POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 7

Funkcje urządzenia i uprawnienia w Expo/React Native

Mateusz Pawełkiewicz

1.10.2025

1. Uprawnienia w Expo (Permissions)

Aplikacje mobilne często muszą uzyskać **uprawnienia użytkownika** na dostęp do wrażliwych funkcji urządzenia (np. aparatu, lokalizacji, galerii zdjęć). W Expo i React Native uzyskiwanie uprawnień odbywa się za pomocą odpowiednich modułów Expo. Dawniej Expo oferowało uniwersalny moduł expo-permissions, jednak od SDK 41 został on **oznaczony jako przestarzały** na rzecz metod w poszczególnych modułach. Oznacza to, że zamiast używać np. Permissions.askAsync(Permissions.CAMERA) obecnie wywołujemy Camera.requestCameraPermissionsAsync() lub analogiczne metody w module, którego uprawnienie dotyczy. Każdy moduł Expo (kamera, lokalizacja, biblioteka mediów, powiadomienia itd.) udostępnia własne funkcje do sprawdzania i proszenia o wymagane zgody.

Żądanie uprawnień (request): Większość modułów Expo udostępnia asynchroniczną metodę request...PermissionsAsync(). Wyświetla ona natywne okienko systemowe z prośbą o pozwolenie. Przykładowo, aby uzyskać dostęp do aparatu, używamy:

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

const { status } = await Camera.requestCameraPermissionsAsync();
if (status !== 'granted') {
  alert('Brak dostępu do aparatu!');
  return;
}
// Uprawnienie przyznane – można korzystać z aparatu
```

Podobnie działa np. Location.requestForegroundPermissionsAsync() dla lokalizacji czy MediaLibrary.requestPermissionsAsync() dla biblioteki zdjęć (galerii). Gdy aplikacja wywoła taką funkcję po raz pierwszy, system (Android lub iOS) wyświetli użytkownikowi okno z prośbą o zgodę.

Sprawdzanie statusu (getPermissionsAsync): Każdy moduł posiada również metodę get...PermissionsAsync(), która umożliwia sprawdzenie obecnego statusu uprawnienia bez wyświetlania okienka. Zwracany status to zwykle jedna z wartości: "granted" (przyznane), "denied" (odmówione) lub "undetermined" (jeszcze nie pytano). Przykład użycia:

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

Często można najpierw wywołać getPermissionsAsync() – jeśli zwróci, że status to "undetermined", wtedy wywołać requestPermissionsAsync(). Jednak wywołanie request… od razu jest również poprawne – jeżeli uprawnienie było już nadane wcześniej, funkcja zwróci od razu status "granted" bez ponownego pytania użytkownika.

Dedykowane moduły i uprawnienia: Poniżej wymieniono kluczowe moduły Expo i odpowiadające im uprawnienia:

- **Kamera (expo-camera)** – wymaga uprawnienia do aparatu (Camera); przy nagrywaniu video z dźwiękiem potrzebny też dostęp do mikrofonu. Metody: Camera.requestCameraPermissionsAsync(), Camera.requestMicrophonePermissionsAsync().

- **Biblioteka Mediów (expo-media-library)** – dostęp do galerii zdjęć na urządzeniu. Uprawnienie do odczytu/zapisu zdjęć (i filmów) na urządzeniu. Metody: MediaLibrary.requestPermissionsAsync() (na iOS domyślnie prosi o odczyt *i* zapis; można rozdzielić przez parametr writeOnly).
- **Wybieranie obrazów (expo-image-picker)** – moduł ten wewnętrznie korzysta z uprawnień kamery lub biblioteki mediów. Posiada metody: ImagePicker.requestCameraPermissionsAsync() oraz ImagePicker.requestMediaLibraryPermissionsAsync() do pobrania zgód od użytkownika.
- **Lokalizacja (expo-location)** – wymaga pozwolenia na lokalizację: *foreground* (podczas używania aplikacji) i opcjonalnie *background* (w tle) – o tym niżej. Metody: Location.requestForegroundPermissionsAsync(), Location.requestBackgroundPermissionsAsync().
- **Powiadomienia (expo-notifications)** – na iOS wymagane jest uzyskanie zgody na wyświetlanie powiadomień. Na Androidzie do Android 12 włącznie zgoda nadawana była automatycznie przy instalacji, ale od Androida 13 również wprowadzono runtime permission na powiadomienia. Metoda: Notifications.requestPermissionsAsync() (można przekazać opcje typu alert/dźwięk/ikonka odznaki do wyświetlania).

Różnice między platformami (Android vs iOS)

Uprawnienia działają nieco inaczej na **Androidzie i iOS**:

- **iOS:** Każde żądanie uprawnień musi być opatrzone wyjaśnieniem w pliku Info.plist aplikacji, dlaczego potrzebujemy danej zgody. W Expo konfigurujemy to w app.json lub przez pluginy – np. dla aparatu klucz NSCameraUsageDescription, dla lokalizacji NSLocationWhenInUseUsageDescription itp. Domyślne komunikaty są dodawane automatycznie przez Expo, ale powinno się je dostosować do kontekstu aplikacji. Użytkownik na iOS może przyznać uprawnienie (Allow) lub odmówić (Don't Allow) za pierwszym razem – jeśli odmówi, kolejne wywołania requestPermissionsAsync() **nie pokażą już ponownie** dialogu, a status pozostanie "denied" (iOS zakłada, że użytkownik nie chce być ponownie pytany). W takiej sytuacji można jedynie zasugerować użytkownikowi zmianę ustawień ręcznie. Niektóre uprawnienia na iOS mają *różne poziomy*: np. lokalizacja ma *When In Use* (tylko podczas używania aplikacji) vs *Always* (również w tle). Poproszenie o **background location** wymaga najpierw zgody na "podczas użycia", a potem dodatkowego dialogu o "Always". iOS może też oferować opcję udostępnienia przybliżonej lokalizacji zamiast dokładnej (od iOS 14) – aplikacja nie ma na to wpływu poza określeniem wymogu *Accuracy* (system i tak pozwoli użytkownikowi zdecydować czy chce udostępniać dokładne dane GPS).
- **Android:** Uprawnienia dzielą się na zwykłe i "niebezpieczne" – te drugie wymagają zgody runtime. Android pozwala użytkownikowi zaznaczyć "Nie pytaj ponownie" przy odmawianiu – wówczas metoda request... zwróci status "denied" oraz pole canAskAgain = false, co oznacza, że nie wolno już pokazywać dialogu (kolejne prośby będą automatycznie odrzucane). Trzeba wtedy, podobnie jak na iOS, poprowadzić użytkownika do ustawień aplikacji. Android od wersji 11 wprowadził bardziej szczegółowe uprawnienia do plików i mediów: np. osobno **READ_MEDIA_IMAGES** i **READ_MEDIA_VIDEO** (zamiast ogólnego READ_EXTERNAL_STORAGE), a od Androida 13 pojawiło się natywne okno systemowego selektora zdjęć jako preferowana metoda. Expo ImagePicker w najnowszych wersjach dostosował się do tych zmian –

jeśli korzystamy z systemowego picker'a, możemy ograniczyć zakres dostępu do mediów zamiast prosić o pełne uprawnienie do czytania wszystkich plików (to pomaga spełnić wymagania Google Play dotyczące dostępu do zdjęć). W praktyce, wywołując ImagePicker.launchImageLibraryAsync na Androidzie 13+, aplikacja może nie potrzebować w ogóle zgody READ_MEDIA jeśli użyty zostanie systemowy picker – Expo automatycznie może to obsłużyć w nowszych SDK. Innym przykładem zmian na Androidzie 13 jest uprawnienie **POST_NOTIFICATIONS** – teraz aplikacja musi poprosić o zgodę na wyświetlanie powiadomień push (wcześniej użytkownik zgadzał się instalując aplikację). Expo Notifications.requestPermissionsAsync() uwzględnia to i poprosi o tę zgodę na Androidzie 13+.

- **Automatyczna konfiguracja:** Expo stara się automatycznie dodać większość niezbędnych deklaracji uprawnień do natywnych plików konfiguracyjnych. Gdy dodajemy moduł jak expo-camera czy expo-location i zbudujemy aplikację, to wymagane <uses-permission> w AndroidManifest.xml czy klucze Info.plist są zwykle dodawane przez tzw. config plugins Expo. Na przykład, do AndroidManifest zostanie wpisane android.permission.CAMERA przy użyciu expo-camera, a do Info.plist dodany zostanie domyślny tekst *"Allow $(PRODUCT_NAME) to access your camera"*. Programista może dodatkowo **usunąć** niechciane uprawnienia (jeśli pakiet dodaje coś zbędnego) za pomocą android.blockedPermissions w app.json – np. zablokowanie RECORD_AUDIO jeśli używamy aparatu tylko do zdjęć, aby nie prosić niepotrzebnie o mikrofon.

Dobre praktyki UX przy proszeniu o uprawnienia

Prośba o uprawnienia to moment, w którym użytkownik może zdecydować, czy zaufa naszej aplikacji w danym zakresie. Kilka wskazówek, jak robić to z poszanowaniem UX:

- **Pytaj tylko wtedy, gdy to potrzebne:** Nie wyświetlaj szeregu dialogów zaraz po uruchomieniu aplikacji. Zamiast tego, poproś o dane uprawnienie tuż przed funkcją, która go wymaga. Np. przed zrobieniem zdjęcia pokaż dialog prośby o aparat, a przed zapisaniem pliku – dialog dostępu do plików. Użytkownik lepiej rozumie kontekst prośby, gdy jest ona powiązana z jego akcją (np. tapnął „Zrób zdjęcie" to logiczne, że pojawia się pytanie o aparat).
- **Komunikuj się jasno i zwięźle:** Systemowe okno i tak wyświetli tekst z Info.plist/Manifestu, więc upewnij się, że ten komunikat jest zrozumiały (np. „Pozwól aplikacji XYZ na dostęp do aparatu, aby umożliwić robienie zdjęć"). W samym interfejsie aplikacji nie zaszkodzi zapowiedzieć użytkownikowi, dlaczego pojawi się pytanie. Można np. mieć własny ekran z informacją „Aby dodać zdjęcie, potrzebujemy dostępu do Twojego aparatu. Za chwilę wyświetli się prośba systemowa." – to nie zawsze konieczne, ale bywa pomocne, zwłaszcza przy bardziej wrażliwych danych.
- **Szanuj decyzję użytkownika i oferuj alternatywę:** Jeśli użytkownik odmówi uprawnienia, nie zmuszaj go w kółko do akceptacji. Zamiast tego, zaadaptuj się. Przykładowo, jeśli nie ma zgody na lokalizację GPS – umożliw użytkownikowi ręczne wpisanie adresu lub po prostu pokaż statyczną mapę domyślnej lokacji. Gdy brak dostępu do aparatu – pozwól wybrać zdjęcie z galerii (bo być może to uprawnienie chętniej przyzna), albo przekaż komunikat, że funkcja robienia zdjęć będzie

niedostępna. Ważne, by aplikacja nadal była użyteczna, nawet z ograniczonymi uprawnieniami.

- **Ponowne próby i ustawienia:** Jeśli użytkownik odmówił i zaznaczył "nie pytaj ponownie" (Android) lub po prostu odmówił na iOS (gdzie domyślnie to oznacza "nie pytaj ponownie"), jedyną opcją jest poproszenie go o zmianę zdania w ustawieniach systemowych. Możesz wyświetlić komunikat w stylu: „Funkcja X jest wyłączona, ponieważ nie nadano uprawnień. Możesz je włączyć w ustawieniach aplikacji.” i np. przycisk „Otwórz Ustawienia”. Expo nie ma specjalnego API do otwierania ustawień, ale można skorzystać z Linking.openSettings() z React Native, które przekieruje do ustawień aplikacji. Nie bombarduj jednak użytkownika takimi monitami – pokazuj je tylko jeśli funkcja jest naprawdę kluczowa w danym momencie.
- **Błędy i wyjątki:** Zawsze obsługuj obietnice (Promise) z funkcji pytających o uprawnienia. Użytkownik może nie tylko wybrać „Allow” lub „Deny”, ale czasem dialog może zostać przerwany, lub nastąpi inny błąd. Kod powinien przewidywać, że requestPermissionsAsync() może rzucić wyjątek lub zwrócić obiekt ze statusem, który nie jest "granted". Dlatego wstawiaj warunki if (status !== 'granted') i reaguj na nie (np. przerwij wykonywanie danej akcji jak pokazano wyżej w przykładzie z aparatem). Zapobiegnie to sytuacjom, w których aplikacja próbuje użyć zasobu bez uprawnień i np. otrzymuje błąd lub (co gorsza) zawiesza się.

Stosując powyższe zasady, zwiększamy szansę, że użytkownik **przyzna uprawnienia** (bo rozumie po co są potrzebne), a nawet jeśli nie – aplikacja nadal będzie działać sensownie, zamiast frustrować komunikatami o błędach.

2. Kamera, galeria, pliki – multimedia w Expo

W tej części omówimy, jak w Expo/React Native korzystać z **aparatu urządzenia**, jak uzyskać zdjęcia z galerii, jak modyfikować i przechowywać pliki oraz jak je udostępniać. Wykorzystamy do tego następujące biblioteki Expo: expo-camera, expo-image-picker, expo-image-manipulator, expo-file-system, expo-media-library oraz expo-sharing. Wszystkie te bibliotki są częścią ekosystemu Expo SDK (w wersjach aktualnych na rok 2025).

2.1 Korzystanie z aparatu (expo-camera)

Moduł **expo-camera** pozwala na obsługę aparatu fotograficznego (zarówno przedniego, jak i tylnego) bezpośrednio w aplikacji. Umożliwia wyświetlenie podglądu na żywo z kamery jako komponent React oraz wykonanie zdjęcia lub nagranie wideo. Możliwe jest także dostosowanie parametrów aparatu (zoom, ostrość, balans bieli, włączenie lampy błyskowej itp.) oraz skanowanie kodów kreskowych/QR w czasie rzeczywistym.

Instalacja: Aby z niego skorzystać, instalujemy paczkę komendą npx expo install expo-camera. Następnie importujemy potrzebne elementy, np. komponent Camera lub hook useCameraPermissions itp. Ponieważ aparat to funkcja wymagająca uprawnień prywatności, **przed użyciem musimy uzyskać zgodę użytkownika** na dostęp do kamery (oraz mikrofonu, jeśli planujemy nagrywać audio wraz z wideo).

Podgląd z kamery: expo-camera udostępnia komponent React <Camera> do wstawienia w drzewo renderowania. Typowe użycie to umieszczenie go na części (lub całości) ekranu wraz z przyciskiem wyzwalacza migawki. Przykład uproszczonego komponentu korzystającego z kamery:

```
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 zrobionego zdjęcia do podglądu

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

 if (hasPermission === null) {
  return <Text>Proszę czekać...</Text>; // w trakcie pytania o uprawnienie
 }
 if (hasPermission === false) {
  return <Text>Brak dostępu do aparatu.</Text>; // odmowa uprawnień
 }

 const takePhoto = async () => {
  try {
   const result = await cameraRef.current.takePictureAsync({
    quality: 0.8, // jakość zdjęcia (0-1)
    base64: false, // można true jeśli chcemy base64 (np. do podglądu miniatury)
    skipProcessing: false // domyślnie false, jeśli true to pomija post-process (np. rotację)
   });
   setPhoto(result.uri);  // zapisz URI zdjęcia do stanu
  } catch (e) {
   console.error('Błąd wykonania zdjęcia', e);
  }
 };

 return (
  <View style={styles.container}>
   {photo ? (
    // Jeśli zrobiono zdjęcie, pokazujemy jego podgląd
    <>
     <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>Wyślij</Text>
      </TouchableOpacity>
     </View>
    </>
```

```
    ) : (
      // Jeśli nie zrobiono jeszcze zdjęcia, wyświetlamy podgląd z kamery i przycisk
      <>
        <Camera style={styles.camera} ref={cameraRef} type={Camera.Constants.Type.back} />
        <TouchableOpacity onPress={takePhoto} style={styles.captureButton}>
          <Text style={styles.captureText}>Zrób zdjęcie</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 }
});
```

Powyższy kod ilustruje podstawy: pytamy o uprawnienia do aparatu (w useEffect na starcie komponentu), następnie jeśli brak uprawnień – informujemy o tym. Gdy mamy zgodę, wyświetlamy <Camera ref={...}>. Używamy referencji cameraRef żeby odwołać się do metody takePictureAsync() komponentu kamery. Po naciśnięciu przycisku "Zrób zdjęcie" wykonujemy zdjęcie i otrzymujemy obiekt zawierający m.in. uri zdjęcia. Używamy tego URI, aby pokazać podgląd (<Image source={{ uri: photo }} />). Dodaliśmy też dwa przyciski: "Retake" (usuwa podgląd i wraca do trybu kamery) oraz "Wyślij" (tu tylko wyświetlamy alert symulujący wysyłkę).

Uwagi: takePictureAsync pozwala podać opcje takie jak jakość (0-1), czy zwrócić obraz jako base64, czy pominąć przetwarzanie. Na iOS np. expo-camera automatycznie obraca zdjęcie zgodnie z orientacją urządzenia; skipProcessing: true może pominąć te kroki (przydatne np. do szybszego zrobienia zdjęcia kosztem braku rotacji). Zdjęcie wynikowe zostaje zapisane automatycznie w pamięci podręcznej aplikacji (tzw. cache). W result.uri otrzymujemy ścieżkę do pliku (np. file:///data/user/0/.../somefilename.jpg). **Jeśli chcemy zachować to zdjęcie na dłużej**, powinniśmy przenieść je w trwałe miejsce (np. do FileSystem.documentDirectory lub do galerii urządzenia – o tym w sekcji 2.4).

Możemy także zmieniać **typ kamery** (przednia/tylna) dynamicznie – w naszym przykładzie użyliśmy Camera.Constants.Type.back. Można np. dodać przycisk "Switch camera", który ustawia type na Camera.Constants.Type.front lub back. Expo-camera obsługuje również lampę błyskową (flashMode), zoom (zoom), autofocus itd., poprzez propsy przekazywane do komponentu <Camera>.

2.2 Wybór zdjęcia z galerii lub szybkie zdjęcie (expo-image-picker)

Nie każda aplikacja potrzebuje pełnego podglądu kamery i własnego interfejsu do robienia zdjęć. Czasem wygodniej jest skorzystać z natywnego selektora mediów – takiego, który pozwala użytkownikowi wybrać istniejące zdjęcie z galerii, albo uruchamia domyślną aplikację aparatu, a po zrobieniu zdjęcia wraca do naszej aplikacji. Do tego służy **expo-image-picker**.

Instalacja: npx expo install expo-image-picker. Ten moduł kryje w sobie zarówno funkcje do otwarcia **galerii** urządzenia, jak i do uruchomienia natywnej aplikacji **aparatu**. W obu przypadkach po wykonaniu akcji (wybór zdjęcia lub zrobienie nowego) wynik jest przekazywany do naszej aplikacji.

Uprawnienia: W przypadku expo-image-picker musimy zadbać o odpowiednie uprawnienia:

- Jeśli chcemy wybierać pliki z biblioteki zdjęć, potrzebujemy zgody na dostęp do mediów (Photo/Media Library). Na Androidzie (do 12) było to READ/WRITE_EXTERNAL_STORAGE, na Androidzie 13 – wspomniane READ_MEDIA_IMAGES, a na iOS – dostęp do Photos (z opisem w Info.plist). Expo-image-picker udostępnia metodę requestMediaLibraryPermissionsAsync() do wywołania przed otwarciem picker'a. Można przekazać writeOnly: true jeśli planujemy tylko zapisywać (np. zapisać zdjęcie do galerii) bez czytania istniejących – wtedy na iOS np. poprosi tylko o ograniczone uprawnienie zapisu.
- Jeśli chcemy użyć aparatu poprzez image-picker (czyli otworzyć natywną apkę aparatu), potrzebujemy uprawnienia do Camera. Tutaj używamy requestCameraPermissionsAsync() z expo-image-picker (wewnątrz działa podobnie jak w expo-camera). Dodatkowo, na starszych Androidach i iOS 10 potrzebny był też dostęp do "camera roll" (biblioteki), ale na nowszych systemach z reguły natywna aplikacja aparatu zapisuje zdjęcie do swojej lokalizacji i zwraca do naszej appki bez dodatkowych wymagań – expo-image-picker abstrakcyjnie tym zarządza.

Użycie: expo-image-picker oferuje dwie główne funkcje:

- ImagePicker.launchImageLibraryAsync(options) – otwiera systemową galerię/pliki, pozwala wybrać zdjęcie lub film.
- ImagePicker.launchCameraAsync(options) – otwiera aparat (natywny UI do zrobienia zdjęcia).

Obie funkcje zwracają Promise, który po zakończeniu zwraca obiekt wyniku. W najnowszych wersjach expo-image-picker wynik ma strukturę zawierającą pole canceled (boolean) oraz assets (tablica obiektów media). Dla pojedynczego zdjęcia będzie to tablica jednoelementowa, z obiektem posiadającym m.in. uri (ścieżka do pliku zdjęcia), width, height, type (typ mediów), ewentualnie base64 (jeśli zażądano) i exif (jeśli zażądano metadanych). Dla prostoty, możemy traktować to tak, że dostajemy result.assets[0].uri jako ścieżkę do wybranego zdjęcia.

Przykład: użytkownik chce wybrać awatar z galerii:

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

async function pickImageFromGallery() {
  // Najpierw upewnijmy się, że mamy uprawnienie do czytania mediów
  const perm = await ImagePicker.requestMediaLibraryPermissionsAsync();
  if (!perm.granted) {
    alert("Aplikacja potrzebuje dostępu do Twoich zdjęć, aby wybrać obraz.");
    return;
  }
  // Otwórz galerię i pozwól użytkownikowi wybrać obraz
  const result = await ImagePicker.launchImageLibraryAsync({
    mediaTypes: ImagePicker.MediaTypeOptions.Images, // tylko zdjęcia (nie wideo)
    allowsEditing: false,  // true dałoby możliwość przycięcia zdjęcia w interfejsie systemowym
    quality: 1            // skala 0-1, 1 = oryginalna jakość
  });
  if (result.canceled) {
    console.log("Użytkownik anulował wybór obrazka");
    return;
  }
  const selectedAsset = result.assets[0];
  console.log("Wybrano zdjęcie:", selectedAsset.uri,
          "wymiary:", selectedAsset.width, "x", selectedAsset.height);
  // tutaj można np. ustawić stan z tym zdjęciem, żeby wyświetlić podgląd
}
```

W powyższym fragmencie najpierw żądamy zgody (pokazane jest obsłużenie sytuacji, gdy użytkownik odmówi – wyświetlamy alert i przerywamy). Jeśli zgoda jest, wywołujemy launchImageLibraryAsync. W opcjach określamy, że interesują nas obrazy (nie np. filmy) i czy pozwolić na edycję. Opcja allowsEditing: true spowodowałaby, że po wybraniu zdjęcia użytkownik dostałby możliwość przycięcia go (do kwadratu) w wbudowanym edytorze. Ta funkcja korzysta z natywnych mechanizmów i np. na iOS zwróci zawsze obraz JPEG nawet jeśli źródło było HEIC, natomiast na Androidzie ma pewne ograniczenia (np. przy allowsEditing:true i quality<1 animowane GIFy zostaną zredukowane do statycznego obrazu). W naszym przykładzie wyłączamy edycję i pobieramy pełną jakość.

Analogicznie, możemy zrobić **zdjęcie z aparatu** bez wbudowywania komponentu kamery:

```
async function takePhotoViaSystemCamera() {
  // Upewnij się o zgodzie na aparat:
  const permCam = await ImagePicker.requestCameraPermissionsAsync();
  if (!permCam.granted) {
    alert("Brak dostępu do aparatu.");
    return;
  }
  // (Opcjonalnie: na Androidzie 10- prosba o MediaLibrary moze byc potrzebna
  // ale expo-image-picker sam o to zadba jeśli konieczne)
  // Uruchom natywną aplikację aparatu:
  const result = await ImagePicker.launchCameraAsync({
    allowsEditing: true,
    quality: 0.5, // zmniejsz jakość do 50% dla mniejszego pliku
    base64: false // można true jeśli chcemy uzyskać też base64 obrazu
  });
  if (!result.canceled) {
    const photo = result.assets[0];
```

```
    console.log("Zrobiono zdjęcie:", photo.uri);
    // np. ustaw zdjęcie w stanie, aby wyświetlić podgląd w UI:
    setPhotoUri(photo.uri);
  }
}
```

Tutaj po uzyskaniu uprawnienia do kamery, wywołujemy launchCameraAsync. Natywny aparat zostanie otwarty (poza naszą aplikacją, jako systemowy widok). Po zrobieniu zdjęcia użytkownik zwykle ma opcję **akceptuj** lub **ponów** (to zapewnia system). Gdy zaakceptuje, wraca do naszej aplikacji, a launchCameraAsync rozwiązuje Promise zwracając obiekt zdjęcia. W tym przykładzie włączyliśmy allowsEditing:true, co oznacza, że użytkownik po zrobieniu zdjęcia dostanie możliwość np. przycięcia lub potwierdzenia zdjęcia (na iOS pokaże okienko z możliwością przesunięcia/skalowania). Z ustawioną jakością 0.5 uzyskamy zdjęcie skompresowane (mniejszy rozmiar pliku, kosztem jakości).

Co dalej z wybranym zdjęciem? W obu przypadkach (galeria lub aparat) otrzymujemy *uri* pliku lokalnego. Możemy od razu wyświetlić obraz (np. w <Image source={{uri: ...}} />), bo ten plik jest lokalnie dostępny dla naszej aplikacji. Jeśli to zdjęcie ma być wysłane na serwer, można je przekazać dalej (np. w formularzu lub przez upload via fetch). Jeśli chcemy je zapisać do pamięci trwałej aplikacji lub do galerii – patrz sekcja 2.4 poniżej.

Uwaga dot. iOS (ograniczony dostęp do zdjęć): Na iOS od wersji 14, gdy prosimy o dostęp do biblioteki, użytkownik może wybrać *ograniczony dostęp* – czyli pozwolić tylko do wybranych zdjęć. Expo-image-picker stara się to obsłużyć – np. jeśli użytkownik da dostęp ograniczony, to przy próbie launchImageLibraryAsync pojawi się natywne okno wybierania tych dozwolonych zdjęć. Dla dewelopera oznacza to, że requestMediaLibraryPermissionsAsync() może zwrócić status "granted" nawet jeśli dostęp jest ograniczony, a pole accessPrivileges może wskazywać limited. W przypadku ograniczonego dostępu, użytkownik może nie móc wybrać dowolnego zdjęcia, tylko te wcześniej wybrane. W razie potrzeby można wykryć perm.accessPrivileges === "limited" i np. wyświetlić komunikat zachęcający do nadania pełnego dostępu w ustawieniach, jeśli dana funkcjonalność tego wymaga.

2.3 Przycinanie i kompresja obrazów (expo-image-manipulator)

Często po zrobieniu lub wybraniu zdjęcia chcemy je **przekształcić** – np. zmniejszyć rozdzielczość (żeby zaoszczędzić transfer danych przy wysyłaniu), obrócić, przyciąć do kwadratu lub zmienić format/kompresję. Expo oferuje bibliotekę **expo-image-manipulator**, która pozwala te operacje wykonać na urządzeniu.

Instalacja: npx expo install expo-image-manipulator.

Biblioteka ta umożliwia manipulację obrazem zapisanym w pliku lokalnym (lub base64). Kluczowa funkcja (w starszym stylu API) to ImageManipulator.manipulateAsync(uri, actions, saveOptions). Przyjmuje ona:

- uri – lokalizację pliku źródłowego (np. photo.uri zrobione aparatem lub wybrane z pickera).

- actions – tablicę obiektów opisujących działania (np. resize, rotate, flip, crop). Każdy z tych obiektów ma klucz określający operację i wartości.
- saveOptions – obiekt z opcjami zapisu wyniku (format pliku, kompresja, czy dołączyć base64).

Funkcja zwraca Promise z obiektem zawierającym m.in. uri nowo utworzonego pliku (z zmodyfikowanym obrazem), width, height i opcjonalnie base64 (jeśli zażądano).

Przykład: załóżmy, że chcemy wziąć zdjęcie użytkownika (np. zrobione aparatem) i **zmniejszyć je oraz skompresować** przed wysłaniem na serwer, aby ograniczyć rozmiar pliku. Możemy to zrobić tak:

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

// ... załóżmy że mamy photo.uri z aparatu o rozdzielczości np. 4000x3000

const manipResult = await ImageManipulator.manipulateAsync(
  photo.uri,
  [{ resize: { width: 1000 } }], // actions: zmień rozmiar do szerokości 1000px, wysokość proporcjonalnie
  { compress: 0.7, format: ImageManipulator.SaveFormat.JPEG }
);
console.log("Nowy plik:", manipResult.uri, "rozmiar:",
        manipResult.width, "x", manipResult.height);
```

Powyżej przekazujemy jedną akcję: resize do szerokości 1000 pikseli (wysokość zostanie dobrana automatycznie, zachowując proporcje). W saveOptions ustawiamy compress: 0.7 (70% jakości, czyli umiarkowana kompresja JPEG) oraz format: JPEG (można też PNG lub WEBP). Wynikiem będzie nowy plik JPEG o mniejszej rozdzielczości, którego URI dostajemy w manipResult.uri. Taki plik możemy teraz np. wysyłać na serwer – będzie znacząco mniejszy niż oryginalne zdjęcie.

Inne akcje dostępne w ImageManipulator:

- rotate: degrees – obraca obraz o podany kąt (90, 180, 270, ...).
- flip: FlipType.Horizontal lub FlipType.Vertical – lustrzane odbicie w poziomie lub pionie.
- crop: { originX, originY, width, height } – wycina prostokąt z obrazu o podanych współrzędnych początkowych i rozmiarach. Np. by przyciąć środek obrazu do kwadratu 1000x1000, musielibyśmy obliczyć odpowiedni originX/Y.
- *(Od SDK 49+)* extend (rozszerzenie płótna) – mniej typowe, pozwala np. dodać margines wokół obrazka.

Należy zauważyć, że manipulateAsync **tworzy nowy plik** przy każdym wywołaniu (nie nadpisuje oryginału, bo iOS/Android mają mechanizmy cache'owania obrazów po ścieżce). Jeśli wielokrotnie będziemy manipulować, może być sens usuwania plików tymczasowych po użyciu (np. FileSystem.deleteAsync starych plików).

Nowy interfejs API: Dokumentacja Expo wspomina, że od pewnego czasu manipulateAsync jest oznaczone jako *deprecated*, a zalecane jest używanie nowego API opartego o kontekst (hook useImageManipulator i obiekty obrazów). Nowe API pozwala łańcuchowo wywoływać

manipulacje i ma nieco inny styl (bardziej obiektowy). Jednak dla prostych zastosowań (jak powyżej) wciąż można śmiało używać ImageManipulator.manipulateAsync, ponieważ jest proste i skuteczne. W kontekście tego wykładu koncentrujemy się na tej prostszej formie.

2.4 Zapisywanie plików lokalnie (expo-file-system i expo-media-library)

Kiedy dysponujemy plikiem (np. zdjęciem) w aplikacji Expo, możemy chcieć go **zapisać trwale** lub udostępnić użytkownikowi poza aplikacją. Mamy dwie główne ścieżki:

- zapisać plik w wewnętrznym systemie plików aplikacji (tzw. sandbox, niedostępny bezpośrednio z poziomu innych aplikacji),
- zapisać plik do publicznej biblioteki urządzenia (np. zdjęcie do galerii, plik do katalogu publicznego), tak by użytkownik mógł go zobaczyć poza naszą aplikacją.

Expo FileSystem: Biblioteka expo-file-system umożliwia dostęp do systemu plików **wewnątrz sandboxu aplikacji**. Domyślnie expo udostępnia ścieżki takie jak:

- FileSystem.documentDirectory – katalog trwały dla aplikacji (dane tu pozostają, dopóki użytkownik nie odinstaluje aplikacji lub ich sam nie usunie).
- FileSystem.cacheDirectory – katalog tymczasowy (cache), który system może czyścić w razie potrzeby.

Funkcjami takimi jak FileSystem.moveAsync, copyAsync, writeAsStringAsync, readAsStringAsync, deleteAsync itd., możemy manipulować plikami.

Przykład: zrobiliśmy zdjęcie aparatem i otrzymaliśmy np. photo.uri = "file:///.../Camera/abc.jpg" (lokalizacja w cache Expo). Jeśli chcemy je zachować np. w naszym folderze dokumentów pod nazwą "profile.jpg", możemy to zrobić:

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

const sourceUri = photo.uri;  // ścieżka źródłowa (plik w cache)
const destUri = FileSystem.documentDirectory + "profile.jpg";
await FileSystem.copyAsync({ from: sourceUri, to: destUri });
console.log("Plik skopiowany do dokumentów:", destUri);
```

Powyższe skopiuje plik. Można użyć moveAsync zamiast copyAsync, jeśli chcemy przenieść (usunąć z źródła). W ten sposób plik będzie dostępny na przyszłość pod znanym nam adresem (np. możemy go potem wczytać i wyświetlić później, nawet po ponownym uruchomieniu aplikacji). Wewnętrzna pamięć aplikacji jest odizolowana – np. na Androidzie to zwykle /data/data/<package>/files/..., na iOS Documents/... dla aplikacji – i nie pojawi się to automatycznie w galerii użytkownika czy menedżerze plików.

Expo MediaLibrary: Jeśli chcemy, aby plik (np. zdjęcie) trafił do **galerii zdjęć użytkownika**, powinniśmy użyć expo-media-library. Ta biblioteka integruje się z systemową biblioteką multimediów (aplikacja **Zdjęcia** na iOS lub biblioteka mediów na Androidzie). Pozwala m.in. odczytywać zdjęcia/albumy, ale także zapisywać pliki do albumu. Żeby z niej skorzystać, potrzebne jest uprawnienie do zapisu/odczytu mediów (o czym mówiliśmy w sekcji uprawnień). Zakładamy, że użytkownik wyraził zgodę.

Aby zapisać zdjęcie do galerii, używamy funkcji MediaLibrary.createAssetAsync(uri). Np.:

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

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

Pierwsza linia tworzy *asset* ze wskazanego pliku URI – to znaczy dodaje zdjęcie do systemowej bazy mediów (w domyślnym albumie, zazwyczaj *Camera Roll*). Zwraca obiekt assetu (zawierający m.in. unikalny ID, typ, URI itp.). Druga linia tworzy album o nazwie „MojaAplikacja" i przenosi ten asset do niego (parametr false oznacza, że jeśli album już istnieje, nie duplikuj pliku). Można pominąć createAlbumAsync, jeśli chcemy po prostu wrzucić do domyślnej lokalizacji.

Expo-media-library automatycznie zajmie się tym, żeby np. na iOS zapisać obraz do Photos (co użytkownik zobaczy w aplikacji Zdjęcia), a na Androidzie umieścić plik w DCIM/ lub Pictures/.

Uwaga: Od Androida 10+ wprowadzono koncept **Media Store**, więc expo-media-library korzysta z tego API by zapisywać pliki. Wymaga to deklaracji uprawnień jak READ/WRITE_MEDIA_IMAGES (co Expo robi automatycznie, choć z uwagi na politykę Google, jeśli nasze użycie jest sporadyczne, można rozważyć użycie systemowego pickera zamiast pełnych uprawnień). Niemniej, jeśli potrzebujemy programowo zapisywać pliki, expo-media-library jest właściwym narzędziem.

Podsumowując: expo-file-system służy do zarządzania plikami wewnątrz aplikacji (cache, dokumenty), a expo-media-library pozwala na interakcję z galerią użytkownika. W typowym scenariuszu:

- używamy expo-camera lub image-picker -> dostajemy plik w cache,
- jeśli użytkownik zapisuje zdjęcie – pytamy o zgodę (jeśli nie była dana) i używamy MediaLibrary, by zapisać do galerii,
- dodatkowo albo zamiast tego, możemy trzymać zdjęcie w sandboxie aplikacji (jeśli to np. prywatne dane tylko dla naszej aplikacji).

2.5 Udostępnianie plików (systemowy „share sheet" z expo-sharing)

Często chcemy umożliwić użytkownikowi **udostępnienie** zdjęcia lub pliku poprzez inne aplikacje – np. wysłać zdjęcie mailem, poprzez komunikator, zapisać na Dysku itp. Systemy mobilne udostępniają tzw. *share sheet* – standardowe okno „Udostępnij", które pokazuje listę aplikacji i akcji możliwych dla danego pliku. W Expo możemy wywołać to okno przez bibliotekę **expo-sharing**.

Instalacja: npx expo install expo-sharing.

Expo-sharing ma bardzo prosty interfejs: najpierw warto sprawdzić, czy udostępnianie jest dostępne przez Sharing.isAvailableAsync(). Na platformach mobilnych natywnych będzie dostępne zawsze; na web może nie być (Web Share API jest ograniczone). Następnie

używamy Sharing.shareAsync(url, options) – gdzie url to **URI pliku lokalnego**, który chcemy udostępnić.

Załóżmy, że mamy ścieżkę do obrazka imageUri (np. zdjęcie zrobione aparatem, zapisane w plikach aplikacji). Możemy udostępnić je tak:

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

async function shareFile(uri) {
  const canShare = await Sharing.isAvailableAsync();
  if (!canShare) {
    alert("Udostępnianie nie jest dostępne na tej platformie");
    return;
  }
  try {
    await Sharing.shareAsync(uri);
    console.log("Plik został udostępniony.");
  } catch (error) {
    console.error("Błąd udostępniania:", error);
  }
}
```

Wywołanie shareAsync spowoduje otwarcie natywnego panelu udostępniania. Użytkownik zobaczy listę aplikacji, do których może wysłać ten plik – np. na iOS pojawią się ikonki AirDrop, iMessage, Mail, Zapisywanie grafiki itd., na Androidzie np. Gmail, Messenger, Drive itp. Gdy wybierze jedną i udostępnianie się powiedzie, Promise się rozwiąże (nie dostajemy może zbyt wielu szczegółów, zwykle nie wiemy czy adresat odebrał itp., tylko że nasz plik został przekazany do systemu).

Ograniczenia: Expo-sharing działa tylko z plikami lokalnymi i na platformach natywnych. W przeglądarzu web (PWA) **Web Share API** pozwala udostępniać tylko pewne typy danych i wymaga HTTPS – expo-sharing wykorzystuje go jeśli dostępny. Jednak przeglądarki nie pozwalają udostępniać lokalnych plików z URI bezpośrednio, więc ta funkcja na web może być niepraktyczna (trzeba by wcześniej np. wgrać plik na jakiś URL). Generalnie expo-sharing jest najbardziej przydatny na Android/iOS. Nie umożliwia też *odbierania* udostępnionych treści z innych aplikacji (to znaczy, nasza aplikacja nie może zadeklarować „otwieraj ten typ pliku w mojej aplikacji” za pomocą expo-sharing – to wymaga natywnej konfiguracji i custom pluginów).

3. Lokalizacja i mapy

Kolejną ważną funkcjonalnością urządzeń jest **geolokalizacja** – ustalanie pozycji GPS użytkownika – oraz prezentacja tej pozycji na mapach. W ekosystemie Expo mamy moduł expo-location do pozyskiwania lokalizacji oraz możemy wykorzystać bibliotekę react-native-maps do wyświetlania map (Google Maps / Apple Maps) w aplikacji.

3.1 Pozyskiwanie lokalizacji (expo-location)

Moduł **expo-location** umożliwia odczyt **aktualnej lokalizacji GPS** urządzenia, subskrybowanie zmian położenia, a także geokodowanie (zamianę współrzędnych na adres i

odwrotnie). Korzystanie z niego, podobnie jak z innych, wymaga uprzedniego uzyskania uprawnień od użytkownika.

Uprawnienia lokalizacji: Wyróżniamy dwa rodzaje – **foreground location** (podczas używania aplikacji) i **background location** (w tle). W większości przypadków potrzebujemy tylko tej pierwszej (np. aby pokazać na mapie gdzie jest user lub wyszukać coś w okolicy). Background location jest potrzebna, gdy aplikacja ma śledzić położenie nawet, gdy jest nieaktywna (np. aplikacja treningowa rejestrująca bieg, tracker itp.). Zapytanie o background od razu jest możliwe na Androidzie (choć lepiej najpierw foreground, a potem background), a na iOS jest dwuetapowe jak wspomnieliśmy wcześniej. Expo udostępnia odpowiednio requestForegroundPermissionsAsync() i requestBackgroundPermissionsAsync(). **Na iOS**, by background w ogóle działała, trzeba dodać do Info.plist właściwość i włączyć tryb background – Expo umożliwia to przez konfigurację isIosBackgroundLocationEnabled: true w pluginie expo-location oraz musi być podany klucz NSLocationAlwaysAndWhenInUseUsageDescription. Bez tego i tak nie dostaniemy zgody od Apple.

W typowej aplikacji mapowej wystarczy foreground permission.

Pobieranie bieżącej pozycji: Najprostszą metodą jest Location.getCurrentPositionAsync(options). Wywołuje ona wewnętrznie odczyt z GPS i/lub innych źródeł (WiFi, sieć) i zwraca Promise z obiektem lokalizacji. Ten obiekt zawiera m.in. współrzędne coords.latitude i coords.longitude, dokładność coords.accuracy (w metrach), a także np. wysokość altitude, prędkość speed czy kierunek heading (gdy dostępne). Zawiera też znacznik czasu.

Przykład użycia (zakładając, że uprawnienie już mamy):

```
const location = await Location.getCurrentPositionAsync({
  accuracy: Location.Accuracy.High,  // można Balanced, Low etc. High używa GPS, może być wolniejsze
  maximumAge: 10000,                 // jeśli jest ostatnia znana pozycja nie starsza niż 10s, może ją dać od razu
});
console.log("Moje położenie: ", location.coords.latitude, location.coords.longitude);
```

Takie wywołanie włącza GPS (jeśli nie był włączony) i może potrwać chwilę (kilka sekund) zanim zwróci dokładną pozycję, szczególnie przy pierwszym uruchomieniu.

Śledzenie pozycji (watchPosition): Jeśli chcemy otrzymywać aktualizacje pozycji ciągle (np. co określony dystans lub czas), używamy Location.watchPositionAsync(options, callback). Ta funkcja **uruchamia subskrypcję** – będzie wywoływać nasz callback za każdym razem, gdy lokalizacja spełni kryteria zmiany. Zwraca obiekt subskrypcji (do późniejszego anulowania .remove()).

Przykład: trackowanie ruchu użytkownika:

```
const subscription = await Location.watchPositionAsync(
  {
    accuracy: Location.Accuracy.High,
    timeInterval: 5000,    // co najmniej co 5 sekund
    distanceInterval: 10   // lub co 10 metrów
  },
  (loc) => {
    const { latitude, longitude } = loc.coords;
```

```
    console.log(`Nowa pozycja: ${latitude}, ${longitude}`);
    // tutaj np. zaktualizuj stan aplikacji z nowymi współrzędnymi
  }
);
// ... później, gdy już nie potrzebujemy śledzić:
subscription.remove();
```

W powyższym kodzie co ~5 sekund lub co ~10 metrów (w zależności co nastąpi wcześniej) otrzymamy nową pozycję. Opcje timeInterval i distanceInterval pomagają ograniczyć zbyt częste odczyty (oszczędność baterii). Samo accuracy też wpływa – High włączy GPS, Balanced może korzystać z sieci co jest szybsze ale mniej dokładne, Low tylko sieć (celuje w dokładność kilku km).

Background location: Expo-location umożliwia także śledzenie pozycji w tle, nawet gdy aplikacja nie jest aktywna. Realizuje się to poprzez rejestrację zadania z użyciem modułu TaskManager (trzeba stworzyć task definicją w kodzie) i wywołanie Location.startLocationUpdatesAsync(taskName, options). Jeśli użytkownik nadał *Allow all the time* (Always) uprawnienie i konfiguracja natywna jest odpowiednia, aplikacja będzie otrzymywać aktualizacje w tle, a nawet w stanie zamkniętym (do pewnego stopnia – system może je ograniczać). To temat dość zaawansowany, dlatego tylko sygnalizujemy jego istnienie. Trzeba pamiętać, że użytkownik widzi np. na iOS niebieski pasek, że aplikacja używa lokalizacji w tle, co może budzić obawy – dlatego należy zawsze wyjaśnić po co to robimy. Wiele aplikacji w ogóle nie potrzebuje background location – korzystajmy z tego oszczędnie.

Geokodowanie: Wspomnę krótko, że expo-location ma też funkcje Location.reverseGeocodeAsync(coords) – zamienia współrzędne na czytelny adres (ulica, miasto itp.), oraz Location.geocodeAsync(address) – odwrotnie, adres na potencjalne współrzędne. To korzysta z usług platformy (iOS / Android) i wymaga połączenia internetowego (na iOS używa Apple Maps API, na Androidzie Google geocoding). W wielu wypadkach może być przydatne, ale należy pamiętać, że nie zawsze zwróci wynik (np. jak adres niejasny). Te funkcje nie wymagają specjalnych dodatkowych uprawnień poza tym, że jeśli korzystają z lokalizacji użytkownika to musimy mieć tę zgodę.

3.2 Wyświetlanie map (react-native-maps)

Do integracji map w React Native standardem jest biblioteka **react-native-maps**. Expo obsługuje ją (jest wymieniona jako kompatybilna, instalujemy przez expo install react-native-maps). Pozwala ona osadzić komponent MapView, który wyświetli mapy Google (na Androidzie i opcjonalnie na iOS) lub Apple Maps (domyślnie na iOS). Możemy na mapie umieszczać **markery**, kształty, obsługiwać zdarzenia (tapnięcia na mapie, przeciągnięcia itp.).

Instalacja i konfiguracja: W trybie Expo Go nie musimy nic więcej robić – Mapy Google/Apple powinny działać od razu (Expo dostarcza klucze API dla Expo Go). Natomiast dla własnych buildów na iOS często trzeba podać klucz Google Maps API, jeśli chce się używać Google jako dostawcy map (alternatywnie, można zostać przy Apple Maps na iPhone). Szczegóły konfiguracji Google Maps na Androida/iOS są w dokumentacji, ale w skrócie: na Androidzie potrzebny jest klucz API w pliku manifest (Expo plugin maps to ułatwia), a na iOS dodanie klucza do Info.plist i biblioteki GoogleMaps w projekcie – Expo

dostarcza to poprzez config plugin. Jednak w Expo SDK 54+ większość jest automatyczna dla standardowego użycia (o ile nie używamy niestandardowych funkcji).

Użycie MapView i Marker: Podstawowy przykład wyświetlenia mapy z pojedynczym markerem:

```
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,  // im mniejsze delty, tym większe przybliżenie (zoom)
  longitudeDelta: 0.01
 };
 return (
  <View style={styles.container}>
   <MapView style={styles.map} initialRegion={region}>
    <Marker
     coordinate={userLocation}
     title="Tu jestem"
     description="Moja bieżąca lokalizacja"
    />
   </MapView>
  </View>
 );
}

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

Tutaj zakładamy, że mamy już współrzędne użytkownika (np. wcześniej pobrane przez expo-location). Ustawiamy initialRegion mapy na obszar skupiony wokół tego punktu. latitudeDelta i longitudeDelta określają zoom – mniejsza wartość to bliżej (0.01 ~ kilka ulic, 0.1 ~ całe miasto, 1 ~ cały kraj itd.). Na mapie umieszczamy komponent <Marker> we współrzędnych użytkownika. Ustawiamy mu title (to pojawi się jako nagłówek dymka po kliknięciu na marker) oraz description (mniejszy tekst w dymku). W rezultacie użytkownik zobaczy mapę z pinezką opisaną "Tu jestem". Marker domyślnie jest czerwony (Google) lub czerwony pin (Apple). Można go customizować (prop pinColor albo własny obrazek jako marker).

Interakcja i kontrola mapy: Mapę można przesuwać i skalować gestami multi-touch (domyślnie włączone). Możemy też kontrolować region z poziomu stanu aplikacji – np. używając prop region (zamiast initialRegion) wiążąc go ze state, ale wówczas musimy pamiętać o aktualizacji go (bo region staje się kontrolowany – jeśli użytkownik przewinie mapę, musimy to odnotować by nie „zawieźć" mapy do starej pozycji). Często stosuje się initialRegion dla prostego pokazu, a gdy potrzeba dynamicznie sterować, używa się referencji i metod animacji:

- mapRef.current.animateToRegion(region, duration) lub
- na iOS mapRef.current.animateCamera(camera, duration).

Jeśli chcemy pokazać **bieżącą pozycję użytkownika** dynamicznie, react-native-maps oferuje także prop showsUserLocation={true}. To automatycznie narysuje niebieską kropkę (i okrąg dokładności) tam gdzie system widzi lokalizację urządzenia. Jednak do tego trzeba mieć uprawnienie i musimy wcześniej pobrać lokalizację choć raz (na iOS inaczej nie zadziała, bo potrzebny trigger). Gdy używamy expo-location do subskrypcji, możemy w callbacku aktualizować np. marker lub region.

Marker i inne elementy: Marker może reagować na tapnięcia (prop onPress), może mieć callout (dymek) bardziej złożony – np. <Marker><Callout><Text>Jakaś info</Text></Callout></Marker> aby customizować zawartość dymka. Możemy też dodawać inne elementy jak <Polyline> (ścieżki), <Polygon>, <Circle> itd. – wymagają one odpowiedniego zasilenia danymi (listy punktów). To już zależy od potrzeb aplikacji (np. trasa biegu, obrys obszaru itp.).

Mapy na różnych platformach: Domyślnie:

- Na **Androidzie** używane są Mapy Google (wymagają Google Play Services).
- Na **iOS** używane są Apple Maps domyślnie, ale *można* przełączyć na Google Maps (ustawiając pewne flagi i dostarczając klucz API). Wiele aplikacji zostaje jednak przy Apple Maps na iOS, bo nie wymaga to dodatkowych kluczy, a Apple Maps są całkiem dobre dla większości zastosowań.
- Na **Web** (Expo web) react-native-maps nie działa natywnie, ale jest możliwość użycia np. Mapy na bazie Leaflet poprzez inny pakiet. Zwykle jednak, jeśli targetujemy też web, to trzeba dodatkowych rozwiązań (poza zakresem tego wykładu).

Wydajność i ograniczenia: Należy pamiętać, że mapy to natywny komponent – na iOS jest to MKMapView, na Androidzie MapView od Google. Reagują one na style (trzeba im dać konkretną wysokość/szerokość). Lepiej jest opakować mapę we view z konkretnymi wymiarami (jak zrobiliśmy w style, flex:1 wypełnia rodzica). Rysowanie wielu markerów (np. setek) może wpływać na wydajność – biblioteka oferuje clusterowanie markerów i optymalizacje, ale to zaawansowane tematy.

3.3 Lokalizacja w tle (wprowadzenie)

Jak wcześniej wspomniano, możliwe jest ciągłe śledzenie lokalizacji użytkownika w tle. Ponieważ to temat zaawansowany, tutaj jedynie go zarysujemy:

Expo udostępnia mechanizm zadań w tle poprzez moduł expo-task-manager. Definiujemy zadanie, np.:

```
import * as TaskManager from 'expo-task-manager';
TaskManager.defineTask("lokalizacjaTlo", ({ data, error }) => {
 if (error) {
  console.error("Błąd w background location task:", error);
  return;
 }
 if (data) {
```

```
  const { locations } = data;  // tablica nowych lokalizacji
  const loc = locations[0];
  console.log("Background location:", loc.coords);
  // tutaj można np. wysłać dane na serwer lub zapisać w pamięci
 }
});
```

Następnie, gdzieś w kodzie (gdy chcemy zacząć), wołamy:

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

To spowoduje, że aplikacja (a właściwie system) będzie co pewien czas budził naszą aplikację i wywoływał zadanie "lokalizacjaTlo" z nowymi danymi. Warunkiem jest posiadanie uprawnienia *Zawsze* (Always) na iOS oraz odpowiednich wpisów w Info.plist (Background modes -> Location), a na Androidzie uprawnienia ACCESS_BACKGROUND_LOCATION (Expo doda automatycznie jeśli isAndroidBackgroundLocationEnabled: true w app.json). Trzeba się liczyć z tym, że system może ograniczać częstotliwość – np. w iOS jak aplikacja nie jest włączona, dostaniemy lokalizacje co ~ kilka minut nawet jak damy 1 sekundę (system dba o baterię). Android Q+ wymaga, by użytkownik dodatkowo potwierdził *Allow in background* w osobnym dialogu.

Przykłady zastosowań background location: aplikacje trackingowe (fitness – śledzenie trasy biegu, jazdy), lokalizatory znajomych/dzieci (które wysyłają pozycję na serwer co jakiś czas), logowanie trasy przejazdu itp. Należy informować użytkownika o takiej funkcjonalności, bo ma ona wpływ na prywatność i baterię. Z punktu widzenia sklepu – Apple może odrzucić aplikację proszącą o Always Location bez dobrego powodu popartego opisem.

Podsumowując, expo-location pokrywa większość potrzeb geolokalizacyjnych – od jednorazowego pobrania pozycji po ciągłe śledzenie, zarówno w foreground jak i background. W połączeniu z mapami, można zbudować bogate funkcje związane z położeniem.

4. Powiadomienia (Notifications)

Powiadomienia to mechanizm informowania użytkownika o ważnych zdarzeniach, nawet gdy nie korzysta aktywnie z aplikacji. W ekosystemie Expo obsługujemy je za pomocą modułu **expo-notifications**. Dzielimy je na **powiadomienia lokalne** (generowane przez samą aplikację) i **push (zdalne)** wysyłane z serwera do urządzenia. Omówimy obie kategorie oraz różnice między platformami.

4.1 Powiadomienia lokalne (w aplikacji)

Expo-notifications umożliwia tworzenie powiadomień lokalnych – czyli np. przypomnień wyzwalanych o określonym czasie lub w reakcji na jakąś akcję użytkownika (np.

powiadomienie „zrobiłeś 10000 kroków!” w aplikacji fitness, albo natychmiastowe powiadomienie o otrzymaniu wiadomości chat gdy jesteśmy w innej części aplikacji).

Aby korzystać z expo-notifications, instalujemy paczkę expo-notifications i importujemy ją. Na iOS **musimy poprosić użytkownika o zgodę** na powiadomienia (typowy systemowy dialog „App chce wysyłać Ci powiadomienia – zezwól/nie zezwól”). W Androidzie <13 nie było takiego dialogu – powiadomienia były domyślnie dozwolone (użytkownik mógł je wyłączyć w ustawieniach). Jednak od Androida 13, również pojawia się runtime permission (Expo to obsłuży w tej samej funkcji). Dlatego dobrą praktyką jest zawsze wywołać Notifications.requestPermissionsAsync() podczas inicjalizacji powiadomień.

Prośba o zgodę na notyfikacje:

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

const { status } = await Notifications.requestPermissionsAsync();
if (status !== 'granted') {
  alert('Nie uzyskano zgody na powiadomienia.');
  // Można zakończyć procedurę lub kontynuować bez powiadomień
}
```

Na iOS możliwe jest przekazanie opcji do requestPermissionsAsync, np. które rodzaje chcemy (allowAlert, allowSound, allowBadge, provideAppNotificationSettings, allowAnnouncements). Domyślnie jeśli nie podamy, prosi o standardowe (alerty, dźwięki, odznaki). Można też poprosić o tzw. **provisional** permission (ciche powiadomienia, które nie wyświetlają alertu, tylko pojawiają się w centrum powiadomień) ustawiając np. ios: { allowProvisional: true } – wtedy użytkownik nie widzi dialogu, a appka może wysyłać „ciche” noty, które użytkownik może włączyć w pełni później. To już zaawansowana możliwość.

Wyświetlenie powiadomienia lokalnego: Gdy mamy zgodę (lub na Androidzie nie była wymagana), możemy „wystrzelić” powiadomienie. Są dwa sposoby:

- **Natychmiastowe powiadomienie** – w praktyce powiadomienie lokalne *też* korzysta z systemu, więc nawet do natychmiastowego używamy metody harmonogramującej, tylko z zerowym opóźnieniem.
- **Zaplanowane powiadomienie** – po upływie określonego czasu lub o konkretnej godzinie.

Expo-notifications upraszcza to poprzez jedną funkcję scheduleNotificationAsync. Podajemy obiekt z zawartością powiadomienia (content) oraz trygger (trigger). Trigger może być:

- określony w sekundach (opóźnienie czasowe),
- określony datą (dokładny moment),
- powtarzalny (np. co dzień o 9:00, co tydzień itp. – na iOS jest wsparcie dla kalendarzowych powtórzeń).

Przykład: chcemy natychmiast wyświetlić powiadomienie z informacją o sukcesie jakiejś operacji:

```
await Notifications.scheduleNotificationAsync({
  content: {
    title: "Operacja zakończona",
    body: "Twoje dane zostały pomyślnie zapisane.",
    sound: 'default'  // dźwięk domyślny (na Androidzie trzeba wcześniej zdefiniować kanał z dźwiękiem lub użyć domyślnego)
  },
  trigger: null  // null oznacza natychmiast (zaraz po wywołaniu)
});
```

Jeśli `trigger: null`, expo wyśle powiadomienie od razu. Alternatywnie, można użyć `trigger: { seconds: 5 }` – co spowoduje pokazanie powiadomienia za 5 sekund. Ta funkcja zwraca identyfikator powiadomienia (string), który można użyć np. do ewentualnego anulowania przed czasem (`Notifications.cancelScheduledNotificationAsync(id)`).

Treść powiadomienia (`content`): Można ustawić:

- `title` – tytuł (duży tekst, zwykle pogrubiony),
- `body` – treść,
- `data` – obiekt z danymi (np. jakieś ID, które chcemy przekazać gdy użytkownik kliknie w powiadomienie, by wiedzieć co zrobić),
- `sound` – nazwa dźwięku z zasobów lub `'default'` by użyć domyślnego. (Na Androidzie dźwięki przypisuje się do kanałów – expo-notifications tworzy domyślny kanał "Default" automatycznie ze standardowym dźwiękiem. Jeśli chcemy własne dźwięki, trzeba definiować kanały).
- `badge` – liczba, ustawia ikonkę odznaki na ikonie aplikacji (iOS).
- `subtitle`, `body`, itp. – iOS wspiera subtitle.
- `attachments` – można załączyć obraz (na iOS/Android 13+).

Na potrzeby tego wykładu skupiamy się na prostych polach: tytule i treści.

Reagowanie na powiadomienia: W kontekście powiadomień lokalnych, warto wspomnieć, że expo-notifications pozwala nasłuchiwać zdarzeń:

- otrzymania powiadomienia (gdy aplikacja jest na pierwszym planie – inaczej system sam wyświetla, ale jak jesteśmy w aplikacji, domyślnie iOS nie pokaże banera, więc można przechwycić i np. wyświetlić własny alert lub badge w UI),
- kliknięcia w powiadomienie przez użytkownika.

Można użyć `Notifications.addNotificationReceivedListener` i `Notifications.addNotificationResponseReceivedListener`. W callbacku dostajemy obiekt `Notification` lub `NotificationResponse` (zawierający m.in. `notification` i informację o akcji). W prostych zastosowaniach możemy to pominąć, ale np. jeśli chcemy, że jak użytkownik kliknie powiadomienie w tray, to aplikacja nawigowała do konkretnego ekranu, wtedy w tym listenerze implementujemy taką logikę.

4.2 Powiadomienia push (powiadomienia zdalne – przegląd)

Powiadomienia push to wiadomości wysyłane z serwera do konkretnej aplikacji na telefonie użytkownika. W odróżnieniu od lokalnych, są inicjowane poza aplikacją (np. ktoś napisze do nas wiadomość – serwer czatu wysyła push, żeby powiadomić odbiorcę). Realizacja push notyfikacji wymaga integracji z usługami **Apple Push Notification service (APNs)** dla iOS i **Firebase Cloud Messaging (FCM)** dla Androida. Expo udostępnia tu dużą pomoc poprzez **Expo Push Service**, działającą jako pośrednik.

Aby skorzystać z push:

1. Aplikacja na urządzeniu musi **zarejestrować się po token**. W klasycznym RN to oznacza wywołanie APNs i FCM osobno, ale expo-notifications upraszcza to.
2. Ten token przekazujemy na nasz **serwer**.
3. Nasz serwer (lub inny mechanizm) wysyła żądanie do serwerów Apple/Google z informacją dla urządzenia.

Expo Push Service skraca krok 3 – pozwala nam wysłać powiadomienie do **serwera Expo**, a on dalej przekaże do APNs/FCM. Dzięki temu nie musimy implementować w pełni własnego połączenia z Apple/Google, co jest wygodne zwłaszcza w trybie Expo Managed.

Uzyskanie tokena Expo: W expo-notifications wywołujemy `Notifications.getExpoPushTokenAsync(options)`. Ta funkcja wewnątrz:

- Na iOS zarejestruje aplikację w APNs i uzyska `deviceToken`.
- Na Androidzie zarejestruje w FCM i uzyska `deviceToken` (wymaga to mieć skonfigurowany projekt Firebase w naszej aplikacji – expo przy EAS Build to robi za nas jeśli podamy klucz server).
- Następnie wyśle te dane do serwisu Expo i otrzyma unikalny **Expo Push Token** (string zaczynający się od `"ExponentPushToken["...]`).

Ten expo push token identyfikuje nasze urządzenie + konkretną aplikację. **Uwaga:** Od SDK 42/43 expo wymaga podania `experienceId` lub `projectId` by wygenerować poprawny token (zwłaszcza w bare workflow), więc wywołujemy to zazwyczaj jako `Notifications.getExpoPushTokenAsync({ projectId: '<GUID naszego projektu expo>' })`. W managed workflow Expo zazwyczaj sam zna ID, ale w razie problemów trzeba to podać.

Przykład pobierania tokena (po wcześniejszym uzyskaniu zgody na powiadomienia):

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

async function registerForPushNotifications() {
  const tokenData = await Notifications.getExpoPushTokenAsync({
    projectId: '<twój-expo-project-id>'
  });
  const expoPushToken = tokenData.data;
  console.log("Expo push token:", expoPushToken);
  // Tu zwykle wysyłamy ten token do naszego backendu, np.:
  await fetch('https://moj-backend.example.com/register-token', {
```

```
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({ token: expoPushToken })
  });
}
```

Załóżmy, że nasz backend zapisze token powiązany z użytkownikiem. Teraz **wysyłanie powiadomienia** z backendu jest proste: wysyłamy żądanie HTTP POST do **endpointu Expo** https://exp.host/--/api/v2/push/send z JSON-em zawierającym token(y) i treść powiadomienia. Przykładowy payload:

```
{
  "to": "ExponentPushToken[xxxxxxxxxxxxxxxxxxxxxx]",
  "title": "Nowa wiadomość",
  "body": "Użytkownik Jan napisał do Ciebie: Hej!",
  "data": { "conversationId": "12345" }
}
```

Expo Push Service przyjmie to i zwróci tzw. **tickety** (potwierdzenia przyjęcia). Następnie asynchronicznie przekaże wiadomość do APNs lub FCM. Te z kolei dostarczą ją na urządzenie, gdzie system wyświetli powiadomienie.

Schemat działania powiadomień push w Expo: aplikacja mobilna rejestruje się i otrzymuje token Expo Push. Następnie nasz serwer (backend) wysyła za pomocą tego tokena żądanie do **Expo Push API**. Expo serwer przekazuje wiadomość do odpowiedniej platformy – Apple Push Notification service dla iOS lub Firebase Cloud Messaging dla Android. W końcowym etapie to Apple/Google doręczają powiadomienie na urządzenie użytkownika, gdzie pojawi się ono w centrum powiadomień. Dzięki temu mechanizmowi deweloper może wysyłać powiadomienia push bezpośrednio przez serwery Expo, zamiast integrować osobno z APNs i FCM.

Warto zaznaczyć, że do użycia powiadomień push w prawdziwej aplikacji produkcyjnej trzeba również:

- **Na iOS**: dostarczyć klucze/pem do Expo (w przypadku korzystania z Expo Push) lub skonfigurować odpowiednio nasz APNs certyfikat w projekcie. Expo w trybie managed pozwala w eas build łatwo wgrać push key (.p8) do naszego projektu.
- **Na Androidzie**: w przypadku Expo Push – podać klucz FCM Server Key w ustawieniach projektu (Expo Developer Dashboard) lub przez expo-cli, aby Expo mogło wysyłać do naszych użytkowników. Jeśli budujemy własny backend bez Expo – musielibyśmy w aplikacji użyć native FCM i tam generować token itd., ale w Expo managed Expo Push to wygodna droga.

Odbiór push w aplikacji: Gdy użytkownik otrzyma push:

- Jeśli aplikacja jest w tle lub ubita – system wyświetli powiadomienie w tray. Po tapnięciu w nie, aplikacja się otworzy. Możemy przechwycić to zdarzenie (NotificationResponse) i np. nawigować do odpowiedniego ekranu (np. otworzyć konkretny czat).

- Jeśli aplikacja jest na pierwszym planie (otwarta i widoczna), to:
    - Na **Androidzie** powiadomienie również pojawi się w tray (domyślnie).
    - Na **iOS** powiadomienie *nie zostanie wyświetlone* jako baner (Apple zakłada, że skoro jesteś w aplikacji, to sama aplikacja może to obsłużyć – zapobiega to dublowaniu komunikatów). W takiej sytuacji expo-notifications umożliwia nam nasłuch na zdarzenie odebrania powiadomienia i np. wyświetlenie własnego alertu lub badge w UI. Ewentualnie można wymusić iOS by pokazywał powiadomienia też na pierwszym planie ustawiając flagę `shouldShowAlert` w handlerze lub konfigurując *Notification Service Extension*, ale to rzadziej się stosuje.

Podsumowanie push: Implementacja powiadomień push wymaga połączenia wielu elementów – aplikacji (rejestracja tokenu), backendu (wysyłka poprzez Expo API lub bezpośrednio APNs/FCM) i konfiguracji kluczy. Expo ułatwia to ogromnie dzięki Push Service, jednak pamiętajmy o limitach:

- expo push ma pewne limity szybkości (np. nie należy wysyłać więcej niż 100 powiadomień/sek na jedno IP, warto batchować wysyłki).
- Powiadomienia nie są gwarantowane – mogą nie dotrzeć natychmiast jeśli urządzenie jest offline, albo w ogóle jeśli np. użytkownik odinstalował aplikację (dlatego warto usuwać nieaktywny token, Expo zwróci błąd typu DeviceNotRegistered).
- Na iOS użytkownik może w ustawieniach wyłączyć powiadomienia dla naszej appki już po wyrażeniu zgody, więc warto reagować na ewentualne `Notifications.getPermissionsAsync()` gdzie `status` może przejść w `denied` w trakcie używania aplikacji (wtedy możemy np. informować, że powiadomienia są off).
- **Debugowanie:** W symulatorze iOS nie dostaniemy pushy (Apple nie obsługuje ich w symulatorach), na Androidzie emulator może otrzymać push tylko jeśli zainstalujemy tam usługę Firebase i użyjemy push bez expo go. Expo Go od SDK 53 nie obsługuje pushy na Androidzie w ogóle, więc do testowania pushy robimy własny **Development Build** lub **standalone build**. Lokalne notyfikacje za to można testować wszędzie, także w Expo Go.

4.3 Różnice i ograniczenia powiadomień na Android vs iOS

Pisząc aplikację, warto znać pewne **różnice platformowe** w systemie powiadomień:

- **Ikony i kanały (Android):** Android wymaga, aby każda notyfikacja była przypisana do **kanału** (Notification Channel). Kanał określa m.in. dźwięk, priorytet, wibracje – i użytkownik może nim zarządzać w ustawieniach (wyłączyć dźwięk dla danego kanału np.). Expo-notifications automatycznie tworzy kanał "Default" jeśli nie podamy innego, i używa go. Możemy sami utworzyć kanały przez `Notifications.setNotificationChannelAsync("nazwakanału", options)`. Warto to zrobić np. jeśli chcemy różne typy powiadomień (np. „wiadomości” z dźwiękiem i „promocje” bez dźwięku). Ponadto, Android wymaga dodania własnej ikony powiadomień – inaczej domyślnie może pokazać białe kółko. Expo umożliwia ustawienie ikony powiadomień w app.json (pole `notification.icon`).

- **Badges (odznaki):** iOS ma natywne wsparcie tzw. badge count – liczby na ikonie aplikacji. Android tego nie miał systemowo (niektóre launchery wspierały), dopiero niektóre nakładki. Expo-notifications pozwala użyć Notifications.setBadgeCountAsync(n) na iOS, a na Androidzie po prostu nic to nie robi lub korzysta z Support Lib (na nowym Androidzie 13 są wprowadzane tzw. notification dots, ale nie liczby).
- **Prezentacja gdy app w foreground:** Jak już wspomniano, iOS domyślnie nie pokaże banera, Android pokaże (chyba że powiemy mu w opcjach notyfikacji inaczej). Jeśli chcemy jednak spójnie obsłużyć wewnątrz, expo-notifications umożliwia wykorzystanie listenera i np. ręcznie wywołać Notifications.scheduleNotificationAsync od razu z otrzymanego powiadomienia (co jest troszkę hack żeby wymusić baner na iOS – istnieje też metoda Notifications.presentNotificationAsync(content) w niektórych wersjach, która wyświetla notyfikację natychmiast nawet w foreground).
- **Limit danych:** Zarówno APNs jak i FCM mają limity rozmiaru payloadu push. APNs ~4KB, FCM ~4KB (w przypadku expo, nas to nie obchodzi bezpośrednio, bo expo serwer powie nam jak przekroczymy). Więc nie wysyłamy ogromnych JSONów w data.
- **Przyciski akcji, odpowiedzi tekstowe:** Bardziej zaawansowane funkcje jak interaktywne powiadomienia (z przyciskami akcji) wymagają natywnych konfiguracji (na iOS definicja kategorii, na Androidzie dodanie akcji do pendingIntent). Expo-notifications obecnie tego nie wspiera out-of-the-box, więc w managed workflow jesteśmy ograniczeni do prostych powiadomień z tapnięciem. Jeśli potrzebujemy np. przycisku "Odpowiedz" bez otwierania aplikacji, wymaga to wyjścia poza Expo managed.
- **Harmonogram powtarzający się:** Na iOS expo pozwala zaplanować powtarzające się powiadomienie z triggerem typu daily/weekly (np. codziennie o 9:00) – to realizuje poprzez Calendar triggers Apple. Na Androidzie do niedawna nie było prostego API do powtarzających dokładnie co tydzień (trzeba użyć AlarmManager), ale jeśli ustawimy trigger { repeats: true, hour: 9, minute: 0 } to powinno zadziałać i tu, i tu.
- **Expo Go / dev builds:** Jak wspomniano, w trybie czysto developerskim (Expo Go) **Android** od SDK 53 nie obsłuży push (decyzja Expo ze względów ograniczeń implementacji FCM w Expo Go), i trzeba używać development build (który jest jak nasza własna aplikacja). iOS Expo Go tradycyjnie w ogóle nie wspiera push (bo aplikacja Expo Go nie ma push dla każdej testowej apki). Zatem **do testów push** musimy mieć własny build na urządzeniu fizycznym.

5. Demo aplikacji: aparat i mapa (praktyczne połączenie)

Na koniec połączmy powyższe zagadnienia w mini-demo. Załóżmy scenariusz: tworzymy aplikację, w której użytkownik może zrobić zdjęcie i zapisać je (lub wysłać), a na innej zakładce podejrzeć na mapie swoją bieżącą lokalizację oznaczoną markerem "Tu jestem". To obejmie wykorzystanie aparatu, uprawnień do kamery/galerii, zapis pliku, lokalizację oraz mapę. Poniżej omówienie implementacji dwóch ekranów tej przykładowej aplikacji.

5.1 Ekran "CameraScreen" – zrobienie i zapis zdjęcia

Założenia: Ten ekran ma umożliwić użytkownikowi zrobienie zdjęcia przy użyciu aparatu. Po zrobieniu zdjęcia wyświetla jego podgląd i daje opcje: "Zapisz" (do galerii) lub "Wyślij" (symulacja uploadu). Jeśli użytkownik nie ma uprawnień do aparatu, pokaże odpowiedni

komunikat. Dodatkowo, dodamy przycisk do wyboru zdjęcia z galerii jako alternatywę (fallback, gdy np. kamera niedostępna).

```
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 zrobionego zdjęcia

  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("Brak uprawnień", "Udostępnij dostęp do zdjęć, aby wybrać obraz z galerii.");
      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("Brak uprawnień do zapisu", "Nie można zapisać zdjęcia bez dostępu do galerii.");
      return;
    }
    try {
      const asset = await MediaLibrary.createAssetAsync(photoUri);
      await MediaLibrary.createAlbumAsync("DemoApp", asset, false);
      Alert.alert("Sukces", "Zdjęcie zapisane w albumie DemoApp!");
      setPhotoUri(null); // czyścimy i wracamy do trybu kamery
```

```
  } catch (e) {
    console.error("Save error", e);
    Alert.alert("Błąd", "Nie udało się zapisać zdjęcia.");
  }
 };

 const uploadPhoto = async () => {
  // Tu normalnie byłby kod wysyłający plik na serwer, np. przez fetch / FormData.
  // My zasygnalizujemy to tylko alertem:
  Alert.alert("Wysyłanie", "Symulacja wysyłania pliku " + photoUri);
  setPhotoUri(null);
 };

 if (hasCamPermission === null) {
  return <Text>Sprawdzanie uprawnień...</Text>;
 }
 if (hasCamPermission === false) {
  return (
   <View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
    <Text>Nie udzielono dostępu do aparatu.</Text>
    <Button title="Wybierz zdjęcie z galerii" onPress={pickImage} />
   </View>
  );
 }

 return (
  <View style={{ flex: 1 }}>
   {photoUri ? (
    // Po zrobieniu/wybraniu zdjęcia - ekran podglądu
    <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="Wyślij" onPress={uploadPhoto} />
      <Button title="Anuluj" onPress={() => setPhotoUri(null)} />
     </View>
    </View>
   ) : (
    // Ekran z kamerą i przyciskami
    <Camera style={{ flex: 1 }} ref={ref => setCamera(ref)} />
   )}
   {!photoUri && ( // przyciski do zrobienia lub wybrania zdjęcia, gdy nie ma zrobionego
    <View style={{ position: 'absolute', bottom: 20, alignSelf: 'center' }}>
     <Button title="Zrób zdjęcie" onPress={takePhoto} />
     <Button title="Galeria..." onPress={pickImage} />
    </View>
   )}
  </View>
 );
}
```

Objaśnienia:

- Po montażu komponentu prosimy o uprawnienie Camera. Jeśli odmowa, zamiast widoku kamery wyświetlamy komunikat i przycisk pozwalający skorzystać z

alternatywy – wyboru zdjęcia z galerii (by jednak umożliwić użytkownikowi dodanie obrazka mimo braku aparatu). To dobry przykład fallbacku.

- Gdy uprawnienie jest, renderujemy <Camera ref={...}>. Ustawiamy ref poprzez setCamera(ref) w atrybucie ref (takie obejście, bo hooki z expo-camera użyliśmy klasycznie). Można by też użyć useRef i przypisać do cameraRef.current.
- Przycisk "Zrób zdjęcie" wywołuje takePhoto, który korzysta z referencji kamery (camera tutaj) i jej metody takePictureAsync. Ustawiamy jakość 0.7 dla mniejszego pliku. Po sukcesie zapisujemy URI w stanie photoUri.
- Gdy photoUri jest ustawione, zamiast widoku kamery pokazujemy podgląd (Image). Pod nim trzy przyciski: "Zapisz", "Wyślij", "Anuluj".
    - "Zapisz" wywołuje savePhotoToGallery – tam najpierw prosimy o (ew. brakujące) uprawnienie do MediaLibrary, następnie tworzymy asset i album (album o nazwie "DemoApp"). Jeśli się powiedzie, pokazujemy Alert z komunikatem sukcesu i czyścimy photoUri (wracamy do trybu kamery).
    - "Wyślij" wywołuje uploadPhoto – tutaj nie mamy prawdziwego serwera, więc po prostu pokazujemy alert z informacją i również czyścimy photoUri (zakładamy, że po wysłaniu już nie potrzebujemy podglądu).
    - "Anuluj" też czyści photoUri (odrzucamy zdjęcie i wracamy do aparatu).
- Przycisk "Galeria..." wywołuje pickImage – prosi o uprawnienie do biblioteki, potem otwiera expo-image-picker. Jeśli użytkownik wybierze zdjęcie (result.canceled false), pobieramy asset = result.assets[0] i ustawiamy jego URI jako photoUri. To przeniesie nas do tego samego widoku podglądu jak po zrobieniu zdjęcia aparatem. Tam użytkownik może zapisać je lub wysłać – czyli wykorzystujemy wspólną logikę. Dzięki temu fallback "wybierz z galerii" działa zarówno w przypadku braku kamery, jak i normalnie (nawet daliśmy ten przycisk obok "Zrób zdjęcie" dla wygody).
- Ostatecznie, jeśli kamera jest aktywna (brak photoUri), na dole ekranu mamy dwa przyciski do robienia zdjęcia i otwarcia galerii.

Ten ekran demonstruje wykorzystanie uprawnień (camera, media library), expo-camera i expo-image-picker, a także expo-media-library do zapisu. Zastosowaliśmy dobre praktyki:

- pytamy o uprawnienia w momencie wejścia na ekran (użytkownik świadomie wybrał funkcję aparatu),
- oferujemy alternatywę (galerię) gdy brak aparatu,
- obsługujemy odmowę dostępu do galerii przy zapisie (komunikat),
- informujemy użytkownika o rezultacie akcji (Alert po zapisaniu czy błędzie).

5.2 Ekran "MapScreen" – mapa z lokalizacją użytkownika

Założenia: Ten ekran pokaże mapę (MapView) z markerem wskazującym bieżącą pozycję użytkownika i podpisem "Tu jestem". Po wejściu na ekran aplikacja powinna uzyskać uprawnienie do lokalizacji i pobrać aktualne współrzędne. Jeśli odmowa – wyświetli komunikat o braku dostępu do GPS.

Za pomocą react-native-maps wyświetlimy mapę i użyjemy <Marker> aby zaznaczyć pozycję.

```
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>Ładowanie...</Text>;
  }
  if (hasLocationPerm === false) {
    return (
      <View style={styles.center}>
        <Text>Brak dostępu do lokalizacji GPS.</Text>
        <Text>Aby zobaczyć swoją pozycję na mapie, włącz uprawnienia lokalizacji.</Text>
      </View>
    );
  }

  // Gdy mamy lokalizację:
  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="Tu jestem" />
        </MapView>
      ) : (
        <View style={styles.center}>
          <Text>Pobieranie lokalizacji...</Text>
        </View>
      )}
    </View>
  );
}

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

});

Objaśnienia:

- W useEffect prosimy o Location.requestForegroundPermissionsAsync. Jeśli wynik to granted, od razu wywołujemy getCurrentPositionAsync aby pobrać współrzędne. Ustawiamy je w stanie location.
- Jeśli odmowa, ustawiamy hasLocationPerm na false i w render pokazujemy komunikat. (Można dodać przycisk np. "Spróbuj ponownie" lub instrukcję otwarcia ustawień – tu po prostu statyczny tekst).
- Gdy mamy dane, konfigurujemy region – daliśmy bardzo małe delta (0.005 ~ 0.5 km), więc mapa będzie dość blisko.
- Render: dopóki location jest null (np. trwa pobieranie), pokazujemy tekst "Pobieranie lokalizacji...". Gdy już jest, wyświetlamy MapView z ustawionym initialRegion.
- Marker otrzymuje coordinate={location} (czyli obiekt {latitude, longitude}) i tytuł "Tu jestem". Nie dajemy opisu (description) – nazwa wystarczy.
- Styl map: { flex: 1 } sprawia, że mapa wypełnia cały ekran.

Ta implementacja zakłada jednorazowe pobranie lokalizacji. W praktyce, jeśli użytkownik przemieści się i chce zaktualizować, trzeba by dodać np. odświeżanie (przycisk "Odśwież" wywołujący ponownie getCurrentPositionAsync i setLocation). Można też było użyć watchPositionAsync i aktualizować marker w czasie rzeczywistym, ale to generuje dużo aktualizacji i może nie być potrzebne w demo.

Działanie: Po wejściu na MapScreen, jeśli zgoda jest już nadana wcześniej (np. użytkownik wyraził ją kiedyś), to od razu zobaczy mapę z pinezką. Jeśli nie, pojawi się dialog systemowy – po wybraniu "Zezwól" komponent się zrerenderuje (status zmieni się na granted) i powinna pojawić się mapa. Jeśli wybierze "Nie zezwalaj", zobaczy nasz komunikat. Aplikacja może działać dalej, tylko bez mapy (ew. moglibyśmy wyświetlić mapę centrum kraju jako tło, ale to już decyzja projektowa – tu po prostu nic nie pokazujemy oprócz info).

Uwaga: W app.json dla iOS powinniśmy mieć klucz NSLocationWhenInUseUsageDescription (np. "Aplikacja używa Twojej lokalizacji do wyświetlenia Twojego położenia na mapie.") – inaczej App Store by nas nie przepuścił, a i system wyświetliłby domyślny mało opisowy tekst.

Podsumowanie demo

W powyższych dwóch ekranach połączyliśmy różne moduły Expo:

- **expo-camera** (Camera component) do zrobienia zdjęcia.
- **expo-image-picker** jako alternatywa do wybrania z galerii.
- **expo-media-library** do zapisania zdjęcia w albumie.
- **expo-location** do uzyskania współrzędnych GPS.
- **react-native-maps** do wyświetlenia mapy i markera.

Zaimplementowaliśmy logikę uprawnień zgodnie z najlepszymi praktykami: w kontekście (na danym ekranie), z obsługą odmowy i informowaniem użytkownika. Kod zawiera też elementy obsługi błędów (try/catch przy foto, alerty w przypadku braku zgód).

W realnej aplikacji te dwa ekrany mogłyby być częścią zakładek (np. używając React Navigation – Tab Navigator: jedna zakładka "Kamera", druga "Mapa"). Wymagałoby to drobnej konfiguracji nawigacji, ale to już osobny temat. Ważne, że poszczególne funkcjonalności działają niezależnie.

Testowanie: Funkcje aparatu i map wymagają fizycznego urządzenia lub emulatora z określonymi uprawnieniami:

- Aparat zadziała na prawdziwym urządzeniu. W symulatorze iOS kamera jest niedostępna – expo-camera zwróci błąd lub czarny ekran (można przetestować użycie pickera). Na emulatorze Android można aktywować obraz z kamery (działa jako kamera wirtualna).
- Lokalizacja: w symulatorze iOS można ustawić Lokacja (Debug -> Location -> Freeway Drive lub Custom Location), żeby symulować dane GPS. W emulatorze Android można wysłać współrzędne przez Extended Controls. Inaczej getCurrentPosition może czekać wiecznie.
- Mapy: wymagają połączenia internetowego (pobranie map). W Expo Go lub dev build musimy mieć sieć. Jeśli nie pojawią się kafelki mapy, może brak klucza API (na expo go nie powinno być problemu). W debug buildach własnych trzeba pamiętać by w AndroidManifest wstawić meta-data z API Key Google lub użyć Apple maps na iOS.

Na zakończenie, mamy aplikację, która ilustruje praktyczne użycie funkcji urządzenia i uprawnień Expo. Mam nadzieję, że to demo wraz z wcześniejszymi objaśnieniami ułatwi zrozumienie, jak w kontrolowany i przyjazny dla użytkownika sposób korzystać z aparatów, galerii, plików, lokalizacji i powiadomień w Expo/React Native. Wszystkie zaprezentowane biblioteki w najnowszej wersji Expo SDK (2025) oferują stabilne API, z którego korzystaliśmy. Przestrzegając zaleceń co do uprawnień i UX, aplikacje mogą bezpiecznie i efektywnie wykorzystywać możliwości urządzeń mobilnych.

Literatura:

1. **https://docs.expo.dev/versions/latest/sdk/camera/** (Data dostępu: 1.10.2025) – Oficjalna dokumentacja modułu **expo-camera**, opisująca zasady uzyskiwania uprawnień oraz obsługę aparatu fotograficznego i wideo.
2. **https://docs.expo.dev/versions/latest/sdk/location/** (Data dostępu: 1.10.2025) – Dokumentacja modułu **expo-location**, omawiająca pobieranie bieżącej lokalizacji GPS, śledzenie pozycji użytkownika oraz procesy geokodowania.
3. **https://docs.expo.dev/versions/latest/sdk/notifications/** (Data dostępu: 1.10.2025) – Przewodnik po module **expo-notifications**, zawierający szczegółowe instrukcje konfiguracji powiadomień lokalnych oraz zdalnych (push).
4. **https://github.com/react-native-maps/react-native-maps** (Data dostępu: 1.10.2025) – Dokumentacja biblioteki **react-native-maps**, niezbędnej do integracji interaktywnych map Google i Apple oraz obsługi markerów w aplikacjach mobilnych.