RUNNING SALESFORCE APPLICATIONS ON ELECTRON

A detailed guide to running salesforce applications on electron

Authors: Vitalii Siukaiev

Categories: Salesforce

Keywords: SFDX, DevOps

Author Bio: Vitalii Siukaiev is a certified senior Salesforce developer at SoftServe. Vitalii has five years of experience with Salesforce and eight years of experience in web development. He is passionate about technical innovations and further developing his programming skills.

softserve

INTRODUCTION

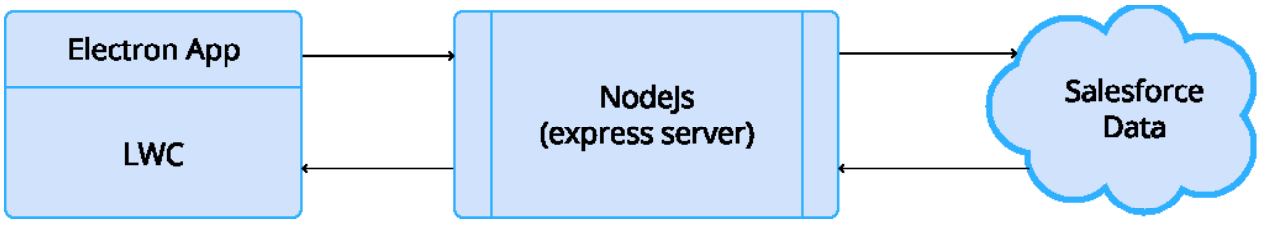
Since it's possible to build standalone web applications based on web components, it's a good idea to use this to your advantage.

One framework that works well with Salesforce applications is Electron. This framework was developed and is maintained by GitHub. Electron combines the Chromium rendering engine and the Node.js runtime. It allows users to create desktop applications using just HTML, CSS, and JavaScript.

Running Salesforce applications on Electron is beneficial for several reasons. Because Electron uses Node.js, you gain all of the benefits it brings to the table. It means you can learn more about the user's local environment, which Salesforce applications cannot do independently. For example, you will have the ability to send information to the user's printer.

Utilizing Electron's framework also means you can also build offline applications compatible with Linux, Windows, and macOS and can sync with Salesforce when needed. These benefits and many others make Electron a strong choice for building your Salesforce applications.

Here we will walk through each step of using Electron to access Salesforce data and then render it using Lightning Web Components locally.



PLANNING ELEMENTS

The key components needed to run Salesforce applications on Electron are the Electron framework, NodeJS, expressServer, the lwc basic components bundle, and JSForce to connect the framework to the organization and access the data.

You must have a basic knowledge of Nodejs and express js to undertake this process. You'll also need to know how to handle Lightning Web Components.

If you can do those things, then you'll be able to follow our step-by-step plan for building Salesforce applications using the Electron Framework.

Step-by-Step Plan

- Step 1: Create Salesforce Connected app
- Step 2: Install LWC Basic Components bundle
- Step 3: Install Electron
- Step 4: The Application
- Step 5: Create Accounts LWC component
- Step 6: Update Account record in Salesforce from Electron app

The commits for this plan are also available in this GitHub repository for yo

STEP 1: CREATE SALESFORCE CONNECTED APP

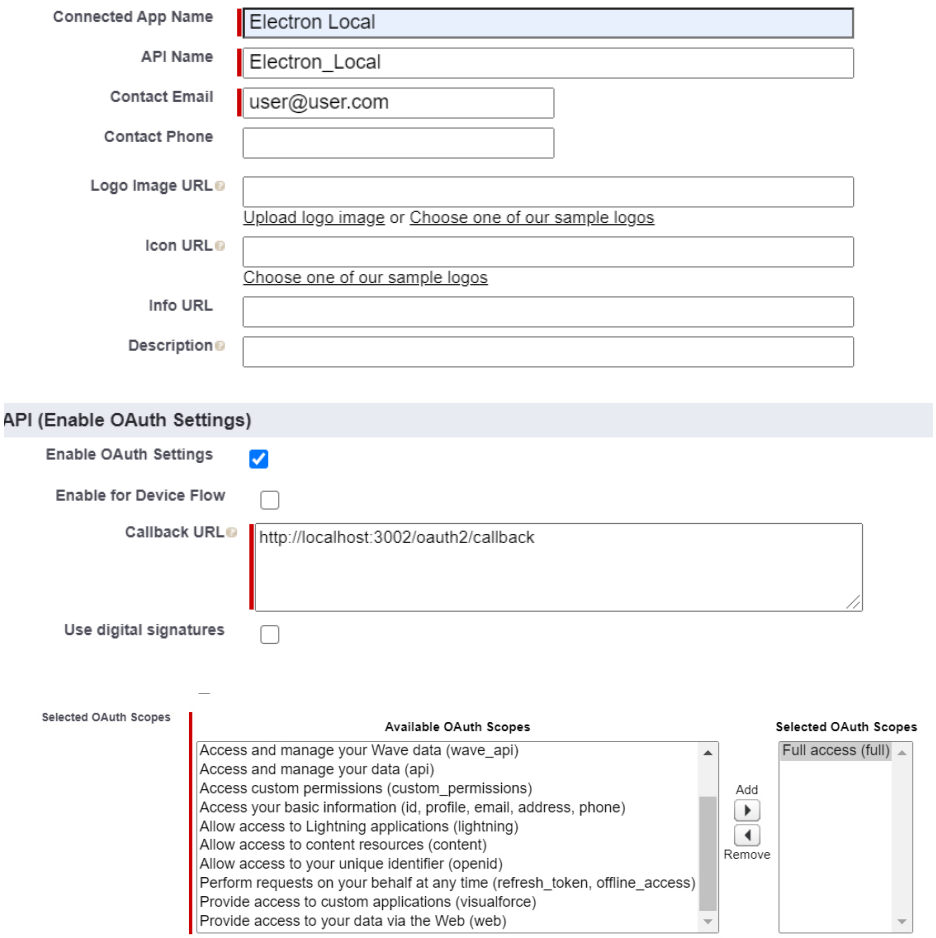
Before we can start using Electron, we first need to log in to our Salesforce org and create a connected app.

Here is how you do that:

- Navigate to **Setup → App Manager**
- Create **New Connected App**
- You can then use the following information for the corresponding requests:

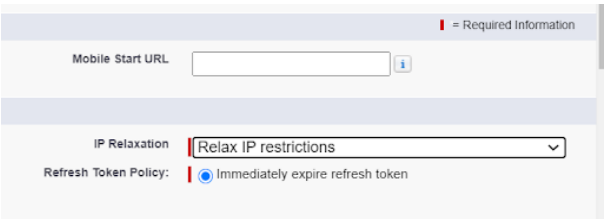
1. **Connected App Name:** Electron Local
2. **API Name:** Electron_Local
3. **Email:** [Use your email address]
4. **Enable OAuth Settings:** [Check this box]
5. **Callback URL:** http://localhost:3002/oauth2/callback
6. **Selected OAuth Scopes:** [Provide full access]

At the end of this, it should look similar to the image below.



Next, you'll need to click **Save** and click the **Manage** button. You should then select **Edit Policies**.

After that, you will need to update the IP Relaxation setting to: **Relax IP restrictions**.

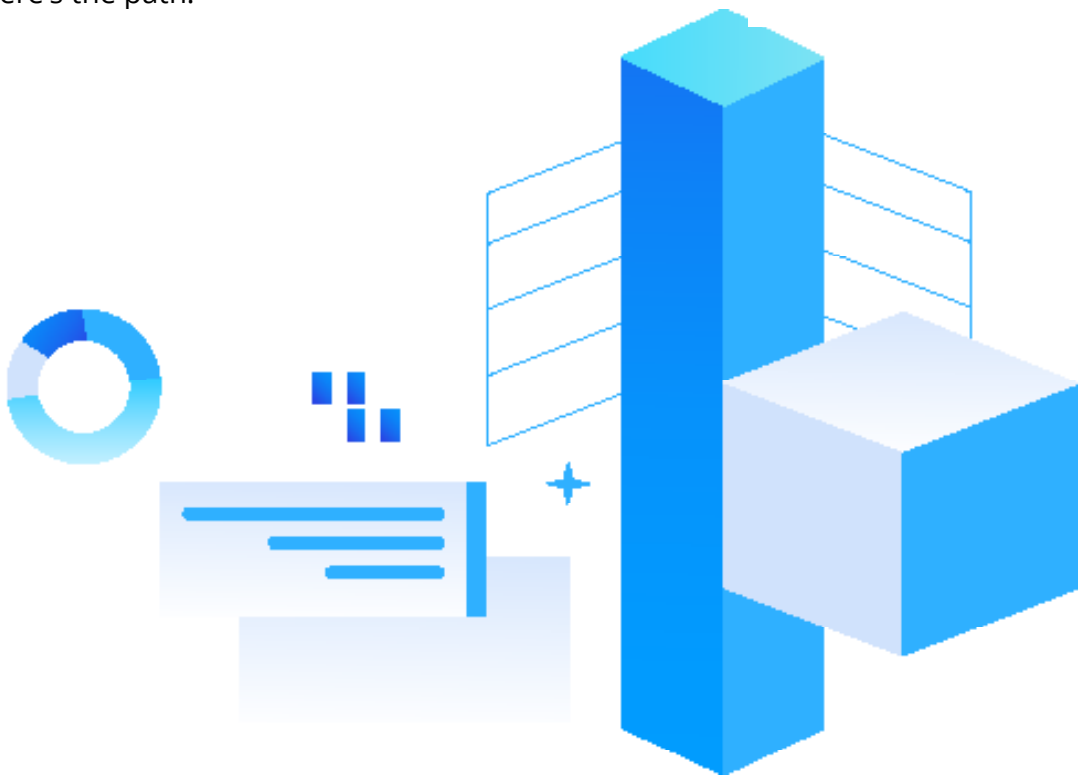


Don't forget to save this part by clicking **save**.

Now you will have the connected app to use to connect to your org.

If you want to copy the consumer key and secret for future use, here's the path:

Navigate to App Mar
your connected app
Consumer Key and



STEP 2: INSTALL LWC BASIC COMPONENTS BUNDLE

To begin this next step, you will need to create a project folder called electron-lwc-demo. This is where you will install and configure the LWC components bundle.

You should then run this command in the terminal:

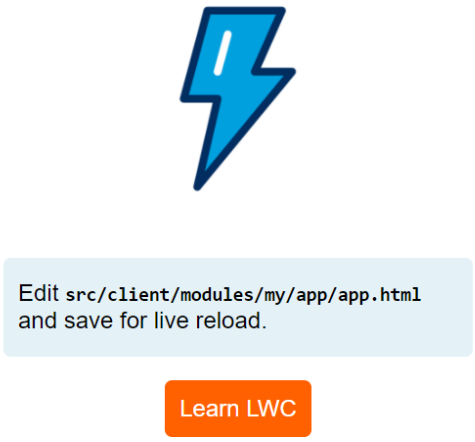
npx create-lwc-app electron-lwc-demo

Put 'y' as the option for **simple setup** and **basic express server configuration**.

Next, navigate to the **electron-lwc-demo** and run the following command:

npm run watch

This command runs Express Server and the LWC components app. If you navigate to http://localhost:3001 in your browser, you will see this page:



One thing to keep in mind is that the LWC components and Express Server run at different endpoints.

The LWC runs at http://localhost:3001, and ExpressJs runs at http://localhost:3002.

The proxy configuration stored in **lwc-services.config.js** handles how LWC components connect to the ExpressJS server.

Once you've done this, you need to install the lightning-base-components package.

First, navigate to the **electron-demo-app** folder and run this command in the terminal:

npm install lightning-base-components

After this components package is installed, we then need to update the LWC module resolver config file. This is important because it tells the module resolver where to look for our web components and then correctly resolve dependencies.

In the root of this project, we currently have **lwc.config.json**. You want to update it with this code:

```
...
  {
"npm": "lightning-base-components"
  }
...
```

After that, you then need to install SLDS resources package. Installing this package will ensure the components look exactly as they appear in the Lightning interface.

Luckily, this is as simple as running one command line in our terminal:

```
npm install @salesforce-ux/design-system
```

Once you've done this, you then need to copy the **assets** folder from **node_modules/@salesforce-ux/design-system** to **src/client/resources**.

Before we can use the **slds** resources we need, you must first add the following

```
<link
  rel="stylesheet"
  href="/resources/assets/styles/
salesforce-lightning-design-system.min.css"
/>
```

You're almost done with this step. The last important thing to accomplish is to update **src/client/index.js** by using the following command:

```
import '@lwc/synthetic-shadow';
```

This synthetic shadow exposes the DOM structure of the Lightning Web Components. From here, you can add styles to components and create your own look.

NOTE: At this moment, the Lightning Web Components bundle is still relatively raw and buggy. To make it work, you may need to go to the **node_modules/lightning-base-components/src/lightning** folder and replace all from 'c/ with from 'lightning/.

You might also need to copy formatted Lookup from this component because it may be missing. If it is, the lightning-

You should feel proud of yourself at this point, as you can now use the LWC components in your application.

Now we are ready to create the Electron application.

STEP 3:
INSTALL ELECTRON

First, you need to install it by running this command:

npm install electron

The starting point of the Electron app is the **package.json** file. You will need to specify the entry point. To do that, modify the **package.json** file in the root folder as follows:

```
...
  "main": "src/main.js",
  ...
  "scripts": {
    ...
    "watch": "run-p watch:client
watch:server start-electron",
    "start-electron": "electron ."
  }
```

This modification adds the **start-electron** command and adds it to the **watch** command. When the **start-electron** command is executed, it will look for the js file. This file was specified in the **main** configuration.

Next, you need to create the **src/main.js** file.

```
const electron = require('electron');
const {app, BrowserWindow} = electron;
const PORT = 3001;
app.on('ready', _=>{
  let mainWindow = new BrowserWindow({
      width 1200,
       height: 800,
      webPreferences: {
        nodeIntegration: true
      }
    });
    mainWindow.loadURL(`http://
localhost:${PORT}/`);
    mainWindow.on('close', _=> {
    mainWindow= null
  });
})
```

Once you've done that, it's time to test your efforts by running this command in the terminal:

npm run watch

Well done! Now you've created your Electron app and pointed your Lightning Base Components app to the main window.

STEP 4:
THE APPLICATION

This is where you get to the part of the process where you can access the Salesforce data and render it in **Electron** using **LWC**.

To keep it simple, you want to modify the main page so that it has a **Login** button. You can set it up so that after the user clicks it, it will connect to Salesforce. Then it can be added to the **Accounts** records, and you can render those with **lightning-data table component**.

Let's start by modifying the **src/client/modules/my/app/app.html**

```
<template>
        <div class="center">
            <img src="./resources/lwc.png" />
    </div>
     <!-- Page code -->
    <div class="center">
      <div>
        <lightning-button
          variant="brand"
          label="Login"
          title="Login"
          onclick={handleClick}
          class="slds-m-left_x-small">
        </lightning-button>
      </div>
    </div>
</template>
```

You've now removed the greetings section and added the lightning-button component.

Next, you'll need to add the handler for the button click in the **src/client/modules/my/app/app.js file**.

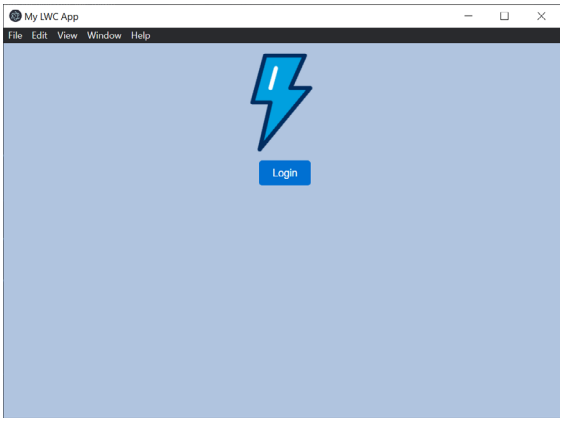
```
import { LightningElement } from 'lwc';
export default class App extends
LightningElement {
    handleClick() {
      let port = 3002;
      window.location.href = `http://
localhost:${port}/oauth2/auth`;
    }
}
```

As you can see, it points to the **URL / oauth2/auth**. While you don't have a route for it yet, you can still go ahead and create it.

After you've done that, you can check yourself by running: .

npm run watch

It will look something like this:



Next, you will update the **src/server/api.js** file:

```
require('dotenv').config();
// Simple Express server setup to serve for local testing/dev API server
const compression = require('compression');
const helmet = require('helmet');
const express = require('express');
const jsforce = require('jsforce');
const connectionService = require('../../scripts/connectionService');
const {CLIENT_ID, CLIENT_SECRET, REDIRECT_URL, LOGIN_URL} = process.env;
let LocalStorage = require('node-localstorage').LocalStorage;
let lcStorage = new LocalStorage('./config');
const app = express();
let oauth2 = new jsforce.OAuth2({
    loginUrl : LOGIN_URL,
    clientId : CLIENT_ID,
    clientSecret : CLIENT_SECRET,
    redirectUri : REDIRECT_URL
});
app.use(helmet());
app.use(compression());
const HOST = process.env.API_HOST || 'localhost';
const PORT = 3002;
let conn;
app.get('/api/v1/endpoint', (req, res) => {
    res.json({ success: true });
});
```

```
app.listen(PORT, () =>
  console.log(
    `✅API Server started:
http://${HOST}:${PORT}/api/v1/endpoint`
  )
);
app.get('/oauth2/auth', function(req, res) {
  //res.redirect(oauth2.
getAuthorizationUrl({ scope : 'full' }));
  res.redirect(oauth2.
getAuthorizationUrl({ scope : 'full' }));
});
app.get('/oauth2/callback',
function(req, res) {
  conn = new jsforce.Connection({
oauth2 : oauth2 });
  let code = req.param('code');
  conn.authorize(code,
function(err, userInfo) {
    if (err) { return console.error(err); }
    lcStorage.setItem('accessToken', conn.
accessToken ? conn.accessToken : '');
    lcStorage.setItem('refreshToken', conn.
refreshToken ? conn.accessToken : '');
    lcStorage.setItem('instanceUrl', conn.
instanceUrl ? conn.instanceUrl : '');
    res.redirect(`http://
localhost:${PORT}/getAccounts`);
  });
});
app.get('/getAccounts', (req, res) => {
  let connection = connectionService.
getConnection();
  if(connection){
    connection.query("SELECT Id, Name
FROM Account", function(err, result) {
      if (err) {
        console.log('err');
        console.log(err);
      }else {
        console.log(result);
        res.json(result);
      }
    });
  }
});
```

You will need to create **scripts\connectionService.js** like this:

```
let LocalStorage = require('node-localstorage').LocalStorage;
let lcStorage = new LocalStorage('./config');
const jsforce = require('jsforce');
let  conn;
let getConnectionParams = function() {
   return {
      instanceUrl: lcStorage.getItem('instanceUrl'),
      accessToken: lcStorage.getItem('accessToken'),
      version: lcStorage.getItem('refreshToken')
   }
}
let getConnection = function(connectionParams) {
    //localStorage.clear();
   if (conn != null) return conn;
   let instanceUrl = lcStorage.getItem('instanceUrl');
   let accessToken = lcStorage.getItem('accessToken');
   if (instanceUrl !== '' && instanceUrl !== null && accessToken !== '' && accessToken !== null) {
      return new jsforce.Connection(
         (connectionParams == null) ? getConnectionParams() : connectionParams
      );
   }
   return null;
}
module.exports.getConnection = getConnection;
module.exports.getConnectionParams = getConnectionParams;
```

After that, you will need to install modules by running the following scripts:

```
npm install jsforce
npm install dotenv
npm install node-localstorage
```

You will then create a .env file in the root folder and populate **CLIENT_ID** and **CLIENT_SECRET** with the Electron_Local app credentials from Salesforce.

You should see something like this:

CLIENT_
ID=3MVG9Lu3LaaTCEgJchaOtnMX5TfBUwI8ZeCXng47FFiZA6v5TV1mWwRRoCRXjCCWxs...

CLIENT_SECRET=4966032253448E037744F5D75E295F52568D1565D451671E92BF5AFC...

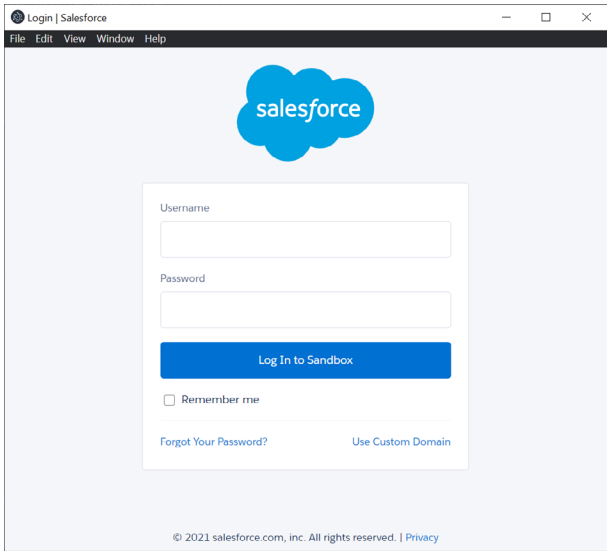
REDIRECT_URL=http://localhost:3002/oauth2/callback

LOGIN_Url=https://test.salesforce.com

Your Electron app changes by running:

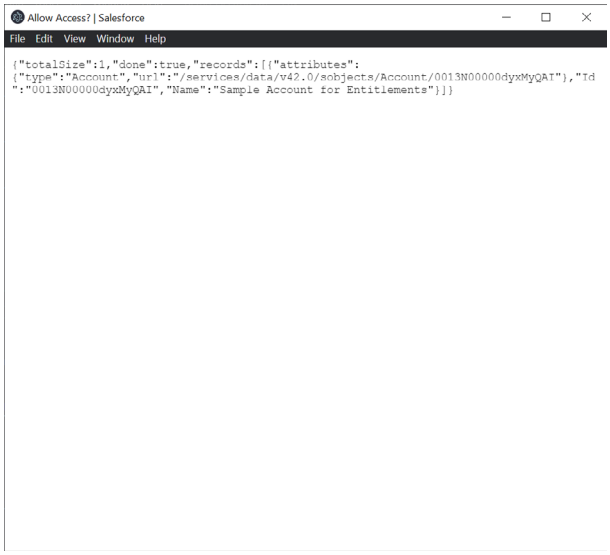
npm run watch

After you click login, you should see the Salesforce standard login form



At this point, you can see how the user credentials will work.

Fill in test user credentials, and you will be redirected to the /getAccounts route. This is what you should expect to see there:



That is the information from your Accounts from the app. You will soon be able to show this data with the **lightning-datatable** component.

To begin this step, you need to update **src/server/api.js** as follows:

```
...
app.get('/isAuthorized', (req, res) => {
  let result = connectionService.
getConnection() ? true : false;
  res.end(result)
});
...
```

Next, create the accounts component in **src/client/modules/my/ folder** with **src/client/modules/my/accounts/accounts.html**:

```
<template>
  <div class='slds-m-bottom--medium'>
    <lightning-button
      variant="brand"
      label="Get Accounts"
      title="Get Accounts"
      onclick={getAccounts}
      class="slds-m-left_x-small">
    </lightning-button>
  </div>
```

```
    <!-- Check if data is available -->
    <template if:true={isAccountsAvailable}>
        <div class='slds-p-around--large'>
            <lightning-datatable
                key-field="Id"
                data={accounts}
                columns={columns}>
            </lightning-datatable>
        </div>
    </template>
</template>
```

Followed by **src/client/modules/my/accounts/accounts.js:**

```
import { LightningElement } from 'lwc';
const columns = [
    { label: 'Id', fieldName: 'Id' },
    { label: 'Name', fieldName: 'Name'}
];
export default class Accounts
extends LightningElement {
    accounts = [];
    columns = columns;
    getAccounts(){
        //using javascrip native fetch
method to get data from server
        fetch('/getAccounts').then(res => {
            res.json().then(data =>{
                this.accounts = data.records;
            });
        }).catch(err => {
            console.error(err);
        });
    }
    //getter method to check if
students array have value
    get isAccountsAvailable(){
        return this.accounts.length > 0;
    }
}
```

After that, you'll create it in CSS by using **src/client/modules/my/accounts/accounts.css.**

```
a[role="menuitemcheckbox"] {
    display: none !important;
}
```

Then you will need to update **src/client/modules/my/app/app.html** with the following commands:

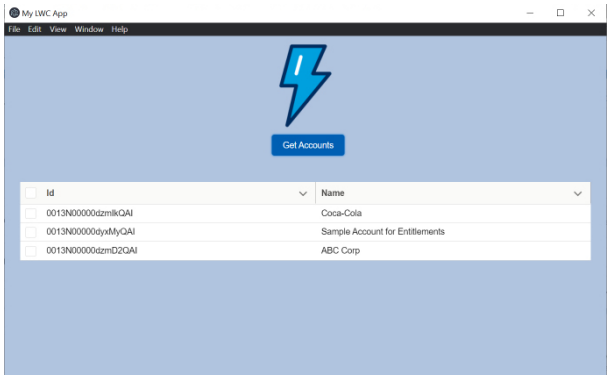
```
<template>
  <div class="center">
    <img src="./resources/lwc.png" />
  </div>
  <!-- Page code -->
  <div class="center">
    <template if:false={isAuthorized}>
      <div>
        <lightning-button
            variant="brand"
            label="Login"
            title="Login"
            onclick={handleClick}
            class="slds-m-left_x-small">
        </lightning-button>
      </div>
    </template>
    <template if:true={isAuthorized}>
      <div>
        <my-accounts></my-accounts>
      </div>
    </template>
  </div>
</template>
```

Lastly, you will update src/client/modules/my/app/app.js:

```
import { LightningElement } from 'lwc';
export default class App extends LightningElement {
  isAuthorized;
  connectedCallback() {
    fetch('/isAuthorized').then(res => {
      res.json().then(data =>{
        this.isAuthorized = data;
        console.log(data);
      });
    }).catch(err => {
      console.error(err);
    });
  }
  handleClick() {
```

```
        window.location.href = `/oauth2/auth`;
    }
}
```

You can then run **npm run watch** to check your results. You will see something like this.



While you could stop here, it's better to keep going and add to this app the ability to update account records in Salesforce.

This will mean creating two more Lightning Custom Components. The first is a custom data table cell for a lightning-datatable component, and the second is a single account record component.

STEP 6: UPDATE ACCOUNT RECORD IN SALESFORCE FROM ELECTRON APP

To begin creating your custom cell component, you'll need to use the following: **src/client/modules/my/customLightningDatatable/editRecordCustomType.html**.

```
<template>
    <a class='slds-p-horizontal--small'
onclick={fireOpenRecordEditAction}>
{recordId}
</a>
</template>
```

Next, you'll create **src/client/modules/my/editRecordCustomType/editRecordCustomType.js** with:

```
import { LightningElement, api } from 'lwc';
export default class
EditRecordCustomType extends
LightningElement {
  @api recordId;
  fireOpenRecordEditAction(e) {
    console.log('fire:' + this.recordId);
    e.preventDefault();
    const event = new
CustomEvent('openrecordeditaction', {
```

```
            composed: true,
            bubbles: true,
            cancelable: true,
            detail: {
                recordId: this.recordId
            },
        });
        this.dispatchEvent(event);
    }
}
```

To create your custom data table component, you will add the **src/client/modules/my/customLightningDatatable** component with the following files:

First, add **src/client/modules/my/customLightningDatatable/customLightningDatatable.html**

```
<template></template>
```

Then add **src/client/modules/my/customLightningDatatable/customLightningDatatable.js.**

```
import LightningDatatable
from 'lightning/datatable';
import editRecordCustomType from
'./editRecordCustomType.html';
export default class
CustomLightningDatatable
extends LightningDatatable {
    static customTypes = {
        editRecordCustomType: {
            template: editRecordCustomType,
            typeAttributes: ['recordId']
        }
    }
}
```

Next, add **src/client/modules/my/customLightningDatatable/editRecordCustomType.html.**

```
<template>
    <my-edit-record-custom-type record-
id={value}></my-edit-record-custom-type>
</template>
```

After that, you need to update **src/client/modules/my/accounts/accounts.html**.

```
<template>
  <div class='slds-m-bottom--medium'>
    <lightning-button
          variant="brand"
          label="Get Accounts"
          title="Get Accounts"
          onclick={getAccounts}
          class="slds-m-left_x-small">
    </lightning-button>
  </div>
  <!-- Check if data is available -->
  <template if:true={isAccountsAvailable}>
    <div class='slds-p-around--large'>
      <my-custom-lightning-datatable
          key-field="Id"
          data={accounts}
          columns={columns}
onopenrecordeditaction={handleRecord
EditAction}
          hide-checkbox-column
      >
      </my-custom-lightning-datatable>
    </div>
  </template>
</template>
```

Followed by updating **src/client/modules/my/accounts/accounts.js**:

```
import { LightningElement } from 'lwc';
const columns = [
  { label: 'Custom Type A', fieldName:
'Id', type: 'editRecordCustomType'},
  { label: 'Name', fieldName: 'Name'}
];
export default class Accounts
extends LightningElement {
  accounts = [];
  columns = columns;
getAccounts(){
    //using javascrip native fetch
method to get data from server
    fetch('/getAccounts').then(res => {
```

```
            res.json().then(data =>{
                this.accounts = data.records;
            });
        }).catch(err => {
            console.error(err);
        });
    }
    //getter method to check if
students array have value
    get isAccountsAvailable(){
        return this.accounts.length > 0;
    }
    handleRecordEditAction(event) {
        const { recordId } = event.detail;
        let account = this.accounts.
find(function(item, index) {
            if(item.Id === recordId){
                return item;
            }
        });
    }
}
```

Once you've reached this point, we will now create the /updateAccount route. This will send account records from the front end. To do that, we need to install a body-parser module, using:

```
npm install body-parser
```

From there, you will need to update s**rc/server/api.js** by using the following script:

```
...
const bodyParser = require("body-parser");
...
app.use(bodyParser.urlencoded({ extended:
false })); // support encoded bodies
app.use(bodyParser.json()); //
support json encoded bodies
...
app.get('/getAccounts', (req, res) => {
    console.log('Getting Accounts...');
    let connection = connectionService.
getConnection();
    if(connection){
        connection.query("SELECT
Id, Name, AccountNumber FROM
Account", function(err, result) {
            if (err) {
console.log(err);
```

```
            lcStorage.clear();
            res.redirect(`http://
localhost:3001/`);
        }else {
            console.log(result);
            res.json(result);
        }
    });
  }
});
app.post('/updateAccount', (req, res) => {
  console.log('Updating account...');
  // Single record update
  let connection = connectionService.
getConnection();
  connection.sobject("Account").update({
    Id : req.body.recordId,
    Name : req.body.accountName
  }, function(err, ret) {
    if (err || !ret.success) {
      lcStorage.clear();

      return res.json({status:
'ERROR', err: err});

    }
      res.redirect(`http://localhost:3001/`);
      return res.json({status:
'ERROR', err: err});
    }
    return res.json({status: 'SUCCESS'});
  });
})
```

As we've done before, you'll need to create a data service component on .js by using the following: **src/client/modules/my/dataService/dataService.js**.

```
let updateAccount = async function
updateAccount(params) {
  const searchParams = Object.
keys(params).map((key) => {
    return encodeURIComponent(key) +
'=' + encodeURIComponent(params[key]);
  }).join('&');
 let response = await fetch('http://
localhost:3002/updateAccount', {
    method: 'POST',
    mode: 'no-cors',
   headers: {
      'Content-Type': 'application/x-
www-form-urlencoded;charset=UTF-8'
```

```
        },
        body: searchParams
    });
    const data = await response;
    return data;
}
let getAccounts = async function(){
    let response = await fetch('/getAccounts');
    const data = await response.json();
    return data.records;
}
export {updateAccount, getAccounts};
```

You will then need to create the account component **src/client/modules/my/account** and include the following files:

First, **src/client/modules/my/account/account.html**.

```
<template>
    <article class="slds-card slds-m-top--large">
        <div class="slds-card__header slds-grid">
            <header class="slds-media slds-media_center slds-has-flexi-truncate">
                <div class="slds-media__figure">
                    <lightning-icon icon-name="standard:account" size="small"></lightning-icon>
                </div>
                <div class="slds-media__body">
                    <h2 class="slds-card__header-title">
                        <a href="#" class="slds-card__header-link slds-truncate" title="Accounts">
                        <span>Account</span>
                        </a>
                    </h2>
                </div>
            </header>
        </div>
        <div class="slds-card__body slds-card__body_inner">
            <lightning-input
                type="text"
                name="name"
```

```
            class="recordName"
            label="Name"
            value='{record.Name}'
            onblur='{handleNameChange}'
          ></lightning-input>
        </div>
    </article>
</template>
```

Next, **src/client/modules/my/account/account.js.**

```
import { LightningElement, api } from 'lwc';
import {updateAccount}
from 'my/dataService';
export default class Account
extends LightningElement {
    @api record;
    handleNameChange() {
        let element = this.template.
querySelector(".recordName");
        updateAccount({
            recordId:this.record.Id,
            accountName: element.value
        }).then(() => {
            const event = new
CustomEvent('afterupdate', {
                composed: true,
                bubbles: true,
                cancelable: true,
                detail: {
                    recordId: this.recordId
                },
            });
            this.dispatchEvent(event);
        })
    }
}
```

Once again, you will need to update **src/client/modules/my/accounts/accounts.html** with this script:

```
<template>
    <div class='slds-p-around--large'>
        <div class='slds-m-bottom--medium'>
            <lightning-button
                variant="brand"
```

```
            label="Get Accounts"
            title="Get Accounts"
            onclick={getAccounts}
            class="slds-m-left_x-small">
          </lightning-button>
        </div>
        <!-- Check if data is available -->
        <template
if:true={isAccountsAvailable}>
            <my-custom-lightning-datatable
              key-field="Id"
              data={accounts}
              columns={columns}
onopenrecordeditaction={handleRecord
EditAction}
              hide-checkbox-column
            >
            </my-custom-lightning-datatable>
        </template>
        <template if:true={isAccountSelected}>
          <my-account
record='{selectedAccount}'
onafterupdate='{handleAccountUpdate}'>
          </my-account>
        </template>
      </div>
</template>
```

Followed by updating **src/client/modules/my/accounts/accounts.js**.

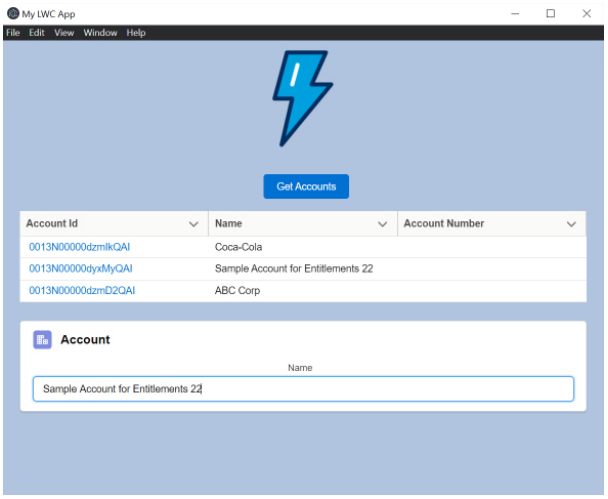
```
import { LightningElement } from 'lwc';
import {getAccounts} from
'my/dataService';
const columns = [
  { label: 'Account Id', fieldName: 'Id',
type: 'editRecordCustomType'},
  { label: 'Name', fieldName: 'Name'},
  { label: 'Account Number',
fieldName: 'AccountNumber'}
];
export default class Accounts
extends LightningElement {
  accounts = [];
  columns = columns;
  selectedAccount;
  isAccountSelected = false;
 getAccounts(){
    getAccounts().then(data => {
```

```
        this.accounts = data;
        });
    }
    //getter method to check if
students array have value
    get isAccountsAvailable(){
        return this.accounts.length > 0;
    }
    handleRecordEditAction(event) {
        const { recordId } = event.detail;
        let account = this.accounts.
find(function(item, index) {
            if(item.Id === recordId){
                return item;
            }
        });
        if(account !== null) {
            this.isAccountSelected = true;
            this.selectedAccount = account;
        }
    }
```

```
    handleAccountUpdate() {
        getAccounts().then(data => {
            this.accounts = data;
        });
    }
```

Congratulations! You did it!

You can run your app check one more time using: **npm run watch:**



One last detail to mention is that when you are making queries to Salesforce, it will respect CRUD, FLS, and sharing settings. If you don't have access to a specific field, you will get an error, and you won't be able to see records.

CONCLUSION

Using Electron to run your Salesforce application gives you the power to do more than just retrieving and updating data in Salesforce. It also includes the full capabilities and power of Node. Js. This means you can learn more about who interacts with your app and their environment. It also means

ABOUT US

SoftServe is a digital authority that advises and provides at the cutting-edge of technology. We reveal, transform, accelerate, and optimize the way enterprises and software companies do business. With expertise across healthcare, retail, energy, financial services, and more, we implement end-to-end solutions to deliver the innovation, quality, and speed that our clients' users expect.

SoftServe delivers open innovation, from generating compelling new ideas, to developing and implementing transformational products and services.

Our work and client experience is built on a foundation of empathetic, human-focused experience design that ensures continuity from concept to release.

We empower enterprises and software companies to (re)identify differentiation, accelerate solution development, and vigorously compete in today's digital economy-no matter where you are in your journey.

Visit our **website**, **blog**, **LinkedIn**, **Facebook**, and **Twitter** pages.

NORTH AMERICAN HQ

201 W 5th Street, Suite 1550
Austin, TX 78701 USA
+1 866 687 3588 (USA)
+1 647 948 7638 (Canada)

EUROPEAN HQ

30 Cannon Street
London EC4M 6XH
United Kingdom
+44 333 006 4341

APAC HQ

6 Raffles Quay
#14-07
Singapore 048580
+65 31 656 887

info@softserveinc.com
www.softserveinc.com

softserve