POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 10

Bezpieczeństwo, dostępność i i18n

Mateusz Pawełkiewicz

1.10.2025

Bezpieczeństwo aplikacji mobilnych

OWASP Mobile Top 10. Organizacja OWASP co pewien czas publikuje listę **Top 10** najważniejszych zagrożeń bezpieczeństwa aplikacji mobilnych. Aktualna lista obejmuje m.in. niewłaściwe użycie mechanizmów platformy, niebezpieczne przechowywanie danych, słabe uwierzytelnianie, błędy walidacji danych, niebezpieczną komunikację sieciową, braki w kryptografii i inne aspekty (M1–M10). Lista ta służy jako przewodnik dla deweloperów, wskazując obszary, na które trzeba zwrócić szczególną uwagę podczas tworzenia aplikacji mobilnych.

Najważniejsze praktyki bezpieczeństwa: W odniesieniu do OWASP Top 10 warto stosować sprawdzone praktyki zabezpieczające aplikację:

- **Trzymanie sekretów z dala od kodu:** Należy **unikać hardcodowania sekretów** (kluczy API, haseł itp.) w kodzie aplikacji lub w niezabezpieczonych plikach konfiguracyjnych. Takie dane powinny być przechowywane bezpiecznie (np. pobierane z serwera po uwierzytelnieniu lub zapisane w szyfrowanym magazynie urządzenia). W przeciwnym razie atakujący może łatwo wydobyć te informacje, np. przez inżynierię wsteczną aplikacji. Zalicza się to do typowych błędów – często spotykane jest **przechowywanie kluczy i sekretów w kodzie** aplikacji. Zamiast tego korzystamy z bezpiecznych magazynów platformowych.
- **Bezpieczne przechowywanie danych:** Urządzenia mobilne często są narażone na fizyczną utratę lub złośliwe oprogramowanie, dlatego **wrażliwe dane lokalne należy odpowiednio zabezpieczyć**. Używaj **platformowych magazynów bezpiecznych** – na iOS jest to **Keychain**, a na Androidzie **Keystore** – do przechowywania haseł, tokenów sesji itp.. Dane tam zapisane są chronione na poziomie sprzętowym i systemowym (np. szyfrowane, powiązane z ekranem blokady). Unikaj przechowywania poufnych informacji w zwykłych preferencjach, plikach czy bazach SQLite bez szyfrowania. **Minimalizuj lokalne przechowywanie** – zapisuj tylko to, co konieczne, i **czyść dane przy wylogowaniu** użytkownika. Czyszczenie danych (cache, danych sesyjnych) po wylogowaniu zapewnia, że kolejna osoba używająca urządzenia nie uzyska dostępu do cudzych informacji.
- **Logi bez wrażliwych informacji:** Podczas debugowania aplikacji łatwo zostawić w logach informacje o użytkowniku, tokeny czy inne wrażliwe dane. **Unikaj logowania poufnych danych** w produkcyjnej wersji aplikacji. Nawet jeśli Android od pewnej wersji ogranicza dostęp do logcat dla aplikacji trzecich, na wielu urządzeniach uprzywilejowane aplikacje systemowe nadal mogą czytać logi. Zasada jest prosta: **loguj tylko to, co niezbędne**, najlepiej statyczne komunikaty, a przed publikacją usuń lub wyłącz szczegółowe logowanie debugowe. Jeśli już musisz coś zalogować dla celów diagnostycznych, **maskuj wrażliwe fragmenty** (np. pokazuj tylko ostatnie 4 cyfry numeru karty) aby nie wyciekły pełne dane. Warto stosować mechanizmy usuwania logów debug w kompilacji produkcyjnej (np. przez narzędzia typu ProGuard/R8 na Androidzie, które mogą usuwać wywołania logowania).

Bezpieczna komunikacja sieciowa (MITM i pinning). Komunikacja aplikacji z serwerem powinna być zawsze odpowiednio zabezpieczona, aby uniemożliwić **podsłuch (atak typu man-in-the-middle, MITM)**. Należy **używać protokołu HTTPS (TLS)** do wszelkiej komunikacji

z backendem i **weryfikować certyfikaty SSL**. Aplikacja nie może akceptować dowolnych certyfikatów (np. samopodpisanych) bez weryfikacji – takie błędy pozwalają atakującemu podszyć się pod serwer. Według OWASP jednym z głównych zagrożeń jest **niezabezpieczona komunikacja**, np. brak TLS lub akceptowanie każdego certyfikatu, co umożliwia MITM. Standardem jest więc korzystanie z HTTPS z poprawną walidacją certyfikatu *CA*.

Ponadto zalecaną praktyką jest **certyfikaty pinning**, czyli przypięcie certyfikatu lub klucza publicznego serwera w aplikacji. Polega to na tym, że aplikacja ma zapisany wzorzec certyfikatu lub klucz publiczny, któremu ufa – i podczas nawiązywania połączenia TLS sprawdza, czy certyfikat serwera odpowiada temu wzorcowi. Jeśli nie, zrywa połączenie nawet jeśli systemowa lista zaufanych urzędów certyfikacji by go zaakceptowała. Pinning zabezpiecza przed sytuacją, w której ktoś zdoła uzyskać fałszywy certyfikat od urzędu certyfikacji lub przechwycić ruch na urządzeniu (np. za pomocą własnego zaufanego certyfikatu). **Implementacja pinningu** wymaga aktualizacji aplikacji, gdy certyfikat serwera wygaśnie lub się zmieni – należy to uwzględnić (można np. pinować klucz publiczny, który zmienia się rzadziej niż cały certyfikat). OWASP zaleca wykorzystywać pinning tam, gdzie to możliwe, jako dodatkową warstwę obrony. Podsumowując: **zawsze używaj szyfrowania transportu (TLS)** i **obsługuj błędy certyfikatów poprawnie** (nie ignoruj ich), a w newralgicznych aplikacjach rozważ pinning.

Ograniczanie liczby żądań (Rate limiting). Aplikacje mobilne często korzystają z API, które są narażone na nadużycia – np. próby siłowego odgadnięcia hasła, spamowania endpointów czy generowania nadmiernego ruchu. **Rate limiting** to mechanizm ograniczania częstotliwości wywołań danego działania w określonym czasie. Choć zwykle implementuje się go po stronie serwera, deweloper mobilny powinien go uwzględnić w projektowaniu systemu. Przykładowo: **ograniczenie liczby prób logowania** (np. kilka nieudanych prób skutkuje czasową blokadą kolejnych) zapobiega atakom słownikowym. Limitowanie może dotyczyć też np. liczby zapytań do API na minutę z jednego tokena czy adresu IP. Według zaleceń bezpieczeństwa, **stosowanie limitów** i odpowiednich kodów błędów/odpowiedzi (np. HTTP 429 Too Many Requests) jest kluczowe dla ochrony przed automatycznymi atakami i nadmiernym obciążeniem usług. Z perspektywy aplikacji mobilnej warto również lokalnie **throttlować** pewne akcje użytkownika (np. nie pozwolić wysłać 100 żądań na sekundę z UI, nawet jeśli API to utnie). Mechanizmy te nie wpływają na zwykłych użytkowników, a znacząco podnoszą bezpieczeństwo i odporność systemu.

Dostępność aplikacji (Accessibility)

Dostępność, określana często skrótem **a11y**, to cecha aplikacji, która pozwala na wygodne korzystanie z niej również osobom z niepełnosprawnościami lub ograniczeniami (np. wzroku, słuchu, motoryki). Tworząc aplikacje mobilne musimy zapewnić, że **każdy użytkownik** będzie w stanie z niej skorzystać – w praktyce oznacza to wsparcie dla czytników ekranu (VoiceOver na iOS, TalkBack na Androidzie), możliwość obsługi aplikacji bez patrzenia na ekran lub bez użycia dotyku, dobre kontrasty kolorów, skalowalny tekst itp. W wielu krajach zadbanie o dostępność nie jest tylko dobrą praktyką, ale wręcz wymogiem prawnym (np. aplikacje publiczne muszą spełniać standardy WCAG 2.1 na poziomie AA). Poniżej omówimy główne aspekty dostępności w kontekście aplikacji (zwłaszcza React Native, ale zasady są podobne dla natywnych aplikacji).

Role i etykiety elementów interfejsu

Etykiety dostępności (accessibilityLabel). Każdy element interfejsu, który przekazuje informację lub jest interaktywny, powinien mieć **opis dla czytnika ekranu**. W React Native służy do tego właściwość accessibilityLabel, w komponentach iOS/Android analogicznie ustawia się etykiety dostępności. Czytnik ekranu odczytuje tę etykietę na głos, gdy fokus znajdzie się na danym elemencie. Etykieta powinna zwięźle opisywać element, np. dla ikony przycisku bez tekstu należy ustawić accessibilityLabel="Otwórz ustawienia użytkownika" zamiast samej nazwy ikony. W RN jeżeli elementem jest tekst, atrybut ten domyślnie zaciąga treść tekstu – ale często trzeba doprecyzować. Przykład w RN:

```
<Pressable accessibilityLabel="Otwórz ustawienia użytkownika" onPress={openSettings}>
  <Text>Ustawienia</Text>
</Pressable>
```

Tutaj przycisk wizualnie pokazuje ikonę/tekst "Ustawienia", lecz dla czytnika ekranu przekazujemy pełniejszy opis **co robi** (np. "Otwórz ustawienia użytkownika"). **Dobre etykiety** mówią **co to jest i do czego służy** element, nie zawierają szczegółów wizualnych (np. kolor przycisku) – bo to nieistotne dla użytkownika niewidomego.

Role (accessibilityRole). Każdy element interfejsu ma pewną rolę (rodzaj) – np. przycisk, nagłówek, obraz, pole tekstowe itp. Określanie roli pomaga technologiom asystującym poinformować użytkownika, z czym ma do czynienia. W React Native służy do tego prop accessibilityRole. Przykładowe wartości to m.in. "button" (przycisk), "header" (nagłówek sekcji), "image", "text", "link", "search", "checkbox" itd. Powinniśmy ustawiać role zgodnie z przeznaczeniem elementu – np. jeśli zbudowaliśmy własny komponent dotykowy pełniący funkcję przycisku, nadajmy mu accessibilityRole="button". Czytnik wtedy ogłosi go jako “przycisk”. W RN elementy interaktywne (TouchableOpacity, Pressable itp.) domyślnie są dostępne i często traktowane jak przyciski, ale dla pewności można rolę podać. Przykład:

```
<View accessible={true} accessibilityRole="button" onTouchEnd={...}>
  <Text>Kliknij mnie</Text>
</View>
```

Tutaj zwykły widok View został oznaczony jako dostępny i jego rola to *button*, więc VoiceOver/TalkBack potraktuje całość jako przycisk. Z drugiej strony, elementy czysto dekoracyjne, które **nie powinny** być czytane, powinny mieć wyłączoną dostępność (accessible={false}), aby nie “zaśmiecały” użytkownikowi informacji.

Porządek fokusu i nawigacja

Kolejność fokusu. Użytkownik niewidomy lub mający problemy z interakcją często nawiguje po aplikacji za pomocą przycisków sprzętowych lub gestów, które przemieszczają **fokus** kolejno między elementami. Bardzo ważne jest, aby **logiczna kolejność fokusu** odpowiadała intuicyjnej kolejności treści. Zazwyczaj jest to kolejność zgodna z układem UI (np. od góry do dołu, lewa do prawa). W React Native (i ogólnie w natywnych aplikacjach) kolejność czytania odpowiada kolejności dodania elementów w hierarchii widoków. Dlatego należy tworzyć layout w takiej strukturze, by **sekwencja elementów w kodzie** była logiczna do czytania. Jeśli

np. coś jest na ekranie po lewej a potem po prawej, ale w kodzie zostanie dodane odwrotnie, czytnik może czytać w złej kolejności.

W trudniejszych przypadkach platformy oferują mechanizmy ręcznego ustalenia kolejności. W RN eksperymentalnie istnieje prop accessibilityOrder pozwalający definiować konkretną kolejność elementów, gdy domyślna jest niewłaściwa. W Androidzie natywnie można użyć atrybutu android:accessibilityTraversalAfter, a w iOS ułożyć elementy w odpowiedniej kolejności lub użyć kontenerów dostępności. **Kontenery dostępności** (np. grupowanie pod accessible={true} na rodzicu w RN) mogą sprawić, że cała grupa jest czytana jako jedna całość, co też wpływa na kolejność. Podsumowując: upewnij się, że **nawigacja po elementach jest logiczna**, i używaj mechanizmów kontenerów lub wytycznych platformy, by to osiągnąć. Testuj swoją aplikację za pomocą czytnika ekranu, **przechodząc krokowo przez elementy** – czy fokus przeskakuje w sensowny sposób. Jeśli nie, popraw kolejność w kodzie lub zastosuj odpowiednie atrybuty.

Kontrast kolorów

Kontrast tekstu do tła musi być wystarczająco wysoki, aby tekst był czytelny dla osób słabowidzących lub przy gorszym oświetleniu. Wytyczne WCAG 2.1 zalecają kontrast co najmniej **4.5:1** dla tekstu małej wielkości (oraz 3:1 dla dużych nagłówków). Oznacza to, że np. szary tekst na jasnym tle może być nieczytelny. Należy dobierać kolory tekstu i tła tak, by różnica jasności i koloru spełniała te kryteria. **Sprawdzaj kontrast** za pomocą dostępnych narzędzi – np. strona WebAIM udostępnia Color Contrast Checker, gdzie podajemy kolory i otrzymujemy ich współczynnik kontrastu. W fazie designu warto od razu przewidzieć odpowiednią paletę barw.

Ponadto aplikacje mobilne powinny respektować tryby wysokiego kontrastu, jeśli system takowe oferuje. Np. w Androidzie jest opcja wysokiego kontrastu tekstu – nasza aplikacja powinna nie nadpisywać systemowych ustawień (nie blokować zmiany kolorów). W iOS można użyć dynamicznych kolorów z palety systemowej (które automatycznie dostosowują się do trybu zwiększonego kontrastu lub trybu ciemnego/jasnego). Generalnie, **unikaj małych, jasnoszarych literek na białym tle** czy kombinacji kolorów jak ciemnoczerwony tekst na czarnym tle, itp. Zapewnij też możliwość odróżnienia elementów interfejsu bez polegania wyłącznie na kolorze – np. podkreśl linki (nie tylko kolor) dla osób z daltonizmem. **Dostateczny kontrast to podstawa dostępności wizualnej.**

Dynamiczne skalowanie tekstu (Dynamic Type)

Wielu użytkowników potrzebuje **większego tekstu** na ekranie – np. osoby słabowidzące lub nawet w określonych warunkach (mały ekran, silne światło słoneczne). Systemy mobilne umożliwiają ustawienie preferowanego rozmiaru czcionki. **Dynamic Type** to funkcja iOS (oraz odpowiedniki w Androidzie), która skaluje czcionki w aplikacjach zgodnie z preferencjami użytkownika. Jako twórcy aplikacji musimy **wspierać skalowanie tekstu** – to znaczy używać stylów tekstu, które są skalowalne, i testować nasz interfejs na powiększonych czcionkach. W React Native standardowy komponent <Text> obsługuje dynamiczne skalowanie jeśli nie zablokujemy tego. RN używa mechanizmu Apple **Dynamic Type** automatycznie dla systemowych fontów, ale developer musi upewnić się, że interfejs nadal wygląda poprawnie

przy większym tekście. Absolutnie nie powinno się ustawiać wszędzie sztywno font-size na małą wartość ignorując ustawienia systemowe.

Dobra praktyka: weź urządzenie, wejdź w ustawienia **Accessibility > Display & Text Size** (iOS) lub **Ułatwienia dostępu > Rozmiar tekstu** (Android) i ustaw największą czcionkę, a następnie uruchom swoją aplikację. Czy wszystkie napisy są nadal czytelne i mieszczą się na ekranie? Czy nie obcinają się lub nie nachodzą na inne elementy? Jeśli tak – musisz poprawić layout (np. użyć komponentów przewijalnych dla długich tekstów, dać elementom więcej elastyczności). Zapewnienie dynamicznego tekstu często oznacza też stosowanie jednostek względnych (np. EMS) zamiast pikseli przy definiowaniu własnych fontów. W RN można też odpytać moduł AccessibilityInfo o preferowany **rozmiar tekstu** i reagować na zmiany, jeśli potrzeba. Ogólnie, **szanuj ustawienia użytkownika** – ktoś celowo ustawił duży tekst, więc Twoja aplikacja powinna to respektować zamiast np. wyświetlać tekst w formie obrazu, którego nie da się skalować.

Podsumowując część o dostępności: **projektuj interfejs z myślą o wszystkich**. Oznacz elementy właściwymi etykietami i rolami, dbaj o logiczną nawigację po nich, używaj czytelnych kontrastów i umożliwiaj skalowanie treści. Testuj aplikację używając czytnika ekranu (warto samemu włączyć VoiceOver/TalkBack i spróbować użyć aplikacji) oraz innych ułatwień (zmiana czcionki, wysoki kontrast). Dzięki temu aplikacja będzie **bardziej użyteczna i przyjazna** nie tylko dla osób z niepełnosprawnościami, ale dla wszystkich użytkowników (dostępność często poprawia ogólną użyteczność produktu).

Internationalizacja (i18n) w aplikacjach

i18n (internationalization) to przystosowanie aplikacji do łatwego tłumaczenia interfejsu na różne języki i dostosowania do różnych regionów. Z kolei **lokalizacja (l10n)** to konkretne przetłumaczenie i dostosowanie do danego języka/kultury. W kontekście React Native (i ogólnie aplikacji JavaScriptowych) popularnym rozwiązaniem jest biblioteka **i18next** wraz z integracją **react-i18next**. Pozwala ona zarządzać słownikami wielu języków i przełączać je w locie. **Dlaczego to ważne?** Jeśli aplikacja ma użytkowników w różnych krajach, oczekują oni interfejsu w swoim języku, jak również lokalnych formatów dat, liczb, jednostek miar czy walut. Brak lokalizacji oznacza gorsze doświadczenie użytkownika, a nawet może uniemożliwić korzystanie z aplikacji (np. jeśli ktoś nie zna angielskiego). Dlatego już na etapie tworzenia trzeba zadbać o i18n.

i18next w React Native. Wykorzystanie i18next w RN jest podobne jak w React na web. Aplikacja ładuje biblioteki i18next i react-i18next, inicjalizuje je z listą obsługiwanych języków i plikami tłumaczeń (np. pliki JSON z kluczami i przetłumaczonymi tekstami), a następnie zamiast pisać teksty na sztywno w komponentach, używa funkcji tłumaczącej. Typowy przepływ:

1. **Instalacja**: npm install i18next react-i18next (ew. dodatkowo detector języka użytkownika, backend do ładowania plików, jeśli potrzebne).
2. **Pliki tłumaczeń**: Utworzenie np. folderu locales, a w nim plików JSON dla każdego języka, np. en.json, pl.json. W środku obiekty klucz-wartość, np. { "login": "Log In", "welcome": "Welcome {{name}}" } dla angielskiego i odpowiedniki po polsku { "login":

"Zaloguj", "welcome": "Witaj {{name}}" }. Klucze mogą mieć też zagnieżdżoną strukturę (kropki) dla organizacji.

3. **Inicjalizacja i konfiguracja**: Wywołujemy i18next.init() (lub korzystamy z useTranslation hook z react-i18next) gdzie podajemy dostępne języki, domyślny język, sposób wykrywania języka (np. na podstawie ustawień urządzenia) i ładujemy przygotowane słowniki. Np.:

```
import i18n from 'i18next';
import { initReactI18next } from 'react-i18next';
import en from './locales/en.json';
import pl from './locales/pl.json';

i18n.use(initReactI18next).init({
  resources: { en: { translation: en }, pl: { translation: pl } },
  lng: 'pl',       // domyślny język (np. ustawiony na polski)
  fallbackLng: 'en',// język zapasowy
  interpolation: { escapeValue: false }
});
```

   Po takiej konfiguracji biblioteka udostępnia kontekst tłumaczeń w aplikacji.

4. **Użycie w komponentach**: Dzięki react-i18next możemy wywołać hook const { t } = useTranslation(); i następnie używać t('login') czy t('welcome', { name: 'Jan' }) aby uzyskać przetłumaczony tekst. Zamiast <Text>Welcome</Text> piszemy <Text>{t('welcome', { name: userName })}</Text>. Cała treść interfejsu powinna pochodzić ze słowników – dzięki temu łatwo przełączyć język w całej aplikacji.

Liczba mnoga (pluralizacja). Różne języki mają różne formy liczby mnogiej, co trzeba uwzględnić w tłumaczeniach. i18next wspiera pluralizację automatycznie – jeśli przekażemy do t parametr count, to biblioteka wybierze odpowiednią formę. W plikach tłumaczeń definiujemy wtedy teksty z sufiksami, np. klucze: "message_one": "Masz 1 nową wiadomość", "message_few": "Masz {{count}} nowe wiadomości", "message_many": "Masz {{count}} nowych wiadomości". Dla języka polskiego występują różne formy w zależności od liczby (1, 2-4, 5-21 itd.), więc możemy zdefiniować _one, _few, _many itp. Dla angielskiego są zwykle tylko dwie formy: one i other. Następnie w kodzie wywołujemy t('message', { count: ileWiadomosci }) bez dodawania sufiksu – i18next sam dobierze odpowiedni wariant na podstawie wartości count i reguł danego języka. Ważne, by **używać parametru count** o takiej nazwie, bo biblioteka go rozpoznaje. Dzięki temu unikamy ręcznego wybierania form gramatycznych – biblioteka korzysta z wbudowanych reguł (np. dla polskiego zdefiniowane są wyjątki dla liczebników zakończonych na 2,3,4 ale nie na 12,13,14 itd.). To znacząco ułatwia przygotowanie komunikatów typu "X elementów". Warto również przewidzieć tłumaczenie dla liczby **0**, bo np. po polsku często jest osobna konstrukcja ("0 wiadomości" – tutaj akurat forma jest jak dla mnogiej, ale w innych zwrotach może być unikalna). I18next domyślnie użyje formy other dla 0, chyba że zdefiniujemy specjalnie klucz z sufiksem zero dla danego tekstu.

Formaty dat i liczb. Internacjonalizacja to nie tylko słowa, ale też **formatowanie dat, godzin, liczb i walut** zgodnie z lokalnymi zwyczajami. Np. data 03/04/2025 znaczy co innego w USA (kwiecień 3) niż w Europie (3 kwietnia). Podobnie liczba **1,234.56** jest zapisywana jako **1 234,56** w Polsce (przecinek dziesiętny zamiast kropki, spacja jako separator tysięcy). **Każdy**

region ma własne formaty – nawet kraje z tym samym językiem różnią się notacją (np. angielski w USA vs angielski w Kanadzie mają inny format daty i jednostki miar). Dlatego aplikacja powinna używać odpowiednich ustawień regionalnych (*locale*) do formatowania. W JavaScript mamy wbudowane API **Intl** (Internationalization API) – obiekty takie jak Intl.DateTimeFormat i Intl.NumberFormat pozwalają formatować daty oraz liczby zgodnie z locale. i18next od wersji 21+ ma wbudowane funkcje formatowania korzystające z Intl. Można więc np. w tekście tłumaczenia umieścić placeholder z formatowaniem: "today": "Dziś jest {{date, datetime}}" i przekazać t('today', { date: new Date() }), a biblioteka sformatuje datę bieżącą według ustawień. Oczywiście trzeba zdefiniować, jakie formaty nas interesują (np. krótsza czy dłuższa data). Alternatywnie, można samodzielnie użyć Intl.DateTimeFormat(locale, options).format(new Date()) – np. dla polskiego locale="pl-PL".

Ważne jest, by **ustawić poprawny locale z kodem kraju**, nie tylko język. Np. en-US vs en-GB – obie to angielski, ale US używa miesiąc/dzień/rok, a GB dzień/miesiąc/rok i innej waluty. Z kolei dla języków wieloregionalnych jak arabski, przeglądarki mogą domyślnie brać różne regiony (np. ar-SA vs ar-EG) co wpływa np. na używany kalendarz czy system numeracyjny. W i18next można wymusić formatowanie liczb/dat pod konkretny region poprzez dostosowanie formattera lub wskazanie locale. Generalnie jednak, gdy aplikacja używa **języka urządzenia**, zwykle system podaje kod z regionem, więc można go wykorzystać.

Przykład formatowania liczby w i18next: w pliku tłumaczeń:

"file_size": "Rozmiar pliku: {{val, number}} MB"

Jeśli użytkownik ma locale pl-PL, to t('file_size', { val: 1234.5 }) zwróci np. *"Rozmiar pliku: 1 234,5 MB"*, a dla en-US *"File size: 1,234.5 MB"*. To zasługa Intl.NumberFormat pod spodem. Co więcej, można przekazywać opcje formatowania, np. {{val, number(minimumFractionDigits: 2)}} aby zawsze były 2 miejsca po przecinku. i18next udostępnia także skrót currency, np. {{price, currency(USD)}} wyświetli kwotę w dolarach z symbolami.

Podsumowując, **należy korzystać z mechanizmów formatowania odpowiednich dla locale zamiast sklejania dat ręcznie.** To gwarantuje poprawność i dostosowanie do przyzwyczajeń użytkownika. Upewnijmy się też, że w różnych językach uwzględniamy różne długości tekstów – np. tłumaczenie może być znacznie dłuższe i trzeba to zmieścić w UI (projektując elastyczne kontenery).

Podejście Offline-First w aplikacjach

Tradycyjnie aplikacje zakładały stałe połączenie z internetem – to podejście *online-first*. **Offline-first** to filozofia projektowania, w której aplikacja jest w stanie działać (przynajmniej częściowo) bez dostępu do sieci. Oznacza to, że **rdzenne funkcje aplikacji działają offline**, a gdy internet jest dostępny, następuje synchronizacja danych. W kontekście aplikacji mobilnych jest to niezwykle cenna cecha: użytkownicy często tracą zasięg (metro, samolot, słaby sygnał), a chcieliby nadal korzystać z aplikacji. Aplikacja offline-first nie wyświetli wtedy komunikatu "brak internetu – nie da się nic zrobić", tylko **pozwoli dalej przeglądać dane, wprowadzać zmiany, które zostaną zapisane i wysłane później**.

Najważniejsze składowe podejścia offline-first to **cache danych**, **kolejka zmian do synchronizacji** oraz **odpowiedni UI awaryjny (fallback)**:

- **Cache danych (pamięć podręczna):** Aplikacja powinna **przechowywać lokalnie kopie najważniejszych danych** pobranych z serwera, aby móc je wyświetlić, gdy połączenie zniknie. Przy pierwszym użyciu lub przy połączeniu, dane są pobierane i zapisywane w lokalnej bazie danych, plikach (np. AsyncStorage w RN, baza SQLite, Realm, itp.). W razie braku internetu aplikacja korzysta z **danych z cache**, dzięki czemu użytkownik ma dostęp do ostatnio załadowanych informacji. Przykłady: aplikacja newsowa może cache'ować ostatnie artykuły, aby dało się je czytać offline; mapy cache'ują kafelki map; sklep internetowy może przechować listę produktów przeglądanych ostatnio. Ważne jest, by implementować **mechanizmy odświeżania** – np. wersjonowanie cache lub jego unieważnianie, aby po odzyskaniu sieci pobrać nowsze dane (żeby nie tkwić na zawsze w starych informacjach). Caching to podstawa trybu offline – sprawia, że nasza aplikacja w ogóle **ma co pokazać** bez internetu.
- **Kolejka mutacji (operacji do wykonania):** Samo czytanie danych offline to jedno, ale co jeśli użytkownik **wykona akcję** offline – np. napisze komentarz, wypełni formularz, doda obiekt? Podejście offline-first zakłada, że takie operacje nie zostaną utracone. Zamiast odrzucić działanie, **aplikacja zapisuje je w kolejce** i oznacza do wysłania, gdy tylko połączenie wróci. Czyli np. wiadomość wysłana w komunikatorze offline powinna zostać zapisana lokalnie i oznaczona jako "oczekująca". Gdy sieć się pojawi, aplikacja automatycznie podejmie próbę wysłania tych zaległych akcji do serwera (**background sync**). Taka kolejka powinna być **trwała** (persistent), czyli nawet jak użytkownik zamknie aplikację zanim wróci sieć, to po ponownym uruchomieniu i odzyskaniu połączenia, synchronizacja powinna się odbyć. Technicznie można to zrobić np. zapisując akcje w lokalnej bazie z timestampem. Po odzyskaniu sieci – co można wykryć przez eventy systemowe lub bibliotekę typu NetInfo – aplikacja **próbuje wysłać** akcje z kolejki (w tle, niewidoczne dla użytkownika). Jeśli serwer potwierdzi ich przyjęcie, usuwa je z kolejki lokalnej. Mogą tu wystąpić konflikty (np. użytkownik zmienił coś offline, a w międzyczasie online nastąpiła inna zmiana) – wtedy trzeba przewidzieć **strategię rozwiązywania konfliktów** (np. ostatnia zmiana wygrywa, albo pytamy użytkownika). Najważniejsze jednak, by **żadne dane wpisane offline nie przepadły**. Przykłady: aplikacja ankietowa pozwala głosować offline i trzyma głosy w kolejce, by wysłać je później; aplikacja do zarządzania zadaniami zapisuje dodane/edytowane zadania offline i synchronizuje z serwerem przy najbliższej okazji. Implementując to, warto też wprowadzić **limit ponawiania** i mechanizm backoff (np. jeśli wysyłanie się nie udaje, próbuj ponownie coraz rzadziej) oraz zadbać o bezpieczeństwo – dane w kolejce też mogą być wrażliwe, więc powinny być przechowywane bezpiecznie i przesyłane po odzyskaniu sieci w sposób zabezpieczony (TLS).
- **Fallback UI (interfejs awaryjny):** Użytkownik powinien być **informowany o stanie offline** i o tym, co aplikacja robi. Dobre praktyki to np. **oznaczanie treści, które pochodzą z cache** (żeby było jasne, że mogą być nieaktualne), komunikat "Jesteś w trybie offline – przeglądasz ostatnio pobrane dane". Jeśli użytkownik wykonał akcję (np. wysłał wiadomość), można ją od razu pokazać w interfejsie (tzw. **optymistyczna aktualizacja UI**), ale oznaczyć np. szarym kolorem lub ikonką zegara, że czeka na wysłanie. W razie gdyby synchronizacja się nie powiodła, należy dać możliwość

ponownego wysłania lub poinformować o problemie. **Obsługa błędów i mechanizmy zapasowe** są ważne: aplikacja może np. jeśli kolejka nie może się wysłać przez dłuższy czas, powiadomić użytkownika "Twoje dane zostaną wysłane gdy tylko odzyskamy połączenie" zamiast trzymać go w niepewności. Powinna też mieć **plan awaryjny**: jeśli naprawdę nic nie możemy załadować (pierwsze uruchomienie offline, brak cache) – wyświetlamy przyjazny komunikat lub ograniczoną funkcjonalność zamiast zawieszenia. OWASP (oraz inne źródła) zalecają np. **cache fallback** – pokazywanie *"last known good state"* danych, gdy nie można pobrać nowszych. Użytkownik wtedy widzi przynajmniej coś, zamiast pustego ekranu. Ponadto **informuj o stanie synchronizacji** – np. jak tylko sieć wróci, możesz pokazać toast "Połączenie przywrócone – dane zostają zsynchronizowane". Gdy operacje się powiodą, usuń oznaczenia oczekujących elementów. Taka transparentność zwiększa zaufanie użytkownika do aplikacji.

Podsumowując, aplikacja offline-first wykorzystuje **lokalną pamięć** jako źródło prawdy, a sieć służy do wymiany danych w tle. Wymaga to więcej pracy przy implementacji, ale znacząco poprawia UX – aplikacja jest **szybka** (bo czyta lokalnie) i **odporna na słaby internet**. Przykłady udanych wdrożeń to choćby mobilna aplikacja Trello, gdzie tablice i karty można przeglądać i edytować offline, a zmiany zostaną zsynchronizowane automatycznie po odzyskaniu sieci. W dobie roku 2025 użytkownicy wręcz **oczekują** pewnej dozy działania offline, a narzędzia (np. biblioteki Redux Offline, WatermelonDB, Apollo Client z cache persist, itp.) znacznie to ułatwiają.

Przykład: i18n, dostępność i cache w praktyce

Na koniec połączmy powyższe zagadnienia w mini demonstrację. Wyobraźmy sobie prosty formularz logowania w aplikacji React Native, do którego dodamy **obsługę wielu języków (i18n)** oraz **ulepszenia dostępności**, a także zaimplementujemy **prosty mechanizm offline** dla listy danych.

1. Dodanie i18n do formularza: Załóżmy, że mamy formularz z polami "Email", "Hasło" i przyciskiem "Zaloguj". Chcemy obsługiwać angielski i polski. Korzystamy z i18next:

Przygotowujemy pliki tłumaczeń `en.json` i `pl.json`:

```
// pl.json
{
  "login_title": "Logowanie",
  "email_label": "Email",
  "password_label": "Hasło",
  "login_button": "Zaloguj"
}
// en.json
{
  "login_title": "Login",
  "email_label": "Email",
  "password_label": "Password",
  "login_button": "Log In"
}
```

- Inicjalizujemy i18next z tymi zasobami (podobnie jak opisano wcześniej, używając initReactI18next). Ustawiamy domyślny język na np. polski, ale też mechanizm wykrywania (żeby np. dopasować do języka telefonu).
- W komponencie formularza korzystamy z hooka useTranslation(). Następnie zamiast na sztywno wpisywać teksty, używamy t('email_label'), t('password_label') itp.:

```
import { useTranslation } from 'react-i18next';
...
const { t } = useTranslation();
return (
  <View>
    <Text style={styles.title}>{t('login_title')}</Text>
    <TextInput placeholder={t('email_label')} ... />
    <TextInput placeholder={t('password_label')} secureTextEntry ... />
    <Button title={t('login_button')} onPress={handleLogin} />
  </View>
);
```

- Teraz interfejs będzie w wybranym języku. Aby zmienić język, wystarczy wywołać i18n.changeLanguage('en') – wszystkie komponenty korzystające z t zareagują i pokażą teksty po angielsku. Dzięki i18n łatwo rozbudujemy aplikację o kolejne języki bez modyfikacji kodu logiki.

2. Etykiety dostępności: Kontynuując nasz formularz – dodamy atrybuty ułatwiające życie osobom używającym czytników ekranu:

- **Label dla tekstu tytułowego:** Jeśli "Logowanie" jest nagłówkiem ekranu, można oznaczyć go rolą nagłówka. W RN moglibyśmy zrobić <Text accessibilityRole="header">....
- **Pola formularza:** element <TextInput> nie jest automatycznie odczytywany z etykietą (placeholder bywa czytany, ale lepiej dać własną). Można podejść dwojako: albo użyć komponentu <Label> związanego z polem (w natywnym iOS/Android można łączyć etykiety z polami), albo w RN skorzystać z accessibilityLabel. Np.:

```
<TextInput
  placeholder={t('email_label')}
  accessibilityLabel={t('email_label')}
  ... />
```

W ten sposób VoiceOver przeczyta "Email, pole edycji tekstu". Podobnie dla hasła:

```
<TextInput
  placeholder={t('password_label')}
  accessibilityLabel={t('password_label')}
  secureTextEntry={true}
  accessibilityHint="Pole hasła. Tekst niewidoczny podczas pisania."
  ... />
```

Tutaj użyliśmy też accessibilityHint, aby dodać wskazówkę, że wpisywany tekst będzie maskowany (czytnik mógłby i tak to wiedzieć, bo secureTextEntry ustawia rolę hasła).

- **Przycisk logowania:** Używając komponentu <Button> w RN, on sam przekazuje tekst jako etykietę. Ale jeśli to byłby np. ikona zamiast tekstu, konieczne byłoby dodanie accessibilityLabel={t('login_button')}. Upewniamy się też, że rola jest właściwa (Button zwykle ma już role button).
- **Grupowanie i kolejność:** Sprawdzamy, czytnikiem ekranu, że fokus idzie kolejno: tytuł "Logowanie" (nagłówek), pole Email, pole Hasło, przycisk Zaloguj. Jeśli np. UI jest w kolumnie, to tak będzie. Gdyby kolejność była niepoprawna, moglibyśmy owinąć pola w kontener z accessible={false} lub zastosować wspomniane atrybuty kolejności (ale to raczej zbędne tutaj).
- **Rozmiar przycisków i czytelność:** Upewnimy się, że elementy dotykowe są wystarczająco duże (min. 44x44pt według Apple – w RN łatwo osiągalne jeśli używamy standardowych przycisków). Kolory tekstu i tła – np. biały tekst "Zaloguj" na niebieskim przycisku – powinny mieć kontrast > 4.5:1. Jeśli tło jest jasne, tekst musi być ciemny i vice versa.

3. Prosty cache offline dla listy: Teraz załóżmy, że po zalogowaniu aplikacja wyświetla listę np. ostatnich wiadomości lub produktów. Chcemy zaimplementować mechanizm cache + offline:

- **Przechowywanie danych:** Wybierzemy najprostszą metodę – **AsyncStorage** (w RN jest to moduł zapewniający prosty asynchroniczny magazyn klucz-wartość). Gdy pobieramy listę z API, po udanym pobraniu zapiszemy ją również w AsyncStorage. Np.:

```
async function fetchData() {
  try {
    const response = await fetch(API_URL);
    const data = await response.json();
    setItems(data);
    await AsyncStorage.setItem('itemsCache', JSON.stringify(data));
  } catch (err) {
    console.error(err);
  }
}
```

  Ta funkcja pobiera dane i zapisuje cache pod kluczem 'itemsCache'.

- **Odczyt z cache przy braku internetu:** Potrzebujemy wykryć stan offline. Możemy skorzystać z biblioteki @react-native-community/netinfo – daje eventy o zmianie sieci. Jeśli okaże się, że użytkownik jest offline (lub fetch zakończy się błędem sieci), to:

```
const cached = await AsyncStorage.getItem('itemsCache');
if(cached) {
  setItems(JSON.parse(cached));
} else {
  // Brak cache - pokaż komunikat o braku danych
}
```

  Czyli wyświetlamy dane z cache (jeśli istnieją). Warto przy tym np. oznaczyć w UI, że to dane offline (np. szary baner "Offline mode: showing cached data").

- **Aktualizacja cache:** Przy ponownym połączeniu, możemy automatycznie odświeżyć dane. Jeśli używamy NetInfo, możemy nasłuchiwać kiedy `isConnected` zmienia się na true i wtedy wykonać `fetchData()` ponownie. Po odświeżeniu lista pokazuje najnowsze dane, a cache zostaje zaktualizowany.
- **Kolejka zmian (prostota):** Jeśli nasza lista pozwala np. usuwać elementy, a użytkownik zrobi to offline – możemy zapisać taką akcję w kolejce (np. AsyncStorage.setItem 'queuedDeletes', itp.). W ramach tego prostego przykładu można założyć, że lista jest tylko do odczytu offline, a zmiany wymagają internetu (to upraszcza sprawę). W bardziej rozbudowanej appce użylibyśmy np. biblioteki **Redux Offline** albo napisali własny mechanizm kolejek jak opisano wcześniej.
- **Fallback UI:** W naszym przykładzie, **gdy brak internetu i brak cache**, powinniśmy pokazać użytkownikowi np. ekran z komunikatem "Brak połączenia. Spróbuj ponownie później." zamiast pustej listy. Gdy cache jest, pokażemy listę z cache + np. ikonę offline. Dobrze jest też udostępnić przycisk "Spróbuj ponownie" do ręcznego odświeżenia, który wymusi sprawdzenie połączenia.
- **Bezpieczeństwo danych cache:** Jeśli lista zawiera bardzo wrażliwe dane, należałoby rozważyć szyfrowanie przed zapisem do AsyncStorage (bo AsyncStorage na iOS/Android jest co prawda prywatne dla aplikacji, ale przechowywane w zwykłym pliku). Dla naszych potrzeb demo, można pominąć.

Tak zaimplementowany prosty mechanizm zapewni, że użytkownik **widzi ostatnie znane dane nawet bez internetu**, co znacząco poprawia doświadczenie. W realnej aplikacji można pójść dalej – np. użyć biblioteki typu **Realm** lub **WatermelonDB** do przechowywania danych lokalnie i mechanizmów sync. Ważne jednak, by już na etapie projektowania przewidzieć te scenariusze offline.

Literatura:

1. **https://owasp.org/www-project-mobile-top-10/** (Data dostępu: 1.10.2025)
2. **https://reactnative.dev/docs/accessibility** (Data dostępu: 1.10.2025)
3. **https://www.i18next.com/overview/typescript** (Data dostępu: 1.10.2025
4. **https://developer.android.com/training/articles/keystore** (Data dostępu: 1.10.2025)
5. **https://developer.apple.com/documentation/security/keychain_services** (Data dostępu: 1.10.2025)