COMP 2406
Winter 2020 - Tutorials #10

Objectives

- Add session support to an existing Express-based web application
- Store session data in a Mongo database
- Implement a simple recommender system

Problem Background

This tutorial will involve adding session functionality to the existing store example we have looked at in several lectures. To start, download the `T10-Base-Code.zip` file, extract the contents, and run `npm install` to install the dependencies. The provided code is a completed version of the store database that uses Express and MongoDB/Mongoose for storage. To initialize the dataset, make sure the Mongo daemon is running and execute `node store-generator.js`. You should then be able to run the `store-server.js` file with Node, access http://localhost:3000, and navigate through the store's data. Take a few minutes to use the Mongo shell or GUI to browse the available data in the `store` database and get an idea of what information is available. As a summary, the following important collections exist:

1. products – contains the name, price, stock of each store product
2. users – contains the user profile data (name, address, etc.) of each user
3. purchases – contains the user ID and product ID for each purchase.

Problem 1 (Adding Session Data)

The first thing you will do in this tutorial is add session data into the application. This will allow you to track the actions of a client as they use the application. To start, you will need to install and use the Express session module. To install the module, use `npm install express-session`. You can then require the module in your server and configure your Express app to use the express-session module using:

```
const session = require('express-session')
app.use(session({ secret: 'some secret key here'}))
```

After you have done this, run the server and access the page. You should be able to see cookie data in the Chrome developer tools that shows a connect.sid property. You can also try logging the session data within a middleware function on your server with `console.log(req.session);`.

Problem 2 (Using Session Data)

You now have an object on the server (`req.session`) that you can use to remember anything about the current client. For example, in assignment #5 you can remember the username of the client and whether they are currently logged in or not. For this application, lets assume you want to remember the IDs of products the client has viewed while the session is active.

Modify the `sendSingleProduct` middleware function in the `products-router.js` file to update the `req.session` object so that it remembers a list of product IDs that have been viewed. For now, you can just log the list of the current session's products every time a request is made. Request some different products and make sure the list is updated correctly. It may be difficult to simulate multiple different clients using a single computer, though creating an "incognito" browser window may allow you to establish a second, separate session.

Problem 3 (Storing Session Data in Mongo)

The main issue with the current approach to storing session data is that it will use a significant amount of RAM if we have many users viewing many products. In this step, we will move the storage of session data into MongoDB. Assuming you have used the `express-session` module for your session data, moving the data into MongoDB using the `connect-mongodb-session` module is a relatively trivial change to make in your code. To start, install the module with NPM using `npm install connect-mongodb-session`. Once the module is installed, require it in your server with `const MongoDBStore = require('connect-mongodb-session')(session)`, where session is your `express-session` object. Then, to configure the express-session module to store its session data in Mongo, you just need to create an instance of the MongoDB store and change your use directive to specify the newly created store should be used for session data storage, like:

```
const store = new MongoDBStore({
  uri: 'mongodb://localhost:27017/store', //store is the DB name
  collection: 'sessiondata' //this is the collection name
});

app.use(session({ secret: 'some secret here', store: store }))
```

Now if you restart your server, everything should work the same. However, if you open the Mongo shell and query the store database, you should see a collection for the session data. If you search for documents in this session collection, you should see all your application's session data is stored in the database. The middleware you have

used handles the reading/writing of the data automatically without requiring further changes to your code (this is a good example of modular code!). You will even find now that if you restart your server, the session data persists, since it is stored in the database.

Problem 4 (Recommending Products)

Now that you know what products a client is viewing, you can update your server to make product recommendations. Update your server and the product.pug template to provide a list of 5 recommended products to the user whenever they are viewing a product page. You can come up with whatever rules you want to use to select recommendations. For a simple solution – recommend products the user has previously viewed, if there are any. If you are looking for something more complex – try finding products that share common buyers with the current product. That is, if the client is viewing product X, find the 5 products that have been purchased the most by people who have already purchased X. The data is present in the database, it is just a matter of extracting it and computing the correct result. If you hate Pug and refuse to use it – sorry! You can always print out the recommended products in the server console and call it a day.

Another common approach to recommending products is to recommend 'similar' products. There are many ways to define and calculate similarity (e.g., using names, categories, reviews, descriptions, keywords). Much of this useful data is not contained in the current dataset, but you could use the dimensions object as a starting point to try out this type of recommendation. For example, try looking for the 5 'nearest neighbours' by finding other products with minimal Euclidean distance between that product's dimensions and the dimensions of the requested product. If you do so, you should end up with recommended products that have similar dimensions to product that the client is currently viewing.

Another measure often used for this type of comparison is called Cosine Similarity. If you are interested, Google it and implement a version using the dimension data as the vectors for each product. While this doesn't make a lot of sense unless you consider similar-sized products to be similar, it is trivial to change the implementation to compute similarity using any other vector you store for each product (e.g., one based on categories or keywords).