How to Build the jobCent Application Using Our Developer Tools

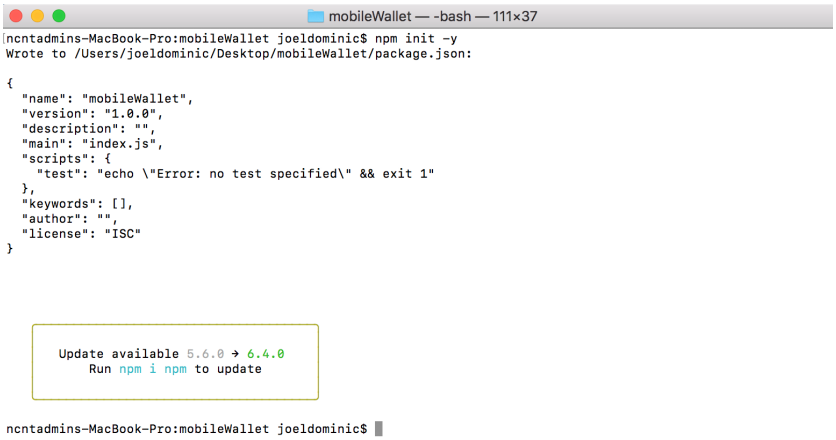
This walkthrough will guide how to build your own jobCent Application.

You can use this to recreate our app, or modify these steps as needed to accommodate your own application ideas.

Step 1 - Install and Setup Necessary Packages

To build this project we need to install the following tools and libraries.

1. **PostgreSQL:**
    a. Use this resource to gain access to the PSQL shell terminal
    b. Set defaults as follows: port=5432, password="dickey", user="postgres"
2. **NodeJS:**
    a. Download here.
    b. Run the following commands on a separate terminal window to make sure installation succeeded
        i. npm -v
        ii. node -v
3. **nCent Sandbox Repo:**
    a. In the terminal, run:
        i. git clone https://github.com/ncent/ncent.github.io.git
    b. Install the Sandbox dependencies by running the following commands:
        i. cd ncent.github.io
        ii. cd Sandbox/Sandbox\ API
        iii. npm install
4. **jobCent Dependencies:**
    a. Return to the root of the ncent.github.io directory
        i. cd ../..
    b. Go to the Applications folder
        i. cd /applications
    c. Create a new jobCentEmail folder:
        i. mkdir jobCentEmail
        ii. cd jobCentEmail
    d. Initialize client-level node dependencies and packages.json file
        i. npm init -y



e. Install the nCent SDK:
    i. npm i ncent-sdk-public

Additional Resources:

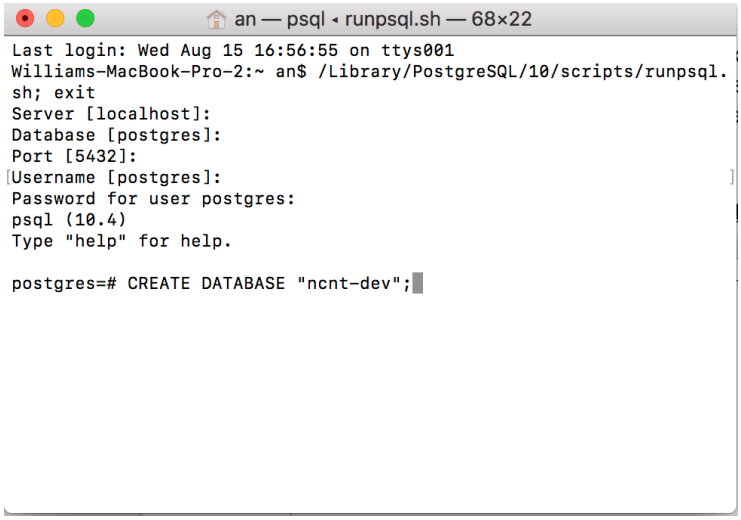
- nCent SDK installation tutorial
- Install PostgreSQL & Setup First Database Tutorial
- nCent Sandbox repository

Pro tip: npm install --save will automatically save anything you npm install to the dependency section of your package.json file

Step 2 - Setup our Sandbox Environment Database Locally

Test your application using a local instance of our Sandbox.

1. Open PSQL shell and login with the permissions you set up in step 2
2. List all the databases by the following command in the shell using the following command:
    a. \l
3. Create a database instance for the Sandbox with the command:
    a. CREATE DATABASE “ncnt-dev”;



4. Connect to your newly created database instance:
    a. \connect “ncnt-dev”
5. If you configured your permissions in step 2 differently, go to **Sandbox/Sandbox\ API/config/config.json**. Ensure that your information in the “development” object matches your setup in PostgreSQL

```
{
  "development": {
    "username": "postgres",
    "password": "dickey",
    "database": "ncnt-dev",
    "host": "127.0.0.1",
    "port": 5432,
    "dialect": "postgres"
  },
  "test": {
    "username": "postgres",
    "password": "dickey",
    "database": "ncnt-test",
    "host": "127.0.0.1",
    "port": 5432,
    "dialect": "postgres"
  }
}
```

6. Now, let’s migrate the schema into the database. In your current terminal, make your current directory **server** under **Sandbox\ API** and run:
    a. node_modules/.bin/sequelize db:migrate
7. You should see the following in your terminal:

```
admins-MacBook-Pro:server admin$ ../node_modules/.bin/sequelize db:migrate

Sequelize CLI [Node: 8.11.3, CLI: 4.0.0, ORM: 4.37.10]

Loaded configuration file "config/config.json".
Using environment "development".
sequelize deprecated String based operators are now deprecated. Please use Symbol
 based operators for better security, read more at http://docs.sequelizejs.com/ma
nual/tutorial/querying.html#operators ../node_modules/sequelize/lib/sequelize.js:
242:13
== 20180621230529-create-token-type: migrating =======
== 20180621230529-create-token-type: migrated (0.027s)

== 20180622161155-create-transaction: migrating =======
== 20180622161155-create-transaction: migrated (0.018s)

== 20180622225632-create-wallet: migrating =======
== 20180622225632-create-wallet: migrated (0.013s)

admins-MacBook-Pro:server admin$
```

8. Then, go back to the PSQL shell and check that you have the right tables:
    a. \dt
9. Your terminal should have printed the following:

```
ncnt-dev=# \dt
              List of relations
 Schema |     Name      | Type  |  Owner
--------+---------------+-------+----------
 public | SequelizeMeta | table | postgres
 public | TokenTypes    | table | postgres
 public | Transactions  | table | postgres
 public | Wallets       | table | postgres
(4 rows)
```

10. Run the following command to see that the table is empty:
    a. **select * from “table_name”;**

```
ncnt-dev=# SELECT * FROM "Wallets";
 uuid | wallet_uuid | tokentype_uuid | balance | createdAt | updatedAt
------+-------------+----------------+---------+-----------+-----------
(0 rows)
```

Additional Resources:

- **nCent Tutorial video** (coming soon)
- PostgreSQL documentation
- PostgreSQL video tutorial

PRO-TIP: Do not use **createdb** command in the terminal. Create the database only in PSQL shell with the **CREATE DATABASE “database_name”** command. Using **createdb** will make duplicate databases that are disjointed.

Step 3 - Run the Sandbox and Test the Environment

1. Enter the following command from your Sandbox terminal
    a. npm run start:dev {path to your Sandbox}
2. You should see the following:

```
> ncnt_api@1.0.0 start:dev /Users/admin/Documents/ncnt/Sandbox/Sandbox API
> nodemon ./bin/www

[nodemon] 1.17.5
[nodemon] to restart at any time, enter `rs`
[nodemon] watching: *.*
[nodemon] starting `node ./bin/www`
sequelize deprecated String based operators are now deprecated. Please use Symbol based operators for better security, read more at http://docs.sequelizejs.com/manual/tutorial/querying.html#operators node_modules/sequelize/lib/sequelize.js:242:13
Executing (default): CREATE TABLE IF NOT EXISTS "TokenTypes" ("Name" VARCHAR(255) NOT NULL UNIQUE, "uuid" UUID NOT NULL , "ExpiryDate" TIMESTAMP WITH TIME ZONE NOT NULL, "sponsor_uuid" VARCHAR(255) NOT NULL, "totalTokens" INTEGER NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): CREATE TABLE IF NOT EXISTS "TokenTypes" ("Name" VARCHAR(255) NOT NULL UNIQUE, "uuid" UUID NOT NULL , "ExpiryDate" TIMESTAMP WITH TIME ZONE NOT NULL, "sponsor_uuid" VARCHAR(255) NOT NULL, "totalTokens" INTEGER NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'TokenTypes' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'TokenTypes' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): CREATE TABLE IF NOT EXISTS "Transactions" ("uuid" UUID NOT NULL , "amount" INTEGER NOT NULL, "fromAddress" VARCHAR(255) NOT NULL, "toAddress" VARCHAR(255) NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, "tokentype_uuid" UUID REFERENCES "TokenTypes" ("uuid") ON DELETE SET NULL ON UPDATE CASCADE, PRIMARY KEY ("uuid"));
Executing (default): CREATE TABLE IF NOT EXISTS "Transactions" ("uuid" UUID NOT NULL , "amount" INTEGER NOT NULL, "fromAddress" VARCHAR(255) NOT NULL, "toAddress" VARCHAR(255) NOT NULL, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, "tokentype_uuid" UUID REFERENCES "TokenTypes" ("uuid") ON DELETE SET NULL ON UPDATE CASCADE, PRIMARY KEY ("uuid"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Transactions' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Transactions' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): CREATE TABLE IF NOT EXISTS "Wallets" ("uuid" UUID NOT NULL , "wallet_uuid" VARCHAR(255) NOT NULL, "tokentype_uuid" UUID NOT NULL DEFAULT '498cc1fe-62d3-4863-a0e0-a42049b90fff', "balance" INTEGER NOT NULL DEFAULT 0, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): CREATE TABLE IF NOT EXISTS "Wallets" ("uuid" UUID NOT NULL , "wallet_uuid" VARCHAR(255) NOT NULL, "tokentype_uuid" UUID NOT NULL DEFAULT '498cc1fe-62d3-4863-a0e0-a42049b90fff', "balance" INTEGER NOT NULL DEFAULT 0, "createdAt" TIMESTAMP WITH TIME ZONE NOT NULL, "updatedAt" TIMESTAMP WITH TIME ZONE NOT NULL, PRIMARY KEY ("uuid"));
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Wallets' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): SELECT i.relname AS name, ix.indisprimary AS primary, ix.indisunique AS unique, ix.indkey AS indkey, array_agg(a.attnum) as column_indexes, array_agg(a.attname) AS column_names, pg_get_indexdef(ix.indexrelid) AS definition FROM pg_class t, pg_class i, pg_index ix, pg_attribute a WHERE t.oid = ix.indrelid AND i.oid = ix.indexrelid AND a.attrelid = t.oid AND t.relkind = 'r' and t.relname = 'Wallets' GROUP BY i.relname, ix.indexrelid, ix.indisprimary, ix.indisunique, ix.indkey ORDER BY i.relname;
Executing (default): CREATE UNIQUE INDEX "wallets_wallet_uuid_tokentype_uuid" ON "Wallets" ("wallet_uuid", "tokentype_uuid")
Executing (default): CREATE UNIQUE INDEX "wallets_wallet_uuid_tokentype_uuid" ON "Wallets" ("wallet_uuid", "tokentype_uuid")
```

3. If any errors appear enter the following command:
    a. rs
4. Now, open another terminal and go to the SDK directory in your local git repository.
5. Use the following command to test the tables in the database:
    a. node test.js
6. You should see the following:

```
ncnt-dev=# SELECT * FROM "Wallets";
                uuid                |                wallet_uuid                |           tokentype_uuid             | balance |        createdAt         |        updatedAt
------------------------------------+-------------------------------------------+--------------------------------------+---------+--------------------------+--------------------------
 0d475e8f-1e78-4b1e-90fc-e067f56ca003 | GBKEY62GRQ4CTEZ4XDUKMXDBHHF34CZE4UVFFRHW4FJCUERXIISNTOFL | dd8f431e-9641-4e57-bfb0-59cc6e65b86d |  999990 | 2018-08-15 17:55:21.093-07 | 2018-08-15 17:55:21.17-07
 9659a087-c752-4626-a1c7-f55adad9ce85 | GCZFSFCAISMRO236EVOXSKU56726OCRARY54KAB6NBJNWW6FIQZKMSVF | dd8f431e-9641-4e57-bfb0-59cc6e65b86d |      10 | 2018-08-15 17:55:21.179-07 | 2018-08-15 17:55:21.182-07
(2 rows)
```

7. Now, we will clear the database. In your terminal with the sandbox is running, enter the following commands:
    a. ctrl-c
    b. node_modules/.bin/sequelize db:migrate:undo:all
    c. node_modules/.bin/sequelize db:migrate
8. Now, your tables will be empty. Check the tables on your shell to see that it is clear.

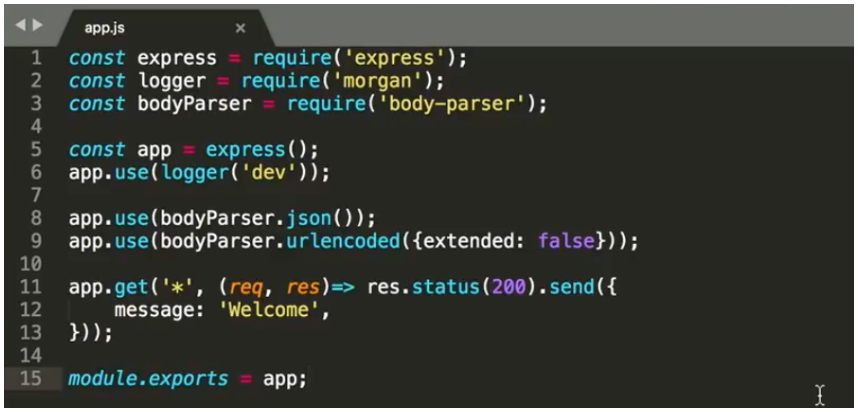
Resources:

- nCent Video Tutorial on testing the SDK
- Sequalize documentation
- Sequalize tutorial video

PROTIP: Often, when you are testing your backend, it can help to clear your database for easier visibility into how your data is being handled. In order to do this, use the command **DROP DATABASE <table_name>;**

Step 4 - Set up the jobCent Database

Like the blockchain technology it mirrors, our Sandbox environment and SDK only manage wallet address and transaction information, in addition to token stamping. At the application level, like with jobCent, we need a friendly way to link wallet address with user accounts. Therefore, we need a jobCent database to store user information such that a user could log in with an email and password and be linked to wallet credentials.



We used a Node.js environment for development of jobCent. Make sure to install Node.JS and npm.

1. **Follow our tutorial for setting up a server**
    a. Make the directory for your project. Include a bin and a server folder
    b. Initialize your directory
        i. npm init -y
    c. If you haven't already, install express, body-parser, and morgan npm packages
        i. npm install express body-parser morgan
    d. Create an app.js file like the following:

```
const express = require('express');
const logger = require('morgan');
const bodyParser = require('body-parser');

// Set up the express app
const app = express();

// Log requests to the console.
app.use(logger('dev'));

// Parse incoming requests data (https://github.com/expressjs/body-parser)
app.use(bodyParser.json());
app.use(bodyParser.urlencoded({ extended: false }));

// Setup a default catch-all route that sends back a welcome message in JSON format.
app.get('*', (req, res) => res.status(200).send({
  message: 'Welcome to the beginning of nothingness.',
}));

module.exports = app;
```

e. Make a file called ‘www’ in the bin folder like the following:

```
// This will be our application entry. We'll setup our server here.
const http = require('http');
const app = require('../app'); // The express app we just created

const port = parseInt(process.env.PORT, 10) || 8000;
app.set('port', port);

const server = http.createServer(app);
server.listen(port);
```

f. Use nodemon so that your server restarts every time you change code
   i. npm i -D nodemon
g. Navigate to your package.json file, and under the scripts section add
   i. "start:dev": "nodemon ./bin/www"
h. Run the following command:
   i. npm run start:dev
i. Navigate to localhost:8000 to see your default message

2. **Ensure you have sequelize installed:**
   a. npm install --save sequelize pg pg-hstore
3. **Initialize sequelize:**
   a. sequelize init
4. **Create your database:**
   a. createdb ‘database_name’
5. **Create your models:**
   a. Execute the following command
      i. sequelize model:create --name ‘{model_name}’ --attributes title:string
   b. This is to create a model with a single string attribute that is the title. You can add more attributes later on in the file or list them in this command

c. Do this for each model you would like to create
d. This will also create your migrations

6. **Migrate your database:**
    a. sequelize migrate
7. **Create controllers:**
    a. Require the relevant models
    b. Develop the functionality of the controller by writing the methods you will need.
    c. See example below:

```
const Bug = require('../models').Bug;
const User = require('../models').User;
const bugUser = require('../models').bugUser;
const bcrypt = require('bcrypt');
const path = require('path');
const ncentSDK = require('../../../../../../SDK/source/');
const ncentSdkInstance = new ncentSDK();

module.exports = {
  getBalance(req, res){
    return User
      .findById(req.session.user.uuid, {
      })
      .then(user => {
        console.log('here');
        if (!user) {
          return res.status(404).send({
            message: 'User Not Found',
          });
        }
        return new Promise(function(resolve, reject) {
          return ncentSdkInstance.getTokenBalance(user.email,'9d91db6b-f33a-4392-a583-a6ea14bd368f',resolve);
        })
        .then(data => res.status(200).send(data))
        .catch(error=> console.log(error));

      })
      .catch(error => res.status(400).send(error));
  },
  updateBugPage(req, res){
    res.sendFile(path.resolve(__dirname + '/public/updatebug.html'));
  },
  logOut(req, res){
    if (req.session.user && req.cookies.user_sid) {
      res.clearCookie('user_sid');
    }
    res.sendFile(path.resolve('__dirname' + '../../../../index.html'));
  },
```

8. **Create an index.js file in controllers and export all your controllers**
    a. Navigate to the controllers
        i. cd controllers
    b. Create an index.js file
        i. touch index.js
    c. Add the following to your index.js file:

```
const controller_name = require('./controller_name');

module.exports = {
        controller_name
};
```

*Replace controller_name with the name of your controllers. Make sure you do this for all of your controllers.

9. **In the index.js file in the routes folder,**
    a. Require and define all the controllers

b. Define your routes and the methods they will use from the appropriate controllers.
c. As an example (where wallet is a model):

```
“”
const walletController = require('../controllers').wallet;
app.get(‘/getBalance’, walletController.getBalance);
“”
```

d. You must decide whether it will be a post (for creating new information), put (for updating information), or get (for retrieving information), then add the route as a string for the first parameter and the controller_name.functionName as the second parameter.

10. **Require the routes you just made in the app.js file with this code:**

```
require('./server/routes')(app);
```

Additional Resources:

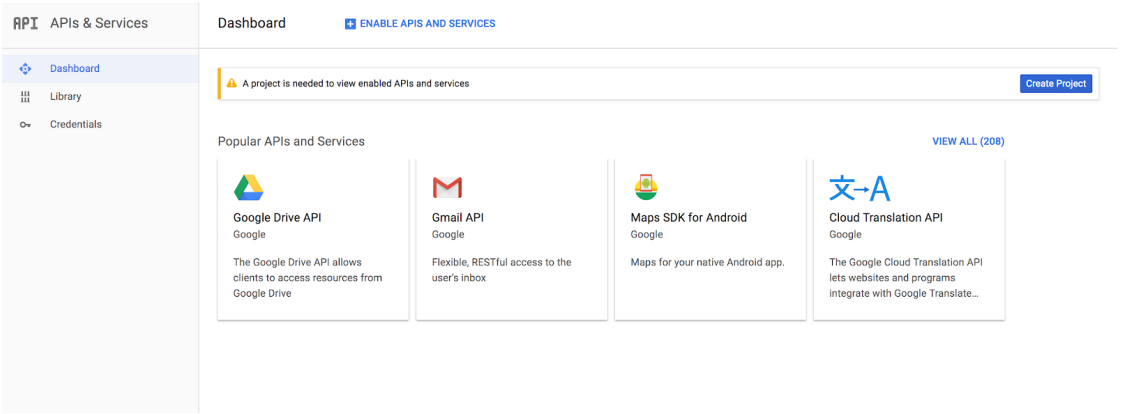
- nCent Labs tutorial on setting up your server
- Getting started with node express and postgres using sequelize
- Firebase tutorial video
- Official docs
- Official "getting started" guide
- Getting Started with Node.js

PRO-TIP: For speed, you can use a free server such as Firebase to get your app running quickly.

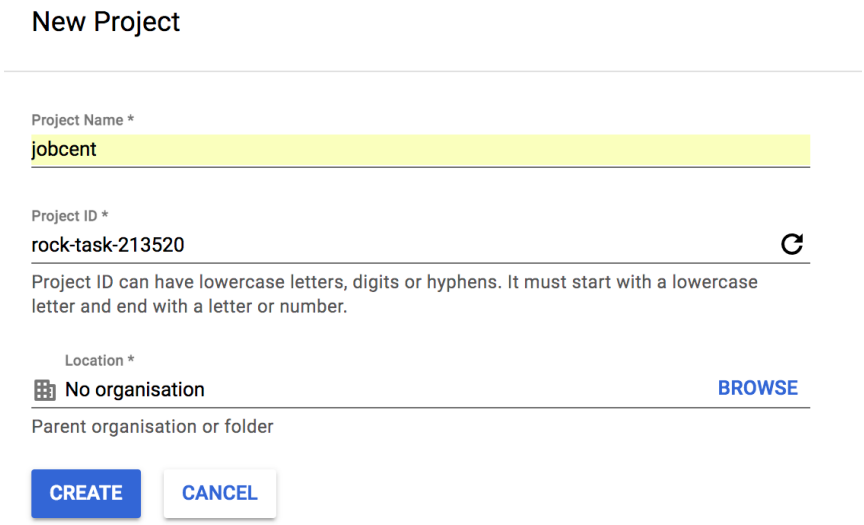
PRO-TIP 2: Read up on Package.json files to get a better understanding of node and how it works.

Step 5 - Create and Configure Google Oauth Client

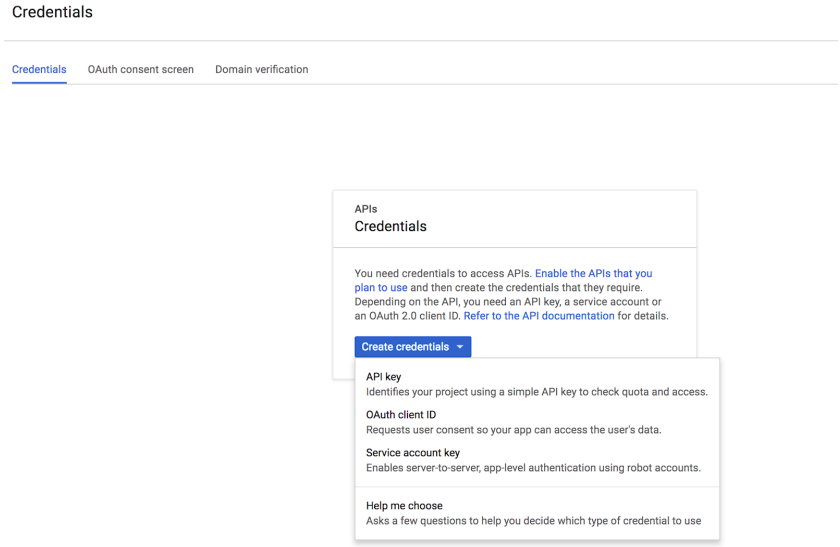
1. **Go to Google Cloud Console and sign in with your Google account. You should see this:**



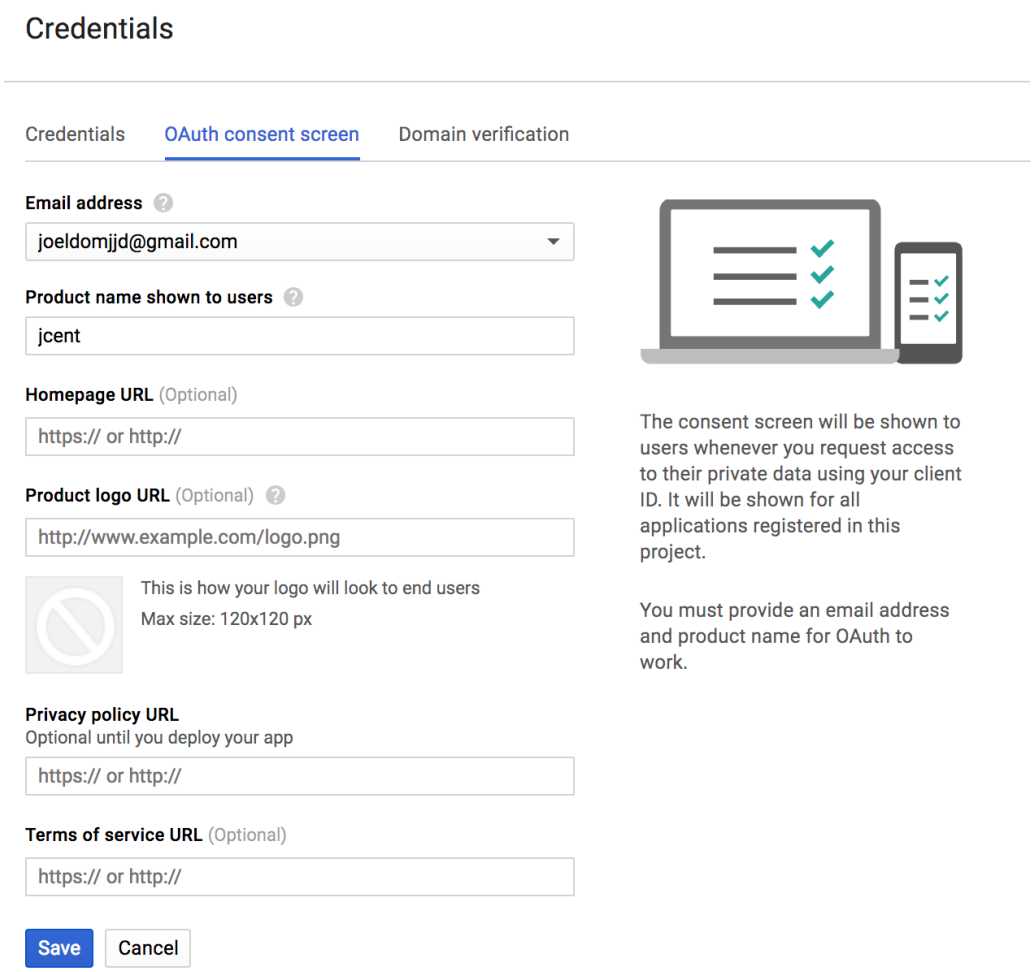
2. **Click on the “Create Project” button and name your project**



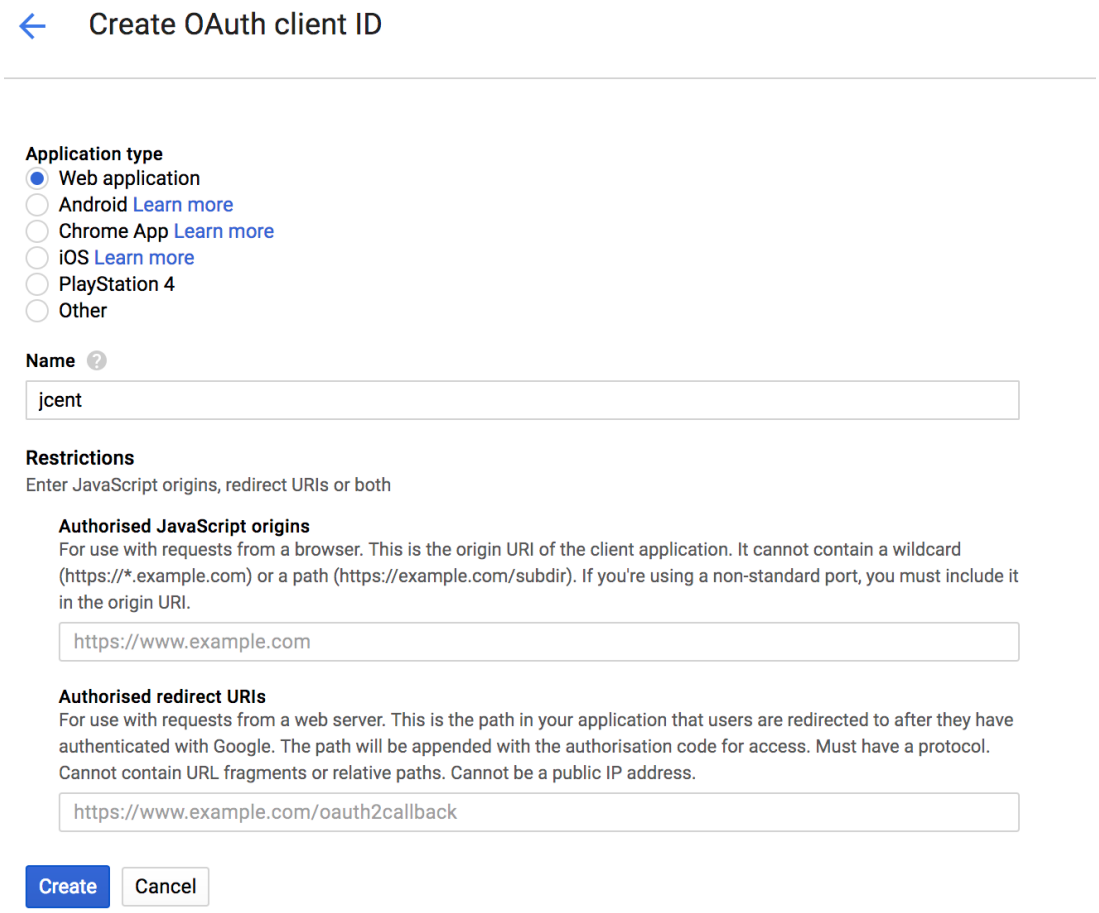
3. **On the window that appears, click “Create Credentials”, then “0Auth client ID” as shown below:**



4. **The following screen will appear, where you will need to fill in your email address and “Product name shown to users”. The rest is optional.**

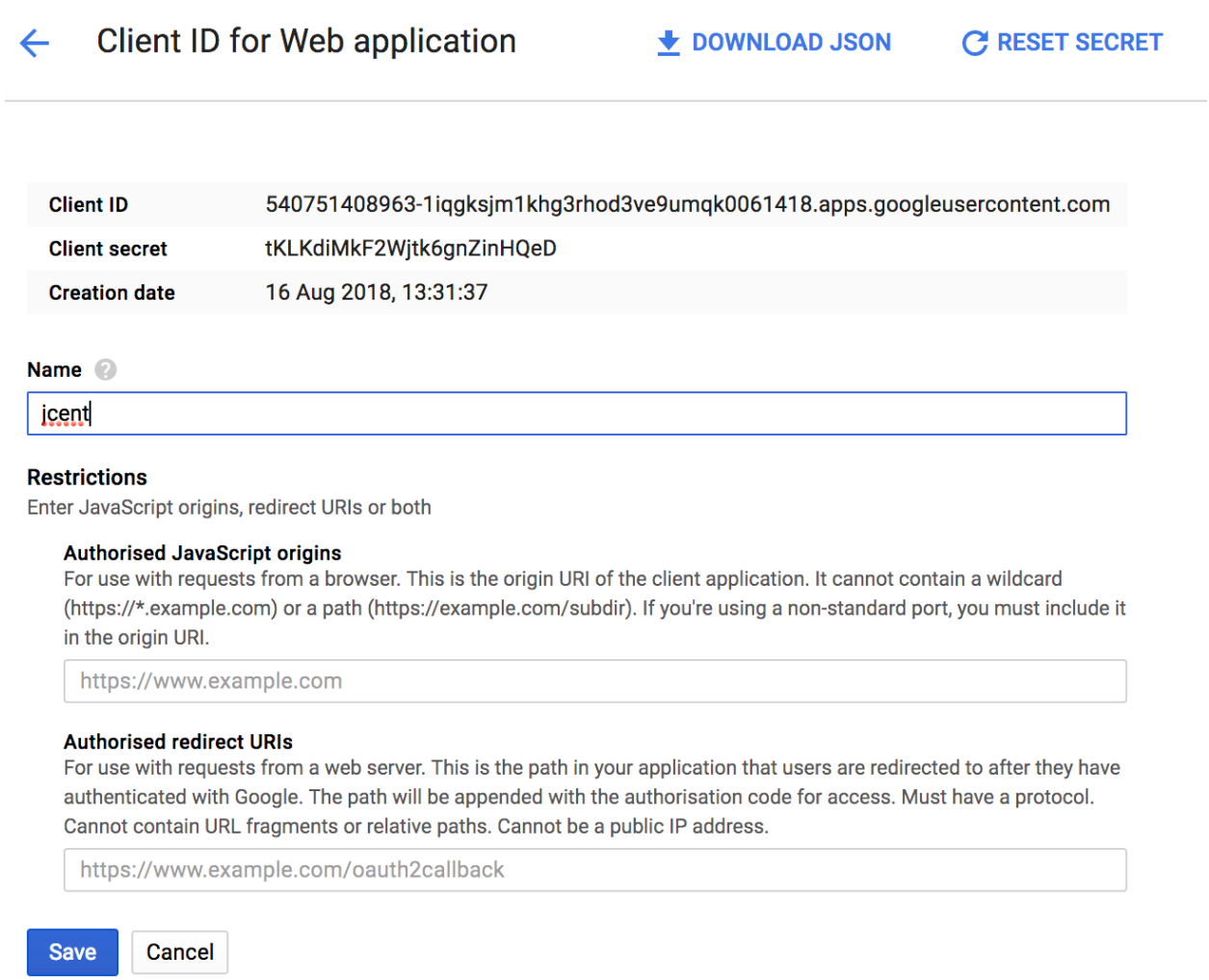


5. **Name your client and the application type.**



6. **Click “jcent” to see your client ID, and client secret. It should look like the below screenshot, and store this information in the jobCentEmail directory you created by**

clicking “Download JSON”



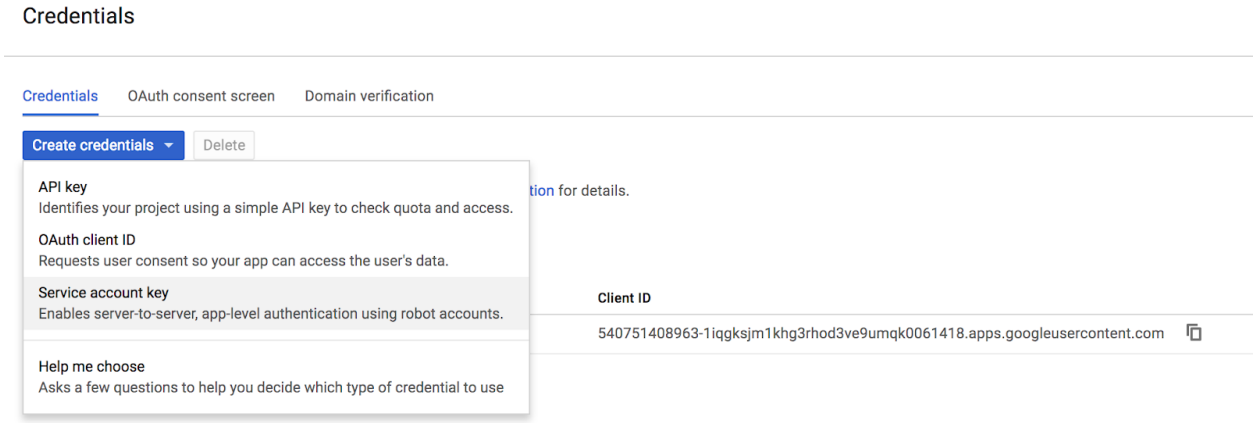
Additional Resources

- 0Auth2 docs
- How to create Google 0Auth2 client ID

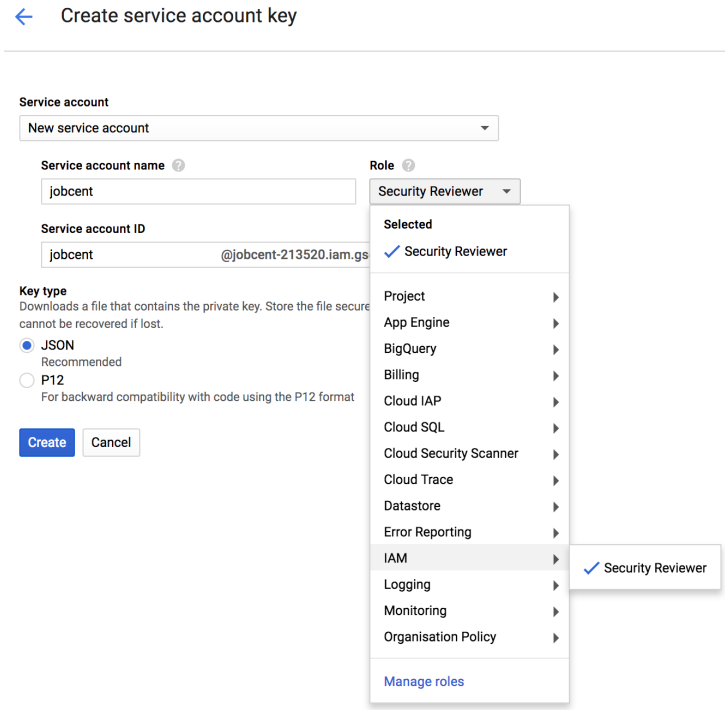
PRO-TIP: Never reveal your client secret to anyone. (We our client secret above so you can’t use ours :)

Step 6 - Setup Google Service Account for PubSub

1. **Go back to the console home page and click on create credentials again, but this time, select “Service Account Key”**



2. **Configure as follows and press create - you would now have a json file downloaded - store that in jobCentEmail.**
    a. Do not share this either - it contains the private key to your service account



Additional Resources

- Setting up Google Play API access
- Set up an API account for 0Auth2
- Creating and Using Service Accounts video

PRO-TIP: Read up on PubSub in this link: https://cloud.google.com/pubsub/docs/overview

Step 7 - Setup Primary jobCent Script

1. **Install and initialize Google Cloud SDK**
    a. Download the Google Cloud SDK
    b. Add cloud SDK tools to your path
        i. `'./google-cloud-sdk/install.sh'`
    c. Restart your terminal
    d. Run the following in your terminal
        i. gcloud init'
    e. Accept the option to log in using your Google user account
    f. Select the project (if you only have one, it will be chosen for you)
2. **Create a pubSub topic/subscription by running the following commands (replacing appropriate placeholders with desired names):**

```
gcloud pubsub topics create myTopic
gcloud pubsub subscriptions create --topic myTopic mySubscription
gcloud pubsub topics publish myTopic --message "hello"
gcloud pubsub subscriptions pull --auto-ack mySubscription
```

3. Now we are ready to create our index.js in the jobCentEmail directory and configure it as below:

```
///////////////things to npm install\\\\\\\\\\\\\\
const express = require('express');
const fetch = require("isomorphic-fetch");
const fs = require("fs");
const opn = require('opn');
const PubSub = require(`@google-cloud/pubsub`); //must use gcloud init / install, read google cloud SDK for more info
const gmailApiSync = require('gmail-api-sync');

const pubsub = new PubSub({
    keyFilename: './JobCent-850fe0fe5e43.json'
});
const subscriptionName = 'projects/PROJECT_ID/subscriptions/emailWatcher';
const subscription = pubsub.subscription(subscriptionName);

const gmailPort = 3001;
const app = express();

const scopes = [
                    'https://mail.google.com/',
                    'https://www.googleapis.com/auth/gmail.modify',
                    'https://www.googleapis.com/auth/gmail.readonly',
                    'https://www.googleapis.com/auth/gmail.send'
                ];
const {google} = require('googleapis');
const gmailClass = google.gmail('v1');
const oauth2Client = new google.auth.OAuth2(
    OAUTH CLIENT ID
    CLIENT_SECRET
    `http://localhost:3001/`
);
//const oauthUrl = oauth2Client.generateAuthUrl({access_type: 'offline', scope: scopes});

gmailApiSync.setClientSecretsFile('./client_secret.json');

const ncentSDK = require('../../SDK/source/');
const ncentSdkInstance = new ncentSDK();
```

4. For SERVICE_FILE, put the name of the JSON file you downloaded in Step 6 for the Google service account.
5. For PROJECT_ID, put the id of your project which you can find on the Google Developer console.
6. For OAUTH CLIENT ID and CLIENT_SECRET, put the client id and secret that you saw when you pressed jcent in Step 5 earlier.
7. Run the following for any modules required above
    a. npm install --save {module_name}

Additional Resources:

- Our index.js Implementation
- RequireJS Documentation
- Google Cloud SDK Quick Start - Mac
- Google Cloud SDK Quickstart - CLI

PRO-TIP: It is best to write your “requires” into **const** variables, so that they are not able to be mutated. This is a common best practice when writing Javascript code.

Step 8 - Initializing jobCent

1. First, create a function called “main” in your index.js file.
2. We will then initialize the application by stamping the jobCent token, followed by seeding the proper wallets with tokens.
3. For integration with gmail, we need to open the authentication url for user consent and login and respond to it with a callback function. We named ours “getHomePageCallback”.
4. Finally, we need to start listening on the gmail port.
5. When steps 1-4 are completed, your code should look something like the below screenshot, where “initJobCent” is an abstracted function for stamping the token and seeding the wallets (more detail below):

```
function main() {
    initJobCent();
    opn(oauthUrl);
    app.get('/', getHomePageCallback);
    app.listen(gmailPort, (err) => {
        if (err) {
            console.log(`failed to listen to ${gmailPort}`, err);
        }
    });
}
main();
```

6. We set up our “initJobCent” function - as seen below - by stamping the token and storing “token_id” as a global variable

```
function initJobCent() {
    return new Promise(function(resolve, reject) {
        return ncentSdkInstance.stampToken('jobcent@ncnt.io', 'jobCent', 1000000, '2021', resolve);
    })
    .then(function(response) {
        token_id = response.data["tokenTypeResponseData"]["uuid"];
    })
    .then(function() {
        return createAndTransfer('mb@ncnt.io', 2000);
    })
}
```

7. Remember to declare your “token_id” above the function closure so that it stores the ID globally across the file.
8. Once the auth url is opened for the user and the user acknowledges the request, we can request an access token with the code that we get from the response. We then initialize

the email watcher.

```
function getHomePageCallback (request, response) {
    getOauthTokens(request.query.code)
        .then(function(tokens) {
        setOauthCredentials(tokens);
        gmail = google.gmail({version: 'v1', oauth2Client});
        initEmailWatcher();
        response.send("Done with authentication.");
    }, function(reason){
        console.log("get auth tokens failed" + reason)
    });
}
```

9. A closer look at setting the credentials for the oauth2Client:

```
function getOauthTokens (tokenCode) {
    //console.log("get auth tokens");
    return oauth2Client.getToken(tokenCode);
}

function setOauthCredentials () {
    //console.log("set auth credentials");
    oauth2Client.credentials = tokens.tokens;
}
```

10. A closer look at setting up the gmail watcher. Ensure that under topicName, projects/PROJECT_ID is your own project id.

```
function initEmailWatcher() {
    let options = {
        userId: 'me',
        auth: oauth2Client,
        resource: {
            labelIds: ['INBOX'],
            topicName: "projects/jobcent-210021/topics/emailTransaction"
        }
    };
    gmail.users.watch(options, function (err, res) {
        if (err) {
            //console.log(err);
            return;
        }
    });
}
```

Additional Resources:

- Gmail API Reference
- Using OAuth 2.0 to Access Google APIs

Step 9 - Syncing Messages with Email History

1. After we get the Oauth credentials, we need to sync the existing inbox messages with the email history. Each time a new email is received, the ID of the email history gets updated, so we need to make sure that this record is always up to date. You can see our

invocation of our syncMessages function in the promise chaining below.

```
function getHomePageCallback (request, response) {
    const fullSyncOptions = {query: 'from: ncnt.io'};
    getOauthTokens(request.query.code)
        .then(function(tokens) {
        setOauthCredentials(tokens);
        gmail = google.gmail({version: 'v1', oauth2Client});
        initEmailWatcher();
        syncMessages(true, fullSyncOptions, resolve, reject);
        response.send("Done with authentication.");
    }, function(reason){
        console.log("get auth tokens failed" + reason)
    });
}
```

2. Below is our implementation of the syncMessages function. The first parameter, “full”, determines whether or not we do a full sync or partial sync via the gmail API. First, we use our gmail-sync-api NPM package to authorize with our authentication token, then we either do a full sync for first-time setup, or partial for subsequent syncs.

```
function syncMessages(full, syncOptions, resolve, reject) {
    gmailApiSync.authorizeWithToken(tkn, function (err, oauth) {
        if (err) {
            return reject(err);
        }
        else {
            if (full) {
                gmailApiSync.queryMessages(oauth, syncOptions, function (err, response) {
                    if (err) {
                        return reject(err);
                    }
                    return resolve(response);
                    //console.log(response);
                });
            } else {
                gmailApiSync.syncMessages(oauth, syncOptions, function (err, response) {
                    if (err) {
                        return reject(err);
                    }
                    //console.log(response.emails);
                    return resolve(response);
                })
            }
        }
    });
}
```

Additional Resources:

- Gmail API Sync npm package
- Gmail API Guide for Sync

Step 10 - Inbound Message Handling

1. Now that our watcher is synced up with the inbox itself, we can handle each new message that comes in. In order to set this up, we have to first set up a subscription handler that triggers whenever a message is received.

```
function getHomePageCallback (request, response) {
    const fullSyncOptions = {query: 'from: ncnt.io'};
    getOauthTokens(request.query.code)
        .then(function(tokens) {
        setOauthCredentials(tokens);
        gmail = google.gmail({version: 'v1', oauth2Client});
        initEmailWatcher();
        new Promise(function(resolve, reject) {
            syncMessages(true, fullSyncOptions, resolve, reject);
        })
        .then(function(response) {
            console.log(response);
            startHistoryId = response.historyId;
            console.log(startHistoryId);
            subscription.on(`message`, messageHandler);
        })
        .catch(function (error) {
            console.log(error);
        })
        response.send("Done with authentication.");
    }, function(reason){
        console.log("get auth tokens failed" + reason)
    });
}
```

2. Setup the subscriber in a Promise chain to ensure that the inbox sync process has completed, so you can store the latest historyId before handling the messages.
3. For any message that isn't associated with jobcent, we acknowledge the message without taking any further action. We also ignore any message ID that we have already received. Sometimes an existing message will trigger the subscriber upon the initial setup process, creating the need for this logic.

4. Store the current history id as the first new message that we received and do a partial synchronisation to deal with any influx of messages and deal with each of them.
5. Mark each message as already processed after we have dealt with it.

```
function messageHandler(message) {
    let messageJSON = JSON.parse(message.data);
    //console.log("in message handler");
    if(messageJSON.emailAddress !== 'jobcent@ncnt.io') {
        message.ack();
        return;
     }
    if ((message.id in alreadyProcessed)) return;
    printMessage(message);
    currHistoryId = messageJSON.historyId;
    if (startHistoryId === undefined || currHistoryId < startHistoryId) {
        message.ack();
        return;
    }
    console.log("startHistoryId: " + startHistoryId + ", currHistoryId: " + currHistoryId);
    const syncOptions = {historyId: startHistoryId};
    new Promise(function(resolve, reject) {
        return syncMessages(false, syncOptions, resolve, reject);
    })
    .then(function(response){
        if (response.emails.length !== 0) {
            for (i = 0; i < response.emails.length; i++) {
                const msgOptions = {'userId': messageJSON.emailAddress, 'auth': oauth2Client, 'id': response.emails[i].id};
                alreadyProcessed[response.emails[i].id] = 1;
                dealNewMessage(msgOptions, message);
            }
        } else {
            console.log("array is empty again");
        }
    })
    .catch(function(error) {
        console.log(error);
        return;
    })
    startHistoryId = currHistoryId;
};
```

6. The function in the screenshot below contains the logic for “dealNewMessage”, which determines whether or not jobCents are being exchanged.
    i. We first get all the relevant messages from the gmail API based on the message options.
    ii. Then, we go through the headers in the email to check if it is in fact cc’d to jobcent@ncnt.io or sent directly to jobcent@ncnt.io.
        1. If jobCent is cc’d, we will attempt to send one jobCent from the sender to the receiver.
        2. If jobCent is the receiver directly, we are running a promotion such that if the user has never had a jobCent, they will be given one for free.
    iii. We also check if there are multiple addresses being sent to (that is multiTo) and if they are, then we send an error email below.

```
            console.log("out of loop" + ", FromEmail: " + fromEmail + ", ToEmail: " + toEmail +
            ", ccFound: " + ccFound + ", toJbCent: " + toJbCent);
            if(((ccFound !== 1) && !toJbCent) || toEmail === fromEmail) {
                return;
            }
            if(multiTo) {
                console.log("Too many addresses by " + fromEmail + " to " + toEmail);
                sendEmail(fromEmail, './manyAddressesnew.html', "Error: You've entered too many addresses in the To line");
                return;
            }
            console.log('\nSending one jobCent from ' + fromEmail + ' to ' + toEmail);
            processTransaction(toEmail, fromEmail);
        })
        .catch(function(error){
            console.log(error.message);
        })
    })
    .then(function() {
        console.log("message acknowledged");
        message.ack();
        return;
    })
    .catch(function(error){
        console.log(error.message);
        return;
    });
}
```

```
function dealNewMessage(msgOptions, message) {
    let toEmail = '';
    let fromEmail = '';
    new Promise (function(resolve, reject) {
        gmail.users.messages.get(msgOptions)
        .then(function(response){
            console.log("getresponse: " + response.data.id);
            let ccFound = -1;
            let multiTo = false;
            let toJbCent = false;
            let headers = response.data.payload.headers;
            for(idx in headers) {
                if (headers[idx].name === 'To') {
                    console.log("full headers: " + headers[idx].value);
                    console.log("first header: " + headers[idx].value.substring(1, headers[idx].value.indexOf('<') - 1));
                    if (!((((headers[idx].value.match(/@/g) || []).length === 2)
                    && (headers[idx].value.match(/</g)) && (headers[idx].value.substring(1, headers[idx].value.indexOf('<') - 1)
                    === getEmailString(headers[idx]))) || ((headers[idx].value.match(/@/g) || []).length === 1))) multiTo = true;
                    toEmail = getEmailString(headers[idx]);
                    let stIdx = headers[idx].value.indexOf('jobcent@ncnt.io');
                    if (stIdx !== -1) toJbCent = true;
                }
                if (headers[idx].name === "From") {
                    fromEmail = getEmailString(headers[idx]);
                }
                if (headers[idx].name === "Cc") {
                    let startIdx = headers[idx].value.indexOf('jobcent@ncnt.io');
                    if (startIdx !== -1) {
                        ccFound = 1;
                    } else {
                        ccFound = 0;
                    }
                }
                if (toEmail !== '' && fromEmail !== '' && ccFound !== -1) break;
            }
```

Additional Resources:

- [Google PubSub Event Subscriber Documentation]
- [Javascript Regular Expressions Guide]

PRO-TIP: Note that we used a regex expression (/@/g) in the match above, read more about regex expressions in the link above - they are very powerful and useful (more than Ctrl-F :))

Step 11 - Transaction Processing

PLACEHOLDER FOR AFTER THE JOBCENT REFACTOR TO ACCOMMODATE NEW SDK CHANGES

Step 12 - Sending jobCents!

a. We're almost ready to send emails! First, develop the html templates for the emails sent by the jobCent server. You can view an example here.

b. Finally, write the function for sending the email. In our function in the screenshot below, you can see we pass in the receiver of the email and the HTML file template to be sent, since this will vary depending on the success of the transaction itself. We use the file reader in node to append the appropriate email headers to the HTML.

```
const sendEmail = async (receiver, file, subject) => {
    fs.readFile(file, (err,data) => {
      let email_lines = [];

      email_lines.push('From: "nCnt Hiring" <jobcent@ncnt.io>');
      email_lines.push('To: ' + receiver);
      email_lines.push('Content-type: text/html;charset=utf-8');
      email_lines.push('MIME-Version: 1.0');
      email_lines.push('Subject: ' + subject);
      email_lines.push('');

      email_lines.push(data);

      let email = email_lines.join('\r\n').trim();

      let base64EncodedEmail = new Buffer(email).toString('base64');
      base64EncodedEmail = base64EncodedEmail.replace(/\+/g, '-').replace(/\//g, '_');

      gmailClass.users.messages.send({
            auth: oauth2Client,
            userId: 'me',
            resource: {
                raw: base64EncodedEmail
            }
      });

    });
}
```

c. Feel free to get creative with your email templates! Here is an example of ours:

Congratulations! You've received a jobCent.

Get paid to help nCent find our all-stars:

- Apply for a job at nCent Labs
- Get an extra $10,000 if you get hired
- If you're not looking for a job, send it to a friend:
  1. Emailing them and cc'ing jobcent@ncnt.io
  2. If they get hired, you get $5,000 and they get $10,000
  3. If they send to a friend that gets hired, you get $2,500, they get $5,000, and their friend gets $10,000

Come help us build the future in Redwood City!

Apply Now

Best,
nCent Labs

Connect with us.

Phone: 650.503.8785

Email: kk@ncnt.io

To find out more about jobCent, please navigate to jobCent.io

Additional Resources:

Node.js readFile Documentation

Sending Emails Through the Gmail API