# Wrap Up Project

In this lab, you will write a complete application, using the concepts you learned in the class.
The application will display a list of Pension objects in a web page.

## Objectives:

- To access PostgreSQL from a Node.js server application
- To expose a simple RESTful API
- To create an SPA front-end using Angular

### Part 1: Installing and Configuring PostgreSQL

#### Steps:

_1. First, you need to install PostgreSQL on your class VM. Follow these instructions:

   a. Open a web browser and navigate to:

      ```
      https://www.enterprisedb.com/downloads/postgres-postgresql-downloads
      ```

      Download the Version 10.14, Windows x86-32 installer to a directory on your VM.

   b. Use Windows Explorer to browse to the directory where the PostgreSQL installer was downloaded. Right-click the installer file and choose *Run as Administrator* to start installation.

   c. Run the installer, accepting all defaults. Please note the password you assign for the PostgreSQL superuser. You can decline the invitation to launch the Stack Builder tool at the end of the installation.

_2. Once the installation finishes, open a command prompt and then change to the PostgreSQL installation directory:

   ```
   cd C:\Program Files\PostgreSQL\10\bin
   ```

   Start the PostgreSQL interactive terminal as the superuser:

   ```
   psql -U postgres
   ```

   Enter the password you set during installation.

_3. Create a new non-superuser user and assign rights to create databases (obviously not very secure):

   ```
   CREATE ROLE me WITH LOGIN PASSWORD 'password';
   ALTER ROLE me CREATEDB;
   ```

   List users to verify creation:


Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
\du
```

You should see the "me" user.

Exit the terminal so you can relogin as non-superuser:

```
\q
```

_4. Restart the terminal and login as "me":

```
psql -d postgres -U me
```

_5. Create a database for our application, then connect to it:

```
CREATE DATABASE mydb;
\c mydb
```

_6. Create a table that will hold pension information:

```
CREATE TABLE pension (
   id SERIAL PRIMARY KEY,
   name VARCHAR(100),
   amount INTEGER
);
```

_7. Insert a few rows:

```
INSERT INTO pension (name, amount) VALUES('Sam Jones', 30000);
INSERT INTO pension (name, amount) VALUES('Sue Smith', 40000);
INSERT INTO pension (name, amount) VALUES('Jo Lee', 50000);
```

_8. Verify insertion:

```
SELECT * FROM pension
```

## Part 2: Install a Modern Version of Node.js

### Steps:

_1. Use Windows Add and Remove Programs to un-install the current Node.js installation.

_2. Open a web browser and navigate to:



Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
https://nodejs.org/en/
```

Under the LTS green box, click "Other downloads" and download the 32-bit Windows Installer (.msi).

_3. Find the downloaded .msi file and run it, accepting all installation defaults.

_4. Close any open command prompts, then open a new one. Test the Node.js install:

```
node -v
```

You should see the version of Node you downloaded and installed.

## Part 3: Writing a Node.js RESTful Service

### Steps:

_1. As you have done in numerous labs, create a subdirectory for your project named **pension-service**, and in it, use *npm init* to generate a starter package.json.

_2. Use *npm install* with the --save option to install the following packages, latest versions:

- express
- pg

The *pg* package is node-postgres, which provides an API to connect to PostgreSQL from a Node program.

_3. Open Visual Studio code in the project directory, then create a file named **pension-service.js**.

_4. Start working on the pension-service.js file:

a. At the top, write code to use the node-postgres driver to connect to the database you created earlier:

```
const Pool = require('pg').Pool;

const pool = new Pool({
  user: 'me',
  host: 'localhost',
  database: 'mydb',
  password: 'password',
  port: 5432,
});
```

b. As a quick test, write a bit of code to see if everything is good so far:



Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
pool.query("SELECT * FROM pension", function(error, results){
    if(error) {
        throw error;
    }

    results.rows.forEach(row => {
      console.log('id %d, name %s, amount %d',
            row.id, row.name, row.amount);
    });
  });
```

Note that this code executes a simple DB query and then prints the returned rows.

c. In a command prompt or the Visual Studio Code Terminal, run the pension-service.js to verify that you can query rows.

d. Leave the pool definition, but comment-out the code you did in the last step.

_5. Within pension-service.js, use the *Express* package to expose the rows from the table with an HTTP GET to the **/pensions** URL. Here are some hints:

- Use Lab 13, Part 3 as a guide and inspiration.
- In this simple service, we will not separate the code into a "DAO" - you can code the Express and PostgreSQL access code in the same file.
- Be sure to require() the *express* package. We don't need the *bodyParser* for this simple service.
- Be sure to code a listen() on some HTTP port number.

_6. Run pension-service.js on Node, then use *curl* to test your endpoint. You should see a returned JSON array that looks like:

```
[{"id":1,"name":"Sam Jones","amount":30000},
 {"id":2,"name":"Sue Smith","amount":40000},
 {"id":3,"name":"Jo Lee","amount":50000}]
```

## Part 4: Writing an Angular Front End

In this part, you will use the Angular skills you learned in this course to create a simple page that displays the pension information.

Use Labs 1, 3 and 4 for inspiration.

### Steps:

_1. Use the *ng new* command to generate a starter Angular project into a directory named **pension-web**. The pension-web directory should be a sibling of the pension-service directory you created earlier in this lab.

_2. Change to the pension-web directory, then use the *ng generate* command to create a component named **pension-list**.

_3. Edit app.component.html so that the root component composes the pension-list component you will complete in a moment:



Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

```
<h1>Pensions</h1>

<app-pension-list></app-pension-list>
```

_4. Using Lab 4 for inspiration, update the Angular code so that the pension-list component uses Angular's HttpClientModule to fetch the pension array from your service, and display it in an unordered list. Here are some hints:

- Update app.module.ts so the module imports HttpClientModule.
- Update pension-list.component.ts so that its ngOnInit() method use the HttpClient to retrieve the pension list. The component should define a field that's an array of Pension objects. In the Observable's subscribe(), assign the returned array into the local field. NOTE: Unlike Lab 4, for simplicity, you can do all of the work in the component instead of writing a separate service.
- Update pension-list.component.html so it uses the *ngFor* directive to display the pension array in an unordered list. For inspiration, see page 4.8 in the Lecture PDF.

_5. Be sure the pension-service is running, then start the Angular development web server. Open a browser and browse to:

```
http://localhost:4200
```

You should see the pension list.

**Note:** It's likely the program will not work as is, even if you coded it perfectly. If you don't see the pension list, open the browser developer tools and look in the console. You may see an error complaining about "blocked by CORS policy". If you see other errors, you'll need to fix them first, but we can solve the CORS problem.

CORS errors are generated by browsers to prevent malign scripts from doing bad things in a web page - if the browser sees a script making an Ajax request to a different server from which the script is loaded, the browser will prevent that access. And that's what our little app does - the Angular program itself is from localhost:4200, but it makes an Ajax request to localhost:3000 where the pension-service lives.

We can fix the problem by telling the browser it's OK to allow the access by putting a special header in the response. The good news is that there's an Express middleware package that will do all of the work for us. Follow these steps to fix the CORS error:

a. Go back to the pension-service, and kill it. Then do an *npm install cors* to get the middleware.

b. Edit pension-service.js, and require() the middleware module:

```
const cors = require('cors');
```

c. Then modify the Express route definition so it calls cors():

```
app.get('/pensions', cors(), function(req, res){.........}
```

d. Run pension-service again, and then refresh the browser that's on localhost:4200. Assuming you have no other problems, the pension list should display and there should be no CORS diagnostics it the browser's console.



Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.

Copyright © Descriptor Systems 2020. All Rights Reserved. Materials may not be reproduced in any form without prior written consent of Joel Barnum.