# WDC Web App Project Documentation

# Sample Data for Testing

These values have been added to the database inside the webDatabase.sql file which also includes the database schema.

Users:
chris.hemsworth@gmail.com
Thor5678
hugh.jackman@gmail.com
Wolverine123

Venue:

| | | |
|---|---|---|
| University of Adelaide | adelaideuni@gmail.com | AdelaideUni1234 |
| Adelaide Airport | adelaide.airport@gmail.com | AdelaideAirport1234 |
| Rundle Mall | rundle.mall@gmail.com | RundleMall1234 |

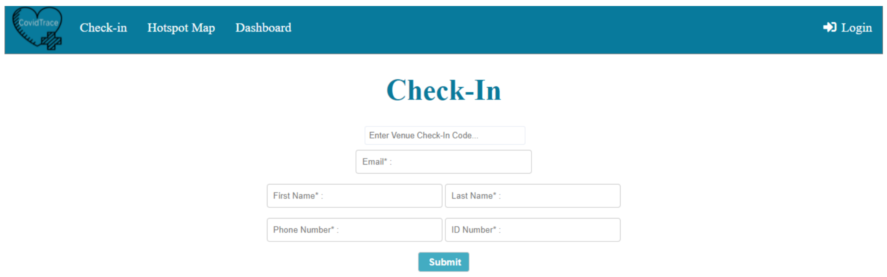
Admin:
admin@wdcproject.com
Admin123
admin2@wdcproject.com
Admin456

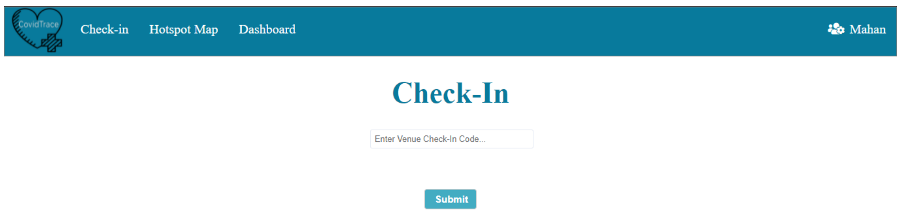
Hotspots:
196 Grenfell St
Jeffrey Smart Building, 217/243 Hindley St
299 Montacute Rd, Newton 3 Weeks old

CheckIn:
Chris.hemsworth -> Adelaide Airport
Chris.hemsworth -> Rundle Mall
Hugh Jackman -> Rundle Mall

Notes:
The hotspots should make Rundle Mall and University of Adelaide Hotspots because they are within a kilometer. Removing “196 Grenfell St” hotspot will result in Uni of Adelaide not being a hotspot but Rundle Mall still being a hotspot (because Rundle mall is still within a km of Jeffrey Smart Building. These will be reflected in Chris Hemsworth’s user history.
Running the weekly code will result in 299 Montacute Hotspot being deleted because it is older than two weeks.

## Features and Design Choices





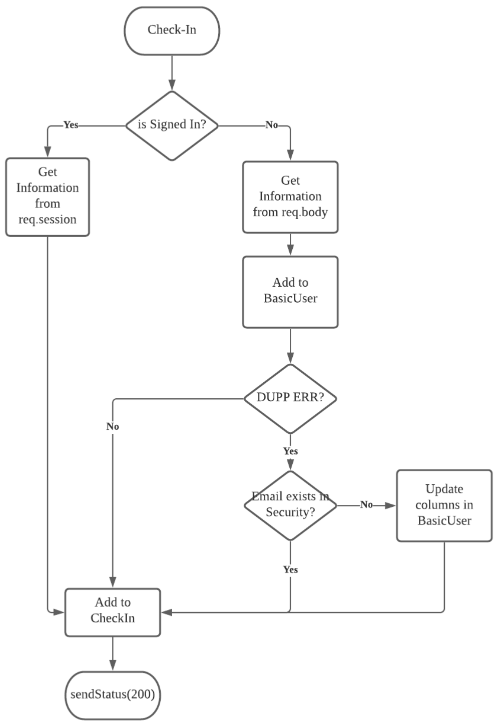
### Homepage / Check-in.html

For our homepage, we decided to set our check-in page as our homepage. Since the task that most users will perform day to day on our website is checking in, we made it the default homepage which reduced the cognitive and kinematic load for checking in.

This page utilizes vue to view or hide input elements. If a user is signed in, all input elements except the check-in code input element are hidden because we already have the required information. This helps with quickly checking in.

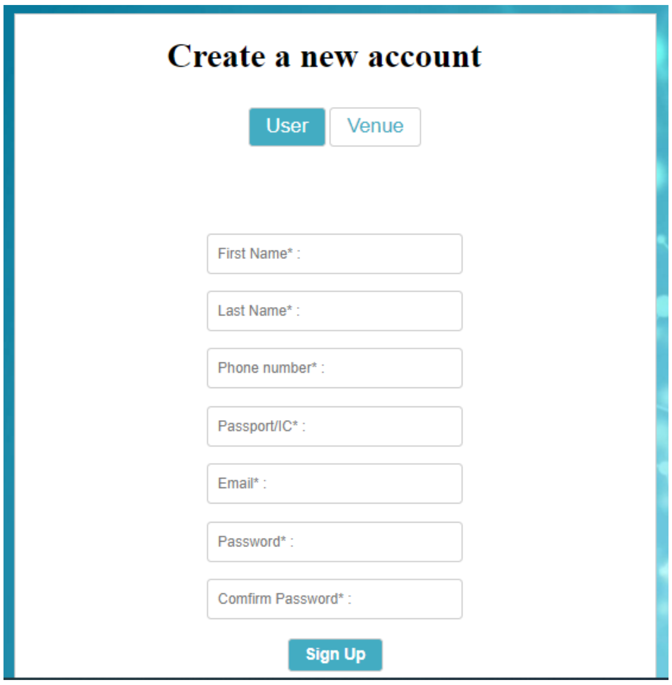
The check-in code input is a vue template imported. It is a searchable dropdown box that looks like a normal text input field. As the user starts to type the check-in code, an "autocomplete" recommendation is given. We implemented this because we found that manually entering check-in codes was difficult and hard to remember.

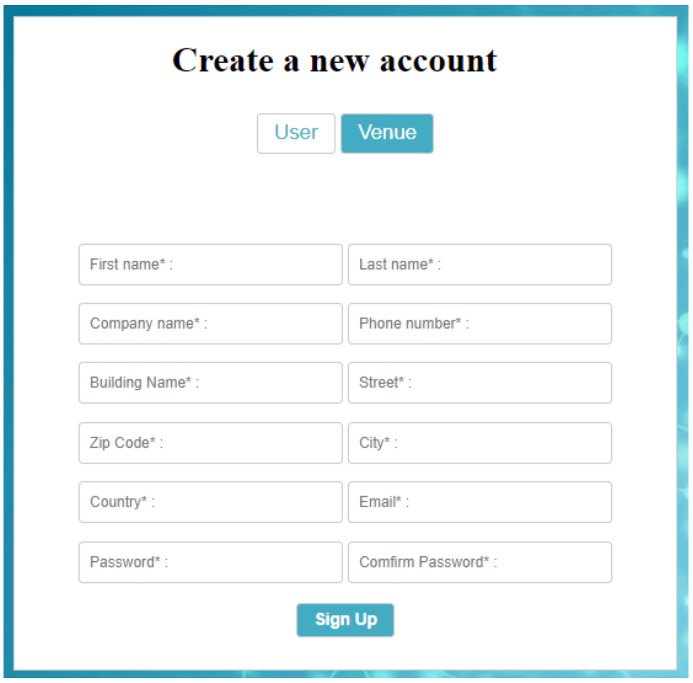
During our research, we discovered that other contact tracing applications, allowed users to check-in to venues without being signed in or even having an account at all. The following flowchart clearly explains the workflow of this feature:


Check-In
is Signed In?
Yes
No
Get Information from req.session
Get Information from req.body
Add to BasicUser
DUPP ERR?
No
Yes
Email exists in Security?
No
Update columns in BasicUser
Yes
Add to CheckIn
sendStatus(200)

The manual check in will update the records of the user in our database ONLY if they have never signed up with us. This was to ensure security because anyone can do the manual check-in and update a user's data but if the user has signed up with us then we take the information provided during sign up to be the concrete.

## Signing up





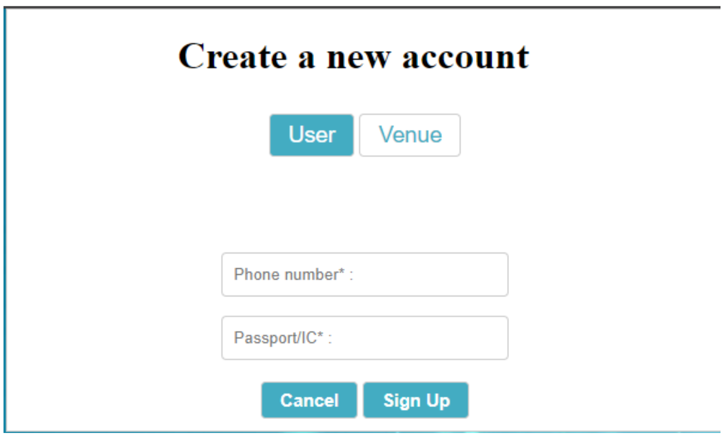
The signup has been combined into one html page which seamlessly changes between user and venue signup based on the client’s choice. The fields use placeholder text to indicate what data it requires and throughout our testing, we found that these fields were accurately suggested by Google Autofill. It is able to fill in addresses correctly.

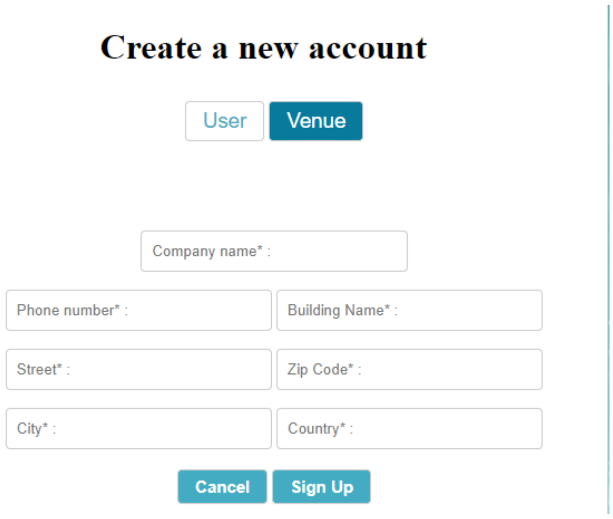
## Forward Geolocation

In our venue signup and hotspot adding features, we implemented a forward geolocation API which we found online. We used the free geolocation API provided by LocationIQ to convert addresses to longitude and latitude coordinates.
This was done because we needed geographic coordinates to display locations on mapbox. Therefore, every time a venue is signed up or an admin/health official adds a new hotspot, the client javascript code makes a HTTP request to the API to retrieve the geographical coordinates before sending everything to the server.
Although the API worked really well, sometimes it takes multiple API calls for it to find the location. If this happens we simply ask the user to try again. It is somewhat of an inconvenience which can be fixed by changing to other geolocation APIs or even using Google’s APIs.

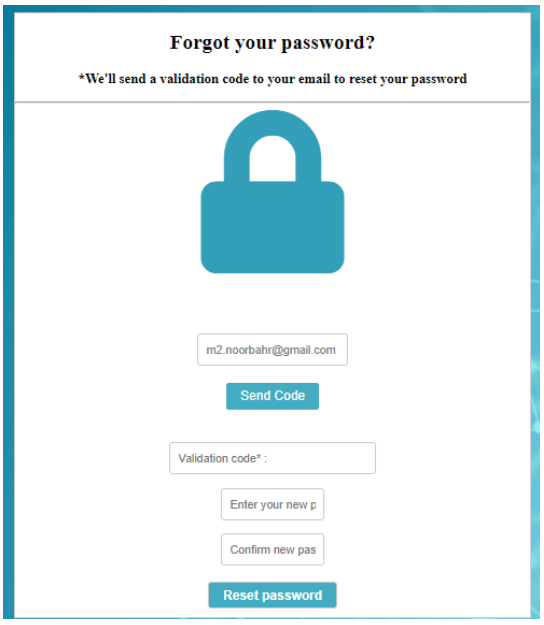
## OpenID Login & Signup

The Google sign in button is found in the login page of the website. If the Google account used to sign in is already registered, we log them in. However, if they have not registered with us before, we need some extra information that the Google sign in does not provide. The client is then taken to a special sign up page that only includes the required information fields.



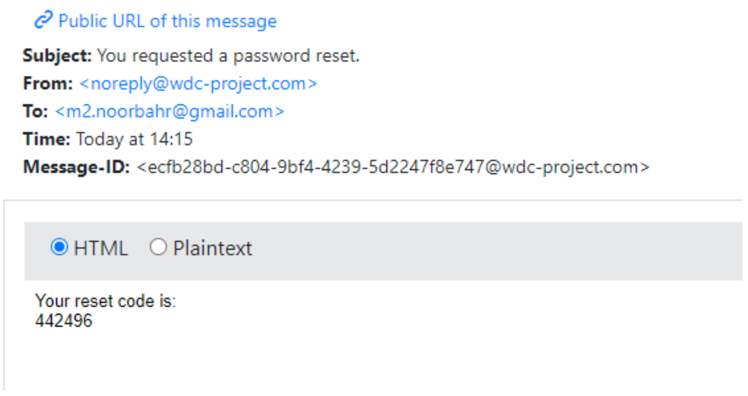


The Google sign in button is also present in the HTML but is hidden. This is because it will run a function in the javascript that helps us retrieve the info that we need to send to the user. They can also choose to cancel and they will be logged out of Google sign in.



## Forgetting your password

We have implemented a “forgot your password” feature which emails the user a unique reset code that they can use to authenticate themselves before they are given the option to reset their password. An example of this email:



Public URL of this message

**Subject:** You requested a password reset.
**From:** <noreply@wdc-project.com>
**To:** <m2.noorbahr@gmail.com>
**Time:** Today at 14:15
**Message-ID:** <ecfb28bd-c804-9bf4-4239-5d2247f8e747@wdc-project.com>

● HTML ○ Plaintext

Your reset code is:
442496

## Dashboard

In the dashboard, each account type has their own set of tabs which allow for different functionalities.

## Editing Profile Information

Each account type can edit their profile information. We decided that some information is immutable. For example, we decided that accounts cannot change their emails.

## User / Venue History:

Users can view their history in the dashboard. They can also click to see the history displayed on a map. The map centers on the latest venue they have visited so that they don't have to scroll out and look for their country.  In the list of venues they have visited, it can also say if that venue is currently a hotspot or not. Venues become hotspots when they are within 1km of a hotspot.
Venues can view their history which is a list of users.
Both lists are searchable.

## Email Subscriptions

Users are automatically signed up for email notifications when they sign up. They can disable these in the dashboard.

## Venue Check-In Code

The venue check-in code needs to be unique and easy to remember / share. We decided to create our own algorithm which generates these check-in codes. It generates codes that include letters from the business name, making it easier to remember (they don't need to memorise the whole thing since checking-in has autocomplete functionality). The codes are fixed length (10 characters) and consist of two segments, text and number.
The text segment is created using the business name. The algorithm tries to strip any vowels from the name. For example, McDonalds becomes MCD or MCDND. If the stripped version is too short or unusable, it will take a slice of random length from the original name .The length of the text segment is chosen randomly from 3 to 7 characters. The remaining characters are numbers, made using the account's email address. Since the email address is unique, the numbers generated are very likely to be unique.

## Admin Features

"Admins" refers to both admins and health officials. They are under one account type.

## Adding hotspots

Admins can add hotspots using the address of a hotspot. We decided that hotspots are completely separate from venues. When an admin adds a hotspot, the server compares the distance of venues to the hotspot and if the distance is a kilometer or less, the venue is marked as dangerous.
This allows hotspots to be locations on the map and "bubble" areas instead of just one venue. A hotspot can mark many venues as dangerous if they are all close to it.

## Editing / Deleting Everything

Everything can be created, can be destroyed. Users, venues and hotspots can all be edited or deleted from the database by an administrator. The server takes care of all deleting and ensures nothing breaks.

## Signing up Admins

Admins can only be created by other admins. This is a tab in the dashboard page for admins. Password validation and account duplication is also checked by the client side javascript.

## Weekly Tasks

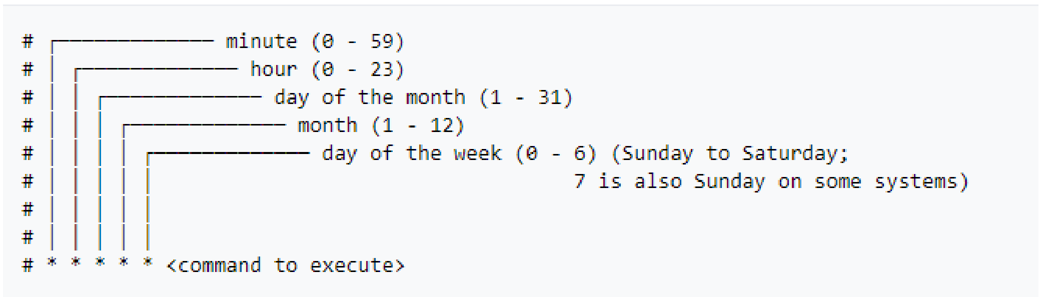
We have set up the server to run a function every week on Monday at 10am. This function does the following tasks:

- Delete Hotspots older than 2 weeks from the Hotspots table.
- Check if venues that were marked as hotspots stopped being close to a hotspot.
- Email users about hotspots that were added in the past week.

**NOTE:**
The final code has this weekly task running every Monday at 10am. This is set in this line in app.js

**cron.schedule('0 10 * * 1', function()**

The schedule consists of 5 values. A start represents the “every” value for the unit. The following illustration demonstrates the meaning of each value:



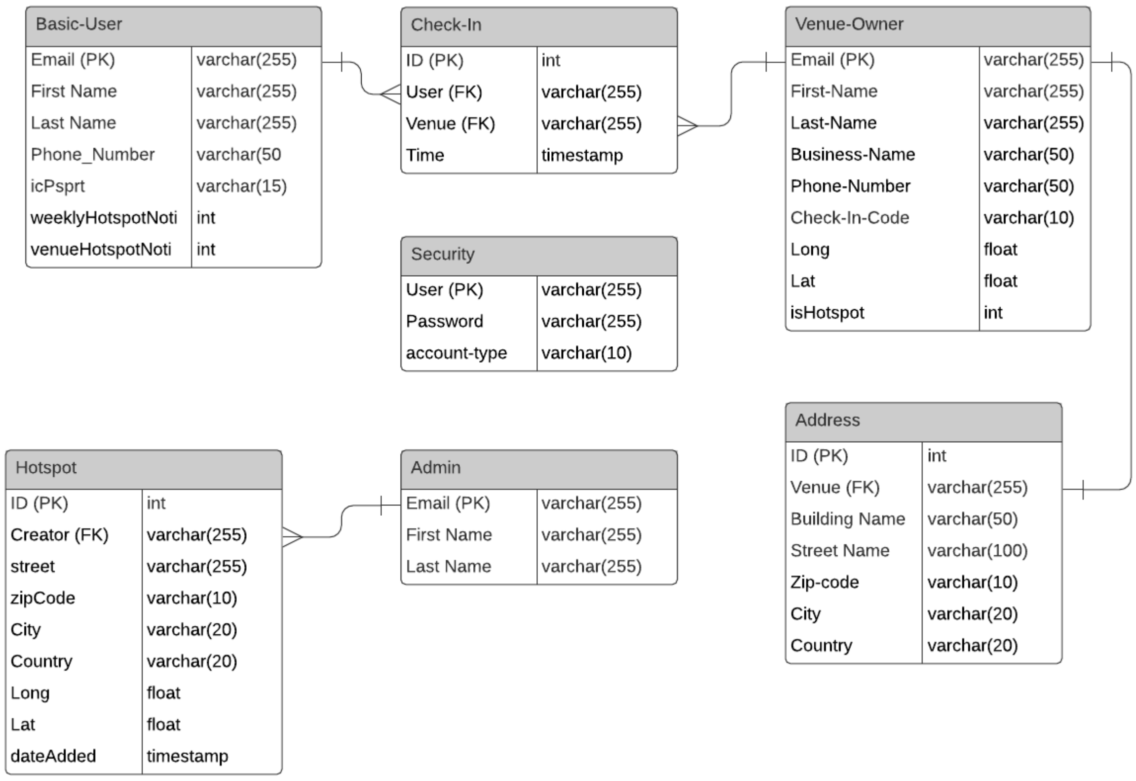
```
# ┌───────────── minute (0 - 59)
# │ ┌───────────── hour (0 - 23)
# │ │ ┌───────────── day of the month (1 - 31)
# │ │ │ ┌───────────── month (1 - 12)
# │ │ │ │ ┌───────────── day of the week (0 - 6) (Sunday to Saturday;
# │ │ │ │ │                                   7 is also Sunday on some systems)
# │ │ │ │ │
# │ │ │ │ │
# * * * * * <command to execute>
```

We suggest, for the purpose of testing this function, set every value to * and then uncomment the test code present in the function. This makes sure the function only runs once even though it is scheduled for every minute of every hour. This method of testing is presented in the video demonstration.

## Security and Passwords

Passwords are NOT stored in the database as plain text. They are hashed and salted before being stored in the database.
All data that the user directly puts into the database (data that is not processed) is stored as text. This ensures that SQL injection does not happen on our website.
The only data that is not stored as text is longitude and latitude coordinates. These two values are not entered by the user but instead by the geolocation API.
Middlewares are used throughout the server to ensure authorization. For details of how each route handles authorization, you can refer to the routes documentation chapter in this document.

# Final Database Schema



The overall structure of the database has not changed much from the initial schema. Some fields were added and some were deleted during development based on need for features.
The password field in the Security table does not store plain passwords. It stores hashed and salted passwords which ensure safety of passwords if the database was to be accessed by an unauthorized party.
The Security table is not specifically related to any other table, however, the server code will, depending on the value of account-type, access other tables using Security's user column. During development, this worked very well and caused no issues.

Required Modules:

| | |
|---|---|
| mysql | (For databases) |
| express-session | (For sessions) |
| argon2 | (For password hashing) |
| node-cron | (For scheduling weekly task) |
| nodemailer | (For sending emails to users). |

# Updated routes list

## General functions

### queryDatabase(req,res,next,query,finish);

This is a function that is excessively used throughout the server to minimize repetition of code. It will make a connection with and query the database using the given req object and the query string. finish is a boolean which indicates whether we want to call res.json on the result of the query. This option was implemented because often in routes, we want to call multiple queries back to back and so we don't want the first query to call res.json on the result. This is crucial for routes that add or alter multiple tables.

### getDistance(lat1, lng1, lat2, lng2);

This is a function that is used to calculate the distance between two pairs of longitude and latitude coordinates. Crucial for calculating the distance between a hotspot and venues nearby. This is because every venue within a kilometer radius of a hotspot is considered to be dangerous and the users who have visited that venue will be notified via email.

## app.js

### cron.schedule();

This is not an express route but it is a cron scheduled function that is run every Monday at 10am. This function will take care of weekly tasks such as removing hotspots that are older than 2 weeks old. Updating venues that are no longer affected by hotspots that were just deleted. Send weekly email updates to users who have subscribed to them.

### app.use();

Global route that enforces authorization of certain pages. This route is responsible for ensuring that only signed in users can access dashboard.html or historyMapView.html. It also stops users who are signed in, from accessing login.html and signup.html.

## index.js

### GET '/'

This is the default route. We use this to redirect the user to check-in.html since we chose that to be our "homepage".

### GET '/hotspots.ajax'

This is a route that is index.js because no authority is needed to access this route. This route will send details about every current hotspot to the client. This route can handle URL parameter called "columns". If columns are provided, the route will query the database for only the selected columns. Useful for hotspot-map.html where we only need longitude and latitude values of hotspots.

## users.js

### generateCheckInCode(email,bName);

is a self explanatory function that uses an algorithm to generate a near* unique check-in code for a venue. The check-in code is split into two parts, text and numbers which forms a check-in code of length 10. A venue is uniquely identified using its email address so having the check-in code be generated using the email address ensures a very high chance of uniqueness. The text segment is generated using a stripped version of the business name (bName), around 3 to 7 letters in length, and a string of numbers calculated from the email address. A degree of randomness is introduced for the length of each segment, further ensuring uniqueness.

### generateResetCode(email);

is a function that uses an account's email address to generate a random password reset code. This code is then emailed to the user for authentication when the user forgets their password and wants to reset it. It works very similar to many other websites.

### POST '/sendRequestCode.ajax'

this route is called when a user requests for a password reset. This route uses the generateResetCode function to generate a unique reset code for the user. It will then email the code to the user while also storing it in the session object so that it can authenticate the password reset in 'resetPassword.ajax' route. Responds with code 401 if the correct values are not provided or if the email is not in the database.

### POST '/resetPassword.ajax'

this route will reset a user's password after authenticating the provided reset code against the reset code that was emailed, sending a 401 status code if any problems arise. The password is hashed before being put in the database.

### GET '/details.ajax'

this route is a very crucial route that is used by every single page in the website. The prep.js file that is included in every html page, calls this route to get the current session details. Important information such as whether they are logged in, account type and others. Since different account types have different values, the route will put every property of req.session (except some like cookie) into a json object to be sent. If this data is not present in session (marked by rest property missing), the route will get the information from the database and add each column as a property of req.session.

### POST '/tokenLogin.ajax'

this route is used to sign in a user when they choose to sign in using Google. It will authenticate the token provided and extract the email of the user. It will then check if that email exists in the **Security** table. If it exists then it will set the appropriate session data. If not, it will send a response code of 404 which the client will understand and redirect the user to a special sign up page. It will respond with 401 if the provided token is not legitimate.

### POST 'login/ajax'

this route is used for normal, non-google, logins. It will query the database for the given email. It will then use argon2's library to verify the plain password with the hashed password from the database. If the verification is successful, it will set up the needed session properties. If any problems occur during this process, a response code 401 is sent.

### GET '/logout.ajax'

this self-explanatory route will clear every property of req.session except for cookies. Throughout the server, a lack of req.session.user and other properties is seen as being logged out.

### POST '/token-user-signup.ajax'

this route signs a user up who signed up using Google. It will insert the data into 2 tables, Security and BasicUser, setting the required columns. If insertion into Security results in ER_DUP_ENTRY then we will respond with 400 which the client will understand and display an appropriate message. If insertion into BasicUser results in ER_DUP_ENTRY then the route will instead of inserting, update the values for that user. We allow this because a user can be in BasicUser without having an account because they can check-in to a venue without being logged in.

### POST '/user-signup.ajax'

this route signs a user up who signed up manually. It will insert the data into 2 tables, Security and BasicUser, setting the required columns. If insertion into Security results in ER_DUP_ENTRY then we will respond with 400 which the client will understand and display an appropriate message. If insertion into BasicUser results in ER_DUP_ENTRY then the route will instead of inserting, update the values for that user. We allow this because a user can be in BasicUser without having an account because they can check-in to a venue without being logged in.

### POST ‘/token-venue-signup.ajax’

this route signs a venue up who signed up using Google. It will insert the given data into 3 tables, Security, VenueOwner and Address, setting the required columns. Unlike user signup, if for any of the insertions, a duplicate error occurs, we respond with code 400 which the client will understand and display an appropriate error message.

### POST ‘/venue-signup.ajax’

this route signs a venue up who signed up manually. It will insert the given data into 3 tables, Security, VenueOwner and Address, setting the required columns. Unlike user signup, if for any of the insertions, a duplicate error occurs, we respond with code 400 which the client will understand and display an appropriate error message.

### GET ‘/check-in-codes.ajax’

this simple route will send the email and checkInCode of venues in the database. Used by check-in.html to populate the searchable dropdown menu.

### POST ‘/check-in.ajax’

this route will check in a user. If the session is signed in, it will get the user’s email from req.session and insert a row into the CheckIn table using the provided venue email. If the user is not signed in, it will first insert the user details into BasicUser. If the user is a duplicate in the table, it will update the columns. FInally, it will add a row to the CheckIn table. This means that users can check into a a venue without being signed in. If they do this multiple times for the same email, their details will be updated each time.

### router.use();

a global middleware that authorizes the user before they can access any routes past this point. All routes past this middleware require the user to be signed in so this middleware performs the authorization.

### POST ‘/updateInfo.ajax’

this route is used by dashboard.html ‘s profile section to update the user’s information. Since a user’s information can not only be updated by themselves, but also by an admin, this route will update anyone’s information, not just the session’s user. To prevent anyone from changing other’s information, we make sure that the change target is the session’s user OR the session’s account type is admin. The route then converts the req.body’s properties into a formatted string of “property = value, “ which can be concatenated and passed to the query.

### GET ‘/mapHistory.ajax’

this route sends the client the longitude and latitude coordinates of each venue that the user has visited. The route will only send the session’s user’s check-in history and not anyone else’s.

### GET ‘/venueAddress.ajax’

since venue addresses are not stored inside VenueOwner table, they do not appear in the result of /users/details.ajax therefore the venue’s address must be retrieved separately. This route returns the currently signed in venue’s address.

### POST ‘/update-venue-Address.ajax’

since a venue’s profile information is updated using ‘/users/updateInfo.ajax’, the address needs to be updated separately. This route updates the currently signed in venue’s address.

### GET ‘/venueHistory.ajax’

this router simply returns a json list of every user who has visited the currently signed in venue.

### GET ‘/:user/checkInHistory.ajax’

this route returns a json list of every venue the user has visited.

### GET ‘/:user/updateEmailPref.ajax’

this route updates the user’s email preferences. Either subscribing or unsubscribing them from the two email lists.

### router.use()

a global route that filters any requests NOT from an admin account. All routes beyond this point need the currently signed in account to be an admin account.

### GET ‘/:venueEmail/venueAddress.ajax’

this route returns the address of the venue requested. Used for the “Manage Venue” tab of admin’s dashboard.

### GET ‘/user-details.ajax’

this route returns a json list of users. Basically every row from the BasicUser table. Used for the “Manage Users” tab of admin’s dashboard.

### GET ‘/venue-details.ajax’

this route returns a json list of venues. Basically every row from the VenueOwner table. Used for the “Manage Venues” tab of admin’s dashboard.

### POST ‘/admin-signup.ajax’

this route will use the provided data to sign an admin up. It inserts data into 2 tables, Security and Admin, setting the required columns. If any insertion results in a duplication error, a response code of 400 is returned and the client will display an appropriate message.

### POST ‘/add-hostpot.ajax’

this route adds a hotspot to the database. After adding the hotspot to the database, It will calculate the distance between the newly added hotspot and every venue in the database. If the distance is less than a kilometer, it will mark that venue as a hotspot and email everyone who has been to that hotspot in the past two weeks.

### POST ‘/update-hotspot.ajax’

this route updates a hotspot in the database. It receives a modified hotspot object from the client and updates the correct row based on the hotspot id.

### POST ‘/delete-hotspot.ajax’

this route deletes a hotspot in the database according to the given id. It then calculates the distance from every venue (that is currently a hotspot) to every hotspot to see if those venues have stopped becoming a hotspot as a result of the hotspot delete.

### POST ‘/delete-venue.ajax’

this route deletes a venue. It deletes rows from 4 tables in specific order, CheckIn, Address, VenueOwner and Security. This removes any trace of the venue from our database.

### POST ‘/delete-user.ajax’

this route deletes a user. It deletes rows from 3 tables, CheckIn, BasicUser and Security. This removes any trace of the venue from our database.