POLITECHNIKA ŚWIĘTOKRZYSKA

Aplikacje mobilne – wykład 5

Formularze i walidacja w React Native

Mateusz Pawełkiewicz

1.10.2025

Wprowadzenie

Formularze stanowią kluczowy element większości aplikacji mobilnych – pozwalają użytkownikom wprowadzać dane, logować się, rejestrować konta czy składać zamówienia. Implementacja formularzy w **React Native** wymaga uwzględnienia specyfiki platform mobilnych, takich jak obsługa klawiatury ekranowej czy zapewnienie odpowiedniej **walidacji** wprowadzanych danych. W 2025 roku dysponujemy nowoczesnymi bibliotekami i wzorcami, które upraszczają tworzenie formularzy i poprawiają doświadczenie użytkownika (UX). W niniejszym wykładzie omówimy szczegółowo:

- Podstawowe komponenty formularza w RN, m.in. TextInput, obsługę zdarzeń fokus/blur oraz sposoby unikania zasłonięcia pól przez klawiaturę (np. za pomocą KeyboardAvoidingView).
- Wykorzystanie biblioteki **react-hook-form** do łatwego zarządzania stanem formularza, w tym komponent Controller, obsługę błędów i zdarzenia onSubmit.
- Walidację schematyczną przy użyciu bibliotek **Zod** lub **Yup** – porównamy je, pokażemy integrację z react-hook-form i sposób generowania komunikatów błędów.
- Najlepsze praktyki **UX formularzy mobilnych**: użycie pól wyboru (pickerów), date-pickerów, masek na pola tekstowe (np. numer telefonu), przełączanie focusu między polami, obsługa przycisku submit oraz kwestii dostępności (accessibility).
- Kompleksowy **przykład (demo)**: zaimplementujemy od podstaw formularz rejestracji/logowania z wykorzystaniem react-hook-form i Zod, z pełną walidacją i poprawnym UX mobilnym.

Komponenty formularza w React Native

React Native dostarcza podstawowe komponenty do budowy formularzy, z których najważniejszym jest **TextInput** – służący do wprowadzania tekstu przez użytkownika. Oprócz niego często wykorzystujemy przełączniki (Switch), przyciski (Button lub dotykowe komponenty z rodziny Touchable*), a także komponenty z bibliotek zewnętrznych (np. pickery dat czy list rozwijanych). W tej części skupimy się na TextInput oraz powiązanych zagadnieniach: zdarzeniach focus/blur i obsłudze klawiatury ekranowej.

TextInput – podstawy i zdarzenia focus/blur

TextInput jest bazowym komponentem RN do wpisywania tekstu. Działa podobnie jak <input> w React (web), lecz posiada własne właściwości i metody. Najważniejsze cechy:

- Możemy nasłuchiwać zdarzeń **onChangeText** (każda zmiana tekstu), **onFocus** (wejście w pole) i **onBlur** (wyjście z pola). Umożliwia to np. dynamiczne zmiany stylu pola, walidację po opuszczeniu, itp.
- Komponent udostępnia metody **.focus()** i **.blur()** do programowego ustawiania lub zabierania fokusu. Dokumentacja RN wskazuje, że *TextInput posiada metody .focus() i .blur() umożliwiające odpowiednio ustawienie fokusu na polu lub jego usunięcie*. Dzięki nim możemy np. automatycznie przenieść fokus na kolejne pole formularza.

- Wiele właściwości pozwala dostosować klawiaturę ekranową: np. keyboardType (określa typ klawiatury, np. numeryczna, email), secureTextEntry (tryb hasła), returnKeyType (tekst przycisku return/enter na klawiaturze – np. „Next” lub „Done”), czy autoCapitalize (auto-kapitalizacja liter). Odpowiednie ustawienie tych opcji poprawia UX – np. dla pola email ustawiamy keyboardType="email-address" i autoCapitalize="none".
- Na iOS możemy użyć też textContentType i autoComplete aby skorzystać z mechanizmów autofill systemu (np. textContentType="emailAddress" podpowiada zapamiętane e-maile użytkownika). Warto to wykorzystywać wrażliwie, by ułatwić użytkownikom wypełnianie formularzy.

Zdarzenia fokus i blur: React Native pozwala reagować na moment, gdy pole zostaje aktywowane (fokus) lub opuszczone (utrata fokusu). Typowe zastosowania:

- Zmiana obramowania lub tła pola, aby wyróżnić aktualnie edytowane pole.
- Walidacja po opuszczeniu pola – np. gdy użytkownik wyjdzie z pola, sprawdzamy czy wartość jest poprawna i ewentualnie wyświetlamy komunikat błędu.
- Śledzenie „odwiedzonych” pól (tzw. touched fields) – to ważne przy wyświetlaniu błędów dopiero po tym, jak użytkownik próbował coś wpisać.

W praktyce do obsługi tych zdarzeń przypisujemy funkcje do propsów onFocus i onBlur. Jeśli korzystamy z biblioteki formularzy (jak react-hook-form), to często nie musimy ręcznie obsługiwać touched – biblioteka może oznaczać pole jako „touched” automatycznie przy blur.

Unikanie zasłonięcia pól przez klawiaturę

Na urządzeniach mobilnych klawiatura ekranowa potrafi zająć sporą część ekranu i **zasłonić pola tekstowe** znajdujące się niżej. Bez odpowiednich działań użytkownik może nie widzieć, co wpisuje. Istnieje kilka technik radzenia sobie z tym problemem:

- **KeyboardAvoidingView:** Jest to wbudowany komponent RN, który automatycznie **dostosowuje wysokość lub pozycję widoku** rodzica, gdy pojawia się klawiatura, tak aby aktywne pole pozostało widoczne. Najczęściej opakowuje się cały ekran (lub sekcję z formularzem) w <KeyboardAvoidingView behavior="padding" /> lub "position" – dla iOS zwykle sprawdza się behavior="padding", dla Androida czasem lepszy jest "height". Należy także ustawić keyboardVerticalOffset jeśli używamy np. headera – pozwala to skorygować pozycję o wysokość nagłówka. KeyboardAvoidingView to proste rozwiązanie, ale czasem bywa niewystarczające (np. przy bardzo długich formularzach).
- **ScrollView z opcją przewijania:** Opakowanie formularza w ScrollView pozwala przewijać zawartość, dzięki czemu użytkownik może ręcznie przesunąć ekran, by zobaczyć pola spod klawiatury. Dobrą praktyką jest ustawienie keyboardShouldPersistTaps="handled" lub "always" – dzięki temu dotknięcie obszaru ScrollView poza polem zamknie klawiaturę (jeśli żaden inny element nie obsłuży tego dotyku). Rozwiązanie to zapobiega sytuacji, gdzie naciskanie na tło **tylko** chowa klawiaturę (zamiast np. otworzyć inny przycisk). Wspomniany parametr sprawia, że

tapnięcia są **przekazywane dalej**, skutkując zamknięciem klawiatury tylko gdy tapnięto w pusty obszar.

- **Dismiss na tapnięcie w tło:** Możemy ręcznie obsłużyć ukrycie klawiatury, gdy użytkownik tapnie poza polem. RN udostępnia moduł Keyboard z metodą dismiss(), która chowa klawiaturę. Typowy wzorzec to opakowanie całego ekranu w TouchableWithoutFeedback lub Pressable, którego onPress wywołuje Keyboard.dismiss(). W ten sposób kliknięcie w dowolne miejsce tła zamyka klawiaturę.

```
import { Keyboard, TouchableWithoutFeedback } from 'react-native';

const DismissKeyboardView: React.FC = ({ children }) => (
 <TouchableWithoutFeedback onPress={Keyboard.dismiss} accessible={false}>
  {children}
 </TouchableWithoutFeedback>
);
```

Następnie używamy <DismissKeyboardView> jako kontenera najwyższego poziomu ekranu formularza. Ważny szczegół: ustawienie accessible={false} na tym wrapperze sprawia, że *element ten będzie ignorowany przez czytniki ekranu (VoiceOver/TalkBack) i nie zakłóci dostępu do pól wejściowych*. Gdybyśmy o tym zapomnieli, nasz dotykowy wrapper mógłby zostać potraktowany jako element interfejsu przez mechanizmy dostępności, utrudniając korzystanie z formularza osobom niewidomym.

Podsumowanie: Najlepsze efekty daje **kombinacja powyższych podejść**: np. cały ekran objęty KeyboardAvoidingView + ScrollView z możliwością tapnięcia w tło + mechanizm automatycznego dismiss. W praktyce można utworzyć komponent wyższego rzędu (HOC) lub po prostu zagnieździć te elementy: KeyboardAvoidingView -> ScrollView (z keyboardShouldPersistTaps) -> nasza zawartość formularza wewnątrz TouchableWithoutFeedback. Taki układ gwarantuje, że pola nie zostaną zasłonięte, można przewijać długie formularze, a kliknięcie obok pola schowa klawiaturę.

Przełączanie fokusu między polami

Na mobilnym UX istotne jest ułatwienie użytkownikowi szybkiego przechodzenia przez formularz. Gdy użytkownik wypełni jedno pole i naciśnie przycisk „Dalej” (Next) na klawiaturze, chcemy automatycznie przenieść fokus do kolejnego pola. W RN realizujemy to następująco:

- Każdy TextInput (poza ostatnim) ustawiamy returnKeyType="next". Ostatniemu polu (np. hasło przy logowaniu) dajemy returnKeyType="done" lub go – tak by użytkownik widział, że kończy wprowadzanie.
- Na komponentach TextInput nasłuchujemy onSubmitEditing – zdarzenie wywoływane po naciśnięciu przycisku „Enter/Next” na klawiaturze. Dla pierwszego pola onSubmitEditing powinno wywołać .focus() na refie do drugiego pola, dla drugiego – fokus na trzecie, itd. W ten sposób użytkownik może przechodzić przez pola bez dotykania ekranu.
- Implementacja: korzystamy z referencji (useRef) do kolejnych TextInputów. Przykład dla dwóch pól:

```
const passwordRef = useRef<TextInput>(null);

<TextInput
  placeholder="Email"
  returnKeyType="next"
  onSubmitEditing={() => passwordRef.current?.focus()}
/>
<TextInput
  ref={passwordRef}
  placeholder="Hasło"
  returnKeyType="done"
  onSubmitEditing={handleSubmit(onSubmit)}  // dla ostatniego pola wywołujemy submit
/>
```

W powyższym kodzie, gdy użytkownik wpisując email naciśnie „Next”, wywołujemy `passwordRef.current.focus()`, przenosząc kursor do pola hasła. Gdy jest w polu hasła i naciśnie „Done”, wywołujemy `handleSubmit(onSubmit)` (metoda z react-hook-form) aby wysłać formularz. Taka nawigacja znacznie poprawia ergonomię formularza.

Podsumowanie sekcji

Budując formularz w RN musimy zwrócić uwagę nie tylko na same pola, ale także na **otoczenie**: klawiaturę i nawigację między polami. Stosując `KeyboardAvoidingView`, przewijanie oraz mechanizm chowania klawiatury na tapnięcie zapewnimy, że użytkownik zawsze widzi aktywne pole. Z kolei obsługa przycisku Next/Done na klawiaturze umożliwia szybkie uzupełnianie formularza bez odrywania rąk od klawiatury. W kolejnych częściach przejdziemy do biblioteki ułatwiającej zarządzanie stanem formularza i walidacją.

React Hook Form – zarządzanie stanem formularzy

React Hook Form (RHF) to obecnie jedna z najpopularniejszych bibliotek do obsługi formularzy w ekosystemie React (w tym React Native). Jej pojawienie się zrewolucjonizowało podejście do formularzy dzięki skupieniu na wydajności i prostocie.

Dlaczego React Hook Form?

Tradycyjne podejście do formularzy w React (kontrolowanie wartości inputów przez state i obsługa na onChange) bywa nieefektywne – każda zmiana powoduje render komponentu, co w przypadku wielu pól jest kosztowne. **React Hook Form został zaprojektowany z myślą o wykorzystaniu niekontrolowanych komponentów i referencji, minimalizując liczbę renderów** potrzebnych do obsługi formularza. Według dokumentacji: *react-hook-form buduje formularze w oparciu o niekontrolowane inputy, dążąc do maksymalnej wydajności i minimalnej liczby ponownych renderowań*. Dzięki temu idealnie nadaje się do React Native, gdzie nadmierne renderowanie pól (zwłaszcza tych z animacjami lub formatowaniem) może powodować widoczne opóźnienia.

Kilka kluczowych zalet RHF:

- **Wydajność:** RHF nie trzyma wartości każdego pola w stanie komponentu React, tylko polega na natywnych elementach (TextInput) i referencjach. Aktualizuje stan Reacta tylko wtedy, gdy jest to konieczne (np. wystąpi błąd walidacji). To oznacza *minimalne ponowne renderowanie i lepszą wydajność w porównaniu z podejściem kontrolowanym*.
- **Prostota API:** Biblioteka udostępnia hook useForm, który dostarcza narzędzia do obsługi formularza (np. register, handleSubmit, errors). Integruje się z natywnymi elementami formularza w React i RN, nie wymuszając użycia specjalnych komponentów formularza (jak to czyni Formik).
- **Integracja z walidatorami:** RHF łatwo łączy się z zewnętrznymi bibliotekami walidacji schematów (Yup, Zod itp.) poprzez tzw. *resolvers*. Możemy więc definiować reguły walidacji w jednym miejscu i mieć zarówno walidację jak i typowanie danych.
- **Mniejsze gabaryty i zależności:** RHF jest dość lekką biblioteką, bez dużych zależności, co ma znaczenie w aplikacjach mobilnych (rozmiar pakietu).
- **Społeczność i wsparcie:** Stała się standardem de-facto w nowych projektach, stąd dużo materiałów, przykładów i aktywne wsparcie.

Podstawy użycia react-hook-form w RN

Aby skorzystać z RHF, należy zainstalować pakiet:

```
npm install react-hook-form @hookform/resolvers
```

Uwaga: @hookform/resolvers to dodatkowy pakiet, który zawiera tzw. resolvery do integracji z bibliotekami walidacji (Yup, Zod itp.). Wrócimy do niego w sekcji o walidacji schematów.

Najważniejszy hook to **useForm** – wywołujemy go wewnątrz komponentu, który zawiera formularz. Przykład użycia w komponencie funkcyjnym:

```
import { useForm } from 'react-hook-form';

type FormData = {
  email: string;
  password: string;
};

const { control, handleSubmit, formState: { errors } } = useForm<FormData>();
```

Tutaj wywołaliśmy useForm<FormData>() przekazując opcjonalnie generyczny typ formularza (dzięki temu errors itp. będą typowane). Otrzymujemy obiekt z kilkoma właściwościami:

- **control:** obiekt kontrolny formularza, potrzebny m.in. do powiązania z komponentem Controller.
- **handleSubmit:** funkcja służąca do obsługi wysłania formularza. Używamy jej, by owrapować naszą funkcję onSubmit – zapewnia walidację i przekazuje nam zebrane dane jeśli wszystko jest OK.

- **formState: { errors }:** obiekt zawierający ewentualne błędy walidacji dla pól (właściwości odpowiadają nazwom pól). Jeśli dane pole ma błąd, errors.fieldName będzie zawierać np. message z komunikatem.
- (Opcjonalnie register – jednak w kontekście React Native zwykle zamiast manualnego rejestrowania inputów korzysta się z Controller, o czym dalej).

Standardowo w RN nie mamy elementu <form> jak w web, więc nie ma zdarzenia onSubmit formularza – dlatego używamy handleSubmit. W praktyce często robimy coś takiego przy przycisku **Submit**:

```
<TouchableOpacity onPress={handleSubmit(onSubmit)}>
  <Text>Wyślij</Text>
</TouchableOpacity>
```

Wywołanie handleSubmit(onSubmit) zwraca funkcję, która po kliknięciu: przeprowadzi walidację wszystkich pól, a jeśli przejdzie pomyślnie, wywoła nasz onSubmit(data) z obiektem danych. Dzięki temu mamy pewność, że onSubmit dostaje **tylko poprawne dane** (w przeciwnym razie onSubmit się nie wykona, a błędy zostaną zapisane w errors).

Użycie komponentu Controller w React Native

W React Hook Form na web (np. z <input>), często stosuje się atrybut ref lub register bezpośrednio na polu, np. <input {...register('email')}>. Jednak w przypadku React Native i TextInput nie mamy łatwego sposobu na zarejestrowanie go poprzez ref (komponent nie jest czysto HTML-owy). Zamiast tego biblioteka RHF dostarcza komponent **<Controller>**, który pełni rolę „mostu" między naszą logiką formularza a komponentem interfejsu.

Controller przyjmuje kilka propsów:

- name – nazwa pola (musi odpowiadać kluczowi w obiekcie danych formularza).
- control – przekazujemy tutaj obiekt control uzyskany z useForm().
- rules (opcjonalnie) – obiekt z podstawowymi regułami walidacji (jeśli nie korzystamy z resolvera schematów). Możemy tu ustawić np. required: true albo bardziej szczegółowo: maxLength: { value: 100, message: "Max 100 znaków" }.
- render – funkcja renderująca, która powinna zwrócić nasz właściwy komponent inputu. Ta funkcja otrzymuje pewne parametry (rozpakowywane często jako { field: { onChange, onBlur, value } }), które musimy przekazać do naszego komponentu wejściowego.

Aby lepiej zrozumieć, spójrzmy na fragment kodu z użyciem Controller dla pola tekstowego:

```
<Controller
 control={control}
 name="email"
 rules={{
  required: "Email jest wymagany",
  pattern: { value: /\S+@\S+\.\S+/, message: "Nieprawidłowy email" }
 }}
 render={({ field: { onChange, onBlur, value } }) => (
  <View style={styles.inputGroup}>
```

```
    <TextInput
      placeholder="E-mail"
      keyboardType="email-address"
      autoCapitalize="none"
      value={value}
      onChangeText={onChange}
      onBlur={onBlur}
      style={styles.input}
    />
    {errors.email && (
      <Text style={styles.errorText}>{errors.email.message}</Text>
    )}
  </View>
 )}
/>
```

Wyjaśnienie:

- Przekazujemy control z naszego formularza oraz name="email" – dzięki temu Controller „wie", z którym polem pracuje.
- W rules zdefiniowaliśmy, że email jest wymagany (komunikat błędu, jeśli pusty) i powinien pasować do prostego regexu adresu email (inaczej pokaże komunikat nieprawidłowego formatu). **Uwaga:** Powyższe podejście z rules pokazuje wbudowaną walidację RHF. W dalszej części zobaczymy, jak można użyć zamiast tego walidacji schematem (Yup/Zod).
- Prop render to funkcja, która dostaje obiekt zawierający m.in. field: { onChange, onBlur, value, name }. Destrukturyzujemy to i używamy:
    - onChange przypisujemy do onChangeText TextInputa – dzięki temu każda zmiana tekstu uaktualni wartość w stanie formularza. Ważne: *nie wywołujemy tu setState własnego – RHF sam zarządza wartością.*
    - onBlur przypisujemy do onBlur TextInputa – w momencie opuszczenia pola RHF oznaczy to pole jako „dotknięte" i ewentualnie uruchomi walidację (np. pokaże błąd "required" jeśli puste).
    - value przypisujemy do value komponentu – wartość kontrolowana jest przez RHF.
- Następnie w JSX obok pola warunkowo renderujemy komunikat błędu jeśli errors.email istnieje. errors.email.message będzie zawierało tekst błędu przekazany w rules (lub z resolvera schematu).

Dzięki Controllerowi możemy korzystać z niekontrolowanych komponentów RN, a jednocześnie podpiąć je pod kontrolę biblioteki formularza.

Kilka uwag praktycznych:

- Nie musimy używać Controller dla **każdego** elementu. Jeśli np. mamy prosty przełącznik Switch lub suwak, możemy często zarejestrować go inaczej. Jednak w większości przypadków w RN jest to najwygodniejsza metoda.
- Istnieje również hook useController dający podobny efekt w komponencie bez JSX, ale zazwyczaj prostszy jest komponent <Controller> w JSX.

- rules obsługuje podstawowe walidacje – jednak przy bardziej złożonych warunkach lub wielu polach zależnych od siebie, lepiej użyć walidatora schematów.
- Gdy używamy walidacji schematem (resolver), nie trzeba duplikować reguł w rules – możemy je wtedy pominąć lub użyć do drobnych dodatkowych spraw (np. rules={{ required: true }} tylko do oznaczenia obowiązkowości – choć i to może być w schemacie).

Obsługa błędów i komunikatów

RHF udostępnia w formState.errors informacje o błędach. Każdy wpis errors[namePola] zawiera m.in. message (jeśli zdefiniowaliśmy komunikat w rules lub dostarcza go walidator schematu), type (typ błędu, np. "required", "maxLength" itp.), a także inne informacje (np. actual i required przy walidacji długości).

Najprostsze podejście to wyświetlać błąd pod polem, co zostało pokazane wyżej. Kilka dobrych praktyk:

- Komunikaty błędów powinny być krótkie, zrozumiałe i pomagać poprawić dane (np. „Hasło musi mieć co najmniej 8 znaków").
- Warto stylować błędy wyróżniającym kolorem (czerwony) i np. mniejszą czcionką.
- Można też oznaczać pola z błędem wizualnie (np. czerwonym obramowaniem). W tym celu możemy dodać do stylu TextInputa warunek: style={[styles.input, errors.password && styles.inputError]} – wcześniej definiując w styles.inputError np. borderColor: 'red'.

Jeśli chodzi o **UX wyświetlania błędów** – często lepiej pokazywać błędy dopiero gdy użytkownik zakończył interakcję z polem (onBlur) lub próbował wysłać formularz. RHF wspiera to – domyślnie handleSubmit oznacza wszystkie pola jako touched przy próbie wysłania, więc błędy się pokażą. Możemy też zmienić domyślne ustawienia, np. useForm({ mode: 'onBlur' }) spowoduje, że walidacja będzie wykonywana po opuszczeniu pola, a mode: 'onChange' – na bieżąco w trakcie pisania (co czasem bywa zbyt agresywne). Domyślne mode to 'onSubmit' (walidacja głównie przy submit, ale błędy i tak możemy wyświetlać wcześniej jeśli pole jest touched).

Podsumowując pracę z RHF: Nasz komponent formularza w RN będzie zawierał:

- Inicjalizację useForm z odpowiednim resolver (jeśli używamy Yup/Zod) lub z defaultValues (jeśli potrzebujemy).
- Kilka <Controller> odpowiadających polom, wewnątrz których znajdują się konkretne <TextInput> lub inne elementy (Picker, Switch etc.) powiązane przez onChange/value.
- Elementy tekstowe wyświetlające błędy pod polami.
- Przycisk Submit (TouchableOpacity/Button) wywołujący handleSubmit(onSubmit).
- Ewentualnie dodatkowe przyciski, np. „Reset" (który może użyć reset() z RHF) lub nawigacja (np. link „Masz już konto? Zaloguj" – choć to poza samym formularzem).

Przejdźmy teraz do kluczowej kwestii walidacji – zwłaszcza z użyciem zewnętrznych bibliotek do definiowania reguł.

Schematyczna walidacja formularzy (Zod vs Yup)

Walidacja schematyczna polega na zdefiniowaniu struktury danych (schematu) i reguł dla poszczególnych pól, a następnie wykorzystaniu tego schematu do zweryfikowania poprawności danych. Podejście to ma kilka zalet:

- **Centralizacja reguł:** Wszystkie zasady walidacji zebrane są w jednym miejscu (schemacie), a nie porozrzucane po komponentach.
- **Możliwość ponownego użycia:** Ten sam schemat można zastosować po stronie frontendu (dla wstępnej walidacji) i backendu (dla ostatecznej walidacji danych np. przed zapisaniem do bazy), co redukuje duplikację.
- **Lepsza integracja z TypeScript:** Biblioteki takie jak **Zod** pozwalają automatycznie wyprowadzić typ TypeScript na podstawie schematu walidacji. Dzięki temu nasze dane formularza mogą mieć typy ściśle zgodne z regułami walidacji – co zwiększa bezpieczeństwo i wygodę pracy.

W ekosystemie React dominują dwie biblioteki do walidacji schematów: **Yup** (długo popularna, dobrze zintegrowana z Formikiem) i **Zod** (stosunkowo nowsza, zyskująca na popularności ze względu na ścisłą integrację z TypeScript). Przyjrzyjmy się krótko obu:

- **Yup:** Walidator wzorowany na bibliotece Joi (znanej z Node.js). Pozwala deklaratywnie tworzyć schematy za pomocą łańcuchowania metod (np. `yup.string().email().required()` dla pola email). Ma wbudowane walidacje typów prostych, stringów, liczb, tablic itp., obsługuje zależności między polami (`ref` do innego pola, metoda `when` do warunkowej walidacji).
- **Zod:** Biblioteka od podstaw zaprojektowana pod TypeScript. Tworzenie schematu polega na wywoływaniu funkcji (np. `z.string().email()`), bardzo podobnie do Yup, ale *każdy schemat Zod jest jednocześnie typem TypeScript* – możemy użyć `z.infer<typeof schema>` by uzyskać typ. Zod wymusza parsing danych (metoda `.parse()` albo bezpieczna `.safeParse()`), integruje walidację i parse w jedno (co zmniejsza ryzyko niespójności typów).

Porównanie: Obie biblioteki osiągają podobne cele, składnia jest zbliżona. W praktyce:

- Yup jest „starszy", więc wiele przykładów i projektów (zwłaszcza z Formikiem) go używa. Ma dojrzałe API, ale integracja z TS jest nieco sztukowana (Yup potrafi generować typy, ale bywa to zawodne przy bardziej złożonych schematach).
- Zod jest „nowszy" i **TS-first** – każdy schemat to źródło prawdy dla walidacji *i* typów. Zod nie pozwoli np. użyć wartości spoza zdefiniowanego enumu bez zgłoszenia błędu typów (przy użyciu infer). Posiada też lepsze mechanizmy walidacji złożonej (refinements) i łatwiej nim walidować zagnieżdżone struktury oraz pola zależne logicznie.

Integracja schematów z react-hook-form (resolvers)

React Hook Form udostępnia wspomniany pakiet @hookform/resolvers, który zawiera gotowe integracje z różnymi bibliotekami walidacji (Yup, Zod, Joi, AJV itp.). Dzięki temu możemy dodać do useForm opcję resolver, a RHF zajmie się resztą – tj. przy wywołaniu handleSubmit automatycznie zweryfikuje dane schematem i wypełni obiekt errors ewentualnymi błędami.

Przykład z Zod: Załóżmy, że mamy schemat Zod:

```
import { z } from 'zod';

const LoginSchema = z.object({
  email: z.string().email("Nieprawidłowy format email").nonempty("Email jest wymagany"),
  password: z.string().min(6, "Hasło musi mieć min. 6 znaków").nonempty("Hasło jest wymagane"),
});
```

Tutaj zdefiniowaliśmy, że email musi być niepusty i w formacie email, a hasło niepuste i min. 6 znaków. Możemy teraz zrobić:

```
import { useForm } from 'react-hook-form';
import { zodResolver } from '@hookform/resolvers/zod';

type LoginData = z.infer<typeof LoginSchema>; // automatyczny typ danych na podstawie schematu

const { control, handleSubmit, formState: { errors } } = useForm<LoginData>({
  resolver: zodResolver(LoginSchema)
});
```

To jedno przypisanie resolver: zodResolver(LoginSchema) sprawia, że gdy użytkownik będzie wysyłał formularz, RHF:

- pobierze aktualne wartości wszystkich pól,
- przekaże je do walidatora Zod,
- otrzyma wynik walidacji – jeżeli są błędy, automatycznie wypełni errors i NIE wywoła onSubmit; jeżeli brak błędów, pozwoli wywołać onSubmit z danymi.

Warto wspomnieć, że resolvery mogą także wykonywać pewne transformacje – np. Zod może sparsować dane (np. zamienić string na number jeśli tak zdefiniujemy). RHF domyślnie używa mode: 'onSubmit' przy użyciu resolvera, ale można to zmienić (np. mode: 'onBlur' aby walidować każde pole po wyjściu).

Komunikaty błędów: W schemacie powyżej zauważmy, że przy każdym ograniczeniu przekazaliśmy komunikat (np. "Nieprawidłowy format email"). Zod pozwala w metodach jak .email() czy .min() od razu podać wiadomość błędu, która będzie zwrócona. Yup ma podobny mechanizm – np. .min(6, "Hasło za krótkie"). Dobrą praktyką jest zdefiniowanie wszystkich tekstów błędów w jednym miejscu (schemacie), co ułatwia ewentualne tłumaczenia lub modyfikacje. RHF poprzez resolver automatycznie ustawi te komunikaty w errors[field].message, więc w komponencie możemy je wyświetlać tak jak wcześniej.

Walidacja między polami: Często potrzebujemy sprawdzić zależności, np. **potwierdzenie hasła** musi się zgadzać z hasłem. Jak to zrobić? Można wykorzystać mechanizmy schematu:

- W Yup można użyć .oneOf([yup.ref('password')], "Hasła muszą być takie same") dla pola confirmPassword, albo użyć .test() z dostępem do całego obiektu (this.parent).
- W Zod możemy użyć .superRefine() lub .refine() na obiekcie. Przykład:

```
const RegisterSchema = z.object({
  password: z.string().min(6, "Min 6 znaków"),
  confirm: z.string()
}).refine(data => data.confirm === data.password, {
  path: ['confirm'],  // wskazujemy, że błąd dotyczy pola confirm
  message: "Hasła nie są zgodne"
});
```

Powyższe wywołanie .refine doda własną regułę walidacyjną na poziomie całego obiektu: jeżeli warunek (confirm === password) nie będzie spełniony, to wygeneruje błąd przypisany do pola confirm z podanym komunikatem. Zod pozwala też w superRefine mieć dostęp do kontekstu walidacji, w którym można dodać wiele błędów dla różnych pól naraz (np. sprawdzanie spójności daty „od" i „do" itp.).

Wynik walidacji a TypeScript: W przypadku Zod, użycie z.infer<typeof schema> zapewnia, że obiekt danych LoginData dokładnie odpowiada schematowi (np. email jest stringiem, hasło stringiem, itp.). W Yup, możemy użyć InferType<typeof schema> z pakietu yup (Yup ma yup.InferType), ale jest pewne ryzyko, że definicja typów nie odda wszystkich złożonych zależności. Zod gwarantuje, że *jeśli walidacja przejdzie, dane wyjściowe spełniają podany typ*, bo samo walidowanie odbywa się metodą .parse która rzuci wyjątek przy niezgodności typów. W Yup walidacja jest oddzielona od mechanizmu typowania (bardziej opiera się na zaufaniu programisty, że dane pasują do zadanego interfejsu). Jak wskazuje analiza, Zod eliminuje ryzyko niespójności między typami a walidacją – schemat jest jedynym źródłem prawdy dla obu.

Którą bibliotekę wybrać?

W 2025 roku wybór często pada na **Zod** w nowych projektach TS, natomiast **Yup** w istniejących projektach może pozostać z uwagi na dojrzałość i przyzwyczajenie. Obie spełnią swoje zadanie. Z perspektywy React Native i wydajności nie ma dużej różnicy – walidacja i tak odbywa się w JavaScripcie (choć można tu dodać, że Zod bywa minimalnie szybszy przy prostszych schematach, a Yup nieco szybszy przy bardzo złożonych – ale nie jest to zauważalna różnica).

Jeśli zależy nam na pełnym wykorzystaniu TypeScript – Zod daje przewagę. Jeśli mamy gotowe schematy w Yup (np. z wcześniejszego projektu) i działają – nie ma konieczności przepisywania na Zod na siłę.

Integracja z RHF jest świetna dla obu: wystarczy wybrać odpowiedni resolver (yupResolver lub zodResolver). Przykład integracji z Yup z kodu : użyto resolver: yupResolver(loginSchema) przy inicjacji useForm, co analogicznie jak w naszym przykładzie z Zod włącza walidację schematu przy submit.

Na zakończenie tej sekcji warto podkreślić: *Niezależnie czy użyjemy Yup czy Zod, centralne definiowanie walidacji bardzo ułatwia rozwój i utrzymanie formularzy.* Dzięki temu unikamy rozproszenia logiki po wielu miejscach i możemy łatwo modyfikować reguły (np. zmiana minimalnej długości hasła w jednym miejscu).

UX formularzy mobilnych – dobre praktyki

Oprócz poprawnej funkcjonalności i walidacji formularza, musimy zadbać o **doświadczenie użytkownika (UX)**. Użytkownicy mobilni oczekują, że formularz będzie wygodny, intuicyjny oraz dostosowany do urządzenia (np. użyje odpowiedniej klawiatury dla pola numeru telefonu). Omówmy kluczowe aspekty UX formularzy na platformach mobilnych:

Przyjazne komponenty wejściowe

Select / Picker (listy wyboru): Czasem pola formularza powinny pozwolić użytkownikowi wybrać jedną z predefiniowanych opcji (np. kraj, płeć, przedział wiekowy). Na web zrobilibyśmy <select>, a w RN? RN do wersji 0.65 posiadał wbudowany Picker, obecnie jest on dostępny jako osobny pakiet **@react-native-picker/picker**. Picker w iOS prezentuje się jako klasyczny obracany wybierak na dole ekranu, a na Androidzie jako rozwijana lista lub modal. Dobre praktyki:

- Dla małych list (kilka opcji) można ewentualnie użyć ActionSheet (iOS) lub Modal z własną listą opcji – ale **najlepiej użyć natywnego pickera** dla spójności doświadczenia użytkownika.
- Integracja pickera z RHF: można opakować go Controllerem podobnie jak TextInput. Ponieważ picker nie ma onChangeText, ale np. onValueChange, musimy odpowiednio przekazać: onChange -> onValueChange, value -> selectedValue.
- Upewnij się, że dla pickera ustawiona jest wartość domyślna lub placeholder typu „Wybierz opcję…", aby użytkownik wiedział, że trzeba coś wybrać. W walidacji możemy traktować brak wyboru jako błąd wymagany.

DatePicker (wybór daty/czasu): Wiele formularzy wymaga dat (np. data urodzenia, termin rezerwacji). W mobilnym UX nie zmuszamy użytkownika do wpisywania daty ręcznie – zamiast tego używamy natywnych kontrolek wyboru daty/czasu. W RN standardem jest biblioteka **@react-native-community/datetimepicker**, która udostępnia natywne okna wyboru daty i czasu (iOS: rolki lub kalendarz, Android: dialog). Jak to włączyć:

- Można użyć trybu *inline* (embedding) na iOS, jednak częściej pojawia się jako modal/dialog po interakcji. Zazwyczaj robimy tak: w miejscu gdzie data ma być wybrana dajemy tekstowe pole (np. TextInput lub po prostu TouchableOpacity wyświetlające bieżący wybór), i po kliknięciu ustawiamy stan showDatePicker = true, który powoduje wyświetlenie komponentu DateTimePicker. Po wyborze daty wywoła on onChange – tam wyłączamy widoczność i ustawiamy wybraną datę w stanie formularza.
- Integracja z RHF: możemy traktować pole daty jako zwykłą wartość. Najprościej – przechowywać datę w stanie komponentu (useState) i w onChange pickera wywoływać setValue('birthDate', selectedDate) z RHF (dostajemy setValue z useForm()). Alternatywnie,

można zrobić własny kontrolowany komponent DatePicker i użyć Controller (gdzie w render będzie nasz mechanizm wyświetlania i modalu).
- Walidacja daty: Zod ma typ z.date() do walidacji obiektu daty, Yup ma date(). Pamiętajmy, że date-picker zwraca obiekt typu Date (nie string), więc nasz schemat i typ formularza też powinien to przewidywać.

Masked Input (maski formatowania): Wprowadzanie danych takich jak **numer telefonu, PESEL, karta kredytowa, kod pocztowy** itp. bywa obarczone specyficznym formatem. Maski wejściowe to rozwiązanie, które *dynamicznie formatuje wpisywany tekst* zgodnie z określonym wzorcem, ułatwiając użytkownikowi zadanie i zapobiegając błędom formatowania. Przykładowo: numer telefonu może automatycznie pojawiać się jako 123-456-789 w trakcie wpisywania, a kod kredytowy w grupach po 4 cyfry.

W RN istnieje kilka bibliotek do masek, m.in.:

- **react-native-text-input-mask** – popularna biblioteka napisana częściowo w natywnym kodzie (iOS/Android), oferująca wydajne maskowanie bez opóźnień.
- **react-native-mask-input** – biblioteka czysto JS, łatwa w użyciu, korzystająca z wyrażeń regularnych do definicji maski. Pozwala np. maskę numeru telefonu zdefiniować jako mask={['+', /\d/, /\d/, ' ', /\d/, /\d/, /\d/, ...]}. Jest nieco mniej wydajna niż natywne rozwiązania i może powodować krótkie mignięcie niepoprawnego znaku przed usunięciem (bo maskowanie odbywa się po stronie JS).
- Inne: react-native-masked-text (starsza), react-input-mask (webowe podejście, raczej nie RN), lub własne rozwiązanie (np. poprzez onChangeText i formatowanie tekstu manualnie).

Czy warto używać maski? Tak, jeśli format jest ściśle określony i powszechnie używany (telefony, daty, karty). Zwiększa to *walidację prewencyjną* – użytkownik nie wpisze złego znaku, bo maska na to nie pozwoli. Ponadto poprawia czytelność (np. łatwiej odczytać numer karty w grupach cyfr). Należy jednak:

- Wybrać bibliotekę maskującą rozważnie – np. react-native-text-input-mask zapewni płynne działanie (ważne przy długich maskach), podczas gdy czysto JS-owe mogą sporadycznie powodować efekt „jumpowania" kursora (zwłaszcza w nowych architekturach RN). Wspomniana analiza pokazuje, że biblioteki działające w JS **mogą chwilowo pokazać niedozwolony znak zanim go usuną**, bo maskowanie następuje asynchronicznie po wpisaniu. Wrażenia te minimalizuje natywna implementacja.
- Pamiętać o integracji z RHF – maskowany input w gruncie rzeczy wciąż jest TextInputem. Najprościej użyć Controller i wewnątrz w render użyć komponentu maskującego (np. <MaskInput value={value} onChangeText={onChange} mask={...} />). W razie problemów z wydajnością, można rozważyć nie kontrolować go przez RHF na każde wpisanie, tylko użyć maski zewnętrznie i przekazywać gotową wartość np. dopiero przy onBlur. To zaawansowany temat, ale wspominam, bo np. przy bardzo długich formularzach z wieloma maskami, kontrolowanie ich wszystkich może jednak generować pewne overheady.
- W walidacji i tak warto sprawdzić dane – np. zdjąć maskę i policzyć cyfry. Maski pomagają, ale nie zastępują walidacji biznesowej (np. czy numer karty jest poprawny,

czy numer telefonu ma 9 cyfr – bo ktoś może wkleić tekst zamiast wpisywać manualnie itp.).

Przykład użycia maski (fragment):

```
<Controller
  control={control}
  name="phone"
  render={({ field: { onChange, value } }) => (
    <MaskInput
      value={value}
      onChangeText={(formatted, raw) => {
        onChange(raw); // zapisujemy tylko cyfry (surowe)
      }}
      mask={['+','4','8',' ', /\d/, /\d/, /\d/, ' ', /\d/, /\d/, /\d/, ' ', /\d/, /\d/, /\d/]}
      placeholder="+48 ___ ___ ___"
      keyboardType="number-pad"
    />
  )}
/>
```

Tutaj maska wymusza format polskiego numeru kierunkowego +48 i 9 cyfr z odstępami co 3. Używamy funkcji onChangeText komponentu maski, która dostarcza formatted (sformatowany tekst do wyświetlenia) oraz raw (sam ciąg cyfr) – zapisujemy w stanie formularza tylko raw (np. do walidacji). Jednocześnie użytkownik widzi formatowany numer podczas wpisywania.

Nawigacja i interakcja

Kolejność focus (tab order): Już wcześniej omówiliśmy mechanizm przełączania fokusu poprzez onSubmitEditing. Ważne jest, by ustawić tę kolejność logicznie odpowiadającą ułożeniu pól na ekranie. Np. jeśli layout ma dwa pola w wierszu (co na mobilnych rzadziej, ale możliwe), zastanówmy się czy focus ma iść wierszami czy kolumnami – zwykle naturalnie wierszami. W RN kolejność fokusów to kolejność w strukturze komponentów, a onSubmitEditing daje pełną kontrolę – co oznacza, że musimy to jawnie obsłużyć. Testujmy na urządzeniu: wpisywanie i naciskanie Next powinno intuicyjnie przejść do następnego pola, a Done wysłać formularz.

Przycisk Submit: Powinien być wyraźnie widoczny po wypełnieniu formularza. Często styluje się go wyróżniającym kolorem i umieszcza na dole ekranu (ale tak, by przy wysuniętej klawiaturze też był dostępny – np. wewnątrz ScrollView, który można przewinąć).

- **Dezaktywacja przycisku:** Dobrym zwyczajem jest dezaktywowanie przycisku „Wyślij” dopóki formularz ma niepoprawne lub brakujące dane. Np. jeśli korzystamy z RHF, można wykorzystać formState.isValid (dostępne, gdy podamy mode: 'onChange' lub użyjemy useForm({ mode: 'onChange', reValidateMode: 'onChange' }) – wtedy isValid będzie true tylko gdy wszystkie reguły spełnione). Alternatywnie, można po prostu disable'ować przycisk, dopóki wymagane pola nie są wypełnione – choć to wymaga własnej logiki. W każdym razie zapobiegamy wtedy frustracji, że kliknięcie nic nie robi (albo wyświetla nagle dużo błędów). Z perspektywy UX: albo przycisk jest aktywny i

po kliknięciu wyświetlamy błędy, albo w ogóle nieklikalny i np. wyszarzony dopóki mamy niepoprawne dane. Obie drogi są stosowane, byle konsekwentnie.

- **Reakcja na submit:** Po pomyślnym wysłaniu zwykle nawigujemy użytkownika dalej (np. komunikat sukcesu, przejście do następnego ekranu). Jeśli wysłanie trwa (np. rejestracja wymaga komunikacji z serwerem), pokażmy jakiś **Loader** lub przynajmniej zmieńmy stan przycisku na „Submitting...". RHF pozwala łatwo kontrolować stan submitu przez formState.isSubmitting. Możemy np. zmienić tekst przycisku na „Wysyłanie..." i zablokować go, gdy isSubmitting = true.

Dostosowanie pól do platformy:

- Dla pól hasła korzystajmy z secureTextEntry={true} – powoduje to maskowanie znaków.
- Dla pól email/username wyłączmy autokorektę (autoCorrect={false}), bo autouzupełnianie słów nie ma tam sensu, a mogłoby wprowadzać błędne poprawki.
- Dla pól liczbowych ustawmy keyboardType="numeric" lub "number-pad", a nawet atrybuty typu maxLength (RN TextInput obsługuje maxLength – co może uchronić przed wprowadzeniem więcej znaków niż dopuszczamy, np. 6-cyfrowy kod SMS).
- Skorzystajmy z atrybutów **autouzupełniania**: np. textContentType="oneTimeCode" dla pola kodu SMS spowoduje, że na iOS klawiatura podpowie kod z SMS jeśli system wykrył, że przyszedł. textContentType="username" / "password" pozwoli menedżerom haseł podpowiedzieć hasło. W iOS jest nawet textContentType="newPassword" do sugerowania silnego hasła przy rejestracji. W Androidzie analogicznie autoComplete="password" itp. – to wszystko sprawia, że aplikacja integruje się z systemowymi mechanizmami autofill, co jest dużym plusem UX. Nie wymaga to wiele wysiłku, a bywa zapominane przez programistów.

Błędy dostępności (Accessibility):
Tworząc formularz, musimy zadbać, aby był **dostępny dla osób z niepełnosprawnościami**, np. używających czytników ekranu (VoiceOver na iOS, TalkBack na Androidzie). Kilka wskazówek:

- **Etykiety pól:** Każde pole powinno mieć etykietę, którą czytnik ekranu odczyta. Jeśli mamy widoczny <Text> z nazwą pola tuż przed nim (np. "Email:"), to zazwyczaj czytnik powiąże to automatycznie (zwłaszcza gdy używamy komponentów takich jak <TextInput accessibilityLabel="Email">). Można jawnie ustawić accessibilityLabel na TextInput, np. "Email, pole tekstowe". Dobrze jest w labelu zawrzeć informację o błędzie, jeśli występuje – np. dynamicznie: accessibilityLabel={ error ? "Email, błąd: " + error.message : "Email" }. W ten sposób VoiceOver przeczyta od razu, że jest błąd i jaki. Alternatywnie można użyć accessibilityHint do przekazania dodatkowej informacji (np. "wymagany format: **@**.___").
- **Fokus na błędzie:** Po walidacji nieudanej, dobrze jest **przenieść fokus** na pierwsze pole z błędem lub w jakiś sposób poinformować użytkownika niewidomego, że wystąpił błąd. Prostym podejściem jest użycie właściwości accessibilityLiveRegion. Jeśli element z błędem (np. <Text style={errorText} z komunikatem) ma accessibilityLiveRegion="polite" i pojawi się w wyniku zmian, TalkBack powinien go automatycznie odczytać. Np. Android potrafi ogłosić błąd TextInputa ustawiony metodą native (TextView.setError), ale w RN musimy to sami obsłużyć.

- **Grupowanie elementów formularza:** Czasami łączy się label, pole i komunikat błędu w jeden element dostępności, aby czytnik czytał to wszystko razem. Można to uzyskać np. nadając accessible={true} na View obejmujący te elementy i ustawić accessibilityLabel tej View jako złączenie: "Email, pole edycji. Wartość: ... Błąd: ..." – ale to bywa skomplikowane i może interferować z fokusem samego pola. Alternatywnie, na iOS można użyć accessibilityElementsHidden lub accessibilityViewIsModal aby kontrolować, co jest czytane. Ogólna zasada: **dostępność powinna być testowana manualnie** (np. odpalamy TalkBack i przechodzimy przez formularz).
- **Zamknięcie klawiatury a czytnik ekranu:** Wspomniany mechanizm TouchableWithoutFeedback onPress={Keyboard.dismiss} – musieliśmy tam dać accessible={false}, żeby wrapper nie przechwytywał fokusu. Inaczej screen reader widziałby całą zawartość jako jeden wielki przycisk. To pokazuje, że drobne detale mają znaczenie – zawsze ustawiajmy accessible: false na takich technicznych obiektach dotykowych, które nie niosą znaczenia dla UX (bo ich jedyną rolą jest obsłużenie tapnięcia poza pola).

Podsumowując UX: Tworząc formularz myślmy jak użytkownik:

- Czy łatwo wprowadzić dane? (odpowiednia klawiatura, maski, automatyczne przejścia, autofill)
- Czy wiadomo co wpisać? (podpowiedzi, placeholdery, etykiety)
- Czy wiadomo gdzie jest błąd i jak go naprawić? (jasne komunikaty, przeniesienie uwagi na błąd)
- Czy na małym ekranie wszystko widać? (unikanie zasłonięcia przez klawiaturę)
- Czy obsługa jest spójna z platformą? (natywne pickery, odczytywanie pól przez system)
- Czy zapewniamy równy dostęp wszystkim użytkownikom? (dostępność, czytniki ekranu, kontrast kolorów np. czerwony komunikat na ciemnym tle – czytelność)

Zapewnienie dobrego UX często wymaga dodatkowego nakładu pracy, ale przekłada się na **mniejszą frustrację użytkowników i wyższą konwersję** (więcej wypełnionych i wysłanych formularzy). W aplikacjach, gdzie formularz jest krytyczny (np. rejestracja, zakup).

Przykład: Kompletny formularz rejestracji z walidacją (react-hook-form + Zod)

Teraz połączymy wszystkie omówione elementy w spójny przykład. Załóżmy, że tworzymy ekran **Rejestracji** w aplikacji. Formularz będzie zawierał takie pola jak: *imię*, *email*, *hasło*, *potwierdzenie hasła* oraz np. *numer telefonu*. Zaimplementujemy walidację tych pól przy użyciu **Zod** i pokażemy integrację z **react-hook-form**. Dodatkowo zadbamy o UX: maskę na numer telefonu, przełączanie focusu, właściwe typy klawiatur, itp.

Krok 1: Definicja schematu walidacji (Zod) – definiujemy nasz schemat rejestracji oraz typ wyprowadzony dla formularza:

```
import { z } from 'zod';
```

```
const RegisterSchema = z.object({
  name: z.string().min(2, "Imię jest za krótkie").max(50, "Imię jest za długie"),
  email: z.string().email("Nieprawidłowy format email"),
  password: z.string().min(8, "Hasło musi mieć co najmniej 8 znaków"),
  confirmPassword: z.string(),
  phone: z.string().regex(/^\d{9}$/, "Numer telefonu musi mieć 9 cyfr")
}).refine(data => data.password === data.confirmPassword, {
  path: ["confirmPassword"],
  message: "Hasła nie są zgodne"
});

type RegisterData = z.infer<typeof RegisterSchema>;
```

Objaśnienia:

- Imię (name) – wymagamy min 2 znaki i max 50 (błędy gdy poza zakresem). Nie użyliśmy .nonempty(), więc puste imię też złapie min(2) jako błąd „za krótkie".
- Email – metoda .email() sprawdza poprawność formatu oraz implicit wymaga niepustego. Zod domyślnie komunikat generuje po angielsku, dlatego podaliśmy własny.
- Hasło – min 8 znaków. (Można by dodać np. wymóg litery i cyfry – wtedy użyjemy .regex() z odpowiednim patternem i komunikatem).
- ConfirmPassword – tu celowo nie dajemy .min czy czegoś, bo walidacja nastąpi w .refine. Warto jednak upewnić się, że jest stringiem. Domyślnie jak nie podamy .min, to puste też przejdzie, ale refine i tak złapie niezgodność jeśli hasło było wprowadzone. Alternatywnie, można by dać .min(8, "...") żeby od razu krzyczało, że potwierdzenie też min 8.
- Phone – zakładamy polski numer 9-cyfrowy, bez prefiksu. Używamy regexu ^\d{9}$. Ten regex dopuszcza tylko dokładnie 9 cyfr (więc jeśli user wpisze inny format, będzie błąd). Komunikat jest odpowiedni.
- .refine – tutaj sprawdzamy czy password === confirmPassword. Jeśli nie, to przypisujemy błąd do pola confirmPassword. W ten sposób użytkownik zobaczy komunikat przy polu potwierdzenia (co jest naturalne, bo zwykle nie wie, które nie pasuje – domyślnie zakładamy, że to confirm jest wpisane inaczej niż password).

Krok 2: Implementacja komponentu formularza w RN – wykorzystamy RHF z resolverem Zod. Pokażemy kod TypeScript + JSX z komentarzami:

```
import React, { useRef } from 'react';
import { View, Text, TextInput, StyleSheet, TouchableOpacity, ScrollView } from 'react-native';
import { useForm, Controller } from 'react-hook-form';
import { zodResolver } from '@hookform/resolvers/zod';
import MaskInput from 'react-native-mask-input';  // biblioteka do maskowania telefonu (należy zainstalować)
import { RegisterSchema, RegisterData } from './validation'; // zakładamy, że schemat wyeksportowaliśmy z pliku validation.ts

const RegisterScreen: React.FC = () => {
  const { control, handleSubmit, formState: { errors, isValid } } = useForm<RegisterData>({
    resolver: zodResolver(RegisterSchema),
    mode: 'onChange'  // walidacja na bieżąco (dla isValid)
  });
```

```
  const emailRef = useRef<TextInput>(null);
  const passwordRef = useRef<TextInput>(null);
  const confirmRef = useRef<TextInput>(null);
  const phoneRef = useRef<TextInput>(null);

  const onSubmit = (data: RegisterData) => {
    console.log("Rejestracja - dane:", data);
    // Tu można wysłać na serwer lub nawigować dalej
  };

  return (
    <TouchableOpacity style={{ flex: 1 }} activeOpacity={1} onPress={() => { /* kliknięcie tła - nic nie robi,
klawiatura schowana automatycznie przez KeyboardAvoidingView */ }}>
    <KeyboardAvoidingView style={{ flex: 1 }} behavior="padding" keyboardVerticalOffset={80}>
      <ScrollView contentContainerStyle={styles.container} keyboardShouldPersistTaps="handled">

        <Text style={styles.label}>Imię:</Text>
        <Controller
          control={control}
          name="name"
          render={({ field: { onChange, onBlur, value } }) => (
            <TextInput
              style={[styles.input, errors.name && styles.inputError]}
              placeholder="Twoje imię"
              onBlur={onBlur}
              onChangeText={onChange}
              value={value}
              returnKeyType="next"
              onSubmitEditing={() => emailRef.current?.focus()}
            />
          )}
        />
        {errors.name && <Text style={styles.errorText}>{errors.name.message}</Text>}

        <Text style={styles.label}>Email:</Text>
        <Controller
          control={control}
          name="email"
          render={({ field: { onChange, onBlur, value } }) => (
            <TextInput
              ref={emailRef}
              style={[styles.input, errors.email && styles.inputError]}
              placeholder="adres email"
              keyboardType="email-address"
              autoCapitalize="none"
              autoCorrect={false}
              textContentType="emailAddress"
              onBlur={onBlur}
              onChangeText={onChange}
              value={value}
              returnKeyType="next"
              onSubmitEditing={() => passwordRef.current?.focus()}
            />
          )}
        />
        {errors.email && <Text style={styles.errorText}>{errors.email.message}</Text>}
```

```
<Text style={styles.label}>Hasło:</Text>
<Controller
  control={control}
  name="password"
  render={({ field: { onChange, onBlur, value } }) => (
    <TextInput
      ref={passwordRef}
      style={[styles.input, errors.password && styles.inputError]}
      placeholder="Hasło"
      secureTextEntry
      textContentType="newPassword"
      onBlur={onBlur}
      onChangeText={onChange}
      value={value}
      returnKeyType="next"
      onSubmitEditing={() => confirmRef.current?.focus()}
    />
  )}
/>
{errors.password && <Text style={styles.errorText}>{errors.password.message}</Text>}

<Text style={styles.label}>Powtórz hasło:</Text>
<Controller
  control={control}
  name="confirmPassword"
  render={({ field: { onChange, onBlur, value } }) => (
    <TextInput
      ref={confirmRef}
      style={[styles.input, errors.confirmPassword && styles.inputError]}
      placeholder="Potwierdź hasło"
      secureTextEntry
      textContentType="password"
      onBlur={onBlur}
      onChangeText={onChange}
      value={value}
      returnKeyType="next"
      onSubmitEditing={() => phoneRef.current?.focus()}
    />
  )}
/>
{errors.confirmPassword && <Text style={styles.errorText}>{errors.confirmPassword.message}</Text>}

<Text style={styles.label}>Telefon:</Text>
<Controller
  control={control}
  name="phone"
  render={({ field: { onChange, onBlur, value } }) => (
    <MaskInput
      ref={phoneRef}
      style={[styles.input, errors.phone && styles.inputError]}
      placeholder="Numer telefonu"
      keyboardType="number-pad"
      onBlur={onBlur}
      value={value}
      onChangeText={(formatted, extracted) => {
        // extracted to tylko cyfry
        onChange(extracted);
```

```
          }}
          mask={[/\d/, /\d/, /\d/, ' ', /\d/, /\d/, /\d/, ' ', /\d/, /\d/, /\d/]}
          returnKeyType="done"
          onSubmitEditing={handleSubmit(onSubmit)}
         />
        )}
       />
       {errors.phone && <Text style={styles.errorText}>{errors.phone.message}</Text>}

       <TouchableOpacity
        style={[styles.submitButton, !isValid && styles.submitButtonDisabled]}
        onPress={handleSubmit(onSubmit)}
        disabled={!isValid}
       >
        <Text style={styles.submitButtonText}>Zarejestruj się</Text>
       </TouchableOpacity>

     </ScrollView>
   </KeyboardAvoidingView>
   </TouchableOpacity>
  );
};

const styles = StyleSheet.create({
  container: {
   padding: 20
  },
  label: {
   fontSize: 16,
   marginBottom: 4
  },
  input: {
   borderWidth: 1,
   borderColor: '#ccc',
   padding: 10,
   borderRadius: 4,
   marginBottom: 8
  },
  inputError: {
   borderColor: 'red'
  },
  errorText: {
   color: 'red',
   marginBottom: 8
  },
  submitButton: {
   backgroundColor: '#4caf50',
   padding: 15,
   borderRadius: 4,
   alignItems: 'center',
   marginTop: 10
  },
  submitButtonDisabled: {
   backgroundColor: '#9E9E9E'
  },
  submitButtonText: {
   color: '#fff',
```

```
    fontSize: 16
  }
});
```

To sporo kodu – przeanalizujmy go krok po kroku pod kątem spełnienia założeń:

- Używamy useForm<RegisterData> z resolver: zodResolver(RegisterSchema). Dodaliśmy też mode: 'onChange' aby mieć dostęp do isValid na bieżąco (przycisk rejestracji jest aktywny tylko gdy formularz poprawny).
- Zdefiniowaliśmy referencje do kolejnych TextInputów: emailRef, passwordRef, itd. – służą one do zarządzania fokusem sekwencyjnie.
- W onSubmit na razie tylko logujemy dane – w prawdziwej aplikacji tu wywołalibyśmy API rejestracji lub dispatch do kontekstu/redux, a następnie np. nawigowali do ekranu głównego lub logowania.
- Cały formularz jest opakowany w KeyboardAvoidingView (behavior padding, offset 80 – zakładamy może jakiś header na ~80px). Całość jest wewnątrz ScrollView z keyboardShouldPersistTaps="handled". Dodatkowo opakowaliśmy wierzch w TouchableOpacity z activeOpacity={1} i onPress pustym – to trik: kliknięcie na tło ScrollView i tak obsłuży handled (czyli schowa klawiaturę), a dodatkowy TouchableOpacity z activeOpacity={1} zabezpiecza, że klik nie zrobi nic więcej (mógłby nie być potrzebny – to tylko ilustracja, czasem daje się po prostu <View> z style flex:1 i do niego TouchableWithoutFeedback).
- Każde pole jest zbudowane z Text label, Controller + TextInput/MaskInput, i ewentualnego Text z błędem.
- **Focus chaining:** Widzimy np. w polu Imię onSubmitEditing -> emailRef.current.focus(), w Email -> passwordRef.focus(), Password -> confirmRef.focus(), Confirm -> phoneRef.focus(), a Phone (ostatnie) -> handleSubmit(onSubmit). To zapewnia płynną nawigację klawiaturą.
- **Właściwości klawiatury:** Imię – domyślnie klawiatura tekstowa (można by dodać autoCapitalize="words" żeby imię z dużej litery – to UX, którego tu nie dodaliśmy, ale warto!). Email – keyboardType email, autoCapitalize none, autoCorrect false, textContentType emailAddress (dzięki czemu iOS np. podpowie email z ustawień). Hasło – secureTextEntry true (maskuje), textContentType newPassword (iOS sugeruje mocne hasło), confirm – secureTextEntry, textContentType password (może zaproponować automatycznie hasło z pola powyżej? Czasem iOS tak robi). Telefon – keyboardType number-pad (tylko cyfry).
- **Maska telefonu:** Użyliśmy MaskInput (z biblioteki react-native-mask-input) z maską dzielącą 9 cyfr na grupy 3-3-3. Zwraca nam formatted i extracted – przekazujemy do RHF tylko extracted (same cyfry), bo taki format oczekuje walidator (9 cyfr). Dzięki temu użytkownik może wpisywać z odstępami, a my i tak sprawdzamy surowe cyfry. Gdybyśmy chcieli prefiks kraju, maskę byśmy zrobili np. ['+','4','8',' ', …] i wtedy walidator musiałby pozwalać na 11 znaków w sumie (9 cyfr + 2 znaki +48 i spacje), lub raczej zdjęlibyśmy znaki specjalne zanim sprawdzimy. Tu dla uproszczenia bierzemy polski numer bez +48.
- **Wyświetlanie błędów:** Każde pole ma warunkowe <Text style={styles.errorText}> z komunikatem. Style: czerwony tekst i margines. Dodatkowo obramowanie pola zmienia na czerwone jeśli jest błąd (styles.inputError).

- **Przycisk rejestracji:** Jest zablokowany (disabled) i wyszarzony stylem, gdy !isValid. Po wypełnieniu wszystkich wymaganych w stylu danych i spełnieniu reguł, isValid zmieni się na true (dzięki mode: 'onChange' walidacja następuje na bieżąco) i wtedy styl podmieni się na zielony, a disabled na false. Tekst przycisku jest biały na zielonym tle dla czytelności.

Potencjalne ulepszenia:

- Można by dodać jeszcze checkbox „Akceptuję regulamin” – wtedy Controller z komponentem Switch lub CheckBox i walidacja boolean (Zod: z.literal(true) lub z.boolean().refine(val => val === true, "Musisz zaakceptować...")). Pominęliśmy to dla zwięzłości.
- Można dodać komunikaty toast lub dialog na wypadek błędu sieci przy rejestracji albo na sukces (np. „Konto utworzone pomyślnie!”). Sam RHF tego nie robi – to już logika wyżej (np. obsługa w onSubmit catch error i ustawienie np. globalnego errorMessage do pokazania).
- W kwestii dostępności: warto by dodać np. accessible={true} i accessibilityLabel dla przycisku (domyślnie powinno przeczytać tekst, więc jest OK). Dla komunikatów błędów można dodać accessibilityLiveRegion="polite". Dla TextInput można by ustawić accessibilityHint z podpowiedzią. Te rzeczy zależą od wymagań – ale ważne, że mamy etykiety jako <Text> przed polami, co zwykle screen reader interpretuje jako label.

Uruchomienie i test: Gdybyśmy uruchomili tę aplikację na urządzeniu/emulatorze, scenariusz byłby taki:

- Użytkownik otwiera ekran rejestracji. Wpisuje imię (mniej niż 2 litery) i przechodzi dalej – border robi się czerwony, pojawia się błąd „Imię jest za krótkie”.
- Wpisuje poprawne imię, błąd znika.
- Wpisuje niepoprawny email (np. „abc”) i naciśnie dalej – pojawia się błąd pod emailem.
- Poprawia email, błąd znika.
- Hasło: wpisze np. „123” – za krótkie, ale jeśli jeszcze nie wyszedł z pola (w trybie onChange i tak pewnie pokaże błąd od razu, bo isValid sprawdza całość – tu można by rozważyć criteriaMode lub reValidateMode, by nie pokazywać od razu – ale zostawmy). Po przejściu dalej błąd „min 8 znaków”.
- Confirm: wpisze inne niż hasło – błąd „Hasła nie są zgodne”. Jak poprawi, błąd zniknie.
- Telefon: jeśli wpisuje litery, maska i tak nie pozwoli (tylko cyfry). Jak wpisze mniej niż 9 cyfr i kliknie Done/Submit – dostanie błąd od walidacji schematu.
- Gdy wszystkie pola spełnią warunki, przycisk „Zarejestruj się” stanie się zielony i klikalny. Po naciśnięciu konsola pokaże zebrane dane (w realnej apce poszłoby do serwera).
- Cały czas klawiatura nie przykrywa pól dzięki KeyboardAvoidingView/ScrollView (jeśli testujemy, warto emulować np. mały ekran). Kliknięcie w scrollowalny obszar poza polem chowa klawiaturę (dzięki keyboardShouldPersistTaps).
- UX jest na poziomie zbliżonym do natywnych aplikacji: np. na iOS w polu hasło pojawi się ikonka strzeżonego hasła, autofill podpowie mocne hasło; w emailu może pojawić

się sugestia email; w telefonie w iOS user zobaczy klawiaturę numeryczną i może z AutoFill z SMS wprowadzić kod.

Dobre praktyki architektoniczne i projektowe

Na koniec zwróćmy uwagę na organizację kodu i utrzymanie czystości przy rozbudowanych formularzach:

- **Separacja logiki od widoku:** W powyższym przykładzie cały kod jest w jednym miejscu. Przy prostym formularzu to akceptowalne, ale w większych projektach warto wydzielić pewne części. Np. schemat walidacji trzymaliśmy w osobnym module (validation.ts). Podobnie, można utworzyć własne komponenty dla powtarzających się elementów formularza: np. komponent <FormTextInput> który wewnątrz używa Controller i TextInput, przyjmując propsy: name, label, placeholder, secure itp. Wtedy nasz ekran rejestracji staje się bardziej deklaratywny:

```
<FormTextInput name="email" label="Email" placeholder="adres email" keyboardType="email-address" />
<FormTextInput name="password" label="Hasło" placeholder="Hasło" secure />
```

  Taki komponent wewnętrznie może korzystać z kontekstu form (np. poprzez przekazanie control propsem albo nawet użycie useFormContext jeśli korzystamy z RHF FormProvider do zagnieżdżonych formularzy). To podejście czyni kod bardziej DRY – nie powtarzamy za każdym razem <Controller render=...> dla każdego pola, tylko wywołujemy nasz komponent polowy.

- **Modularność i reużywalność:** Jeżeli pewne grupy pól występują w kilku miejscach (np. adres składający się z ulicy, kodu, miasta), można zrobić z tego osobny komponent AddressForm z własnym schematem i własnymi polami, a następnie włączać go w większe formularze (RHF umożliwia łatwe zagnieżdżanie poprzez FormProvider i użycie useFormContext w dzieciach).
- **Czystość kodu:** Stosujmy czytelne nazwy dla pól (np. w schemacie i kodzie używaliśmy confirmPassword – konsekwentnie). Unikajmy „magicznych" stringów w wielu miejscach – lepiej zdefiniować raz. Np. komunikaty błędów, jeżeli powtarzają się lub mogą być wielojęzyczne, warto trzymać w osobnym obiekcie konfiguracyjnym lub plikach lokalizacyjnych. Wtedy schemat może korzystać z wcześniej przygotowanych stringów (można je importować).
- **Zrozumiałość dla innych deweloperów:** Formularze mogą bywać zawiłe. Dobrze jest komentarzami wyjaśnić nietypowe sztuczki (np. czemu używamy accessible={false} gdzieś, albo jak działa ta maska). Innym aspektem jest testowanie – warto napisać testy jednostkowe walidacji (schemat Zod można przetestować z różnymi danymi wejściowymi) oraz testy integracyjne ekranu (np. za pomocą Jest + React Native Testing Library symulować wpisywanie i sprawdzać czy błędy się pojawiają). To pomoże wychwycić regresje, gdy ktoś zmieni reguły.
- **Performance w bardzo dużych formularzach:** React Hook Form radzi sobie dobrze nawet z dziesiątkami pól dzięki niekontrolowaniu ich wszystkich na raz. Jednak jeśli formularz jest naprawdę ogromny, rozważ podzielenie go na kroki (tzw. wizard) – użytkownik nie powinien być przytłoczony, a i my zaoszczędzimy renderowania tylu

elementów na raz. RHF pozwala zachować stan między krokami lub po prostu można w każdym ekranie-wizardzie trzymać własny useForm i na końcu scalić dane.

- **Aktualność bibliotek:** Pamiętajmy, by kontrolować wersje zależności. React Hook Form stale rozwija się (w grudniu 2025 jest to wciąż wersja 7.x, więc nasz kod tego używa; upewnijmy się czy np. przy ewentualnym RHF 8 zmieni się API). Zod również ewoluuje, choć raczej zachowuje kompatybilność. Przy aktualizacjach RN zwracajmy uwagę na ewentualne nowe API dotyczące klawiatury czy inputów (np. RN 0.70+ wprowadził pewne zmiany w TextInput związane z nową architekturą). Na ten moment jednak podejścia tu przedstawione są zgodne z nowoczesnymi wersjami RN.

Literatura:

1. https://reactnative.dev/docs/textinput (Data dostępu: 1.10.2025) – Oficjalna dokumentacja podstawowego komponentu obsługi tekstu w React Native.
2. https://react-hook-form.com/get-started (Data dostępu: 1.10.2025) – Oficjalny przewodnik po bibliotece React Hook Form, omawiający integrację i zarządzanie stanem formularza.
3. https://zod.dev/?id=basic-usage (Data dostępu: 1.10.2025) – Dokumentacja biblioteki Zod, opisująca tworzenie schematów walidacji danych.
4. https://react-hook-form.com/get-started#SchemaValidation (Data dostępu: 1.10.2025) – Przewodnik dotyczący łączenia React Hook Form z zewnętrznymi walidatorami takimi jak Zod czy Yup.
5. https://reactnative.dev/docs/keyboardavoidingview (Data dostępu: 1.10.2025) – Dokumentacja komponentu służącego do rozwiązywania problemów z klawiaturą zasłaniającą pola formularza.