# Credit and Reference: -

# Link: <https://blog.prototypr.io/how-to-build-google-login-into-a-react-app-and-node-express-api-821d049ee670>

**The code is the implementation of above article and full credit is given to writer of this article. I have implemented this code under guidance of this article**

# How to build Google login into a React app and Node/Express API

[Jaye Hackett](https://dinosaurenby.medium.com/?source=post_page-----821d049ee670--------------------------------)

[Jaye Hackett](https://dinosaurenby.medium.com/?source=post_page-----821d049ee670--------------------------------)

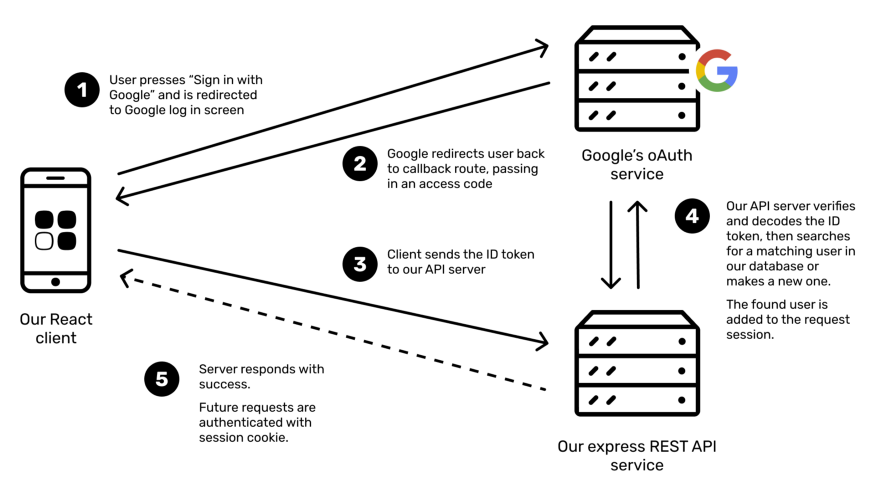
[Dec 14, 2020·6 min read](https://blog.prototypr.io/how-to-build-google-login-into-a-react-app-and-node-express-api-821d049ee670?source=post_page-----821d049ee670--------------------------------)

Let’s make an React app and API that lets us log in with Google.

We’ll show a “Log in with Google” button on our homepage, and rely on Google‘s servers to tell us some facts about the user (like their email address), which we’ll then store and use to authenticate the user in future.

This flow is triangular: we have our client React app, our own API server, and Google’s oAuth servers. Doing it right means making API requests between all three.





The three-part flow we’re creating.

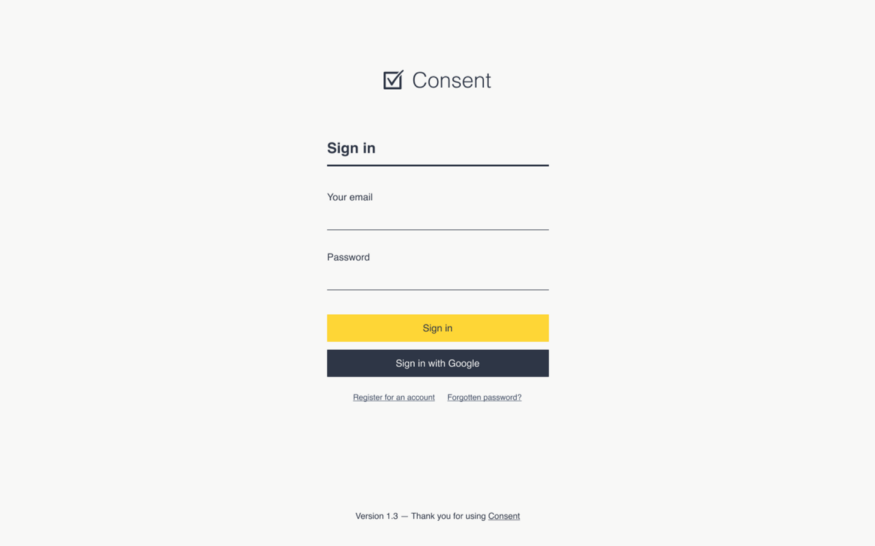
Why do this? Well, at minimum we can save the user from having to create and remember another password. We can also re-use other information from Google, like their profile picture and full name. Longer term, we could integrate other Google services on the user’s behalf.

# **You’ll need**

* A React application that makes requests of your API ([create-react-app](https://create-react-app.dev/) is just fine!)
* An express.js API we want to authenticate against

At the moment, our app only has basic username and password login.





The completed sign in screen for our example app. The “Sign in with Google” button is the important part.

# **1. Set up a Google Cloud project**

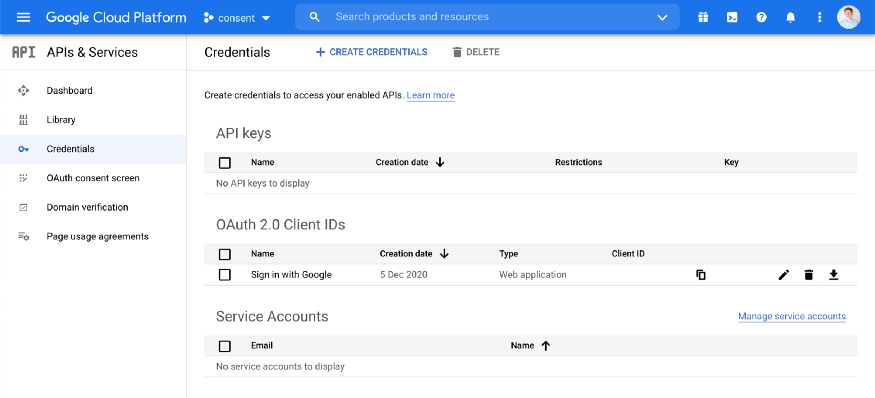
First, you’ll need to [create a Google Cloud project](https://console.cloud.google.com/).

You’ll need to configure your **OAuth consent screen**. Choose**external**. Google will then ask for the app’s name and logo, plus some developer contact details.

It will then ask what **scopes** we want to request — special permissions to read data and take action on behalf of our users. For our example we don’t need any of these.

Lastly, we’ll add our own Google account as a test user.





Once we’ve created our credentials, we should see something like this.

Now we can go to the **Credentials** screen and create a new **oAuth client ID**for a web application. Name it something like “Log in with Google”.

Google will ask us for authorised origins and a redirect URI. We just care about local testing for now, so we can use this value for both:

http://localhost:3000

Google will then give us a **client ID** and a client secret. We’ll only need the client ID for this example.

# **2. Add a login component to our React app**

First, make the client ID from earlier [available in a .env file](https://create-react-app.dev/docs/adding-custom-environment-variables/).

We’ll use the popular [**react-google-login**](https://www.npmjs.com/package/react-google-login) package to display a “Log in with Google” button that will handle displaying the Google login prompt and retrieving information about the user.

Install it with:

npm i react-google-login

We can render the component in our app like:

<GoogleLogin  
 clientId={process.env.REACT\_APP\_GOOGLE\_CLIENT\_ID}  
 buttonText="Log in with Google"  
 onSuccess={handleLogin}  
 onFailure={handleLogin}  
 cookiePolicy={'single\_host\_origin'}  
/>

When we click the button, it will trigger a pop-up where we can choose the Google account we want to log in with.

Then, the component will trigger the handleLogin function, passing in a Google user profile object, which contains an encrypted **ID token**.

We want to use this information to log a user into our own back-end, so the next step is to send theID token to our own API:

const handleLogin = async googleData => { const res = await fetch("/api/v1/auth/google", {  
 method: "POST",  
 body: JSON.stringify({  
 token: googleData.tokenId  
 }),  
 headers: {  
 "Content-Type": "application/json"  
 }  
 }) const data = await res.json()  
 // store returned user somehow  
}

Next, we need to write an API route and controller to handle this request and complete the cycle.

# **3. Authenticate the user against our own API**

We need to verify the Google-provided ID token is valid, and if so, either update or create a matching user in our database.

We’ll use Google’s official [node.js auth library](https://www.npmjs.com/package/google-auth-library) to verify and decode the ID token. This is a vital step — accepting a plain user ID or email would provide no security at all.

npm i google-auth-library

We’ll also need to provide in the same CLIENT\_ID from earlier as [environment config](https://www.npmjs.com/package/dotenv).

In our API, we could write some Express route like:

const { OAuth2Client } = require('google-auth-library')  
const client = new OAuth2Client(process.env.CLIENT\_ID)server.post("/api/v1/auth/google", async (req, res) => { const { token } = req.body const ticket = await client.verifyIdToken({  
 idToken: token,  
 audience: process.env.CLIENT\_ID  
 });  
 const { name, email, picture } = ticket.getPayload(); const user = await db.user.upsert({   
 where: { email: email },  
 update: { name, picture },  
 create: { name, email, picture }  
 }) res.status(201)  
 res.json(user)  
})

This handles POST requests to /api/v1/auth/google , verifying and decoding the token, pulling out the three pieces of information we want to store, performs an [upsert](https://www.prisma.io/docs/concepts/components/prisma-client/crud#upsert) operation on our database, and returns the retrieved user as JSON.

Here, we’re using [Prisma](https://www.prisma.io/) to speak to our database, but Mongoose, Sequelize, Knex or anything else will look similar.

Doing an upsert means we can keep our local database up to date if the user changes their picture or name on Google.

# **4. Manage the user’s session**

We’re nearly there.

We’ve verified the user and stored some information about them, but we can’t expect them to sign in every time they call our API, so we need to remember the currently logged in user.

We’re going to do this with plain old sessions and cookies.

We could use a JSON web token (JWT) instead, but since this is an API that will only be consumed by our own app, and won’t be operating on a massive scale, we don’t need to bother with the extra overhead of JWTs.

npm i express-session cookie-parser

Once you’ve set up sessions, we need to make a small modification to the /api/v1/auth/google POST handler from earlier:

...const user = await db.user.upsert({   
 where: { email: email },  
 update: { name, picture },  
 create: { name, email, picture }  
 })**req.session.userId = user.id**res.status(201)  
res.json(user)...

Now, anywhere else in our API we can check the currently signed in user by accessing req.session.userId, and send an unauthorised response if it isn’t set.

Here’s some middleware that does that:

server.use(async (req, res, next) => {  
 const user = await db.user.findFirst({where: { id: req.session.userId }})  
 req.user = user  
 next()  
})

Making a call to our database ensures that we’re always getting the most recent information about the user — if their account is removed from our database, their access is immediately revoked.

Once we have that middleware in place, asign out route is very simple to create:

server.delete("/api/v1/auth/logout", async (req, res) => {  
 **await req.session.destroy()**  
 res.status(200)  
 res.json({  
 message: "Logged out successfully"  
 })  
})

As is a route to simply return the currently logged in user:

server.get("/me", async (req, res) => {  
 res.status(200)  
 res.json(req.user)  
})