React-admin V4: Zbuduj własny framework

François Zaninotto
11 kwietnia 2022

#reaguj _ # reakcja-admin # samouczek

React-admin znajduje swoje korzenie w procesie usuwania standardowego kodu i pozwalania programistom skupić się na logice biznesowej. W przypadku React-admin v4 staje się to jeszcze bardziej widoczne. Zobaczmy, jak strona edycyjna zbudowana za pomocą React-admin wypada w porównaniu ze stroną wykonaną ręcznie.

Widok edycji zbudowany ręcznie

Widoki edycji są bardzo popularne w aplikacjach jednostronicowych. Najczęstszym sposobem umożliwienia użytkownikowi aktualizacji rekordu jest pobranie rekordu z interfejsu API na podstawie parametrów adresu URL, zainicjowanie formularza rekordem, zaktualizowanie danych wejściowych, gdy użytkownik zmieni wartości, i wywołanie interfejsu API w celu zaktualizowania zapis z nowymi wartościami po przesłaniu.

Na przykład tutaj jest widok edycji książki z formularzem wyświetlającym 3 dane wejściowe (dwa wpisy tekstowe i jedno wybrane dane wejściowe) oraz przekierowanie do widoku listy

książek po pomyślnym przesłaniu. Ten widok renderuje się pod `/books/:id` trasą.

Zapomnijmy na chwilę o response-admin. Oto jak napisałbym ten komponent w czystym React, wykorzystując router React do obsługi parametrów adresu URL i nawigacji oraz formularz React -hook-form do powiązania danych wejściowych formularza z obiektem rekordu:

```
import * as React from 'react';
import { useState, useEffect } from 'react';
import { useParams, useNavigate } from 'react-router-dom';
import { useForm, Controller } from 'react-hook-form';
import { Card, TextField, Button, Stack, MenuItem } from '@mui/material';

export const BookEdit = () => {
    const { id } = useParams();
    const { handleSubmit, reset, control } = useForm();
```



```
    // load book record on mount
    const [isLoading, setIsLoading] = useState(true);
    useEffect(() => {
        fetch(`/api/books/${id}`)
            .then(res => res.json())
            .then(({ data }) => {
                // initialize form with the result
                reset(data)
                setIsLoading(false);
            });
    }, [id]);

    // update book record on submit
    const [isSubmitting, setIsSubmitting] = useState(false);
    const navigate = useNavigate();
    const onSubmit = data => {
        setIsSubmitting(true);
        const options = {
            body: JSON.stringify({ data })
            method: 'POST',
            headers: { 'Content-Type': 'application/json' },
        };
        fetch(`/api/books/${id}`, options)
            .then(() => {
                setIsSubmitting(false);
                navigate('/books');
            });
    };

    if (isLoading) return null;

    return (
        <div>
            <h1>Book Edition</h1>
            <Card>
                <form onSubmit={handleSubmit(onSubmit)}>
                    <Stack spacing={2}>
```

```
                        <Controller
                            name="title"
                            render={({ field }) => (
                                <TextField label="Title" {...field} />
                            )}
                            control={control}
                        />
                        <Controller
                            name="author"
                            render={({ field }) => (
                                <TextField label="Author" {...field} />
                            )}
                            control={control}
                        />
                        <Controller
                            name="availability"
                            render={({ field }) => (
                                <TextField
                                    select
                                    label="Availability"
                                    {...field}
                                >
                                    <MenuItem value="in_stock">
                                        In stock
                                    </MenuItem>
                                    <MenuItem value="out_of_stock">
                                        Out of stock
                                    </MenuItem>
                                    <MenuItem value="out_of_print">
                                        Out of print
                                    </MenuItem>
                                </TextField>
                            )}
                            control={control}
                        />
                        <Button type="submit" disabled={isSubmitting}>
                            Save
```



```
                        </Button>
                    </Stack>
                </form>
            </Card>
        </div>
    );
};
```

To dużo kodu jak na tak prostą stronę! Jednak nie zarządza wartościami domyślnymi danych wejściowych, walidacji ani danych wejściowych zależnych. Nie obsługuje nawet przypadków uwierzytelniania ani błędów w wywołaniach API.

To bardzo powszechny składnik. W rzeczywistości wiele jego funkcji można wyodrębnić do ponownego wykorzystania na innych stronach.

Jak ulepszyłbyś kod i doświadczenie programisty?

Interlude: Doświadczenie programisty i złożoność

Out of the Tar Pit , słynny artykuł informatyczny opublikowany w 2006 roku, naprawdę otworzył mi oczy na rozwój oprogramowania i doświadczenie programistów.

Identyfikuje **złożoność** jako pojedynczą główną trudność w udanym rozwoju systemów oprogramowania na dużą skalę. Inne właściwości utrudniają budowanie oprogramowania (takie jak te zidentyfikowane w innym dobrze znanym dokumencie *No Silver Bullet* z 1986 r.: zgodność, zmienność i niewidzialność), ale są to szczególne formy złożoności lub spowodowane pewną złożonością systemu .

W artykule jako główne przyczyny złożoności uznano **stan** , **kontrolę** (tzn. kolejność, w jakiej się dzieje) i **ilość kodu** . Sugeruje również techniki ich redukcji:

- Orientacja obiektowa
- Programowanie funkcjonalne
- Programowanie deklaratywne

Artykuł jest bardzo czytelny, nawet jeśli nie masz dyplomu CS. Zachęcam do poświęcenia godziny na odkrycie go *in extenso* .

Użyję technik zachęcanych przez *Out Of The Tar Pit* , aby zmniejszyć złożoność początkowego przykładu kodu.

Wyodrębnianie wywołań API do niestandardowych hooków

Obsługa wywołań API jest tak powszechnym zadaniem w React, że istnieją dziesiątki bibliotek, które obsługują to za Ciebie. Do obsługi wywołań API możemy wykorzystać na przykład response-query .

Ale React-query `useQuery`i `useMutation`hooki są prymitywami niskiego poziomu. Możemy sobie wyobrazić haki wyższego poziomu, takie jak `useGetOne`i `useUpdate`do zarządzania typowymi operacjami CRUD.

Nie pokażę implementacji tych niestandardowych haków; chcę się skupić na tym, jak te haki zmieniają kod, gdy są używane zamiast `fetch()`wywołań niestandardowych:

```
 import * as React from 'react';
-import { useState, useEffect } from 'react';
 import { useParams, useNavigate } from 'react-router-dom';
 import { useForm, Controller } from 'react-hook-form';
+import { useGetOne, useUpdate } from 'b2b-framework';
 import { Card, TextField, Button, Stack, MenuItem } from '@mui/material';
```



```
 export const BookEdit = () => {
     const { id } = useParams();
     const { handleSubmit, reset, control } = useForm();

     // load book record on mount
-    const [isLoading, setIsLoading] = useState(true);
-    useEffect(() => {
-        fetch(`/api/books/${id}`)
-            .then(res => res.json())
-            .then(({ data }) => {
-                // initialize form with the result
-                reset(data)
-                setIsLoading(false);
-            });
-    }, [id]);
+    const { isLoading } = useGetOne('books', { id }, {
+        onSuccess: (data) => reset(data)
+    });

     // update book record on submit
-    const [isSubmitting, setIsSubmitting] = useState(false);
+    const [update, { isLoading: isSubmitting }] = useUpdate();
     const navigate = useNavigate();
     const onSubmit = (data) => {
-        setIsSubmitting(true);
-        const options = {
-            body: JSON.stringify({ data })
-            method: 'POST',
-            headers: { 'Content-Type': 'application/json' },
-        };
-        fetch(`/api/books/${id}`, options)
-            .then(() => {
-                setIsSubmitting(false);
-                navigate('/books');
-            });
+        update('books', { id, data }, {
+            onSuccess: () => { navigate('/books'); }
```

```
+       });
    };

    // ... no change to the rest of the component
};
```

Niestandardowe haki są dostarczane przez wyimaginowany pakiet, który wywołam `b2b-framework`. Ponieważ logika pobierania interfejsu API jest teraz obsługiwana przez te haki, dodanie tam uwierzytelniania i obsługi błędów powinno być łatwiejsze.

Usunięcie `useState`i `useEffect`usunięcie pewnego stanu z kodu oraz zmniejszenie objętości kodu. To świetny pierwszy krok w kierunku usunięcia złożoności. Ale możemy zrobić lepiej.

`<Form>`: Logika formularzy

W celu użycia `react-hook-form`z danymi wejściowymi MUI, poprzedni przykład wykorzystuje `<Controller>`znacznik, który oczekuje `control`obiektu wygenerowanego przez `useForm`zaczep (zobacz powiązany `react-hook-form`dokument).

Możemy uniknąć wywołania `useForm`, umieszczając jego logikę w niestandardowym komponencie. Nazwijmy to `<Form>`. Powinien oczekiwać, że `record`atrybut zainicjuje dane wejściowe na podstawie pól rekordu - i zresetuje, gdy `record`nastąpią zmiany. `<Form>`Powinniśmy również stworzyć formularz reakcji hooka `<FormProvider>`, dzięki czemu nie musimy już przekazywać `control`prop do każdego `<Controller>`elementu:

```
import * as React from 'react';
import { useParams, useNavigate } from 'react-router-dom';
-import { useForm, Controller } from 'react-hook-form';
+import { Controller } from 'react-hook-form';
-import { useGetOne, useUpdate, Title } from 'b2b-framework';
+import { useGetOne, useUpdate, Title, Form } from 'b2b-framework';
import { Card, TextField, Stack, MenuItem } from '@mui/material';
```



```
export const BookEdit = () => {
    const { id } = useParams();
-   const { handleSubmit, reset, control } = useForm();
-   const { isLoading } = useGetOne('books', { id }, {
-       onSuccess: (data) => reset(data)
-   });
+   const { isLoading, data } = useGetOne('books', { id });
    const [update, { isLoading: isSubmitting }] = useUpdate();
    const navigate = useNavigate();
    const onSubmit = (data) => {
        update('books', { id, data }, {
            onSuccess: () => { navigate('/books'); }
        });
    };
    if (isLoading) return null;
    return (
        <div>
            <Title title="Book Edition" />
            <Card>
-               <form onSubmit={handleSubmit(onSubmit)}>
+               <Form record={data} onSubmit={onSubmit}>
                    <Stack spacing={2}>
                        <Controller
                            name="title"
                            render={({ field }) => <TextField label="Title" {...field} />}
-                           control={control}
                        />
                        <Controller
                            name="author"
                            render={({ field }) => <TextField label="Author" {...field} />}
-                           control={control}
                        />
                        <Controller
                            name="availability"
                            render={({ field }) => (
                                <TextField select label="Availability" {...field}>
```

```
                                    <MenuItem value="in_stock">In stock</MenuItem>
                                    <MenuItem value="out_of_stock">Out of stock</MenuItem>
                                    <MenuItem value="out_of_print">Out of print</MenuItem>
                                </TextField>
                            )}
-                           control={control}
                        />
                        <Button type="submit" disabled={isSubmitting}>
                            Save
                        </Button>
                    </Stack>
-               </form>
+               </Form>
            </Card>
        </div>
    );
};
```

`<SimpleForm>`: Układ warstwowy

Wyświetlanie danych wejściowych w a `<Stack>`to powszechny wzorzec interfejsu użytkownika, który pojawia się w wielu formach. Przedstawmy `<SimpleForm>`, wygodne opakowanie `<Form>`, które zapewnia ten układ piętrowy. Powinien zawierać przycisk przesyłania, dzięki czemu `BookEdit`kod komponentu może być bardziej skoncentrowany na logice biznesowej.

```
import * as React from 'react';
import { useParams, useNavigate } from 'react-router-dom';
import { Controller } from 'react-hook-form';
-import { useGetOne, useUpdate, Title, Form } from 'b2b-framework';
+import { useGetOne, useUpdate, Title, SimpleForm } from 'b2b-framework';
-import { Card, TextField, Stack, MenuItem } from '@mui/material';
+import { Card, TextField, MenuItem } from '@mui/material';

export const BookEdit = () => {
    const { id } = useParams();
```



```
    const { isLoading, data } = useGetOne("books", { id });
    const [update, { isLoading: isSubmitting }] = useUpdate();
    const navigate = useNavigate();
    const onSubmit = (data) => {
        update('books', { id, data }, {
            onSuccess: () => { navigate('/books'); }
        });
    };
    if (isLoading) return null;
    return (
        <div>
            <Title title="Book Edition" />
            <Card>
-               <Form record={data} onSubmit={onSubmit}>
+               <SimpleForm record={data} onSubmit={onSubmit} saving={isSubmitting}>
-                   <Stack spacing={2}>
                        <Controller
                            name="title"
                            render={({ field }) => <TextField label="Title" {...field} />}
                        />
                        <Controller
                            name="author"
                            render={({ field }) => <TextField label="Author" {...field} />}
                        />
                        <Controller
                            name="availability"
                            render={({ field }) => (
                            <TextField select label="Availability" {...field}>
                                <MenuItem value="in_stock">In stock</MenuItem>
                                <MenuItem value="out_of_stock">Out of stock</MenuItem>
                                <MenuItem value="out_of_print">Out of print</MenuItem>
                            </TextField>
                            )}
                        />
-                       <Button type="submit" disabled={isSubmitting}>
-                           Save
-                       </Button>
```

```
-                   </Stack>
-               </Form>
+               </SimpleForm>
            </Card>
        </div>
    );
};
```

`<SimpleForm>`jest *składnikiem układu* . Powinien zawierać tylko logikę prezentacji, delegując rzeczywistą obsługę formularza do podstawowego `<Form>`komponentu.

Korzystanie z komponentów wejściowych

Zawijanie danych wejściowych formularza `<Controller>`tagiem to powszechny wzorzec, więc przedstawmy „Komponenty wejściowe”, które robią to w sposób wielokrotnego użytku:

- `<TextInput>`otula `<TextInput>`wnętrze a`<Controller>`
- `<SelectInput>`otula `<TextInput select>`wnętrze a`<Controller>`

Oznacza to, że `BookEdit`składnik nie musi używać bezpośrednio `react-hook-form`'s `<Controller>`:

```
 import * as React from 'react';
 import { useParams, useNavigate } from 'react-router-dom';
-import { Controller } from 'react-hook-form';
-import { useGetOne, useUpdate, Title, SimpleForm } from 'b2b-framework';
+import { useGetOne, useUpdate, Title, SimpleForm, TextInput, SelectInput } from 'b2b-framework'
-import { Card, TextField, MenuItem } from '@mui/material';
+import { Card } from '@mui/material';

 export const BookEdit = () => {
```



```
    const { id } = useParams();
    const { isLoading, data } = useGetOne("books", { id });
    const [update, { isLoading: isSubmitting }] = useUpdate();
    const navigate = useNavigate();
    const onSubmit = (data) => {
        update('books', { id, data }, {
            onSuccess: () => { navigate('/books'); }
        });
    };
    if (isLoading) return null;
    return (
        <div>
            <Title title="Book Edition" />
            <Card>
                <SimpleForm record={data} onSubmit={onSubmit} saving={isSubmitting}>
-                   <Controller
-                      name="title"
-                       render={({ field }) => <TextField label="Title" {...field} />}
-                   />
+                   <TextInput source="title" />
-                   <Controller
-                       name="author"
-                       render={({ field }) => <TextField label="Author" {...field} />}
-                   />
+                   <TextInput source="author" />
-                   <Controller
-                       name="availability"
-                       render={({ field }) => (
-                           <TextField select label="Availability" {...field}>
-                               <MenuItem value="in_stock">In stock</MenuItem>
-                               <MenuItem value="out_of_stock">Out of stock</MenuItem>
-                               <MenuItem value="out_of_print">Out of print</MenuItem>
-                           </TextField>
-                       )}
-                   />
+                   <SelectInput source="availability" choices={[
+                       { id: "in_stock", name: "In stock" },
```

```
+                       { id: "out_of_stock", name: "Out of stock" },
+                       { id: "out_of_print", name: "Out of print" },
+                   ]} />
                </SimpleForm>
            </Card>
        </div>
    );
};
```

To o wiele lepsze: usunięcie właściwości `Controller`renderowania zmniejsza również obciążenie umysłowe funkcji wbudowanych (i ich kolejność zagadkowych symboli, takich jak {{{}}.

`<EditContext>`Udostępnia dane i wywołania zwrotne

Zamiast przekazywać `record`i `onSubmit`callback do `<SimpleForm>`elementu, moglibyśmy umieścić je w niestandardowym kontekście React - nazwijmy to `EditContext`. `<SimpleForm>`odczytałbym `record`iz `onSubmit`tego kontekstu. Pozwala to dowolnemu potomkowi `<SimpleForm>`elementu na odczytanie `record`bez konieczności przekazywania go składnik po składniku:

```
import * as React from 'react';
import { useParams, useNavigate } from 'react-router-dom';
-import { useGetOne, useUpdate, Title, SimpleForm, TextInput, SelectInput } from 'b2b-framework'
+import { useGetOne, useUpdate, Title, EditContextProvider, SimpleForm, TextInput, SelectInput }
import { Card } from '@mui/material';

export const BookEdit = () => {
    const { id } = useParams();
    const { isLoading, data } = useGetOne("books", { id });
    const [update, { isLoading: isSubmitting }] = useUpdate();
    const navigate = useNavigate();
```



```
    const onSubmit = (data) => {
        update('books', { id, data }, {
            onSuccess: () => { navigate('/books'); }
        });
    };
    if (isLoading) return null;
    return (
+       <EditContextProvider value={{
+           record: data,
+           isLoading,
+           save: onSubmit,
+           saving: isSubmitting,
+       }}>
            <div>
                <Title title="Book Edition" />
                <Card>
-               <SimpleForm record={data} onSubmit={onSubmit} saving={isSubmitting}>
+                   <SimpleForm>
                        <TextInput source="title" />
                        <TextInput source="author" />
                        <SelectInput source="availability" choices={[
                            { id: "in_stock", name: "In stock" },
                            { id: "out_of_stock", name: "Out of stock" },
                            { id: "out_of_print", name: "Out of print" },
                        ]} />
                    </SimpleForm>
                </Card>
            </div>
+       </EditContextProvider>
    );
};
```

Może to wyglądać nieco bardziej szczegółowo, ale jak `<SimpleForm>`wiadomo, jak wykonać w `EditContext`, "pobiera" dane i wywołanie zwrotne z kontekstu, zamiast oczekiwać, że

programista "wypchnie" je do niego. Komponenty są inteligentniejsze, więc programista nie musi zarządzać komunikacją między nimi. Po raz kolejny zmniejsza się złożoność.

Zauważ, że jest to forma odwrócenia kontroli .

`useEditController`: Logika kontrolera

Podkreśla `ContextProvider`to fakt, że JSX potrzebuje kilku informacji, aby móc renderować formularz. Kod, który przygotowuje te informacje, może być postrzegany jako część „kontrolera" komponentu (zgodnie ze wzorcem projektowym Model-View-Controller). Zawiera początkową logikę, która pobiera identyfikator z lokalizacji, pobiera rekord z interfejsu API i buduje `save`wywołanie zwrotne.

Wyodrębnijmy go do hooka o nazwie `useEditController`, który może być ponownie użyty we wszystkich innych widokach edycji:

```
 import * as React from 'react';
-import { useParams, useNavigate } from 'react-router-dom';
-import { useGetOne, useUpdate, Title, EditContextProvider, SimpleForm, TextInput, SelectInput }
+import { useEditController, Title, EditContextProvider, SimpleForm, TextInput, SelectInput } fr
 import { Card } from "@mui/material";

 export const BookEdit = () => {
-    const { id } = useParams();
-    const { isLoading, data } = useGetOne("books", { id });
-    const [update, { isLoading: isSubmitting }] = useUpdate();
-    const navigate = useNavigate();
-    const onSubmit = (data) => {
-        update('books', { id, data }, {
-            onSuccess: () => { navigate('/books'); }
-        });
-    };
+    const editContext = useEditController();
```



```
-   if (isLoading) return null;
+   if (editContext.isLoading) return null;
    return (
-       <EditContextProvider value={{
-           record: data,
-           isLoading,
-           save: onSubmit,
-           saving: isSubmitting,
-       }}>
+       <EditContextProvider value={editContext}>
            <div>
                <Title title="Book Edition" />
                <Card>
                <SimpleForm>
                    <TextInput source="title" />
                    <TextInput source="author" />
                    <SelectInput source="availability" choices={[
                        { id: "in_stock", name: "In stock" },
                        { id: "out_of_stock", name: "Out of stock" },
                        { id: "out_of_print", name: "Out of print" },
                    ]} />
                </SimpleForm>
                </Card>
            </div>
        </EditContextProvider>
    );
};
```

Zauważ, że `useEditController`nie powinna potrzebować nazwy zasobu (w tym przypadku „książki"), ponieważ może ją odgadnąć z adresu URL (jeśli komponent jest renderowany w `/books/123`trasie).

`<EditBase>`: Wersja komponentu kontrolera

Wywołanie kontrolera i umieszczenie jego wyniku w kontekście jest w zasadzie tym, co powinna zrobić każda strona. Powinniśmy być w stanie dokonać refaktoryzacji tej logiki w innym komponencie — nazwijmy to `<EditBase>`. Będzie zarządzał `useEditController`połączeniem i `EditContextProvider`połączeniem:

```
 import * as React from 'react';
-import { useEditController, Title, EditContextProvider, SimpleForm, TextInput, SelectInput } fr
+import { EditBase, Title, SimpleForm, TextInput, SelectInput } from 'b2b-framework';
 import { Card } from "@mui/material";

 export const BookEdit = () => {
-    const editContext = useEditController();
-    if (editContext.isLoading) return null;
     return (
-        <EditContextProvider value={editContext}>
+        <EditBase>
             <div>
                 <Title title="Book Edition" />
                 <Card>
                 <SimpleForm>
                     <TextInput source="title" />
                     <TextInput source="author" />
                     <SelectInput source="availability" choices={[
                         { id: "in_stock", name: "In stock" },
                         { id: "out_of_stock", name: "Out of stock" },
                         { id: "out_of_print", name: "Out of print" },
                     ]} />
                 </SimpleForm>
                 </Card>
             </div>
-        </EditContextProvider>
+        </EditBase>
     );
 };
```


`<Edit>`Renderuje tytuł, pola i czynności

`<EditBase>`jest komponentem bezgłowym: renderuje tylko swoje dzieci. Ale prawie każdy widok edycji wymaga opakowania `<div>`, tytułu i `<Card>`. Aby umożliwić programistom skupienie się na logice biznesowej, wyodrębnijmy ten wspólny interfejs użytkownika do `<Edit>`komponentu:

```
 import * as React from 'react';
-import { EditBase, Title, SimpleForm, TextInput, SelectInput } from 'b2b-framework';
+import { Edit, SimpleForm, TextInput, SelectInput } from 'b2b-framework';

 export const BookEdit = () => (
-    <EditBase>
-        <div>
-            <Title title="Book Edition" />
-            <Card>
+    <Edit>
         <SimpleForm>
         <TextInput source="title" />
         <TextInput source="author" />
         <SelectInput source="availability" choices={[
             { id: "in_stock", name: "In stock" },
             { id: "out_of_stock", name: "Out of stock" },
             { id: "out_of_print", name: "Out of print" },
         ]} />
         </SimpleForm>
-            </Card>
-        </div>
-    </EditBase>
+    </Edit>
 );
```

I to wszystko. Pozostały kod to czysta logika biznesowa (układ formularza, który wprowadza dane do renderowania, wraz z ich nazwami i opcjami). Nie ma już nic do usunięcia.

Ramy od pierwszych zasad

Proces wyjaśniony w tym artykule jest dokładnie tym, co doprowadziło nas do zbudowania reakcji admin. Pokazuje, że w ramach nie ma magii - wynika to z pierwszych zasad. W rzeczywistości wystarczy zastąpić `b2b-framework`w `react-admin`powyższym kodzie, a aplikacja będzie działać bezproblemowo.

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

export const BookEdit = () => (
    <Edit>
        <SimpleForm>
            <TextInput source="title" />
            <TextInput source="author" />
            <SelectInput
                source="availability"
                choices={[
                    { id: 'in_stock', name: 'In stock' },
                    { id: 'out_of_stock', name: 'Out of stock' },
                    { id: 'out_of_print', name: 'Out of print' },
                ]}
            />
        </SimpleForm>
    </Edit>
);
```

Wszystkie niestandardowe haki i komponenty ujawnione w tym artykule są w rzeczywistości eksportowane przez React-admin. Kliknij ich nazwę, aby zobaczyć ich dokumentację:

- useGetOne
- useUpdate
- <Form>
- <SimpleForm>
- <TextInput>
- <SelectInput>
- <EditContextProvider>
- useEditController
- <EditBase>
- <Edit>

Przykład React miał ponad 90 linii kodu. Po refaktoryzacji za pomocą React-admin kod jest zwięzły (20 linii kodu), wyrazisty i łatwiejszy w utrzymaniu. Jest deklaratywny i ukrywa ciężar obsługi stanu:

React-admin wykorzystuje 2/3 technik z *Out of the Tar Pit* (programowanie funkcjonalne i programowanie deklaratywne - nie używamy programowania obiektowego). Ostatni przykład kodu pokazuje, jak skuteczne są te techniki w zmniejszaniu złożoności i jak pomagają programistom skupić się na logice biznesowej.

Ale szczerze mówiąc, do wersji 4 kod React-admin nie był wystarczająco czysty, aby umożliwić pojawienie się tej architektury. Komponenty były bardziej powiązane, a ich obowiązki mniej rozdzielone. Nie dałoby się napisać takiego artykułu przy pomocy React-admin v3.

Zbuduj swój własny framework

W tym artykule pokazano również `<Edit>` i `<SimpleForm>` używamy komponentów niższego poziomu (`<EditBase>`, `<Form>`) i zaczepów (`useEditController`, `useForm`), których można również użyć, jeśli potrzebujesz tylko częściowej funkcjonalności. React-admin v4 zaprojektowaliśmy tak, aby był bardzo podatny na ataki hakerów i umożliwiał programistom zastąpienie dowolnej części własnym kodem.

W rzeczywistości możesz bardzo dobrze zbudować swój własny framework, używając tylko haków React-admin i innej biblioteki interfejsu użytkownika. Zorganizowaliśmy kod, aby było to możliwe: większość podpięć znajduje się w podpakietu o nazwie `ra-core`, a opakowania MUI znajdują się w podpakietu o nazwie `ra-ui-materialui`.

Dzięki czystszej architekturze, **React-admin v4 to framework, który sam stworzyłbyś** , gdybyś wielokrotnie tworzył złożone aplikacje B2B – i gdybyś miał dużo czasu!

Ponieważ rzeczywiste hooki i komponenty React-admin pojawiające się w końcowym kodzie (`<Edit>`, `<SimpleForm>`, `<TextInput>`, `<SelectInput>`) robią znacznie więcej niż to, co robił kod początkowy. Zajmują się przypadkami błędów, walidacją po stronie klienta i serwera, powiadomieniami o błędach, wariantami interfejsu użytkownika, i18n, tworzeniem motywów, uwierzytelnianiem itp.

Wniosek

Kiedy programiści odkrywają response-admin, czasami mówią:

> *Nie potrzebuję złożoności tego frameworka. Wolałbym robić rzeczy w czystym React.*

W tym artykule starałem się pokazać, że React-admin *usuwa* złożoność kodu aplikacji.

Upraszczanie rzeczy jest trudne - dlatego reakcja-admin v4 powstawała przez 6 miesięcy. Bardziej niż jakakolwiek poprzednia wersja, React-admin v4 pozwala być bardzo produktywnym, bez marnowania czasu na problemy, które zostały już rozwiązane przez innych.