

# How to build a GenAI App Using React

Let's build a Data App using the SEMOSS SDK.



## Report Generator

AI-Powered Status Report Generator curated to generate a summarized report in seconds.

### Step 1. User Role

Program Manager






### Step 2. Report Sections

- ☐ (Select All)
- ☒ Executive Summary
- ☒ Scope
- ☒ Schedule
- ☒ Cost
- ☐ Sentiment Analysis
- ☐ Recommendations

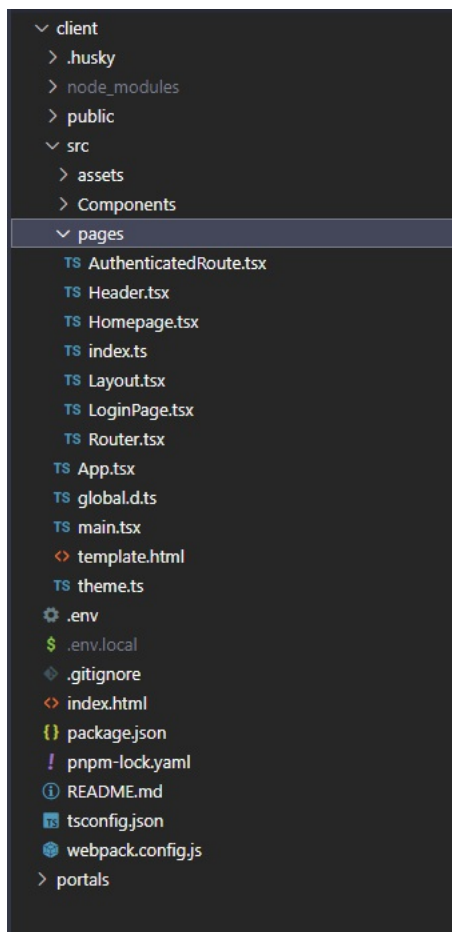
### Step 3. Select File

Enter Report

To start, let's understand what an app is:  
It is a hosted front end with the potential to have some custom backend functionality that is deployed on the Cfg.AI Server.  
The basic file structure can be seen below:

 classes	8/17/2023 2:18 PM	File folder
 client	8/17/2023 2:18 PM	File folder
 java	8/17/2023 2:18 PM	File folder
 portals	8/17/2023 2:18 PM	File folder
 py	8/17/2023 2:18 PM	File folder

For this sample app we will only be focusing on the client and portals folder. There will be further documentation for creating a Java or Python folder, which are largely only necessary for custom reactors and functionality.



This is our Git Structure. We have all of our frontend code within the client folder, including our routes and login, which have been converted to using the SEMOSS SDK.

## Connecting to a CFG Instance

Work in Progress

## Incorporating SDK into Application

Previously we mentioned how to wrap your app in the InsightProvider. Part of the SEMOSS SD package is also the useInsight() hook. This will give us access to the SEMOSS backend data without having to set up a separate store structure. To put this another way, InsightProvider takes the place of the normal store that a React Single Page App would utilize.

To see the useInsight hook in action, we can look at our Router.tsx.

```
export const Router = () => {  
  const { isInitialized, error } = useInsight();  
  
  const [state, setState] = useState({  
    action: history.action,  
    location: history.location,  
  });  
  
  useEffect(() => history.listen(setState), [history]);  
  
  // don't load anything if it is pending  
  if (!isInitialized) {  
    return (  
      <StyledContainer>  
        <CircularProgress />  
      </StyledContainer>  
    );  
  }  
  
  if (error) {  
    return <>Error</>;  
  }  
}
```

Here we see that we are getting initialization and error messaging straight from the backend with the useInsight hook.

The other important functionality that useInsight() provides us with is the ability to make pixel calls. It is worthwhile to look inside your node modules for the specific actions that useInsight() has access to. The most used however, are login, logout, and run. Run makes a pixel call using the standardized string structure that most SEMOSS projects utilize.

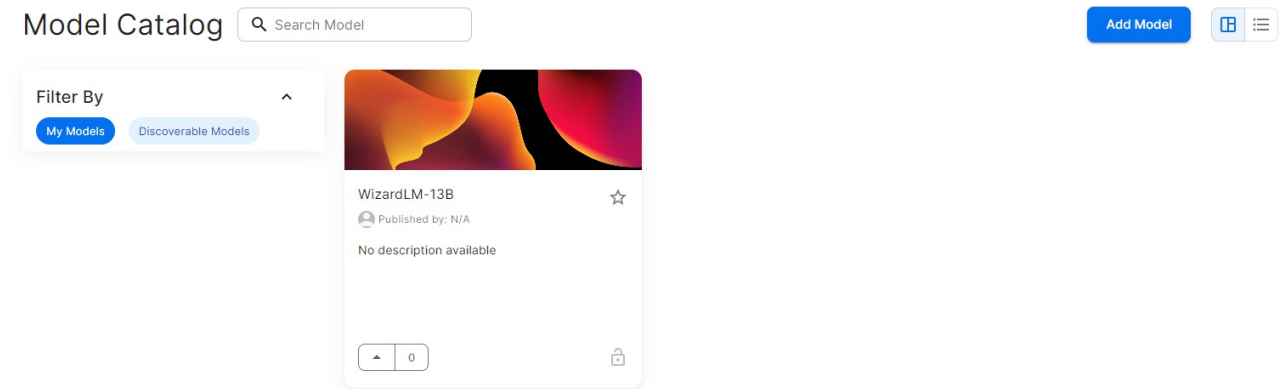
You can use the data appsgit repository to get a sense of how to use the SDK.

<https://repo.semoos.org/data-app>

### After the SDK has been integrated into your Data App

Now that we have replaced our previous store and pixel calls with the SDK, we are finally ready to create our portals folder, zip our folders and upload our app to CFG.ai.

There are some important things to note when using the useInsight() hook. Mainly that this has access to the databases and models that you have stored in your CFG instance or your local server. Therefore, if you want access to a LLM or a storage catalog, the first step you have to take is to upload those in your CFG.ai instance.



The same is true for any storage catalogs that you intend on using. The SDK connects your data app to your CFG instance, using it as a backend. Therefore, accessing your data also requires running pixel API calls. For example, accessing all of your models would look like this:

```
let pixel = `MyEngines ( engineTypes=["MODEL"]);`
```

CFG has a reactor that allows you to call any LLM that it has stored. The pixel call for this requires that you specify an engine, instructions, and a prompt.

Here's an example pixel string for accessing a uploaded LLM.

```
let pixel = `LLM(engine=${JSON.stringify(selectedModel.database_id)}, context=${JSON.stringify(instruction)}, command=${JSON.stringify(data.prompt)}) ;`
```

## Bundling the Gen AI App

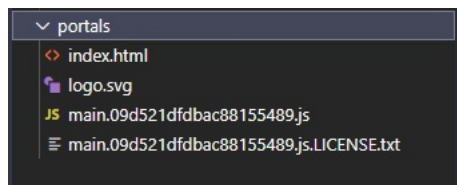
### Creating the portals folder

In the Getting Started section we mentioned that whatever compiler option you use, that you should make sure that the output goes to a portals folder. This makes creating the portals folder very easy. Our data app is using webpack, so running

```
pnpm build
```

within the client folder should be all that is needed to automatically generate the portals folder.

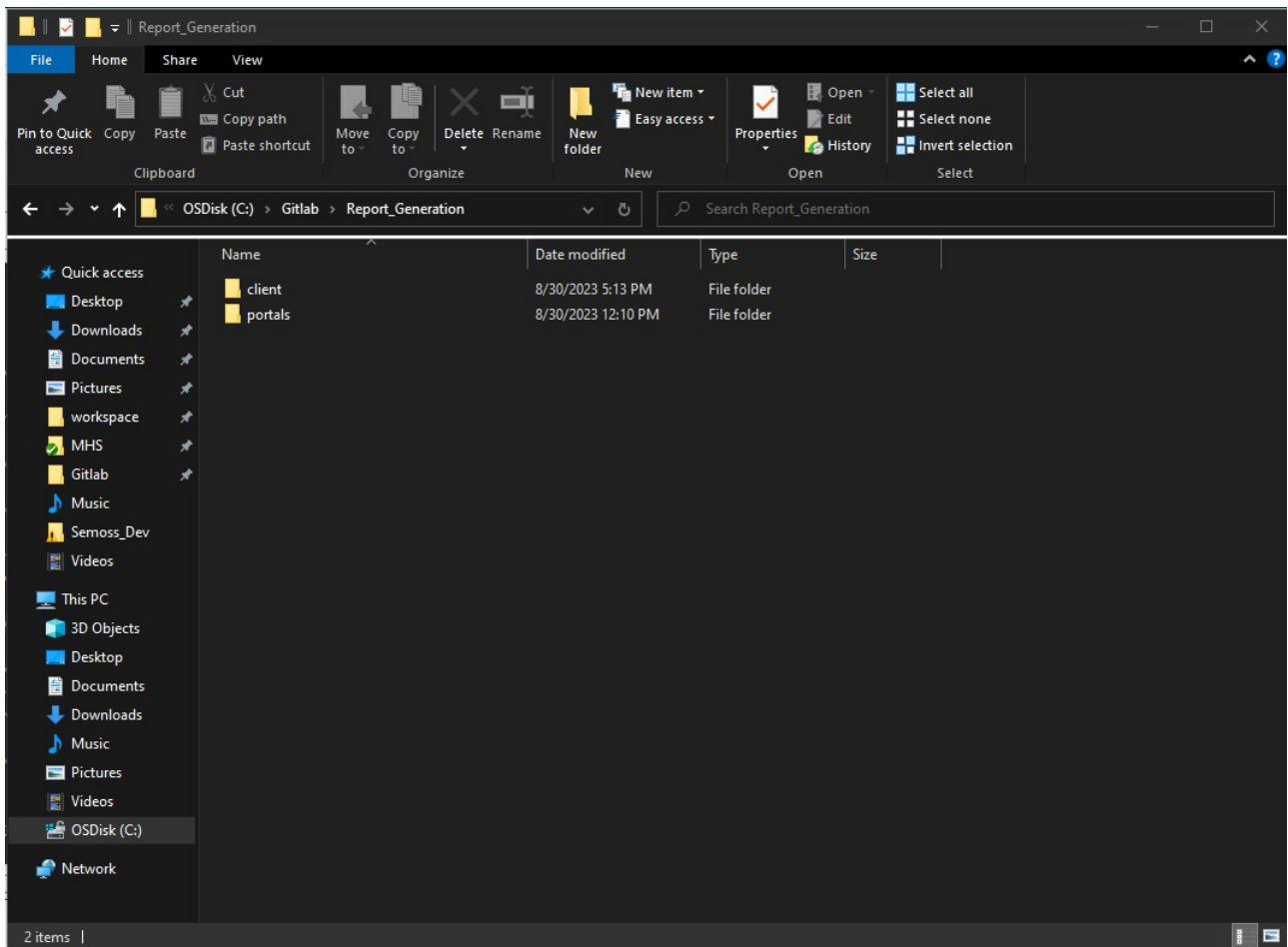
Our sample app's portals folder looks like the following:



### Bundling your app

Now that we have the App's portal folder created we want to delete node\_modules from the client root directory. This is because they will be rebuilt when we upload it into CFG.ai.

Next we need to compress the files so that the .zip file opens up to the directory containing our client file. For our app this meant only these files need to get zipped:

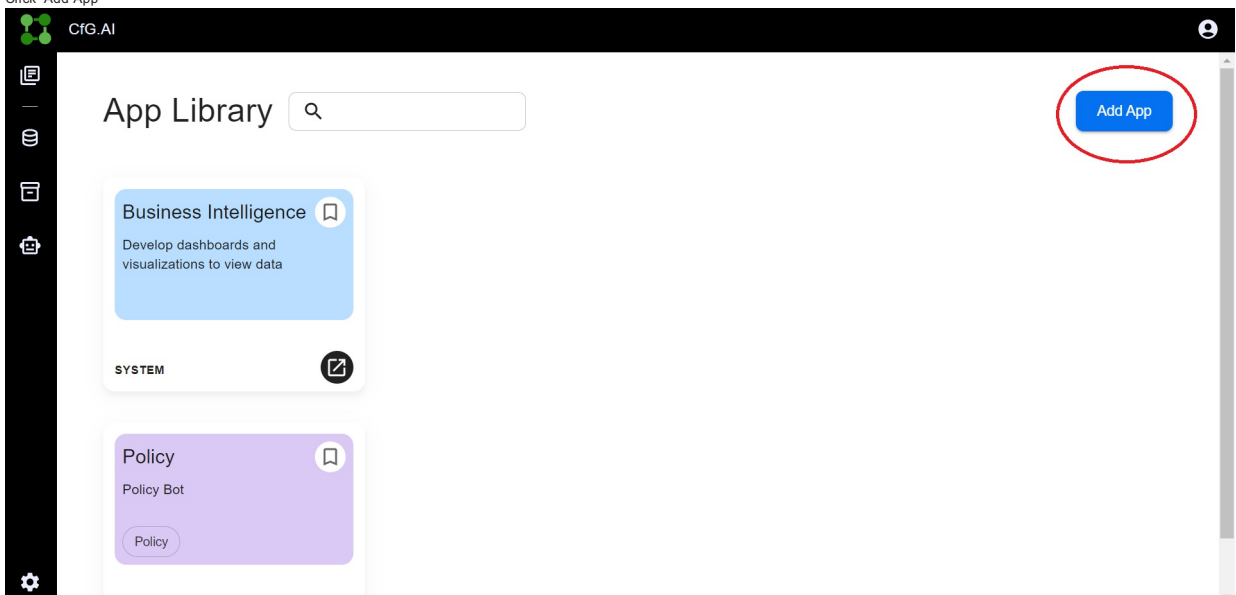


You are now ready to upload your add.

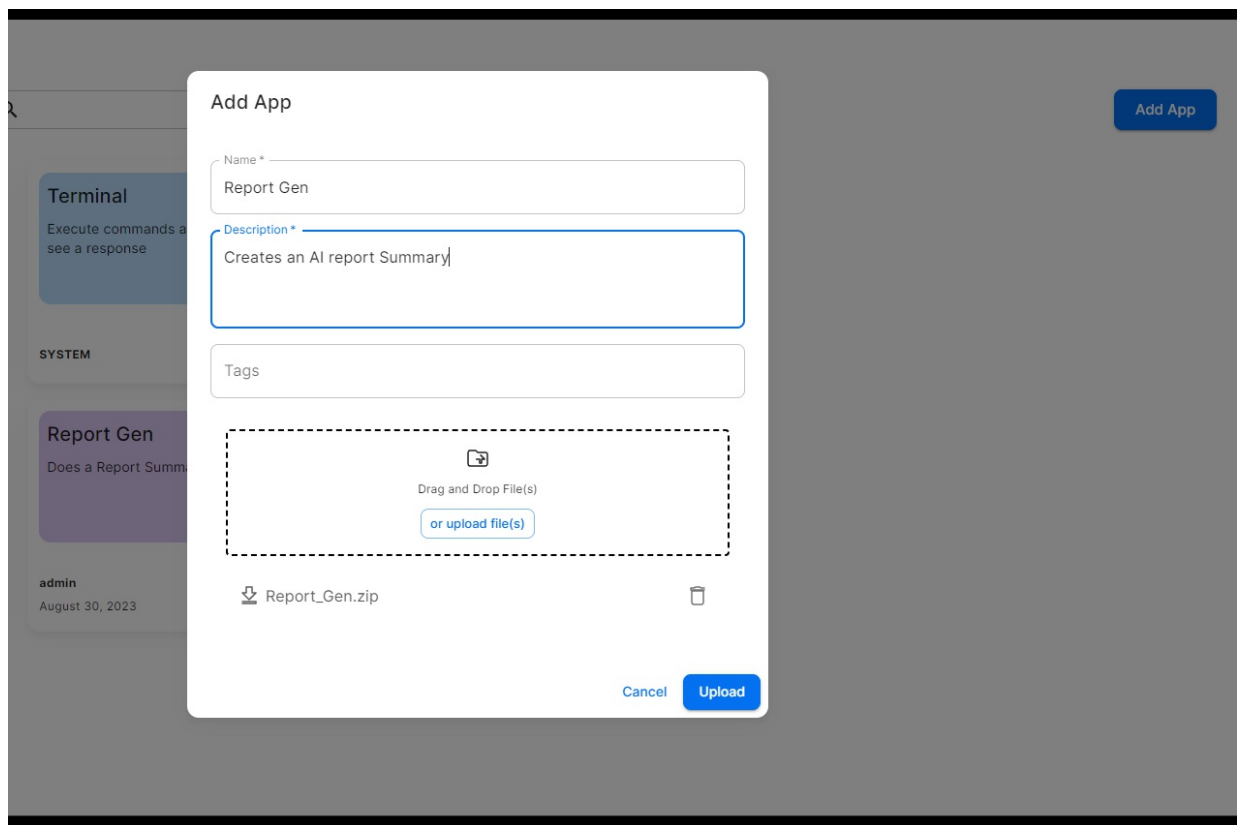
### Adding your Gen AI App to your App Catalog

Once you have added all your files into the structure:

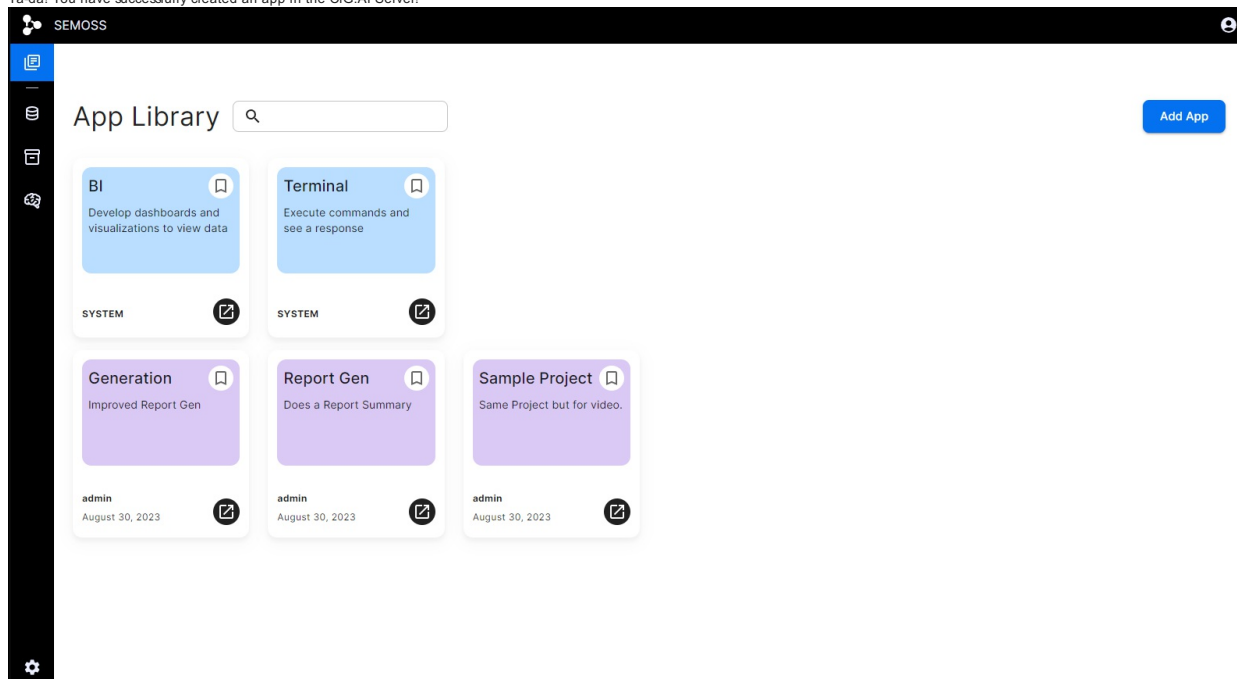
1. Navigate to <https://workshop.cfg.deloitte.com/cfg-ai-dev/SemossWeb/#/>
2. Click "Add App"



3. Fill in the required fields



4. Upload your zip file
5. Click "Upload" (Note: this may take a few minutes to spin up)
6. Ta-da! You have successfully created an app in the CFG.AI Server!



7. Note the final portion of the url when you navigate to your app. This will be the APP id that is necessary for local development.