Deep Instinct - WebCenter

React Admin Documentation

Pedro Rodriguez

Front End Developer, Xprt.

October, 2020

Index

Deep Instinct Web Center

Before starting, it would be great to know a little bit about the company who owns this product; Deep Instinct, is a company that applies deep learning to cybersecurity. They use Deep learning in their products, this lets their product the ability of learning to identify an object, its identification becomes second nature. Similarly, as Deep Instinct's artificial brain learns to detect any type of cyber threat, its prediction capabilities become instinctive. As a result, zero-day and APT attacks are detected and prevented in real-time with unmatched accuracy. Deep Instinct brings a different approach to cybersecurity that is proactive and predictive.

But calm down, their product is already built. What Deep Instinct wants to implement is just a high-level admin application, which can create, edit, search, preview and delete certain types of resources, Essentially, **CRUD** applications. Just an application where they can have all stored and well-managed.

So don't worry, this is not something from another world as long as you have experience, or just even familiarized with the Project's tech stack, you'll get used to it really quickly.

Web Center TechStack

Currently we use **Docker** as a tool to help us set up our development environment quickly, which includes database, frontend and backend server in one whole docker-compose. Web Center contains **MySQL 8.0** as a database, and it runs a server written in **Python 3.7,** with **Flask** as framework, which works flawlessly as APIs management; but again this is not FrontEnd developers concerns, just to have an idea of how is the whole application structured.

Now, as a Front End tools we have:

- **React JS v16.13**; this is our base framework of development, everything is built with react, so if you don't have much experience with it I'll suggest starting learning about it.
- **React-Admin v3.3.1:** This is a tool of development based on React which helps us to create a simple **CRUD** admin web page. This Framework has several features and components that can be reused in the whole app.
- **Material-UI v4.9.5:** React Admin includes Material UI library which is basically the style of the app (this is similar to Bootstrap but cuter), it contains Material Design which uses more grid-based layouts, responsive animations and transitions, padding, and depth effects such as lighting and shadows, giving a really good looking application interface. You can see more of Material UI in its documentation.

We manage version control using Git, more specifically **Github** as a git repository. Here's the repository link where all commits should be. We have a few standards to follow in order to make version management even easier, but we'll see that later in this document.

So that 's it! If you have git, docker and those front end tools knowledge you're well prepared for this project, from now on this document will contain some help regarding problems or doubts that I've passed through in this development process. So, **Good Luck!**

React in Project Implementation

It is important to clarify some standards we use in this project, we use something called **Stateless Components**, which is basically a component that is just a plain javascript function which takes props as an argument and returns a react element. It's important to make this clarification because this is a totally different approach than a **Class Component**.

Stateless components are a long way more efficient than class components, so we need to say goodbye to functionalities like it would be componentDidMount, componentDidUpdate, or componentWillUnmount; instead of that we are gonna use **Hooks**.

Stateless Components

A stateless component has no state(obvious, isn't it?), it means that you can't reach `this.state` inside it. It also has no lifecycle so you can't use componentDidMount and other hooks. The literal difference is that one has state, and the other doesn't. That means the stateful components are keeping track of changing data, while stateless components print out what is given to them via props, or they always render the same thing. Next you can see the difference between stateful and stateless components.

Stateful/Container/Smart component:

```
class Main extends Component {
 constructor() {
   super()
   this.state = {
     books: []
   }
 }
 render() {
   <BooksList books={this.state.books} />
 }
}
```

Stateless/Presentational/Dumb component:

```
const BooksList = ({books}) => {
 return (
   <ul>
     {books.map(book => {
       return <li>book</li>
     })}
   </ul>
 )
}
```

Notice the stateless component is written as a function. As cool as state is, you should always aim to make your components as simple and stateless as possible, so different components can be reused easily, even if you don't have immediate plans to reuse a component.

As you can see, the stateless component receives books as props, so that's what we're printing, we don't have any other function inside BookList, just a **return** statement which will be similar to a **render** call.

Function Expression

So, as we don't have functions inside these stateless components, **we must use function expression** instead of function declaration; this is just assigning a function to a constant or a variable. Just like this:

A function declaration, that is:

```
function add( x, y ) { return x + y; }

console.log( add( 1, 2 ) ); // 3
```

A function expression, that is:

```
const add = ( x, y ) => { x + y };

console.log( add( 1, 2 ) ); // 3
```

Using const to define functions comes with some great advantages that make it superior:

- It makes the function immutable, so you don't have to worry about that function being changed by some other piece of code.
- You can use fat arrow syntax, which is shorter & cleaner.
- Using arrow functions takes care of this binding for you.

Let's remember that javascript is synchronous so if we want to make an function expression to be asynchronous, we need to specify it in the props of the constant, using async and await, just like this example:

```
const updateData = async updatedData => {
   const data = await dataProvider.update(props.resource, {
      data: updatedData,
      id: updatedData.id
   })
   if ( data ) {
      printData(data)
   }
}
```

Props and Parameters

It's important to understand how the props are passed in these stateless components. For this we'll see an example of how to use parameters, first shown in the current component and then how are passed in the parent component:

In this case, we are gonna receive 3 specific parameters from the parent, they are called, fromEdit, onCancel and isModal, and depending on which values are passed from the parent it's gonna print a different content. The ...props value will contain every other piece of parameter that we don't have in count or there's no need to specify.

```
const ActionForm = ({ fromEdit, onCancel, isModal, ...props }) => {

    console.log(fromEdit)   //this should print a bool value

    console.log(onCancel)   //this should print a Function Expression

   console.log(isModal)   //this should print a bool value

... }
```

And in the parent we should pass the parameter just like this:

```
return (

   <Edit {...props} successMessage={successMessage} undoable={false}>

      <ActionForm fromEdit onCancel={onCancel} isModal={false} />

   </Edit>

)
```

As you can see, the fromEdit parameter is passed without value, when you set this it means you're passing the parameter as True, onCancel we have a Function and isModal we set it as false.

We can set all the data types as parameters, we can pass arrays, strings, numbers, booleans, functions and even Objects. But what would happen if I assign the wrong value to a parameter, let's say set True to a number parameter, here is when propTypes and defaultProps come to action.

PropTypes and DefaultProps

PropTypes are a mechanism to ensure that components use the correct data type and pass the right data, and that components use the right type of props, and that receiving components receive the right type of props.

In the section above, we saw how to pass information to any component using props. We passed props directly as an attribute to the component. But we didn't check what type of values we are getting in our component through props or that everything still works.

To use PropTypes, we first need to import PropTypes from the prop-types package:

```
import PropTypes from 'prop-types';
```

Let's see how we can add ProTypes in our previous ActionForm component. Also, PropTypes are useful in catching bugs. And we can enforce passing props by using **isRequired**, if the attribute is not needed and a value can be set as default then we just simply don't add it:

```
ActionForm.propTypes = {
    fromEdit: PropTypes.bool,
    isModal: PropTypes.bool,
    resource: PropTypes.string.isRequired,
    onCancel: PropTypes.func.isRequired
}
```

So with these we've secured and specified which value type are gonna have our props, PropTypes have a lot of validators. Here are some of the most common ones:

```
Component.proptypes = {
   stringProp: PropTypes.string,        // The prop should be a string
   numberProp: PropTypes.number,        // The prop should be a number
   anyProp: PropTypes.any,              // The prop can be of any data type
   booleanProp: PropTypes.bool,         // The prop should be a function
   functionProp: PropTypes.func,        // The prop should be a function
   arrayProp: PropTypes.array           // The prop should be an array
}
```

If we want to pass some default information to our components using props, React allows us to do so with something called **defaultProps**. In cases where PropTypes are optional (that is, they are **not using isRequired**), we can set defaultProps. Default props ensure that props have a value, in case nothing gets passed. Here is an example with our previous dataSet:

```
ActionForm.defaultProps = {

    fromEdit: false,

    isModal2: false,

    onCancel: () => {},

    resource: ""

}
```

This prevents any error when no prop is passed. I advise you always to use defaultProps for every optional PropType.

Stateless Components Hooks

Hooks are a new addition to React 16.8. They allow us to use state and other React features without writing a class. This is why it is so important to explain how to use it in this project. So imagine you have stateless Components, since you don't have a Class you cannot create a state like we used to. To do that we must start using **Hooks.**

React Hooks were invented by the React team to introduce state management and side-effects in function components. It's their way of making it more effortless to use only React function components without the need to refactor a React function component to a React class component for using lifecycle methods, in order to use have side-effects, or local state. React Hooks enable us to write React applications with only function components. Thus, there is no need to use class components anymore.

What is a Hook? A Hook is a special function that lets you “hook into” React features. For example, **useState** is a Hook that lets you add React state to function components. We’ll learn other Hooks later.

When would I use a Hook? If you write a function component and realize you need to add some state to it, previously you had to convert it to a class. Now you can use a Hook inside the existing function component. We're going to do that right now!

Hook: useState

In order to use states in this project we need to use useState Hook in order to make them work, since we are using Stateless Components. You can see here the exact same equivalent Class and Stateless components:

Class Component

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

As you can see, in the previous piece of code, we had a state called count, if we want to show it we need to reach it via: this.state.count; if we want to set the value we need to use this.setState({ count: newValue }). Note the differences between Classes and Stateless Components.

Equivalent Stateless Component:

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

In a class, we initialize the count state to 0 by setting this.state to { count: 0 } in the constructor. In a stateless component, we have no `this`, so we can't assign or read `this.state`. Instead, we call the `useState` Hook directly inside our component.

What does calling useState do? It declares a "state variable". Our variable is called count but we could call it anything else. This is a way to "preserve" some values between the function calls — **`useState` is a new way to use the exact same capabilities that `this.state` provides in a class**.

What does useState return? It returns a pair of values: the current state and a function that updates it. This is why we write `const [count, setCount] = useState()`. This is similar to this.state.count and this.setState in a class, except you get them in a pair.

Note: In Web Center Application we only implement `useState` Hooks, there's no `this.state` in the code.

Hook: useEffect

The Hook `useEffect` is - as the name suggests - that performs arbitrary side effects during the life of a component. **It is basically a hook replacement for** the "old-school" lifecycle methods **componentDidMount, componentDidUpdate and componentWillUnmount**. It allows you to execute lifecycle tasks without a need for a class component. So you can now make side effects inside a functional component. This was not possible before, because creating side effects directly in a render method (or a body of a functional component) is strictly prohibited. Mainly because we don't really control (and shouldn't really think about) how many times the

`render` function will be called. Now let's see an equivalent of using the hook vs using class component:

Class Component

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

Note how we have to duplicate the code between these two lifecycle methods in class.

This is because in many cases we want to perform the same side effect regardless of whether the component just mounted, or if it has been updated. Conceptually, we want it to happen after every render — but React class components don't have a method like this. We could extract a separate method but we would still have to call it in two places.

Now let's see how we can do the same with the `useEffect` Hook.

```jsx
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

What does useEffect do? By using this Hook, you tell React that your component needs to do something after render. React will remember the function you passed (we'll refer to it as our "effect"), and call it later after performing the DOM updates. In this effect, we set the document title, but we could also perform data fetching or call some other imperative API.

Does useEffect run after every render? Yes! By default, it runs both after the first render and after every update. Instead of thinking in terms of "mounting" and "updating", you might find it easier to think that effects happen "after render". React guarantees the DOM has been updated by the time it runs the effects.

Tip: Optimizing Performance by Skipping Effects

Now imagine you have a lot of states, and the use effect will run each time any state is updated, we will face a performance issue there. In class components, we can solve this by writing an extra comparison with prevProps or prevState inside componentDidUpdate:

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

This requirement is common enough that it is built into the `useEffect` Hook API. You can tell React to skip applying an effect if certain values haven't changed between re-renders. To do so, pass an array as an optional second argument to `useEffect`:

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

In the example above, we pass [count] as the second argument. What does this mean? If the count is 5, and then our component re-renders with count still equal to 5, React will compare [5] from the previous render and [5] from the next render. Because all items in the array are the same (5 === 5), React would skip the effect. **That's our optimization.**

When we render with `count` updated to `6`, React will compare the items in the `[5]` array from the previous render to items in the `[6]` array from the next render. This time, React will re-apply the effect because `5 !== 6`. If there are multiple items in the array, React will re-run the effect even if just one of them is different.

Well that completes everything you need to know, to handle the React basics part inside the Deep Instinct Web Center. Once you learnt all of these concepts you'll be ready for the next tech stack required: **React Admin. Let 's do it!**

React Admin

For this project we implemented React Admin as a tool to speed up the front end app. It's a frontend Framework for building B2B (business-to-business) applications running in the browser on top of REST/GraphQL APIs, using ES6, React and Material Design. In other words, it is an easy-to-use React.js framework with the goal of cutting down on the time required to deploy an admin dashboard for your application.

This framework provides a variety of out of the box components that allow one to build a fully functional admin panel in little time. The only problem is that, in React Admin is a little difficult to customize every single piece of functionality, if it is not a simple CSS change, but a whole new functionality you want to add to a React Admin component, it would be hard or nearly impossible to customize the component.

In this document we're going to review all topics, functionalities and components you'll need to be aware of.

sData Providers

React-admin can communicate with any API, whether it uses REST, GraphQL, or even SOAP, regardless of the dialect it uses. For REST servers, it can be JSON API, HAL, OData or a custom dialect. The only thing react-admin needs is a **Data Provider function**. This is the place to translate data queries to HTTP requests, and HTTP responses to data responses.

In our code we have a custom Data Provider, since there are several different APIs to cover, but every Data Provider must have at least the following methods:

```
const dataProvider = {

    getList: (resource, params) => Promise,

    getOne: (resource, params) => Promise,

    getMany: (resource, params) => Promise,

    getManyReference: (resource, params) => Promise,
```

```
    create: (resource, params) => Promise,

    update: (resource, params) => Promise,

    updateMany: (resource, params) => Promise,

    delete: (resource, params) => Promise,

    deleteMany: (resource, params) => Promise,

};
```

These are the basic data providers that React Admin needs, so it is needed to have them all set up. For instance, all these providers should have a counterpart in the server, so the client only can reach it. Next, a piece of out data provider will be shown, to see how a call should be structured.

Our data provider file is located in `./providers/dataProvider.js`, let 's see how our `getList` is structured:

```
 getList: async (resource, params) => {

     try {

         const { data } = await api.get(`${resource}/`, {

             params: parseParams(params)

         })

         return data

     } catch (error) {

         catchError(error)

     }

 },
```

The first thing we need to visualize is that this function is **asynchronous**, so it can wait to execute the request and return the response data. Second we receive two parameters, **`resource`** which should be the name of the resource we are going to get the list, f.e. `"users"`, and **`params`** which are basically parameters that are going to change how the data will be presented, f.e. could be **JSON** that contains fields like `page, perPage, filter, order`.

Another thing we visualize is the api element, which is basically an `axios` that help us to do requests to the server, in the code below you can see the structure of the const `api`; `config.apiBaseUrl` is basically the base url of the server APIs routes.

```
const api = axios.create({

        baseURL: config.apiBaseUrl

})
```

Root Component

Now that we have a main idea of what data providers are, we can start with visual and core components of React Admin. First of all we'll need to know how the root component is composed. The root components is the `App.js` file located in the root of the project

In the picture below there are several aspects we need to be aware of, I'll tell you what mean every single one of them:

- **Admin Component**:The `<Admin>` component creates an application with its own state, routing, and controller logic. `<Admin>` requires only a **dataProvider** prop, and at least one child `<Resource>` to work, like shown below. Also, we see multiple props inside this component, let's explain some of them. (You can see all the props available here)
    - **Data and Auth provider**: this are basically two different providers, one dedicated to the credentials requests, and the other to the general resources requests.
    - **Dashboard**: By default, the homepage of an admin app is the list of the first child `<Resource>`. But we specify it as a custom component instead. More info here.
    - **LoginPage**: If you want to customize the Login page, or switch to another authentication strategy than a username/password form, pass a component of

your own as the loginPage prop. React-admin will display this component whenever the /login route is called. You can also disable it completely along with the /login route by passing false to this prop.

- **Layout**: Which is basically the static design is gonna have the entire header application, for example AppBar, sideBar or even quick settings menu; we'll deep into that right away.

- **FileResource**: A <Resource> component maps one API endpoint to a CRUD interface. If you have multiple resources that you want to include as part of the CRUD interface, then add it as a sibling of the previous resource component (FileResource in this case).

```
import React from "react"
import { Admin, Resource } from "react-admin"
import FileResource from "components/file"
import customDataProvider from "providers/dataProvider"
import Dashboard from "components/dashboard"
import CustomLayout from "components/shared/Layout"
import MenuItemIcon from "@material-ui/icons/FiberManualRecord"

const dataProvider = customDataProvider()

const App = () => (
  <Admin
    dataProvider={dataProvider}
    authProvider={authProvider}      You, a few seconds ago • Un
    loginPage={LoginPage}
    locale="en"
    dashboard={Dashboard}
    layout={CustomLayout}
  >
    <FileResource
      name="file"
      icon={MenuItemIcon}
      options={{ label: "Files" }}
    />
  </Admin>
)

export default App
```

Layout Component

If we want to deeply customize the app header, the menu, or the notifications, the best way is to provide a custom layout component. It must contain a {children} placeholder, where react-admin will render the resources. If you use material UI fields and inputs, it should contain a <ThemeProvider> element. And finally, if you want to show the spinner in the app header when the app fetches data in the background, the Layout should connect to the redux store.

Your custom layout can simply extend the default <Layout> component if you only want to override the appBar, the menu, the notification component, or the error page. For instance:

```
// in src/MyLayout.js
import { Layout } from 'react-admin';
import MyAppBar from './MyAppBar';
import MyMenu from './MyMenu';
import MyNotification from './MyNotification';

const MyLayout = (props) => <Layout
    {...props}
    appBar={MyAppBar}
    menu={MyMenu}
    notification={MyNotification}
/>;

export default MyLayout;
```

As we can see, our base component will be `{ Layout } from 'react-admin'`, so we declare MyLayout which will be our custom Layout component, and pass our components as props, we pass appBar, menu and notification.

Also is important to recall that the `{...props}` attribute should be **before** our custom components, if not, this `{...props}` elements will substitute them with the default ones.

Theming

Now, one of the most important things we need to understand when learning how to use React Admin is how to theme or customize CSS of certains components. Whether you need to adjust a CSS rule for a single component, or change the color of the labels in the entire app, **you're covered!**

Hook: makeStyles

React-admin v3 uses material-ui v4+, and that new version offers a hook-based alternative called **`makeStyles`**, which returns a hook to be used in a component at runtime:

```
import { makeStyles } from '@material-ui/core/styles';

const useStyles = makeStyles({
  button: {
    fontWeight: 'bold'
  },
  buttonColorGreen: {
    color: 'green'
  },
});

const MyEditButton = props => {
    const classes = useStyles();
    return <EditButton className={classes.button} {...props} />;
};
```

In the previous piece of code we can visualize how we should implement the makeStyle hook, it is necessary to create like class names, that would be `button` and `buttonColorGreen` f.e. Inside the class name we specify which CSS elements should be change, please note how the attributes are **not exactly the same as CSS**, for example we have `fontWeight` which is the same as `font-weight` in CSS; all the attributes are the same but erasing the "-" char and capitalizing each word.

Now, what happens when we want to change the CSS attributes of an existing class element? Simple, we **override** it, first of all we need to inspect the element in the browser to see whats the name of the class assigned, after that we write it down in our class names just like this:

As you can see in the previous picture, we added a new element to our class name: `'& .MuiButton-label'` which is basically the class assigned to the CSS by React Admin. What makes this class overridable is the **&** char before the **selector**.

Last but not least, let's say you want to add **multiple classes** to a ClassName prop. Easy just need to write the classes with the following syntax:

```
const MyEditButton = props => {
    const classes = useStyles();
    return <EditButton className={`${classes.button} ${otherClassName}`} {...props} />;
};
```

And **that's it**, you're ready to start customizing and themeing the Web Center.

ReferenceField Component

`<ReferenceField>` is useful for displaying many-to-one and one-to-one relationships. This component fetches a referenced record (using the dataProvider.getMany() method), and passes it to its child. A <ReferenceField> displays nothing on its own, it just fetches the data and expects its child to render it. Usual child components for <ReferenceField> are other <Field> components.

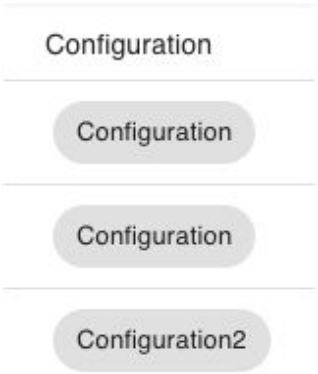
These reference fields are commonly used to get another resource data, besides the already one shown. For instance, if a `post` has one author from the `users` resource, referenced by a `user_id` field, here is how to fetch the user related to each `post` record in a list, and display the name for each:

```
import * as React from "react";
import { List, Datagrid, ReferenceField, TextField, EditButton } from 'react-admin';

export const PostList = (props) => (
    <List {...props}>
        <Datagrid>
            <TextField source="id" />
            <ReferenceField label="User" source="user_id" reference="users">
                <TextField source="name" />
            </ReferenceField>
            <TextField source="title" />
            <EditButton />
        </Datagrid>
    </List>
);
```

So when there is a Reference Field, all the components inside of it will refer to the resource specified, and not the main resource. We have plenty of examples in our project, on of the most used ReferenceFields are the reference chips:

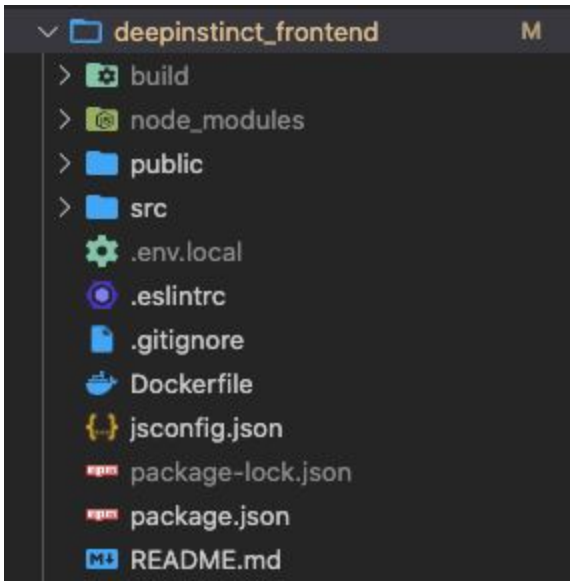
```
<ReferenceField
  label="Configuration"
  source="configuration"
  reference="configurations"
>
  <ConfigurationField source="name" />
</ReferenceField>
```

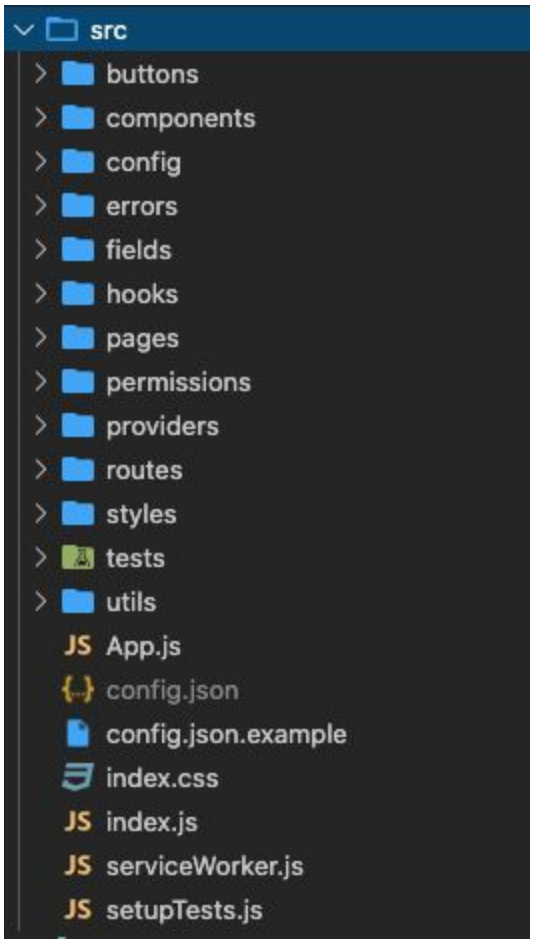


As you can see in our previous code, in order to get the name of a reference, we needed to use `ReferenceField`, here we specify the `source` name which is how it's called in the other Resource, and reference it should be to which resource is being referenced, in this case `configurations`.

Project Structure

If you are starting to get to know the project, one of the things that will help you a lot will be to know the structure of the project, where everything is located, would save you a lot of time. The general overview of the project's structure will look like this:





As you can see, we have a common react application structure, in the **public folder** we will have some local files that are going to be reachable to be shown on the page, these will be favicon, logos and other kinds of reachable file we want to add.

We'll have our **docker file** which contains necessary info to run the application, we also have an **eslintrc file** which basically contains rules or standards to follow in the code.

Package.json is also found, this contains the libraries necessary to run the application. These are really basic files needed in every react app, so we are going to focus on which files are we going to find the **src folder**, where is basically **all the code**.

App.js and Index.js

App.js is one of the most important files in the structure, here's where we describe the Admin component (which is the root of react admin application); inside this we specify all resources we want to treat as CRUD elements. We've specified this file before in this documentation, just give it a check here.

Index.js on the other hand, is the base file, which only contains the App component exported from App.js. That 's it!

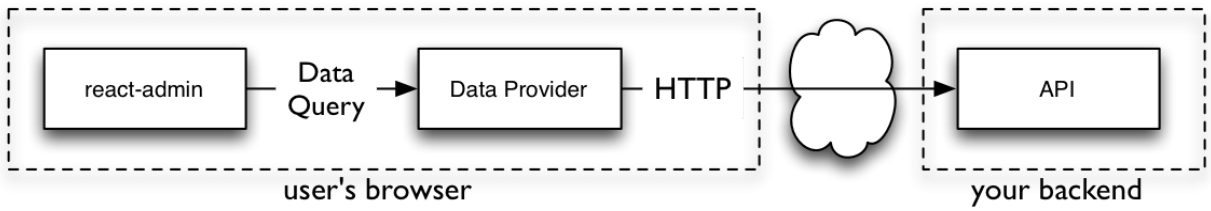
Config.json

Our config.json file is basically where we are going to set out environment variables. For example in the picture on the side we can specify our base url to do requests to the server depending on which environment we are in.

```
{
  "development": {
    "apiBaseUrl": "http://127.0.0.1:5000/api"
  },
  "production": {
    "apiBaseUrl": "http://127.0.0.1:5000/api"
  }
}
```

src / Providers

We've explained this before, in this folder we are going to store our providers, which are basically in charge of **sending and receiving all the data requests** to the servers through APIs. Here's a little scheme in order to explain it better:



In this folder there will be at least two files, one called **DataProvider.js** and the other **authProvider.js**. DataProvider will be managing ALL the common requests to the API's, including each CRUD functionality, we talk more about it here.

AuthProvider

React-admin lets you secure your admin app with the authentication strategy of your choice. Since there are many possible strategies (Basic Auth, **JWT**, OAuth, etc.), react-admin delegates authentication logic to our authProvider, and provides hooks to execute your authentication code. It's important to recall that by default, react-admin apps don't require authentication. To restrict access to the admin, pass an `authProvider` to the `<Admin>` component.

Just like a dataProvider, an authProvider is an object that handles authentication logic. It exposes methods that react-admin calls when needed, and that return a Promise. Once an admin has an **authProvider**, react-admin enables a new page on the **`/login` route**, which displays a login form asking for a username and password.

Upon submission, this form calls the `authProvider.login({ login, password })` method. It's the ideal place to authenticate the user, and store their credentials. For example, to query an authentication route via HTTPS and store the credentials (an Access Token) in local storage.

You can see more about AuthProviders in the Documentation.

Web Center AuthProvider

Our Web Center application login process consists of 2 steps, first we show the basic login screen called LoginForm; which contains user and password fields. As you can see in the code above, after logging in user and password data, it loads a new page called TwoFactorAuthForm; which will contain a Two-Factor Authentication; which only receives the 2FA code and send it to the server and this verifies if its correct; if is then redirects you to the dashboard, if it doesn't then returns a error message.

```
<div>
  {authTempCode ? (
    <TwoFactorAuthForm authTempCode={authTempCode} />
  ) : (
    <LoginForm setAuthTempCode={setAuthTempCode} />
  )}
  <Notification />
</div>
```

So let's see how we manage this data:

The AuthProvider changes are going to be seen not in LoginForm but in TwoFactorAuthForm. Inside LoginForm we can see how we send data directly to the server without using the dataProvider or AuthProvider:

```
const submit = async ({ username, password }) => {
  // Send a login request to the server
  const request = new Request(`${config.apiBaseUrl}/authentication/login`, {
    credentials: "include",
    method: "POST",
    body: new URLSearchParams({ username, password }),
    headers: new Headers({
      "Content-Type": "application/x-www-form-urlencoded"
    })
  })
```

After verifying the basic login data (user and password), then we proceed to execute the Two-Factor Authentication process. In TwoFactorAuthForm we can see we use another React Admin Hook called **useLogin** which basically maps our Login Function from our AuthProvider.

```
const login = useLogin()
const redirectTo = useRedirect()

const [submittedOnce, setSubmittedOnce] = useState(false)
    Shachar Langer, 7 months ago • Authentication initial commit
const onSubmit = async ({ otp }) => {
  // Trying to login. If login fails, return the error
  const errorObj = await login({ otp, authTempCode }).catch(error => {
    return error
  })
```

As you can see in both of these pictures, **`useLogin()`** has the same parameters as our Login custom function in AuthProvider. So the function Login actually contains the Two-Factor Auth and verifies if it's correct on the server-side.

```
// called when the user attempts to log in
login: async ({ otp, authTempCode }) => {
  const request = new Request(
    `${config.apiBaseUrl}/authentication/twofactor`,
    {
      credentials: "include",
      method: "POST",
      body: new URLSearchParams({
        otp_code: otp,
        auth_temp_code: authTempCode
      }),
      headers: new Headers({
        "Content-Type": "application/x-www-form-urlencoded"
      })
    }
  )
```

src / Routes

The routes folder, like its name says, contain all the custom routes that needed to be added manually. Let's remember that React Admin covers all the routes by his own, but there are cases that we need to specify some route. For example, we added extra routes like these ones.

```
export default [
  // <Route exact path="/login" component={LoginPage} noLayout />,
  <Route exact path="/password_reset" component={ResetPasswordPage} noLayout />,
  <Route exact path="/error" component={ErrorPage} noLayout />,
  <Route exact path="/logs" component={LogsPage} />,
  <Route exact path="/import" component={ImportPage} />
]
```

If you want to know how these custom routes work check the Documentation.

src / Styles

Here are saved all our custom stylesheets, let's remember that we cannot add pure CSS with react admin, so we need makeStyles hook in order to create our styles. So this is gonna be our stylesheet folder.

src / Utils

This is a really varied mix of files; This folder contains **several tools, functions and data** that are going to be **reused** in our entire project. First we face **config.js**, here we are going to set which environment will be running our application, by default it will be in development.

Then, we have **constants.js**, this is a really helpful file, it contains several const elements, and inside them we set multiple kinds of data. For example, which attribute has each resource, trust me, this is gonna be useful later.

Exception.js as it names says contains exceptions. Then we have **Logger.js**, this is basically like a debugger for React Admin; it **prints error messages** when they find any, also this is set to development environment only.

Another important file is **helpers.js**; inside this file it **contains several of our own functions**, functions that can be reused in another piece of code. It's really helpful and we **don't** need to **rewrite existing functions.**

And last but not least, we have **messages.js**, where we specify some **standard messages to be shown**; for example when creating any resource we can retrieve a message from here, like "this resource was created successfully".

src / Hooks

Like we saw before, React Admin uses several hooks in order to maintain lifecycle methods, state management and side-effects in Stateless Components. For example, useState is a React Admin hook that helps us to manage states inside the application. This is exactly what we are going to store in this folder, **custom Hooks**.

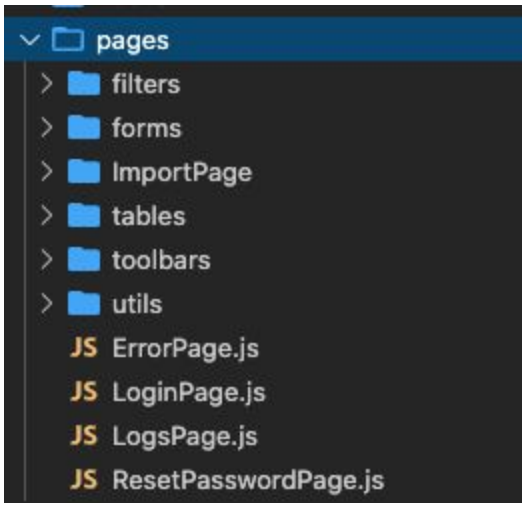
In this folder we have one hook called **useChoices**. This Hook basically helps us to load data to where we need choices, like probably dropdowns and autocompletes, from resources that are not specified in the `<Admin>` component. This is commonly used to load resources like types, OS types, file type and those complementary resources without the need of declaring them in the `<Admin>` component.

If you want to know how Hook's behavior is and how to create any you can check the Documentation.

src / Permissions

In Web Center we manage multiple kinds of users, and each user has certain permissions; this folder contains a **Permissions file**. This file specifies which roles are going to be defined on each user and what power will have over each resource. In our application we have 4 types of role : viewer, editor, super_editor and admin; depending on which role, the user will determine which actions he can execute in the application.

src / Pages

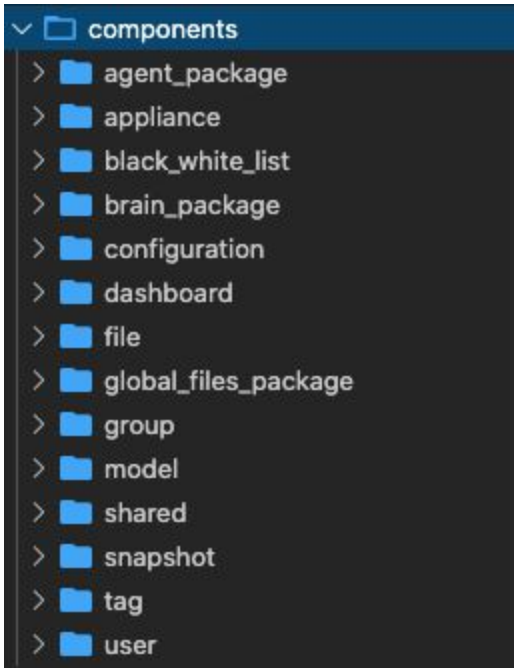


Inside our pages folder we will have our individual pages that are not contained by default in React Admin. For example, we have our LogsPage.js which contains all functionalities regardings logs. These are here because are not just individual components, they are whole pages.

The Login and ResetPassword page are here too, if you want to customize them here is the place.

The other folders are simply components and functions that these pages use; like toolbars, forms, tables and filters, which are components; utils which are functionalities and last but not least importPage which is just another page but inside a folder.

src / Components



This will be our main folder, here are all major components of our application. We can understand **Components** as the building blocks of any React app and a typical React app will have many of these. Simply put, a component is a JavaScript class or function that optionally accepts inputs i.e. properties(props) and returns a React element that describes how a section of the UI (User Interface) should appear.

As you can see in the picture on the side, there are **ALL** the resources we specify in our `<Admin>` component at `App.js`, inside each of them we can find files like `index.js`, `list.js`, `create.js`, `edit.js`, `toolbar.js`, `filter.js` and `expandCard.js`; which are basically main components that together they build a **CRUD** Interface of each resource.

Besides having our resources CRUD UI components, we'll have a folder called **shared**; this folder contains several various components, in here we basically store our **customized React Admin components**; since they are pretty hard to customize we need to customize their components in order to show a new feature for example. Also, we have our Layout.js file, which will define how it will look on our sidebar, appBar and other main components. You can go back and give it a check here.

src / Fields and src / Buttons

We created the **Fields folder** in order to store our custom inputs, which are based on React Amin inputs but with a lot more features. For example, in here we have TagsInput, which contains multiples React Admin components inside them:

As you can see, the TagsInput Field has a mix of **ArrayAutocomplete** and a + **Create button** that help us to create tags on real time. That's why it is a custom component, because it contains multiple elements inside them. And basically in our **Buttons Folder** we got all of these create buttons including the functionality of creating the elements in real time. That's why these folders work together.

Git Standards & Conventions

When managing version control, we ensure to accomplish certain git standards, either when we are going to do a commit, push, merge or even a PR.

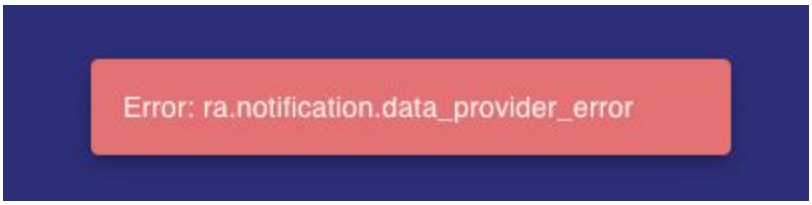
- We **never touch** either **develop** or **master** branch, those are for deployment and main branches.
- If we are going to create a new PR, we should create a new branch, the standard for the branch name is `<your name>/<bug or feature>/<name of the bug or feature>`, also you need to make sure to be up-to-date from develop when creating the branch. This can make identification even easier.
- It is good to **create a PR even when you haven't finished** solving the bug or feature, thanks to this, the PR can have a tracking of which tasks are being done. If it is not done yet you should **add a label** like "Not done yet" to the PR. Also you should add the PR link to the equivalent Trello task.
- When a task is **done**, you should remove the "Not done yet" label and add the "Done" label; also **move the equivalent trello card** to "Pending integration". Also it would be a good idea to inform your PM about the tasks accomplished.

With this information you should be able to manage the standards of version control to this project.

TroubleShooting

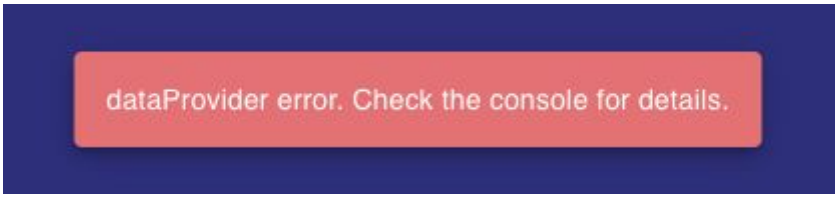
Here we'll see some of the common error we confront alongside our development process:

Error: ra.notification.data_provider_error



This error is commonly shown if we have any errors inside our Dataprovider, since the dataprovider is on layout over the actual web page, if there is an execution error it won't show the specific piece of code that contains the issue and will just show `Error: ra.notification.data_provider_error`, the solution is to debug the Dataprovider in order to check what issue could be happening. Once solved, it will return another message.

dataProvider error. Check the console for details.



This is a common error, this is normally printed when some error is catched inside the try and catch statements, if the page is redirected to the login screen, then the AccessToken is expired, and it would ask the user to re-login.

References

Introducing Hooks, React Documentation. Retrieved from https://reactjs.org/docs/hooks-intro.html

Using the State Hook, React Documentation. Retrieved from https://reactjs.org/docs/hooks-state.html

Using the Effect Hook, React Documentation. Retrieved from https://reactjs.org/docs/hooks-effect.html#tip-optimizing-performance-by-skipping-effects

Getting Started: React Admin. Retrieved from https://teacode.io/blog/getting-started-react-admin

Stateful and Stateless Components in React. Retrieved from https://programmingwithmosh.com/javascript/stateful-stateless-components-react/

React: Class Component vs Functional Component. Retrieved from https://itnext.io/react-component-class-vs-stateless-component-e3797c7d23ab

Mastering Props And PropTypes In React. Retrieved from https://www.smashingmagazine.com/2020/08/mastering-props-proptypes-react/

Understanding React Components. Retrieved from here.