POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 1

Wprowadzenie do tworzenia aplikacji mobilnych z Expo i React Native

Mateusz Pawełkiewicz

1.10.2025

Architektura React Native – Bridge i natywne UI

React Native posiada unikalną architekturę, która łączy świat JavaScript z natywnymi komponentami interfejsu użytkownika. Kod aplikacji piszemy w JavaScript lub TypeScript (np. z użyciem React), ale interfejs renderowany jest z użyciem **natywnych widoków** (komponentów UI specyficznych dla iOS/Android). Jak to możliwe? Rdzeń stanowi tzw. **Bridge** (most) – warstwa pośrednicząca między kodem JavaScript a platformowym kodem natywnym. Bridge jest asynchronicznym mechanizmem wymiany komunikatów (w formacie JSON) pomiędzy "światem" JavaScript a "światem" natywnym. W praktyce oznacza to, że logika aplikacji działa w środowisku JavaScript (wewnątrz mobilnej aplikacji uruchamiany jest silnik JavaScript, np. **JavaScriptCore** na iOS lub dołączony JavaScriptCore/Hermes na Androidzie), a gdy trzeba coś narysować na ekranie lub wywołać funkcję natywną – wysyłany jest komunikat przez Bridge do natywnej części aplikacji.

Wielowątkowość: Przy uruchomieniu aplikacji RN działa kilka wątków (threads) odpowiedzialnych za różne zadania. Główny wątek (UI) odpowiada za renderowanie widoków i obsługę interakcji użytkownika – to na nim działają natywne komponenty i reakcje na dotyk, scroll itp. Równolegle działa **wątek JavaScript**, na którym wykonywany jest nasz kod biznesowy (logika aplikacji napisana w React/JS). Istnieje także **wątek cieni (shadow thread)**, który oblicza układy widoków – RN wykorzystuje silnik **Yoga** do przeliczania układu flexbox w osobnym wątku, po czym wyniki są przekazywane do wątku natywnego, który na ich podstawie rysuje elementy UI. Taka architektura sprawia, że interfejs może pozostać płynny – renderowanie i obsługa dotyku odbywa się niezależnie, a ewentualne cięższe obliczenia w JS nie blokują natychmiastowo UI.

Bridge – komunikacja JS z natywnym: Most jest kluczowym elementem RN – to kanał komunikacji pomiędzy kodem JS a natywnymi modułami. Wykorzystuje on seryjnie przesyłane, zserializowane wiadomości (typowo JSON) – dzięki czemu kod napisany w innym języku (JS) może "rozmawiać" z kodem Objective-C/Swift czy Java/Kotlin. Przykładowo, jeśli w natywnym module zdarzy się event (np. naciśnięcie przycisku lub zdarzenie z akcelerometru), zostanie on przekazany jako komunikat do JS, gdzie nasza aplikacja może go obsłużyć. Następnie, jeżeli na skutek tego potrzebna jest zmiana UI, strona JS wysyła przez Bridge komunikat z informacją o zmianie (np. "zmień tekst etykiety na X"), który trafia do natywnego wątku UI i tam wywołuje odpowiednie metody aktualizujące widoki. Cały ten proces jest asynchroniczny i zbuforowany – RN pakuje wiele operacji razem zanim przekaże je przez Bridge, aby zminimalizować narzut komunikacyjny. Warto zauważyć, że takie podejście (asynchroniczne komunikaty) zapewnia dużą elastyczność, ale ma pewne ograniczenia wydajnościowe. W skrajnych przypadkach (np. bardzo wiele interakcji i aktualizacji UI naraz) Bridge może stać się wąskim gardłem – każda informacja musi zostać zserializowana, przesłana i zdeserializowana, co przy dużej częstotliwości zdarzeń może powodować opóźnienia lub "klatkowanie" interfejsu. Zespół RN pracuje nad **nową architekturą (Fabric)**, która eliminuje tradycyjny Bridge na rzecz synchronizowanego interfejsu JSI.

Podsumowując, architektura React Native pozwala nam pisać aplikację w React/JS, która działa we własnym środowisku uruchomieniowym, a interfejs jest renderowany z natywnych

komponentów dzięki mechanizmowi Bridge. To podejście daje szybki rozwój w JavaScript z możliwością wykorzystania natywnych elementów UI i API urządzenia.

Instalacja środowiska i narzędzia

Zanim zaczniemy tworzyć aplikację, musimy przygotować odpowiednie środowisko deweloperskie:

- **Node.js** – wymagany jest aktualny Node.js (zalecana wersja LTS), który będzie służył do uruchamiania narzędzi Expo oraz bundlera JavaScript (Metro). Wraz z Node instalowany jest menedżer pakietów npm, który wykorzystamy.
- **Expo CLI / narzędzia Expo** – Expo udostępnia narzędzie wiersza poleceń do zarządzania projektami. Obecnie nie ma konieczności instalowania go globalnie – zaleca się korzystanie z niego poprzez **npx**, co zapewnia użycie najnowszej wersji (np. `npx expo start`). Paczka `expo` zawiera lekkie CLI dostępne właśnie przez polecenie `expo` (lub `npx expo`). *Uwaga:* Dawniej istniało globalne narzędzie `expo-cli`, jednak obecnie `npx create-expo-app` i `npx expo` zastępują tamto podejście.
- **Edytor kodu** – warto zainstalować ulubiony edytor programistyczny. Polecany jest Visual Studio Code (ze względu na dobre wsparcie TypeScript, ESLint itp.).
- **Expo Go (na urządzeniu mobilnym)** – aplikacja **Expo Go** na telefon/tablet (dostępna w App Store i Google Play). Pozwala ona uruchamiać aplikacje Expo w trybie deweloperskim na fizycznym urządzeniu, łącząc się z naszym serwerem deweloperskim. Zainstaluj Expo Go na swoim urządzeniu – posłuży do testowania aplikacji bez konieczności budowania pełnych paczek.
- **Android Studio (dla emulatora Android)** – jeśli nie masz fizycznego urządzenia lub chcesz testować na emulowanym Androidzie, zainstaluj **Android Studio** wraz z Android SDK. Podczas instalacji upewnij się, że wybrane są komponenty: Android SDK, Android SDK Platform, Android Virtual Device (AVD). Android Studio posłuży do utworzenia i konfiguracji wirtualnych urządzeń Android (AVD), na których można uruchamiać aplikację. Po instalacji Android Studio warto też dodać zmienną środowiskową `ANDROID_HOME` wskazującą lokalizację SDK oraz dopisać do PATH narzędzia platform-tools (adb) – umożliwi to Expo automatyczne wykrywanie emulatora.
- **Xcode (dla symulatora iOS)** – *(dotyczy użytkowników macOS)*. Aby emulować aplikację na iPhonie, potrzebny jest Xcode (dostępny w Mac App Store). Po instalacji Xcode można uruchamiać **iOS Simulator** i testować aplikację. Uwaga: Symulator iOS jest dostępny tylko na macOS; użytkownicy Windows/Linux mogą testować na fizycznym iPhonie przez Expo Go, ale nie uruchomią natywnego symulatora.
- **Watchman (macOS)** – opcjonalnie na macOS zaleca się instalację narzędzia Watchman (Facebook) do nasłuchiwania zmian w plikach, co przyspiesza działanie Metro bundlera. Można zainstalować go np. przez Homebrew (`brew install watchman`).

Po zainstalowaniu powyższych narzędzi, możemy zweryfikować konfigurację. Sprawdź wersję Node (`node -v`) oraz npm (`npm -v`) w terminalu. Upewnij się, że Android Studio ma zdefiniowane co najmniej jedno wirtualne urządzenie (AVD). W macOS sprawdź, czy otwiera się iOS Simulator przez Xcode.

Expo Developer Tools: Po uruchomieniu projektu (poleceniem expo start) Expo otwiera w konsoli interaktywne menu (tzw. **Terminal UI**) oraz (opcjonalnie) przeglądarkową stronę z narzędziami deweloperskimi. Z poziomu Terminal UI mamy skróty klawiszowe do typowych czynności: m.in. a – uruchom na Android Emulator, i – uruchom na iOS Simulator, w – uruchom podgląd w przeglądarce web (Expo może też budować wersję webową). Podczas pierwszego uruchomienia na Android Emulator, może pojawić się pytanie o **uruchomienie dedykowanego emulatora** – Expo wykryje zainstalowane AVD i pozwoli wybrać. Warto też zalogować się w emulatorze do sklepu Google Play i zaktualizować w nim aplikację Expo Go (na emulatorach bez usług Google Expo CLI może też zaoferować zainstalowanie Expo Go poprzez expo client:install:android).

Tryb połączenia – LAN vs Tunnel: Domyślnie Expo Dev Server nasłuchuje na lokalnym adresie IP naszego komputera (tryb LAN). Urządzenie z Expo Go musi być w tej samej sieci WiFi, aby móc połączyć się przez QR kod lub adres. Jeśli komputer i telefon nie są w jednej sieci (np. korzystasz z sieci publicznej lub masz problemy z połączeniem), Expo umożliwia użycie trybu **Tunnel** – tuneluje ruch przez serwery zewnętrzne. Aby z niego skorzystać, uruchom serwer poleceniem expo start --tunnel lub podczas pracy serwera naciśnij shift + t w konsoli/wybierz opcję **Tunnel**. W trybie tunnel Expo Go po zeskanowaniu QR łączy się poprzez wygenerowany adres URL (np. wykorzystując ngrok). *Należy pamiętać, że połączenie Tunnel jest znacznie wolniejsze od lokalnego* – odświeżanie aplikacji trwa dłużej. Dlatego zaleca się korzystać z tunelowania tylko gdy to konieczne, a najlepiej używać emulatora lub fizycznego urządzenia w tej samej sieci lokalnej dla szybszego cyklu deweloperskiego.

Tworzenie nowego projektu Expo

Mając gotowe środowisko, możemy stworzyć pierwszy projekt Expo. Expo udostępnia wygodny kreator projektów: **create-expo-app**. W najnowszych wersjach korzystamy z niego przez **npx**. Przykładowo, aby utworzyć nowy projekt o nazwie *MojaAplikacja*, w terminalu wykonujemy:

```
npx create-expo-app@latest MojaAplikacja --template blank
```

Powyższe polecenie utworzy katalog MojaAplikacja z podstawową strukturą aplikacji Expo. Użyliśmy tu parametru --template blank, który wskazuje, że chcemy skorzystać z szablonu "Blank" (pusty projekt z minimalnymi zależnościami, bez prekonfigurowanej nawigacji). Expo udostępnia kilka oficjalnych szablonów: domyślny (default) zawiera już konfigurację TypeScript i przykładową nawigację kart (Expo Router), *blank* – czysty projekt minimalny, *blank (TypeScript)* – czysty projekt z włączonym TypeScriptem, *tabs* – projekt z gotowym routerem kart (tabs) i TypeScriptem, oraz szablon *bare-minimum* – minimalny projekt z wygenerowanymi katalogami android/ios (dla ewentualnego **“eject”** do projektu natywnego).

W naszym przypadku wybraliśmy blank – czyli absolutne podstawy. **Uwaga:** Można od razu utworzyć projekt z TypeScriptem, np. poleceniem --template blank-typescript, albo później dodać konfigurację TS. Nowe wersje Expo (SDK 49+) domyślnie inicjują projekt z włączonym TS, jeśli tylko wykryją pliki .tsx. W dokumentacji Expo znajdziemy potwierdzenie: domyślny szablon ma już skonfigurowany TypeScript i rekomendowane narzędzia.

Po utworzeniu projektu, zobaczmy jak wygląda struktura folderów i plików:

- **App.js / App.tsx** – główny plik aplikacji, zawierający komponent React stanowiący punkt wejścia. W projekcie Expo standardowo jest to App.js (lub App.tsx jeśli używamy TS). Ten komponent jest automatycznie rejestrowany i uruchamiany przez Expo. W szablonie blank zobaczymy tu prosty kod wyświetlający tekst: *"Open up App.js to start working on your app!"* (lub analogiczny komunikat powitalny). Naszym celem będzie edycja tego pliku.
- **package.json** – plik definiujący nasz projekt node'owy: nazwa aplikacji, wersja, zależności npm (wygenerowane przez create-expo-app), skrypty (np. start, android, ios itp. do uruchamiania). Wygenerowany package.json zawiera już zależność expo (skorelowaną z określoną wersją SDK Expo), react i react-native (odpowiednie wersje do Expo SDK), oraz kilka skryptów ułatwiających pracę.
- **app.json / app.config.js** – plik konfiguracyjny Expo (o nim więcej w kolejnym rozdziale). Domyślnie create-expo-app tworzy statyczny plik **app.json** z podstawowymi ustawieniami (m.in. "name" i "slug" aplikacji). Plik ten jest wykorzystywany przez Expo m.in. do generowania plików natywnych (przy ewentualnym prebuild) i do konfiguracji aplikacji w Expo Go.
- **assets/** – katalog na zasoby statyczne, takie jak obrazy i czcionki. Domyślnie znajdziemy tu np. ikonę aplikacji (adaptive-icon.png) i obraz tła ekranu startowego (splash.png). Możemy tu umieszczać np. własne grafiki, które załadujemy przez require w kodzie RN.
- **node_modules/** – katalog z zainstalowanymi zależnościami JavaScript (pakiety npm wymagane przez naszą aplikację). Po utworzeniu projektu i zainstalowaniu zależności (co create-expo-app robi automatycznie, chyba że użyliśmy opcji --no-install), w node_modules znajdą się m.in. paczki expo, react, react-native i wiele innych potrzebnych do działania ekosystemu Expo.
- **tsconfig.json** – (jeśli korzystamy z TypeScript) plik konfiguracyjny TypeScript. W szablonie blank-typescript będzie on obecny i odpowiednio skonfigurowany pod RN. W razie samodzielnego przełączania na TS, utworzenie takiego pliku z właściwymi opcjami jest wymagane.
- Inne pliki konfiguracyjne: np. **babel.config.js** (konfiguracja Babel, zazwyczaj generowana automatycznie z ustawieniem preset babel-preset-expo), **.gitignore** (lista ignorowanych plików dla git), itp. – te pliki z reguły nie wymagają uwagi na starcie.

Tak przygotowany projekt możemy już uruchomić. W terminalu przejdź do folderu projektu (cd MojaAplikacja) i uruchom serwer deweloperski poleceniem:

```
npx expo start
```

To polecenie uruchomi lokalny serwer Metro bundler nasłuchujący na porcie 8081 i przygotuje naszą aplikację do działania. W terminalu zobaczysz interfejs Expo Dev Tools z kilkoma informacjami. Pojawi się m.in. **QR kod**, który możemy zeskanować za pomocą aplikacji Expo Go na telefonie, aby uruchomić naszą aplikację na fizycznym urządzeniu. Alternatywnie, skorzystaj ze skrótów: wciśnij a, aby automatycznie wystartować aplikację na uruchomionym emulatorze Android (Expo spróbuje otworzyć AVD i zainstalować/uruchomić w nim klienta Expo Go), albo i (tylko macOS) aby odpalić aplikację w iOS Simulator. Możesz

także wcisnąć w, by uruchomić aplikację jako stronę web (Expo wykorzysta wtedy webpack – warto jednak pamiętać, że nie wszystkie natywne API działają na web).

Jeśli skanujesz QR kod telefonem, upewnij się, że telefon i komputer są w tej samej sieci WiFi. W przeciwnym razie użyj wspomnianego trybu tunnel. Przy połączeniu bezpośrednim (LAN) aplikacja Expo Go powinna wykryć nasz serwer i załadować aplikację.

Pierwsze uruchomienie: Po załadowaniu aplikacji (czy to na emulatorze, czy na urządzeniu), powinniśmy zobaczyć ekran powitalny z komunikatem (np. *"Open up App.js to start working on your app!"* oraz kilka wskazówek). To jest zawartość domyślnego komponentu **App**.

Konfiguracja projektu Expo (app.json, app.config.ts, .env)

Kiedy projekt jest już utworzony, warto zrozumieć, jak konfigurujemy różne aspekty aplikacji. Expo używa specjalnego pliku konfiguracyjnego, który definiuje m.in. nazwę aplikacji, identyfikatory pakietu, ikony, ekrany powitalne i inne ustawienia. Dodatkowo często korzystamy z plików .env do przechowywania konfiguracji środowiskowych (np. różne API endpointy dla wersji deweloperskiej i produkcyjnej) oraz narzędzi linting/formatting do utrzymania jakości kodu.

Plik konfiguracyjny Expo (app.json / app.config.js)

W głównym katalogu projektu znajduje się plik **app.json** (lub alternatywnie **app.config.js/ts**), który zawiera konfigurację Expo. W najprostszej postaci app.json może wyglądać tak:

```
{
  "name": "My app",
  "slug": "my-app"
}
```

Powyżej zdefiniowano tylko nazwę aplikacji i tzw. *slug* (unikalny identyfikator projektu, używany np. w linkach Expo). W praktyce config ten może zawierać wiele więcej pól – np. orientację ekranu, ikony i obraz splash (ekranu startowego), schematy URL (deep linking), uprawnienia aplikacji, informacje dla sklepu Google Play/Apple App Store, itp.. Pełną listę dostępnych właściwości definiuje schema Expo.

Ważne jest, że plik konfiguracyjny **musi** być umieszczony w katalogu głównym projektu, obok package.json, ponieważ Expo CLI go automatycznie odczytuje. Jeśli używamy formatu JSON, wszystkie ustawienia są statyczne. Expo umożliwia jednak wykorzystanie pliku JavaScript/TypeScript jako konfiguracji – wystarczy utworzyć app.config.js lub app.config.ts. Gdy taki plik istnieje, jest on preferowany zamiast app.json i pozwala na dynamiczne generowanie konfiguracji.

Zalety dynamicznego configu: W pliku app.config.js/ts możemy pisać normalny kod (z pewnymi ograniczeniami). Możemy np. uzależnić pewne wartości od zmiennych środowiskowych, łączyć config z kilku źródeł czy używać komentarzy. Taki plik eksportuje obiekt konfiguracyjny. Przykładowo, moglibyśmy napisać:

```
// app.config.js
const myValue = 'My App';

module.exports = {
  name: myValue,
  version: process.env.MY_CUSTOM_PROJECT_VERSION || '1.0.0',
  extra: {
    fact: 'kittens are cool',
  },
};
```

Tutaj używamy zmiennej środowiskowej MY_CUSTOM_PROJECT_VERSION (jeśli jest zdefiniowana) lub domyślnie '1.0.0' jako wersję aplikacji. Dodaliśmy też pole **extra** z własnymi danymi (przykładowy fakt). Wszystkie pola z extra zostaną osadzone w aplikacji i dostępne w czasie działania poprzez moduł **expo-constants**. Można je odczytać np. tak:

```
import Constants from 'expo-constants';
console.log(Constants.expoConfig.extra.fact); // wypisze "kittens are cool"
```

Dane z extra (oraz większość konfiguracji) są dostępne w Constants.expoConfig podczas działania aplikacji. Dzięki temu możemy przekazywać np. różne klucze API dla różnych środowisk (ustawiając je w extra w zależności od zmiennej ENV przy buildzie).

Uwaga dot. wrażliwych danych: Plik konfiguracyjny (czy to JSON czy JS) jest dołączany do aplikacji i dostępny w niej jawnym tekstem. Dlatego **nie należy umieszczać tam sekretów** (np. tajnych kluczy API, haseł). Expo co prawda automatycznie filtruje pewne pola przy udostępnianiu configu w runtime (np. klucze wrażliwe jak certyfikaty Code Signing są wykluczane), niemniej jednak generalnie config służy do ustawień publicznych. Wrażliwe dane lepiej trzymać poza aplikacją lub korzystać z mechanizmów bezpiecznego przechowywania (Expo oferuje np. EncryptedStorage lub konfigurację secrets w EAS).

Zmienne środowiskowe (.env)

Wiele aplikacji potrzebuje oddzielić konfigurację dla różnych środowisk – np. inne URL do API w trakcie developmentu, a inne na produkcji. Zamiast “zaszywać” takie wartości w kodzie, dobre praktyki nakazują użyć **zmiennych środowiskowych**. W projektach Expo jest to ułatwione – Expo CLI automatycznie potrafi odczytać pliki .env.

Aby skorzystać, tworzymy plik .env w katalogu głównym projektu i definiujemy w nim pary klucz-wartość w osobnych liniach, np.:

```
EXPO_PUBLIC_API_URL=https://staging.example.com
EXPO_PUBLIC_API_KEY=abc123
```

Prefiks EXPO_PUBLIC_: Expo z premedytacją wymaga, by zmienne, które mają być dostępne w aplikacji, zaczynały się od EXPO_PUBLIC_. Tylko takie zmienne z pliku .env zostaną załadowane i wstrzyknięte do aplikacji. Dzięki temu jasne jest, że każda zmienna widoczna dla aplikacji *będzie dostępna publicznie* (po zbundlowaniu kodu). W naszym przykładzie zdefiniowaliśmy bazowy URL API i jakiś klucz. Teraz w kodzie JS/TS możemy odczytać te wartości przez obiekt process.env:

```
const apiUrl = process.env.EXPO_PUBLIC_API_URL;
const apiKey = process.env.EXPO_PUBLIC_API_KEY;
```

Gdy uruchomimy aplikację komendą `expo start`, Expo CLI w locie zastąpi wszystkie odwołania `process.env.EXPO_PUBLIC_API_URL` odpowiednim stringiem z pliku .env (np. `https://staging.example.com`) . Co ważne, dzieje się to podczas bundlowania – nie musimy wcale korzystać z dodatkowych bibliotek typu dotenv; Expo obsługuje to natywnie. Podczas pracy deweloperskiej nawet zmiany w .env mogą być podchwycone bez pełnego restartu CLI (wymagają jedynie przeładowania aplikacji, np. shake > Reload).

Nazwa prefiksu `EXPO_PUBLIC_` nieprzypadkowa – ma przypominać, że takie zmienne *nie są prywatne*. **Nigdy nie umieszczaj sekretnych kluczy w .env bez tego prefiksu z nadzieją, że nie trafią do aplikacji!**. Expo i tak domyślnie nie załaduje zmiennych bez tego prefiksu, a te z prefiksem są wkompilowane w bundle JavaScript aplikacji (czyli każdy użytkownik mógłby je odczytać z plików aplikacji). Dobrym wzorcem jest trzymanie sekretów tylko po stronie serwera, a w aplikacji mobilnej umieszczanie jedynie publicznych kluczy lub identyfikatorów. Jeśli potrzebujemy różnych konfiguracji (dev/prod), możemy użyć kilku plików .env (Expo obsługuje standard .env, .env.local, .env.production itp. zgodnie z priorytetami) albo skorzystać z funkcjonalności **EAS Secrets** gdy budujemy aplikację przez EAS (Expo Application Services). Na potrzeby wykładu wystarczy jednak wiedza, że `.env` z `EXPO_PUBLIC_` jest wygodnym sposobem na dostarczenie konfiguracji do kodu.

Narzędzia ESLint i Prettier – jakość kodu

Pisząc aplikacje w JavaScript/TypeScript, warto korzystać z narzędzi wspomagających utrzymanie jakości kodu. **ESLint** to linter statyczny – analizuje kod i potrafi wychwycić błędy lub niezalecane konstrukcje, zanim jeszcze uruchomimy aplikację. **Prettier** to z kolei automatyczny formater kodu – dba o jednolity styl formatowania (wcięcia, średniki, cudzysłowy, itd.), co sprawia, że kod jest spójny niezależnie od tego, kto go pisał. W projekcie Expo możemy łatwo zintegrować oba narzędzia.

ESLint: Nowe projekty Expo nie mają domyślnie pliku konfiguracyjnego ESLint, ale Expo udostępnia prosty sposób na dodanie go. Wykonaj komendę:

```
npx expo lint
```

Spowoduje to zainstalowanie potrzebnych zależności ESLint oraz utworzenie pliku konfiguracyjnego (od SDK 53 jest to `eslint.config.js` zgodny z nowym formatem *Flat Config*). Domyślna konfiguracja rozszerza zalecany zestaw reguł Expo (`eslint-config-expo`). Po utworzeniu configu, możesz edytować go wedle potrzeb – np. dostosować listę ignorowanych plików czy dodać własne reguły.

Aby sprawdzić kod za pomocą lintera, uruchom ponownie `npx expo lint` – powinien przeskanować pliki i wypisać ewentualne ostrzeżenia lub błędy naruszenia reguł. Warto zintegrować ESLint z edytorem – np. w VS Code zainstaluj rozszerzenie ESLint, aby problemy były podkreślane na bieżąco podczas pisania kodu.

Prettier: Prettier można używać samodzielnie (np. konfigurować formatowanie przy zapisie pliku w VSCode), ale najlepiej zintegrować go z ESLint, aby łamanie reguł formatowania było traktowane jako ostrzeżenie/błąd. Instalacja jest prosta – dodajemy paczki Prettiera i integracji:

```
npx expo install prettier eslint-config-prettier eslint-plugin-prettier --dev
```

Powyższe polecenie doinstalowuje Prettiera oraz dwa komponenty: **eslint-config-prettier** (ustawia ESLint tak, by wyłączał zasady konfliktujące z Prettier) i **eslint-plugin-prettier** (pozwala traktować problemy formatowania jako błędy ESLint). Następnie należy zmodyfikować konfigurację ESLint, aby uwzględnić Prettiera. W przypadku nowego `eslint.config.js` (Flat Config) dokumentacja sugeruje:

```
const expoConfig = require('eslint-config-expo/flat');
const eslintPluginPrettierRecommended = require('eslint-plugin-prettier/recommended');

module.exports = [expoConfig, eslintPluginPrettierRecommended, { ignores: ['dist/*'] }];
```

Jeśli korzystasz z klasycznego `.eslintrc.js`, konfiguracja wyglądałaby np. tak:

```
module.exports = {
  extends: ['expo', 'prettier'],
  plugins: ['prettier'],
  ignorePatterns: ['/dist/*'],
  rules: {
    'prettier/prettier': 'error'
  }
};
```

W powyższym ustawiliśmy, że rozszerzamy bazowy config Expo i dodajemy rozszerzenie `prettier` (które wyłącza konfliktujące reguły ESLint), aktywujemy wtyczkę prettier i dodajemy regułę `'prettier/prettier': 'error'` – oznacza to, że jeśli kod nie spełnia reguł formatowania Prettiera, zostanie to zgłoszone jako błąd ESLint. (Można użyć `'warn'` zamiast `'error'`, jeśli wolimy, by były to ostrzeżenia).

Od tej pory, po uruchomieniu `npx expo lint`, linter będzie wychwytywał także formatowanie niezgodne z Prettier. Dobrze jest również dodać plik **.prettierrc** z własną konfiguracją Prettiera (np. jeśli chcemy używać podwójnych cudzysłowów, średników itp.). Jeśli tego nie zrobimy, Prettier użyje swoich domyślnych ustawień.

W praktyce, mając ESLint + Prettier, możemy automatycznie formatować kod przy każdym zapisie pliku (VSCode: opcja “Format on save”) i być pewnym, że cały zespół trzyma się tych samych standardów kodowania. To zapobiega wielu błędom i utrzymuje czystość kodu.

Uruchamianie aplikacji: emulator, urządzenie, logi, szybkie odświeżanie

Mając projekt i skonfigurowane narzędzia, skupmy się na efektywnym uruchamianiu i testowaniu aplikacji.

Emulator Android: Jeśli skonfigurowałeś(aś) Android Studio i AVD, możesz uruchomić emulator i w Expo Dev Tools wcisnąć klawisz **a**. Expo automatycznie zlokalizuje zainstalowane Android SDK i spróbuje uruchomić aplikację na emulatorze. Przy pierwszym razie może to zająć chwilę – Expo zainstaluje w emulatorze aplikację **Expo Go** (o ile nie jest zainstalowana) oraz połączy się z naszym serwerem Metro. Jeśli wszystko pójdzie dobrze, zobaczysz okno emulatora z uruchomioną aplikacją (powinien wyświetlać się ekran powitalny Expo/Hello World). Warto zalogować się w emulatorze na konto Google i zaktualizować Expo Go przez Play Store – ułatwi to przyszłe uruchomienia i zapewni kompatybilność z najnowszym SDK.

Symulator iOS: Na macOS, uruchom **iOS Simulator** (np. z Xcode menu Open Developer Tool -> Simulator). Gdy jest aktywny, w Expo Dev Tools naciśnij **i** – projekt powinien otworzyć się w symulatorze. Expo automatycznie zbuduje *bundle* JS i wyświetli aplikację. Jeżeli symulator jest świeżo zainstalowany, Expo CLI może poprosić o zainstalowanie klienta Expo – potwierdź i poczekaj. Po chwili aplikacja powinna działać na wirtualnym iPhonie.

Fizyczne urządzenie: Alternatywnie, możesz testować bezpośrednio na telefonie/tablecie. Upewnij się, że urządzenie i komputer są w jednej sieci WiFi. Uruchom `expo start` i zeskanuj aparatem (iOS) lub przez opcję **Scan QR Code** w Expo Go (Android) wyświetlony kod QR. Aplikacja Expo Go połączy się z twoim serwerem i uruchomi projekt. Tę metodę doceniamy zwłaszcza, gdy chcemy testować funkcje wymagające fizycznego sprzętu (np. aparat, czujniki) – Expo Go daje dostęp do API urządzenia bez dodatkowej konfiguracji.

Logi aplikacji: Podczas uruchomienia w trybie deweloperskim, Expo nasłuchuje logów z aplikacji. Każdy `console.log` w kodzie RN pojawi się w konsoli (terminalu) Dev Tools. Błędy runtime również zostaną tam wyświetlone (wraz ze stack trace). W trybie development RN wyświetla też **overlay z błędem** na ekranie aplikacji, ułatwiając debugowanie. Możesz również otworzyć menu deweloperskie i wybrać **Remote JS Debugging** – spowoduje to połączenie z debugerem (domyślnie w przeglądarce Chrome), gdzie można użyć narzędzi devtools (console, debugger). Jednak od Expo SDK 48+ zalecane jest używanie nowego debuggera **Expo Developer Tools** lub Flipper. Na początek jednak konsola i komunikaty w niej powinny wystarczyć. Jeśli potrzebujesz wyświetlić logi natywne (np. logcat Androida lub console Xcode), Expo CLI udostępnia komendy `expo run:android`/`expo run:ios` po prebuild, ale w przypadku pure Expo raczej nie jest to potrzebne – większość rzeczy loguje się w Metro.

Menu deweloperskie: Zarówno na emulatorze, jak i na urządzeniu możesz otworzyć tzw. **Developer Menu** Expo. Zawiera ono przydatne opcje (Reload, enabling/disabling Fast Refresh, debug, performance monitor itp.). Aby je wywołać:

- **Android:** potrząśnij urządzeniem lub (emulator) użyj skrótu **Ctrl+M**.
- **iOS:** potrząśnij urządzeniem lub (symulator) użyj **Cmd+D**.

Menu to pozwala m.in. przełączyć tryb **Fast Refresh** (szybkie odświeżanie). Fast Refresh powinno być domyślnie włączone – oznacza to, że każda zmiana kodu w edytorze spowoduje automatyczne przeładowanie fragmentu aplikacji na urządzeniu w ciągu ~1 sekundy, z zachowaniem stanu komponentów (o ile to możliwe). Jeśli zmiany nie pojawiają się, upewnij się w menu, że opcja *Enable Fast Refresh* jest aktywna (jeśli widzisz *Disable Fast Refresh*, to

znaczy, że już jest włączona). Szybkie odświeżanie znacząco przyspiesza iterację – po zapisaniu pliku od razu widzisz efekt w aplikacji, bez potrzeby ręcznego odświeżania.

W menu deweloperskim znajdziesz także opcję **Reload** (przeładowanie całej aplikacji), **Debug Remote JS** (o którym wspomniano wyżej), **Performance Monitor** (wyświetla FPS i zużycie CPU/GPU, przydatne do optymalizacji) i inne. Zapoznaj się z nimi, bo to twoja "skrzynka narzędziowa" podczas developmentu RN.

Hot Reload a stan aplikacji: Warto dodać, że Fast Refresh w nowych wersjach RN stara się zachować stan komponentów przy odświeżaniu. Tzn. jeśli np. masz komponent z hookiem useState i zmienisz tylko markup, stan nie zresetuje się. Jednak nie zawsze jest to możliwe (np. zmiana struktury komponentu może spowodować pełny reload). Gdy potrzebujesz pełnego restartu aplikacji, możesz użyć opcji Reload z menu lub po prostu zakończyć i ponownie uruchomić `expo start`.

Demo: Ekran "Hello, RN" – komponent, SafeAreaView, pierwsze style

*Przykład ekranu "Hello World". Zastosowanie komponentu **SafeAreaView** zapewnia, że niebieski pasek z tekstem jest widoczny poniżej notcha (zachowane są marginesy bezpiecznego obszaru).*

Skoro nasza aplikacja działa, dokonajmy w niej pierwszych zmian – stworzymy prosty ekran **"Hello, RN"**. Pokaże to, jak definiować komponenty w React Native, użyć SafeAreaView oraz jak stosować style.

Otwórz plik `App.js` (lub `App.tsx`). Domyślnie zobaczysz tam coś w stylu:

```
<View style={styles.container}>
  <Text>Open up App.js to start working on your app!</Text>
  <StatusBar style="auto" />
</View>
```

Zamiast tego, zmodyfikujmy kod aby wyświetlał "Hello, RN". Przy okazji skorzystamy z komponentu **SafeAreaView** na górze hierarchii. SafeAreaView to komponent dostarczany przez RN (na iOS) lub bibliotekę *react-native-safe-area-context* (Expo ją dołącza), który działa analogicznie do zwykłego View, ale automatycznie dodaje odpowiednie *paddingi* tak, aby zawartość znalazła się w tzw. *safe area* ekranu. Innymi słowy, chroni przed nachodzeniem elementów na notch, status bar czy zaokrąglone krawędzie. Zaleca się opakowywać główną zawartość ekranu w SafeAreaView, dzięki czemu na nowych iPhone'ach nasz tekst nie schowa się pod notch'em, a na Androidzie nie zostanie przykryty przez pasek statusu.

W kodzie zmienimy również komponent na **funkcyjny** z użyciem TypeScript i zdefiniujemy go jako typ `React.FC`. Oto przykładowa implementacja:

```
import React from 'react';
```

```
import { SafeAreaView, Text, StyleSheet } from 'react-native';

export const App: React.FC = () => {
  return (
    <SafeAreaView style={styles.container}>
      <Text style={styles.text}>Hello, RN!</Text>
    </SafeAreaView>
  );
};

const styles = StyleSheet.create({
  container: {
    flex: 1,
    backgroundColor: '#fff',
    alignItems: 'center',
    justifyContent: 'center',
  },
  text: {
    fontSize: 24,
  },
});
```

Kilka wyjaśnień do powyższego kodu:

- Użyliśmy SafeAreaView zamiast zwykłego View jako kontener najwyższego poziomu. Dzięki temu na urządzeniach z notchem zawartość będzie automatycznie odsunięta od krawędzi ekranu. (W Expo komponent SafeAreaView pochodzi z biblioteki *react-native-safe-area-context*, która działa zarówno na iOS, jak i Androidzie – nie musimy jej dodatkowo instalować, ponieważ Expo ją bundluje).
- Wewnątrz umieściliśmy komponent Text z napisem "Hello, RN!". Dla czytelności dodaliśmy styl styles.text, który ustawia większy rozmiar czcionki (24).
- Nasz komponent App został zadeklarowany jako React.FC (React Functional Component). Nie jest to obowiązkowe, ale w TypeScript pomocne – określa, że jest to komponent React z poprawnie zdefiniowanymi typami props/state (tu akurat bez propsów). Użyliśmy eksportu nazwowego (export const App...), choć równie dobrze można zrobić export default function App() { ... }. W przypadku Expo, jeśli korzystamy z eksportu nazwowego, to i tak w pliku package.json powinniśmy mieć wskazanie, który plik jest główny (domyślnie Expo oczekuje App.(j|t)sx). W naszym przykładzie pozostańmy przy nazwanym eksporcie dla demonstracji.
- Zastosowaliśmy **StyleSheet.create** do zdefiniowania stylów. StyleSheet w RN to wbudowane API, które pomaga tworzyć styl w sposób zbliżony do CSS, ale zapisany w obiektach JS. Styl container ustawia flex: 1 (czyli kontener wypełnia całą wysokość – to ważne, by SafeAreaView mogło zadziałać poprawnie), białe tło, oraz centrowanie elementów (alignItems, justifyContent na 'center' – podobnie jak w CSS Flexbox). Styl text ustawia tylko rozmiar czcionki. W RN style definiujemy za pomocą obiektów JavaScript, gdzie nazwy atrybutów są camelCase zamiast kebab-case (np. backgroundColor zamiast background-color). Większość właściwości pokrywa się z CSS (np. color, fontSize, margin, padding itd.), co ułatwia naukę stylowania w RN. Możemy tworzyć style poprzez StyleSheet (co daje ewentualne optymalizacje), albo inline jako zwykłe obiekty (np. <Text style={{ fontSize: 24 }}>).

- Upewnij się, że zaimportowałeś(aś) SafeAreaView z właściwego modułu. W czystym RN jest on w react-native (działa tylko na iOS 11+), natomiast w Expo zaleca się import z react-native-safe-area-context dla pełnej cross-platformowości. W powyższym kodzie dla uproszczenia importujemy z react-native – w nowszych wersjach Expo i tak podmieni to na implementację z safe-area-context.

Zapisz zmiany i obserwuj **Fast Refresh** w akcji – aplikacja powinna automatycznie się zaktualizować. Na ekranie zobaczysz białe tło, na środku napis "Hello, RN!". Jeśli testujesz na iPhonie z notchem, zauważ, że tekst nie przylega do górnej krawędzi – to efekt SafeAreaView (treść jest w bezpiecznym obszarze).

Literatura:

1. https://reactnative.dev/docs/getting-started (Data dostępu: 1.10.2025) - Oficjalna dokumentacja React Native.
2. https://docs.expo.dev/ (Data dostępu: 1.10.2025) - Oficjalna dokumentacja Expo.
3. https://docs.expo.dev/guides/new-architecture/ (Data dostępu: 1.10.2025) - Kluczowa dokumentacja Expo wyjaśniająca Nową Architekturę (Fabric).
4. https://docs.expo.dev/router/introduction/ (Data dostępu: 1.10.2025) - Dokumentacja Expo Router, nowoczesnego standardu nawigacji (wspomnianego w wykładzie).
5. https://docs.expo.dev/guides/environment-variables/ (Data dostępu: 1.10.2025) - Szczegółowy opis zarządzania zmiennymi środowiskowymi (.env) w Expo.
6. https://reactnavigation.org/ (Data dostępu: 1.10.2025) - Dokumentacja React Navigation (alternatywnej, popularnej biblioteki do nawigacji).
7. https://nodejs.org/ (Data dostępu: 1.10.2025) - Strona oficjalna Node.js.
8. https://code.visualstudio.com/ (Data dostępu: 1.10.2025) - Strona oficjalna Visual Studio Code.