KIELCE UNIVERSITY OF TECHNOLOGY

Mobile Applications – lecture 5

Forms and Validation in React Native

Mateusz Pawełkiewicz

1.10.2025

Introduction Forms are a crucial element of most mobile applications – they allow users to enter data, log in, register accounts, or place orders. Implementing forms in React Native requires considering the specifics of mobile platforms, such as on-screen keyboard handling or ensuring appropriate validation of input data. In 2025, we have modern libraries and patterns at our disposal that simplify form creation and improve User Experience (UX). In this lecture, we will discuss in detail:

- **Basic form components in RN**, including TextInput, handling focus/blur events, and ways to avoid fields being obscured by the keyboard (e.g., using KeyboardAvoidingView).
- **Using the react-hook-form library** for easy form state management, including the Controller component, error handling, and onSubmit events.
- **Schema validation using Zod or Yup libraries** – we will compare them, show integration with react-hook-form, and how to generate error messages.
- **Mobile form UX best practices:** using pickers, date-pickers, input masks (e.g., for phone numbers), switching focus between fields, handling the submit button, and accessibility issues.
- **Comprehensive example (demo):** we will implement a registration/login form from scratch using react-hook-form and Zod, with full validation and proper mobile UX.

Form Components in React Native

React Native provides basic components for building forms, the most important of which is TextInput – used for user text input. Besides this, we often use switches (Switch), buttons (Button or touchable components from the Touchable* family), as well as components from external libraries (e.g., date pickers or dropdown lists). In this part, we will focus on TextInput and related topics: focus/blur events and on-screen keyboard handling.

TextInput – Basics and Focus/Blur Events

TextInput is the base RN component for entering text. It works similarly to <input> in React (web) but has its own properties and methods. Key features:

- We can listen for events: onChangeText (every text change), onFocus (entering the field), and onBlur (leaving the field). This enables, for example, dynamic field style changes, validation upon leaving, etc.
- The component exposes .focus() and .blur() methods for programmatically setting or removing focus. Thanks to these, we can, for instance, automatically move focus to the next form field.
- Many properties allow customizing the on-screen keyboard: e.g., keyboardType (defines keyboard type, e.g., numeric, email), secureTextEntry (password mode), returnKeyType (text of the return/enter button on the keyboard – e.g., "Next" or "Done"), or autoCapitalize. Setting these options correctly improves UX – for example, for an email field, we set keyboardType="email-address" and autoCapitalize="none".

- On iOS, we can also use textContentType and autoComplete to utilize system autofill mechanisms (e.g., textContentType="emailAddress" suggests the user's saved emails). It is worth using this sensitively to make filling out forms easier for users.

Focus and Blur Events: React Native allows reacting to the moment a field becomes active (focus) or is left (loss of focus). Typical applications:

- Changing the border or background of the field to highlight the currently edited one.
- Validation upon leaving the field – e.g., when the user exits the field, we check if the value is correct and potentially display an error message.
- Tracking "visited" fields (so-called *touched fields*) – this is important for displaying errors only after the user has attempted to type something.

In practice, to handle these events, we assign functions to the onFocus and onBlur props. If we use a form library (like react-hook-form), we often don't need to manually handle "touched" status – the library can mark the field as "touched" automatically upon blur.

Avoiding Field Obstruction by the Keyboard

On mobile devices, the on-screen keyboard can take up a significant portion of the screen and cover text fields located lower down. Without appropriate measures, the user might not see what they are typing. There are several techniques to deal with this problem:

1. **KeyboardAvoidingView:** This is a built-in RN component that automatically adjusts the height or position of the parent view when the keyboard appears so that the active field remains visible. Most often, the entire screen (or form section) is wrapped in <KeyboardAvoidingView behavior="padding" /> or "position". For iOS, behavior="padding" usually works well; for Android, "height" is sometimes better. You should also set keyboardVerticalOffset if you use, for example, a header – this allows correcting the position by the header's height. KeyboardAvoidingView is a simple solution but sometimes insufficient (e.g., with very long forms).
2. **ScrollView with Scrolling Option:** Wrapping the form in a ScrollView allows content scrolling, enabling the user to manually move the screen to see fields under the keyboard. A good practice is setting keyboardShouldPersistTaps="handled" or "always" – this ensures that touching the ScrollView area outside a field closes the keyboard (if no other element handles that touch). This solution prevents the situation where pressing the background only hides the keyboard (instead of, for example, triggering another button). The mentioned parameter ensures taps are passed through, resulting in closing the keyboard only when tapping an empty area.
3. **Dismiss on Background Tap:** We can manually handle hiding the keyboard when the user taps outside a field. RN provides the Keyboard module with the dismiss() method. A typical pattern is wrapping the entire screen in TouchableWithoutFeedback or Pressable, whose onPress calls Keyboard.dismiss().

   TypeScript

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

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

We then use <DismissKeyboardView> as the top-level container of the form screen. **Important detail:** setting accessible={false} on this wrapper ensures that this element will be ignored by screen readers (VoiceOver/TalkBack) and will not interfere with access to input fields. If we forgot this, our touch wrapper could be treated as an interface element by accessibility mechanisms, hindering form usage for blind users.

Summary: The best results come from a **combination of the above approaches**: e.g., the entire screen covered by KeyboardAvoidingView + ScrollView with background tap capability + automatic dismiss mechanism. In practice, you can create a Higher-Order Component (HOC) or simply nest these elements: KeyboardAvoidingView -> ScrollView (with keyboardShouldPersistTaps) -> our form content inside TouchableWithoutFeedback. Such a layout guarantees fields won't be covered, long forms can be scrolled, and clicking beside a field will hide the keyboard.

Switching Focus Between Fields

In mobile UX, it is important to facilitate the user's quick passage through the form. When the user fills in one field and presses the "Next" button on the keyboard, we want to automatically move focus to the next field. In RN, we achieve this as follows:

1. Set returnKeyType="next" for every TextInput (except the last one). Give the last field (e.g., password during login) returnKeyType="done" or "go" – so the user sees they are finishing input.
2. Listen to onSubmitEditing on TextInput components – an event triggered after pressing the "Enter/Next" button on the keyboard. For the first field, onSubmitEditing should call .focus() on the ref of the second field; for the second – focus on the third, etc. This way, the user can move through fields without touching the screen.
3. **Implementation:** We use references (useRef) to subsequent TextInputs. Example for two fields:

   TypeScript

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

   <TextInput
     placeholder="Email"
     returnKeyType="next"
     onSubmitEditing={() => passwordRef.current?.focus()}
   />
   <TextInput
     ref={passwordRef}
     placeholder="Password"
     returnKeyType="done"
     onSubmitEditing={handleSubmit(onSubmit)}  // call submit for the last field
   />
   ```

In the code above, when the user types an email and presses "Next", we call passwordRef.current.focus(), moving the cursor to the password field. When they are in the password field and press "Done", we call handleSubmit(onSubmit) (a method from react-hook-form) to send the form. Such navigation significantly improves form ergonomics.

Section Summary: When building a form in RN, we must pay attention not only to the fields themselves but also to the environment: the keyboard and navigation between fields. By using KeyboardAvoidingView, scrolling, and the keyboard hiding mechanism on tap, we ensure the user always sees the active field. Meanwhile, handling the Next/Done button on the keyboard enables filling out the form quickly without taking hands off the keyboard. In the next parts, we will move on to a library that facilitates form state management and validation.

React Hook Form – Form State Management

React Hook Form (RHF) is currently one of the most popular libraries for handling forms in the React ecosystem (including React Native). Its emergence revolutionized the approach to forms by focusing on performance and simplicity.

Why React Hook Form?

The traditional approach to forms in React (controlling input values via state and handling onChange) can be inefficient – every change causes a component render, which is costly in the case of many fields. React Hook Form was designed to utilize **uncontrolled components** and references, minimizing the number of renders needed to handle the form. According to the documentation: react-hook-form builds forms based on uncontrolled inputs, striving for maximum performance and minimal re-renders. This makes it ideal for React Native, where excessive rendering of fields (especially those with animations or formatting) can cause visible delays.

Key advantages of RHF:

- **Performance:** RHF does not keep the value of every field in the React component state but relies on native elements (TextInput) and references. It updates React state only when necessary (e.g., a validation error occurs). This means minimal re-rendering and better performance compared to the controlled approach.
- **API Simplicity:** The library provides the useForm hook, which supplies tools for handling the form (e.g., register, handleSubmit, errors). It integrates with native form elements in React and RN without forcing the use of special form components (as Formik does).
- **Integration with Validators:** RHF easily connects with external schema validation libraries (Yup, Zod, etc.) via so-called *resolvers*. We can thus define validation rules in one place and have both validation and data typing.

- **Smaller Size and Dependencies:** RHF is a fairly lightweight library without large dependencies, which matters in mobile apps (bundle size).
- **Community and Support:** It has become the de-facto standard in new projects, hence plenty of materials, examples, and active support.

Basics of Using react-hook-form in RN

To use RHF, install the package: npm install react-hook-form @hookform/resolvers

Note: @hookform/resolvers is an additional package containing resolvers for integration with validation libraries (Yup, Zod). We will return to this in the schema validation section.

The most important hook is useForm – called inside the component containing the form. Example usage in a functional component:

TypeScript

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

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

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

Here, we called useForm<FormData>(), optionally passing a generic form type (so errors etc. will be typed). We receive an object with several properties:

- **control**: The form control object, needed mainly for binding with the Controller component.
- **handleSubmit**: A function used to handle form submission. We use it to wrap our onSubmit function – it ensures validation and passes gathered data to us if everything is OK.
- **formState: { errors }**: An object containing potential validation errors for fields (properties correspond to field names). If a given field has an error, errors.fieldName will contain, for example, a message.
- *(Optional register – however, in the context of React Native, instead of manually registering inputs, the Controller is usually used, as described below).*

Standardly in RN, we don't have a <form> element like in web, so there is no form onSubmit event – that's why we use handleSubmit. In practice, we often do something like this on the Submit button:

JavaScript

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

Calling handleSubmit(onSubmit) returns a function that, upon clicking: will validate all fields, and if successful, will call our onSubmit(data) with the data object. This ensures onSubmit receives only valid data (otherwise, onSubmit won't execute, and errors will be saved in errors).

Using the Controller Component in React Native

In React Hook Form on the web (e.g., with <input>), the ref attribute or register is often used directly on the field, e.g., <input {...register('email')}>. However, in React Native and TextInput, we don't have an easy way to register it via ref (the component is not purely HTML). Instead, the RHF library provides the <Controller> component, which acts as a "bridge" between our form logic and the interface component.

Controller accepts several props:

- **name**: Field name (must correspond to the key in the form data object).
- **control**: We pass the control object obtained from useForm() here.
- **rules** (optional): An object with basic validation rules (if we are not using a schema resolver). We can set e.g., required: true or more specifically: maxLength: { value: 100, message: "Max 100 chars" }.
- **render**: A render function that should return our actual input component. This function receives certain parameters (often unpacked as { field: { onChange, onBlur, value } }), which we must pass to our input component.

To better understand, let's look at a code snippet using Controller for a text field:

JavaScript

```
<Controller
 control={control}
 name="email"
 rules={{
  required: "Email is required",
  pattern: { value: /\S+@\S+\.\S+/, message: "Invalid 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>
 )}
/>
```

Explanation:

1. We pass control from our form and name="email" – so the Controller "knows" which field it works with.
2. In rules, we defined that email is required (error message if empty) and should match a simple email regex (otherwise, show an invalid format message). *Note: The above approach with rules shows built-in RHF validation. Later, we will see how to use schema validation (Yup/Zod) instead.*
3. The render prop is a function that receives an object containing, among others, field: { onChange, onBlur, value, name }. We destructure this and use it:
    - onChange is assigned to TextInput's onChangeText – thanks to this, every text change updates the value in the form state. **Important:** we don't call our own setState here – RHF manages the value.
    - onBlur is assigned to TextInput's onBlur – upon leaving the field, RHF will mark it as "touched" and potentially trigger validation (e.g., show a "required" error if empty).
    - value is assigned to the component's value – the value is controlled by RHF.
4. Then in JSX next to the field, we conditionally render an error message if errors.email exists. errors.email.message will contain the error text passed in rules (or from the schema resolver).

Thanks to the Controller, we can use uncontrolled RN components while simultaneously connecting them to the form library's control.

A few practical notes:

- We don't have to use Controller for every element. If we have a simple Switch or slider, we can sometimes register it differently. However, in most cases in RN, this is the most convenient method.
- There is also a useController hook giving a similar effect in a non-JSX component, but the <Controller> component in JSX is usually simpler.
- rules handles basic validations – however, for more complex conditions or multiple interdependent fields, it is better to use schema validation.
- When using schema validation (resolver), there is no need to duplicate rules in rules – we can omit them or use them for minor extras (e.g., rules={{ required: true }} just to mark mandatoriness – though this can also be in the schema).

Error Handling and Messages

RHF provides error information in formState.errors. Each entry errors[fieldName] contains, among others, message (if we defined a message in rules or the schema validator provides it), type (error type, e.g., "required", "maxLength"), and other info (e.g., actual and required for length validation).

The simplest approach is to display the error under the field, as shown above. **Some best practices:**

- Error messages should be short, understandable, and help correct the data (e.g., "Password must be at least 8 characters").
- It's worth styling errors with a distinct color (red) and, for example, a smaller font.
- Fields with errors can also be marked visually (e.g., with a red border). To do this, we can add a condition to the TextInput style: style={[styles.input, errors.password && styles.inputError]} – having previously defined e.g., borderColor: 'red' in styles.inputError.

Regarding **UX for displaying errors** – it is often better to show errors only after the user has finished interacting with the field (onBlur) or attempted to submit the form. RHF supports this – by default, handleSubmit marks all fields as *touched* upon submission attempt, so errors will appear. We can also change default settings, e.g., useForm({ mode: 'onBlur' }) will cause validation to be performed after leaving the field, and mode: 'onChange' – continuously while typing (which can sometimes be too aggressive). The default mode is 'onSubmit' (validation mainly at submit, but errors can still be displayed earlier if the field is *touched*).

Summarizing work with RHF: Our form component in RN will contain:

1. Initialization of useForm with an appropriate resolver (if using Yup/Zod) or defaultValues.
2. Several <Controller>s corresponding to fields, inside which are specific <TextInput>s or other elements (Picker, Switch, etc.) bound via onChange/value.
3. Text elements displaying errors under fields.
4. A Submit button (TouchableOpacity/Button) calling handleSubmit(onSubmit).
5. Possibly additional buttons, e.g., "Reset" (which can use reset() from RHF).

Let's now move to the key issue of validation – especially using external libraries to define rules.

Schema-based Form Validation (Zod vs Yup)

Schema validation involves defining a data structure (schema) and rules for individual fields, and then using this schema to verify data correctness. This approach has several advantages:

- **Centralization of rules:** All validation rules are gathered in one place (the schema), not scattered across components.
- **Reusability:** The same schema can be applied on the frontend (for preliminary validation) and backend (for final data validation e.g., before saving to the database), reducing duplication.
- **Better TypeScript integration:** Libraries like Zod allow automatically deriving a TypeScript type from the validation schema. Thanks to this, our form data can have types strictly consistent with validation rules – increasing safety and ease of work.

In the React ecosystem, two schema validation libraries dominate: **Yup** (popular for a long time, well-integrated with Formik) and **Zod** (relatively newer, gaining popularity due to strict TypeScript integration). Let's briefly look at both:

- **Yup:** A validator modeled after the Joi library (known from Node.js). It allows declaratively creating schemas via method chaining (e.g., yup.string().email().required()). It has built-in validations for simple types, strings, numbers, arrays, etc., handles dependencies between fields (ref to another field, when method for conditional validation).
- **Zod:** A library designed from the ground up for TypeScript. Creating a schema involves calling functions (e.g., z.string().email()), very similar to Yup, but every Zod schema is simultaneously a TypeScript type – we can use z.infer<typeof schema> to get the type. Zod enforces data parsing (method .parse() or safe .safeParse()), integrating validation and parsing into one (reducing the risk of type inconsistencies).

Comparison: Both libraries achieve similar goals, syntax is similar. In practice:

- **Yup** is "older", so many examples and projects (especially with Formik) use it. It has a mature API, but TS integration is a bit patchy (Yup can generate types, but it can be unreliable with complex schemas).
- **Zod** is "newer" and TS-first – every schema is the source of truth for validation and types. Zod won't allow, for example, using a value outside a defined enum without reporting a type error (using infer). It also possesses better complex validation mechanisms (refinements) and makes validating nested structures and logically dependent fields easier.

Integrating Schemas with react-hook-form (resolvers)

React Hook Form provides the mentioned @hookform/resolvers package, which contains ready-made integrations with various validation libraries (Yup, Zod, Joi, AJV, etc.). Thanks to this, we can add the resolver option to useForm, and RHF will take care of the rest – i.e., upon calling handleSubmit, it will automatically verify data against the schema and fill the errors object with any errors.

Example with Zod: Suppose we have a Zod schema:

TypeScript

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

const LoginSchema = z.object({
  email: z.string().email("Invalid email format").nonempty("Email is required"),
  password: z.string().min(6, "Password must be min. 6 chars").nonempty("Password is required"),
});
```

Here we defined that email must be non-empty and in email format, and password non-empty and min. 6 characters. Now we can do:

TypeScript

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

type LoginData = z.infer<typeof LoginSchema>; // automatic data type based on schema
```

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

This single assignment `resolver: zodResolver(LoginSchema)` ensures that when the user submits the form, RHF:

1. Retrieves current values of all fields.
2. Passes them to the Zod validator.
3. Receives the validation result – if there are errors, it automatically fills `errors` and DOES NOT call `onSubmit`; if no errors, it allows `onSubmit` to be called with data.

It is worth mentioning that resolvers can also perform certain transformations – e.g., Zod can parse data (e.g., convert string to number if we define so). RHF defaults to `mode: 'onSubmit'` when using a resolver, but this can be changed (e.g., `mode: 'onBlur'` to validate each field upon exit).

Error Messages: In the schema above, notice that we passed a message with each constraint (e.g., "Invalid email format"). Zod allows providing an error message immediately in methods like `.email()` or `.min()`, which will be returned. Yup has a similar mechanism – e.g., `.min(6, "Password too short")`. It is good practice to define all error texts in one place (the schema), facilitating potential translations or modifications. RHF via the resolver will automatically set these messages in `errors[field].message`, so we can display them in the component as before.

Cross-field Validation: Often we need to check dependencies, e.g., password confirmation must match the password. How to do this?

- In **Yup**, one can use `.oneOf([yup.ref('password')], "Passwords must match")` for the `confirmPassword` field.
- In **Zod**, we can use `.superRefine()` or `.refine()` on the object. Example:

  TypeScript

  ```
  const RegisterSchema = z.object({
    password: z.string().min(6, "Min 6 chars"),
    confirm: z.string()
  }).refine(data => data.confirm === data.password, {
    path: ['confirm'],  // indicates error concerns the confirm field
    message: "Passwords do not match"
  });
  ```

  The call to `.refine` adds a custom validation rule at the entire object level: if the condition (`confirm === password`) is not met, it generates an error assigned to the `confirm` field with the provided message.

Validation Result vs TypeScript: In the case of **Zod**, using `z.infer<typeof schema>` ensures that the `LoginData` data object corresponds exactly to the schema (e.g., email is a string, password is a string, etc.). In Yup, we can use `InferType<typeof schema>` from the `yup` package, but there is a risk that the type definition won't reflect all complex dependencies. Zod guarantees that if

validation passes, the output data meets the given type, because validation itself occurs via the .parse method which throws an exception on type mismatch. As analysis indicates, Zod eliminates the risk of inconsistency between types and validation – the schema is the single source of truth for both.

Which Library to Choose? In 2025, the choice often falls on **Zod** in new TS projects, whereas **Yup** may remain in existing projects due to maturity and habit. Both will do the job. From a React Native and performance perspective, there is no big difference – validation happens in JavaScript anyway. If you care about fully utilizing TypeScript – Zod gives an advantage. If you have ready schemas in Yup and they work – there is no necessity to rewrite to Zod by force.

Integration with RHF is great for both: just choose the appropriate resolver (yupResolver or zodResolver).

Mobile Form UX – Best Practices

Besides correct functionality and validation, we must take care of User Experience (UX). Mobile users expect the form to be comfortable, intuitive, and adapted to the device (e.g., uses the correct keyboard for a phone number field). Let's discuss key aspects of form UX on mobile platforms:

Friendly Input Components

Select / Picker: Sometimes form fields should allow the user to choose one of predefined options. On web, we'd use <select>, but in RN? RN up to version 0.65 had a built-in Picker; currently, it is available as a separate package @react-native-picker/picker. The Picker on iOS appears as a classic wheel selector at the bottom of the screen, and on Android as a dropdown list or modal. **Best practices:**

- For small lists (a few options), one might use ActionSheet (iOS) or Modal with a custom option list – but it's best to use the native picker for UX consistency.
- **Integration with RHF:** wrap it with Controller just like TextInput. Since the picker doesn't have onChangeText but e.g. onValueChange, we must pass appropriately: onChange -> onValueChange, value -> selectedValue.
- Ensure the picker has a default value or a placeholder like "Select an option..." so the user knows they need to pick something. In validation, we can treat no selection as a required error.

DatePicker (Date/Time Selection): Many forms require dates. In mobile UX, we don't force the user to type the date manually – instead, we use native date/time selection controls. In RN, the standard is the @react-native-community/datetimepicker library, providing native date/time selection windows (iOS: wheels or calendar, Android: dialog). **How to enable:**

- Usually, it works like this: where the date is to be selected, we put a text field (e.g., TextInput or just TouchableOpacity displaying the current selection), and upon clicking,

we set state showDatePicker = true, which causes the DateTimePicker component to display. Upon date selection, it calls onChange – there we disable visibility and set the selected date in the form state.
- **Integration with RHF:** we can treat the date field as a normal value. Simplest way – store date in component state (useState) and in the picker's onChange call setValue('birthDate', selectedDate) from RHF. Alternatively, make a custom controlled DatePicker component and use Controller.
- **Validation:** Zod has z.date() type, Yup has date(). Remember that the date-picker returns a Date object (not a string), so our schema and form type should anticipate this.

Masked Input: Entering data like phone numbers, IDs, credit cards, postal codes, etc., involves specific formats. Input masks dynamically format typed text according to a pattern, facilitating the task and preventing formatting errors. In RN, there are several libraries for masks, e.g.:

- react-native-text-input-mask – popular, partly native (iOS/Android), efficient.
- react-native-mask-input – pure JS library, easy to use, uses RegEx for mask definition. Allows defining a phone mask as mask={['+', /\d/, /\d/, ' ', ...]}. Slightly less efficient than native but sufficient.
- **Should you use a mask?** Yes, if the format is strictly defined. It increases preventive validation. However, remember to integrate with RHF (use Controller, render the masking component inside) and potentially validate raw data (e.g., remove the mask and count digits).

Navigation and Interaction

Tab Order: We already discussed focus switching via onSubmitEditing. It is important to set this order logically corresponding to field layout. Test on a device: typing and pressing Next should intuitively move to the next field, and Done should submit the form.

Submit Button: Should be clearly visible.

- **Deactivation:** A good habit is deactivating the "Send" button until the form has valid data. If using RHF, utilize formState.isValid (available when mode: 'onChange').
- **Reaction to submit:** After successful sending, navigate the user further. If sending takes time, show a Loader or change button state to "Submitting...". RHF allows controlling submit state via formState.isSubmitting.

Platform Adaptation:

- Use secureTextEntry={true} for passwords.
- Disable autocorrect (autoCorrect={false}) for email/username.
- Use keyboardType="numeric" or "number-pad" for numeric fields.
- Use autocomplete attributes: e.g., textContentType="oneTimeCode" for SMS codes (iOS keyboard suggests code), textContentType="newPassword" on iOS suggests a strong password.

Accessibility Errors

When creating a form, we must ensure it is accessible to people with disabilities (e.g., using VoiceOver/TalkBack).

- **Field Labels:** Every field should have a label. If we have a visible <Text> label, usually the reader associates it automatically. You can explicitly set accessibilityLabel on TextInput. Good practice: include error info in the label dynamically, e.g., accessibilityLabel={ error ? "Email, error: " + error.message : "Email" }.
- **Focus on Error:** After failed validation, it's good to move focus to the first field with an error or use accessibilityLiveRegion="polite" on the error message so TalkBack announces it.
- **Hiding Technical Elements:** Remember accessible={false} on the TouchableWithoutFeedback used for keyboard dismissal.

Example: Complete Registration Form with Validation (react-hook-form + Zod)

Now we will combine all discussed elements into a coherent example. We will create a Registration screen with fields: name, email, password, password confirmation, and phone number. We will implement validation using Zod and show integration with react-hook-form. Additionally, we'll handle UX: phone mask, focus switching, keyboard types.

Step 1: Validation Schema Definition (Zod)

TypeScript

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

const RegisterSchema = z.object({
  name: z.string().min(2, "Name is too short").max(50, "Name is too long"),
  email: z.string().email("Invalid email format"),
  password: z.string().min(8, "Password must have at least 8 characters"),
  confirmPassword: z.string(),
  phone: z.string().regex(/^\d{9}$/, "Phone number must have 9 digits")
}).refine(data => data.password === data.confirmPassword, {
  path: ["confirmPassword"],
  message: "Passwords do not match"
});

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

Step 2: Component Implementation in RN

TypeScript

```
import React, { useRef } from 'react';
import { View, Text, TextInput, StyleSheet, TouchableOpacity, ScrollView, KeyboardAvoidingView } from 'react-native';
import { useForm, Controller } from 'react-hook-form';
```

```
import { zodResolver } from '@hookform/resolvers/zod';
import MaskInput from 'react-native-mask-input';  // masking library
import { RegisterSchema, RegisterData } from './validation';

const RegisterScreen: React.FC = () => {
  const { control, handleSubmit, formState: { errors, isValid } } = useForm<RegisterData>({
    resolver: zodResolver(RegisterSchema),
    mode: 'onChange'  // validation on the fly (for isValid)
  });

  // Refs for focus chaining
  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("Registration - data:", data);
    // Here you can send to server or navigate further
  };

  return (
    <TouchableOpacity style={{ flex: 1 }} activeOpacity={1} onPress={() => { /* background click - handled by
scrollview config */ }}>
      <KeyboardAvoidingView style={{ flex: 1 }} behavior="padding" keyboardVerticalOffset={80}>
        <ScrollView contentContainerStyle={styles.container} keyboardShouldPersistTaps="handled">

          <Text style={styles.label}>Name:</Text>
          <Controller
            control={control}
            name="name"
            render={({ field: { onChange, onBlur, value } }) => (
              <TextInput
                style={[styles.input, errors.name && styles.inputError]}
                placeholder="Your name"
                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="Email address"
                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}>Password:</Text>
<Controller
  control={control}
  name="password"
  render={({ field: { onChange, onBlur, value } }) => (
   <TextInput
    ref={passwordRef}
    style={[styles.input, errors.password && styles.inputError]}
    placeholder="Password"
    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}>Confirm Password:</Text>
<Controller
  control={control}
  name="confirmPassword"
  render={({ field: { onChange, onBlur, value } }) => (
   <TextInput
    ref={confirmRef}
    style={[styles.input, errors.confirmPassword && styles.inputError]}
    placeholder="Confirm password"
    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}>Phone:</Text>
<Controller
  control={control}
  name="phone"
```

```
          render={({ field: { onChange, onBlur, value } }) => (
            <MaskInput
              ref={phoneRef}
              style={[styles.input, errors.phone && styles.inputError]}
              placeholder="Phone number"
              keyboardType="number-pad"
              onBlur={onBlur}
              value={value}
              onChangeText={(formatted, extracted) => {
                onChange(extracted); // save only digits (raw)
              }}
              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}>Register</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 }
});
```

Code Analysis:

- **Setup:** Used `useForm` with `zodResolver` and `mode: 'onChange'`.
- **Wrapper:** `KeyboardAvoidingView` + `ScrollView` (`keyboardShouldPersistTaps="handled"`) ensures visibility and keyboard dismissal functionality.
- **Focus Chaining:** `onSubmitEditing` triggers `.focus()` on the next ref, creating a smooth flow.
- **Field Props:** Correct `keyboardType`, `textContentType`, `secureTextEntry` for optimal UX.
- **Masking:** `MaskInput` formats the phone visually but saves raw digits to RHF state.
- **Validation:** Visual feedback (red borders, error text) and submit button state (`disabled={!isValid}`).

Architectural and Design Best Practices

Finally, let's note code organization:

- **Separation of Logic:** In large projects, keep validation schemas in separate files (e.g., `validation.ts`).
- **Modularity:** Create reusable components like `<FormTextInput>` that wrap `Controller` and `TextInput` to avoid repetition (DRY).
- **Clean Code:** Avoid magic strings; keep error messages in config or localization files.
- **Performance:** For huge forms, consider a "wizard" approach (multi-step) to save rendering cost.
- **Library Versions:** React Hook Form and Zod evolve; always check for API changes in newer versions (the code above is compatible with modern v7+ approaches).

Literature:

1. https://reactnavigation.org/docs/getting-started/ (Access Date: 1.10.2025) - Official React Navigation documentation (main page).
2. https://reactnavigation.org/docs/typescript/ (Access Date: 1.10.2025) - Official guide for integrating React Navigation with TypeScript.
3. https://reactnavigation.org/docs/auth-flow/ (Access Date: 1.10.2025) - Key documentation describing the recommended authentication flow pattern.
4. https://reactnavigation.org/docs/deep-linking/ (Access Date: 1.10.2025) - Official guide for configuring Deep Links.
5. https://reactnavigation.org/docs/navigating/ (Access Date: 1.10.2025) - Documentation for basic operations (navigate, push, goBack).
6. https://reactnavigation.org/docs/params/ (Access Date: 1.10.2025) - Documentation on passing and receiving parameters (route.params).
7. https://reactnavigation.org/docs/hooks/ (Access Date: 1.10.2025) - Documentation for useNavigation and useRoute hooks.
8. https://reactnavigation.org/docs/native-stack-navigator/ (Access Date: 1.10.2025) - Documentation for Native Stack Navigator (recommended for performance).
9. https://reactnavigation.org/docs/bottom-tab-navigator/ (Access Date: 1.10.2025) - Documentation for Bottom Tab Navigator.
10. https://docs.expo.dev/routing/linking/ (Access Date: 1.10.2025) - Expo guide regarding linking configuration (including expo-linking).