Refine Documentation

Youth Innovation Lab

Prepared By

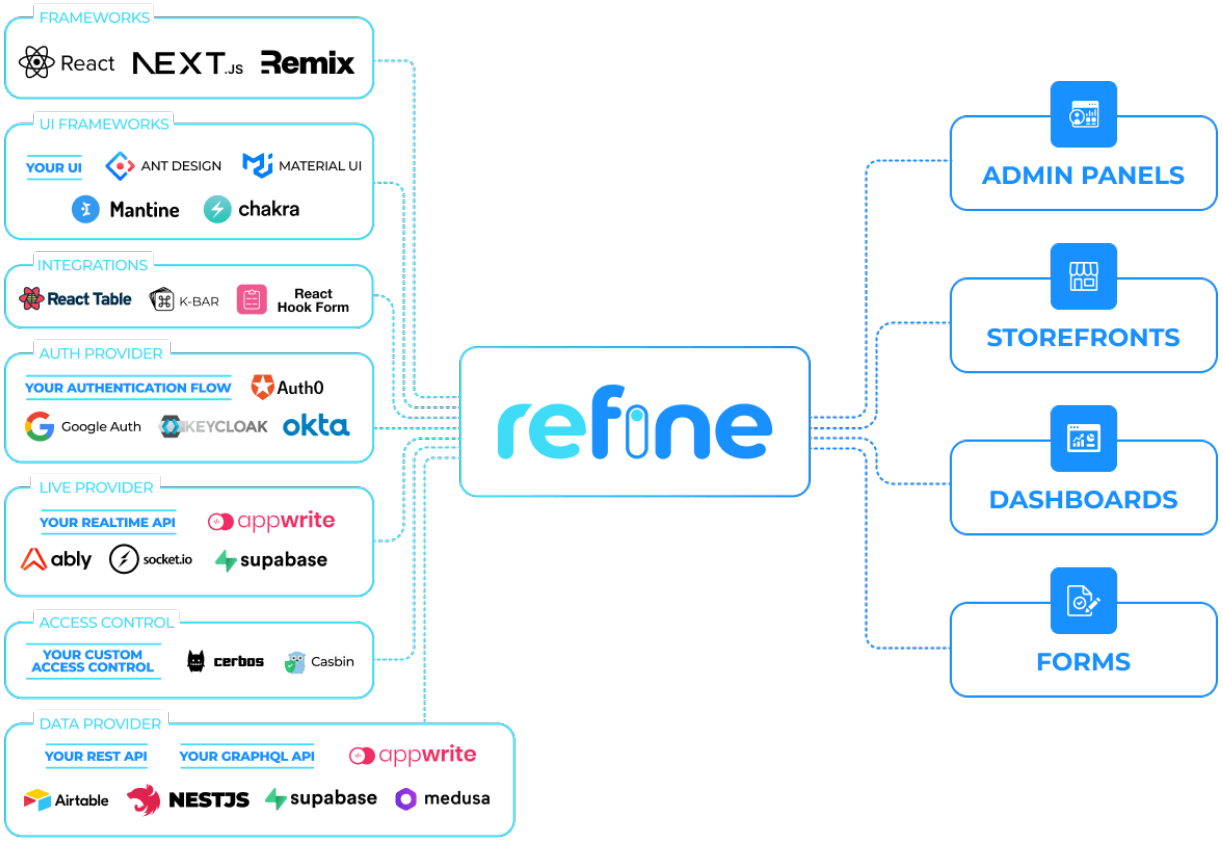
Suraj Karki

What is refine ?

- **Refine** is a React-based framework for the rapid development of web applications.
- **Refine** is *headless by design.*
- **Refine** is a collection of helper **hooks**, **components**, and **providers**. They are all decoupled from your **UI** components and business logic, so they never keep you from customizing your UI or coding your own flow.
- **Refine** seamlessly works with any **custom design** or **UI framework you favor**. For convenience, it ships with ready-made integrations for *Ant Design System*, *Material UI*, *Mantine*, and *Chakra UI*.

Usages

- **Refine** shines on *data-intensive* applications like ***admin panels, dashboards*** and ***internal tools*.**
- It has built-in **SSR** (server-side rendering) support.



Features

- SSR support with Next.js or Remix
- Auto-generated CRUD UIs from your API data structure
- Perfect **state management** & **mutations** with **React Query**
- Advanced routing with any router library of your choice
- Providers for seamless authentication and access control flows
- Out-of-the-box support for live / real-time applications
- Support for any i18n framework

Quick Start Guide

- **Refine** works on any environment you can run **React** (incl. *CRA, Next.js, Remix, Vite* etc.)
- Normally to install refine app with AntDesign UI framework

  ```
  $ npm create refine-app@latest -- -o refine-antd tutorial
  ```

- In this guide we are going to use Vite with React.
- Create a react app using Vite:

  ```
  $ yarn create vite
  ```

- Then install dependencies

  ```
  $ yarn install
  ```

- Install refine base dependencies for Ant Design

  ```
  $ yarn add @pankod/refine-core @pankod/refine-antd @pankod/refine-react-router-v6
  ```

- We have to install **@pankod/refine-simple-res** for now to run the demo app

  ```
  $ yarn add @pankod/refine-simple-res
  ```

- Replace the contents of **App.tsx** with the following code:

```
import { Refine } from "@pankod/refine-core";
import {
    Layout,
    ReadyPage,
    notificationProvider,
    ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";

const App: React.FC = () => {
    return (
        <Refine
            routerProvider={routerProvider}
            dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
            Layout={Layout}
            ReadyPage={ReadyPage}
            notificationProvider={notificationProvider}
            catchAll={<ErrorComponent />}
        />
    );
};

export default App;
```

- Delete all style files, imports from this react app.
- Run the following command to launch the app in development mode:

```
$ yarn run dev
```

On your browser, you should now see a page in localhost. That's it; you've just created an empty refine project in react with vite.

Now, we successfully setup the project and now we can continue our learning journey from here.

1. General Concepts

- Refine core is fully independent of UI. So you can use core components and hooks without any UI dependency.
- All the **data** related hooks**(useTable, useForm, useList** etc.) of **refine** can be given some common properties like **resource, metaData, queryOptions** etc.

- **resource**

  refine passes the **resource** to the **dataProvider** as a params. This parameter is usually used to as a API endpoint path. It all depends on how to handle the **resource** in your **dataProvider**. See the creating a data provider section for an example of how **resource** are handled.

  How does refine know what the resource value is?

  1- The **resource** value is determined from the active route where the component or the hook is used.

  Like below, if you are using the hook in the **<PostList>** component, the **resource** value defaults to "**posts**".

```
import { Refine } from "@pankod/refine-core";
import {
  Layout,
  ReadyPage,
  notificationProvider,
  ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";

const App: React.FC = () => {
  return (
    <Refine
      routerProvider={routerProvider}
      dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
      Layout={Layout}
      ReadyPage={ReadyPage}
      notificationProvider={notificationProvider}
      catchAll={<ErrorComponent />}
      resources={[
        {
          name: "posts",
          list: PostList,
        },
      ]}
    />
  );
};

export default App;
```

2- The resource value is determined from the **resource** prop of the hook.

Here, PostList is a react component that shows a list of posts. Now implement this PostList component.

Creating a **List** page

✔ First, we'll need an interface to work with the data from the API endpoint.

✔ Create a new folder named *"interfaces"* under *"/src"* if you don't already have one. Then create a *"index.d.ts"* file with the following code:

```
export interface IPost {
  id: number;
  title: string;
  status: "published" | "draft" | "rejected";
  createdAt: string;
}
```

✔ We'll be using **title**, **status** and **createdAt** fields of every **post** record.

✔ Now, create a new folder named *"pages/posts"* under *"/src"*. Under that folder, create

```
import {
  List,
  TextField,
  TagField,
  DateField,
  Table,
  useTable,
} from "@pankod/refine-antd";

import { IPost } from "../../interfaces/index";
export const PostList: React.FC = () => {
  const { tableProps } = useTable<IPost>();
  return (
    <List>
      <Table {...tableProps} rowKey="id">
        <Table.Column dataIndex="title" title="Title" />
        <Table.Column
          dataIndex="status"
          title="Status"
          render={(value) => <TagField value={value} />}
        />
        <Table.Column
          dataIndex="createdAt"
          title="CreatedAt"
          render={(value) => <DateField format="LLL" value={value} />}
        />
      </Table>
    </List>
  );
};
```

Let's break down the **<PostList/>** component to understand what's going on here:

- ✔ **<Table/>** is a native **Ant Design** component. It renders records row by row as a table. **<Table/>** expects a **rowKey** prop as the unique key of the records. **useTable<IPost>();** is passed to the **<Table/>** component as **{...tableProps}.**
- ✔ **Refine** hook **useTable()** fetches data from API and wraps them with various helper hooks required for the **<Table/>** component. Data interaction functions like sorting, filtering, and pagination will be instantly available on the **<Table/>** with this single line of code.
- ✔ **refine** depends heavily on hooks and **useTable()** is only one among many others. On useTable() Documentation you may find more information about the usage of this hook.
- ✔ **<Table.Column>** components are used for mapping and formatting each field shown on the **<Table/>. dataIndex** prop maps the field to a matching key from the API response. **render** prop is used to choose the appropriate **Field** component for the given data type.
- ✔ **<List>** is a **refine** component. It acts as a wrapper to **<Table>** to add some extras like *Create Button* and *title*.
- ✔ Open your application in your browser. You will see posts are displayed correctly in a table structure and even the pagination works out-of-the box.

◆ Showing a single record

- ✔ At this point we are able to list all *post* records on the table component with pagination, sorting and filtering functionality. Next, we are going to add a *details page* to fetch and display data from a single record.
- ✔ Let's create a **<PostShow>** component on **/pages/posts** folder by creating a file with name **show.tsx** and put the following code.

```tsx
import { useShow, useOne } from "@pankod/refine-core";
import { Show, Typography, Tag } from "@pankod/refine-antd";

const { Title, Text } = Typography;

export const PostShow = () => {
  const { queryResult } = useShow();
  const { data, isLoading } = queryResult;
  const record = data?.data;

  const { data: categoryData } = useOne({
    resource: "categories",
    id: record?.category.id || "",
    queryOptions: {
      enabled: !!record?.category.id,
    },
  });

  return (
    <Show isLoading={isLoading}>
      <Title level={5}>Title</Title>
      <Text>{record?.title}</Text>

      <Title level={5}>Status</Title>
      <Text>
        <Tag>{record?.status}</Tag>
      </Text>

      <Title level={5}>Category</Title>
      <Text>{categoryData?.data.title}</Text>
    </Show>
  );
};
```

✔ Now we can add the newly created component to our resource with **show** prop:

✔ Modify **App.tsx** to:

```
import { Refine } from "@pankod/refine-core";
import {
    Layout,
    ReadyPage,
    notificationProvider,
    ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";

const App: React.FC = () => {
    return (
        <Refine
            routerProvider={routerProvider}
            dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
            Layout={Layout}
            ReadyPage={ReadyPage}
            notificationProvider={notificationProvider}
            catchAll={<ErrorComponent />}
            resources={[
                {
                    name: "posts",
                    list: PostList,
                    show: PostShow
                },
            ]}
        />
    );
};

export default App;
```

- ✔ And then we can add a **<ShowButton>** on the list page to make it possible for users to navigate to detail pages:
- ✔ Change the **list.tsx** to following

```
import {
  List,
  TagField,
  DateField,
  Table,
  useTable,
  ShowButton,
} from "@pankod/refine-antd";

import { IPost } from "../../interfaces/index";

export const PostList: React.FC = () => {
  const { tableProps } = useTable<IPost>();
  return (
    <List>
      <Table {...tableProps} rowKey="id">
        <Table.Column dataIndex="title" title="Title" />
        <Table.Column
          dataIndex="status"
          title="Status"
          render={(value) => <TagField value={value} />}
        />
        <Table.Column
          dataIndex="createdAt"
          title="CreatedAt"
          render={(value) => <DateField format="LLL" value={value} />}
        />
        <Table.Column<IPost>
          title="Actions"
          dataIndex="actions"
          render={(_text, record): React.ReactNode => {
            return (
              <ShowButton size="small" recordItemId={record.id} hideText />
            );
          }}
        />
      </Table>
    </List>
  );
};
```

- ✔ **useShow()** is a **refine** hook used to fetch a single record of data. The **queryResult** has the response and also **isLoading** state.
- ✔ Refer to the **useShow** documentation for detailed usage information. →
- ✔ To retrieve the category title, we need to make a call to **/categories** endpoint. This

time we used **useOne()** hook to get a single record from another resource.

- ✔ Refer to the **useOne** documentation for detailed usage information. →
- ✔ Since we've got access to raw data returning from **useShow()**, there is no restriction on how it's displayed on your components. If you prefer presenting your content with a nicer wrapper, **refine** provides you the **<Show>** component which has extra features like **list** and **refresh** buttons.
- ✔ Refer to the **<Show>** documentation for detailed usage information. →

◆ Editing a record

- ✔ Until this point, we were basically working with reading operations such as fetching and displaying data from resources. From now on, we are going to start creating and updating records by using **refine**.
- ✔ Add new interface name ICategory in **index.d.ts** inside interfaces directory by copying following code.

```
export interface ICategory {
  id: number;
  title: string;
}
```

- ✔ Also update th IPost interface of **index.d.ts** to the following code:

```
export interface IPost {
  id: number;
  title: string;
  status: "published" | "draft" | "rejected";
  category: { id: number };
  createdAt: string;
}
```

✔ Let's start by creating a new **<PostEdit>** component page responsible for editing a single record, for this create **edit.tsx** file inside **posts** directory and paste:

```
import {
    useForm,
    Form,
    Input,
    Select,
    Edit,
    useSelect,
} from "@pankod/refine-antd";
import { IPost } from "../../interfaces/index";

export const PostEdit: React.FC = () => {
    const { formProps, saveButtonProps, queryResult } = useForm<IPost>();

    const { selectProps: categorySelectProps } = useSelect<IPost>({
        resource: "categories",
        defaultValue: queryResult?.data?.data?.category.id,
    });

  return (
        <Edit saveButtonProps={saveButtonProps}>
            <Form {...formProps} layout="vertical">
                <Form.Item
                    label="Title"
                    name="title"
                    rules={[
                        {
                            required: true,
                        },
                    ]}
                >
                    <Input />
                </Form.Item>
```

```
<Form.Item
                label="Status"
                name="status"
                rules={[
                  {
                    required: true,
                  },
                ]}
              >
                <Select
                  options={[
                    {
                      label: "Published",
                      value: "published",
                    },
                    {
                      label: "Draft",
                      value: "draft",
                    },
                    {
                      label: "Rejected",
                      value: "rejected",
                    },
                  ]}
                />
              </Form.Item>
              <Form.Item
                label="Category"
                name={["category", "id"]}
                rules={[
                  {
                    required: true,
                  },
                ]}
              >
                <Select {...categorySelectProps} />
              </Form.Item>
            </Form>
          </Edit>
    );
};
```

✔ Now we can add the newly created component to our resource with **edit** prop:

✔ Change the **App.tsx** to following code:

```
import { Refine } from "@pankod/refine-core";
import {
  Layout,
  ReadyPage,
  notificationProvider,
  ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";

const App: React.FC = () => {
  return (
    <Refine
      routerProvider={routerProvider}
      dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
      Layout={Layout}
      ReadyPage={ReadyPage}
      notificationProvider={notificationProvider}
      catchAll={<ErrorComponent />}
      resources={[
        {
          name: "posts",
          list: PostList,
          show: PostShow,
          edit: PostEdit,
        },
      ]}
    />
  );
};

export default App;
```

✔ We are going to need an *edit* button on each row to display the **<PostEdit>** component. **refine** doesn't automatically add one, so we have to update our **<PostList>** component to add a **<EditButton>** for each record:

- ✔ Refer to the **<EditButton>** documentation for detailed usage information. →
- ✔ You can try using edit buttons which will trigger the edit forms for each record, allowing you to update the record data.
- ✔ Let's see what's going on in our **<PostEdit>** component in detail:
- ✔ **useForm** is a refine hook for handling form data. In the example, it returns **formProps** and **saveButtonProps**, where the former includes all necessary props to build the form and the latter has the ones for the save button.
- ✔ In the edit page, **useForm** hook initializes the form with current record values.
- ✔ Refer to the **useForm** documentation for detailed usage information . →
- ✔ **<Form>** and **<Form.Item>** are Ant Design components to build form inputs.
- ✔ **<Edit>** is a wrapper **refine** component for **<Form>.** It provides save, delete and refresh buttons that can be used for form actions.
- ✔ Form data is set automatically, whenever children input **<Form.Item>**'s are edited.
- ✔ Save button submits the form by executing the **useUpdate** method provided by the **dataProvider**. After a successful response, the application will be redirected to the listing page.

◆ Creating a record

- ✔ Creating a record in refine follows a similar flow as editing record.
- ✔ First we will create a create.tsx file in */pages/posts/* and copy the following code:

```
import {
  Create,
  Form,
  Input,
  Select,
  useForm,
  useSelect,
} from "@pankod/refine-antd";

import { IPost } from "../../interfaces/index";
```

```
export const PostCreate = () => {
 const { formProps, saveButtonProps } = useForm<IPost>();
 const { selectProps: categorySelectProps } = useSelect<IPost>({
  resource: "categories",
 });

 return (
  <Create saveButtonProps={saveButtonProps}>
   <Form {...formProps} layout="vertical">
    <Form.Item
     label="Title"
     name="title"
     rules={[
      {
       required: true,
      },
     ]}
    >
     <Input />
    </Form.Item>
    <Form.Item
     label="Status"
     name="status"
     rules={[
      {
       required: true,
      },
     ]}
    >
     <Select
      options={[
       {
        label: "Published",
        value: "published",
       },
       {
        label: "Draft",
        value: "draft",
       },
       {
        label: "Rejected",
        value: "rejected",
       },
      ]}
     />
    </Form.Item>
```

```
<Form.Item
        label="Category"
        name={["category", "id"]}
        rules={[
          {
            required: true,
          },
        ]}
      >
        <Select {...categorySelectProps} />
      </Form.Item>
    </Form>
  </Create>
 );
};
```

✔ After creating the **<PostCreate>** component, add it to the resource with **create** prop in **App.tsx**:

```
import { Refine } from "@pankod/refine-core";
import {
  Layout,
  ReadyPage,
  notificationProvider,
  ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";
import { PostCreate } from "./pages/posts/create";

const App: React.FC = () => {
```

```
return (
  <Refine
    routerProvider={routerProvider}
    dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
    Layout={Layout}
    ReadyPage={ReadyPage}
    notificationProvider={notificationProvider}
    catchAll={<ErrorComponent />}
    resources={[
      {
        name: "posts",
        list: PostList,
        show: PostShow,
        edit: PostEdit,
        create: PostCreate,
      },
    ]}
  />
 );
};

export default App;
```

- ✔ And that's it! Try it on the browser and see if you can create new posts from scratch.
- ✔ We should notice some minor differences from the edit example:
- ✔ **<Form>** is wrapped with **<Create>** component.
- ✔ Save button submits the form by executing the useCreate method provided by the **dataProvider**.
- ✔ No **defaultValue** is passed to **useSelect**.

◆ Deleting a record

- ✔ Deleting a record can be done in two ways.
- ✔ The first way is adding a delete button on each row since *refine* doesn't automatically add one, so we have to update our **<PostList>** component to add a **<DeleteButton>** for each record:
- ✔ Update the **list.tsx** to:

```tsx
import {
  List,
  TagField,
  DateField,
  Table,
  useTable,
  ShowButton,
  Space,
  EditButton,
  DeleteButton,
} from "@pankod/refine-antd";

import { IPost } from "../../interfaces/index";

export const PostList: React.FC = () => {
  const { tableProps } = useTable<IPost>();
  return (
    <List>
      <Table {...tableProps} rowKey="id">
        <Table.Column dataIndex="title" title="Title" />
        <Table.Column
          dataIndex="status"
          title="Status"
          render={(value) => <TagField value={value} />}
        />
        <Table.Column
          dataIndex="createdAt"
          title="CreatedAt"
          render={(value) => <DateField format="LLL" value={value} />}
        />
        <Table.Column<IPost>
          title="Actions"
          dataIndex="actions"
          render={(_text, record): React.ReactNode => {
            return (
              <Space>
                <ShowButton size="small" recordItemId={record.id} hideText />
                <EditButton size="small" recordItemId={record.id} hideText />
                <DeleteButton size="small" recordItemId={record.id} hideText />
              </Space>
            );
          }}
        />
      </Table>
    </List>
  );
};
```

- ✔ Refer to the **<DeleteButton>** documentation for detailed usage information. →
- ✔ Now you can try deleting records yourself. Just click on the delete button of the record you want to delete and confirm.
- ✔ The second way is by showing delete button in **<PostEdit>** component. To show delete button in edit page, **canDelete** prop needs to be passed to resource object.
- ✔ Update the **App.tsx** code to following:

```
import { Refine } from "@pankod/refine-core";
import {
  Layout,
  ReadyPage,
  notificationProvider,
  ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";
import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";
import { PostCreate } from "./pages/posts/create";
const App: React.FC = () => {
  return (
    <Refine
      routerProvider={routerProvider}
      dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
      Layout={Layout}
      ReadyPage={ReadyPage}
      notificationProvider={notificationProvider}
      catchAll={<ErrorComponent />}
      resources={[
        {
          name: "posts",
          list: PostList,
          show: PostShow,
          edit: PostEdit,
          create: PostCreate,
          canDelete: true,
        },
      ]}
    />
  );
};
export default App;
```

✔ After adding **canDelete** prop, **<DeleteButton>** will appear in edit form.

2. Core API

a) Auth Provider

- refine let's you set authentication logic by providing the **authProvider** property to the **<Refine>** component.
- **authProvider** is an object with methods that refine uses when necessary. These methods are needed to return a Promise. They also can be accessed with specialized hooks.
- An auth provider must include following methods:

```
const authProvider = {
  login: () => Promise.resolve(),
  register: () => Promise.resolve(),
  forgotPassword: () => Promise.resolve(),
  updatePassword: () => Promise.resolve(),
  logout: () => Promise.resolve(),
  checkAuth: () => Promise.resolve(),
  checkError: () => Promise.resolve(),
  getPermissions: () => Promise.resolve(),
  getUserIdentity: () => Promise.resolve(),
};
```

- refine consumes these methods using **authorization hooks**. Authorization hooks are used to manage authentication and authorization operations like login, logout and catching HTTP errors etc.

◆ Creating an authProvider

- Before building an auth provider lets first integrate custom Django backend here.
- To integrate Django rest APIs with this refine project, follow the steps outlined below.
- Git clone the following repo
- **$ git clone https://github.com/surajkarki66/refine-antd-dashboard-demo.git**

- Change the directory into above project.

  $ cd refine-antd-dashboard-demo

- Change the directory to **drf-todolist-app**Install dependencies of Django project.

  $ cd drf-todolist-app

- Install dependencies of Django project.

  $ pip install -r “requirements.txt”

- Migrate the database

  $ python manage.py migrate

- Super user credentials:

  Email: admin@admin.com

  Username: admin

  Password: suraj123

- Run the django app

  $ python manage.py runserver

- Now, the backend is up and running.

- We will build a simple **authProvider** from scratch to show the logic of how **authProvider** methods interact with the app.

  - ## login

    - ✔ **refine** expects this method to return a resolved Promise if the login is successful, and a rejected Promise if it is not.

    - ✔ First install the specified dependencies by entering the following command.

      $ yarn add axios jwt-decode

    - ✔ Create IRegister interface in **index.d.ts** which is inside interfaces directory and add the following code:

```
export interface IRegister {
  username: string;
  email: string;
}
```

✔ Create ILogin interface in **index.d.ts** which is inside interfaces directory and add the following code:

```
export interface ILogin {
  email: string;
  username: string;
  token: string;
  is_staff: boolean;
  is_superuser: boolean;
}
```

✔ Create a directory with name **providers** root directory and then inside this directory create a file with name **authProvider.ts**. Then paste the following code.

```
import axios, { AxiosRequestConfig } from "axios";
import { AuthProvider } from "@pankod/refine-core";

import { ILogin } from "../interfaces/index";

const axiosInstance = axios.create({ baseURL: "http://127.0.0.1:8000/api" });

axiosInstance.interceptors.request.use(
  // Here we can perform any function we'd like on the request
  (request: AxiosRequestConfig) => {
    // Retrieve the token from local storage
    const token = localStorage.getItem("auth-token");
    // Check if the header property exists
    if (request.headers) {
      // Set the Authorization header if it exists
      request.headers["Authorization"] = `Bearer ${token}`;
    } else {
      // Create the headers property if it does not exist
      request.headers = {
        Authorization: `Bearer ${token}`,
      };
    }

    return request;
  }
);

export { axiosInstance };
```

```
export const authProvider: AuthProvider = {
  login: async ({
    email,
    username,
    password,
  }: {
    email: string;
    username: string;
    password: string;
  }) => {
    try {
      const data = await axiosInstance.post<ILogin>("/users/login/", {
        email,
        username,
        password,
      });
      if (data) {
        const { is_staff, is_superuser } = data.data;
        if (is_superuser && is_staff) {
          localStorage.setItem("auth-token", data.data.token);
          localStorage.setItem("role", "admin");
          return Promise.resolve();
        } else if (!is_superuser && is_staff) {
          localStorage.setItem("auth-token", data.data.token);
          localStorage.setItem("role", "editor");
          return Promise.resolve();
        } else {
          return Promise.reject({
            message: "Forbidden!",
            name: "You don't have a permission to access the dashboard.",
          });
        }
      }
    } catch (error: any) {
      return Promise.reject({
        name: "Login error occurred",
        message: "Login failed!",
      });
    }
  },
  register: () => Promise.resolve(),
  updatePassword: () => Promise.resolve(),
  forgotPassword: () => Promise.resolve(),
  logout: () => Promise.resolve(),
  checkError: () => Promise.resolve(),
  checkAuth: () => Promise.resolve(),
  getPermissions: () => Promise.resolve(),
  getUserIdentity: () => Promise.resolve(),
};
```

- ✔ If the login is successful, pages that require authentication becomes accessible.
- ✔ If the login fails, refine displays an error notification to the user.
- ✔ **login** method will be accessible via **useLogin** auth hook.

```
import { useLogin } from "@pankod/refine-core";

const { mutate: login } = useLogin<{ email: string; username: string;
password: string }>();

login(values);
```

- ✔ Refer to useLogin documentation for more information. →
- ✔ Default login page: If an authProvider is given, refine shows a default login page on "/" and "/login" routes and a login form if a custom **LoginPage** is not provided. Rest of the app won't be accessible until successful authentication. After submission, login form calls the **login** method from **authProvider**.
- ✔ Change **App.tsx** to the following code:

```
import { Refine } from "@pankod/refine-core";
import {
  Layout,
  ReadyPage,
  notificationProvider,
  ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";
import { PostCreate } from "./pages/posts/create";
import { authProvider } from "./providers/authProvider";
```

```
const App: React.FC = () => {
  return (
    <Refine
      routerProvider={routerProvider}
      dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
      authProvider={authProvider}
      Layout={Layout}
      ReadyPage={ReadyPage}
      notificationProvider={notificationProvider}
      catchAll={<ErrorComponent />}
      resources={[
        {
          name: "posts",
          list: PostList,
          show: PostShow,
          edit: PostEdit,
          create: PostCreate,
          canDelete: true,
        },
      ]}
    />
  );
};

export default App;
```

✔ In this way, we can add other methods of authProvider.

✔ Update the authProvider.ts as follows:

```
import axios, { AxiosRequestConfig } from "axios";
import jwt_decode from "jwt-decode";
import { AuthProvider } from "@pankod/refine-core";
import { notification } from "@pankod/refine-antd";

import { ILogin, IRegister } from "../interfaces/index";


const axiosInstance = axios.create({ baseURL: "http://127.0.0.1:8000/api" });
axiosInstance.interceptors.request.use(
  // Here we can perform any function we'd like on the request
  (request: AxiosRequestConfig) => {
    // Retrieve the token from local storage
    const token = localStorage.getItem("auth-token");
```

```
// Check if the header property exists
  if (request.headers) {
    // Set the Authorization header if it exists
    request.headers["Authorization"] = `Bearer ${token}`;
  } else {
    // Create the headers property if it does not exist
    request.headers = {
      Authorization: `Bearer ${token}`,
    };
  }
  return request;
 }
);
export { axiosInstance };
export const authProvider: AuthProvider = {
 login: async ({
  email,
  username,
  password,
 }: {
  email: string;
  username: string;
  password: string;
 }) => {
  try {
   const data = await axiosInstance.post<ILogin>("/users/login/", {
    email,
    username,
    password,
   });
   if (data) {
    const { is_staff, is_superuser } = data.data;
    if (is_superuser && is_staff) {
     localStorage.setItem("auth-token", data.data.token);
     localStorage.setItem("role", "admin");
     return Promise.resolve();
    } else if (!is_superuser && is_staff) {
     localStorage.setItem("auth-token", data.data.token);
     localStorage.setItem("role", "editor");
     return Promise.resolve();
    } else {
     return Promise.reject({
      message: "Forbidden!",
      name: "You don't have a permission to access the dashboard.",
     });
    }
   }
  } catch (error: any) {
   return Promise.reject({
    name: "Login error occurred",
    message: "Login failed!",
   });
  }
 },
```

```
  register: async ({ email, username, password }) => {
    try {
      const data = await axiosInstance.post<IRegister>("/users/register/", {
        email,
        username,
        password,
      });
      if (data) {
        return Promise.resolve();
      }
    } catch (error: any) {
      return Promise.reject({
        message: "Register Failed!",
        name: "Register error occurred",
      });
    }
  },
  updatePassword: async () => {
    notification.success({
      message: "Updated Password",
      description: "Password updated successfully",
    });
    return Promise.resolve();
  },
  forgotPassword: async ({ email }) => {
    notification.success({
      message: "Reset Password",
      description: `Reset password link sent to "${email}"`,
    });
    return Promise.resolve();
  },
  logout: () => {
    localStorage.removeItem("auth-token");
    localStorage.removeItem("role");
    return Promise.resolve();
  },
 checkError: () => Promise.resolve(),
  checkAuth: async () => {
    return localStorage.getItem("auth-token")
      ? Promise.resolve()
      : Promise.reject({ redirectPath: "/login" });
  },
  getPermissions: async () => {
    // fetching role of user from server
    // for now hardcoding the role
    return Promise.resolve(["admin", "editor"]);
  },
```

```
getUserIdentity: async () => {
  const token = localStorage.getItem("auth-token");
  if (!token) {
    return Promise.reject();
  }
  const decoded: {
    username: string;
    email: string;
    exp: number;
    role: string;
  } = jwt_decode(token);
  return Promise.resolve({
    username: decoded.username,
    email: decoded.email,
    avatar: "https://i.pravatar.cc/150",
    role: localStorage.getItem("role"),
  });
 },
};
```

✔ Now, we need to pass LoginPage prop in <Refine> component. Add this inside the App.tsx file in the <Refine> component.

```
import { Refine } from "@pankod/refine-core";
import {
 Layout,
 ReadyPage,
 notificationProvider,
 ErrorComponent,
 AuthPage,  //  refine has a default auth page form served on the /login route
            // when the authProvider configuration is provided.
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";
import { PostCreate } from "./pages/posts/create";
import { authProvider } from "./providers/authProvider";
```

```
const App: React.FC = () => {
 return (
  <Refine
   routerProvider={routerProvider}
   dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
   authProvider={authProvider}
   Layout={Layout}
   ReadyPage={ReadyPage}
   notificationProvider={notificationProvider}
   catchAll={<ErrorComponent />}
   LoginPage={AuthPage}  // Changes is here =>Custom login component
                         // can be passed to the LoginPage property.
   resources={[
    {
     name: "posts",
     list: PostList,
     show: PostShow,
     edit: PostEdit,
     create: PostCreate,
     canDelete: true,
    },
   ]}
  />
 );
};

export default App;
```

- ✔ For more details about authProvider please visit this link https://refine.dev/docs/api-reference/core/providers/auth-provider/
- ✔ Here, when you run the app first it ask you to logged into system if you are not authenticated. But this form not works here due to django api. To make it work follow the given steps.
- ✔ Now, lets create a custom login page for our dashboard.
- ✔ For this inside the **pages** folder create a new folder and named it **login**.

✔ Then inside the login folder create a file and named it **index.tsx** and copy the following code:

```
import React from "react";
import { useLogin, useRouterContext } from "@pankod/refine-core";
import {
  Row,
  Col,
  AntdLayout,
  Card,
  Typography,
  Form,
  Input,
  Button,
  Checkbox,
} from "@pankod/refine-antd";
import "./styles.css";

const { Text, Title } = Typography;

export interface ILoginForm {
  email: string;
  username: string;
  password: string;
  remember: boolean;
}
export const Login: React.FC = () => {
  const [form] = Form.useForm<ILoginForm>();
  const { Link } = useRouterContext();
  const { mutate: login } = useLogin<ILoginForm>();

  const CardTitle = (
    <Title level={3} className="title">
      Sign in your account
    </Title>
  );
 return (
    <AntdLayout className="layout">
      <Row
        justify="center"
        align="middle"
        style={{
          height: "100vh",
        }}
      >
        <Col xs={22}>
          <div className="container">
            <Card title={CardTitle} headStyle={{ borderBottom: 0 }}>
```

```
<Form<ILoginForm>
            layout="vertical"
            form={form}
            onFinish={(values) => {
              login(values);
            }}
            requiredMark={false}
            initialValues={{
              remember: false,
            }}
          >
            <Form.Item
              name="email"
              label="Email"
              rules={[{ required: true }]}
            >
              <Input type="email" size="large" placeholder="Email" />
            </Form.Item>
            <Form.Item
              name="username"
              label="Username"
              rules={[{ required: true }]}
            >
              <Input size="large" placeholder="Username" />
            </Form.Item>
            <Form.Item
              name="password"
              label="Password"
              rules={[{ required: true }]}
              style={{ marginBottom: "12px" }}
            >
              <Input type="password" placeholder="●●●●●●●●" size="large" />
            </Form.Item>
          <div style={{ marginBottom: "12px" }}>
              <Form.Item name="remember" valuePropName="checked" noStyle>
                <Checkbox
                  style={{
                    fontSize: "12px",
                  }}
                >
                  Remember me
                </Checkbox>
              </Form.Item>
```

```
          <Link
                style={{
                 float: "right",
                 fontSize: "12px",
                }}
                to="/forgot-password"
              >
                Forgot password?
              </Link>
            </div>
            <Button type="primary" size="large" htmlType="submit" block>
              Sign in
            </Button>
          </Form>
          <div style={{ marginTop: 8 }}>
            <Text style={{ fontSize: 12 }}>
              Don’t have an account?{" "}
              <Link to="/register" style={{ fontWeight: "bold" }}>
                Sign up
              </Link>
            </Text>
          </div>
        </Card>
      </div>
    </Col>
  </Row>
 </AntdLayout>
 );
};
```

```
✔.layout {                                                                    g
   background: radial-gradient(50% 50% at 50% 50%, #63386a 0%, #310438
 100%);
   background-size: "cover";
 }

 .container {
   max-width: 408px;
   margin: auto;
 }
```

```
.title {
  text-align: center;
  color: #626262;
  font-size: 30px;
  letter-spacing: -0.04em;
 }

 .imageContainer {
  display: flex;
  align-items: center;
  justify-content: center;
  margin-bottom: 16px;
 }
```

✔ Now, change the **App.tsx** by changing LoginPage value to **Login** component that we created just before ago

```
import { Refine } from "@pankod/refine-core";
import {
 Layout,
 ReadyPage,
 notificationProvider,
 ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";
import { PostCreate } from "./pages/posts/create";
import { authProvider } from "./providers/authProvider";
import { Login } from "./pages/login";

const App: React.FC = () => {
 return (
  <Refine
   routerProvider={routerProvider}
   dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
   authProvider={authProvider}
   Layout={Layout}
   ReadyPage={ReadyPage}
   notificationProvider={notificationProvider}
   catchAll={<ErrorComponent />}
   LoginPage={Login}
```

```
resources={[
      {
        name: "posts",
        list: PostList,
        show: PostShow,
        edit: PostEdit,
        create: PostCreate,
        canDelete: true,
      },
    ]}
  />
);
};

export default App;
```

- ✔ In similar way, we can create custom registration page , etc
- ✔ Note: For now in authentication, only login works. So , you can use a super user credentials I already created.

  Email: admin@admin.com

  Username: admin

  Password: suraj123

- ✔ Now, to make registration page, we have to follows almost same process, first

```
import React from "react";
import { useRegister, useRouterContext } from "@pankod/refine-core";
import {
  Row,
  Col,
  AntdLayout,
  Card,
  Typography,
  Form,
  Input,
  Button,
} from "@pankod/refine-antd";
import "./styles.css";
```

```
const { Text, Title } = Typography;

export interface IRegisterForm {
  email: string;
  username: string;
  password: string;
}

export const Register: React.FC = () => {
  const [form] = Form.useForm<IRegisterForm>();
  const { Link } = useRouterContext();
  const { mutate: register } = useRegister<IRegisterForm>();

  const CardTitle = (
    <Title level={3} className="title">
      Sign up your account
    </Title>
  );

  return (
    <AntdLayout className="layout">
      <Row
        justify="center"
        align="middle"
        style={{
          height: "100vh",
        }}
      >
        <Col xs={22}>
          <div className="container">
            <Card title={CardTitle} headStyle={{ borderBottom: 0 }}>
              <Form<IRegisterForm>
                layout="vertical"
                form={form}
                onFinish={(values) => {
                  register(values);
                }}
                requiredMark={false}
              >
                <Form.Item
                  name="email"
                  label="Email"
                  rules={[{ required: true }]}
                >
                  <Input type="email" size="large" placeholder="Email" />
                </Form.Item>
```

```
            <Form.Item
              name="username"
              label="Username"
              rules={[{ required: true }]}
            >
              <Input size="large" placeholder="Username" />
            </Form.Item>
            <Form.Item
              name="password"
              label="Password"
              rules={[{ required: true }]}
              style={{ marginBottom: "12px" }}
            >
              <Input type="password" placeholder="●●●●●●●●" size="large" />
            </Form.Item>
            <div style={{ marginBottom: "12px" }}>
              <Link
                style={{
                  float: "right",
                  fontSize: "12px",
                }}
                to="/forgot-password"
              >
                Forgot password?
              </Link>
            </div>
            <Button type="primary" size="large" htmlType="submit" block>
              Sign up
            </Button>
          </Form>
          <div style={{ marginTop: 8 }}>
            <Text style={{ fontSize: 12 }}>
              Already have an account?{" "}
              <Link to="/login" style={{ fontWeight: "bold" }}>
                Sign in
              </Link>
            </Text>
          </div>
        </Card>
      </div>
    </Col>
  </Row>
 </AntdLayout>
 );
};
```

✔ Also create **styles.css** file in same directory as **index.ts** and copy the following code:

```
.layout {
  background: radial-gradient(50% 50% at 50% 50%, #63386a 0%, #310438 100%);
  background-size: "cover";
}

.container {
  max-width: 408px;
  margin: auto;
}

.title {
  text-align: center;
  color: #626262;
  font-size: 30px;
  letter-spacing: -0.04em;
}

.imageContainer {
  display: flex;
  align-items: center;
  justify-content: center;
  margin-bottom: 16px;
}
```

✔ Now, one final step is, we have to create a custom routerProvider. For now, don't focus on router provider for that we will check the documentation and only focus the following steps:

✔ In **App.tsx**, import **routerProvider** from **@pankod/refine-react-router-v6** and then pass prop **routerProvider** in Refine component of **App.tsx**.

✔ Update **App.tsx**

```
import { Refine } from "@pankod/refine-core";
import {
  Layout,
  ReadyPage,
  notificationProvider,
  ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";
```

```tsx
import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";
import { PostCreate } from "./pages/posts/create";
import { authProvider } from "./providers/authProvider";
import { Login } from "./pages/login";
import { Register } from "./pages/register";

const App: React.FC = () => {
 return (
  <Refine
   dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
   authProvider={authProvider}
   Layout={Layout}
   ReadyPage={ReadyPage}
   notificationProvider={notificationProvider}
   catchAll={<ErrorComponent />}
   LoginPage={Login}
   routerProvider={{
    ...routerProvider,
    routes: [
     {
      path: "/register",
      element: <Register />,
     },
    ],
   }}
   resources={[
    {
     name: "posts",
     list: PostList,
     show: PostShow,
     edit: PostEdit,
     create: PostCreate,
     canDelete: true,
    },
   ]}
  />
 );
};

export default App;
```

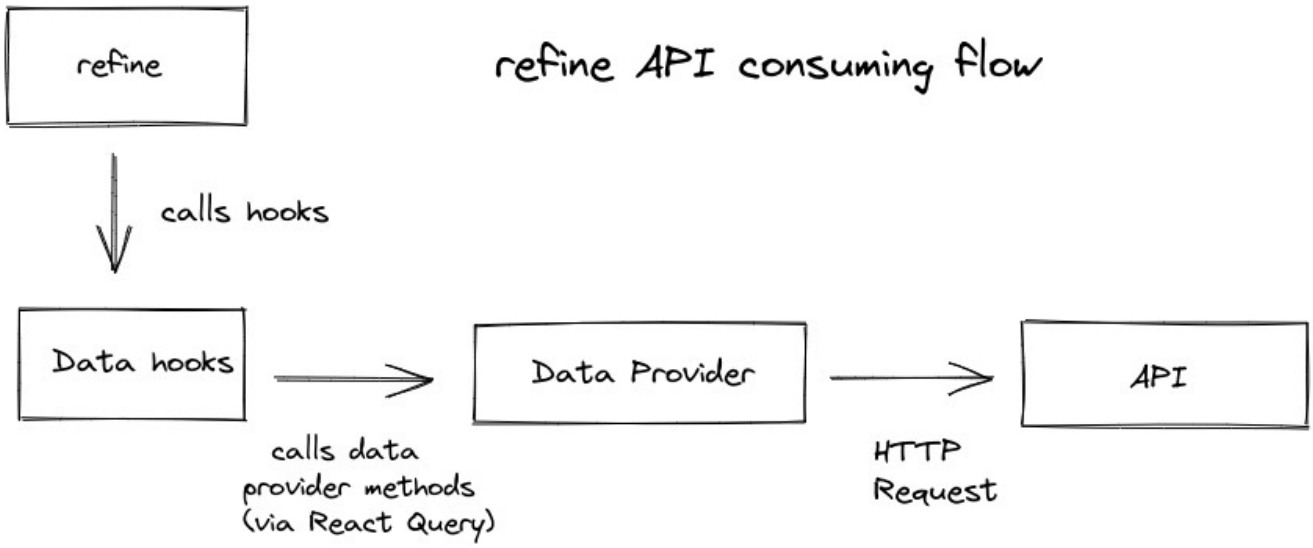
- ✔ Now, run the app and make sure that django server is also up and running.
- ✔ You can check the register page by changing the url to **/register.**
- ✔ To further research on routerProvider you can check this link https://refine.dev/docs/api-reference/core/components/refine-config/#routerprovider

b) Data Provider

- A data provider is the place where a refine app communicates with an API. Data providers also act as adapters for refine making it possible to consume different API's and data services conveniently. A data provider makes **HTTP** requests and returns response data back using predefined methods.
- A data provider must include following methods:

```
const dataProvider = {
  create: ({ resource, variables, metaData }) => Promise,
  createMany: ({ resource, variables, metaData }) => Promise,
  deleteOne: ({ resource, id, variables, metaData }) => Promise,
  deleteMany: ({ resource, ids, variables, metaData }) => Promise,
  getList: ({
    resource,
    pagination,
    hasPagination,
    sort,
    filters,
    metaData,
  }) => Promise,
  getMany: ({ resource, ids, metaData }) => Promise,
  getOne: ({ resource, id, metaData }) => Promise,
  update: ({ resource, id, variables, metaData }) => Promise,
  updateMany: ({ resource, ids, variables, metaData }) => Promise,
  custom: ({
    url,
    method,
    sort,
    filters,
    payload,
    query,
    headers,
    metaData,
  }) => Promise,
  getApiUrl: () => "",
};
```

- **refine** consumes these methods using data hooks.
- Data hooks are used to operate CRUD actions like creating a new record, listing a resource or deleting a record, etc.
- Data hooks use React Query to manage data fetching. React Query handles important concerns like caching, invalidation, loading states, etc.



Usage

- To activate data provider in refine, we have to pass the **dataProvider** to the **<Refine />** component in **App.tsx**.
- For eg:

```
import { Refine } from "@pankod/refine-core";

import dataProvider from "./dataProvider";

const App: React.FC = () => {
   return <Refine dataProvider={dataProvider} />;
};
```

◆ Creating a data provider

- We will build **"Django REST Dataprovider"** from scratch to show the logic of how data provider methods interact with the API.
- We will continue our refine app from previous part.
- First create a file inside **providers** directory name the file **dataProvider.ts**
- Install **querystring**

```
$ yarn add querystring
```

- Let's build a method that returns our data provider:
- Copy the following code inside the **dataProvider.ts**

```
import { AxiosInstance } from "axios";
import {
  DataProvider,
  HttpError,
  CrudOperators,
  CrudFilters,
  CrudSorting,
} from "@pankod/refine-core";
import { stringify } from "querystring";

import { axiosInstance } from "./authProvider";

axiosInstance.interceptors.response.use(
  (response) => {
    return response;
  },
  (error) => {
    const customError: HttpError = {
      ...error,
      message: error.response?.data?.message,
      statusCode: error.response?.status,
    };

    return Promise.reject(customError);
  }
);
```

```
const mapOperator = (operator: CrudOperators): string => {
  switch (operator) {
    case "ne":
    case "gte":
    case "lte":
      return `__${operator}`;
    case "contains":
      return "_like";
    case "eq":
    default:
      return "";
  }
};
const generateSort = (sort?: CrudSorting) => {
  if (sort && sort.length > 0) {
    const _sort: string[] = [];
    const _order: string[] = [];

    sort.forEach((item) => {
      _sort.push(item.field);
      _order.push(item.order);
    });

    return {
      _sort,
      _order,
    };
  }

  return;
};
const generateFilter = (filters?: CrudFilters) => {
  const queryFilters: { [key: string]: string } = {};
  if (filters) {
    filters.forEach((filter) => {
      if (filter.operator === "or" || filter.operator === "and") {
        // do according
      }
      if ("field" in filter) {
        const { field, operator, value } = filter;
        if (field === "search") {
          queryFilters[field] = value;
          return;
        }
        const mappedOperator = mapOperator(operator);
        queryFilters[`${field}${mappedOperator}`] = value;
      }
    });
  }
  return queryFilters;
};
```

```
export const DjangoDataProvider = (
  apiUrl: string,
  httpClient: AxiosInstance = axiosInstance
): Omit<
  Required<DataProvider>,
  "createMany" | "updateMany" | "deleteMany"
> => ({
  getList: async ({
    resource,
    hasPagination = true,
    pagination = { current: 1, pageSize: 7 },
    filters,
    metaData,
    sort,
  }) => {
    const url = `${apiUrl}/${resource}`;

    const { current = 1, pageSize = 7 } = pagination ?? {};

    const queryFilters = generateFilter(filters);

    const query: {
      limit?: number;
      offset?: number;
      _sort?: string;
      ordering?: string;
    } = hasPagination
      ? {
          limit: pageSize,
          offset: (current - 1) * pageSize,
        }
      : {};
    const generatedSort = generateSort(sort);
    if (generatedSort) {
      const { _sort, _order } = generatedSort;
      query.ordering =
        _order[0] === "asc" ? _sort.join(",") : "-" + _sort.join(",");
    }
    const { data } = await httpClient.get(
      `${url}?${stringify(query)}&${stringify(queryFilters)}`
    );

    return {
      data: data.results,
      total: data.count,
    };
  },
```

```
getMany: async ({ resource, ids }) => {
  const { data } = await httpClient.get(
    `${apiUrl}/${resource}?${stringify({ id__in: String(ids) })}`
  );
  return {
    data,
  };
},

create: async ({ resource, variables }) => {
  const url = `${apiUrl}/${resource}/create/`;
  const { data } = await httpClient.post(url, variables);
  return {
    data,
  };
},

update: async ({ resource, id, variables }) => {
  const url = `${apiUrl}/${resource}/update/${id}/`;
  const { data } = await httpClient.patch(url, variables);
  return {
    data,
  };
},
getOne: async ({ resource, id }) => {
  const url = `${apiUrl}/${resource}/${id}`;
  const { data } = await httpClient.get(url);
  return {
    data,
  };
},
deleteOne: async ({ resource, id, variables }) => {
  const url = `${apiUrl}/${resource}/delete/${id}/`;
  const { data } = await httpClient.delete(url, {
    data: variables,
  });
  return {
    data,
  };
},
getApiUrl: () => {
  return apiUrl;
},
```

```
custom: async ({ url, method, filters, sort, payload, query, headers }) => {
  let requestUrl = `${url}?`;

  if (sort) {
    const generatedSort = generateSort(sort);
    if (generatedSort) {
      const { _sort, _order } = generatedSort;
      const sortQuery = {
        _sort: _sort.join(","),
        _order: _order.join(","),
      };
      requestUrl = `${requestUrl}&${stringify(sortQuery)}`;
    }
  }
  if (filters) {
    const filterQuery = generateFilter(filters);
    requestUrl = `${requestUrl}&${stringify(filterQuery)}`;
  }
  if (query) {
    requestUrl = `${requestUrl}&${stringify(query)}`;
  }
  if (headers) {
    httpClient.defaults.headers = {
      ...httpClient.defaults.headers,
      ...headers,
    };
  }

  let axiosResponse;
  switch (method) {
    case "put":
    case "post":
    case "patch":
      axiosResponse = await httpClient[method](url, payload);
      break;
    case "delete":
      axiosResponse = await httpClient.delete(url, {
        data: payload,
      });
      break;
    default:
      axiosResponse = await httpClient.get(requestUrl);
      break;
  }
  const { data } = axiosResponse;
  return Promise.resolve({ data });
 },
});
```

- I know the above code looks complicated, let us understand in details.
- It will take the API URL as a parameter and an optional HTTP client. We will use axios as the default HTTP client.
- `getMany`, `createMany`, `updateMany` and `deleteMany` properties are optional. If you don't implement them, Refine will use `getOne`, `create`, `update` and `deleteOne` methods to handle multiple requests. If your API supports these methods, you can implement them to improve performance.
- Here, at first we imported axiosInstance from authProvider , we use axios as the default HTTP client.
- Then , we made an axios interceptor to handle error in better way.
- We also create function called **mapOperator** which gives different django query and filter operator based on input operator comes from frontend.
- We also create another function called **generateSort** which combines mutiple sorting field comes as input.
- **GenerateFilter** is a function that combines different query field, params, search field, sort , order field into a single query.
- **DjangoDataProvider** is a function it will take the API URL as a parameter and an optional **HTTP** client. We will use **axios** as the default **HTTP** client.
- Lets discuss each of the methods in details.
- ## getList

  ✔ This method allows us to retrieve a collection of items in a resource.

  - Parameter Types

Name	Type
resource	string
hasPagination?	boolean *(defaults to true)*
pagination?	Pagination;
sort?	CrudSorting;
filters?	CrudFilters;

  ✔ **refine** will consume this **getList** method using the **useList** data hook.

```
import { useList } from "@pankod/refine-core";

const { data } = useList({ resource: "todos" });
```

- ✔ Refer to the useList documentation for more information. →
- ✔ **Adding pagination:** We will send start and end parameters to list a certain size of items. Which you can see inside the **getList** function.

```
import { useList } from "@pankod/refine-core";

const { data } = useList({
   resource: "todos",
   config: {
      pagination: { current: 1, pageSize: 10 },
      hasPagination: true, // This can be omitted since it's default to `true` in the `getList` method of our data provider.
   },
});
```

- ✔ Listing will start from page 1 showing 10 records.
- ✔ **Adding sorting:** We'll sort records by specified order and field.
- ✔ Since our API accepts only certain parameter formats like _sort and _order we may need to transform some of the parameters.
- ✔ So we added the generateSort method to transform sort parameters.

```
import { useList } from "@pankod/refine-core";

const { data } = useList({
   resource: "posts",
   config: {
      pagination: { current: 1, pageSize: 10 },
      sort: [{ order: "asc", field: "title" }],
   },
});
```

- ✔ Listing starts from ascending alphabetical order on title field.
- ✔ **Adding filtering:** Filters allow you to filter queries using **refine's filter operators**. It is configured via field, operator and value properties.

✔ Since our API accepts only certain parameter formats to filter the data, we may need to transform some parameters.

✔ So we added the **generateFilter** and **mapOperator** methods to the transform filter parameters.

```
import { useList } from "@pankod/refine-core";

const { data } = useList({
   resource: "todos",
   config: {
      pagination: { current: 1, pageSize: 10 },
      sort: [{ order: "asc", field: "title" }],
      filters: [
         {
            field: "title",
            operator: "eq",
            value: "todo",
         },
      ],
   },
});
```

✔ Only lists records whose title equals to "todo".

- ## getMany

✔ This method allows us to retrieve multiple items in a resource.

✔ Implementation of this method is optional. If you don't implement it, refine will use **getOne** method to handle multiple requests.

✔ **Parameter Types**

Name	Type	Default
resource	string	
ids	BaseKey[]	

✔ refine will consume this **getMany** method using the **useMany** data hook.

```
import { useMany } from "@pankod/refine-core";
const { data } = useMany({ resource: "todos", ids: ["1", "2"] });
```

✔ Refer to the useMany documentation for more information. →

- **create**

✔ This method allows us to create a single item in a resource.

✔ **Parameter Types**

Name	Type	Default
resource	string	
variables	TVariables	{}

✔ **TVariables** is a user defined type which can be passed to **useCreate** to type **variables**

✔ **refine** will consume this **create** method using the **useCreate** data hook.

```
import { useCreate } from "@pankod/refine-core";

const { mutate } = useCreate();

mutate({
   resource: "tags",
   values: {
      name: "Science",
   },
});
```

✔ Refer to the useCreate documentation for more information. →

- **update**

✔ This method allows us to update an item in a resource.

✔ **Parameter Types**

Name	Type	Default
resource	string	
id	BaseKey	
variables	TVariables	{}

✔ **TVariables** is a user defined type which can be passed to **useUpdate** to type **variables**

✔ refine will consume this **update** method using the **useUpdate** data hook.

```
import { useUpdate } from "@pankod/refine-core";

const { mutate } = useUpdate();

mutate({
  resource: "tags",
  id: "2",
  values: { title: "Programming" },
});
```

✔ Refer to the useUpdate documentation for more information. →

- **getOne**

✔ This method allows us to retrieve a single item in a resource.

✔ **Parameter Types**

Name	Type	Default
resource	string	
id	BaseKey	

✔ **refine** will consume this **getOne** method using the **useOne** data hook.

```
import { useOne } from "@pankod/refine-core";

const { data } = useOne<ITag>({ resource: "tags", id: "1" });
```

✔ Refer to the useOne documentation for more information. →

- **deleteOne**

✔ This method allows us to delete an item in a resource.

✔ **Parameter Types**

Name	Type	Default
resource	string	
id	BaseKey	

Name	Type	Default
variables	TVariables[]	{}

✔ **TVariables** is a user defined type which can be passed to **useDelete** to type **variables**

✔ **refine** will consume this **deleteOne** method using the **useDelete** data hook.

```
import { useDelete } from "@pankod/refine-core";

const { mutate } = useDelete();

mutate({ resource: "tags", id: "2" });
```

✔ Refer to the useDelete documentation for more information. →

- **custom**

✔ An optional method named **custom** can be added to handle requests with custom parameters like URL, CRUD methods and configurations. It's useful if you have non-standard REST API endpoints or want to make a connection with external resources.

✔ **Parameter Types**

Name	Type
url	string
method	get, delete, head, options, post, put, patch
sort?	CrudSorting;
filters?	CrudFilters;
payload?	{}
query?	{}
headers?	{}

✔ **refine** will consume this **custom** method using the **useCustom** data hook.

```
import { useCustom } from "@pankod/refine-core";

const API_URL = useApiUrl();
const url = `${API_URL}/todos/todays-todo/`;
const { isLoading, data } = useCustom<ITodo>({ url, method: "get" });
```

✔ Refer to the useCustom documentation for more information. →

- Now change our **App.tsx** by changing the value of prop passing in **dataProvider** in **<Refine />** component.
- Update the **App.tsx** code to following:

```
import { Refine } from "@pankod/refine-core";
import {
  Layout,
  ReadyPage,
  notificationProvider,
  ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import "@pankod/refine-antd/dist/reset.css";
import { PostList } from "./pages/posts/list";
import { PostShow } from "./pages/posts/show";
import { PostEdit } from "./pages/posts/edit";
import { PostCreate } from "./pages/posts/create";
import { authProvider } from "./providers/authProvider";
import { DjangoDataProvider } from "./providers/dataProvider";
import { Login } from "./pages/login";
import { Register } from "./pages/register";

const App: React.FC = () => {
  return (
    <Refine
      dataProvider={DjangoDataProvider("http://127.0.0.1:8000/api")}
      authProvider={authProvider}
      Layout={Layout}
      ReadyPage={ReadyPage}
      notificationProvider={notificationProvider}
      catchAll={<ErrorComponent />}
      LoginPage={Login}
      routerProvider={{
        ...routerProvider,
        routes: [
          {
            path: "/register",
            element: <Register />,
          },
        ],
      }}
    />
  );
};
export default App;
```

- As you can see in above update **App.tsx**, we first import the **DjangoDataProvider** from **dataProvider.ts** which was previously built. Then we pass the **dataProvider** prop by giving the value get from calling **DjangoDataProvider** method with argument of base url of api server to the **<Refine />** component.
- We also remove the **resources** prop, because we just change the **dataProvider** so it is point less to use that previous resources cause our django api doesn't contain that resource i.e, posts.
- List of resource endpoints of our Django rest api are:
    - http://127.0.0.1:8000/api/todos/
    - http://127.0.0.1:8000/api/users/
    - http://127.0.0.1:8000/api/tags/
    - http://127.0.0.1:8000/api/provinces/
    - http://127.0.0.1:8000/api/subtasks/
- Here, resources are like tables of database that is **todos, users, tags, provinces, subtasks** are resources in refine.
- Now we are going to implement one of the resource from the above.
- Lets go for **tags**.
- Make sure your backend django server is up and running.
- **Creating a Tag:**
    - ✔ For this lets create directory with name **tags** inside the **pages** directory and then inside the **tags** directory create a file with name **create.tsx**
    - ✔ Also create an interface with name **ITag** inside the **index.d.ts** which is inside the **interfaces** directory, And paste the following code:

```
export interface ITag {
  id: BaseKey;
  name: string;
  created_at: Date;
  updated_at: Date;
}
```

    - ✔ Now, Inside the **create.tsx** of **tags** directory file paste the following code:

```
import { Create, Form, Input, useForm } from "@pankod/refine-antd";

import { ITag } from "../../interfaces/index";

export const TagCreate = () => {
 const { formProps, saveButtonProps } = useForm<ITag>();

 return (
  <Create saveButtonProps={saveButtonProps}>
   <Form {...formProps} layout="vertical">
    <Form.Item
     label="Name"
     name="name"
     rules={[
      {
       required: true,
      },
     ]}
    >
     <Input />
    </Form.Item>
   </Form>
  </Create>
 );
};
```

- **Listing tags:**
  - ✔ For this lets create a file with name **list.tsx** inside the **tags** directory of **pages** directory**.**
  - ✔ And paste the following code inside **list.tsx**

```
import {
 List,
 DateField,
 Table,
 useTable,
 ShowButton,
 Space,
 EditButton,
 DeleteButton,
} from "@pankod/refine-antd";

import { ITag } from "../../interfaces/index";
```

```
export const TagList: React.FC = () => {
 const { tableProps } = useTable<ITag>();
 return (
  <List>
   <Table {...tableProps} rowKey="id">
    <Table.Column dataIndex="id" title="Id" />
    <Table.Column dataIndex="name" title="Name" />
    <Table.Column
     dataIndex="created_at"
     title="CreatedAt"
     render={(value) => <DateField format="LLL" value={value} />}
     sorter
     showSorterTooltip
    />
    <Table.Column
     dataIndex="updated_at"
     title="UpdatedAt"
     render={(value) => <DateField format="LLL" value={value} />}
     sorter
     showSorterTooltip
    />
    <Table.Column<ITag>
     title="Actions"
     dataIndex="actions"
     render={(_text, record): React.ReactNode => {
      return (
       <Space>
        <ShowButton size="small" recordItemId={record.id} hideText />
        <EditButton size="small" recordItemId={record.id} hideText />
        <DeleteButton size="small" recordItemId={record.id} hideText />
       </Space>
      );
     }}
    />
   </Table>
  </List>
 );
};
```

- **Showing a tag details:**
  - ✔ For this lets create a file with name **show.tsx** inside the **tags** directory of **pages** directory.
  - ✔ Also install **momentjs** package by running **yarn add moment**
  - ✔ And paste the following code inside **show.tsx**

```tsx
import moment from "moment";

import { IResourceComponentsProps, useShow } from "@pankod/refine-core";
import { Typography, Show, Icons } from "@pankod/refine-antd";

import { ITag } from "../../interfaces/index";

const { Title, Text } = Typography;

export const TagShow: React.FC<IResourceComponentsProps> = () => {
  const { queryResult } = useShow<ITag>(); // used to fetch a single result
  const { data } = queryResult;
  const record = data?.data;
  return (
    <Show>
      <Title level={5}>Name</Title>
      <Text>{record?.name}</Text>
      <br />
      <br />
      <Text>
        <Icons.CalendarOutlined />{" "}
        {moment(record?.created_at).format("MMMM Do YYYY")}
      </Text>
    </Show>
  );
};
```

- **Editing a tag:**
  - ✔ For this lets create a file with name **edit.tsx** inside the **tags** directory of **pages** directory**.**
  - ✔ And paste the following code inside **edit.tsx**

```
import { useForm, Form, Input, Edit } from "@pankod/refine-antd";
import { ITag } from "../../interfaces/index";

export const TagEdit: React.FC = () => {
 const { formProps, saveButtonProps } = useForm<ITag>();

 return (
  <Edit saveButtonProps={saveButtonProps}>
   <Form {...formProps} layout="vertical">
    <Form.Item
     label="Name"
     name="name"
     rules={[
      {
       required: true,
      },
     ]}
    >
     <Input />
    </Form.Item>
   </Form>
  </Edit>
 );
};
```

- Now update the **App.tsx** by copying following code:

```
import { Refine } from "@pankod/refine-core";
import {
 Layout,
 ReadyPage,
 notificationProvider,
 ErrorComponent,
} from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import "@pankod/refine-antd/dist/reset.css";
import { authProvider } from "./providers/authProvider";
import { DjangoDataProvider } from "./providers/dataProvider";
import { Login } from "./pages/login";
import { Register } from "./pages/register";
import { TagList } from "./pages/tags/list";
import { TagShow } from "./pages/tags/show";
import { TagEdit } from "./pages/tags/edit";
import { TagCreate } from "./pages/tags/create";
```

```
const App: React.FC = () => {
 return (
  <Refine
   dataProvider={DjangoDataProvider("http://127.0.0.1:8000/api")}
   authProvider={authProvider}
   Layout={Layout}
   ReadyPage={ReadyPage}
   notificationProvider={notificationProvider}
   catchAll={<ErrorComponent />}
   LoginPage={Login}
   routerProvider={{
    ...routerProvider,
    routes: [
     {
      path: "/register",
      element: <Register />,
     },
    ],
   }}
   resources={[
    {
     name: "tags",
     list: TagList,
     show: TagShow,
     edit: TagEdit,
     create: TagCreate,
     canDelete: true,
    },
   ]}
  />
 );
};

export default App;
```

- Now, you can easily perform CRUD operation on tag using Refine admin dashboard.
- Here, you can see we just pass **resources** prop to the **<Refine />** component in **App.tsx**
- resources prop takes array of objects where each object represent a resource in Refine.

c) Access Control Provider

- Access control is a broad topic where there are lots of advanced solutions that provide different set of features.
- **refine** is deliberately agnostic for its own API to be able to integrate different methods (RBAC, ABAC, ACL, etc.) and different libraries (Casbin, CASL, Cerbos, AccessControl.js). can method would be the entry point for those solutions.
- **refine** provides an agnostic API via the **accessControlProvider** to manage access control throughout your app.
- An **accessControlProvider** must implement only one async method named **can** to be used to check if the desired access will be granted.

◆ Usage

- A basic example looks like:

```
const App: React.FC = () => {
  <Refine
    // other providers and props
    accessControlProvider={{
      can: async ({ resource, action, params }) => {
        if (resource === "posts" && action === "edit") {
          return Promise.resolve({
            can: false,
            reason: "Unauthorized",
          });
        }
        return Promise.resolve({ can: true });
      },
    }}
  />;
};
```

- ***resource**: It returns the resource (ResourceItemProps) object you gave to **<Refine />** component. This will enable **Attribute Based Access Control (ABAC)**, for example granting permissions based on the value of a field in the resource object.
- You can pass a **reason** along with **can**. It will be accessible using **useCan**. It will be shown at the tooltip of the buttons from **refine** when they are disabled.

- **refine** checks for access control in its related components and pages.

◆ Hooks and Components

■ **refine** provides a hook and a component to use the **can** method from the **accessControlProvider**.

■ useCan

- **useCan** uses the **can** as the query function for **react-query**'s **useQuery**. It takes the parameters that **can** takes. For eg:

```
const { data } = useCan({
  resource: "resource-you-ask-for-access",
  action: "action-type-on-resource",
  params: { foo: "optional-params" },
});
```

■ <CanAccess />

- **<CanAccess />** is a wrapper component that uses **useCan** to check for access control. It takes the parameters that can method takes and also a fallback. It renders its children if the access control returns true and if access control returns false renders fallback if provided.

- For eg.

```
<CanAccess
  resource="tags"
  action="edit"
  params={{ id: 1 }}
  fallback={<CustomFallback />}
>
  <YourComponent />
</CanAccess>
```

■ Performance

- As the number of points that checks for access control in your app increases the performance of your app may take a hit especially if its access control involves remote endpoints.

■ List of Default Access Control Points

• Routes

- ➢ Refine router which is react router package integrate access control for CRUD pages at [resource]/[action] and root routes.
- ➢ They will check access control with parameters:
  - dashboard (/): { resource: "dashboard", action: "list" }
  - list (e.g. /tags): { resource: "tags", action: "list", params: { *resource } }
  - create (e.g. /tags/create): { resource: "tags", action: "create", params { *resource } }
  - clone (e.g. /tags/clone/1): { resource: "tags", action: "create", params: { id: 1, *resource }}
  - edit (e.g. /tags/edit/1): { resource: "tags", action: "edit", params: { id: 1, *resource } }
  - show (e.g. /tags/show/1): { resource: "tags", action: "show", params: { id: 1, *resource } }
- ➢ In case access control returns **false** they will show **cathcAll** if provided or a standard error page otherwise.

• Sider

- ➢ Sider is also integrated so that unaccessible resources won't appear in the sider menu.
- ➢ Menu items will check access control with { resource, action: "list" }
- ➢ For example if your app has resource tags it will be checked with { resource: "tags", action: "list" }

• Buttons

- ➢ These buttons will check for access control. Let's say these buttons are rendered where resource is tags and id is 1 where applicable.

- **List**: { resource: "tags", action: "list", params: { *resource } }
- **Create**: { resource: "tags", action: "create", params: { *resource } }
- **Clone**: { resource: "tags", action: "create", params: { id: 1, *resource } }
- **Edit**: { resource: "tags", action: "edit", params: { id: 1, *resource } }
- **Delete**: { resource: "tags, action: "delete", params: { id: 1, *resource } }
- **Show**: { resource: "tags", action: "show", params: { id: 1, *resource } }

➢ These buttons will be disabled if access control returns **{ can: false }**

- From my research I found that using **Cerbos** with accessConrolProvider in Refine application will be efficient if the refine application is running in **Vite**. For that we also need to do research work on Cerbos.
- Also, if the refine application is running in webpack it will be better to use **Casbin**. The implementation of this type is also given in the official documentation. Check this **https://refine.dev/docs/advanced-tutorials/access-control/**

d) Notification Provider

- **refine** let's you set a notification API by providing the **notificationProvider** property to the **<Refine>** component.
- **notificationProvider** is an object with close and open methods. **refine** uses these methods to show and hide notifications. These methods can be called from anywhere in the application with **useNotification** hook.
- An **notificationProvider** must include following methods:

```
const notificationProvider = {
  show: () => {},
  close: () => {},
};
```

- And these methods types like this:

```ts
interface NotificationProvider {
    open: (params: OpenNotificationParams) => void;
    close: (key: string) => void;
}

interface OpenNotificationParams {
    key?: string;
    message: string;
    type: "success" | "error" | "progress";
    description?: string;
    cancelMutation?: () => void;
    undoableTimeout?: number;
}
```

- **Tip**: If you are using Ant Design you can use **notificationProvider** exported from **@pankod/refine-antd** package. It is compatible with Ant Design's **notification** component.

```ts
import { notificationProvider } from "@pankod/refine-antd";

<Refine
    ...
    notificationProvider={notificationProvider}
/>
```

- ## Usage:

    - To use **notificationProvider** in refine, we have to pass the notificationProvider to the **<Refine>** component.

```ts
import { Refine, NotificationProvider } from "@pankod/refine-core";
import routerProvider from "@pankod/refine-react-router-v6";
import dataProvider from "@pankod/refine-simple-rest";

const notificationProvider: NotificationProvider = {
    open: () => {},
    close: () => {},
};

const App = () => {
    return (
        <Refine
            notificationProvider={notificationProvider}
            routerProvider={routerProvider}
            dataProvider={dataProvider("https://api.fake-rest.refine.dev")}
        />
    );
};
```

- By default, **refine** doesn't require **notificationProvider** configuration.
- If an **notificationProvider** property is not provided, **refine** will use the default **notificationProvider**. This default **notificationProvider** lets the app work without an notification. If your app doesn't require **notification**, no further setup is necessary for the app to work.
- Before we start, we need set up the **react-toastify** requirements.
- Install **react-toastify** package.

  $ yarn add react-toastify

- First we have to import the **react-toastify** css file into **App.tsx** and then put the toast container inside the react fragment.

```
import { Refine } from "@pankod/refine-core";
import { Layout, ReadyPage, ErrorComponent } from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import "@pankod/refine-antd/dist/reset.css";
import { ToastContainer } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";

import { authProvider } from "./providers/authProvider";
import { DjangoDataProvider } from "./providers/dataProvider";
import { Login } from "./pages/login";
import { Register } from "./pages/register";
import { TagList } from "./pages/tags/list";
import { TagShow } from "./pages/tags/show";
import { TagEdit } from "./pages/tags/edit";
import { TagCreate } from "./pages/tags/create";

const App: React.FC = () => {
 return (
  <>
   <Refine
    dataProvider={DjangoDataProvider("http://127.0.0.1:8000/api")}
    authProvider={authProvider}
    DashboardPage={() => <div>Dashboard</div>}
    Layout={({ children }) => {
     return <Layout>{children}</Layout>;
    }}
    ReadyPage={ReadyPage}
    notificationProvider={notificationProvider}
    catchAll={<ErrorComponent />}
    LoginPage={Login}
```

```
routerProvider={{
        ...routerProvider,
        routes: [
          {
            path: "/register",
            element: <Register />,
          },
        ],
      }}
      resources={[
        {
          name: "tags",
          list: TagList,
          show: TagShow,
          edit: TagEdit,
          create: TagCreate,
          canDelete: true,
        },
      ]}
    />
    <ToastContainer />
  </>
 );
};

export default App;
```

- **open**
  - **refine** calls this method when it wants to open a notification. It also helps you to get the right notification by sending some parameters to the **refine** open method. For example, message, description, etc…
  - Here we open a **notification** with **react-toastify**.

```
import { toast } from "react-toastify";

const notificationProvider: NotificationProvider = {
    open: ({ message, key, type }) => {
        toast(message, {
            toastId: key,
            type,
        });
    },
};
```

- In case the notification is called repeatedly with the same **key**, let's update the previous notification instead of creating a new one.
- **toast.isActive(key)** returns **true** if the notification is still active. So we can check if the notification is already active and update it instead of creating a new one.

```
import { toast } from "react-toastify";

const notificationProvider: NotificationProvider = {
   open: ({ message, key, type }) => {
      if (toast.isActive(key)) {
         toast.update(key, {
            render: message,
            type,
         });
      } else {
         toast(message, {
            toastId: key,
            type,
         });
      }
   },
};
```

- Now, let's create a custom notification when the mutation mode is **undoable**. In this case, **refine** sends notification's type as **progress** as well as the **cancelMutation** and **undoableTimeout**.
- **undoableTimeout** decreases by 1 every second until it reaches 0. When it reaches 0, the notification is closed. **open** method is called again with the same **key** each countdown. So, the notification should be updated with the new **undoableTimeout** value.
- For this create a file with name **notificationProvider.tsx** inside the **providers** directory.
- Now, paste the following code inside **notificationProvider.tsx**

```
import React from "react";
import { NotificationProvider } from "@pankod/refine-core";
import { toast } from "react-toastify";
import { UndoableNotification } from "../components/undoableNotification";
```

```
export const notificationProvider: NotificationProvider = {
  open: ({ key, message, type, undoableTimeout, cancelMutation }) => {
    if (type === "progress") {
      if (toast.isActive(key as React.ReactText)) {
        toast.update(key as React.ReactText, {
          progress: undoableTimeout && (undoableTimeout / 10) * 2,
          render: (
            <UndoableNotification
              message={message}
              cancelMutation={cancelMutation}
            />
          ),
          type: "default",
        });
      } else {
        toast(
          <UndoableNotification
            message={message}
            cancelMutation={cancelMutation}
          />,
          {
            toastId: key,
            updateId: key,
            closeOnClick: false,
            closeButton: false,
            autoClose: false,
            progress: undoableTimeout && (undoableTimeout / 10) * 2,
          }
        );
      }
    } else {
      if (toast.isActive(key as React.ReactText)) {
        toast.update(key as React.ReactText, {
          render: message,
          closeButton: true,
          autoClose: 5000,
          type,
        });
      } else {
        toast(message, {
          toastId: key,
          type,
        });
      }
    }
  }
};
```

- **Note**: We add **closeButton** and **autoClose** for progress notifications are not closable by default. Because, when progress is done, the progress notification to be updated should be closeable.
- Also create UndoableNotification Component by creating a directory called **components** in the root directory.
- **undoableNotification.tsx**

```
type UndoableNotification = {
  message: string;
  cancelMutation?: () => void;
  closeToast?: () => void;
};

export const UndoableNotification: React.FC<UndoableNotification> = ({
  closeToast,
  cancelMutation,
  message,
}) => {
  return (
    <div>
      <p>{message}</p>
      <button
        onClick={() => {
          cancelMutation?.();
          closeToast?.();
        }}
      >
        Undo
      </button>
    </div>
  );
};
```

- **open** method will be accessible via **useNotification** hook.

```
import { useNotification } from "@pankod/refine-core";

const { open } = useNotification();

open?.({
    message: "Hey",
    description: "I <3 Refine",
    key: "unique-id",
});
```

- **close**
  - **refine** calls this method when it wants to close a notification. **refine** pass the **key** of the notification to the **close** method. So, we can handle the notification close logic with this **key**.
  - Now, also have to add the close method inside that **notificationProvider.tsx**
  - Update the **notificationProvider.tsx**

```
import React from "react";
import { NotificationProvider } from "@pankod/refine-core";
import { toast } from "react-toastify";
import { UndoableNotification } from "../components/undoableNotification";
export const notificationProvider: NotificationProvider = {
  open: ({ key, message, type, undoableTimeout, cancelMutation }) => {
    if (type === "progress") {
      if (toast.isActive(key as React.ReactText)) {
        toast.update(key as React.ReactText, {
          progress: undoableTimeout && (undoableTimeout / 10) * 2,
          render: (
            <UndoableNotification
              message={message}
              cancelMutation={cancelMutation}
            />
          ),
          type: "default",
        });
      } else {
        toast(
          <UndoableNotification
            message={message}
            cancelMutation={cancelMutation}
          />,
          {
            toastId: key,
            updateId: key,
            closeOnClick: false,
            closeButton: false,
            autoClose: false,
            progress: undoableTimeout && (undoableTimeout / 10) * 2,
          }
        );
      }
    }
```

```
    else {
        if (toast.isActive(key as React.ReactText)) {
          toast.update(key as React.ReactText, {
            render: message,
            closeButton: true,
            autoClose: 5000,
            type,
          });
        } else {
          toast(message, {
            toastId: key,
            type,
          });
        }
      }
    },
    close: (key) => toast.dismiss(key),
  };
```

- **close** method will be accessible via **useNotification** hook.

```
import { useNotification } from "@pankod/refine-core";

const { close } = useNotification();

close?.("displayed-notification-key");
```

- If you don't want to create notification provider you have another option which is by using notification object given by **@pankod/refine-antd.**
- For eg.

```
Import { notification } from "@pankod/refine-antd";

notification.success({
  message: "Success message",
  description: "In details",
  placement: "bottom",
});
```

e) Some of the important Hooks

- **Authorization**

  ➔ **useAuthenticated**

  - **useAuthenticated** calls the **checkAuth** method from the **authProvider** under the hood.
  - It returns the result of react-query's **useQuery** which includes many properties, some of which being **isSuccess** and **isError**.
  - Data that is resolved from the **useAuthenticated** will be returned as the **data** in the query result.
  - **Usage:**
    - **useAuthenticated** can be useful when you want to ask for authentication to grant access to **custom pages** manually.
    - We have used this hook in refine's **<Authenticated>** component which allows only authenticated users to access the page or any part of the code.
    - We will demonstrate a similar basic implementation below. Imagine that you have public page but you want to make some specific fields private.
    - We have a logic in **authProvider's checkAuth** method like below

```
const authProvider: AuthProvider = {
 ...
  checkAuth: () => {
     return localStorage.getItem("auth-token")
           ? Promise.resolve()
           : Promise.reject({ redirectPath: "/login" }),
  },
 ...
};
```

    - Let's create a wrapper component that renders children if **checkAuth** method returns the Promise resolved.
    - **./components/authenticated.tsx**

```
import { useAuthenticated, useNavigation } from "@pankod/refine";

export const Authenticated: React.FC<AuthenticatedProps> = ({
   children,
   fallback,
   loading,
}) => {
   const { isSuccess, isLoading, isError } = useAuthenticated();
```

```
  const { replace } = useNavigation();

  if (isLoading) {
    return <>{loading}</> || null;
  }

  if (isError) {
    if (!fallback) {
      replace("/");
      return null;
    }

    return <>{fallback}</>;
  }

  if (isSuccess) {
    return <>{children}</>;
  }

  return null;
};

type AuthenticatedProps = {
  fallback?: React.ReactNode;
  loading?: React.ReactNode;
};
```

- Now, only authenticated users can see the price field.
- **./components/postShow.tsx**

```
import { Authenticated } from "../components/authenticated"

const { Title, Text } = Typography;

export const PostShow: React.FC = () => (
  <div>
    <Authenticated>
      <span>Only authenticated users can see</span>
    </Authenticated>
  </div>
)
```

- **Caution:** This hook can only be used if the **authProvider** is provided.

➔ **useGetIdentity**

- **useGetIdentity** calls the **getUserIdentity** method from the **authProvider** under the hood.
- It returns the result of react-query's **useQuery** which includes many properties, some of which being **isSuccess** and **isError**. Data that is resolved from the **getUserIdentity** will be returned as the data in the query result.
- **Usage:**
  - **useGetIdentity** can be useful when you want to get the user information anywhere in your code.
  - Let's say that you want to show the user's name.
  - We have a logic in **authProvider's getUserIdentity** method like below.

```
const authProvider: AuthProvider = {
 ...
  getUserIdentity: () => {
   const token = localStorage.getItem("auth-token");
   if (!token) {
    return Promise.reject();
   }
   const decoded: {
    username: string;
    email: string;
    exp: number;
    role: string;
   } = jwt_decode(token);
   return Promise.resolve({
    username: decoded.username,
    email: decoded.email,
    avatar: "https://i.pravatar.cc/150",
    role: localStorage.getItem("role"),
   });
  }
 ...
};
```

  - You can access identity data like below.

```
import { useGetIdentity } from "@pankod/refine-core";

export const User: React.FC = () => {
   const { data: identity } = useGetIdentity<{ username: string; email: string;
                                                avatar: string; role: string; }>();

   return <span>{identity?.username}</span>
}
```

- This hook can only be used if the **authProvider** is provided.

➔ **useRegister**

- **useRegister** calls **register** method from **authProvider** under the hood. It registers the app if **register** method from **authProvider** resolves and if it rejects shows an error notification.
- It returns the result of **react-query's useMutation.**
- Data that is resolved from **register** will be returned as the **data** in the query result.

- **Usage:**
  - Normally refine provides a default register page. If you prefer to use this default register page, there is no need to handle register flow manually.
  - If we want to build a custom register page instead of the default one that comes with **refine**, **useRegister** can be used like this:
  - Create a custom register page with name **customRegisterPage.tsx** inside the pages

```
import { useRegister } from "@pankod/refine-core";

type RegisterVariables = {
  email: string;
  username: string;
  password: string;
};

export const RegisterPage = () => {
  const { mutate: register } = useRegister<RegisterVariables>();

  const onSubmit = (values: RegisterVariables) => {
    register(values);
  };
  return (
    <form onSubmit={onSubmit}>
      <label>Username</label>
      <input name="username" value="admin" />
      <label>Email</label>
      <input name="email" value="test@refine.com" />
      <label>Password</label>
      <input name="password" value="refine" />
      <button type="submit">Submit</button>
    </form>
  );
};
```

- **mutate** acquired from **useRegister** can accept any kind of object for values since **register** method from **authProvider** doesn't have a restriction on its parameters. A type parameter for the values can be provided to **useRegister**.

■ **Logged in after successful registration:**

- If you want to log in the user after successful registration, you can use **useLogin** hook after **useRegister** hook **onSuccess** callback.
- Modified **customRegisterPage.tsx** for this will be:

```tsx
import { useRegister, useLogin } from "@pankod/refine-core";

type RegisterVariables = {
  email: string;
  username: string;
  password: string;
};

export const RegisterPage = () => {
  const { mutate: register } = useRegister<FormVariables>();
  const { mutate: login } = useLogin<FormVariables>();

  const onSubmit = (values: FormVariables) => {
    register(values, {
      onSuccess: () => {
        login(values);
      },
    });
  };
  return (
    <form onSubmit={onSubmit}>
      <label>Username</label>
      <input name="username" value="admin" />
      <label>Email</label>
      <input name="email" value="test@refine.com" />
      <label>Password</label>
      <input name="password" value="refine" />
      <button type="submit">Submit</button>
    </form>
  );
};
```

■ Redirection after register

- We have 2 options for redirecting the app after registering successfully .
  - A custom url can be resolved from the promise returned from the **register** method of the authProvider.

```
const authProvider: AuthProvider = {
   ...
   register: () => {
      ...
      return Promise.resolve("/custom-url");
   }
}
```

  - A custom url can be given to mutate the function from the **useRegister** hook if you want to redirect yourself to a certain url.

```
import { useRegister } from "@pankod/refine-core";

const { mutate: register } = useRegister();

register({ redirectPath: "/custom-url" });
```

  - Then, you can handle this url in your **register** method of the **authProvider**.

```
const authProvider: AuthProvider = {
   ...
   register: ({ redirectPath }) => {
      ...
      return Promise.resolve(redirectPath);
   }
}
```

  - If the promise returned from the **register** method of the **authProvider** gets resolved with **false** no redirection will occur.

```
const authProvider: AuthProvider = {
   ...
   register: () => {
      ...
      return Promise.resolve(false);
   }
}
```

- **Tip:** If the promise returned from **register** is resolved with nothing, app will be redirected to the / route by default.
- **Caution:** This hook can only be used if **authProvider** is provided.

➔ There are also many other authorization hooks available in Refine. You can check it out here: **https://refine.dev/docs/api-reference/core/hooks/auth/useAuthenticated/**

- **Data**

➔ **useCreate**

■ **useCreate** is a modified version of react-query's **useMutation** for create mutations. It uses **create** method as mutation function from the **dataProvider** which is passed to **<Refine>**.

■ **Features:**

- Shows notifications after the mutation succeeds or fails.
- Automatically invalidates the **list** queries after mutation is succesfully run.

■ **Usage**

- Let'say we have a resource named **categories**

```
{
        [
            {
                id: 1,
                title: "E-business",
             },
            {
                id: 2,
                title: "Virtual Invoice Avon",
            },
        ];
}
```

- Lets see how useCreate works

```ts
type CategoryMutationResult = {
    id: number;
    title: string;
};

import { useCreate } from "@pankod/refine-core";

const { mutate } = useCreate<CategoryMutationResult>();

mutate({
    resource: "categories",
    values: {
        title: "New Category",
    },
});
```

- **mutate** can also accept lifecycle methods like **onSuccess** and **onError**.

```ts
mutate(
    {
        resource: "categories",
        values: {
            title: "New Category",
        },
    },
    {
        onError: (error, variables, context) => {
            // An error happened!
        },
        onSuccess: (data, variables, context) => {
            // Let's celebrate!
        },
    },
);
```

- **useCreate** returns react-query's **useMutation** result which includes a lot properties, one of which being mutate.
- For more details check out https://refine.dev/docs/api-reference/core/hooks/data/useCreate/

➔ **useList**

■ **useList** is a modified version of **react-query's useQuery** used for retrieving items from a **resource** with pagination, sort, and filter configurations.

- It uses the **getList** method as the query function from the **dataProvider** which is passed to <**Refine**>.
- **Usage:**
  - Let's say that we have a resource named **posts**
  - **useList** passes the query configuration to **getList** method from the **dataProvider**.
  - First of all, we will use **useList** without passing any query configurations.

```
import { useList } from "@pankod/refine-core";

type IPost = {
   id: number;
   title: string;
   status: "rejected" | "published" | "draft";
};

const postListQueryResult = useList<IPost>({ resource: "posts" });
```

  - Although we didn't pass any sort order configurations to **useList**, data comes in descending order according to **id** since **getList** has default values for sort order:

```
{
   sort: [{ order: "desc", field: "id" }];
}
```

  - **getList** also has default values for pagination:

```
{
   pagination: { current: 1, pageSize: 10 }
}
```

  - **Query Configuration**
    - **pagination**
      - Allows us to set page and items per page values. For example imagine that we have 1000 post records:

```
import { useList } from "@pankod/refine-core";

const postListQueryResult = useList<IPost>({
   resource: "posts",
   config: {
      pagination: { current: 3, pageSize: 8 },
   },
});
```

- Listing will start from page 3 showing 8 records.
- **sort**
  - Allows us to sort records by the speficified order and field.

```
import { useList } from "@pankod/refine-core";

const postListQueryResult = useList<IPost>({
   resource: "posts",
   config: {
      sort: [{ order: "asc", field: "title" }],
   },
});
```

  - Listing starts from ascending alphabetical order on the **title** field.
- **filters**
  - Allows us to filter queries using refine's filter operators. It is configured via **field, operator** and **value** properites.

```
import { useList } from "@pankod/refine-core";

const postListQueryResult = useList<IPost>({
   resource: "posts",
   config: {
      filters: [
         {
            field: "status",
            operator: "eq",
            value: "rejected",
         },
      ],
   },
});
```

  - Only lists records whose **status** equals to "rejected".

■ For more details about **useList** hook you can check this out https://refine.dev/docs/api-reference/core/hooks/data/useList/

➔ There are also many other data hooks available in Refine. You can check it out here: **https://refine.dev/docs/api-reference/core/hooks/data/useApiUrl/**

- **Form**

  ➔ **useForm**

  - **useForm** is a hook that allows to manage forms. It has some **action** methods that **create, edit** and **clone** the form.
  - For complete reference check out this link, https://refine.dev/docs/api-reference/core/hooks/useForm/

- **Export**

  ➔ **useExport**

  - **useExport** hook allows you to make your resources exportable. This hook accepts **export-to-csv's** options to create CSV files.

    ```
    import { useExport } from "@pankod/refine-core";

    const { triggerExport, isLoading } = useExport(options);
    ```

  - For complete reference check out this link, https://refine.dev/docs/api-reference/core/hooks/import-export/useExport/

- **Navigation**

  ➔ **useNavigation**

  - **refine** uses **routerProvider** and comes with all redirects out of the box. It allows you to manage your routing operations in refine. Using this hook, you can manage all the routing operations of your application very easily.

    ```
    import { useNavigation } from "@pankod/refine-core";

    const { create, edit, clone, show, list, push, replace, goBack } = useNavigation();
    ```

  - **useNavigation** uses the **useHistory** of the **routerProvider**.
  - For complete implementation please checkout this, https://refine.dev/docs/api-reference/core/hooks/navigation/useNavigation/

Finally, there are so many hooks available which is impossible to me to include in this documentation, so please check this link https://refine.dev/docs/api-reference/core/hooks/accessControl/useCan/

f) Components

- **<AuthPage>**
  - **<AuthPage>** component from **refine** contains authentication pages that can be used to login, register, forgot password and update password.
  - Before using **<AuthPage>** component you need to add `authProvider` that will be used to handle authentication.
  - **Usage:**
    - **<AuthPage>** component can be used like this:

```
import { Refine, AuthPage, useNavigation } from "@pankod/refine-core";
import routerProvider from "@pankod/refine-react-router-v6";

import { authProvider } from "./authProvider";
import { DashboardPage } from "./pages/dashboard";

const App = () => {
   return (
      <Refine
         routerProvider={{
            ...routerProvider,
            routes: [
               { path: "/login", element: <AuthPage type="login" /> },
               { path: "/register", element: <AuthPage type="register" /> },
               { path: "/forgot-password", element: <AuthPage type="forgotPassword"
/> },
            ],
         }}
         authProvider={authProvider}
         LoginPage={AuthPage}
         DashboardPage={DashboardPage}
         resources={[
            {
               name: "posts",
            },
         ]}
      />
   );
};
```

  - **Types:**
    - **<AuthPage>** component has the following types:
    - **"login"** - a type of the login page and default type.

        - **"register"** - type of the registration page.
        - **"forgotPassword"** - type of the forgot password page.
        - **"updatePassword"** - type of the update password page.

    - **Props:**
        - There are different types of props that you can pass to the <AuthPage /> componenet which you can briefly check in the following link, https://refine.dev/docs/api-reference/core/components/auth-page/

- **<Refine>**
    - **<Refine>** component is the entry point of a **refine** app. It is where the highest level of configuration of the app occurs.
    - **dataProvider** and **routerProvider** are required to bootstrap the app. After adding them, **resources** can be added as property.

```
import { Refine } from "@pankod/refine-core";
import dataProvider from "@pankod/refine-simple-rest";

import { PostList } from "./pages/posts/PostList";

const API_URL = "https://api.fake-rest.refine.dev";

const App: React.FC = () => {
  return (
    <Refine
      dataProvider={dataProvider(API_URL)}
      resources={[
        {
          name: "posts",
          list: PostList,
        },
      ]}
    />
  );
};

export default App;
```

    - To detaily known about the this component which is very necessary please check the below link,

https://refine.dev/docs/api-reference/core/components/refine-config/

- **<LayoutWrapper>**
  - **<LayoutWrapper>** wraps its contents in **refine's** layout with all customizations made in <Refine> component. It is the default layout used in resource pages. It can be used in custom pages to use global layout.
  - This component accepts layout customizations to further customize the layout parameters **(Title, Sider, Header, Footer, Layout, OffLayoutArea)** defined in <Refine> component.
  - **Usage:**
    - **<LayoutWrapper>** can be used inside custom pages to use global layout with all its customizations.
    - An example use in a custom page may look like this:
    - **App.tsx**

```
import { Refine, LayoutWrapper, Authenticated } from "@pankod/refine-core";
import { Layout, ReadyPage, ErrorComponent } from "@pankod/refine-antd";
import routerProvider from "@pankod/refine-react-router-v6";
import "@pankod/refine-antd/dist/reset.css";

import { notificationProvider } from "./providers/notificationProvider";
import { authProvider } from "./providers/authProvider";
import { DjangoDataProvider } from "./providers/dataProvider";
import { Login } from "./pages/login";
import { Register } from "./pages/register";
import { TagList } from "./pages/tags/list";
import { TagShow } from "./pages/tags/show";
import { TagEdit } from "./pages/tags/edit";
import { TagCreate } from "./pages/tags/create";
import { ToastContainer } from "react-toastify";
import "react-toastify/dist/ReactToastify.css";

const AuthenticatedCustomPage = () => {
 return (
  <Authenticated>
   <LayoutWrapper Sider={() => <>Sider</>}>
    <div>Custom Page</div>
   </LayoutWrapper>
  </Authenticated>
 );
};
```

```tsx
const App: React.FC = () => {
  return (
    <>
      <Refine
        dataProvider={DjangoDataProvider("http://127.0.0.1:8000/api")}
        authProvider={authProvider}
        DashboardPage={() => <div>Dashboard</div>}
        Layout={Layout}
        ReadyPage={ReadyPage}
        notificationProvider={notificationProvider}
        catchAll={<ErrorComponent />}
        LoginPage={Login}
        routerProvider={{
          ...routerProvider,
          routes: [
            {
              path: "/register",
              element: <Register />,
            },
            {
              exact: true,
              path: "/custom-page",
              element: <AuthenticatedCustomPage />,
            },
          ],
        }}
        resources={[
          {
            name: "tags",
            list: TagList,
            show: TagShow,
            edit: TagEdit,
            create: TagCreate,
            canDelete: true,
          },
        ]}
      />
      <ToastContainer />
    </>
  );
};

export default App;
```

- In this example, we hide the left sider only for this page. The rest should look same as resource pages.

3. Ant Design API

- Since, the Refine can support different UI framework like Antd, Material UI, Chakra UI, etc. So, we have lots of API for these UI framework. In this documentation, we were going to learn some Ant Design API.

a) Hooks

- **Field**
  - **useCheckboxGroup**
    - **useCheckboxGroup** hook allows you to manage an Ant Design **Checkbox.Group** component when records in a resource needs to be used as checkbox options.
    - **Usage:**
      - We will demonstrate how to get data at the **/tags** endpoint from the **https://api.fake-rest.refine.dev** REST API.
      - **https://*api.fake-rest.refine.dev/tags***

```
{
  [
    {
      id: 1,
      title: "Driver Deposit",
    },
    {
      id: 2,
```

```
import { Form, Checkbox, useCheckboxGroup } from "@pankod/refine-antd";
export const PostCreate: React.FC = () => {
  const { checkboxGroupProps } = useCheckboxGroup<ITag>({
    resource: "tags",
  });
  return (
    <Form>
      <Form.Item label="Tags" name="tags">
        <Checkbox.Group {...checkboxGroupProps} />
      </Form.Item>
    </Form>
  );
};
interface ITag {
  id: number;
  title: string;
}
```

- All we have to do is pass the **checkboxGroupProps** it returns to the **<Checkbox.Group>** component. **useCheckboxGroup** uses the **useList** hook for fetching data. Refer to `useList` hook for details. →
- There are different types of options for this hook. To explore please follow the link https://refine.dev/docs/api-reference/antd/hooks/field/useCheckboxGroup/#usage

◦ **useRadioGroup**

▪ **useRadioGroup** hook allows you to manage an Ant Design **Radio.Group** component when records in a resource needs to be used as radio options.

▪ **Usage**

- We will demonstrate how to get data at **/languages** endpoint from the **https://api.fake-rest.refine.dev** REST API.

```
{
  [
    {
      id: 1,
      title: "Turkish",
    },
    {
      id: 2,
      title: "English",
    },
  ].
```

-

```
import { Form, Radio, useRadioGroup } from "@pankod/refine-antd";
export const PostCreate = () => {
  const { radioGroupProps } = useRadioGroup<ILanguage>({
    resource: "languages",
  });
  return (
    <Form>
      <Form.Item label="Languages" name="languages">
        <Radio.Group {...radioGroupProps} />
      </Form.Item>
    </Form>
  );
};
interface ILanguage {
  id: number;
  title: string;
}
```

- All we have to do is pass the radioGroupProps it returns to the **<Radio.Group>** component. **useRadioGroup** uses the **useList** hook for fetching data. Refer to Ant Design useList hook for details. →
- There are different types of options for this hook. To explore please follow the link https://refine.dev/docs/api-reference/antd/hooks/field/useRadioGroup/

- **Form**
  - **useForm**
    - **useForm** is used to manage forms. It returns the necessary properties and methods to control the **Antd Form**. Also, it has been developed by using **useForm** imported from **@pankod/refine-core** package.
    - **Tip:** All the data related hooks(**useTable, useForm, useList** etc.) of refine can be given some common properties like **resource, metaData** etc.
    - **Usage**
      - For more detailed usage examples please refer to the **Ant Design Form** documentation.
      - ```
        import { Edit, Form, Input, useForm, Select } from "@pankod/refine-antd";
        ```
      - ```
        export const PostEdit: React.FC = () => {
            const { formProps, saveButtonProps } = useForm<IPost>();

            return (
                <Edit saveButtonProps={saveButtonProps}>
                    <Form {...formProps} layout="vertical">
                        <Form.Item label="Title" name="title">
                            <Input />
                        </Form.Item>
                        <Form.Item label="Status" name="status">
                         <Form.Item label="Status" name="status">
                            <Select
                                options={[
                                    {
                                        label: "Published",
                                        value: "published",
                                    },
        ```

```
                    {
                        label: "Draft",
                        value: "draft",
                    },
                    {
                        label: "Rejected",
                        value: "rejected",
                    },
                ]}
            />
        </Form.Item>
    </Form>
</Edit>
);
};

interface IPost {
    id: number;
    title: string;
    status: "published" | "draft" | "rejected";
}
```

- **formProps** includes all necessary values to manage Ant Design **Form** components.
- In the example if you navigate to **/posts/edit/1234** it will manage the data of the post with id of **1234** in an editing context. See **Actions** on how **useForm** determines this is an editing context.
- Since this is an edit form it will fill the form with the data of the post with id of **1234** and then the form will be ready to edit further and submit the changes.
- Submit functionality is provided by **saveButtonProps** which includes all of the necessary props for a button to submit a form including automatically updating loading states.
- **useForm** accepts type parameters for the record in use and for the response type of the mutation.
- **IPost** in the example represents the record to edit. It is also used as the default type for mutation response.

- **Tip:** If you want to show a form in a modal or drawer where necessary route params might not be there you can use the **useModalForm** or the **useDrawerForm** hook.
- There are different properties of useForm hook , you can definitely explored here https://refine.dev/docs/api-reference/antd/hooks/form/useForm/

- **List**
  - **useSimpleList**
    - By using **useSimpleList** you get props for your records from API in accordance with Ant Design **<List>** component. All features such as pagination, sorting come out of the box.
    - Refer to Ant Design docs for **<List>** component information →
    - **Usage**
      - ```
        import {                                                    ıd
          PageHeader,
          Typography,
          useMany,
          AntdList,
          useSimpleList,
          NumberField,
          Space,
        } from "@pankod/refine-antd";

        export const PostList: React.FC = () => {
          const { Text } = Typography;

          const { listProps } = useSimpleList<IPost>({
            initialSorter: [
              {
                field: "id",
                order: "asc",
              },
            ],
            pagination: {
              pageSize: 6,
            },
          });

          const categoryIds =
            listProps?.dataSource?.map((item) => item.category.id) ?? [];

          const { data } = useMany<ICategory>({
            resource: "categories",
            ids: categoryIds,
            queryOptions: {
              enabled: categoryIds.length > 0,
            },
          });
        ```
        ```

```tsx
  const renderItem = (item: IPost) => {
    const { title, hit, content } = item;

    const categoryTitle = data?.data.find(
      (category: ICategory) => category.id === item.category.id,
    )?.title;

    return (
      <AntdList.Item
        actions={[
          <Space key={item.id} direction="vertical" align="end">
            <NumberField
              value={hit}
              options={{
                // @ts-ignore
                notation: "compact",
              }}
            />
            <Text>{categoryTitle}</Text>
          </Space>,
        ]}
      >
        <AntdList.Item.Meta title={title} description={content} />
      </AntdList.Item>
    );
  };

  return (
    <PageHeader ghost={false} title="Posts">
      <AntdList {...listProps} renderItem={renderItem} />
    </PageHeader>
  );
};

interface IPost {
  id: number;
  title: string;
  content: string;
  hit: number;
  category: { id: number };
}
interface ICategory {
  id: number;
  title: string;
}
```

- You can use **AntdList.Item** and **AntdList.Item.Meta** like **<List>** component from **Ant Design**
- To disable pagination, you can set **hasPagination** property to **false** which is **true** by default. If **hasPagination** has been set to **false**, pagination elements will be hidden in the **<Table/>**. If you want to handle the pagination on the client-side you can override the **pagination** property in **tableProps**.

- There are also other usefull antd hooks which you can definitely explored check out this link **https://refine.dev/docs/api-reference/antd/hooks/table/useEditableTable/**

b) Components

- **Basic views**
  - **Create**
    - **<Create>** provides us a layout to display the page. It does not contain any logic but adds e
    - V

```
Import { Create, Form, Input, Select, useForm, useSelect} from "@pankod/refine-antd";

const PostCreate: React.FC = () => {
    const { formProps, saveButtonProps } = useForm<IPost>();
    return (
        <Create saveButtonProps={saveButtonProps}>
            <Form {...formProps} layout="vertical">
                <Form.Item
                    label="Title"
                    name="title"
                    rules={[
                        {
                            required: true,
                        },
                    ]}
                >
                    <Input />
                </Form.Item>
            </Form>
        </Create>
    );
};
```

        - There are different properties of Create you can checkout here **https://refine.dev/docs/api-reference/antd/components/basic-views/create/**

    - **Edit**
        - **<Edit>** provides us a layout for displaying the page. It does not contain any logic but adds extra functionalities like a refresh button.
        - We will show what **<Edit>** does using properties with examples.
        - Check out this, **https://refine.dev/docs/api-reference/antd/components/basic-views/edit/**

    - **List**
        - **<List>** provides us a layout to display the page. It does not contain any logic but adds extra functionalities like a create button or giving the page titles.
        - We will show what **<List>** does using properties with examples.
        - Check out this, **https://refine.dev/docs/api-reference/antd/components/basic-views/list/**

    - **Show**
        - **<Show>** provides us a layout for displaying the page. It does not contain any logic but adds extra functionalities like a refresh button or giving title to the page.
        - We will show what **<Show>** does using properties with examples.
        - Check out this, **https://refine.dev/docs/api-reference/antd/components/basic-views/show/**

- **Breadcrumb**
    - A breadcrumb displays the current location within a hierarchy. It allows going back to states higher up in the hierarchy. **<Breadcrumb>** component built with Ant Design's **Breadcrumb** components using the **useBreadcrumb** hook.

```
import { Show, Breadcrumb } from "@pankod/refine-antd";

const PostShow: React.FC = () => {
  return (
    <Show
      breadcrumb={<Breadcrumb />}
    >
      <p>Content of your show page...</p>
    </Show>
  );
};
```

- There are different properties of this Breadcrumb which you can explored from here **https://refine.dev/docs/api-reference/antd/components/breadcrumb/**

- **Buttons**
  - **Create**
    - **<CreateButton>** uses Ant Design's **<Button>** component. It uses the **create** method from **useNavigation** under the hood. It can be useful to redirect the app to the create page route of resource.

```
Import { Table, List, useTable, CreateButton } from "@pankod/refine-antd";

const PostList: React.FC = () => {
  const { tableProps } = useTable<IPost>();
  return (
    <List>
      <Table
        {...tableProps}
        rowKey="id"
        headerButtons={<CreateButton />}
      >
        <Table.Column dataIndex="id" title="ID" />
        <Table.Column dataIndex="title" title="Title" width="100%" />
      </Table>
    </List>
  );
};
interface IPost {
  id: number;
  title: string;
}
```

- **Properties**
  - **resourceNameOrRouteName**
    - It is used to redirect the app to the **/create** endpoint of the given resource name. By default, the app redirects to a URL with **/create** defined by the name property of resource object.

```
import { CreateButton } from "@pankod/refine-antd";

const MyCreateComponent = () => {
    return (
        <CreateButton
            resourceNameOrRouteName="categories"
        />
    );
};
```

    - Clicking the button will trigger the `create` method of **useNavigation** and then redirect to **/categories/create.**
  - **hideText**
    - It is used to show and not show the text of the button. When **true**, only the button icon is visible.

```
import { CreateButton } from "@pankod/refine-antd";

const MyCreateComponent = () => {
    return (
        <CreateButton
            hideText={true}
        />
    );
};
```

  - **accessControl**
    - This prop can be used to skip access control check with its **enabled** property or to hide the button when the user does not have the permission to access the resource with **hideIfUnauthorized** property. This is relevant only when an

```
import { CreateButton } from "@pankod/refine-antd";

export const MyListComponent = () => {
    return (
        <CreateButton
            accessControl={{ enabled: true, hideIfUnauthorized: true }}
        />
    );
};
```

- **Delete**
  - <**DeleteButton**> uses Ant Design's **<Button>** and **<Popconfirm>** components. When you try to delete something, a pop-up shows up and asks for confirmation. When confirmed it executes the useDelete method provided by your dataProvider.

```
import {
  Table,
  List,
  useTable,
  DeleteButton,
} from "@pankod/refine-antd";

const PostList: React.FC = () => {
  const { tableProps } = useTable<IPost>();

  return (
    <List>
      <Table {...tableProps} rowKey="id">
        <Table.Column dataIndex="id" title="ID" />
        <Table.Column dataIndex="title" title="Title" width="50%" />
        <Table.Column<IPost>
          title="Actions"
          dataIndex="actions"
          key="actions"
          render={(_, record) => (
            <DeleteButton size="small" recordItemId={record.id} />
          )}
          width="50%"
        />
      </Table>
    </List>
  );
};

interface IPost {
  id: number;
  title: string;
}
```

  - There are different properties of this Delete which you can explored from here **https://refine.dev/docs/api-reference/antd/components/buttons/delete-button/**

- **Refresh**
  - **<RefreshButton>** uses Ant Design's **<Button>** component to update the data shown on

the page via the **useOne** method provided by your `dataProvider`.

```
import { useShow } from "@pankod/refine-core";
import {
    RefreshButton,
    Show,
    Typography,
} from "@pankod/refine-antd";

const { Title, Text } = Typography;

const PostShow: React.FC = () => {
    const { queryResult } = useShow<IPost>();
    const { data, isLoading } = queryResult;
    const record = data?.data;

    return (
        <Show
            isLoading={isLoading}
            headerButtons={<RefreshButton />}
        >
            <Title level={5}>Id</Title>
            <Text>{record?.id}</Text>

            <Title level={5}>Title</Title>
            <Text>{record?.title}</Text>
        </Show>
    );
};

interface IPost {
    id: string;
    title: string;
}
```

- There are different properties of this Refresh which you can explored from here **https://refine.dev/docs/api-reference/antd/components/buttons/refresh-button/**
- There are also other different types of button which you can check on Refine official documentation.

- **Fields**
  - There are different fields like Boolean, Date, Email, etc you can check these on **https://refine.dev/docs/api-reference/antd/components/fields/boolean/**

- **<FilterDropdown>**
  - **Usage**
    - **<FilterDropdown>** is a helper component for **filter dropdowns in Ant Design <Table>** components.
    - It serves as a bridge by synchronizing between its children's input value and **<Table>**'s filter values.
    - **components/pages/postList.tsx**

```
Import { List, Table, FilterDropdown, Select, useTable, } from "@pankod/refine-antd";

const PostList: React.FC = (props) => {
  const { tableProps } = useTable<IPost>();

  return (
    <List>
      <Table {...tableProps} rowKey="id">
        <Table.Column dataIndex="id" title="ID" />
        <Table.Column
            dataIndex={["category", "id"]}
            title="Category"
            key="category.id"
            filterDropdown={(props) => (
              <FilterDropdown {...props}>
                <Select
                  mode="multiple"
                  placeholder="Select Category"
                  options={[
                    { label: "Ergonomic", value: "1" },
                    { label: "Island", value: "2" },
                  ]}
                />
              </FilterDropdown>
            )}
          />
      </Table>
    </List>
  );
};

interface IPost {
  id: number;
  category: {
    id: number;
  }
}
```

- Selecting categories from dropdown will send the id's of categories as filtering values to **Table** and data will be updated by **refine** under the hood.
- **<FilterDropdown>** will put two buttons for filtering and clearing filter actions.
- **Tip**: We added category options for **<Select>** manually for the sake of simplicity but **useSelect** hook can be used to populate the props of **<Select>**

```
const { selectProps: categorySelectProps } = useSelect<ICategory>({
   resource: "categories",
   optionLabel: "title",
   optionValue: "id",
});

<Select {...categorySelectProps} />
```

- There are different properties of this **<FilterDropdown>** which you can explored from here **https://refine.dev/docs/api-reference/antd/components/filter-dropdown/**

c) Customization

- **Theme**
  - Ant Design allows you to customize design tokens to satisfy UI diversity from business or brand requirements, including primary color, border radius, border color, etc. Design Tokens are the smallest element that affects the theme. By modifying the Design Token, we can present various themes or components.
  - Refer to the Ant Design documentation for more information about customizing Ant Design theme. →
  - **Theme customization**
    - **<ConfigProvider/>** component can be used to change theme. It is not required if you decide to use the default theme.
    - **Overriding the themes:**
      - You can override or extend the default themes. You can also create your own theme. Let's see how to do this.

```
import { Refine } from "@pankod/refine-core";
import dataProvider from "@pankod/refine-simple-rest";
import routerProvider from "@pankod/refine-react-router-v6";
Import { notificationProvider, Layout, ErrorComponent, ConfigProvider } from
"@pankod/refine-antd";
```

```
import { PostList, PostCreate, PostEdit, PostShow } from "./pages/posts";

const API_URL = "https://api.fake-rest.refine.dev";

const App: React.FC = () => {
  return (
    <ConfigProvider
      theme={{
        components: {
          Button: {
            borderRadius: 0,
          },
          Typography: {
            colorTextHeading: "#1890ff",
          },
        },
        token: {
          colorPrimary: "#f0f",
        },
      }}
    >
      <Refine
        dataProvider={dataProvider(API_URL)}
        routerProvider={routerProvider}
        resources={[
          {
            name: "posts",
            list: PostList,
            create: PostCreate,
            edit: PostEdit,
            show: PostShow,
          },
        ]}
        notificationProvider={notificationProvider}
        Layout={Layout}
        catchAll={<ErrorComponent />}
      />
    </ConfigProvider>
  );
};
```

- You can also do lots of stuffs with customization. For, instance can add the functionality of dark and light mode for this please checkout this link https://refine.dev/docs/api-reference/antd/customization/antd-custom-theme/

- **Layout**
  - You can create custom layouts using **<Refine>** and **<LayoutWrapper>** components.
  - Both of these components can accept the listed props for customization. **<Refine>** being for global customization and the **<LayoutWrapper>** being for local.
  - For more detail study please check the link , https://refine.dev/docs/api-reference/antd/customization/antd-custom-layout/

4. Next steps:

You can further perform lots of reasearch and study and can do some advanced research. For that the official Refine documentation is enough. Check this link, https://refine.dev/docs/advanced-tutorials/