**Simplifying User Authentication with Django and React: A Guide to Google Sign-In**

[[Arjun Prakash](https://medium.com/@arjunprakash027?source=post_page-----5d44b25485ac--------------------------------)](https://medium.com/@arjunprakash027?source=post_page-----5d44b25485ac--------------------------------)

[Arjun Prakash](https://medium.com/@arjunprakash027?source=post_page-----5d44b25485ac--------------------------------)

·

[Follow](https://medium.com/m/signin?actionUrl=https%3A%2F%2Fmedium.com%2F_%2Fsubscribe%2Fuser%2F376a20930ef1&operation=register&redirect=https%3A%2F%2Fmedium.com%2F%40arjunprakash027%2Fsimplifying-user-authentication-with-django-and-react-a-guide-to-google-sign-in-5d44b25485ac&user=Arjun+Prakash&userId=376a20930ef1&source=post_page-376a20930ef1----5d44b25485ac---------------------post_header-----------)

19 min read

·

Jun 8

52

**Introduction**

In today’s digital landscape, user authentication is a crucial aspect of web development. As developers, we strive to provide seamless and secure login experiences for our users. One popular authentication method is Google Sign-In, which allows users to access your application using their Google credentials. In this blog post, we’ll explore how to integrate Google Sign-In into a Django and React application, combining the robust backend capabilities of Django with the dynamic and interactive frontend of React. By the end of this guide, you’ll have a solid understanding of how to implement Google Sign-In and provide a seamless authentication experience for your users. So, let’s dive in and discover the power of Django-React Google Sign-In!

*Code link for project at the end of this blog*

**Assumptions**

*a. Familiarity with JavaScript and Python*

*b. Installation of Node.js*

*c. Installation of Python*

**Why Use Google Sign-In**

1. **Simplified User Experience**: Users can sign in to your application using their existing Google accounts, eliminating the need for creating new credentials. This streamlined process reduces friction and encourages user engagement.
2. **Enhanced Security**: Google Sign-In leverages Google’s robust security infrastructure, ensuring the protection of sensitive user information. By offloading authentication to Google, you alleviate concerns about storing and managing user credentials on your own server.
3. **Wide User Adoption**: Google has a massive user base, and by integrating Google Sign-In, you tap into this extensive audience. Users can easily access your application without the hassle of creating a new account, increasing adoption rates.
4. **Cross-Platform Compatibility**: Google Sign-In works seamlessly across web, mobile, and desktop applications. This allows users to authenticate with their Google accounts from various devices and platforms, delivering a consistent experience.

**React Overview and Installation**

React is a popular JavaScript library for building user interfaces. Its component-based architecture and virtual DOM make it efficient and flexible for creating interactive web applications.

React allows you to break down your UI into reusable components, making development more modular and maintainable. It efficiently updates and renders only the necessary parts of the UI, resulting in improved performance. React also provides a rich ecosystem of tools and libraries to enhance development productivity.

**To install React and set up your Django-React project, follow these steps:**

a. Ensure that you have Node.js installed on your machine. You can download it from the official Node.js website ([https://nodejs.org](https://nodejs.org/)) and follow the installation instructions for your operating system.

b. Once Node.js is installed, open your terminal or command prompt and navigate to your project directory.

c. Use the following command to create a new React application using Create React App:

npm install create-react-app

This code will install create-react-app in your system

npx create-react-app my-app

This will create a react app with all dependencies.

Start the React development server with the following command:

cd my-app

npm start

This will launch your React application and provide a local development server for testing and development.

You can now begin building your React components and integrating them with your Django backend to implement the Google Sign-In functionality.

**Django Overview and Installation**

Django is a powerful and popular Python web framework that simplifies the process of building web applications. It follows the model-view-controller (MVC) architectural pattern and emphasizes code reusability and scalability.

Django provides a comprehensive set of tools and libraries for handling various web development tasks, including URL routing, database management, template rendering, and user authentication. It promotes clean and pragmatic design principles, making it an ideal choice for building robust and maintainable web applications.

**To install Django and set up your Django-React project, follow these steps:**

a. Ensure that you have Python installed on your machine. You can download the latest version of Python from the official Python website ([https://www.python.org](https://www.python.org/)) and follow the installation instructions for your operating system.

b. Once Python is installed, open your terminal or command prompt and check if Python is accessible by running the following command:

python --version

c. Create a virtual environment for your Django project to keep dependencies isolated. Run the following command to create a virtual environment:

python -m venv myenv

Replace myenv with the desired name for your virtual environment.

d. Activate the virtual environment using the appropriate command for your operating system:

for Windows:

myenv\Scripts\activate

for macOS/Linux:

source myenv/bin/activate

e. Once the virtual environment is activated, use the following command to install Django:

pip install django

f. Verify the installation by running the following command:

python -m django --version

g. You can now start a new Django project using the following command:

django-admin startproject myproject

Replace myproject with the desired name for your Django project.

h. Navigate into the project directory:

cd myproject

i. Start the development server with the following command:

python manage.py runserver

This will launch your Django application, and you can access it in your web browser at <http://localhost:8000/>.

**Creating the Django Backend**

The name of my project is **backend**and name of my application is**authentication**

You have to install an external library called as djangorestframework to create an API centric django application.

pip install djangorestframework

also install django-cors-headers to allow requests from React frontend.

pip install django-cors-headers

Now go to settings.py in backend folder and navigate to **INSTALLED\_APPS**

and add these to the list:

*“authentication”,(name of your app, I have named it authentication)*

*“rest\_framework”,*

*“rest\_framework.authtoken”,*

*“corsheaders”,*

the list would look something like this:

INSTALLED\_APPS = [  
 "django.contrib.admin",  
 "django.contrib.auth",  
 "django.contrib.contenttypes",  
 "django.contrib.sessions",  
 "django.contrib.messages",  
 "django.contrib.staticfiles",  
 "authentication",  
 "rest\_framework",  
 "rest\_framework.authtoken",  
 "corsheaders",  
]

The “rest\_framework.authtoken” module is utilized to implement token authentication for our API. By employing token authentication, we ensure that only authorized users can access our API, enhancing its security and protecting against unauthorized access.

Add “corsheaders.middleware.CorsMiddleware” in the middleware list

and add a new list called CORS\_ALLOWED\_ORIGINS =[“http://localhost:3000”]

It would look something like this:

MIDDLEWARE = [  
 "django.middleware.security.SecurityMiddleware",  
 "django.contrib.sessions.middleware.SessionMiddleware",  
 "django.middleware.common.CommonMiddleware",  
 "django.middleware.csrf.CsrfViewMiddleware",  
 "django.contrib.auth.middleware.AuthenticationMiddleware",  
 "django.contrib.messages.middleware.MessageMiddleware",  
 "django.middleware.clickjacking.XFrameOptionsMiddleware",  
 "corsheaders.middleware.CorsMiddleware",  
]  
  
CORS\_ALLOWED\_ORIGINS = [  
 "http://localhost:3000",  
]

Add this list to the settings.py file too:

REST\_FRAMEWORK = {  
 "DEFAULT\_AUTHENTICATION\_CLASSES": [  
 "rest\_framework.authentication.TokenAuthentication",  
 ],  
}

This configuration is used to allow token authentication too.

We are now done with all the settings configaration of our application, now lets go to views and urls

**Navigate to views .py in authentication app**

Importing Dependencies: The code begins with importing necessary modules from Django and the Django REST Framework. These modules provide essential functionalities for building the API.

from django.shortcuts import render  
from rest\_framework.decorators import APIView  
from rest\_framework.response import Response  
from rest\_framework.permissions import IsAuthenticated  
from django.contrib.auth.models import User  
from django.contrib.auth.hashers import make\_password

Defining the RegisterNewUser View: The RegisterNewUser class is a Django APIView subclass that handles the user registration process. It accepts a POST request containing the user's username, email, and name. Inside the post method, a new user is created using the User.objects.create\_user method. The user's information is extracted from the request data and saved to the database.

class RegisterNewUser(APIView):  
 def post(self,request):  
 username = request.data.get("username")  
 email = request.data.get("email")  
 name = request.data.get("name")  
   
 try:  
 user = User.objects.create\_user(  
 username = username,  
 password = "random123",  
 email = email,  
 first\_name = name,  
 )  
 user.save()  
 print("{} created successfully".format(user.username))  
 return Response({"message":"User created"})  
 except:  
 return Response({"message":"User creation failed or user already exists"})

Handling User Creation: Upon successful creation of the user, a success message is printed, and a response with the message “User created” is returned. If an exception occurs during user creation, such as a duplicate username or failure to save the user, a response with the message “User creation failed or user already exists” is returned.

Defining the Greeting View: The greeting class is another APIView subclass that handles a GET request. It includes the IsAuthenticated permission class, ensuring that only authenticated users can access the endpoint. The get method returns a personalized greeting message, including the user's first name.

class greeting(APIView):  
 permission\_classes = ( IsAuthenticated, )  
  
 def get(self,request):  
 content = {'message': 'Hello, {}!'.format(request.user.first\_name)}  
 return Response(content)

Token Authentication: Token authentication is automatically handled by the Django REST Framework when the IsAuthenticated permission class is used. This authentication method relies on the rest\_framework.authtoken module, which generates unique tokens for each user. These tokens are sent with subsequent requests for authentication.

**Now making the routings in urls.py**

Create a new urls.py in authentication folder and include these codes.

from django.urls import path  
from .views import greeting,RegisterNewUser  
from rest\_framework.authtoken.views import obtain\_auth\_token  
  
urlpatterns = [  
 path("hello/", greeting.as\_view(),name="greeting"),  
 path("register/", RegisterNewUser.as\_view(),name="register"),  
 path("login/", obtain\_auth\_token,name="create\_token"),  
]

1. Defining URL Patterns: The urlpatterns list is a collection of URL patterns that map specific URLs to corresponding views or actions in the API. Each URL pattern is defined using the path function from Django's urls module.
2. Mapping the “greeting” View: The first URL pattern maps the URL path “hello/” to the greeting view using the greeting.as\_view() method. This associates the "greeting" view with the specified URL path, allowing it to handle incoming requests to that endpoint.
3. Mapping the “RegisterNewUser” View: The second URL pattern maps the URL path “register/” to the RegisterNewUser view using the RegisterNewUser.as\_view() method. This associates the "RegisterNewUser" view with the specified URL path, enabling it to handle incoming requests to that endpoint.
4. Mapping the “obtain\_auth\_token” View: The third URL pattern maps the URL path “login/” to the obtain\_auth\_token view from the rest\_framework.authtoken.views module. This view is provided by the Django REST Framework and handles the generation of authentication tokens for users during the login process.

These URL patterns define the routing for the API endpoints, specifying which views should handle incoming requests at specific URLs. By configuring the URL patterns in this way, you establish the entry points for different API functionalities, such as greeting, user registration, and token-based login.

And finally go to urls.py in backend folder and add this:

from django.contrib import admin  
from django.urls import path,include  
  
urlpatterns = [  
 path("admin/", admin.site.urls),  
 path("api/auth/", include("authentication.urls")),  
]

1. Defining URL Patterns: The urlpatterns list is a collection of URL patterns that map specific URLs to corresponding views or URL configurations. Each URL pattern is defined using the path function from Django's urls module.
2. Mapping the Admin Site: The first URL pattern maps the URL path “admin/” to the Django administration site using the admin.site.urls configuration. This associates the Django administration interface with the specified URL path, allowing administrators to access and manage the site's backend through this URL.
3. Including URL Configurations: The second URL pattern maps the URL path “api/auth/” to an included URL configuration using the include function. The included URL configuration is specified as "authentication.urls", indicating that the URLs for the authentication functionality will be defined in a separate URL configuration file named "authentication.urls".

**Create a superuser to manage backend**

Open the terminal where manage.py is present and type:

python manage.py createsuperuser

1. You will be prompted to enter a username for the superuser. Type a username and press Enter.
2. You will then be prompted to enter an email address for the superuser. Type the email address (optional) and press Enter.
3. Next, you will be prompted to enter a password. Type a secure password and press Enter. Note that the password characters will not be visible as you type.
4. Finally, you will be prompted to confirm the password. Retype the password for confirmation and press Enter.

The createsuperuser command is a built-in Django management command that creates a superuser account. When you run the command, Django prompts you to enter a username, email address (optional), password, and password confirmation. Once you provide this information, Django creates the superuser account in the database.

The superuser account has elevated privileges and can access the Django administration site. It allows you to manage the backend of your Django application, including creating and editing database records, managing users, and performing administrative tasks.

**Obtaining the Token**

To get the token, send a POST request to this endpoint “[http://localhost:8000/api/auth/login/](http://localhost:3000/api/auth/login/)”

The body of the request must be:

{  
"username":your-user-name,  
"password":Your-pass-word  
}

We will get a token on response.

**Using the token**

To test if the token is working, send a GET request to “[http://localhost:8000/api/auth/hello/](http://localhost:3000/api/auth/login/)”

with the header:

"Authorization" : "Token {your-token-from-the-previous-response}"

If you get :  
‘message’: ‘Hello, {your-username}!’, then token is valid and working.

Save it somewhere to use it on every request.

**Setting up google sign-in in google cloud**

follow this blog to setup google sign-in in google cloud. This is a mandatory step, do not skip this

[**The guide to adding Google login to your React app — LogRocket Blog**](https://blog.logrocket.com/guide-adding-google-login-react-app/)

**Designing the Frontend React Website**

Start a react app with this command (I will name my application frontend)

npx create-react-app frontend

A folder would have been created with all the react dependencies

cd into frontend folder and run this command

npm i axios  
npm install @react-oauth/google@latest

Axios is a JavaScript library used for making HTTP requests from a web browser or Node.js. It simplifies the process of sending asynchronous HTTP requests, handling responses, and managing data transfer between the client and server. With its intuitive and concise API, Axios provides a convenient way to perform HTTP operations, such as retrieving data from APIs or submitting form data, making it a popular choice for developers working with JavaScript-based applications.

Navigate to *src* folder and create a new folder *components*

**Inside components create a new file called *navigation\_bar.js***

Add this code inside the file

import React from 'react';  
  
function Navbar({ profile, logout }) {  
 return (  
 <nav>  
 <div className="logo">  
 <img src={profile['picture']} alt={profile['name']} />  
 <h2>Welcome, {profile['name']}!</h2>  
 </div>  
 <div className="profile-info">  
 <h3>Mail ID: {profile['email']}</h3>  
 <button onClick={logout}>Logout</button>  
 </div>  
 </nav>  
 );  
}  
  
export default Navbar;

1. Importing Dependencies: The code begins with importing the necessary dependencies. In this case, the React module is imported, which is required for building React components.
2. Defining the Navbar Component: The Navbar function defines a functional component in React. It takes two props as parameters: profile and logout.
3. Rendering the Navbar HTML: Inside the component, the return statement contains the JSX code that will be rendered as the HTML structure of the Navbar. The JSX code is wrapped in parentheses and follows the rules of JSX syntax.
4. Creating the Navbar Structure: The Navbar is structured using HTML tags. The outermost element is a nav tag, representing the navigation section of the webpage.
5. Displaying the Logo and Profile Information: Within the Navbar, there are two main sections: the logo and the profile information. The logo section is contained within a div with the class name "logo". It includes an img tag that displays the profile picture fetched from the profile prop and an h2 tag that displays a welcome message using the profile prop's name.
6. Displaying the Profile Information: The profile information section is contained within a div with the class name "profile-info". It includes an h3 tag that displays the user's email address retrieved from the profile prop. Additionally, there is a button tag with an onClick event listener that triggers the logout function passed as a prop.
7. Exporting the Navbar Component: The export default statement exports the Navbar component so that it can be imported and used in other parts of the application.

**Now create another file called UserPage.js and add this code**

import React from 'react';  
import { useState, useEffect } from 'react';  
import axios from 'axios';  
  
function UserPage({ profile }) {  
  
 const [token, setToken] = useState(null);  
 const [data , setData ] = useState(null);  
  
 useEffect(() => {  
  
 const delay = 3000;  
  
 const credentials = {  
 username: profile['id'],  
 password: 'random123'  
 };  
   
 const timeout = setTimeout(() => {  
 axios.post('http://127.0.0.1:8000/api/auth/login/', credentials)  
 .then((response) => {  
 console.log(response['data']);  
 setToken(response['data']['token']);  
 });  
 }, delay);  
   
 return () => clearTimeout(timeout);  
 }, []); // Empty dependency array to run the effect only once  
  
 useEffect(() => {  
  
 axios.get("http://127.0.0.1:8000/api/auth/hello/",{  
 headers: {  
 'Content-Type': 'application/json',  
 'Authorization': 'Token '+token  
 }  
 }).then((response) => {  
 console.log(response['data']);  
 setData(response['data']['message']);  
 }, (error) => {  
 console.log(error);  
 }  
 );  
 }, [token]);  
  
 return(  
 <div>  
 <h2> User Page </h2>  
 <br />  
 <p> {data} </p>  
 </div>  
 );  
}  
  
export default UserPage;

**Step by Step Explanation :**

Importing Dependencies: The code begins with importing the necessary dependencies. It imports the React module for building React components, the useState and useEffect hooks from React, and the axios library for making HTTP requests.

import React from 'react';  
import { useState, useEffect } from 'react';  
import axios from 'axios';

Defining the UserPage Component: The UserPage function defines a functional component in React. It takes the profile prop as a parameter.

function UserPage({ profile })

Setting Up State Variables: Inside the component, two state variables are defined using the useState hook: token and data. The token variable is initialized as null, and the data variable is also initialized as null.

const [token, setToken] = useState(null);  
const [data , setData ] = useState(null);

Performing Side Effects with useEffect: The first useEffect hook is used to perform a side effect when the component mounts. It sets a timeout of 3000 milliseconds (3 seconds) using setTimeout and then makes a POST request to the specified URL(http://127.0.0.1:8000/api/auth/login/) with the provided credentials. The response from the API is logged to the console, and the token value is set using the setToken function from the useState hook. The useEffect hook is configured to run only once by passing an empty dependency array ([]). setTimeout is used to not send subsequent post request as soon as a new user has been logged in because it takes around 3 seconds for the backend to register the new user.

useEffect(() => {  
  
const delay = 3000;  
  
const credentials = {  
 username: profile['id'],  
 password: 'random123'  
};  
  
const timeout = setTimeout(() => {  
axios.post('http://127.0.0.1:8000/api/auth/login/', credentials)  
 .then((response) => {  
 console.log(response['data']);  
 setToken(response['data']['token']);  
 });  
}, delay);  
  
return () => clearTimeout(timeout);  
}, []);

Making an Authenticated GET Request: The second useEffect hook is used to make an authenticated GET request to the specified URL (http://127.0.0.1:8000/api/auth/hello/). It includes the token value in the request headers for authorization. The response from the API is logged to the console, and the data state variable is updated with the message obtained from the response. The useEffect hook is configured to run whenever the token value changes.

useEffect(() => {  
  
axios.get("http://127.0.0.1:8000/api/auth/hello/",{  
 headers: {  
 'Content-Type': 'application/json',  
 'Authorization': 'Token '+token  
 }  
}).then((response) => {  
 console.log(response['data']);  
 setData(response['data']['message']);  
 }, (error) => {  
 console.log(error);  
 }  
);  
}, [token]);

Rendering the UserPage HTML: The return statement contains the JSX code that will be rendered as the HTML structure of the UserPage component. It consists of a div element containing an h2 heading with the text "User Page" and a p element that displays the data value obtained from the API.

useEffect(() => {  
  
axios.get("http://127.0.0.1:8000/api/auth/hello/",{  
 headers: {  
 'Content-Type': 'application/json',  
 'Authorization': 'Token '+token  
 }  
}).then((response) => {  
 console.log(response['data']);  
 setData(response['data']['message']);  
 }, (error) => {  
 console.log(error);  
 }  
);  
}, [token]);

Rendering the UserPage HTML: The return statement contains the JSX code that will be rendered as the HTML structure of the UserPage component. It consists of a div element containing an h2 heading with the text "User Page" and a p element that displays the data value obtained from the API.

return(  
<div>  
 <h2> User Page </h2>  
 <br />  
 <p> {data} </p>  
</div>  
);

Exporting the UserPage Component: The export default statement exports the UserPage component so that it can be imported and used in other parts of the application.

export default UserPage;

By following the steps outlined above, you can create and use the UserPage component in your React application. This component performs an initial login request to obtain a token and then uses that token to make authenticated GET requests to retrieve data from the API.

**Now navigate to App.js in src folder and add this code**

import React, { useState, useEffect } from 'react';  
import { googleLogout, useGoogleLogin } from '@react-oauth/google';  
import Navbar from './components/navigation\_bar';  
import UserPage from './components/UserPage';  
import axios from 'axios';  
import './App.css';  
  
function App() {  
 const [user, setUser] = useState(null);  
 const [profile, setProfile] = useState(null);  
  
 const login = useGoogleLogin({  
 onSuccess: (codeResponse) => setUser(codeResponse),  
 onError: (error) => console.log('login failed:', error)  
 });  
  
 useEffect(() => {  
 if (user) {  
 axios  
 .get(`https://www.googleapis.com/oauth2/v1/userinfo?alt=json&access\_token=${user['access\_token']}`, {  
 headers: {  
 'Content-Type': 'application/json',  
 Authorization: `Bearer ${user['access\_token']}`  
 }  
 })  
 .then(  
 (response) => {  
 console.log('changes');  
 setProfile(response['data']);  
 },  
 (error) => {  
 console.log(error);  
 }  
 );  
 }  
 }, [user]);  
  
 useEffect(() => {  
 if (profile === null) {  
 console.log('profile null');  
 } else {  
 console.log('sending data to django');  
  
 const profileData = {  
 name: profile['name'],  
 email: profile['email'],  
 username: profile['id']  
 };  
  
 console.log(profileData);  
  
 axios.post('http://127.0.0.1:8000/api/auth/register/', profileData).then(  
 (response) => {  
 console.log(response['data']);  
 },  
 (error) => {  
 console.log(error);  
 }  
 );  
 }  
 }, [profile]);  
  
 const logout = () => {  
 googleLogout();  
 setProfile(null);  
 };  
  
 if (profile) {  
 console.log('profile present');  
 console.log(profile);  
 }  
  
 return (  
 <div>  
 {profile ? (  
 <div>  
 <Navbar profile={profile} logout={logout} />  
 <UserPage profile={profile} />  
 </div>  
 ) : (  
 <div className="login-container">  
 <h2 className="login-heading">Google Login</h2>  
 <button className="login-button" onClick={login}>  
 Sign in with Google  
 </button>  
 </div>  
 )}  
 </div>  
 );  
}  
  
export default App;

In this code the main magic of google sign in happens

**Step by Step Explanation:**

The App function defines the main component of the application. It initializes two state variables using the useState hook: user and profile. The user state variable represents the user's login information, and the profile state variable represents the user's profile data obtained from Google.

function App() {  
 const [user, setUser] = useState(null);  
 const [profile, setProfile] = useState(null);

Handling Google Login: The login constant uses the useGoogleLogin hook to handle the Google login functionality. It specifies the onSuccess and onError callbacks to handle the successful login and error scenarios, respectively.

const login = useGoogleLogin({  
 onSuccess: (codeResponse) => setUser(codeResponse),  
 onError: (error) => console.log('login failed:', error)  
 });

Fetching User Profile Data: The first useEffect hook is used to fetch the user's profile data from the Google API. It makes a GET request to the specified URL (https://www.googleapis.com/oauth2/v1/userinfo?alt=json&access\_token=${user['access\_token']}) using the axios library. The access token obtained from the user's login response is appended to the URL for authentication purposes. The retrieved profile data is stored in the profile state variable.

useEffect(() => {  
 if (user) {  
 axios  
 .get(`https://www.googleapis.com/oauth2/v1/userinfo?alt=json&access\_token=${user['access\_token']}`, {  
 headers: {  
 'Content-Type': 'application/json',  
 Authorization: `Bearer ${user['access\_token']}`  
 }  
 })  
 .then(  
 (response) => {  
 console.log('changes');  
 setProfile(response['data']);  
 },  
 (error) => {  
 console.log(error);  
 }  
 );  
 }  
 }, [user]);

Sending Profile Data to Django: The second useEffect hook is used to send the user's profile data to a Django API endpoint for registration. It checks if the profile is not null and then creates a profileData object with the necessary profile information. It makes a POST request to the specified URL (http://127.0.0.1:8000/api/auth/register/) using the axios library to register the user's profile data.

useEffect(() => {  
 if (profile === null) {  
 console.log('profile null');  
 } else {  
 console.log('sending data to django');  
  
 const profileData = {  
 name: profile['name'],  
 email: profile['email'],  
 username: profile['id']  
 };  
  
 console.log(profileData);  
  
 axios.post('http://127.0.0.1:8000/api/auth/register/', profileData).then(  
 (response) => {  
 console.log(response['data']);  
 },  
 (error) => {  
 console.log(error);  
 }  
 );  
 }  
 }, [profile]);

Handling Logout: The logout function is invoked when the user clicks the logout button. It calls the googleLogout function to perform the logout operation and sets the profile state variable to null.

const logout = () => {  
 googleLogout();  
 setProfile(null);  
 };

Rendering the App HTML: The return statement contains the JSX code that will be rendered as the HTML structure of the App component. The code uses conditional rendering to display different content based on whether the profile is present or not. If the profile is present, the Navbar and UserPage components are rendered. If the profile is null, a login container with a Google login button is rendered

return (  
 <div>  
 {profile ? (  
 <div>  
 <Navbar profile={profile} logout={logout} />  
 <UserPage profile={profile} />  
 </div>  
 ) : (  
 <div className="login-container">  
 <h2 className="login-heading">Google Login</h2>  
 <button className="login-button" onClick={login}>  
 Sign in with Google  
 </button>  
 </div>  
 )}  
 </div>  
 );  
}

Exporting the App Component: The export default statement exports the App component so that it can be imported and used in other parts of the application.

export default App;

By following the steps outlined above, you can create and use the App component in your React application. This component handles Google login, fetches user profile data, sends profile data to a Django API endpoint, and provides the functionality to logout.

Adding the styles(this is an optional step)

Go to App.css in src folder and add this code

body {  
 margin: 0;  
 padding: 0;  
 font-family: Arial, sans-serif;  
 background-color: #f8f8f8;  
}  
  
.App {  
 text-align: center;  
}  
  
.App-logo {  
 height: 40vmin;  
 pointer-events: none;  
}  
  
@media (prefers-reduced-motion: no-preference) {  
 .App-logo {  
 animation: App-logo-spin infinite 20s linear;  
 }  
}  
  
.App-header {  
 background-color: #282c34;  
 min-height: 100vh;  
 display: flex;  
 flex-direction: column;  
 align-items: center;  
 justify-content: center;  
 font-size: calc(10px + 2vmin);  
 color: white;  
}  
  
.App-link {  
 color: #61dafb;  
}  
  
@keyframes App-logo-spin {  
 from {  
 transform: rotate(0deg);  
 }  
 to {  
 transform: rotate(360deg);  
 }  
}  
  
nav {  
 display: flex;  
 align-items: center;  
 justify-content: space-between;  
 background-color: #333;  
 color: #fff;  
 padding: 10px 20px;  
}  
  
.logo {  
 display: flex;  
 align-items: center;  
}  
  
.logo img {  
 width: 30px;  
 height: 30px;  
 border-radius: 50%;  
 margin-right: 8px;  
}  
  
.logo h2 {  
 font-size: 18px;  
 margin: 0;  
 color: #fff;  
}  
  
.dropdown {  
 position: relative;  
}  
  
.dropdown-button {  
 display: flex;  
 align-items: center;  
 cursor: pointer;  
 position: relative;  
}  
  
.dropdown-button span {  
 display: block;  
 width: 20px;  
 height: 2px;  
 background-color: #fff;  
 margin-bottom: 4px;  
}  
  
.dropdown-menu {  
 position: absolute;  
 top: 100%;  
 left: 0;  
 display: none;  
 list-style: none;  
 padding: 8px 0;  
 background-color: #333;  
 margin-top: 8px;  
 border-radius: 4px;  
 box-shadow: 0 2px 4px rgba(0, 0, 0, 0.2);  
}  
  
.dropdown.open .dropdown-menu {  
 display: block;  
}  
  
.dropdown-menu li {  
 margin-right: 8px;  
}  
  
.dropdown-menu a {  
 display: block;  
 text-decoration: none;  
 color: #fff;  
 font-weight: bold;  
 padding: 8px 16px;  
}  
  
.profile-info {  
 display: flex;  
 align-items: center;  
}  
  
.profile-info h3 {  
 margin-right: 8px;  
 color: #fff;  
}  
  
button {  
 padding: 6px 12px;  
 background-color: #fff;  
 color: #333;  
 border: none;  
 border-radius: 4px;  
 cursor: pointer;  
}  
  
/\* Responsive Styles \*/  
  
@media screen and (max-width: 768px) {  
 nav {  
 flex-wrap: wrap;  
 }  
  
 .dropdown {  
 order: 2;  
 margin-top: 10px;  
 }  
  
 .profile-info {  
 order: 1;  
 margin-top: 10px;  
 }  
}  
  
button {  
 padding: 6px 12px;  
 background-color: #fff;  
 color: #333;  
 border: none;  
 border-radius: 4px;  
 cursor: pointer;  
}  
  
.container {  
 max-width: 1200px;  
 margin: 0 auto;  
 padding: 20px;  
}  
  
/\* Additional Styles \*/  
  
.login-container {  
 display: flex;  
 flex-direction: column;  
 align-items: center;  
 justify-content: center;  
 height: 100vh;  
}  
  
.login-heading {  
 font-size: 24px;  
 margin-bottom: 16px;  
 color: #333;  
}  
  
.login-button {  
 padding: 10px 20px;  
 background-color: #61dafb;  
 color: #fff;  
 border: none;  
 border-radius: 4px;  
 cursor: pointer;  
 font-size: 16px;  
 transition: background-color 0.3s ease;  
}  
  
.login-button:hover {  
 background-color: #45b6d9;  
}

This just beautify your components

**Run the application**

To get started, navigate to the Django project folder containing the manage.py file. Open your terminal or command prompt at this location and enter the command python manage.py runserver. This will launch the Django server and make your application accessible.

Next, open another terminal window and navigate to the frontend folder of your project. Once inside the frontend directory, run the command npm start to start the frontend development server. This will launch your React application.

Upon launching the application, you will see a Google login button on the screen. Clicking this button will initiate the login process. If the user is new and doesn’t have an account, their information will be sent to the backend where it will be saved. The backend will assign a default password (which should be stored securely in an environment file) to the new user.

After the user is successfully authenticated, the backend will generate an authentication token. This token is automatically received by the frontend, allowing it to authorize subsequent requests to the backend API.

Once logged in, the user’s name, email address, and profile picture will be displayed in the navigation bar. Additionally, a short content section will be visible on the page below the navigation bar.

By following these steps, you will have your Django-React application up and running with Google login functionality. Users can log in, have their information saved securely in the backend, and see their profile details displayed in the navbar. The frontend will automatically handle the authentication process using the token received from the backend.

Remember to keep the default password secure and consider using an environment file to store sensitive information.

**Conclusion**

Throughout the blog, we covered various aspects, including the installation of required dependencies, the creation of Django views for user registration and authentication, and the implementation of React components for the login page and user profile display.

By following the step-by-step explanations, I hope readers were able to understand the code and the underlying concepts involved in the Django-React Google Sign-In integration. We discussed the importance of token authentication, the use of Axios for making HTTP requests, and the role of various libraries and modules.

With the completed application, users can log in using their Google accounts, have their profile information securely stored in the backend, and enjoy a personalized experience on the frontend.

**Link to the project in my github** : [arjunprakash027/React-Django-chatapp: A React Django Chat application with google signin (github.com)](https://github.com/arjunprakash027/React-Django-chatapp)

**Consider following me:**

[Arjun P | LinkedIn](https://www.linkedin.com/in/arjunprakash027/)

[arjunprakash027 (Arjun P) (github.com)](https://github.com/arjunprakash027)

[Arjun Prakash (@ArjunPrakash12) / Twitter](https://twitter.com/ArjunPrakash12)

**Check out my website**: [Arjun’s Portfolio (google.com)](https://sites.google.com/view/arjunresume/home)