Aplikacje mobilne

Wykład 2

Podstawowe komponenty React Native

Mateusz Pawełkiewicz

Październik 2026

Expo SDK 58 · React Native 0.88 · React 19.3 · TypeScript

React Native udostępnia zestaw podstawowych komponentów do budowy interfejsu aplikacji mobilnej. Oto najważniejsze z nich:

- View – podstawowy komponent-kontener do budowy interfejsu (odpowiednik w sieci). Służy jako kontener widoku, w którym można układać inne komponenty.
- Text – komponent służący do wyświetlania tekstu. Każdy fragment tekstu w aplikacji musi znajdować się wewnątrz komponentu `<Text>`. Pozwala to na stosowanie stylów tekstowych i jest wymogiem platformy (nie można po prostu wstawić surowego tekstu bez `<Text>`).
- Image – komponent do wyświetlania obrazów (zarówno z sieci, jak i lokalnych zasobów). Pozwala określić źródło obrazka poprzez właściwość source (np. source={`require('path/to/image.png')`} dla zasobów lokalnych lub source={{uri: 'https://adres/obraz.jpg'}} dla obrazów zdalnych). Można go stylować (rozmiary, obramowania itp.) podobnie jak inne komponenty.
- ScrollView – komponent kontenera umożliwiający przewijanie zawartości. Używamy go, gdy chcemy, aby wewnątrz ekranu dało się scrollować większą ilość komponentów lub treści. Pamiętajmy jednak, że ScrollView renderuje wszystkie swoje dzieci naraz, więc przy bardzo długich listach może to być nieefektywne (w takich przypadkach stosujemy komponenty list, o czym dalej).
- Pressable – nowoczesny komponent dotykowy, który może reagować na różne etapy dotyku (naciśnięcie, przytrzymanie, puszczenie itp.). Stanowi uniwersalny zamiennik starszych komponentów dotykowych (jak TouchableOpacity, TouchableHighlight itd.). Przykład użycia:

```
<Pressable onPress={() => console.log('Naciśnięto!')}>
<Text>Dotknij mnie</Text>
</Pressable>
```

Powyższy kod stworzy element reagujący na dotknięcie, a wewnątrz niego wyświetli tekst. Pressable pozwala także na definiowanie callbacków dla zdarzeń takich jak `onPressIn`, `onPressOut`, `onLongPress` itp., dając dużą kontrolę nad zachowaniem podczas interakcji użytkownika.

Powyższe komponenty są fundamentem interfejsu w React Native – większość ekranów będzie zbudowana z View jako kontenerów, Text do wyświetlania napisów, Image do grafik oraz elementów przewijanych (ScrollView lub list). Interakcje użytkownika obsłużymy za pomocą Pressable. Wbudowany `<Button>` to prosty przycisk systemowy z bardzo ograniczonymi możliwościami stylowania – w praktyce własne przyciski budujemy na Pressable. Warto zaznaczyć, że React Native oferuje też wiele innych komponentów (np. TextInput do pól tekstowych, Switch do przełączników, Modal do wyświetlania okien modalnych itp.), ale w tym wykładzie skupiamy się na tych podstawowych, potrzebnych do layoutu i list.

Komponenty Expo, które warto znać od początku

- **expo-image** – zamiennik `Image`: pamięć podręczna na dysku, rozmyty podgląd (`placeholder={{ blurhash }}`), płynne pojawianie się (`transition`), `contentFit` zamiast `resizeMode`, obsługa formatów WebP/AVIF/SVG.
- **expo-linear-gradient** – gradienty (np. przyciemnienie zdjęcia pod tekstem), **expo-blur** – rozmycie tła (np. nagłówek „szklany”).
- **Modal, TextInput, Switch, ActivityIndicator, RefreshControl** – kolejne wbudowane komponenty używane w zadaniu praktycznym do tego wykładu.

expo-image

```
import { Image } from 'expo-image';

<Image
  source={{ uri: movie.posterUrl }}
  placeholder={{ blurhash: 'L5H2EC=PM+yV0g-mq.wG9c010J}I' }}
```

```
  contentFit="cover"
  transition={250}
  style={{ width: '100%', aspectRatio: 2 / 3, borderRadius: 12 }}
/>
```

Flexbox – układ elementów w kontenerze

Domyślnym mechanizmem rozmieszczania komponentów w React Native jest Flexbox. Pozwala on w wygodny sposób definiować układ interfejsu w dwóch osiach – głównej i krzyżowej – analogicznie do Flexboxa w CSS na webie. Uwaga: W React Native Flexbox ma kilka innych wartości domyślnych niż w CSS – np. domyślny kierunek układu (`flexDirection`) to column (kolumna) zamiast row. Oznacza to, że jeśli nie podamy inaczej, komponenty w View będą układać się pionowo (jeden pod drugim).

Główne właściwości Flexbox, z których korzystamy, to:

- `flexDirection` – określa kierunek ułożenia elementów w kontenerze (oś główna). Możliwe wartości to:
  - "column" (domyślnie) – układa dzieci od góry do dołu (pionowo).
  - "row" – układa dzieci od lewej do prawej (poziomo).
  - (również "column-reverse" i "row-reverse" dla odwrócenia kolejności).
- `justifyContent` – definiuje wyrównanie elementów wzdłuż osi głównej (czyli w kierunku ustawionym przez `flexDirection`). Typowe wartości:
  - flex-start (domyślnie) – elementy przy początku osi (lewa strona lub góra kontenera),
  - center – elementy wyśrodkowane wzdłuż osi głównej kontenera,
  - space-between, space-around, space-evenly – różne sposoby równomiernego rozłożenia wolnej przestrzeni między elementami.
- `alignItems` – definiuje wyrównanie wzdłuż osi krzyżowej (czyli prostopadłej do głównej). Przykładowe wartości:
  - stretch (domyślnie) – rozciąga elementy aby wypełniły przestrzeń na osi krzyżowej (np. zazwyczaj wypełnia na pełną szerokość przy `flexDirection`: column),
  - flex-start / flex-end – wyrównanie do początku lub końca osi krzyżowej (np. do lewej lub prawej krawędzi kontenera przy ułożeniu w kolumnie),
  - center – centrowanie elementów na osi krzyżowej (np. poziome wycentrowanie elementów w kolumnie).

Dzięki kombinacji tych właściwości możemy łatwo tworzyć układy typu „wiersz” czy „kolumna” i kontrolować rozmieszczenie elementów (np. wyśrodkować coś, rozsunąć równomiernie, przyciągnąć do krawędzi itp.). Np. aby wycentrować elementy zarówno pionowo jak i poziomo wewnątrz kontenera, ustawiamy styl:

```
<View style={{ flex: 1, justifyContent: 'center', alignItems: 'center' }}>
 {/* ... */}
</View>
```

W powyższym przykładzie kontener `<View>` z `justifyContent`: 'center' i `alignItems`: 'center' spowoduje, że jego dzieci będą wyśrodkowane w obu kierunkach.

Właściwość flex – Oprócz powyższych, często korzystamy też z właściwości flex (liczbowej) nadawanej dzieciom kontenera. Pozwala ona określić, jaką część dostępnego miejsca zajmie dany element. Przykład: jeśli w wierszu mamy trzy kolumny z stylami flex: 1, flex: 2, flex: 3, to będą one zajmować odpowiednio 1/6, 2/6 i 3/6 szerokości wiersza (bo 1+2+3=6).

Oś główna vs. oś krzyżowa: Warto zrozumieć, że przy `flexDirection`: 'column' osią główną jest pion (od góry do dołu), a osią krzyżową poziom; natomiast przy `flexDirection`: 'row' odwrotnie –

główna to poziom, krzyżowa to pion. Dlatego np. `justifyContent`: 'center' przy `flexDirection`: 'row' centrować będzie poziomo, a przy `flexDirection`: 'column' – pionowo.

Gap (odstępy między elementami): Właściwość gap – podobnie jak w CSS – pozwala łatwo ustawić odstęp między wszystkimi elementami wewnątrz kontenera flex. Wartości podajemy w punktach niezależnych od gęstości ekranu (dp), np. gap: 10 to 10 dp przerwy między sąsiadującymi dziećmi kontenera; dostępne są też osobne `rowGap` i `columnGap`. Dzięki gap można uniknąć dodawania marginesów do każdego elementu i np. problemu podwójnych odstępów wewnątrz – Flexbox sam rozdziela równo przestrzeń między elementy. Przykładowo:

```
<View style={{ flexDirection: 'row', gap: 8 }}>
 {/* ... dzieci w wierszu z odstępem 8px ... */}
</View>
```

spowoduje, że wszystkie elementy wewnątrz będą rozdzielone przerwą 8 dp (wewnętrzne krawędzie), bez dodatkowego marginesu na zewnętrzach.

Flex Wrap: Domyślnie kontener Flexbox w RN nie zawija elementów do nowej linii – jeśli elementy się nie mieszczą, zostaną zwyczajnie "wycięte" (poza ekran). Aby to zmienić, używamy stylu `flexWrap`: 'wrap' na kontenerze. Wówczas dzieci, które nie mieszczą się w jednym wierszu/kolumnie, przejdą automatycznie do następnej linii/kolumny. W połączeniu z `flexWrap`, użyteczna może być też właściwość `alignContent`, która określa wyrównanie całych linii względem kontenera (np. `alignContent`: 'center' wycentruje ułożone linie w przestrzeni kontenera). `alignContent` działa tylko gdy `flexWrap` jest włączony i mamy więcej niż jedną linię elementów.

Style znane z CSS

W nowej architekturze React Native dostępne są style znane z CSS:

- **boxShadow** – cień jak w CSS na iOS i Androidzie (zastępuje `shadowColor/shadowOffset/shadowRadius` z iOS i `elevation` z Androida), np. `boxShadow: "0 2px 8px rgba(0,0,0,0.25)"`,
- **filter** – np. `filter: "grayscale(1)"` lub `"brightness(0.8)"` (część filtrów tylko na Androidzie),
- **aspectRatio** – proporcje bez liczenia wysokości (`aspectRatio: 16 / 9`),
- **position: "static"**, wartości procentowe dla wymiarów i marginesów, `StyleSheet.absoluteFill` (wypełnienie całego rodzica).

Responsywność: dostosowanie do różnych ekranów

Flexbox a responsywność: Sam Flexbox został zaprojektowany tak, by ułatwić tworzenie responsywnych układów na różne rozmiary ekranu. W praktyce wiele interfejsów może być zbudowanych tak, że automatycznie rozciągają się lub kurczą, by wypełnić dostępne miejsce (dzięki użyciu flex i właściwości procentowych dla szerokości/wysokości). Często nie trzeba więc ręcznie obliczać pikseli dla różnych urządzeń – elementy z flex: 1 same podzielą przestrzeń, a elementy zdefiniowane np. width: '50%' zajmą połowę rodzica niezależnie od urządzenia.

Dimensions API: Czasem jednak potrzebujemy znać rozmiar ekranu lub dostosować układ warunkowo (np. inny układ dla telefonu, inny dla tabletu). React Native udostępnia moduł Dimensions do pobrania wymiarów okna aplikacji. Przykład użycia:

```
import { Dimensions } from 'react-native';
const windowWidth = Dimensions.get('window').width;
const windowHeight = Dimensions.get('window').height;
Dzięki temu możemy np. sprawdzić if (windowWidth > 600) { /* duży ekran - np. tablet */ } i
```

warunkowo zmienić układ. Należy jednak pamiętać, że wartości z `Dimensions.get()` są stałe w czasie życia aplikacji, chyba że podepniemy się do zdarzeń zmiany. Różne sytuacje (obrót ekranu, tryb podzielonego ekranu, urządzenia składane) mogą zmienić wymiary w trakcie działania aplikacji. Można podpiąć listener: `Dimensions.addEventListener('change', handler)` aby reagować na zmianę rozmiarów, lecz istnieje wygodniejsze podejście – hook.

`useWindowDimensions`: React Native oferuje hook `useWindowDimensions()`, który dostarcza zawsze aktualne wymiary okna i automatycznie aktualizuje wartości przy zmianie rozmiaru okna lub orientacji ekranu. Użycie jest proste:

```
import { useWindowDimensions } from 'react-native';
const { width, height } = useWindowDimensions();
```

Taki kod wewnątrz komponentu React pobierze bieżącą width i height ekranu, a gdy tylko nastąpi zmiana (np. użytkownik obróci urządzenie z pionu do poziomu), powoduje to ponowne renderowanie komponentu z nowymi wymiarami. Zalecenie: W nowoczesnych aplikacjach preferujemy `useWindowDimensions` zamiast statycznego `Dimensions.get()`, właśnie dlatego że automatycznie reaguje na zmiany rozmiaru i lepiej wpisuje się w paradygmat reaktywny Reacta.

Przykład responsywności: Załóżmy, że chcemy zbudować siatkę kafelków, która na węższych ekranach ma jeden kafelek na wiersz, a na szerszych – dwa obok siebie. Możemy to osiągnąć np. tak:

```
const { width } = useWindowDimensions();
const columns = width > 500 ? 2 : 1;
...
<FlatList
 data={elements}
 numColumns={columns}
 renderItem={...}
/>
```

Tutaj określamy liczbę kolumn listy na podstawie szerokości ekranu – powyżej 500 pikseli będziemy mieć 2 kolumny (np. tablet w poziomie), a poniżej – 1 kolumnę (typowy telefon). Dzięki temu interfejs automatycznie się dostosuje. Można też warunkowo stosować inne style: np. dla dużego ekranu ustawić większe marginesy, większą czcionkę itp., wykorzystując zwykłe instrukcje warunkowe w kodzie komponentu.

Uwaga na różne gęstości pikseli: Warto wspomnieć, że wymiary zwracane przez RN (np. w Dimensions czy `useWindowDimensions`) to wymiary w punktach dp, nie w surowych pikselach urządzenia. Oznacza to, że np. szerokość 375 na `iPhonie` oznacza 375 punktów, co może odpowiadać 750 fizycznym pikselom przy Retina 2x. Na szczęście, zazwyczaj nie musimy się tym przejmować, bo RN sam przelicza wielkości elementów na piksele urządzenia, utrzymując spójny wygląd. Jeśli jednak potrzebujemy znać skalę pikseli, jest do tego moduł PixelRatio, ale w kontekście typowej responsywności (dostosowania układu) nie jest to często potrzebne. Podsumowując, responsywność w React Native osiągamy głównie poprzez elastyczne style (Flexbox, procentowe wymiary) oraz ewentualnie poprzez dynamiczne sprawdzanie wymiarów okna (hook `useWindowDimensions`, warunkowe kolumny, warunkowe style). Nie ma tutaj mechanizmu CSS Media Queries jak w webie – musimy polegać na JavaScript i własnej logice do wykrycia np. „trybu tabletowego”. Niektóre biblioteki UI robią to za nas, ale na poziomie RN warto znać narzędzia wymienione powyżej.

Style: StyleSheet vs podejście utility (NativeWind)

Stylowanie w React Native odbywa się za pomocą JavaScriptowych obiektów stylów, inspirowanych CSS. Mamy dwa główne podejścia do definiowania stylów:

1. StyleSheet (wbudowany moduł RN): Używamy statycznych obiektów stylów, zwykle tworzonych przez `StyleSheet.create()`. Na przykład:

```
const styles = StyleSheet.create({
 container: {
  backgroundColor: '#fff',
  padding: 16,
  borderRadius: 8
 },
 title: {
  fontSize: 18,
  fontWeight: 'bold'
 }
});
...
<View style={styles.container}>
 <Text style={styles.title}>Tytuł</Text>
</View>
```

StyleSheet pełni rolę podobną do arkuszy stylów CSS – pozwala zdefiniować zestaw stylów i odwoływać się do nich po nazwie. W aktualnych wersjach RN `StyleSheet.create()` zwraca po prostu przekazany obiekt (bez walidacji i bez „rejestrowania" stylów w kodzie natywnym) – jego zaletami są typowanie w TypeScript, czytelna struktura i stała referencja obiektu między renderami.

Zalety podejścia StyleSheet:

- **Czytelność i struktura:** Stylom nadajemy nazwy (klucze obiektu) co wymusza pewną organizację. Style są zebrane w jednym miejscu, łatwe do edycji.
- **Typowanie:** w TypeScript `StyleSheet.create()` podpowiada nazwy właściwości i wartości – literówkę zobaczymy w edytorze, a nie dopiero na ekranie.
- **Stała referencja:** obiekt stylów tworzony raz, poza komponentem, ma tę samą referencję przy każdym renderze – nie wymusza ponownego renderu dzieci opakowanych w `memo` (w przeciwieństwie do obiektu `style={{ … }}` tworzonego w JSX).

Wady:

- **Wiele plików/wierszy:** Tworzenie wielu stylów powoduje rozbudowę kodu. Dla prostych komponentów mamy sporo boilerplate – najpierw zdefiniuj style, potem używaj.
- **Brak dynamiczności:** StyleSheet jest statyczny. Jeśli chcemy zmieniać styl zależnie od propsów czy stanu (np. kolor tła zależny od zmiennej), często i tak kończy się na konstrukcji typu style={[styles.base, condition && styles.highlight]} lub inline style. StyleSheet nie oferuje mechanizmu generowania stylu „w locie" na podstawie zmiennych poza takimi trikami.

2. Utility-first (np. NativeWind – Tailwind CSS dla RN): To podejście polega na korzystaniu z gotowych klas użyteczności (utility classes), podobnie jak w bibliotece Tailwind CSS na webie. Zamiast definiować własne style, składamy je z predefiniowanych skrótów. Przykład z użyciem NativeWind (biblioteka implementująca Tailwind w RN):

NativeWind 4

```
// NativeWind 4: po konfiguracji (tailwind.config.js, global.css, preset w babel/metro)
// className działa bezpośrednio na komponentach React Native – bez funkcji styled()
import { Text, View } from 'react-native';
import './global.css';

export function Card() {
  return (
    <View className="bg-white p-4 rounded-lg dark:bg-neutral-900">
      <Text className="text-lg font-bold text-blue-500">Tekst</Text>
    </View>
  );
```

```
}
```

> **Konfiguracja NativeWind**
>
> Klasy Tailwind podajemy bezpośrednio w `className` – bez dodatkowych funkcji opakowujących. Konfigurację (Tailwind CSS 3/4, `global.css`, `withNativeWind` w `metro.config.js`) opisuje dokumentacja nativewind.dev – zawsze sprawdzaj wersję pakietu w projekcie.

Tutaj używamy atrybutu `className` jak w HTML – biblioteka NativeWind tłumaczy te klasy na rzeczywiste style RN. Np. bg-white ustawia `backgroundColor`: '#fff', p-4 to padding: 16 (zakładając, że 1 jednostka w Tailwind to 4 px, co w RN przekłada się na 4 punkty), rounded-lg to jakiś predefiniowany `borderRadius`, itd.

Zalety podejścia utility/Tailwind:

- **Szybkość developmentu:** Można bardzo szybko ostylować komponent, pisząc style w jednej linii zamiast deklarować obiekt. Dla osób zaznajomionych z Tailwind na webie, klasy są intuicyjne i spójne między projektami.
- **Spójność designu:** Tailwind narzuca korzystanie z pewnej skali (np. tylko określone rozmiary czcionek, spacingów). To często pomaga utrzymać spójny wygląd aplikacji i zgodność z design systemem.
- **Mniej plików/oddzielenia:** Nie trzeba przełączać się między plikiem ze stylami a kodem komponentu – style (klasy) są blisko oznaczeń JSX, co niektórzy preferują (łatwiej powiązać styl z komponentem, bo widać go obok).

Wady podejścia utility-first:

- **Długawe klasy w kodzie:** Może się zdarzyć, że komponent ma bardzo długi string z klasami Tailwind, co bywa trudne w czytaniu, zwłaszcza jeśli są warunkowe (np. doklejanie klas w zależności od stanu). Kod staje się szerszy zamiast dłuższego – co nie każdemu odpowiada (choć można argumentować, że to rzecz gustu).
- **Ograniczona paleta:** Tailwind ma zdefiniowane konkretne wartości (np. kolory, rozmiary). Gdy potrzebujemy czegoś niestandardowego (np. nietypowy odcień koloru czy niestandardowy margines), musimy albo dopisać własne rozszerzenie do konfiguracji, albo posiłkować się bezpośrednim stylem. To oznacza, że nie wszystkie przypadki da się pokryć gotowymi utility – czasem i tak musimy napisać kawałek zwykłego stylu.
- **Abstrakcja na abstrakcji:** Dla osób, które dobrze znają CSS lub stylowanie RN, Tailwind może być dodatkową warstwą do nauczenia. StyleSheet RN już jest pewną abstrakcją CSS, a Tailwind/NativeWind to kolejna abstrakcja nad tym (specjalne klasy, konfiguracja). Niektórzy programiści wolą mieć pełną kontrolę i jawnie pisać style w JavaScripcie, niż polegać na stringach klas interpretowanych przez bibliotekę.

Wydajność: W kontekście RN różnice wydajności między użyciem StyleSheet a np. NativeWind nie są duże dla prostych ekranów. NativeWind działa podczas renderowania, parsując klasy (prawdopodobnie z pewną optymalizacją, np. kompilacją tych stylów przy starcie). StyleSheet natomiast tworzy gotowe style wcześniej. W ogromnych listach elementów utility-first teoretycznie może być minimalnie wolniejsze przy pierwszym renderze (bo musi przetworzyć wiele stringów stylów). Jednak w praktyce obie metody są używane w środowisku produkcyjnym z powodzeniem.

Podsumowanie wyboru: Wybór między StyleSheet a podejściem utility zależy od preferencji i wymagań projektu:

- StyleSheet daje pełną kontrolę, łatwość tworzenia własnych design systemów w kodzie (np. definicja własnych zestawów kolorów, spacingów jako stałe JS). Jest to natywne dla RN, więc nie wymaga dodatkowych zależności.

- NativeWind/Tailwind przyspiesza pracę, zwłaszcza jeśli ktoś zna Tailwind – dużo rzeczy z marszu zrobimy jednym ciągiem klas. Sprawdza się w szybkim prototypowaniu lub gdy chcemy zachować spójność z projektem webowym wykorzystującym Tailwind. Należy jednak pamiętać, że RN to nie web – brak tam kaskadowości CSS czy media queries, więc niektóre zalety Tailwinda z weba nie mają tu zastosowania (np. Tailwind na webie mocno pomaga w zarządzaniu responsywnością przez breakpoints – w RN i tak zrobimy to w JS; pomaga w walce z kaskadą CSS – w RN nie ma kaskady).

W wielu projektach można z powodzeniem łączyć podejścia. Np. używać Tailwind (NativeWind) do prototypu lub layoutu, a dla bardzo specyficznych rzeczy czy dużych komponentów przejść na StyleSheet. Kluczowe to spójność – najlepiej trzymać się jednego stylu w obrębie projektu, by zespół nie musiał żonglować dwoma systemami.

Listy: FlatList, SectionList, klucze i optymalizacja renderowania

W aplikacjach mobilnych często wyświetlamy listy danych – czy to lista kontaktów, wiadomości, postów itp. React Native oferuje specjalne komponenty list zaprojektowane z myślą o wydajności:

- FlatList – podstawowy komponent listy, który wydajnie renderuje przewijalną listę elementów jednopoziomowych.
- SectionList – odmiana listy, która obsługuje podział na sekcje z nagłówkami (np. lista pogrupowana literami alfabetu, gdzie każda sekcja ma tytuł). Działa podobnie do FlatList, ale przyjmuje dane pogrupowane w sekcje.

Czym różnią się te komponenty od omówionego wcześniej ScrollView? Przede wszystkim wydajnością przy dużej liczbie elementów. ScrollView renderuje wszystkie dzieci od razu, więc jeśli mielibyśmy 1000 elementów, to zostaną one wszystkie utworzone w pamięci (co może być wolne i zużywać dużo RAMu). FlatList/SectionList działają inaczej: renderują tylko elementy aktualnie widoczne na ekranie (oraz trochę zapasu przed i za ekranem). Gdy użytkownik scrolluje, elementy które wyszły poza widok są usuwane (lub „recyklingowane”), a te które się pojawiają – tworzone na bieżąco. To podobny koncept do „wirtualnej listy” czy RecyclerView na Androidzie.

Użycie FlatList: Wymaga przekazania przynajmniej dwóch rzeczy: źródła danych (data) oraz funkcji renderującej element `renderItem`. Przykład:

```
const data = [{ id: '1', title: 'Pierwszy' }, { id: '2', title: 'Drugi' }, ...];

<FlatList
 data={data}
 renderItem={({ item }) => <Text>{item.title}</Text>}
 keyExtractor={item => item.id}
/>
```

Tutaj przekazujemy tablicę obiektów data oraz renderujemy każdy element jako `<Text>` z tytułem. Użyliśmy także `keyExtractor` aby poinformować listę, że kluczem unikalnym każdego elementu jest pole id obiektu. Klucze są bardzo ważne – podobnie jak przy listach w React na webie, klucz umożliwia optymalizację renderowania i jednoznaczną identyfikację elementu. FlatList domyślnie będzie szukać pola key w danych (tj. item.key) i go użyje, a jeśli nie znajdzie, użyje indeksu tablicy jako klucza. Używanie indeksu nie jest zalecane, gdyż jeśli zmieni się kolejność lub dojdą/ubędą elementy, mogą wyniknąć błędne przejścia/animacje. Dlatego albo nazwiemy pole id jako key, albo – tak jak wyżej – skorzystamy z `keyExtractor` by jawnie wskazać właściwość zawierającą identyfikator.

SectionList jest bardzo podobna w użyciu, z tą różnicą, że data ma strukturę tablicy sekcji, np.

```
const sections = [
 { title: 'A', data: ['Ala', 'Albert'] },
 { title: 'B', data: ['Beata', 'Bolesław'] }
```

```
];
<SectionList
 sections={sections}
 renderItem={({ item }) => <Text>{item}</Text>}
 renderSectionHeader={({ section }) => <Text>{section.title}</Text>}
/>
```

Tutaj każda sekcja ma nagłówek (renderowany przez `renderSectionHeader`) i własną listę danych. SectionList sam zadba o wyświetlenie sekcji w odpowiedniej kolejności. Optymalizacja list – klucze i memoizacja: Klucze już omówiliśmy – muszą być unikalne i stałe dla danych. Drugim aspektem wydajności jest unikanie niepotrzebnych renderów elementów, gdy np. zmienia się stan niezwiązany z danym elementem. React Native FlatList wewnętrznie jest zaimplementowana jako PureComponent, co oznacza, że sprawdza płytko propsy i jeśli te się nie zmieniły, to nie przerysowuje całej listy. Jednak i tak każdemu elementowi listy przekazujemy pewne propsy (np. item) i często anonimową funkcję renderującą. Aby ograniczyć renderowanie poszczególnych itemów listy gdy nie jest to potrzebne, możemy skorzystać z kilku technik:

- React.memo dla komponentu elementu listy: Jeśli mamy zdefiniowany osobny komponent dla elementu (np. `<ListItem item={...} />`), warto go opakować w React.memo. Wówczas jeśli ten sam item pojawi się ponownie (np. podczas scrollowania w górę/dół) i jego propsy się nie zmieniły, React pominie ponowny render tego komponentu.
- Funkcja `renderItem` z `useCallback`: Często `renderItem` przekazujemy jako literał funkcyjny (`renderItem`={({item}) => ... }). Powoduje to tworzenie nowej funkcji przy każdym renderze rodzica, co może zmuszać listę do myślenia, że coś się zmieniło. Lepiej wydzielić tę funkcję poza JSX lub użyć hooka `useCallback` do jej zdefiniowania, np.:

```
const renderItem = useCallback(({ item }) => <ListItem item={item} />, []);
<FlatList data={data} renderItem={renderItem} ... />
```

Dzięki temu referencja do funkcji będzie stała między renderami (dopóki zależności w tablicy [] się nie zmienią).

- `extraData` prop: FlatList posiada prop `extraData`, przez który możemy przekazać dodatkowe dane wpływające na rendering listy. Służy to głównie, gdy np. komponent listy jest PureComponent i normalnie nie zauważyłby zmian pewnego zewnętrznego stanu. `extraData`={`selectedId`} – przykład z dokumentacji – powoduje, że jeśli zmieni się wartość `selectedId`, lista wie, że ma się prze-renderować. Jeśli nasza lista zależy od jakiegoś stanu spoza data (np. zaznaczony element), powinniśmy podać go w `extraData`.
- `getItemLayout`: Przy bardzo długich listach o ustalonej wysokości elementu można użyć `getItemLayout` – pozwala on obliczyć pozycję scrolla elementu bez renderowania go (do szybkiego scrollowania). To raczej zaawansowane użycie – wspominam dla świadomości.

W kontekście naszego wykładu, najważniejsze, by pamiętać: FlatList/SectionList są wydajne i preferowane do list, klucze muszą być unikalne, a komponenty listy warto memoizować by nie odświeżały się niepotrzebnie. W praktyce, jeśli lista jest prosta, RN sam sobie poradzi. Gdy jednak zauważymy spadki wydajności przy scrollowaniu (lagi), warto przeanalizować czy np. nie przekazujemy w `renderItem` za każdym razem nowej funkcji/obiektu powodując re- render. Często rozwiązaniem jest wspomniany React.memo lub PureComponent dla elementu listy.

Separator i ListHeader: Warto jeszcze wiedzieć, że FlatList ma propsy ułatwiające życie, np. ItemSeparatorComponent (komponent rysujący separator między elementami listy), ListHeaderComponent (nagłówek listy wyświetlany przed elementami – przydatne np. do wstawienia czegoś na górę, jak profil użytkownika przed listą postów) czy ListFooterComponent (analogicznie na dole). Można też łatwo zrobić pull-to-refresh (`onRefresh` + refreshing) i infinite scroll (`onEndReached`). To wszystko czyni FlatList bardzo wszechstronnym narzędziem.

Wydajne listy

FlashList v2 (`@shopify/flash-list`) to wydajna alternatywa FlatList: recyklinguje widoki zamiast tworzyć nowe i sama mierzy elementy (bez `estimatedItemSize`). Przy długich, złożonych listach to dziś pierwszy wybór.

Z włączonym **React Compilerem** komponenty i funkcje są memoizowane automatycznie – ręczne `useCallback` dla `renderItem` i `React.memo` dla elementu przestają być konieczne w większości przypadków (omówimy to na wykładzie 4).

Ikony i czcionki w RN

Estetyka aplikacji mobilnej często wymaga użycia ikon (np. ikona menu, ikony akcji) oraz niestandardowych czcionek. React Native domyślnie obsługuje czcionki systemowe, ale możemy dodawać własne, a ikony realizujemy przez biblioteki.

Ikony: Popularnym sposobem jest skorzystanie z biblioteki `react-native-vector-icons` (autor: Joel Arvidsson; biblioteka jest podzielona na osobne pakiety rodzin ikon, np. `@react-native-vector-icons/ionicons`). Jest to zestaw kilkunastu popularnych rodzin ikon (FontAwesome, Ionicons, Material Icons, itd.) zebranych w jednej bibliotece. W praktyce po zainstalowaniu paczki (lub korzystając z Expo, gdzie jest ona preinstalowana jako `@expo/vector-icons`) możemy zaimportować np. ikonę Ionicons:

```
import Icon from 'react-native-vector-icons/Ionicons';
// ...
<Icon name="home" size={24} color="#000" />
```

Taki komponent `<Icon>` to w istocie specjalny komponent tekstowy, który wyświetla znak z fontu ikonowego odpowiadający nazwie "home". Biblioteka automatycznie zajmuje się załadowaniem odpowiednich fontów (np. pliku .ttf z ikonami Ionicons) i udostępnia je jako komponenty React. W Expo mamy to uproszczone – `@expo/vector-icons` jest dostępne od razu i oparte na `react-native-vector-icons`. Dzięki temu w Expo można od razu robić import { Ionicons } from '`@expo/vector-icons`'; i używać `<Ionicons name="home" size={32} color="green" />`.

Uwaga: Jeśli nie używamy Expo, po instalacji `react-native-vector-icons` trzeba jeszcze zlinkować fonty ikon do projektu natywnego. Moduły natywne są dołączane automatycznie (autolinking), a fonty ikon dodaje się zgodnie z instrukcją pakietu (w Expo nic nie trzeba robić). Na iOS można też używać natywnych SF Symbols przez `expo-symbols`.

Alternatywnie, można używać ikon w formie obrazków PNG/SVG. Istnieje biblioteka react- native-svg oraz z niej pochodne umożliwiające import plików SVG jako komponentów (co bywa wygodne przy własnych ikonach wektorowych). Jednak do ikon typowych, vector-icons są najbardziej wygodne.

Czcionki (fonty): System iOS i Android mają własne domyślne fonty (San Francisco na iOS, Roboto na Androidzie, plus fallbacki). Jeśli projekt graficzny aplikacji przewiduje własną czcionkę, trzeba ją dołączyć. Proces wygląda następująco:

Fonty w Expo – dwa sposoby

Plugin `expo-font` w `app.json` – pliki czcionek są wbudowane w aplikację podczas buildu (dostępne od razu, bez ładowania). Zalecany w buildach produkcyjnych.

Hook `useFonts` (także z pakietów `@expo-google-fonts/*`) – ładowanie w czasie działania; działa w Expo Go. Splash screen trzymamy wtedy do załadowania fontów (`SplashScreen.preventAutoHideAsync()` / `hideAsync()`).

Każda grubość własnego fontu to osobna rodzina (`Inter_700Bold`) – `fontWeight: "bold"` z własną czcionką na Androidzie zwykle nie zadziała.

app.json – czcionki wbudowane w build

```
{
  "expo": {
    "plugins": [
      ["expo-font", { "fonts": ["./assets/fonts/Inter-Regular.ttf", "./assets/fonts/Inter-
Bold.ttf"] }]
    ]
  }
}
```

1. **Dodanie plików czcionek do projektu:** np. pliki .ttf lub .otf. Dla Androida zazwyczaj umieszcza się je w folderze android/app/src/main/assets/fonts/ (trzeba utworzyć, jeśli nie ma). Dla iOS – dodaje do katalogu projektu w Xcode i upewnia w Info.plist, że fonty są wymienione w liście UIApp Fonts.
2. **Linkowanie/Autolink:** autolinking powinien wykryć fonty w katalogu android/assets automatycznie. Czasem trzeba dodać drobną konfigurację w react-native.config.js. W iOS po dodaniu do projektu Xcode i Info.plist, fonty zostaną spakowane przy kompilacji.
3. **Użycie fontu w stylach:** Kiedy font jest już dostępny, używamy stylu `fontFamily`. Np. jeśli font nazywa się "CustomFont-Bold" (nazwa rodziny zdefiniowana wewnątrz pliku fontu), to w stylu dajemy: `fontFamily`: 'CustomFont-Bold'. W Expo sprawa jest jeszcze inna – Expo zaleca korzystanie z biblioteki `expo-font` do dynamicznego wczytywania fontów na starcie aplikacji (trzeba załadować asynchronicznie i dopiero pokazać UI). Natomiast w czystym RN, po linkowaniu, font jest dostępny od razu natywnie.

Przykład: dodajemy dwie czcionki, "OpenSans-Regular.ttf" i "OpenSans-Bold.ttf". Po poprawnym dodaniu, w kodzie RN możemy użyć:

```
<Text style={{ fontFamily: 'OpenSans-Regular' }}>Przykładowy tekst</Text>
<Text style={{ fontFamily: 'OpenSans-Bold', fontSize: 18 }}>Gruby tekst</Text>
```

Jeśli wszystko jest poprawnie zlinkowane, tekst pojawi się odpowiednią czcionką. W razie problemów (np. czcionka nie działa na Androidzie w release) trzeba sprawdzić ścieżki plików i cache (bywa, że wymaga to ponownej kompilacji aplikacji).

Podsumowując:

- **Ikony:** Najłatwiej przez gotowe biblioteki ikon (vector icons). Dają setki gotowych ikon zgodnych ze standardami projektów.
- **Fonty:** Można użyć niestandardowych fontów, trzeba je dołączyć do projektu natywnie (lub za pomocą Expo). Potem używamy `fontFamily` w stylach tekstu, by je zastosować.

SafeAreaView i StatusBar – bezpieczne obszary ekranu

Nowoczesne urządzenia, szczególnie iPhone'y z notchem (wcięciem na kamerę) czy Androidy z wycięciami i okrągłymi rogami, wprowadziły koncepcję bezpiecznego obszaru (safe area). Treści interfejsu powinny mieścić się w obszarze ekranu, który nie jest zasłonięty przez hardware (notche, zaokrąglenia) ani przez systemowe elementy (status bar, paski nawigacji).

React Native wprowadził komponent SafeAreaView. Jego zadaniem jest automatyczne dodanie odpowiednich paddingów od góry i/lub dołu (oraz ewentualnie boków), aby zawartość przez nas włożona nie nachodziła na obszary newralgiczne urządzenia. Od Androida 15 (wymagany docelowy poziom API w Google Play) aplikacje działają w trybie edge-to-edge – treść rysuje się także pod paskiem statusu i paskiem nawigacji, więc bezpieczne obszary trzeba uwzględniać na OBU platformach.

W praktyce użycie jest proste: opakowujemy główny widok ekranu w `<SafeAreaView style={{ flex: 1 }}>` ... `</SafeAreaView>`. Ten flex:1 jest ważny, by SafeAreaView rozciągnął się na cały ekran. Bez tego SafeAreaView dopasuje wysokość do dzieci i może nie zadziałać jak trzeba.

RN vs biblioteka community: Do obsługi bezpiecznych obszarów używamy biblioteki `react-native-safe-area-context` (wbudowany w RN SafeAreaView działa tylko na iOS). Biblioteka dostarcza komponent SafeAreaView i narzędzia do odczytu wymiarów bezpiecznych obszarów. W nowszych projektach (np. wygenerowanych przez Expo lub React Navigation) ta biblioteka jest często już zainstalowana. Jej użycie jest identyczne z perspektywy JSX, ale oferuje większe możliwości konfiguracji (np. edges prop do określenia, które krawędzie brać pod uwagę).

StatusBar: to pasek systemowy na górze ekranu (pokazuje godzinę, baterię itd.). React Native pozwala nam kontrolować jego styl poprzez komponent/API StatusBar. Możemy np. zmienić kolor tła status bara na Androidzie, lub zmienić styl ikon (jasne/ciemne) na iOS. Przykłady:

```
<StatusBar barStyle="light-content" />
```

To ustawi jasny kolor czcionki/status ikon (czyli status bar będzie jasny – np. białe napisy, dobre na ciemnym tle). Z kolei:

```
import { StatusBar } from 'expo-status-bar';

// style: "auto" (ciemne ikony w trybie jasnym, jasne w ciemnym), "light" lub "dark"
<StatusBar style="light" />
```

W trybie edge-to-edge (w Expo włączony na wszystkich wersjach Androida) pasek statusu jest przezroczysty – właściwość `backgroundColor` jest ignorowana. Kolor „pod" paskiem to po prostu tło naszego ekranu, a o odstęp od paska dba `SafeAreaView` / `useSafeAreaInsets`. W Expo zamiast `StatusBar` z `react-native` używamy komponentu z `expo-status-bar`, który ustawia tylko styl ikon.

```
<StatusBar ... /> na górze drzewa komponentów (np. w pliku App.js) lub w każdym ekranie
```

osobno jeśli wymagane są różne ustawienia per ekran. Można także ukryć status bar:

```
<StatusBar hidden={true} />, choć rzadko jest to zalecane, bo użytkownicy raczej chcą widzieć
zegar/baterię.
```

W połączeniu z SafeAreaView: SafeAreaView domyślnie uwzględnia również obszar status bara jako "niebezpieczny" – czyli da padding od góry dokładnie taki jak wysokość status bara. Dzięki temu nasz UI zaczyna się poniżej. Czasami możemy chcieć inaczej (np. obrazek tła ma się rozciągać pod status bar). W bibliotece safe-area-context można wtedy użyć opcji edges={['left','right','bottom']} żeby pominąć górną krawędź.

Podsumowując:

- SafeAreaView – używamy, aby nasz interfejs nie nachodził na notche i zaokrąglenia ekranu. Obowiązkowo na `iPhone` X i nowszych, by aplikacja wyglądała poprawnie.
- StatusBar – komponent do zmiany wyglądu paska statusu. Umożliwia dostosowanie kolorystyki status bara do naszej aplikacji (np. ciemny tekst na jasnym tle lub odwrotnie).

Przykład: ekran profilu z listą i responsywne kafelki

Na koniec połączmy omówione koncepcje w mini-demo architektury ekranu Profilu użytkownika. Załóżmy, że chcemy zbudować ekran, gdzie u góry widnieją informacje profilowe, a poniżej lista elementów (np. postów użytkownika). Dodatkowo, na większych ekranach lista ta ma wyświetlać kafelki w dwóch kolumnach (responsive design).

Struktura ekranu:

- **Bezpieczny kontener:** użyjemy SafeAreaView jako root.
- **Nagłówek profilu:** Wewnątrz SafeAreaView na górze może być zwykły View zawierający:
  - Obrazek avatara (Image),
  - Nazwę użytkownika (Text),
  - Ewentualnie statystyki (posty, obserwujący itp. – to mogą być 3 kolumny z liczbami i podpisami).
  - **Przykład ułożenia:** `flexDirection`: 'row' dla wiersza lub column zależnie od designu.
- **Lista postów:** Poniżej nagłówka umieszczamy listę postów użytkownika. Ponieważ potencjalnie postów może być dużo, użyjemy FlatList. Każdy element listy mógłby być np. miniaturą obrazka posta lub tytułem wpisu.
  - Dla uproszczenia przyjmijmy, że to kafelki obrazków (jak siatka zdjęć na profilu Instagram).
  - Wykorzystamy prop `numColumns` FlatList, by ułożyć kafelki w kolumnach.

Kod szkicowo mógłby wyglądać tak:

```
import { FlatList, Text, View, useWindowDimensions } from 'react-native';
import { SafeAreaView } from 'react-native-safe-area-context';
import { Image } from 'expo-image';
```

- Używa `useWindowDimensions()` do wykrycia szerokości. Jeśli szerokość > 600, ustawiamy 2 kolumny (`numColumns`=2), inaczej 1 kolumna.
- Nagłówek profilu jest prostym centrowanym widokiem z obrazkiem i tekstami.
- FlatList korzysta z `contentContainerStyle` żeby dać padding 8px dookoła całej listy (dzięki temu kafelki nie stykają się z krawędziami ekranu).
- Każdy kafelek (`renderPost`) to View z marginesem, zawierający obrazek i tytuł. Użyliśmy flex: 1 w stylu View, aby kafelki w wierszu rozkładały się równomiernie na całą szerokość. Ważne: przy użyciu `numColumns`, RN sam dzieli dane na rzędy, a nasz element powinien zająć flex:1, by wypełnić kolumnę.
- ListEmptyComponent pokazuje tekst "Brak postów." jeśli tablica data jest pusta.

Responsywne kafelki: Dzięki `numColumns` i `useWindowDimensions`, na telefonie użytkownik zobaczy pojedynczą kolumnę postów (jeden pod drugim pełną szerokością), a na tablecie w poziomie zobaczy po dwie kolumny obok siebie. Można by to rozwinąć dalej – np. dla bardzo szerokich ekranów (telewizor? duży tablet) ustawić 3 kolumny, itp. W razie potrzeby logikę można skomplikować (np. breakpoints: >600px = 2 kolumny, >900px = 3 kolumny itd.).

SafeArea i StatusBar w praktyce: Ponieważ opakowaliśmy wszystko w SafeAreaView, na `iPhone` z notchem avatar i nazwa użytkownika nie będą schowane pod belką statusu – SafeAreaView doda od góry padding. Gdyby tło nagłówka było w innym kolorze, warto nadać ten kolor także SafeAreaView (style SafeAreaView: `backgroundColor`), bo inaczej u góry (za notch) mogłaby być np. biała przestrzeń. StatusBar w tym komponencie nie ustawialiśmy – domyślnie systemowy status bar będzie widoczny. Możemy jednak globalnie w App np. ustawić `<StatusBar barStyle="dark-content" backgroundColor="#fff" />` jeśli chcemy ciemne ikonki na jasnym tle, aby pasowały do białego tła SafeAreaView.

Ten demo-ekran ilustruje wspólne użycie wielu zagadnień z wykładu: podstawowe komponenty (View, Text, Image), Flexbox (centrowanie nagłówka, kafelki w siatce), responsywność (hook wymiarów i warunkowe kolumny), StyleSheet vs utility (tu akurat użyliśmy inline stylów dla zwięzłości, co jest równoważne definiowaniu w StyleSheet), listy (FlatList z optymalizacją kolumn), ikony/czcionki (można by dodać np. ikonkę edycji profilu obok nazwy), SafeAreaView i kontrola StatusBar. Literatura:

Literatura

Wszystkie źródła sprawdzone w październiku 2026 r.

1. https://reactnative.dev/docs/getting-started – Oficjalna dokumentacja React Native.
2. https://docs.expo.dev/ – Oficjalna dokumentacja Expo.
3. https://docs.expo.dev/guides/new-architecture/ – Kluczowa dokumentacja Expo wyjaśniająca Nową Architekturę (Fabric).
4. https://docs.expo.dev/router/introduction/ – Dokumentacja Expo Router, nowoczesnego standardu nawigacji (wspomnianego w wykładzie).
5. https://docs.expo.dev/guides/environment-variables/ – Szczegółowy opis zarządzania zmiennymi środowiskowymi (.env) w Expo.
6. https://reactnavigation.org/ – Dokumentacja React Navigation (alternatywnej, popularnej biblioteki do nawigacji).
7. https://nodejs.org/ – Strona oficjalna Node.js.
8. https://code.visualstudio.com/ – Strona oficjalna Visual Studio Code.
9. https://reactnative.dev/docs/components-and-apis – komponenty i API React Native.
10. https://reactnative.dev/docs/flexbox oraz https://reactnative.dev/docs/view-style-props – Flexbox i style widoków, m.in. `boxShadow`, `filter`.
11. https://docs.expo.dev/versions/latest/sdk/image/ – `expo-image`.
12. https://docs.expo.dev/develop/user-interface/fonts/ – czcionki w Expo (plugin i `useFonts`).
13. https://docs.expo.dev/develop/user-interface/safe-areas/ – bezpieczne obszary i tryb edge-to-edge.
14. https://shopify.github.io/flash-list/ – FlashList v2.
15. https://www.nativewind.dev/ – NativeWind 4.