

Adani UI Guide

Installation and setup

This documentation describes the UI of Adani Project.

To Clone the project from Github:

```
git clone <SSH or HTTP link of project>
```

Navigate the newly created directory of project using:

```
cd Adani
```

Open the Project in vscode using:

```
code -a .
```

we have used Yarn package manager to install packages.

To install project dependencies run:

```
yarn OR yarn install
```

To run project locally in browser:

```
yarn start
```

To run build for android:

```
cap-update
```

Locate the path of 'capacitor-razorpay' dependency in package.json to your razorpay-capacitor directory

Tech stack used

Adani app is a web based app and wrapped with Capacitorjs to run on Android and IOS.

Tech stack and tools used to build this app is:

Language	TypeScript
UI library	React
Routing	React-router-dom
Styling	TailwindCSS and CSS
Testing	Jest, React-testing library
Async call	Axios
State management library	Redux-toolkit library
Form validation & management	Formik, Yup
Internationalization	i18next
To automate testing on commit	Husky

Installation and setup

This documentation describes the UI of Adani Project.

To Clone the project from Github:

```
git clone <SSH or HTTP link of project>
```

Navigate the newly created directory of project using:

```
cd Adani
```

Open the Project in vscode using:

```
code -a .
```

we have used Yarn package manager to install packages.

To install project dependencies run:

```
yarn OR yarn install
```

To run project locally in browser:

```
yarn start
```

To run build for android:

```
cap-update
```

Locate the path of 'capacitor-razorpay' dependency in package.json to your razorpay-capacitor directory

Design System

For better and Consistent UI we have used Design system, Design system is collection of reusable component and we use this components to achieve clear and consistent UI across the Application. It can be assembled together to build applications and act as a single source of truth for every person involved in the project.

Theme components

Rather than creating perfect big components we build small consistent components that behave same in any large component, Theme components are small components.

Button

Button component provide a quick way for user to interact with Data or other UI components in application in our application the button component accepts a bunch of properties as props,

Required props are `type`, `onClick`, `disabled`, `btnClass`, `btnSize`, `className`, and `style`.

these props are required no matter where you use this button component,

```
children: React.ReactNode;
type?: BtnType;
onClick?: CallbackType;
disabled?: boolean;
btnClass?: BtnClassType;
btnSize?: BtnSizeType;
className?: string;
style?: React.CSSProperties | undefined;
[key: string]: PropsType;
```

Button component can accept an optional props that is `children` prop, `children` prop is typed(Typescript) as `ReactNode`, it mean it can contain any value, whether its React component, a `ReactElement` or any value.

Now there are some pre-defined/fixed values for some props which helps in building consistant UI, like for `btnSize` there are fixed values like `btn-lg` and `btn-md` and for `btnType` there is `button`, `submit` and `reset` values are fixed.

Testing theme components

Tests are crucial part for any big and production level project it verifies that our code do what it meant to do, In theme components we are performing snapshot testing and using storybook library to test components visually.

Snapshot testing - Jest library provides us different methods to test one of the method is Snapshot testing snapshot testing is used to test react components, snapshot testing allow developers to quickly generate tests using simple syntax.

A simple test for button component would look like this

```
import React from "react";
import renderer from "react-test-renderer";
import Button from "../Button";

it("renders correctly when there is default button", () => {
  const tree = renderer.create(<Button>Default</Button>).toJSON();
  expect(tree).toMatchSnapshot();
});
```

written test for passing a children prop.

A single component can have many tests, but mostly for react components we think if passed a certain prop how will it react weather it outputs desired result of it breaks, based on the number of props it receive developer perform tests.

Storybook - Storybook is a development environment for UI components that enables you to create and showcase components in an environment outside your main application. This allows you to work on one module at a time.

Storybook also enables you to document, reuse, and test UI components. In addition to that, it makes building web applications faster and more efficient.

To run Storybook locally run:

```
npm run storybook
```

Once the command has successfully run, a working storybook server start running at <http://localhost:6006/>.

A story shows how a UI component should look on your browser. With Storybook, you can write multiple stories based on every change you want to make to your component. This means that rather than changing elements in the DOM, Storybook updates the information a user sees based on the stories you've created.

Example:

```
import Button from "../../theme/button/Button";
import { Meta, Story } from "@storybook/react";

export default {
  title: "Theme/Button",
  component: Button,
} as Meta;

const Template: Story = (args) => <Button { ... args}>{args.children}
</Button>;

export const Submit = Template.bind({});
Submit.args = {
  children: "Submit",
  type: "submit",
};
```

Like we wrote different scenarios for the testing a component, we write story for storybook to preview a component.

Card

Card component as name says its a visual card that wraps everything inside it with styles, It has some predefined styles and some style might apply via props

₹34,417

EMI per month

CheckBox and Radio Component

Checkboxes are an essential part of any large project it visually look good and fulfill the purpose of rendering something conditionally as, if user checks the checkbox only then we will show the next state of our UI.

Radio buttons in HTML is an interactive page element used to select one option from a list of options. Radio component is handling some state like checked and disabled.

Modal

A message box that is displayed on top of the screen, taking visual precedence over all other elements in application, mostly it is used to create dialog box and popup windows, Modals are common UI patterns to providing information or requiring confirmation from users.

For better control over modal windows we are using React-modal library it focuses on accessibility, The only required prop for the modal object is `isOpen`, which indicates whether the modal should be displayed.

For types of modal it has three types as of now `notification-modal`, `login-modal` and `bottom-modal`

Slider

Slider is a component for html range element it ranges values according how much user has change the range slider, minimum value is 0 default value is set to 100 we can change it.

TextArea and TextField

TextArea will helps users to write multiline inputs it could be anything like address or a message or description that can go multiple lines.

TextField on the other hand is a html element for input and used to gather information from users, The component is getting input type as a prop so type could be anything it's basically

a `HTMLInputTypeAttribute` type so the value for type that it received from prop is valid if it could be use in html input type attribute.

ToggleSwitch

ToggleSwitch is toggle button under the hood its a html input checkbox element, We have use style to it so it can look more appealing to user and not a basic checkbox.

PreAuth & Auth

The applications journey starts from selecting a language, default language is en-GB, selected language got stored in cookies and is global language for entire app.

After choosing language next step is to get authorised.

Registration/Authorization

When user registers his mobile number, a OTP is sent to his mobile number, along with OTP a authToken is sent from back-end, the authToken is stored in localStorage for user's future visits.

The token is then used by all the other services to store user's information like when user create their profile, to store their loans information etc.

Auth

Once the user gets authorised the dashboard is shown to them.

Dashboard is the point where user can access various services like Mandi prices, Loans, Advisory from experts and many more.

Mandi price

According to users location, zone and city all the prices will be shown to user, the mandi data is coming form back-end, user can sort, search and get wholesale or retail price.

Branch locator

Branch is located according to user's Geo-location, and user's current location is used to get nearest branch, if user choose custom(select) location all the branches will shown to them according to their selection.

Quick pay

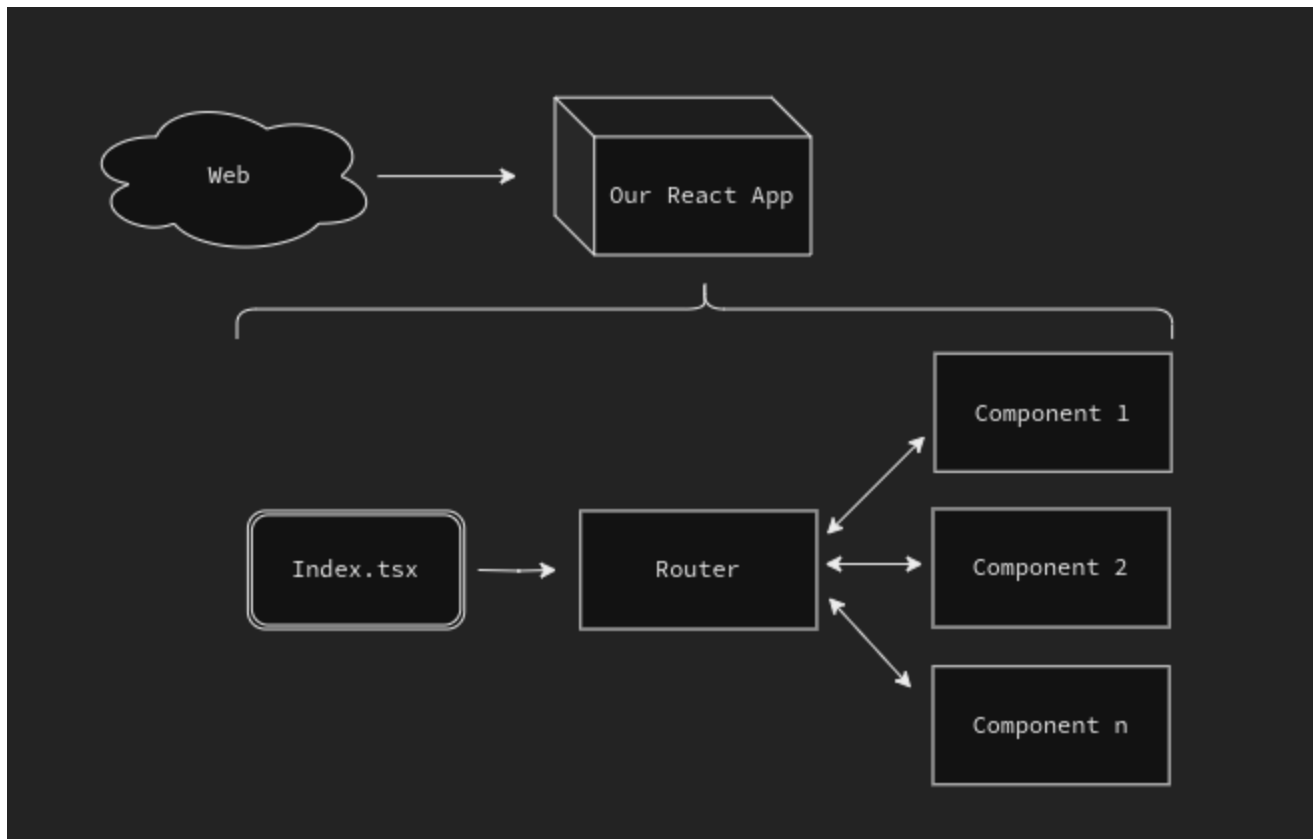
User can pay their loan dues from the app itself, to use the service user have to provide loan number or phone number, loan details or loan list is fetched via API according what credential user has passed.

Once user select the loan due he wants to pay, a Razorpay window opens.

Router

Big application can have many pages and for user to navigate on different page of our application without loading is important, routing is a way of navigate to different parts or different pages of the application based on the user request without reloading the application.

We are using React-router library to use router in our app, Using this library, we can configure the routes and when the user request matches, the respective route will be executed, which will load the specific component.



In the top level App.tsx component we have defined `RouterProvider` and `router` attribute, In router we are passing if user is authorized or not, based on the condition we are showing whether the before authorization routes will appear or after authorization.

Non-authorized user will not be able to access other routes they can only use some of the pages.

PreAuthRoutes is array of object first object is for parent component that renders all the children routes using Outlet, This allows nested UI to show up when child routes are rendered. If the parent route matched exactly, it will render a child index route or nothing if there is no index route.

For pre-auth routes we are rendering-

- Select language
- Allow permission
- Splash screen
- Registration and
- OTP Verification

Other object inside the array is for undefined paths or routes that does not exist in application, And in this case we are rendering a NoPage component.

AuthRoutes is again a array of all the routes that will only shown to user when they are authorized.

In authorized routes there are many nested routes rendering components some of them are Dashboard, Pre-own-tractor-loan and many more.

Web-Mobile app

The Application is build upon web technologies but the application is intended to work on mobile devices so used a tool called Capacitor.js

Capacitor provides a consistent, web-focused set of APIs that enable an app to stay as close to web standards as possible, while accessing rich native device features on platforms that support them. If it works in the browser, it probably works in a mobile app when using Capacitor. Adding native functionality is straightforward with a Plugin API for Swift on iOS, Java on Android, and JavaScript for the web.

In order to develop Android applications using Capacitor, you will need two additional dependencies:

- Android Studio
- An Android SDK installation

To open the project in Android Studio, run:

```
npx cap open android
```

Our project needs to have the following three requirements in order to use Capacitor with our existing application:

- A `package.json` file
- A separate directory for built web assets such as `dist` or `www`
- An `index.html` file at the root of your web assets directory

Once we have created our native projects, we can sync our web application to our native project by running the following command.

```
npx cap sync
```

In order to develop iOS applications using Capacitor, you will need four additional dependencies:

- Xcode
- Xcode Command Line Tools
- Homebrew
- Cocoapods

Testing your Capacitor app

Once you've synced over your web bundle to your native project, it is time to test your application on a mobile device. There are a few different ways to do this, but the easiest way is to use the built in Capacitor CLI commands.

To run a debug build of your Capacitor app on an iOS device, you can run:

```
npx cap run ios
```

Similarly, to run a debug build of your Capacitor app on an Android device, you can run:

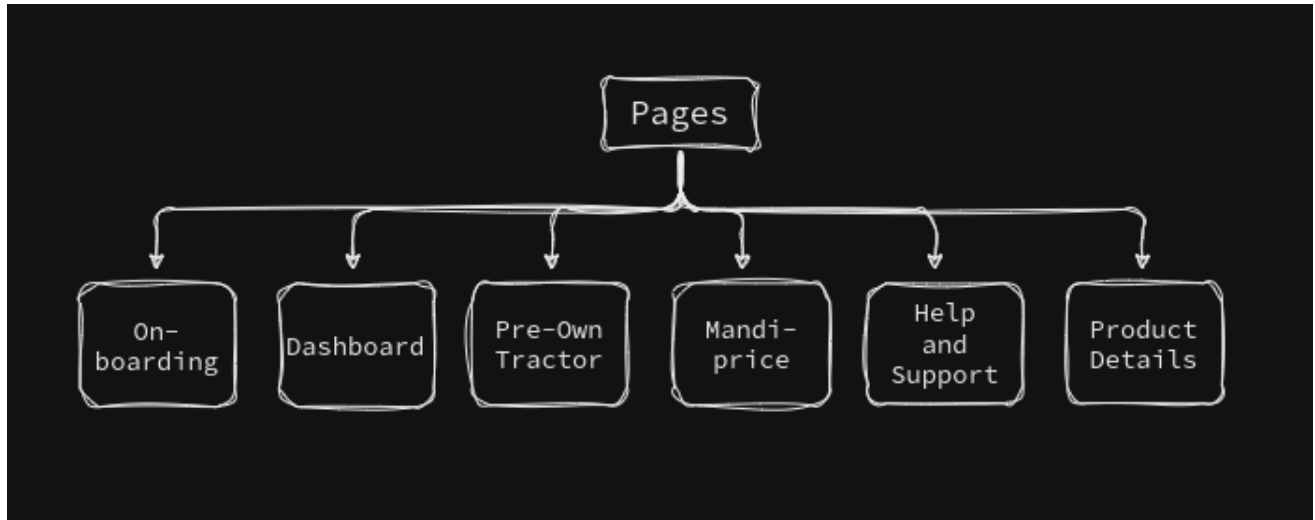
```
npx cap run android
```

Copy the web app build and Capacitor configuration file into the native platform project. Run this each time you make changes to your web app or change a configuration value.

```
npx cap copy
```

Pages

Application journey starts from On-boarding journey, On-boarding journey begins when user first visit our application, This pages are mostly before user gets authorised.



SelectLanguage is first page that appear to user when they first visits the application, It takes language preference from user and sets the preferred language across the application, It does this using `changeLanguage` method from `i18n` package.

As soon as user select's their desired language the page route goes on to the allow-permission page on clicking the Continue button.

```
const setLanguage = () => {  
  i18n.changeLanguage(selectedLanguage);  
  navigate("/allow-permissions");  
};
```

User permission is required for application to gather some information, For now we just need the Camera and Location, microphone permission is just for the UI we are not require microphone access for now.

To take picture and choose from existing pictures from the photo album the application uses capacitor.js Camera plugin and same for Location we need to take permission from user and capacitor.js provide plugin Geolocation and a requestPermissions promise to get location permission if device location services are disabled.

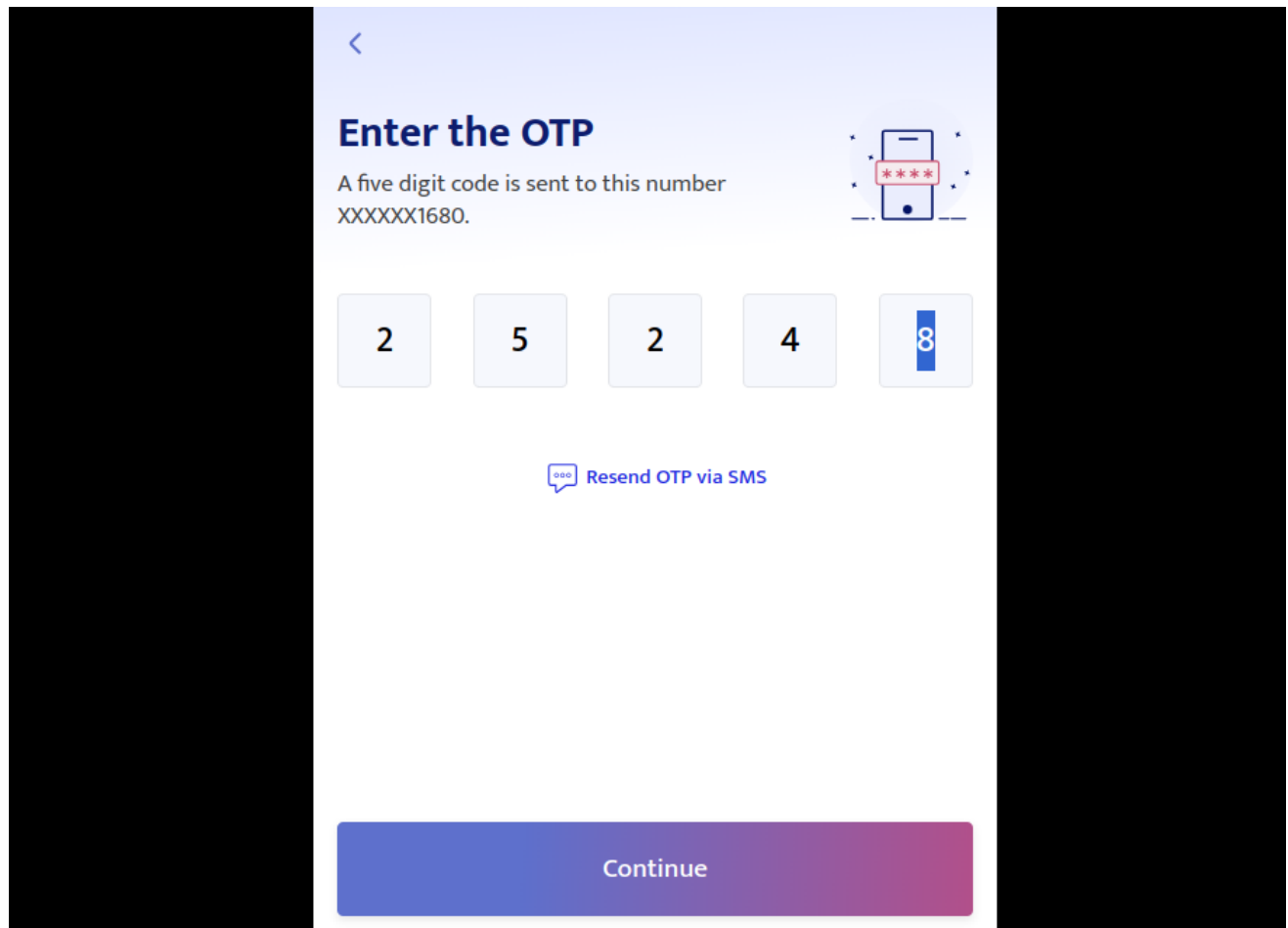
```
const permission = async () => {  
  await Camera.requestPermissions();  
  await Geolocation.requestPermissions();  
};
```

User needs to share phone number to get authorised and proceed with the application once the user give phone number and accept terms and condition he can move to the OTP verification page.

OTP is sent from auth API on users mobile number.

On this page OTP input field(component) is taken from React-input-otp package, The input field component takes some props some of them are to style the input box.

From Registration page the otp is sent via a API once the OTP has been sent to the mobile number a timer starts for 60 second if user fail to provide otp in this 60 second, user through a button can call the authResend API to get otp on his phone.



When user registers his mobile number, A authToken is sent from back-end, the authToken is stored in localStorage for user's future visits and from then user is authorised and can access authorised pages, entry point to see all the important pages are in dashboard.

Dashboard help users get a quick view of critical information at a glance it could be some important sections of our application or some brief data to show user.

Dashboard's home page is the starting point at which user can access their section of interest Home page renders many services like Mandi Price, Loans, Fard Advisory and so

on.

Shows all the loan, If user does not have any active loan or applied loan the section will be empty and all the loans applied or active loans are fetched from API.

When view button gets clicked user can see details of applied loan like Loan type, loan amount nearest branch and contact information of nearest branch.

User can quickly create their profile in the application Setting page has three sections one in which user can create and edit Profile we are using Formik to handle forms in the application and Yup library to validate forms to create the profile user can upload their profile picture, name, phone number, email etc.

The next section is Setting menu here user can see payments for the loans, Download forms to apply for business loan and home loan.

Third is Setting footer here user can read about privacy policy, read about section and logout from the application.

```
signout: (state) => {  
  state.isAuthenticated = false;  
  localStorage.removeItem("authToken");  
}
```

Next set of pages come under Pre-Own Tractor journey pre-own tractor is use to get loan by users on pre owned tractors, journey starts from selecting borrowers identity for now the application only have farmer option.

and

To proceed in pre-owned tractor loan customer need to give Vehicle number/RC number if user provide a valid RC number our API verifies it,

```
export const verifyRC = createAsyncThunk(  
  "loan/pre-own-tractor/verifyRC",  
  async ({ RCNumber }: TractorInfo) => {  
    return await  
    API.setAuthHeader(getAuthToken()).post(`/verify/rc`, {  
      rc: RCNumber,  
    });  
  }  
);
```

After that user selects manufacturer manufacturer is tractor company and next user need to select modal number of selected manufacturer.

User requires to share their documents after getting tractor information we are collecting documents user is not bound to share only aadhar card as identity proof but they can also upload other identity proofs like Pan card, Driving licence, Voter ID, or Passport.

User can upload identity proof via taking a picture of it or from image gallery, To read the picture and getting data from picture of identity proof we are using API.

```
export const readAadhaar = createAsyncThunk(
  "loan/pre-own-tractor/readAadhaar",
  async ({ formData, idType, stage }: Reader) => {
    return await API.setAuthHeader(getAuthToken()).post(
      `/verify/${stage}/${idType}/read`,
      formData,
      {
        "Content-Type": "multipart/form-data",
      }
    );
  }
);
```

Stores, Network and API

Redux Store

This code sets up a Redux store, which is a central place to manage our application's data. Reducers are functions that manage specific parts of your app's data. This code imports various reducers, each responsible for a different aspect of your app's state, like user data, authentication, payments, and more.

Store Configuration

The `configureStore` function is used to create the store. It takes our reducers and some middleware. Middleware adds extra functionality to Redux. Here, we use `redux-logger` to log actions and state changes for debugging.

In here we are exporting some helpful tools:

- `RootState` : A type representing our application's overall state.
- `AppDispatch` : A function to dispatch actions to the store.
- `AppThunk` : A type for asynchronous actions.

- `useAppDispatch` and `useAppSelector` : Hooks to easily use the dispatch function and select data from the store.

The application require to have multiple state slices, we are gonna look at Auth Slice.

Auth Slice

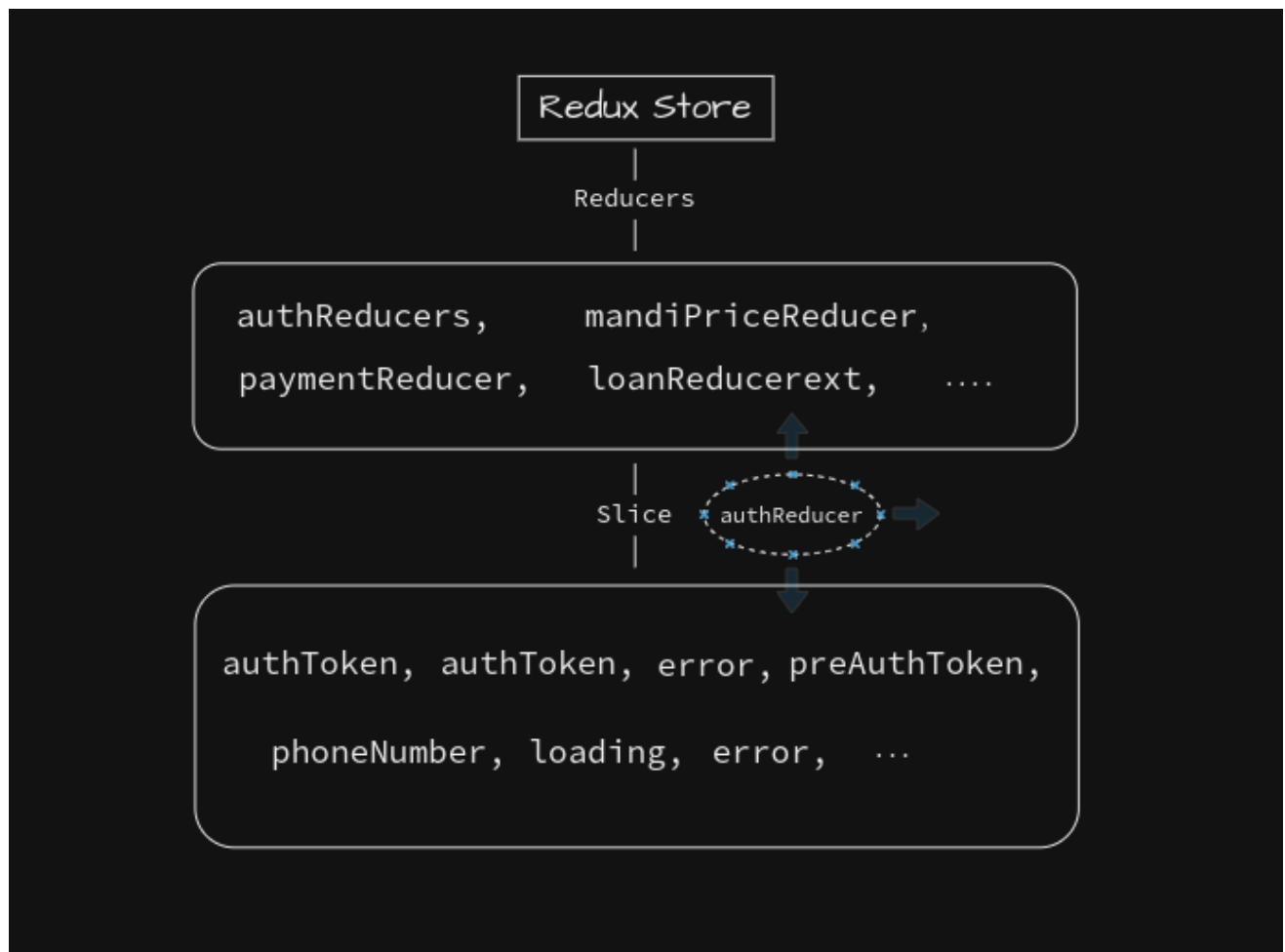
In this file we defines a "auth slice" using the `createSlice` function from RTK. A slice represents a specific part of our application's state and contains reducers and actions related to that state.

The `initialState` object defines the initial state of the slice. It includes properties like `authToken`, `phoneNumber`, `loading`, `error`, `resendOtpError`, `isAuthorized`, and `preAuthToken`. These properties are used to make user verified.

Extra Reducers

The `extraReducers` section adds additional reducers to the slice. These reducers are defined elsewhere in the code and handle various actions related to authentication, verification, and token management.

The `createSlice` function allows us to define reducers and actions. Here are some like `setAuth`: updates the `authToken`, `signout`: makes the user as unauthorized and `resetResendOtpError`: clears the `resendOtpError` property in the state.



Network

To make network requests in our application we are using axios, It has common HTTP methods like GET, POST, PATCH, and DELETE, along with error handling and configuration settings.

The class named `AxiosFetch` that serves as a wrapper around Axios to make HTTP requests.

- **Constructor:** Initializes Axios with a base URL and default JSON headers.
- **setAuthHeader:** Sets an authorization header with a token for authentication.
- **setConfig:** Modifies Axios instance configuration options.

Handling Response

- **responseHandler:** Manages responses by chaining promise.
- **errorHandler:** Handles Axios errors using `HandleError` function.
- **successHandler:** Returns data from successful response.


```

get<T>(url: string, header?: RawAxiosRequestHeaders) {
    return this.responseHandler(
        this.instance.get(url, {
            headers: header,
        })
    ) as T;
}

```

API

API file in slice defines a set of async Redux Thunks for handling authentication-related actions in our application. This thunk make use of the Axios-based `API` to interact with the remote server and manage authentication-related operations.

`fetchAuth`

Does authentication with a phone number.

- **Action Type:** `"auth/info"`
- **API Request:** POSTs the phone number to `/auth`.

`authVerify`

Verifies authentication with an OTP.

- **Action Type:** `"auth/verify"`
- **API Request:** POSTs OTP to `/auth/verify`.

`authResend`

Resends OTP for verification.

- **Action Type:** `"auth/resend"`
- **API Request:** GET request to `/auth/resend`.

Schemas

Form validation is important when building large application, Schemas defines a set of Yup validation, schema used for validating different types of form inputs in our application. Yup is a JavaScript validation library commonly used for form validation.

It's important to note that the Schema imports various validation rules `userNameRule`, `phoneNumberRule`, `emailRule`, `loanAccNumRule` from a `rules` module. These rules

define the validation criteria for specific input fields (e.g., username, phone number).

Rules file

```
export const userNameRule = Yup.string()  
  .min(USER_NAME_MIN_LENGTH, "User name is too short!")  
  .max(USER_NAME_MAX_LENGTH, "User name is too long!")  
  .matches(USER_NAME_REGEX, "Enter valid user name!");
```

min-length is 2 and max length is 25.

Let's see what schema we have

editProfileSchema

Validates user profile editing inputs.

- **Input Validated:** User image, user name, email, gender, date of birth.

```
export const editProfileSchema = Yup.object().shape({  
  user_image: Yup.string(),  
  user_name: userNameRule.required("User name is required!"),  
  email_id: emailRule,  
  gender: Yup.string(),  
  date_of_birth: Yup.date(),  
});
```

signupSchema

Validates user registration input.

- **Inputs validated for:** Phone number and user agreement (checkbox).

userNameSchema

Validates a user's name input.

- **Inputs validated for:** User name.

quickPayLoginSchema

Validates inputs used for quick pay login.

- **Inputs validated for:** Loan account number and phone number.

helpRequestSchema

Validates inputs for submitting help requests.

- **Inputs validated for:** User name, mobile number, email, and a message.

waitlistSchema

Validates inputs for joining a waitlist.

- **Inputs Validated:** User name and mobile number.

loanRequestFormSchema

- **Description:** Validates fields for loan request forms.
- **Inputs validated for:** User name, mobile number, and user agreement.

AgentRequestFormSchema

Validates inputs for agent request forms.

- **Inputs Validated for :** User name and agent code.

VehicleNumberSchema

Validates vehicle number input.

- **Inputs Validated for:** Vehicle number.

KYCInfoSchema

Validates Know Your Customer (KYC) information input.

- **Inputs validated for:** ID number, father's name, date of birth, address, and gender.

```
export const KYCInfoSchema = Yup.object().shape({
  idNumber: Yup.string().required("ID number is required!"),
  fatherName: Yup.string().required("Father name is required!"),
  dob: Yup.date().required("Date of birth is required!"),
  address: Yup.string().required("Address is required!"),
  gender: Yup.string()
    .typeError("Gender is required!")
    .required("Gender is required!"),
});
```

SecondaryIDProofInfoSchema

Validates secondary ID proof information input.

- **Inputs validated for:** ID number, father's name, date of birth, address, and gender.

Util - help & constant

Constant are some of the fixed values that are not changing in our entire application and if we need to change them we can easily change by making changes in this file, It contains some of validation rules like username length, phone number regex and so on.

```
export const BASE_URL =
  process.env.REACT_APP_API_BASE_URL || "http://localhost:3000";
export const MOCK_API =
  process.env.REACT_APP MOCK_API_BASE_URL ||
  "http://localhost:3000";

export const RESEND_OTP_TIME =
  Number(process.env.REACT_APP_RESEND_OTP_TIME) || 48;

export const USER_NAME_MIN_LENGTH = 2;
export const USER_NAME_MAX_LENGTH = 25;
export const MESSAGE_MAX_LENGTH = 200;
export const USER_NAME_REGEX = /^[a-zA-Z\s]+$/;
export const PHONE_NUMBER_REGEX = /^[6-9][0-9]{9}$/;
export const LOAN_ACCOUNT_NUMBER_REGEX = /^[0-9a-zA-Z]{14,15}$/;

export const ADANI_LOGO_URL =

"https://www.adani.com/-/media/8614C9E10D4A4EC3993153E0208322ED.ashx";
```

Helper function

Certainly, I can help you document the provided helper functions. Below is an explanation of what each code snippet does:

Helper Functions Documentation

These helper functions provide some essential functionality for our application, such as smooth scrolling to the top of the page and downloading PDF files while handling different

platforms effectively.

ScrollToTop function- This function is responsible for scrolling the page to the top whenever the route changes in a React application using React Router. It ensures a smooth transition between different pages within the app.

```
const { pathname } = useLocation();

useEffect(() => { document.documentElement.scrollTo({
  top: 0, left: 0, behavior: "auto", });
  document.body.style.overflow = "visible";
}, [pathname]);
```

DownloadPdf - This function allows you to download and open a PDF file. It checks the platform (native or web) and uses Capacitor's plugins for native platforms to save and open the PDF. For web platforms, it creates a download link to initiate the download.

```
const plt = Capacitor.isNativePlatform();

export const downloadPdf = (pdfBase64: string, fileName: string) => {
  if (plt) {

    try {
      Filesystem.writeFile({
        path: fileName,
        data: pdfBase64,
        directory: Directory.Documents,
      }).then((_writeFileResult) => {
        Filesystem.getUri({
          directory: Directory.Documents,
          path: fileName,
        }).then(
          (getUriResult) => {
            const path = getUriResult.uri;
            // Open the saved PDF file on the device.
            FileOpener.open({ filePath: path, contentType:
"application/pdf" })
              .then(() => console.log("File is opened"))
              .catch((error: any) =>
                console.log("Error opening file", error)
              );
          },
          (error) => {
            console.log(error);
          }
        );
      });
    }
  }
}
```

```

        }
    );
});
} catch (error) {
    console.error("Unable to write file", error);
}
}
};

```

UploadImage - Function is used for upload and processing the image files selected via an HTML input element. It checks if the selected file is an image, reads the image data, and triggers callbacks for success or error with image-related information.

```

type ErrorCallbackType = (msg: string) => void;
type SuccessCallbackType = (imagePath: string, imageData?: FormData) => void;

export const uploadImage = (
    e: React.ChangeEvent<HTMLInputElement>,
    onError?: ErrorCallbackType,
    onSuccess?: SuccessCallbackType
) => {
    const image_file = e.target.files?.[0];

    if (!image_file) {
        onError?.("Please select an image!");
        return null;
    }

    if (!image_file.type.startsWith("image/")) {
        onError?.("Please select a valid image file!");
        return null;
    }

    const reader = new FileReader();
    reader.readAsDataURL(image_file);
    reader.onload = () => {
        const img = new Image();
        img.onload = () => {
            const path = URL.createObjectURL(image_file);
            const data = new FormData();
            data.append("file", image_file);

            onSuccess?.(path, data);

```

```

};
img.onerror = () => {
  onError?.("Invalid image content!");
};
img.src = reader.result as string;
};
};

```

ToIndianCurrency - This function is used for formatting a number or string value into an Indian currency representation. It uses the `toLocaleString` method with specific options to achieve this formatting.

```

export const toIndianCurrency = function (
  val: string | number,
  maxFractionDigits = 0
): string {
  return (+val).toLocaleString("en-IN", {
    style: "currency",
    currency: "INR",
    maximumFractionDigits: maxFractionDigits,
  });
};

```

CleanClassName - This function is used for cleaning class names by removing extra spaces and trimming the string. It's useful for ensuring proper formatting of class names in HTML and CSS.

GetNameInitial - This function extracts the initials latter of a given username. It takes the first letter of each word in the username, capitalizes them, and returns the first two characters as initials.

```

export const getNameInitial = (username: string): string => {
  return username
    .replace(/\w+\s|\w+/g, (s) => s[0].toUpperCase())
    .trim()
    .slice(0, 2);
};

```

We have written some test cases for the `getNameInitial` function for example:

```
import { getNameInitial } from ".";

test("Get Username initial of jhon, which is result into J", () => {
    expect(getNameInitial("jhon")).toBe("J");
});
```

Mocks

Mock data is some static data that does not change and used by our application show some fixed values.

Branch list

Location List, lists all the states location where the facility to take loan is available and branch list is list of all the branches available in that particular state mock.

```
import {
  BranchListType,
  LocationDataType,
} from "@store/slice/branchLocator/type";

export const locationList: LocationDataType = [
  "Madhya Pradesh",
  "Andhra Pradesh",
  "Karnataka",
  "Maharashtra",
  "Rajasthan",
  "Tamil Nadu",
];

export const branchList: BranchListType = [
  {
    location: "Vadodara-ACPL/AHFPL",
    address:
      "3rd floor, Sun complex, Race course, Baroda (beside Citi
bank) 300018",
  },
  {
    location: "Vadodara-ACPL/AHFPL",
    address:
      "3rd floor, Sun complex, Race course, Baroda (beside Citi
bank) 300018",
  },
  {
```



```
        location: "Vadodara-ACPL/AHFPL",
        address:
        "3rd floor, Sun complex, Race course, Baroda (beside Citi
bank) 300018",
    },
];
```

Assets

Assets have all the static images, fonts, and icons.

Icons - applications uses many icons and that is good for user interface and attract a lot of attention then just using static text for every sections, In icons folder there are svg's and png's.

Images - images mostly contains all the png's we use in sections of our application like dashboard and PreOwnTractor

Lotties - Lottie is a library created by Airbnb to facilitate the creation of animations for different platforms.

A Lottie file is composed of a JSON file. This means that, if you have knowledge of this type of file, you can modify both its appearance and its variables (although there are also applications that allow you to modify them in an easier way).

Lottie files work on any device and can be scaled without loss of quality, just like an SVG file.

```
<Lottie
  animationData={discountTag}
  style={{ width: 24, height: 24 }}
/>
```

Internationalization

Making our product reach an audience beyond our geographical boundaries, and in accordance with local cultures, requires a process known as internationalization.

We are using i18next library it offers some of the many features available by *i18next* are loading translations on demand through Ajax calls, caching, plurals, namespacing, and browser language detection.

We first create out translation files in locale folder we put out newly created translation JSON files with the translation keys.

In addition to translation files, we'll also need a config file for the internationalization. This config file will be used to manage the i18n features we want to use. To do so, let's create an i18n folder under the src directory, and within it an file.

```
i18n

.use(LanguageDetector)
.use(initReactI18next)
.use(HttpApi)
.init({
  backend: {
    loadPath: "/locales/{{lng}}/{{ns}}.json",
  },
  debug: false,
  fallbackLng: "en",
  fallbackNS: "common",
  ns: [
    "common",
    "dashboard",
    "farmAdvisory",
    "helpAndSupport",
    "mandiPrice",
    "onBoarding",
    "preOwnTractor",
    "quickPay",
    "setting",
    "termsAndCondition",
  ],
  detection: {
    order: [
      "querystring",
      "cookie",
      "localStorage",
      "sessionStorage",
      "navigator",
      "htmlTag",
      "path",
      "subdomain",
    ],
    caches: ["cookie"],

    cookieOptions: { path: "/", sameSite: "strict" },
    lookupQuerystring: "lng",
```

```
    },  
  
    interpolation: {  
      escapeValue: false, // not needed for react as it escapes  
by default  
    },  
  });
```

Our configuration file accepts a range of options. In our application, however, we'll only use some of them. like interpolation, debug, fallbackLng, ns, detection etc.

Interpolation

Interpolation is one of the most used features in internationalization. This feature allows us to load dynamic values within our translation keys. In our case by using React, the value `scapeValue` don't need to be defined as *true*.

fallbackLng

Language to be loaded in case the default user language could not be found