Refine Documentation

Youth Innovation Lab

Prepared By

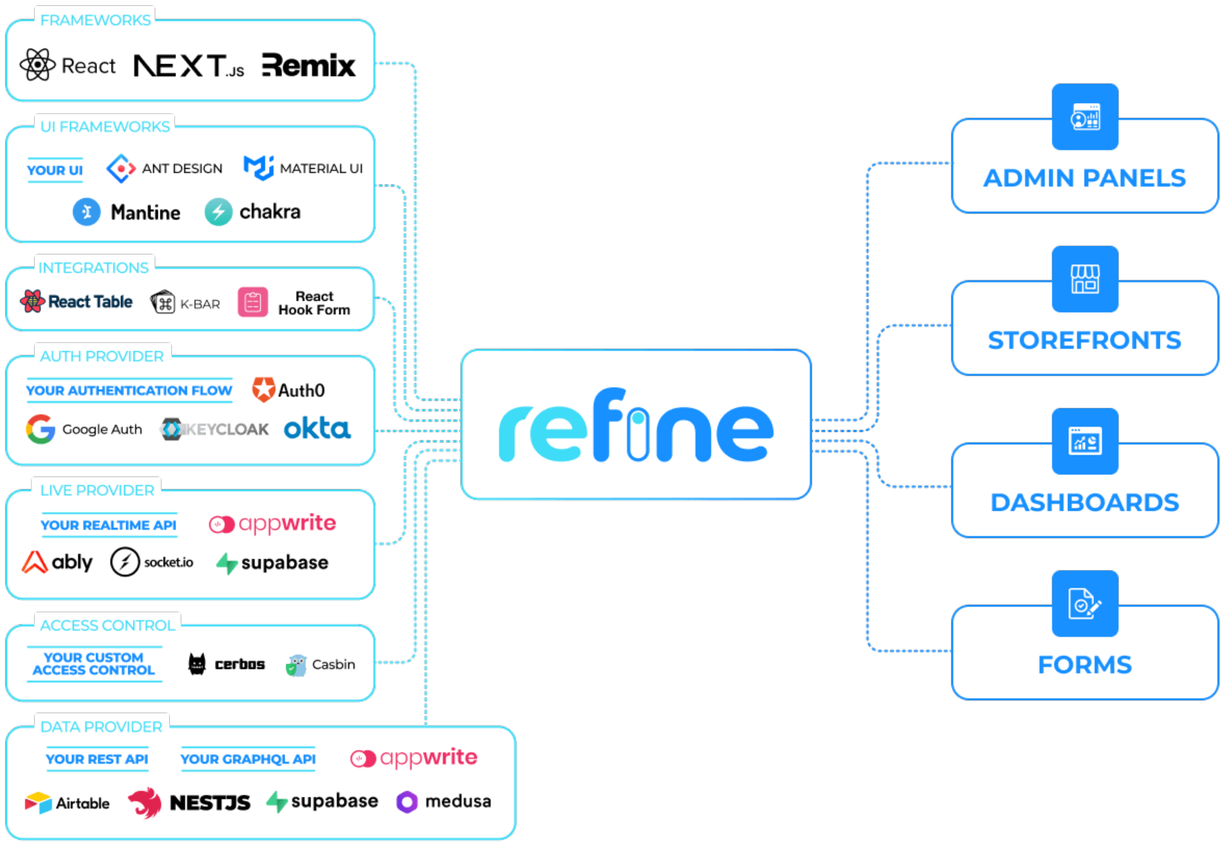
Suraj Karki

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.

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.

```
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.

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>
);
};
```

✔ 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, 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 create a folder and named it **register** and inside this create a file called **index.tsx** and paste the following code:

```
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";
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;
```

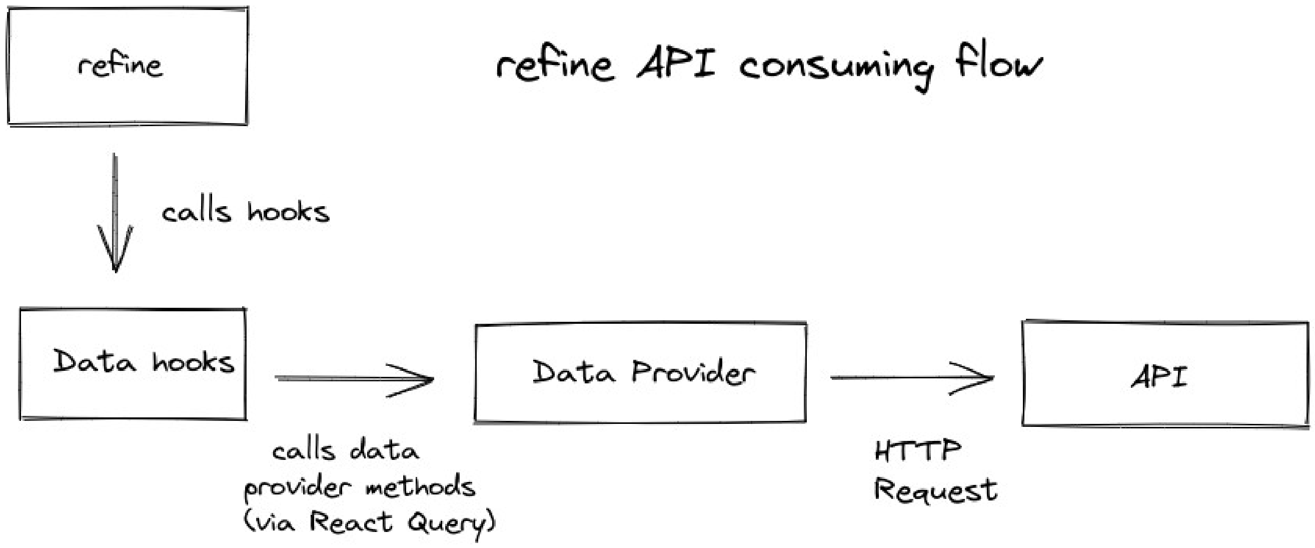
- ✔ Now, run the app and make sure that django server is also up and running.
- ✔ You can check the register page by changing the url to **/register.**
- ✔ To further research on routerProvider you can check this link https://refine.dev/docs/api-reference/core/components/refine-config/#routerprovider

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.


refine
refine API consuming flow
calls hooks
Data hooks
calls data provider methods (via React Query)
Data Provider
HTTP Request
API

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. →

◆