This assignment focuses on building a simple AJAX client-server application ("Software as a Service") using Node/Express/EJS and DynamoDB.

1 Overall goal
The goal of this assignment is to implement a restaurant database for Philadelphia. Users can create accounts and log in; they are then shown a map of downtown Philadelphia with markers for the various restaurants. When the user clicks on a marker, she is shown the name of the restaurant and a brief description or review. Users can also create markers of their own, and delete existing markers (but only the ones they have created, not those of other users).

2 First milestone: Account management (100 points)
For the first milestone, you should implement the user accounts.

2.1 Database schema
You should create two DynamoDB tables, one called users for the user accounts, and another called restaurants for the restaurant names and descriptions/reviews. The easiest way to do this is to use the keyvaluestore.js library that comes with your HW4 framework code. (As in HW1, this will result in tables of the form keyword (string), inx (int), value (string), so you can only store a single value for each key. To convert more complex JavaScript objects to strings, you can use JSON.stringify.)
The schema for these tables should be the following:

- In users, the key should be the username, and the value should be an object with two properties: password, whose value should be the users' password (in plain text for now), and fullname, whose value should be the full name of the user. For testing, please create an entry with username mickey, password mouse, and full name Mickey Mouse (you can use the "Explore Table" command in AWS's DynamoDB console to do this, or you can adapt loader.js).
- In restaurants, the key should be the restaurant name. The value should be an object with four properties: one called latitude, another called longitude (these should show the position of the restaurant), a third one called description, which should contain a brief description or review of the restaurant, and a fourth one called creator, which should be the
user name of the user who entered this restaurant into the database. For testing, please create an entry for the restaurant WhiteDog, latitude 39.953637, longitude -75.192883, description Very delicious, and creator mickey.

To check whether your databases have been created properly, you can use the "Explore Table" tool from the DynamoDB web console. You can also use the web console to create the tables if you prefer.

2.2 Frontend design

Next, you should write the JavaScript code for the web application. Your web application should be able to handle seven different URLs (/checklogin, /signup, /createaccount, /restaurants, /addrestaurant, /logout); in the following, these are written http://yourServer/xxx, but of course you should replace yourServer with the actual name of your server (e.g., localhost:8080).

- **Login page:** When the user opens http://yourServer/, she should see a web form with two input fields (for the user name and for the password), as well as a 'Login' button. The input fields should be labeled appropriately - e.g., you could have the text "Enter user name here:" in front of the first field. There should also be a hyperlink to http://yourServer/signup, where the user can create a new account if she does not already have one. The target URL of the form ('action' attribute) should be http://yourServer/checklogin, and the method should be POST. At the bottom of the login page, display your full name and SEAS login name.

- **Login check:** When the user enters a username and password and submits the form, the server should look up this combination in the users table. If the combination is valid, the user should be redirected to http://yourServer/restaurants (see below), and the server should remember (in the session object) that the user has logged in. If the username or the password is not valid, or if one or both are missing (i.e., the corresponding fields were blank), the user should be redirected back to the login page, and an error message should be shown (in red) that explains what happened.

- **Signup page:** When the user clicks on the 'sign up' link on the login page (which should lead to http://yourServer/signup), she should see three input fields: one for a new username, one for a new password, and one for the full name of the user (first and last name). There should also be a 'Submit' button. The method of the form should be POST, and the action (target URL) should be http://yourServer/createaccount.

- **Account creation:** If the user submits the form to http://yourServer/createaccount, if all three fields are completed, and the username does not yet exist in the users table, the account should be created (by adding a new row to the users table), and the user should be redirected to http://yourServer/restaurants. (At this point, the user should also be logged in, of course, i.e., you should update the session object.) Otherwise the user should be redirected back to the signup page, and an error message should be shown (in red) to explain what went wrong.

- **Restaurants page:** If http://yourServer/restaurants is opened and the user is logged in (or has just created an account), she should see a table with all the restaurants that are currently in the database (you need to read the restaurants table). The table should have five columns: one for the latitude of the restaurant, one for the longitude, one for the name of the
restaurant, one for a brief description, and one for the username of the user who entered that restaurant. At the bottom of the page, there should be another web form with four input fields that can be used to add another restaurant. The first field should be for the latitude, the second for the longitude, the third for the name of the new restaurant, and the fourth for a brief description or review. There should be an 'Add' button that submits the form; the method should be POST again, and the action (target URL) should be http://yourServer/addrestaurant. At the bottom of the page, there should be a hyperlink to http://yourServer/logout. If http://yourServer/restaurants is opened and the user is not logged in, the user should be redirected to http://yourServer/ (the page with the login form).

- **Restaurant addition**: If the 'Add' button on the restaurants page is clicked and the form is posted to http://yourServer/addrestaurant, the new restaurant should be entered into the SimpleDB database (with the username of the current user as the 'creator' of the entry) and the user should be sent back to the http://yourServer/restaurants page. The table should now contain a new entry for the restaurant that was just entered.

- **Logout**: If the user clicks on the logout link (which leads to http://yourServer/logout), the user should be logged out (i.e., the user name should be removed from the session object) and redirected back to the main page http://yourServer/ with the login form.

We recommend that you use the code on the Node.js slide deck as a starting point. This code has already been provided to you in the HW4/ folder in your Git repository; to try it out, all you should need to do is add your AWS credentials to config.json, run node loader.js once, then run node app.js, and finally open http://localhost:8080/ in your browser. You should then gradually add routes and handlers for each of the pages. For instance, you could start by changing the handler for / to show the input form described above; then you could add a handler for /checklogin but simply output the result of the check to the web page (e.g., res.send("It worked"), without redirecting the user. Once that works reliably (test with existing and non-existing accounts), you could finish the handler for /checklogin and add the redirection to /restaurants; then implement /restaurants (but perhaps initially without the form at the end), etc.

Feel free to include the Bootstrap CSS to make the forms 'look pretty', but you don't have to - the first milestone can be written in bare HTML if you like. What counts is that you can write the code for accessing the database and for implementing the seven routes.

### 2.4 Submission checklist

- Your tables are called users and restaurants.
- Your solution displays your name and SEAS login on the login page.
- Your code contains a reasonable amount of useful documentation (required for style points).
- You have completed all the fields in the README file.
- You have checked your final solution into the Git repository before submitting it.
- You are submitting a .zip file (e.g., one created by running ant pack from the HW4 folder).
- You removed your AWS credentials from config.json (maybe replace them with xxx).
- Your .zip file contains all of the following:
  - all the files needed to run your solution (including all .js/.ejs files); and
  - the README file, with all fields completed
☐ Your .zip file is smaller than 250kB. Please do not submit large binaries (such as .jar or .class files) or your node_modules directory.
☐ You have declared all of your dependencies in package.json, i.e., if you delete the node_modules folder and run npm install, all the necessary modules are downloaded.
☐ You submitted your solution as a .zip archive via the web interface (!) before the MS1 deadline on the first page of this assignment. (If you choose to use jokers, each joker will extend this deadline by 24 hours.) Submissions in any other form (email etc.) will not be accepted.

3 Second milestone: Google map and AJAX calls (70 points)
For the second milestone, you will add some AJAX features to the application.

3.1 Adding the map
To make it easier for users to find a restaurant near a particular location, you should replace the table on the /restaurants page with a Google Map. The map should have the following features:

- **Marker for each restaurant:** For each restaurant in the database, you should place a marker on the map, using the location of the restaurant. The marker should be yellow if the restaurant was added by the user who is currently logged in, and red otherwise.
- **Pop-up description:** When the user left-clicks on a marker, a small popup should appear that contains: 1) the name of the restaurant, in bold; 2) the restaurant's description from the database; and 3) the phrase "Added by XYZ", in italics, where XYZ is the username of the user who added the restaurant.
- **Left-click to get coordinates:** When the user left-clicks on the map itself (not on a marker), you should fill the coordinates of the clicked location into the input fields (latitude/longitude) at the bottom of the page.
- **AJAX call for addition:** When the user 'submits' the form to add a new restaurant, the page should not be reloaded. Instead, you should add an event handler to the submit button that makes an AJAX call to the server (to enter the new restaurant into the database). Thus, the marker should simply 'pop up' on the map when the submit button is pushed. If an error occurs (e.g., the user left an input field empty, or the AJAX call fails), you can either use the DOM to make an error message appear on the page, or use alert() to display a pop-up window with a message.
- **AJAX call for deletion:** When the user right-clicks on a marker and the marker was added by that user (yellow color), you should delete the marker and make an AJAX call to the server to remove the restaurant from the database. If the marker was added by another user, you should use alert() to pop up a dialog box with an error message.
- **Auto refresh:** Your page should periodically update the restaurants on the map; for instance, if user A is viewing the map and user B adds or deletes a restaurant, the restaurant should eventually appear on (or disappear from) A's map as well. You can do this by setting a timer (say, every 5-10 seconds) and making an AJAX call to retrieve the restaurants whenever the timer expires. When a response arrives, you can then clear and re-draw all the markers. The latter can cause 'flickering' and will make open pop-ups disappear; a somewhat cleaner approach is to keep the list of restaurants in a local variable on the client side, to compare it to the list that the server returns during the AJAX call, and to only add or delete the markers that have actually changed.
The slides already contain example code for AJAX calls and for working with Google Maps. If you need to look up additional details of the Google Maps API, you can find the documentation here: https://developers.google.com/maps/. Notice that you will need a (free) API key to use the Google Maps JavaScript API; this is also available on the Google Maps API web page. Hint: You can add markers to the map without refreshing it!

### 3.2 Adding Bootstrap

As a second step, you should use Bootstrap to make your user interface look a bit 'prettier'. Bootstrap is a simple framework for web development that is based on Cascading Style Sheets; it comes with visual styles, and even "themes", for a variety of different UI elements, such as buttons, input fields, navigation bars, menus, panels, input groups, dialog boxes, etc. You can find out more about Bootstrap on http://getbootstrap.com, and you can find some examples under http://getbootstrap.com/getting-started/. If you would like to know more about CSS, http://www.w3schools.com/css/ is a good place to get started.

The code at the top of this page shows the HTML code for a simple login page that is based on Bootstrap. The \texttt{<link>} element in the header includes the Bootstrap library. At this point, you can assign various 'classes' to input elements to make them look different; for instance, the \texttt{form-control} class is being assigned to the input fields, and the \texttt{btn} class (along with various options) is being assigned to the submit button. The page also includes some additional CSS code (in the \texttt{<style>} element) that contains some additional customizations; for instance, it sets the background color for the page to white, it defines the width of the \texttt{<div>} that contains the login box (330 pixels), and it makes sure that there is a 10-pixel space between the input fields.
To get full credit for this part of the assignment, you should *at least* apply Bootstrap to the elements on your login page. The design itself is up to you - it can be very simple, just like in the example code. However, you should feel free to apply Bootstrap to the other pages as well, e.g., the signup and the restaurants page.

### 3.3 Implementation hints

If you haven't worked with an event-driven framework before, please be aware that the asynchronous nature of Node can be a bit confusing at first; often, the control flow is not the way you are used to. For instance, when you call a function that requires a callback as an argument (e.g., a DynamoDB lookup), please keep in mind that the function will return immediately, and that the callback will be called sometime later (e.g., when the results of the lookup are available). Another tricky point to watch out for is loops: for instance, you cannot simply iterate over the elements of an array and call some blocking function (e.g., a DynamoDB lookup) during each iteration. Have a look at Node's `async` package; this contains a number of helpful methods for such situations.

### 3.4 Debugging hints

Debugging AJAX applications can be a bit tricky because you cannot necessarily see the effect of your AJAX calls on the current web page. We recommend that you add `console.log()` calls, both on the client side and on the server side, so you can see which calls are being made, and whether or not they succeeded. On the server side, you can see the output in Eclipse's console (if you started the server from there) or in the terminal (if you launched the server using the `node` command); on the client side, you can use the browser's developer console. If you need to test what happens when two users are accessing the page at the same time (needed for the auto refresh feature), a simple trick is to open two different browsers, e.g., Chrome and Firefox - their cookies will be different, so you can use two different sessions.

### 3.5 Submission checklist

- Your submission satisfies all the requirements from Section 2.4.
- You submitted your solution as a `.zip` archive via the web interface (!) before the MS1 deadline on the first page of this assignment. (If you choose to use jokers, each joker will extend this deadline by 24 hours.) Submissions in any other form (email etc.) will not be accepted.

Keep in mind that jokers *cannot* be applied to the final project; if you have any jokers left, this will be your last opportunity to use them. Please remember to use the web interface to apply any remaining jokers; we will *not* apply them for you automatically.