



Globale Validierung in React mit TanStack Form, Zod, shadcn/ui und Zustand

Ausgangssituation

Du hast eine React/TypeScript-Anwendung, die Komponenten aus **shadcn/ui** benutzt und ihre Formulare mit **TanStack Form** verwaltet. Zur Validierung nutzt du **Zod**, und die Zustände der Wizard-Schritte werden mit **Zustand** gespeichert. Jetzt möchtest du eine **globale Validierungsregel** definieren (z.B. *der eingegebene String darf maximal 120 Zeichen lang sein*) und diese Regel in mehreren Formulareingaben innerhalb eines Wizards wiederverwenden.

Die große Stärke von TanStack Form ist, dass Validierung nicht an die UI gebunden ist. Du definierst Validatoren (Funktionen oder Schema-Objekte) zentral und übergibst sie an das Formular; TanStack propagiert die Fehler an die betroffenen Felder ¹. Mit Zod lassen sich Validierungsregeln als wiederverwendbare Schemas kapseln und im Formular oder pro Feld anwenden ¹ . ² .

TanStack Form und Zod verstehen

- **Headless-Formularbibliothek:** TanStack Form ist ein „headless“-Framework – es liefert Form-Logik ohne UI. Dadurch kannst du die Komponenten von shadcn/ui verwenden und bleibst trotzdem typensicher. Die Bibliothek verwaltet den Formularzustand, ruft Validatoren auf und blockiert das Absenden bei Fehlern ³.
- **Schema-basierte Validierung:** TanStack Form unterstützt „Standard Schema Libraries“ wie Zod und Valibot nativ ⁴. Du kannst ein **einziges Schema für das gesamte Formular** definieren und es an die `validators`-Option von `useForm` übergeben; TanStack prüft dann bei jeder Änderung/Blur alle Felder gegen dieses Schema und verteilt die Fehler automatisch ¹. Zod liefert dir darüber hinaus typings, sodass das Formular vom Schema abgeleitete Typen nutzt.
- **Einfache Integration in shadcn/ui:** Bibliotheken wie **shadcn/ui** stellen nur UI-Komponenten bereit. Um beide Welten zu verbinden, kann man eine kleine Wrapper-Funktion schreiben (z. B. `useAppForm` im Dev-to-Beispiel), die `useForm` aufruft, das Zod-Schema einbindet und dann `<form.AppForm>` und `<form.AppField>` Komponenten bereitstellt ⁵. Dieses Pattern kapselt die Konfiguration und sorgt für wiederverwendbare, typensichere Formulare. ⁶

Globale Validierung mit Zod definieren

1. **Zentrale Regel definieren** – Erstelle ein Zod-Schema für Felder, die denselben Constraint teilen. Für maximal 120 Zeichen sieht das Schema so aus:

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

// globale Regel als Funktion (falls du sie mehrfach brauchst)
export const maxLength120 = z.string().max(120, {
```

```

    message: 'Der Text darf höchstens 120 Zeichen enthalten',
});

// Komplettes Schrittschema, das die Regel verwendet
export const step1Schema = z.object({
  title: maxLength120,
  description: maxLength120.optional(),
  // weitere Felder ...
});

export type Step1FormType = z.infer<typeof step1Schema>;

```

Durch die Auslagerung in `maxLength120` kannst du die Regel in vielen Schemas wiederverwenden (z.B. `step2Schema`, `profileSchema` usw.).

1. Schema als Validator einbinden – Beim Erstellen des Formulars übergibst du das Schema an TanStack Form. Das ist entweder pro Feld oder – besser – **form-weit** über `validators` möglich
¹. Beispiel für ein Formular der ersten Wizard-Seite:

```

import { useForm } from '@tanstack/react-form';
import { step1Schema } from './validation';

const Step1 = () => {
  const form = useForm<Step1FormType>({
    defaultValues: { title: '', description: '' },
    // formweiter Validator – Zod überprüft die Werte bei jeder Änderung
    validators: {
      onChange: step1Schema,
    },
    onSubmit: async ({ value }) => {
      // hier handelt man nur korrekte Werte
      // z.B. in den globalen Zustand speichern oder zum nächsten Schritt
      // navigieren
    },
  });
  return (
    <form.Provider>
      {/* Feld „title“ – Fehler vom Schema werden in field.state.meta.errors
      gesetzt */}
      <form.Field name="title">
        {(field) => (
          <div>
            <label htmlFor={field.name}>Titel</label>
            <input
              id={field.name}
              value={field.state.value}
              onChange={(e) => field.handleChange(e.target.value)}>
          </div>
        )}
      </form.Field>
    </form.Provider>
  );
}

```

```

        onBlur={field.handleBlur}
      />
      {/* Fehlermeldung anzeigen */}
      {field.state.meta.errors?.[0] && (
        <p className="text-red-500">{field.state.meta.errors[0]}</p>
      )}
    </div>
  )}
</form.Field>

/* weiteres Feld „description“ ähnlich */

/* Submit-Button, kann deaktiviert werden wenn canSubmit false ist */
<form.Subscribe selector={(s) => [s.canSubmit, s.isSubmitting]}>
  {[canSubmit, isSubmitting]} => (
    <button type="submit" disabled={!canSubmit}>
      {isSubmitting ? 'Speichern...' : 'Weiter'}
    </button>
  )
</form.Subscribe>
</form.Provider>
);
};

```

TanStack ruft das Zod-Schema bei jeder Änderung des Formularwertes auf und blockiert das Absenden, solange `canSubmit` auf `false` steht ³. Fehlermeldungen werden vom Field-Objekt bereitgestellt und können in der UI angezeigt werden.

1. Schema pro Wizard-Schritt kombinieren – Für einen mehrseitigen Wizard definierst du ein Schema pro Schritt (`step1Schema`, `step2Schema` usw.) und speicherst die Daten im Zustand. Du kannst `Zustand` verwenden, um die Daten über mehrere Schritte zu halten, oder alle Schritte in einem einzigen TanStack-Formular modellieren. TanStack ist für mehrseitige Formulare geeignet; die Dokumentation betont, dass die modularen Feldkomponenten komplexe Fälle wie Multi-Step-Formulare oder bedingte Felder problemlos abdecken ⁷. Wenn du jede Seite als eigenes Formular implementierst, übergibst du das jeweilige Schema. Wenn du ein Gesamtformular nutzt, definierst du ein kombiniertes Schema und blendest Felder pro Schritt ein/aus.

Validierungsmuster für Anfänger

- 1. Zentrales Validierungsmodul:** Erstelle einen Ordner `validation/` mit deinen Zod-Schemas und exportiere die Regeln (`maxLength120`) sowie die Schemas (`step1Schema`, `step2Schema`, ...). So bleibt die Validierung unabhängig von UI-Code und du verhindernst Redundanz.
- 2. `useForm` im Hook kapseln:** Wie im Dev-to-Beispiel beschrieben, kannst du einen eigenen Hook (`useAppForm`) schreiben, der `useForm` aufruft und die Schema-Adapter konfiguriert ⁶. Damit werden Standard-Komponenten wie `<form.AppField>` generiert und du musst nur noch deine Schema-Objekte übergeben.

3. **Errors ausgeben:** Zeige die Fehlermeldungen mit `<FieldError>` (shadcn/ui) oder per eigenen Code an. TanStack liefert im Field-Meta-State ein Array `errors`, das du einfach ausliest und in die UI renderst.
Achte darauf, Felder mit einem `aria-invalid`-Attribut zu versehen und Fehler unter dem Eingabefeld auszugeben – das verbessert die Barrierefreiheit.
4. **Vermeide Duplikate:** Verwende pro Feld nie mehrere unterschiedliche Validatoren (etwa Inline-Funktionen und Schema gleichzeitig). Lege die Logik an einer Stelle fest (z. B. Zod-Schema) und importiere sie in allen Formularen.
5. **Zusammenfassung:** Durch die Kombination aus TanStack Form und Zod kannst du Validierungsregeln deklarativ schreiben und mehrfach verwenden. Definiere ein Schema oder eine Validator-Funktion einmal und übergib sie dem Formular. TanStack kümmert sich um das Ausführen der Validierung bei Änderung/Blur, blockiert das Absenden, wenn das Formular ungültig ist, und propagiert Fehlermeldungen an jedes Feld ¹. shadcn/ui sorgt nur für die Darstellung, sodass du das gleiche Formular-API in unterschiedlichen Komponenten verwenden kannst ².

Fazit

TanStack Form empfiehlt *Form-Level*-Validierung, da sie einfacher wiederzuverwenden ist und sauber außerhalb deiner Komponenten definiert werden kann. Durch die Kombination mit Zod lassen sich globale Regeln wie „maximal 120 Zeichen“ als **Schema** definieren und in mehreren Formularen einsetzen ¹. Für einen Wizard mit mehreren Schritten legst du pro Schritt ein Schema an oder kombinierst sie zu einem großen Schema und nutzt Zustand, um die Daten zwischen den Schritten zu halten. Als Anfänger profitierst du von dieser Struktur: Die Validierungsregeln befinden sich an einem zentralen Ort, die UI ist sauber getrennt und durch typisierte `z.infer`-Typen bleibt dein Code sicher und leicht wartbar ⁸. Mit diesem Ansatz kannst du deine Formulare konsistent validieren und die Regeln ohne Copy-Paste in allen Schritten wiederverwenden.

¹ ³ ⁴ Form and Field Validation | TanStack Form React Docs

<https://tanstack.com/form/v1/docs/framework/react/guides/validation>

² ⁵ ⁶ [Updated] Seamless Forms with shadcn/ui and TanStack Form - DEV Community

<https://dev.to/felipestanzani/seamless-forms-with-shadcnui-and-tanstack-form-mng>

⁷ TanStack Form: Simplifying Complex Forms | Bitrock

<https://bitrock.it/blog/tanstack-form-simplifying-complex-forms.html>

⁸ Next-Gen Forms: Build Type-Safe, Validated Forms with TanStack Form and Zod | Vasyl Berkoz

<https://vberkoz.com/posts/tanstack-form-zod-tutorial/>