CPSC304: An Introduction to Postgres and Node.js

Due Date: None

Draft Version: May 11, 2018

1 Introduction

This tutorial aims to give you a brief overview of Postgres, Express.js, and Nuxt.js. There are no deliverables for this tutorial – The goal is to provide you with some options for the course project as well as some exposure to Postgres, which is an open-source DBMS that consistently ranks among the best performing database management systems. Due to its open-source nature, it is very likely that you will use it in the future for your personal projects.

For those who have limited web development experiences, you'll also get some exposure to the Node.js environment, which is one of the most exciting technologies in recent years. Node.js is a JavasScript runtime built on top of Chrome's V8 engine; its package ecosystem is the most vibrant among all open-source projects in the world.

The rest of this document is about setting up the tools for running a demo project using Postgres and Node.js on the UBC Undergraduate CS machines. For those who are interested in learning more, you can find the links below:

1. Postgres – `https://www.postgresql.org/about/`
2. Node.js – `https://nodejs.org/en/about/`
3. Express.js Docs – `http://expressjs.com/en/api.html`
4. Nuxt.js Docs – `https://nuxtjs.org/guide`

Before you continue with the next two sections, you should read Section 4.1 first if you want to install Postgres and Node.js on your personal computer. The entire setup takes about `230MB`. Given that you only have `1GB` of storage on the ugrad machines, depending on how much free space you have left, it may not be possible to complete the entire setup.

2 Setting Up Postgres Database

2.1 Compiling Postgres From Source

If you are installing Postgres DBMS on your personal machines, the easiest way is to find the installers for your operating system on its website. However, since

you do not have root access on the `ugrad.cs.ubc.ca` servers, you will need to compile and install Postgres from source.

1. SSH into `remote.ugrad.cs.ubc.ca`. You should make sure that you have X11 forwarding configured properly, like you did in Tutorial 7.

   ```
   $ ssh -X <a1b2c>@remote.ugrad.cs.ubc.ca
   ```

2. Download compressed source code for release version 10.0 from Github

   ```
   $ wget https://github.com/postgres/postgres/archive/REL_10_0.tar.gz
   ```

3. Unzip the downloaded file, and change into the unzipped folder

   ```
   $ tar -xvzf REL_10_0.tar.gz
   $ cd postgres-REL_10_0/
   ```

4. Configure the source for installation. This will create a postgres/ folder in your home directory in the next step.

   ```
   $ ./configure --prefix=$HOME/postgres/ --with-python PYTHON=/usr/bin/python
   ```

5. Compile and install.

   ```
   $ make -j 4 && make install
   ```

2.2 Creating a Database

You should now have the Postgres DBMS installed, but you don't have a database yet, so we'll initialize a database using the command line tool. Before you can do that, you need to make sure you have your path variable properly configured. Open `~/.bashrc`, and add the lines below to the end of the file.

```
export PATH=$PATH:~/postgres/bin
alias initpostgres='initdb -D ~/postgres/data/'
alias startpostgres='pg_ctl -D ~/postgres/data -l ~/postgres/logfile start'
alias stoppostgres='pg_ctl -D ~/postgres/data -l ~/postgres/logfile stop'
alias startpsql='psql -U <a1b2c> postgres'
```

The first line sets your path environment variable so you can access the Postgres bin/ folder from anywhere. The next line says you want to initialize `~/postgres/data` to store your database. The next three lines are useful aliases that you'll use to start/stop Postgres on the machine and to launch psql (Postgres equivalent of sqlplus). Don't forget to source the bash config file!

```
$ source ~/.bashrc
```

After you've sourced your `.bashrc` file, you should first run `initpostgres`, and then run `startpostgres` in the terminal. You should never run a Linux command without knowing what it does, so here's what it's doing in the back:

1. Create `data/` in `~/postgres`. This is where the database files and data will go. If you have a lot of data in your database, you might need to watch out for the size of this folder.

2. Create a transaction log at `~/postgres/logfile`. This is just a plain text file. If you are going to debug your database, you will find useful error messages here.

3. Finally, it will start the Postgres service, which listens on port 5432. You might hit an error saying that server cannot be started if you are running on the ugrad server. This is most likely because another instance of Postgres is already listening on the default port. In this case, you can manually instruct Postgres to listen on another port, say 5433, by changing the three aliases at the beginning of this section to:

```
alias startpostgres=' ... -o "-p␣5433" -l ~/postgres/logfile start'
alias stoppostgres=' ... -o "-p␣5433" -l ~/postgres/logfile stop'
alias startpsql='psql -U <a1b2c> -p 5433 postgres'
```

   Source your `~/.bashrc` again, and re-run `startpostgres`. You might need to repeat this step multiple times until you find a port that is open.

If you want to delete Postgres, you need to first stop the Postgres service by running `stoppostgres`, then all you need to do is to delete `~/postgres`. You should probably also remove the relevant lines in `~/.bashrc`.

2.3 Using Psql

Once Postgres starts listening on port 5432, you should be able to connect to the database using `startpsql`. By default, when you compiled and installed Postgres, it created a user matching your ugrad ID and a default database called `postgres`. And so `psql -U <a1b2c> postgres` simply says that you want to login to the DBMS as user `<a1b2c>` and connect automatically to the `postgres` database.

The two most useful commands in `psql` are `\?` and `\h`. The first command lists all the Postgres specific action that you can do – Two common ones are `\l` to list all existing databases and `\d` to list all exist tables and views. The second command lists all SQL commands that you can do – These are the `SELECT`, `CREATE`, `DELETE`, etc. that you have learned in class.

For the next part, we are going to create a `Users` table.

```
DROP TABLE IF EXISTS Users;
CREATE TABLE Users (
    userid SERIAL,
    username VARCHAR(32) UNIQUE NOT NULL,
    password TEXT NOT NULL,
    isActive BOOLEAN NOT NULL DEFAULT TRUE,
    PRIMARY KEY (userid)
);
```

3 Setting Up Server and UI With Node.js

3.1 Installing Node.js

The `ugrad.cs.ubc.ca` servers have the stable version of Node.js installed at `/usr/bin/node`. If you want the latest version, you can download the `tar.xz` package from Node's official website.

Every installation of Node comes with `npm`, which is Node's package manager. `npm` lets you quickly download modules that other people have developed and incorporate them into your project.

3.2 Setting Up Starter Project

One of the benefits of using Node.js, and web technologies in general, is that it is extremely portable. As long as you have access to a web browser, you can start developing right away.

1. Clone the starter project from Github

   ```
   $ git clone https://github.com/belinghy/cpsc304.git cs304-starter
   $ cd cs304-starter/demoui/
   ```

2. Install project dependencies listed in `package.json`. This is where the Node.js ecosystem shines. You can install all the dependencies in one command; dependencies will be installed to `node_modules/`. This folder is usually quite large, so you should never commit this folder to your versioning system.

   ```
   $ npm install
   ```

3. The database configuration is defined in `server/configs/sequelize.js`. You'll have to edit this file to change the following line:
   `new Sequelize('<database>', '<studentid>', ...)`.
   Replace `<database>` with `postgres` and `<studentid>` with your student ID, such as a1b2c.

4. In the same file as above, you will need to modify the port if you chose a different port when starting your Postgres server in Section 2.2.

5. Start the web server in development mode and access the demo website. After you run the command below, you should see a message in the terminal like `Server listening on bowen:3000`. You should now be able to access the demo website at `http://bowen:3000` in your favourite web browser (you can launch `firefox` on the remote server, assuming you have X11 forwarding set up).

   ```
   $ npm run dev
   ```

6. Similar to the port problem with Postgres, you might need to run the web server on a different port. You can do this by changing one line in `server/index.js`.

7. Play around with the web app. The demo project contains some basic functionalities that interacts with the database, namely viewing, inserting, and updating data. If you still have `psql` running, you can verify the results by executing SELECT queries.

3.3 Starter Directory Structure

The UI component was built using Nuxt.js, and the easiest way to learn it is to go through the documentation at `https://nuxtjs.org/guide/`. This section provides a very high-level overview of the directory structure.

- `layouts/` – There are two files: `default.vue` and `error.vue`. Every page on the website (as currently configured) uses the layout defined in `default.vue`, and every bad request is redirected to the error page, which is defined in `error.vue`.

- `pages/` – Contains the individual pages you can access in the web app. For instance, the homepage `/` is defined in `/index.vue`, the users page `/users` is defined in `/users/index.vue`. Nuxt.js automatically generates a router based on the directory structure defined in the `pages` folder. More information about routing can be found in the `Nuxt.js` guide.
- `components/` – Contains reusable `Vue.js` UI components. `Vue.js` is a template engine that `Nuxt.js` is built on. Every `.vue` file has three basic components: HTML template, JavaScript script, and CSS styling.
- `assets/` and `static/` – Contains static files of the web app, which would usually be external JavaScript files, CSS templates, and images. `static/` is directly accessible by the outside world when the server is running.
- `server/` – Contains code related to `Express.js` server. As mentioned previously, the database configuration is defined in `configs/sequelize.js`. The `models/` folder contains the object-relational mappings for our database tables. The `api/` folder contains definitions of the REST APIs that the server exposes, and the UI component simply uses these APIs through HTTP to access data in the database. In `api/users.js`, you'll find the SQL statements used to select, insert, and update users.

4 Exercises

4.1 Setting Up Dev Environment on Personal Machine

If you are planning on using Postgres for your project, or for any extended usage, you should probably consider installing and setting up Postgres and Node.js on your personal computer. The setup is extremely easy, since unlike on the ugrad servers, you can simply use GUI installers for each tool. You can find more detailed instructions at `https://github.com/belinghy/cpsc304`.

4.2 Postgres Exercises

1. Add more users to the database.
2. Try out different SELECT queries involving GROUP BY, HAVING, ORDER BY, JOIN, etc.
3. Modify the Users table to add additional columns like Age, Gender, FirstName, and LastName.
4. Create another table to keep track of friendship between users.

4.3 Server and UI Exercises

1. Add functionality to allow for deletion of users.
    - A good starting point is to look at the `server/api/users.js` file. It should be similar to the update logic, but since all you are doing is deleting, you just need an appropriate query and a way to identify the user that you are deleting.

- Then you need to add a button on the UI. When this button is clicked, the UI should send information regarding which user to delete by invoking the API implemented in the previous step. You should be able to achieve this by only modifying the files in `pages/users/_username/` folder.

2. Add filtering conditions to the main users page.

3. Add other pages to display additional tables.