's assume we have a path to an imageimageUri(e.g., a photo taken with a camera, saved in the application files). We can share it like this:

```
import * as Sharing from 'expo-sharing';

async function shareFile(uri) {
  const canShare = await Sharing.isAvailableAsync();
  if (!canShare) {
alert("Sharing is not available on this platform");
    return;
  }
  try {
    await Sharing.shareAsync(uri);
console.log("File has been shared.");
  } catch (error) {
console.error("Sharing error:", error);
  }
}
```

CallshareAsyncwill open the native sharing panel. The user will see a list of apps to which they can send the file – for example, on iOS, icons for AirDrop, iMessage, Mail, Image Saver, etc. will appear, on Android, for example, Gmail, Messenger, Drive, etc. Once they select one and the sharing is successful, the Promise will be resolved (we may not receive many details, usually we don't know if the recipient received it, etc., only that our file was transferred to the system).

Limitations: Expo-sharing only works with local files and on native platforms. In a web browser (PWA)**Web Share API**allows sharing only certain types of data and requires HTTPS – expo-sharing uses it if available. However, browsers do not allow sharing local files from URIs directly, so this feature may be impractical on the web (you would have to, for example, upload the file to a URL first). Generally, expo-sharing is most useful on Android/iOS. It also does not allow*receiving*shared content from other apps (that is, our app cannot declare "open this file type in my app" using expo-sharing - this requires native configuration and custom plugins).

3. Location and maps

Another important functionality of the devices is**geolocation**– determining the user's GPS position – and presenting this position on maps. In the Expo ecosystem, we have a moduleexpo-locationto obtain locations and we can use the libraryreact-native-mapsto display maps (Google Maps / Apple Maps) in the application.

3.1 Acquiring locations (expo-location)

Module**expo-location**enables reading**current GPS location**device, subscribing to location changes, as well as geocoding (converting coordinates to addresses and vice versa). Using it, like others, requires prior authorization from the user.

Location permissions: We distinguish two types –**foreground location**(while using the app) and**background location**(in the background). In most cases, we only need the first one (e.g., to show the user's location on the map or search for something nearby). Background location is needed when the application is to track location even when it is inactive (e.g., a training application that records a run, a tracker, etc.). Querying the background is possible immediately on Android (although it is better to first query the foreground and then the background), and on iOS it is a two-step process, as mentioned earlier. Expo provides appropriaterequestForegroundPermissionsAsync() i requestBackgroundPermissionsAsync(). **On iOS**for background to work at all, you need to add a property to Info.plist and enable background mode – Expo allows this through configurationisIosBackgroundLocationEnabled: truein the expo-location plugin, and the NSLocationAlwaysAndWhenInUseUsageDescription key must be provided. Without it, we won't get approval from Apple anyway.

In a typical map application, foreground permission is enough.

Getting current position: The simplest method isLocation.getCurrentPositionAsync(options)It internally retrieves data from GPS and/or other sources (WiFi, network) and returns a Promise with a location object. This object contains, among other things, coordinatescoords.latitude i coords.longitude, accuracycoords.accuracy(in meters), as well as e.g. heightaltitude, speedspeedor directionheading(where available). Also includes a timestamp.

Usage example (assuming we already have the permission):

```
const location = await Location.getCurrentPositionAsync({
accuracy: Location.Accuracy.High, // can be Balanced, Low, etc. High uses GPS, may be slower
maximumAge: 10000, // if the last known position is not older than 10s, it can give it immediately
});
console.log("My location: ", location.coords.latitude, location.coords.longitude);
```

This call turns on the GPS (if it was not already on) and may take a while (a few seconds) to return an accurate position, especially when first started.

Position tracking (watchPosition): If we want to receive position updates continuously (e.g. after a certain distance or time), we useLocation.watchPositionAsync(options, callback)This function**starts the subscription**– will trigger our callback whenever the location meets the change criteria. Returns a subscription object (for later cancellation.remove()).

Example: tracking user traffic:

```
const subscription = await Location.watchPositionAsync(
  {
   accuracy: Location.Accuracy.High,
timeInterval: 5000, // at least every 5 seconds
distanceInterval: 10 // or every 10 meters
  },
(place) => {
const { latitude, longitude } = loc.coords;
console.log(`New position: ${latitude}, ${longitude}`);
// here e.g. update the application state with the new coordinates
  }
);
// ... later, when we no longer need to track:
subscription.remove();
```

In the above code, every ~5 seconds or every ~10 meters (whichever comes first) we will get a new position.timeInterval i distanceIntervalhelp limit too frequent readings (saves battery).accuracyalso influences –Highturns on GPS,Balancedcan use the network, which is faster but less accurate,Lownetwork only (aims for accuracy of a few km).

Background location: Expo-location also allows for background tracking, even when the app is inactive. This is accomplished by registering a task using the module.TaskManager(you need to create a task definition in the code) and callLocation.startLocationUpdatesAsync(taskName, options)If the user has sent*Allow all the time*(Always) the permission and native configuration are appropriate, the app will receive updates in the background, even when closed (to a certain extent – the system may restrict them). This is a rather advanced topic, so we're only highlighting its existence. It's important to remember that, for example, on iOS, the user sees a blue bar indicating that the app is using location in the background, which may raise concerns – so always explain why you're doing this. Many apps don't need background location at all – use it sparingly.

Geocoding:I will mention briefly thatexpo-locationit also has functionsLocation.reverseGeocodeAsync(coords)– converts the coordinates into a readable address (street, city, etc.), andLocation.geocodeAsync(address)– conversely, address to potential coordinates. This uses platform services (iOS/Android) and requires an internet connection (on iOS, it uses Apple Maps API, on Android, Google geocoding). This can be useful in many cases, but keep in mind that it won't always return a result (e.g., if the address is unclear). These features don't require any special permissions, except that if they use the user's location, we need that permission.

3.2 Displaying maps (react-native-maps)

The standard library for integrating maps in React Native is**react-native-maps**. Expo supports it (it is listed as compatible, install viaexpo install react-native-maps). It allows you to embed a MapView component that will display Google Maps (on Android and optionally on iOS) or Apple Maps (default on iOS). We can place**markers**, shapes, handle events (taps on the map, drags, etc.).

Installation and configuration: In Expo Go mode, we don't need to do anything more – Google/Apple Maps should work out of the box (Expo provides API keys for Expo Go). However, for custom iOS builds, you often need to provide a Google Maps API key if you want to use Google as your map provider (alternatively, you can stick with Apple Maps on iPhone). Details on configuring Google Maps on Android/iOS are in the documentation, but in short: on Android, you need an API key in the manifest file (Expo pluginmapsmakes it easier), and on iOS, adding a key to the Info.plist and GoogleMaps library in the project – Expo provides this via the config plugin. However, in Expo SDK 54+, most of it is automatic for standard use (unless we use custom functions).

Using MapView and Marker: Basic example of displaying a map with a single marker:

```
import MapView, { Marker } from 'react-native-maps';
import { StyleSheet, View } from 'react-native';

export default function MapExample({ userLocation }) {
  // `userLocation` to obiekt { latitude: ..., longitude: ... }
  const region = {
    latitude: userLocation.latitude,
    longitude: userLocation.longitude,
latitudeDelta: 0.01, // the smaller the delta, the greater the zoom
longitudeDelta: 0.01
  };
  return (
    <View style={styles.container}>
      <MapView style={styles.map} initialRegion={region}>
        <Marker
          coordinate={userLocation}
title="I'm here"
description="My current location"
        />
      </MapView>
    </View>
  );
}

const styles = StyleSheet.create({
  container: { flex: 1 },
  map: { flex: 1 }
});
```

Here we assume that we already have the user's coordinates (e.g. previously retrieved byexpo-location). We setinitialRegionmap to the area centered around this point.latitudeDelta i longitudeDeltadetermine the zoom – a smaller value means closer (0.01 ~ a few streets, 0.1 ~ the whole city, 1 ~ the whole country, etc.). We place the component on the map<Marker>in the user's coordinates. We set ittitle(this will appear as a bubble header when you click on the marker) anddescription(smaller text in a bubble). As a result, the user will see a map with a pin labeled "I am here." The marker is red by default (Google) or a red pin (Apple). It can be customized (proppinColoror your own picture as a marker).

Map Interaction and Control: The map can be moved and scaled using multi-touch gestures (enabled by default). You can also control the region from the app's state, for example, using props.region (instead initialRegion) binding it to state, but then we have to remember to update it (becauseregionbecomes controlled - if the user scrolls the map, we have to note it so as not to "take" the map to the old position). It is often usedinitialRegionfor a simple display, and when dynamic control is needed, references and animation methods are used:

- mapRef.current.animateToRegion(region, duration)the
- on iOSmapRef.current.animateCamera(camera, duration).

If we want to show**user's current position**dynamically, react-native-maps also offers propshowsUserLocation={true}This will automatically draw a blue dot (and an accuracy circle) where the system sees the device's location. However, this requires permission and requires that the location be retrieved at least once (it won't work on iOS otherwise, as a trigger is required). When using expo-location for subscriptions, we can update, for example, a marker or region in the callback.

Marker and other elements: The marker can respond to taps (proponPress), may havecallout(bubble) more complex – e.g.<Marker><Callout><Text>Some info</Text></Callout></Marker>to customize the content of the bubble. We can also add other elements such as<Polyline>(paths),<Polygon>, <Circle>etc. – they require appropriate data input (a list of points). This depends on the application's needs (e.g., running route, area outline, etc.).

Maps on different platforms: Default:

- That**Android**Google Maps is used (requires Google Play Services).
- That**iOS**Apple Maps is used by default, but*you can*switch to Google Maps (by setting some flags and providing an API key). However, many apps stick with Apple Maps on iOS because it doesn't require additional keys, and Apple Maps is quite good for most uses.
- That**Web**(Expo web) react-native-maps doesn't work natively, but it's possible to use, for example, Leaflet-based maps through another package. However, if we're also targeting the web, additional solutions are usually required (beyond the scope of this lecture).

Performance and limitations: Please note that maps are a native component – on iOS it is`MKMapView`, on Android`MapView`from Google. They respond to styles (you need to give them a specific height/width). It's better to wrap the map in a view with specific dimensions (as we did in styles,`flex:1`fills the parent). Drawing many markers (e.g. hundreds) can impact performance – the library offers marker clustering and optimizations, but these are advanced topics.

3.3 Background Location (Introduction)

As previously mentioned, it's possible to continuously track a user's location in the background. Since this is an advanced topic, we'll only outline it here:

Expo provides a background task mechanism through the module`expo-task-manager`We define the task, e.g.:

```
import * as TaskManager from 'expo-task-manager';
TaskManager.defineTask("locationBackground", ({ data, error }) => {
  if (error) {
    console.error("Błąd w background location task:", error);
    return;
  }
  if (data) {
const { locations } = data; // array of new locations
    const loc = locations[0];
    console.log("Background location:", loc.coords);
// here you can e.g. send data to the server or save it in memory
  }
});
```

Then, somewhere in the code (when we want to start), we call:

```
await Location.startLocationUpdatesAsync("lokalizacjaTlo", {
  accuracy: Location.Accuracy.Balanced,
timeInterval: 60000,
  distanceInterval: 50,
  pausesUpdatesAutomatically: true
});
```

This will cause the application (or rather the system) to wake up our application from time to time and trigger the task"locationBackground"with new data. The condition is to have the right*Always*(Always) on iOS and appropriate entries in Info.plist (Background modes -> Location), and on Android the ACCESS_BACKGROUND_LOCATION permission (Expo will add it automatically ifisAndroidBackgroundLocationEnabled: truein app.json). You have to take into account that the system may limit the frequency – for example, in iOS, if the application is not running, we will receive locations every ~ several minutes, even if we give 1 second (the system takes care of the battery). Android Q+ requires the user to additionally confirm*Allow in background*in a separate dialogue.

Examples of background location applications: Tracking apps (fitness apps – tracking running or riding routes), friend/child locators (which send their location to the server periodically), route logging, etc. Users should be informed about this functionality, as it impacts privacy and battery life. From the app store's perspective, Apple may reject an app that requests Always Location without a good reason supported by the description.

In summary, expo-location covers most geolocation needs—from single-point position acquisition to continuous tracking, both in the foreground and background. Combined with maps, rich location-based features can be built.

4. Notifications

Notifications are a mechanism for informing users about important events, even when they're not actively using the app. In the Expo ecosystem, we manage them using the**expo-notifications**We divide them into**local notifications**(generated by the application itself) and**push (remote)**sent from the server to the device. We'll discuss both categories and the differences between platforms.

4.1 Local notifications (in-app)

Expo-notificationsallows you to create local notifications – i.e. reminders triggered at a specific time or in response to some user action (e.g. a notification "you've taken 10,000 steps!" in a fitness app, or an immediate notification about receiving a chat message when you're in another part of the app).

To use expo-notifications, we install the packageexpo-notificationsand import it. On iOS**we must ask the user for consent**for notifications (typical system dialog "App wants to send you notifications - allow/don't allow"). In Android <13 there was no such dialog - notifications were allowed by default (the user could disable them in the settings). However, since Android 13, a runtime permission also appears (Expo will handle it in the same function). Therefore, it is good practice to always callNotifications.requestPermissionsAsync()when initializing notifications.

Request for consent to notification:

```
import * as Notifications from 'expo-notifications';
```

```
const { status } = await Notifications.requestPermissionsAsync();
if (status !== 'granted') {
alert('Notification consent not obtained.');
// You can terminate the procedure or continue without notifications
}
```

On iOS it is possible to pass the option to`requestPermissionsAsync`, e.g. which types we want (`allowAlert`, `allowSound`, `allowBadge`, `provideAppNotificationSettings`, `allowAnnouncements`). By default, if we don't provide it, it asks for the standard ones (alerts, sounds, badges). You can also ask for the so-called**provisional**permission (silent notifications that do not display an alert, but only appear in the notification center) by setting e.g.`ios: { allowProvisional: true }`– then the user doesn't see the dialogue, and the app can send "silent" notes, which the user can fully enable later. This is an advanced feature.

Display local notification: Once we have permission (or on Android, it wasn't required), we can trigger a notification. There are two ways:

- **Instant notification**– in practice, local notification*Too*uses the system, so even for immediate we use the scheduling method, only with zero delay.
- **Scheduled notification**– after a certain period of time or at a specific time.

Expo-notifications simplifies this with one function`scheduleNotificationAsync`. We provide an object with the notification content (`content`) and safe (`trigger`). Trigger can be:

- specified in seconds (time delay),
- specified by date (exact moment),
- repeatable (e.g. every day at 9:00, every week, etc. – iOS supports calendar repetitions).

Example: we want to immediately display a notification informing about the success of some operation:

```
await Notifications.scheduleNotificationAsync({
  content: {
title: "Operation completed",
body: "Your data has been successfully saved.",
sound: 'default' // default sound (on Android you need to define the sound channel beforehand or use the default one)
  },
trigger: null // null means immediately (right after the call)
});
```

If `trigger: null`, expo will send the notification immediately. Alternatively, you can use`trigger: { seconds: 5 }`– which will cause the notification to be shown in 5 seconds. This function returns the notification ID (string), which can be used, for example, to cancel it early (`Notifications.cancelScheduledNotificationAsync(id)`).

Notification content (`content`): You can set:

- `title`– title (large text, usually bold),

- body– content,
- data– an object with data (e.g. some ID that we want to pass when the user clicks on the notification to know what to do),
- sound– sound name from resources or'default'to use the default. (On Android, sounds are assigned to channels – expo-notifications creates the "Default" channel automatically with the standard sound. If you want your own sounds, you need to define channels.)
- badge– number, sets the badge icon on the app icon (iOS).
- subtitle, body, etc. – iOS supports subtitle.
- attachments– you can attach an image (on iOS/Android 13+).

For the purposes of this lecture, we will focus on simple fields: title and content.

Responding to notifications: In the context of local notifications, it is worth mentioning that expo-notifications allows you to listen for events:

- receiving a notification (when the application is in the foreground - otherwise the system displays it itself, but when we are in the application, iOS by default will not show a banner, so you can intercept it and, for example, display your own alert or badge in the UI),
- user clicks on the notification.

You can useNotifications.addNotificationReceivedListener i Notifications.addNotificationResponseReceivedListenerIn the callback we get the objectNotificationtheNotificationResponse(containing, among others,notificationand information about the action). In simple applications we can skip this, but for example if we want the application to navigate to a specific screen when the user clicks on a notification in the tray, then we implement such logic in this listener.

4.2 Push Notifications (Remote Notifications – Overview)

Push notifications are messages sent from a server to a specific application on the user's phone. Unlike local notifications, they are initiated outside the application (e.g., someone sends a message to us – the chat server sends a push to notify the recipient). Implementing push notifications requires integration with services.**Apple Push Notification service (APNs)**for iOS and**Firebase Cloud Messaging (FCM)**for Android. Expo provides a lot of help here by**Expo Push Service**, acting as an intermediary.

To use push:

1. The app on your device must**register for a token**In classic RN this means calling APNs and FCMs separately, but expo-notifications simplifies this.
2. We transfer this token to our**server**.

3. Our server (or other mechanism) sends a request to Apple/Google servers with information for the device.

Expo Push Serviceshortens step 3 – allows us to send a notification to**Expo server**, and it will then forward it to APNs/FCM. This means we don't have to implement our own Apple/Google connection, which is especially convenient in Expo Managed mode.

Obtaining an Expo Token: In expo-notifications we callNotifications.getExpoPushTokenAsync(options)This function inside:

- On iOS, register the app in APNs and getdeviceToken.
- On Android it will register with FCM and getdeviceToken(this requires having a Firebase project configured in our application – expo with EAS Build does it for us if we provide the server key).
- Then it will send this data to the Expo service and receive a unique**Expo Push Token**(string starting with"ExponentPushToken["...]).

This expo push token identifies our device + specific application.**Attention:**From SDK 42/43 expo requires the provision ofexperienceIdtheprojectIdto generate a valid token (especially in a bare workflow), so we usually call it asNotifications.getExpoPushTokenAsync({ projectId: '<GUID of our expo project>' })In managed workflows, Expo usually knows the ID itself, but in case of problems, you need to provide it.

Example of downloading a token (after obtaining prior consent to notifications):

```
import * as Notifications from 'expo-notifications';

async function registerForPushNotifications() {
  const tokenData = await Notifications.getExpoPushTokenAsync({
projectId: '<your-expo-project-id>'
  });
  const expoPushToken = tokenData.data;
  console.log("Expo push token:", expoPushToken);
// Here we usually send this token to our backend, e.g.:
  await fetch('https://moj-backend.example.com/register-token', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ token: expoPushToken })
  });
}
```

Let's assume our backend will store a token associated with the user. Now**sending a notification**from the backend is simple: we send an HTTP POST request to**endpointu Expo** https://exp.host/--/api/v2/push/sendwith JSON containing the token(s) and notification content. Sample payload:

```
{
  "to": "ExponentPushToken[xxxxxxxxxxxxxxxxxxxxxx]",
"title": "New message",
"body": "User Jan wrote to you: Hey!",
  "data": { "conversationId": "12345" }
```

}

Expo Push Service will accept it and return the so-called**tickety**(confirmation of receipt). It will then asynchronously forward the message to APNs or FCMs. These, in turn, will deliver it to the device, where the system will display a notification.

How push notifications work in Expo: The mobile app registers and receives an Expo Push token. Our server (backend) then sends a request using this token to**Expo Push API**The Expo server forwards the message to the appropriate platform—Apple Push Notification service for iOS or Firebase Cloud Messaging for Android. Finally, Apple/Google delivers the notification to the user's device, where it appears in the notification center. This mechanism allows developers to send push notifications directly through Expo servers, instead of integrating separately with APNs and FCM.

It is worth noting that to use push notifications in a real production application, you also need:

- **On iOS**: provide keys/pem to Expo (if using Expo Push) or configure our APNs certificate in the project accordingly. Expo in managed mode allows you to`eas build`it is easy to upload push key (.p8) to our project.
- **On Android**: for Expo Push - provide the FCM Server Key in the project settings (Expo Developer Dashboard) or via`expo-cli`so that Expo can send to our users. If we were building our own backend without Expo, we would have to use native FCM in the application and generate the token there, etc., but in Expo Managed, Expo Push is a convenient way.

Push reception in the app: When the user receives a push:

- If the app is in the background or closed, the system will display a notification in the tray. Tapping it will open the app. We can capture this event (NotificationResponse) and, for example, navigate to the appropriate screen (e.g., open a specific chat).
- If the application is in the foreground (open and visible), then:
    - That**Android**the notification will also appear in the tray (default).
    - That**iOS**notification*will not be displayed*as a banner (Apple assumes that since you're in an app, the app itself can handle it – this prevents duplicate messages). In such a situation, expo-notifications allows us to listen for the notification event and, for example, display our own alert or badge in the UI. Alternatively, you can force iOS to show notifications in the foreground by setting a flag`shouldShowAlert`in the handler or by configuring*Notification Service Extension*, but this is used less frequently.

Push Summary: Implementing push notifications requires a combination of multiple elements: the application (token registration), the backend (sending via Expo API or directly via APNs/FCM), and key configuration. Expo makes this much easier with Push Service, but remember the limitations:

- expo push has certain speed limits (e.g. you should not send more than 100 notifications/sec to one IP, it is worth batching the sendings).

- Notifications are not guaranteed - they may not arrive immediately if the device is offline, or at all if, for example, the user has uninstalled the application (therefore, it is worth deleting the inactive token, Expo will return a DeviceNotRegistered error).
- On iOS, the user can turn off notifications for our app in the settings after giving consent, so it is worth reacting to any possibleNotifications.getPermissionsAsync() Where statuscan go intodeniedwhile using the application (then we can, for example, inform you that notifications are off).
- **Debugging:**In the iOS simulator, we won't get push (Apple doesn't support it in simulators). On Android, the emulator can only receive push if we install the Firebase service there and use push without Expo Go. Expo Go from SDK 53 doesn't support push on Android at all, so we make our own push to test it.**Development Build**the**standalone build**Local notifications, on the other hand, can be tested everywhere, including at Expo Go.

4.3 Differences and limitations of notifications on Android vs. iOS

When writing an application, it is worth knowing certain things**platform differences**in the notification system:

- **Icons and channels (Android):**Android requires that each notification be assigned to**channel**(Notification Channel). The channel determines, among other things, sound, priority, vibration – and the user can manage it in the settings (e.g. turn off the sound for a given channel). Expo-notifications automatically creates the "Default" channel if we don't specify another one, and uses it. We can create channels ourselves byNotifications.setNotificationChannelAsync("nazwakanału", options)It's worth doing this, for example, if you want different types of notifications (e.g., "news" with sound and "promotions" without sound). Furthermore, Android requires you to add your own notification icon – otherwise, it may show a white circle by default. Expo allows you to set the notification icon in app.json (fieldnotification.icon).
- **Badges:**iOS has native support for the so-called badge count – the number on the app icon. Android didn't have this system-wide (some launchers supported it), only some overlays. Expo-notifications allows you to useNotifications.setBadgeCountAsync(n)on iOS, and on Android it simply does nothing or uses Support Lib (the new Android 13 introduces so-called notification dots, but not numbers).
- **Presentation when app in foreground:**As already mentioned, iOS won't show the banner by default, Android will (unless we tell it otherwise in the notification options). However, if we want to handle it consistently internally, expo-notifications allows us to use a listener and, for example, manually triggerNotifications.scheduleNotificationAsyncimmediately from the received notification (which is a bit of a hack to force a banner on iOS - there is also a methodNotifications.presentNotificationAsync(content)in some versions, which displays the notification immediately even in the foreground).
- **Data limit:**Both APNs and FCM have push payload size limits. APNs ~4KB, FCM ~4KB (in the case of expo, we don't care about this directly, because the expo server will tell us if we exceed it). So we don't send huge JSONs indata.
- **Action buttons, text responses:**More advanced features, such as interactive notifications (with action buttons), require native configuration (on iOS, defining a

category; on Android, adding an action to pendingIntent). Expo-notifications doesn't currently support this out-of-the-box, so in a managed workflow, we're limited to simple tap notifications. If we need, for example, a "Reply" button without opening the app, this requires going beyond Expo-managed.
- **Recurring schedule:**On iOS, expo allows you to schedule a repeating notification with a trigger likedaily/weekly(e.g., every day at 9:00) – this is implemented via Apple Calendar triggers. On Android, until recently, there was no simple API for repeating exactly every week (you have to use AlarmManager), but if we set a trigger{ repeats: true, hour: 9, minute: 0 }this should work both here and there.
- **Expo Go / dev builds:**As mentioned, in pure developer mode (Expo Go)**Android**from SDK 53 onwards it won't support push (Expo's decision due to the limitations of FCM implementation in Expo Go), and you have to use a development build (which is like your own app). iOS Expo Go traditionally doesn't support push at all (because the Expo Go app doesn't have push for each test app). Therefore**for push testing**we need to have our own build on the physical device.

5. Application demo: camera and map (practical combination)

Finally, let's combine the above topics into a mini-demo. Let's assume a scenario: we're building an app where a user can take a photo and save it (or send it), and in another tab, view their current location on a map marked with an "I'm here" marker. This will cover camera usage, camera/gallery permissions, file saving, location, and the map. Below, we'll discuss the implementation of two screens in this sample app.

5.1 "CameraScreen" screen – taking and saving a photo

Assumptions: This screen allows the user to take a photo using the camera. After taking the photo, it displays a preview and offers the options: "Save" (to the gallery) or "Send" (simulating an upload). If the user doesn't have permission to use the camera, an appropriate message will be displayed. Additionally, we'll add a button to select a photo from the gallery as an alternative (fallback, for example, if the camera is unavailable).

```
import React, { useState, useEffect } from 'react';
import { View, Text, Image, Button, Alert } from 'react-native';
import { Camera } from 'expo-camera';
import * as ImagePicker from 'expo-image-picker';
import * as MediaLibrary from 'expo-media-library';

export default function CameraScreen() {
  const [hasCamPermission, setHasCamPermission] = useState(null);
  const [camera, setCamera] = useState(null);      // referencja do Camera component
const [photoUri, setPhotoUri] = useState(null); // URI of the photo taken

  useEffect(() => {
    (async () => {
      const { status } = await Camera.requestCameraPermissionsAsync();
      setHasCamPermission(status === 'granted');
    })();
  }, []);

  const takePhoto = async () => {
    if (!camera) return;
    try {
      const result = await camera.takePictureAsync({ quality: 0.7 });
      setPhotoUri(result.uri);
    } catch (e) {
      console.error("Error taking photo", e);
    }
  };

  const pickImage = async () => {
    const perm = await ImagePicker.requestMediaLibraryPermissionsAsync();
    if (!perm.granted) {
Alert.alert("No permission", "Share photo access to select an image from the gallery.");
      return;
    }
    const result = await ImagePicker.launchImageLibraryAsync({ quality: 1 });
    if (!result.canceled) {
      const asset = result.assets[0];
      setPhotoUri(asset.uri);
    }
  };

  const savePhotoToGallery = async () => {
    if (!photoUri) return;
    const perm = await MediaLibrary.requestPermissionsAsync();
    if (!perm.granted) {
Alert.alert("No write permission", "Cannot save photo without gallery access.");
      return;
    }
    try {
      const asset = await MediaLibrary.createAssetAsync(photoUri);
      await MediaLibrary.createAlbumAsync("DemoApp", asset, false);
Alert.alert("Success", "Photo saved to DemoApp album!");
setPhotoUri(null); // clear and return to camera mode
    } catch (e) {
      console.error("Save error", e);
Alert.alert("Error", "Failed to save photo.");
    }
  };

  const uploadPhoto = async () => {
// Here would normally be code sending the file to the server, e.g. via fetch / FormData.
// We will only signal this with an alert:
Alert.alert("Sending", "Simulating file upload " + photoUri);
    setPhotoUri(null);
  };

  if (hasCamPermission === null) {
return <Text>Checking permissions...</Text>;
  }
  if (hasCamPermission === false) {
    return (
      <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
```

```
<Text>Access to the camera was denied.</Text>
<Button title="Select image from gallery" onPress={pickImage} />
     </View>
   );
  }

  return (
    <View style={{ flex: 1 }}>
{photoUri ? (
// After taking/selecting a photo - preview screen
       <View style={{ flex: 1 }}>
         <Image source={{ uri: photoUri }} style={{ flex: 1 }} resizeMode="contain" />
         <View style={{ flexDirection: 'row', justifyContent: 'space-around', padding: 10 }}>
           <Button title="Zapisz" onPress={savePhotoToGallery} />
<Button title="Send" onPress={uploadPhoto} />
           <Button title="Anuluj" onPress={() => setPhotoUri(null)} />
         </View>
       </View>
     ) : (
// Screen with camera and buttons
       <Camera style={{ flex: 1 }} ref={ref => setCamera(ref)} />
     )}
{!photoUri && ( // buttons to take or select a photo when there is no one taken
       <View style={{ position: 'absolute', bottom: 20, alignSelf: 'center' }}>
<Button title="Take a photo" onPress={takePhoto} />
         <Button title="Galeria..." onPress={pickImage} />
       </View>
     )}
    </View>
  );
}
```

Explanations:

- After assembling the component, please ask for permissionCameraIf the request is denied, instead of a camera view, we display a message and a button that allows the user to choose an alternative—selecting a photo from the gallery (although this allows the user to add an image even without a camera). This is a good example of a fallback.
- When permission is granted, we render<Camera ref={...}>We set the ref bysetCamera(ref)in the attributeref(this is a workaround, because we used the hooks from expo-camera in the classic way). You could also useuseRefand assign tocameraRef.current.
- The "Take a photo" button brings uptakePhotowhich uses camera references (camerahere) and its methodstakePictureAsyncWe set the quality to 0.7 for a smaller file. After success, we save the URI in the statephotoUri.
- When photoUriis set, instead of the camera view we show the preview (Image). Below it, three buttons: "Save", "Send", "Cancel".
  - "Save" callssavePhotoToGallery– there we first ask for (possibly missing) permissions to MediaLibrary, then we create an asset and an album (an album named "DemoApp"). If successful, we show an Alert with a success message and clean upphotoUri(we return to camera mode).

    - "Send" triggersuploadPhoto– here we don't have a real server, so we just show an alert with information and clean it up as wellphotoUri(we assume that after sending we no longer need a preview).
    - "Cancel" also clearsphotoUri(we discard the photo and return to the camera).
- The "Gallery..." button brings uppickImage– asks for permission to access the library, then opens expo-image-picker. If the user selects an image (result.canceled false), we downloadasset = result.assets[0]and we set its URI asphotoUriThis takes us to the same preview view as after taking a photo with the camera. There, the user can save or send it—meaning we're using common logic. This allows the "select from gallery" fallback to work both in the absence of a camera and normally (we even added this button next to "Take Photo" for convenience).
- Finally, if the camera is active (nophotoUri), at the bottom of the screen we have two buttons for taking a photo and opening the gallery.

This screen demonstrates the use of the permissions (camera, media library), expo-camera and expo-image-picker, as well as expo-media-library for writing. We followed best practices:

- we ask for permissions when entering the screen (the user has consciously selected the camera function),
- we offer an alternative (gallery) when there is no camera,
- we handle the denial of access to the gallery when saving (message),
- we inform the user about the result of the action (Alert after saving or error).

5.2 "MapScreen" screen – map with user location

Assumptions: This screen will display a map (MapView) with a marker indicating the user's current position and the caption "I am here." Upon entering the screen, the app should obtain location permission and retrieve the current coordinates. If denied, it will display a message stating that GPS access is denied.

By using react-native-maps we will display the map and use<Marker>to mark an item.

```
import React, { useState, useEffect } from 'react';
import { View, Text, StyleSheet, Button } from 'react-native';
import MapView, { Marker } from 'react-native-maps';
import * as Location from 'expo-location';

export default function MapScreen() {
  const [hasLocationPerm, setHasLocationPerm] = useState(null);
  const [location, setLocation] = useState(null);

  useEffect(() => {
    (async () => {
      const { status } = await Location.requestForegroundPermissionsAsync();
      setHasLocationPerm(status === 'granted');
      if (status === 'granted') {
        const loc = await Location.getCurrentPositionAsync({});
        setLocation(loc.coords);
      }
    })();
  }, []);

  if (hasLocationPerm === null) {
return <Text>Loading...</Text>;
  }
  if (hasLocationPerm === false) {
    return (
      <View style={styles.center}>
<Text>GPS location access denied.</Text>
<Text>To see your position on the map, enable location permissions.</Text>
      </View>
    );
  }

// Once we have the location:
  const region = {
    latitude: location.latitude,
    longitude: location.longitude,
latitudeDelta: 0.005,
longitudeDelta: 0.005
  };

  return (
    <View style={styles.container}>
      {location ? (
        <MapView style={styles.map} initialRegion={region}>
<Marker coordinate={location} title="I am here" />
        </MapView>
      ) : (
        <View style={styles.center}>
<Text>Getting location...</Text>
        </View>
      )}
    </View>
  );
}

const styles = StyleSheet.create({
  container: { flex: 1 },
  map: { flex: 1 },
  center: { flex: 1, justifyContent: 'center', alignItems: 'center', padding: 20 }
});
```

Explanations:

- IN useEffectpleaseLocation.requestForegroundPermissionsAsyncIf the result isgranted, we immediately callgetCurrentPositionAsyncto retrieve the coordinates. We set them in the statelocation.
- If refused, we sethasLocationPermto false and we show a message in render. (You can add a button, e.g., "Try again" or instructions to open settings – here just static text).
- Once we have the data, we configure itregion– we gave a very small delta (0.005 ~ 0.5 km), so the map will be quite close.

- Render: untillocationis null (e.g. downloading is in progress), we show the text "Loading location...". When it is already there, we displayMapViewwith setinitialRegion.
- Marker receivescoordinate={location}(i.e., the object {latitude, longitude}) and the title "I am here." We don't provide a description – the name is enough.
- Stylemap: { flex: 1 }makes the map fill the entire screen.

This implementation assumes a one-time location fetch. In practice, if the user moves and wants to update, you would need to add, for example, a refresh (a "Refresh" button that triggers the update again)getCurrentPositionAsyncand setLocation). You could also usewatchPositionAsyncand update the marker in real time, but this generates a lot of updates and may not be needed in the demo.

Action: Upon entering MapScreen, if consent has already been granted (e.g., the user has previously granted it), they will immediately see a map with a pin. If not, a system dialog will appear – after selecting "Allow," the component will re-render (the status will change to granted) and the map should appear. If they select "Do not allow," they will see our message. The application can continue running, just without the map (we could possibly display a map of the center of the country as a background, but that's a design decision – here we simply don't show anything except information).

Attention: In app.json for iOS we should have the NSLocationWhenInUseUsageDescription key (e.g. "The app uses your location to display your position on the map.") – otherwise the App Store would not let us through, and the system would display a default, undescriptive text.

Demo Summary

In the above two screens we have combined different Expo modules:

- **expo-camera**(Camera component) to take a photo.
- **expo-image-picker**as an alternative to selecting from the gallery.
- **expo-media-library**to save a photo to an album.
- **expo-location**to obtain GPS coordinates.
- **react-native-maps**to display the map and marker.

We implemented the permission logic according to best practices: contextual (on a given screen), with denial handling and user notification. The code also includes error handling elements (try/catch for photos, alerts for consent denials).

In a real application, these two screens could be part of tabs (e.g., using React Navigation – Tab Navigator: one tab "Camera," the other "Map"). This would require some minor navigation configuration, but that's a separate topic. The important thing is that the individual functionalities work independently.

Testing: Camera and maps features require a physical device or emulator with certain permissions:

- The camera will work on a real device. In the iOS simulator, the camera is unavailable – expo-camera will return an error or a black screen (you can test using the picker). In the Android emulator, you can activate the camera image (it acts as a virtual camera).
- Location: In the iOS simulator, you can set the Location (Debug -> Location -> Freeway Drive or Custom Location) to simulate GPS data. In the Android emulator, you can send coordinates via Extended Controls. Otherwise, getCurrentPosition may wait forever.
- Maps: Requires an internet connection (map download). Expo Go or dev builds require network access. If map tiles don't appear, you may be missing an API key (this shouldn't be a problem on Expo Go). When debugging custom builds, remember to include metadata from Google's API Key in the AndroidManifest or use Apple Maps on iOS.

Finally, we have an app that demonstrates practical use of Expo's device features and permissions. I hope this demo, along with the previous explanations, will make it easier to understand how to use cameras, galleries, files, location, and notifications in Expo/React Native in a controlled and user-friendly way. All the libraries presented in the latest Expo SDK (2025) offer the stable API we used. By following the permissions and user experience guidelines, apps can safely and effectively leverage the capabilities of mobile devices.

Literature:

1. https://docs.expo.dev/versions/latest/sdk/camera/(Access date: 1/10/2025) – Official module documentation**expo-camera**, describing the rules for obtaining permissions and operating a photo and video camera.

2. https://docs.expo.dev/versions/latest/sdk/location/(Access date: 1/10/2025) – Module documentation**expo-location**, discussing getting the current GPS location, tracking the user's position, and geocoding processes.

3. https://docs.expo.dev/versions/latest/sdk/notifications/(Access date: 1/10/2025) – Module Guide**expo-notifications**, containing detailed instructions for configuring local and remote (push) notifications.

4. https://github.com/react-native-maps/react-native-maps(Access date: 1/10/2025) – Library documentation**react-native-maps**, necessary for the integration of interactive Google and Apple maps and the support of markers in mobile applications.