Komponenty wejściowe

`Input`Komponent wyświetla wejście lub listę rozwijaną, listę przycisków radiowych itp. Komponenty takie pozwalają na edycję właściwości rekordu i są wspólne w komponentach `<Edit>`i `<Create>`oraz w Filtrach listy.

```
// in src/posts.js
import * as React from "react";
import { Edit, SimpleForm, ReferenceInput, SelectInput, TextInput, required } from 'react-admin';

export const PostEdit = () => (
    <Edit title={<PostTitle />}>
        <SimpleForm>
            <TextInput disabled source="id" />
            <ReferenceInput label="User" source="userId" reference="users" validate={[required()]}>
                <SelectInput optionText="name" />
            </ReferenceInput>
            <TextInput source="title" label="Post title" validate={[required()]} />
            <TextInput multiline source="body" initialValue="Lorem Ipsum" />
        </SimpleForm>
    </Edit>
);
```

Wspólne rekwizyty wejściowe

Wszystkie komponenty wejściowe akceptują następujące właściwości:

Rekwizyt	Wymagany	Rodzaj	Domyślna	Opis
`source`	Wymagany	`string`	-	Nazwa właściwości encji, która ma być użyta dla wartości wejściowej
`label`	Opcjonalny	`string`	-	Etykieta wejściowa. W aplikacjach i18n etykieta jest przekazywana do `translate`funkcji. Domyślnie humanizowany `source`, gdy zostanie pominięty. Ustaw `label={false}`, aby ukryć etykietę.
`defaultValue`	Opcjonalny	`any`	-	Domyślna wartość wejścia.
`validate`	Opcjonalny	`Function|array`	-	Zasady walidacji dla aktualnej nieruchomości. Szczegółowe informacje można znaleźć w dokumentacji walidacyjnej .
`helperText`	Opcjonalny	`string`	-	Tekst do wyświetlenia pod danymi wejściowymi
`fullWidth`	Opcjonalny	`boolean`	`false`	Jeśli `true`, dane wejściowe rozszerzą się, aby wypełnić szerokość formularza
`className`	Opcjonalny	`string`	-	Nazwa klasy (zwykle generowana przez JSS), aby dostosować wygląd samego elementu pola

Rekwizyt	Wymagany	Rodzaj	Domyślna	Opis
`formClassName`	Opcjonalny	`string`	-	Nazwa klasy, która ma być zastosowana do kontenera danych wejściowych (np. `<div>` tworząca każdy wiersz w `<SimpleForm>`)
`sx`	Opcjonalny	`SxProps`	''	Skrót MUI do definiowania własnych stylów z dostępem do motywu

```
<TextInput source="zb_title" label="Title" initialValue="Foo" />
```

React-admin używa formularza React-Hook -Form do kontrolowania danych wejściowych formularza. Każdy składnik wejściowy akceptuje również wszystkie opcje zaczepu useController typu „react-hook-form” z dodatkiem:

Rekwizyt	Wymagany	Rodzaj	Domyślna	Opis
`defaultValue`	Opcjonalny	`mixed`	-	Wartość do ustawienia, gdy właściwość jest `null` lub `undefined`
`format`	Opcjonalny	`Function`	-	Wywołanie zwrotne pobiera wartość ze stanu formularza i nazwy pola oraz zwraca wartość wejściową. Zobacz sekcję Transformacja wartości wejściowej .
`parse`	Opcjonalny	`Function`	-	Wywołanie zwrotne pobiera wartość wejściową i nazwę pola i zwraca wartość, która ma być przechowywana w stanie formularza. Zobacz sekcję Transformacja wartości wejściowej .

Dodatkowe właściwości są przekazywane do podstawowego komponentu (zwykle komponentu MUI). Na przykład, gdy ustawiasz właściwość `className` na `TextInput` komponencie, bazowy MUI `<TextField>` otrzymuje ją i renderuje z niestandardowymi stylami. Możesz także ustawić podstawowy komponent `variant` i w `margin` ten sposób.

Wskazówka : Jeśli edytujesz rekord o złożonej strukturze, możesz użyć ścieżki jako `source` parametru. Na przykład, jeśli interfejs API zwróci następujący rekord „książka”:

```
{
    "id": 1234,
    "title": "War and Peace",
    "author": {
        "firstName": "Leo",
        "lastName": "Tolstoi"
    }
}
```

Następnie możesz wyświetlić tekst wejściowy, aby edytować imię autora w następujący sposób:

```
<TextInput source="author.firstName" />
```

Wskazówka : jeśli twój interfejs ma obsługiwać wiele języków, nie używaj właściwości `label`i zamiast tego umieść zlokalizowane etykiety w słowniku. Szczegółowe informacje można znaleźć w dokumentacji dotyczącej tłumaczeń.

Wskazówka : ze względu na kompatybilność komponenty wejściowe również akceptują `defaultValue`podporę - która jest po prostu kopiowana jako `initialValue`podpora.

Przepisy

Przekształcanie wartości wejściowej do/z rekordu

Format danych zwracany przez komponent wejściowy może nie odpowiadać oczekiwaniom Twojego interfejsu API. Możesz użyć funkcji `parse`i `format`do przekształcenia wartości wejściowej podczas zapisywania i ładowania z rekordu.

Mnemonik dla dwóch funkcji:

- `parse()`: wejście -> nagraj
- `format()`: rekord -> wejście

Załóżmy, że użytkownik chciałby wprowadzić wartości 0-100 do pola procentowego, ale Twój interfejs API (stąd rekord) oczekuje wartości 0-1,0. Aby zarchiwizować transformację, możesz użyć prostych `parse()`i funkcji:`format()`

```
<NumberInput source="percent" format={v => v * 100} parse={v => parseFloat(v) / 100} label="Formatted number" />
```

`<DateInput>`przechowuje i zwraca ciąg. Jeśli chcesz zamiast tego przechowywać obiekt Data JavaScript w swoim rekordzie:

```
const dateFormatter = v => {
  // v is a `Date` object
  if (!(v instanceof Date) || isNaN(v)) return;
  const pad = '00';
  const yy = v.getFullYear().toString();
  const mm = (v.getMonth() + 1).toString();
  const dd = v.getDate().toString();
  return `${yy}-${(pad + mm).slice(-2)}-${(pad + dd).slice(-2)}`;
};

const dateParser = v => {
  // v is a string of "YYYY-MM-DD" format
  const match = /(\d{4})-(\d{2})-(\d{2})/.exec(v);
  if (match === null) return;
  const d = new Date(match[1], parseInt(match[2], 10) - 1, match[3]);
  if (isNaN(d)) return;
  return d;
};

<DateInput source="isodate" format={dateFormatter} parse={dateParser} />
```

Łączenie dwóch wejść

Formularze edycji często zawierają powiązane dane wejściowe, np. kraj i miasto (wybór tych drugich zależy od wartości pierwszego).

React-admin korzysta z formularza reakcji haka do obsługi formularzy. Możesz pobrać bieżące wartości formularza za pomocą hooka reakcji-hook-form useWatch .

```
import * as React from 'react';
import { Edit, SimpleForm, SelectInput } from 'react-admin';
import { useWatch } from 'react-hook-form';

const countries = ['USA', 'UK', 'France'];
const cities = {
    USA: ['New York', 'Los Angeles', 'Chicago', 'Houston', 'Phoenix'],
    UK: ['London', 'Birmingham', 'Glasgow', 'Liverpool', 'Bristol'],
    France: ['Paris', 'Marseille', 'Lyon', 'Toulouse', 'Nice'],
};
const toChoices = items => items.map(item => ({ id: item, name: item }));

const CityInput = props => {
    const country = useWatch({ name: 'country' });
    const values = getValues();

    return (
        <SelectInput
            choices={country ? toChoices(cities[country]) : []}
            {...props}
        />
    );
};

const OrderEdit = () => (
    <Edit>
        <SimpleForm>
            <SelectInput source="country" choices={toChoices(countries)} />
            <CityInput source="cities" />
        </SimpleForm>
    </Edit>
);

export default OrderEdit;
```

Alternatywnie możesz użyć `<FormDataConsumer>`komponentu React-admin, który pobiera wartości z formularza i przekazuje je do funkcji podrzędnej. Ponieważ `<FormDataConsumer>`używa wzorca render props, możesz uniknąć tworzenia komponentu pośredniego, takiego jak `<CityInput>`komponent powyżej:

```
import * as React from 'react';
import { Edit, SimpleForm, SelectInput, FormDataConsumer } from 'react-admin';

const OrderEdit = () => (
    <Edit>
        <SimpleForm>
            <SelectInput source="country" choices={toChoices(countries)} />
            <FormDataConsumer>
                {({ formData, ...rest }) => (
                    <SelectInput
                        source="cities"
                        choices={
                            formData.country
                                ? toChoices(cities[formData.country])
                                : []
                        }
                        {...rest}
                    />
                )}
            </FormDataConsumer>
        </SimpleForm>
    </Edit>
);
```

Wskazówka : Używając `FormDataConsumer`wewnątrz an `ArrayInput`, funkcja `FormDataConsumer`zapewni trzy dodatkowe właściwości swojej funkcji potomnej:

- `scopedFormData`: obiekt zawierający bieżące wartości aktualnie renderowanego elementu z`ArrayInput`
- `getSource`: funkcja, która przetłumaczy źródło na poprawne dla`ArrayInput`

A oto przykładowe użycie `getSource`inside `<ArrayInput>`:

```
import { FormDataConsumer } from 'react-admin';

const PostEdit = () => (
    <Edit>
        <SimpleForm>
            <ArrayInput source="authors">
                <SimpleFormIterator>
                    <TextInput source="name" />
                    <FormDataConsumer>
                        {({
                            formData, // The whole form data
                            scopedFormData, // The data for this item of the ArrayInput
                            getSource, // A function to get the valid source inside an ArrayInput
                            ...rest
                        }) =>
                            scopedFormData && scopedFormData.name ? (
                                <SelectInput
                                    source={getSource('role')} // Will translate to "authors[0].role"
                                    choices={[{ id: 1, name: 'Head Writer' }, { id: 2, name: 'Co-Writer' }]}
                                    {...rest}
                                />
                            ) : null
                        }
                    </FormDataConsumer>
                </SimpleFormIterator>
            </ArrayInput>
        </SimpleForm>
    </Edit>
);
```

Ukrywanie danych wejściowych na podstawie innych danych wejściowych

Możesz chcieć wyświetlić lub ukryć dane wejściowe na podstawie wartości innych danych wejściowych — na przykład pokaż dane `email`wejściowe tylko wtedy, gdy `hasEmail`zaznaczono dla danych logicznych `true`.

W takich przypadkach możesz zastosować podejście opisane powyżej, korzystając z `<FormDataConsumer>`komponentu.

```
import { FormDataConsumer } from 'react-admin';

 const PostEdit = () => (
     <Edit>
         <SimpleForm>
             <BooleanInput source="hasEmail" />
             <FormDataConsumer>
                 {({ formData, ...rest }) => formData.hasEmail &&
                      <TextInput source="email" {...rest} />
                 }
             </FormDataConsumer>
         </SimpleForm>
     </Edit>
 );
```

Pisanie własnego komponentu wejściowego

Jeśli potrzebujesz bardziej szczegółowego typu wejścia, możesz napisać go bezpośrednio w React. Aby obsłużyć cykl aktualizacji wartości, będziesz musiał polegać na haczyku useController w React-hook-form.

Za pomocą`useController`

Na przykład, napiszmy komponent do edycji szerokości i długości geograficznej bieżącego rekordu:

```
// in LatLongInput.js
import { useController } from 'react-hook-form';

const LatLngInput = () => {
    const input1 = useController({ name: 'lat' });
    const input2 = useController({ name: 'lng' });

    return (
        <span>
            <input {...input1.field} type="number" placeholder="latitude" />
             
            <input {...input2.field} type="number" placeholder="longitude" />
        </span>
    );
};
export default LatLngInput;

// in ItemEdit.js
const ItemEdit = () => (
    <Edit>
        <SimpleForm>
            <LatLngInput />
        </SimpleForm>
    </Edit>
);
```

`LatLngInput`nie przyjmuje właściwości, ponieważ `useController`komponent może uzyskać dostęp do bieżącego rekordu za pośrednictwem kontekstu formularza. `name`Rekwizyt służy jako selektor do edycji właściwości rekordu . Wykonanie tego komponentu wygeneruje mniej więcej następujący kod:

```
<span>
    <input name="lat" type="number" placeholder="latitude" value={record.lat} />
    <input name="lng" type="number" placeholder="longitude" value={record.lng} />
</span>
```

Wskazówka : komponent React-hook-form `useController`obsługuje notację kropkową we właściwości `name`, aby umożliwić powiązanie z zagnieżdżonymi wartościami:

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

const LatLngInput = () => {
    const input1 = useController({ name: 'position.lat' });
    const input2 = useController({ name: 'position.lng' });

    return (
        <span>
            <input {...input1.field} type="number" placeholder="latitude" />
             
            <input {...input2.field} type="number" placeholder="longitude" />
        </span>
    );
};
export default LatLngInput;
```

Za pomocą`<Labeled>`

Ten składnik nie ma etykiety. React-admin zapewnia `<Labeled>`komponent do tego:

```
// in LatLongInput.js
import { useController } from 'react-hook-form';
import { Labeled } from 'react-admin';

const LatLngInput = () => {
    const input1 = useController({ name: 'lat' });
    const input2 = useController({ name: 'lng' });

    return (
        <Labeled label="position">
            <span>
                <input {...input1.field} type="number" placeholder="latitude" />
                 
                <input {...input2.field} type="number" placeholder="longitude" />
            </span>
        </Labeled>
    );
};
export default LatLngInput;
```

Teraz komponent będzie renderowany z etykietą:

```
<label>Position</label>
<span>
    <input name="lat" type="number" placeholder="longitude" value={record.lat} />
    <input name="lng" type="number" placeholder="longitude" value={record.lng} />
</span>
```

Korzystanie z komponentów pola MUI

Zamiast `input`elementów HTML możesz użyć komponentu MUI, takiego jak `TextField`. Aby powiązać komponenty MUI z wartościami formularza, użyj `useController()`zaczepu:

```
// in LatLongInput.js
import TextField from '@mui/material/TextField';
import { useController } from 'react-hook-form';

const BoundedTextField = ({ name, label }) => {
    const {
        field,
        fieldState: { isTouched, invalid, error },
        formState: { isSubmitted }
    } = useController(name);
    return (
        <TextField
            {...field}
            label={label}
            error={(isTouched || isSubmitted) && invalid}
            helperText={(isTouched || isSubmitted) && invalid ? error : ''}
        />
    );
};
const LatLngInput = () => (
    <span>
        <BoundedTextField name="lat" label="latitude" />
         
        <BoundedTextField name="lng" label="longitude" />
    </span>
);
```

Wskazówka : komponent MUI `<TextField>`zawiera już etykietę, więc nie musisz jej używać `<Labeled>`w tym przypadku.

`useController()`zwraca trzy wartości: `field`, `fieldState`i `formState`. Aby dowiedzieć się więcej o tych właściwościach, zapoznaj się z dokumentacją hooka useController .

Zamiast `input`elementów HTML lub komponentów MUI możesz użyć komponentów wejściowych React-admin, jak `<NumberInput>`na przykład. Komponenty React-admin już używają `useController()`i zawierają już etykietę, więc nie musisz ich używać `useController()`ani `<Labeled>`podczas ich używania:

```
// in LatLongInput.js
import { NumberInput } from 'react-admin';
const LatLngInput = () => (
    <span>
        <NumberInput source="lat" label="latitude" />
         
        <NumberInput source="lng" label="longitude" />
    </span>
);
export default LatLngInput;
```

`useInput()`Hak _

React-admin dodaje funkcjonalność do formularza reakcji haka:

- obsługa niestandardowych emiterów zdarzeń, takich jak `onChange`,
- obsługa szeregu walidatorów,
- wykrywanie pól wymaganych do dodania gwiazdki do etykiety pola,
- analizować i formatować, aby przetłumaczyć wartości rekordu na wartości formularza i na odwrót.

Tak więc wewnętrznie, komponenty React-admin używają innego haka, który owija hak typu React-Hook-Form `useController()`. Nazywa się `useInput()`; użyj go zamiast `useController()`tworzyć dane wejściowe formularzy, które mają dokładnie to samo API, co komponenty wejściowe React-admin:

```
// in LatLongInput.js
import TextField from '@mui/material/TextField';
import { useInput, required } from 'react-admin';

const BoundedTextField = (props) => {
    const { onChange, onBlur, ...rest } = props;
    const {
        field,
        fieldState: { isTouched, invalid, error },
        formState: { isSubmitted },
        isRequired
    } = useInput({
        // Pass the event handlers to the hook but not the component as the field property already has them.
        // useInput will call the provided onChange and onBlur in addition to the default needed by react-hook-form.
        onChange,
        onBlur,
        ...props,
    });

    return (
        <TextField
            {...field}
            label={props.label}
            error={(isTouched || isSubmitted) && invalid}
            helperText={(isTouched || isSubmitted) && invalid ? error : ''}
            required={isRequired}
            {...rest}
        />
    );
};
const LatLngInput = props => {
    const {source, ...rest} = props;

    return (
        <span>
            <BoundedTextField source="lat" label="Latitude" validate={required()} {...rest} />
             
            <BoundedTextField source="lng" label="Longitude" validate={required()} {...rest} />
        </span>
    );
};
```

Oto kolejny przykład, tym razem z wykorzystaniem `Select`komponentu MUI:

```
// in SexInput.js
import Select from '@mui/material/Select';
import MenuItem from '@mui/material/MenuItem';
import { useInput } from 'react-admin';

const SexInput = props => {
    const {
        field,
        fieldState: { isTouched, invalid, error },
        formState: { isSubmitted }
    } = useInput(props);

    return (
        <Select
            label="Sex"
            {...field}
        >
            <MenuItem value="M">Male</MenuItem>
            <MenuItem value="F">Female</MenuItem>
        </Select>
    );
};
export default SexInput;
```

Wskazówka : `useInput`akceptuje wszystkie argumenty, do których możesz przekazać `useController`. Poza tym komponenty używające takich właściwości `useInput`jak `format`i `parse`, do konwersji wartości z formularza na dane wejściowe i odwrotnie:

```
const parse = value => {/* ... */};
const format = value => {/* ... */};

const PersonEdit = () => (
    <Edit>
        <SimpleForm>
            <SexInput
                source="sex"
                format={formValue => formValue === 0 ? 'M' : 'F'}
                parse={inputValue => inputValue === 'M' ? 0 : 1}
            />
        </SimpleForm>
    </Edit>
);
```

Komponenty innych firm

W repozytoriach innych firm można znaleźć komponenty do React-admin.

- marmelab/ra-richtext-tiptap : bogaty tekst wejściowy oparty na Tiptap
- vascofg/react-admin-color-input : dane wejściowe koloru za pomocą React Color , zbioru próbników kolorów.
- vascofg/react-admin-date-inputs : zbiór danych wejściowych opartych na selektorach interfejsu użytkownika materiałów
- MrHertal/react-admin-json-view : pole JSON i dane wejściowe dla React-admin.
- @bb-tech/ra-components : `JsonInput`pozwala tylko na prawidłowy JSON jako dane wejściowe, `JsonField`aby poprawnie wyświetlić JSON na karcie pokazu i `TrimField`przyciąć pola podczas wyświetlania `Datagrid`w `List`komponencie.
- @react-page/react-admin : ReactPage to bogaty edytor treści i zawiera gotowy do użycia komponent wejściowy React-admin. sprawdź demo

- **PRZESTARZAŁE V3** LoicMahieu/aor-tinymce-input : komponent TinyMCE, przydatny do edycji HTML