AutoGarcon Final Report

Team: Database
Members: PJ Cappitelli, Emma Sinn, Brandon Tran, and Tucker Urbanski

Table of Contents

<u>Section 1: Introduction</u>

More and more people are moving away from dine-in restaurants. This is due to convenience or reducing awkward interactions with wait staff. AutoGarcon is a product designed to allow these restaurants to interface with their customers. We provide restaurants with an app that customers can use to place orders directly to the cooks; cutting out the interaction between customers and waitstaff. We also provide functionality for customers to order through Amazon Alexa on a table at the restaurant. Our product has features for restaurants to fully customize the user experience including the look and feel of their restaurant on the app.

This report will focus on the database component of the product. AutoGarcon was made using MySQL for our database and uses AWS for storage and server hosting. Our team is also in charge of the API that interfaces with the database. Throughout this report, you will see our product described in detail. Section 2 gives an in-depth description of our end product. Section 3 provides an extensive list of our use cases and a system sequence diagram for one of our endpoints. Section 4 details the nonfunctional requirements of our product. Section 5 displays the implementation details of our product. Section 6 is our reflections on our journey through this software development. Section 7 lists all the tools and resources we used throughout development. Through four months of development, the product started from a basic relational structure and evolved into an interactive database used across three platforms.

Section 2: End Product Description

Database End Product

Our database will include a table composed of general restaurant information. This includes information such as name, address, logo, cuisine, and opening and closing times. This table will also have the customization options such as color scheme, font, and font color. These will be used to customize the appearance of a restaurant in our app.

In addition to the general restaurant information, our database will have a full table composed of every restaurant's menu. Each entry in this table will include the restaurant's name, an item ID, and the item name as well as any necessary information that is needed during a customer's visit (such as price, calories, and when the item is offered). The table includes both foods and drinks and uses a label to differentiate them further into specific categories like entree, appetizer, refillable drink, etc.

Another table contains information pertaining to customers. We store their login information as well as personal information so they can have a personalized account. We also store their favorite restaurants to create a user-friendly app that allows customers to access their favorite restaurants with ease. With each order, the customer ID is attached so customers may reorder past orders. A separate table contains login information of a restaurant's staff (managers and cooks). The manager's account will be able to change the menu to keep it properly updated and the cook's account will be able to access the orders to be able to complete them. In another table we will store each restaurant's Alexa information so an Alexa can properly communicate new orders to the cooks.

We will also be providing a service feature that will allow customers using the app or an Alexa to interface with the restaurant staff. This includes setting up a table to handle service requests such as requesting help at a table or asking for the bill.

Finally, we will have an orders table that is a master list of orders across all of our affiliated restaurants. Each order will have a qualification of "pending", "in-progress", "completed", or "cancelled" in order to properly interact with the kitchen side of the system. The "pending" qualification is solely used for the Alexas and will not be used for orders through the app. We will be keeping track of all the specific items and quantities for each order in a separate table.

Our database team is also in charge of the API attached to the database. This API interfaces with the database to prevent direct access to the database. The API performs various actions such as returning data, adding data, and removing data. The API also ensures security is maintained by salting and hashing passwords before adding them to the database. In addition to this, we are also implementing tokens to verify user authorization. The API is hosted in an Amazon ec2 server with a self-signed certificate in order to use HTTPS for additional security. The API will also handle checking for valid user input to ensure consistent input is entered into the database. We are also implementing javascript prepared statements in order to prevent SQL injection attacks.

Our end product also includes many documentation artifacts. In our documentation artifacts, we include guides for the process of recreating our API server and database. We also provide instructions to import the data from our repository into the actual database through the MySQL Workbench as well as how to export data from the database to a physical backup. A formal template to request adding additional endpoints is included for clearer communication between groups. We include documentation for the current live endpoints that run on our API with descriptions of how to call the endpoints and the expected responses from them.

Database Key Features

The main key feature of our team is our API as it serves as the connection between the restaurant staff, the customers, and the database. All requests for data are made through this API which then sends the necessary queries to the database to receive or modify data. The importance of connectivity is highlighted by the emphasis on security over the server. We are running an HTTPS server as well as implementing hashed passwords and web tokens for authorization.

Customizability is another key feature to our product which we enable through the types of data we store in the database. For restaurants, there is customization as they can change the look and feel of their restaurant's appearance on the app and are able to modify these details after initially setting them up. The restaurants are also in full control of their menu as they are able to add, delete, and modify any of their menu items. There are also customization features on the customer's end. Customers can save favorite restaurants or look back at past orders that they have made as these are details that are stored in our database.

Another key feature is real-time serviceability, which allows the restaurant staff to interface with the customers quickly. One aspect includes customers creating orders that are sent to the cooks to be made without the need for waitstaff. Another part of this is the ability for customers to call for assistance or request the bill through our product to notify the staff remotely without the customer needing to flag down a staff member.

Section 3: Functional Requirement

Use Cases

Registering a New Restaurant

Last Modified: May 14, 2020

Use Case Description:

A manager is setting up a restaurant for the first time.

Actor and actor description:

- The Manager
    - The Manager of a restaurant.

Pre-Conditions:

Before this use case can begin, the manager must have already registered with AutoGarcon and set up a manager account.

Triggers:

The manager needs to set up their restaurant so they can start using the AutoGarcon products.

Main Success Scenario:

1. The manager logs in.
2. The manager clicks on restaurant information.
3. The manager fills out all restaurant information that's necessary set up (i.e. restaurant name, address, phone number, email, opening time, closing time, cuisine type).
4. The manager hits submit and a request is sent to the API to create a new restaurant.
5. The manager receives confirmation that their restaurant is created.

Post-Conditions:

The new restaurant information is stored in the database. The manager can make changes and add menu items now.

Variation 3A: The manager does not input all necessary information and upon submit, it forces the manager to go back and complete the necessary fields.

1. The manager logs in.
2. The manager clicks on restaurant information.
3. The manager fills out all restaurant information that's necessary set up (i.e. restaurant name, address, phone number, email, opening time, closing time, cuisine type) except cuisine type.
4. The manager hits submit and a request is sent to the API to create a new restaurant.
5. An error occurs telling the manager to finish filling out necessary fields.
6. The manager puts in cuisine type and submits.
7. The manager receives confirmation that their restaurant is created.

Variation 3B: The manager wishes to add the menu as a CSV with the initial set up.

1. The manager logs in.
2. The manager clicks on restaurant information.
3. The manager fills out all restaurant information that's necessary set up (i.e. restaurant name, address, phone number, email, opening time, closing time, cuisine type). The manager also uploads a CSV with their menu with all necessary fields filled out.
4. The manager hits submit and a request is sent to the API to create a new restaurant.
5. The manager receives confirmation that their restaurant is created.

Registering a New User Account

Last Modified: May 16, 2020

Description:

A user, either a staff member or a customer is creating an account with AutoGarcon.

Actor and actor description:

- Customer in a restaurant
  - A customer who wants to use the AutoGarcon app.
- Staff Member
  - A staff member of restaurants who use AutoGarcon.

Pre-Conditions:

The API and database must be up and running.

Triggers:

A user wants to make use of all the features AutoGarcon can offer.

Main Success Scenario:

1. The user clicks to make a new account.
2. The user is prompted to enter in their information (username, password, name, and email for both, and position and restaurant id for staff members).
3. The user hits submit and a request is sent to the API to create the new user.
4. All input criteria are met so the request is completed.
5. The user receives confirmation of the creation of their account.

Post-Conditions:

The user can now use the AutoGarcon product.

Variation 4A: The API or database is unreachable.

1. The user clicks to make a new account.
2. The user is prompted to enter in their information (username, password, name, and email for both, and position and restaurant id for staff members).
3. The user hits submit and a request is sent to the API to create the new user.
4. The API or database is unreachable so the user receives an error and is told to contact AutoGarcon for help.

Variation 4B: The first username they would like is already used for an account.

1. The user clicks to make a new account.
2. The user is prompted to enter in their information (username, password, name, and email for both, and position and restaurant id for staff members).
3. The user hits submit and a request is sent to the API to create the new user.
4. All input criteria are met so the request is completed however the username is already in use and they must choose a new username.
5. They do so and hit submit which sends the request again.
6. The user receives confirmation of the creation of their account.

Logging into a Specific Account

Last Modified: May 15, 2020

Description:

This use case will define how a user's data will be inputted for the use of their AutoGarcon account.

Actor and actor description:

- Our user is either a customer, staff, or manager and is either a first-time user of the app or a user with an already existing account.
    - The customer wishes to use AutoGarcon because they feel slightly judged by the amount of food they want to order and do not want to interact with a server.
    - The manager wishes to use AutoGarcon to increase sales and reduce costs associated with real-life servers taking down orders.

Pre-Conditions:

A database exists and holds user information including email, username, and password.

Triggers:

- The user uses their Android phone to open the AutoGarcon app and begin login or first-time setup.
- The user uses the AutoGarcon website to login to their account or begin a first-time setup.

Main Success Scenario:

1. The user enters the username and their password and submits the login request.
2. The password is hashed and sent to the database. The hashed password cannot be accessed by anyone for security purposes.
3. The request is received by the server and is matched with a user stored in the database.
4. The server replies with the general information of the user as well as a token for authentication
5. The user has successfully logged in and can begin using the product.

Post-Conditions:

The database information for each user still exists and is able to take in new user data. Also, the database is able to store the changed user data if they want to change their password.

Variation 1A: User enters incorrect username

1. User types in incorrect information
2. Query with incorrect information is sent to the database
3. Database returns no matching user with the information
4. System displays error message and prompts for user information

Variation 1B: User enters incorrect password

1. The user enters the username and an incorrect password and submits the login request.
2. The password is hashed and sent to the database. The hashed password cannot be accessed by anyone for security purposes.
3. The request is received by the server and the hashed password is not matched with the user's stored hashed password in the database.
4. The database returns an error with an incorrect username/password tag.
5. The user enters the username and their password and submits the login request.
6. The password is hashed and sent to the database.
7. The request is received by the server and is matched with a user stored in the database.
8. The server replies with the general information of the user as well as a token for authentication
9. The user has successfully logged in and can begin using the product.

Variation 2A: The user tries to get around the username and password by using SQL injection. The database catches this however and prevents validation.

1. The user enters in the SQL injection and submits the login request.
2. The info is sent to the database.
3. The request is received by the server and is rejected due to security measures.
4. The server replies with the general information of the user as well as a token for authentication
5. The user is prompted to enter valid information.

Recovering a Forgotten Password

Last Modified: May 16, 2020

Description:

A user forgot their password and goes through the recovery process so they can access their account once more.

Actor and actor description:

- Customer of a restaurant
    - A who wants to use the AutoGarcon app.
- Staff

    - Staff of restaurants that need to use the AutoGarcon cook interface and restaurant data.

Pre-Conditions:

The user must already have an account with AutoGarcon.

Triggers:

The user forgot their password and cannot access their account.

Main Success Scenario:

1. The user (either staff member or customer) tries to log in and fails because they can't remember their password.
2. The user clicks on "Forgot Password?".
3. The user is prompted to enter in their username and click submit.
4. A request is sent to the API.
5. The API checks the username to see if it is in the database. It is so the API generates a temporary password, stores it in the database, sets temp_password to true (or 1), and sends the user an email with the temporary password to the email address associated with the username entered.
6. The API sends a request back to the user saying "email has been sent".
7. The user uses the temporary password to login.
8. Because temp_password is set to true, when the user logs in, they are prompted to change their password.
9. The user enters a new password.
10. A request is sent to the API to update the password and temp_password is set back to false (or 0).

Post-Conditions:

The user has updated their password and has full access to their account once more.

Variation 5A: The user enters a username not associated with an account. The user however is not informed that the username entered is not associated with an account due to security reasons.

1. The user (either staff member or customer) tries to log in and fails because they can't remember their password.
2. The user clicks on "Forgot Password?".
3. The user is prompted to enter in their username and click submit.
4. A request is sent to the API.
5. The API checks the username to see if it is in the database. It is not so the API doesn't change any data.
6. The API sends a request back to the user saying "email has been sent".
7. The user uses the temporary password to login.
8. Because temp_password is set to true, when the user logs in, they are prompted to change their password.
9. The user enters a new password.

10. A request is sent to the API to update the password and temp_password is set back to false (or 0).

Modifying Restaurant Information

Last Modified: May 15, 2020

Use Case Description:

This use case goes into how a manager would change/create customization details for their restaurant.

Actor and actor description:

- Manager
    - Manager of the restaurant using our app. The manager is the primary client as they are the ones we directly deal with.

Pre-Conditions:

Before the use case starts, we have already set up an existing database. In this database, we will need to add the restaurant the manager is trying to access. If it is a new restaurant to us then at the very least, a new id has been established.

Triggers:

- The manager is adding the restaurant details for the first time.
- The manager is changing the restaurant details because it is changing locations, setting up new contact information, or undergoing some form of rebranding that requires a different look and feel to their interface.

Main Success Scenario:

1. The manager uses their login information to login to their account.
2. Their password is sent through a hash-map in order for it to be encrypted.
3. Once successfully logged in, the manager goes to the page that displays the restaurant's information.
4. The manager makes the changes to the restaurant information as intended.
5. The manager hits a button to submit these changes.
6. The manager is verified as a proper manager and the changes are updated in the database.

Post-Conditions:

Once the manager commits the changes, the database needs to send the new and updated information through the API to the app, website, and the appropriate Alexa's.

Variation 4A: The manager inputs an incorrect input. Example: Puts in 9 for a phone number which is invalid.

1. The manager uses their login information to login to their account.
2. Their password is sent through a hash-map in order for it to be encrypted.
3. Once successfully logged in, the manager goes to the page that displays the restaurant's information.
4. The manager makes the changes to the restaurant information as intended.

5. The manager hits a button to submit these changes.
6. The manager is verified as a proper manager but updates are not updated in the database due to the error in input. An error is sent back to the manager to correctly fill out the information.

Modifying Menu Information

Last Modified: May 16, 2020

Use Case Description:

This goes through the manager using the web application to access and change their restaurant. The manager is the only one on the client side who has access to change the database. The only tables the manager can change however, are the menu tables.

Actor and actor description:

- Manager
  - Manager of the restaurant using our app. The manager is the primary client as they are the ones we directly deal with.

Pre-Conditions:

Before the use case starts, we have already set up an existing database. In this database, we will need to add the restaurant the manager is trying to access if it is a new restaurant to us and fill in any necessary information that only we are allowed to change and pertain to the restaurant in question. However, in this use case, the database is already set up.

Triggers:

The menu for the manager's restaurant has changed. Some price increases occurred as well as adding new dishes to the menu. The restaurant also ran out of certain ingredients and so the manager must update those item's availability.

Main Success Scenario:

1. The manager uses their login information to login to their account.
2. Their password is sent through a hash-map in order for it to be encrypted.
3. Once successfully logged in, the manager only has access to their own restaurant's menus.
4. The manager inputs the new menu item "Parmesan Chicken" by filling out the necessary fields of name, price, calories, price, any customer inputs, category (entrée in this case), and when it's available.
5. Then the manager updates an existing item as a price increase occurred for that item.
6. During their last service, the restaurant ran out of ribs to sell.
7. Finally, the manager changes the "In Stock?" field of the ribs entrée to "no" in order to prevent customers from even being able to order the entrée.
8. All necessary changes have been made so the manager logs out and returns to running their restaurant.

Post-Conditions:

Once the manager commits the changes, the database needs to send the new and updated information through the API to the app, website, and the appropriate Alexa's.

Variation 4A: The manager is adding an item that comes with a variety of sides and must choose which sides the new item can be served with.

1. The manager uses their login information to login to their account.
2. Their password is sent through a hash-map in order for it to be encrypted.
3. Once successfully logged in, the manager only has access to their own restaurant's menus.
4. The manager inputs the new menu item "Parmesan Chicken" by filling out the necessary fields of name, price, calories, price, any customer inputs, category (entrée in this case), and when it's available. The manager also chooses "Fries", "Rice", and "Mash Potatoes" as options to come with the "Parmesan Chicken".
5. Then the manager updates an existing item as a price increase occurred for that item.
6. During their last service, the restaurant ran out of ribs to sell.
7. Finally, the manager changes the "In Stock?" field of the ribs entrée to "no" in order to prevent customers from even being able to order the entrée.
8. All necessary changes have been made so the manager logs out and returns to running their restaurant.

Variation 4A: The manager must update allergens to prevent people from having allergic reactions and thus getting sued for not stating so.

1. The manager uses their login information to login to their account.
2. Their password is sent through a hash-map in order for it to be encrypted.
3. Once successfully logged in, the manager only has access to their own restaurant's menus.
4. The manager inputs the new menu item "Parmesan Chicken" by filling out the necessary fields of name, price, calories, price, any customer inputs, category (entrée in this case), when it's available,and any sides it comes with. The manager also adds "Gluten" from the drop-down menu of allergens to the allergens that "Parmesan Chicken" has.
5. Then the manager updates an existing item as a price increase occurred for that item.
6. During their last service, the restaurant ran out of ribs to sell.
7. Finally, the manager changes the "In Stock?" field of the ribs entrée to "no" in order to prevent customers from even being able to order the entrée.
8. All necessary changes have been made so the manager logs out and returns to running their restaurant.

Modifying User Information

Last Modified: May 16, 2020

Description:

The user is changing their personal information such as password, username, email, etc.

Actor and actor description:

- Customer in a restaurant
    - A who uses the AutoGarcon app.
- Staff
    - A staff member of a restaurant who uses AutoGarcon.

Pre-Conditions:

The user (either a customer or a staff member) must already have an account with AutoGarcon.

Triggers:

The user needs to update their personal information for their account.

Main Success Scenario:

1. The user logs in.
2. The user clicks on their account information and on edit information.
3. The user is prompted to enter in their password again to ensure they have access to the account they are changing.
4. The user wishes to change the email associated with their account so they enter in a new email address.
5. The user also wishes to update their password so they enter in a new password twice to ensure they did not misspell the password.
6. The user has updated everything they want and clicks submit.
7. A request is sent to the API with the updates the user entered.

Post-Conditions:

The user now has a new email and password associated with their account.

Variation 4A: The change the user wants to implement is the username.

1. The user logs in.
2. The user clicks on their account information and on edit information.
3. The user is prompted to enter in their password again to ensure they have access to the account they are changing.
4. The user wishes to change the username associated with their account so they enter in a new username.
5. The user has updated everything they want and clicks submit.
6. A request is sent to the API with the updates the user entered and the API ensures the username is not already in use, if it is, the user is prompted to enter in a new username.

Retrieving Restaurant Information

Last Modified: May 15, 2020

Use Case Description:

This use case is when a customer is accessing a restaurants section of the app and the app must provide them to proper information

Actor and actor description:

- User
    - The user of the app. Also is the customer at the restaurant in question.

Pre-Conditions:

Before the use case starts, we have already set up an existing database. In this database, the user has an account to access this information and the restaurant has the proper data inputted.

Triggers:

The user wants a restaurant's specific information with menu and location.

Main Success Scenario:

1. The user logs in (see user login case for more information on this).
2. The user wants to order from a restaurant.
3. User clicks the specific tab for the restaurant.
4. A request is sent to the API to retrieve a restaurant's info.
5. The input of a specific restaurant_id is used to query the database.
6. The results from the query are sent back to the app.

Post-Conditions:

The app displays the restaurant information in a user friendly way that is determined by the Android team.

Variation 1A: Retrieving restaurant information on an Alexa.

1. The user wakes up the Alexa.
2. The user wants to order from a restaurant.
3. User asks the Alexa to read the restaurant information.
4. A request is sent to the API to retrieve a restaurant's info.
5. The input of a specific restaurant_id is used to query the database.
6. The results from the query are sent back to the Alexa and are read out loud.

Retrieving Menu Information

Last Modified: May 16, 2020

Use Case Description:

This use case is when a customer looks at a restaurant's menu either through the app or through Alexa.

Actor and actor description:

- User
    - The user of the app. Also is the customer at the restaurant in question.

Pre-Conditions:

Before the use case starts, the customer already has an account and the restaurant menu has already been input.

Triggers:

The user wants to read the menu of a restaurant.

Main Success Scenario:

1. The user logs in (see user login case for more information on this).
2. The user clicks on a restaurant and their menu.
3. A request is sent to the API to retrieve the menu from the database where the restaurant_id matches the input id.
4. The database sends the menu as a JSON object back through the API to the customer.
5. The menu is displayed for the user to see.

Post-Conditions:

The menu is displayed for the customer.

Variations 1A: Accessing the menu on the Alexa.

1. The user wakes up the Alexa at the restaurant.
2. The user prompts the Alexa to read the restaurant's menu.
3. A request is sent to the API to retrieve the menu from the database where the restaurant_id matches the input id.
4. The database sends the menu as a JSON object back through the API to the customer.
5. The menu is read out by the Alexa for the user.

Retrieving Alexa Information

Last Modified: May 15, 2020

Use Case Description:

The Alexa needs to retrieve the Alexa greeting and the table number assigned to the Alexa upon skill start up in order to properly function.

Actor and actor description:

- The Alexa
  - The Alexa at a restaurant.

Pre-Conditions:

Before this use case can begin, the manager must have already set up a restaurant, menu, and registered the Alexa in question to be associated with their restaurant and a table.

Triggers:

A customer boots up the Alexa and wants to place an order.

Main Success Scenario:

1. The customer says “open AutoGarcon”.
2. A request is sent to the API to get the table number and greeting associated with the specific Alexa ID.
3. The Alexa uses the greeting to greet the customer.
4. When the customer tells the Alexa to add to an order, the Alexa uses the table number to input the order.

Post-Conditions:

The Alexa holds this information until it times out.

Variation 4A: The customer is adding to an existing pending order.

1. The customer says "open AutoGarcon".
2. A request is sent to the API to get the table number and greeting associated with the specific Alexa ID.
3. The Alexa uses the greeting to greet the customer.
4. When the customer tells the Alexa to add to an order, the Alexa uses the table number and their ID to see if there is an existing order, if there is the Alexa asks the customer if they'd like to continue the order.

Creating a New Order

Last Modified: May 15, 2020

Description:

Diners in the restaurant order food through either the app or Alexa and the cooks receive the orders in the kitchen.

Actor and actor description:

- Customer in a restaurant
  - A who wants to submit an order of food and/or drinks.
- Cooks
  - Cooks in the kitchen that will receive orders and prepare the food for diners.

Pre-Conditions:

The database and API are both working. If the diner is on the app, they must have an account, be signed in, have a restaurant selected, and have food in their cart before they can click submit. If the diner is using Alexa, they must have food in their cart before asking Alexa to submit the order.

Triggers:

The diner clicks the submit order button in the app or tells Alexa to submit their order.

Main Success Scenario:

1. When the diner clicks the submit order button or tells Alexa to submit the order, the app or Alexa will craft a POST request and send it to the API.
2. After this POST request is received, the web server will craft and send a query to the database to add the items ordered by the diner to a table in the database that keeps track of orders.
3. The web interface in the kitchen will send GET requests to the API at set intervals to check for orders and display orders that need to be prepared for the cooks to see.

Post-Conditions:

Order is received by the cooks and they start to prepare the food for the order.

Variation 2A: The API or database is unreachable.

1. When the diner clicks the submit order button or tells Alexa to submit the order, the app or Alexa will craft a POST request and send it to the API.
2. After this POST request is received, the web server will craft and send a query to the database to add the items ordered by the diner to a table in the database that keeps track of orders. However, the database is unreachable so the query does not run.
3. A message is sent back to the customer telling them the order was not placed and seek staff assistance.

Retrieving Orders for Cooks

Last Modified: May 14, 2020

Use Case Description:

The cooks need to retrieve orders in order to complete them and see which ones have been completed.

Actor and actor description:

- The Cook
  - The cook at a restaurant.

Pre-Conditions:

Before this use case can begin, the cook's restaurant must already be set up with AutoGarcon and have a menu. Also, customers have already ordered from either the app or the Alexa.

Triggers:

A cook needs to see all "In Progress" and complete orders to properly do their job at their restaurant.

Main Success Scenario:

1. The cook logs into the website to access the cook interface.
2. The cook clicks on the orders tab.
3. A request is sent to the API to get all "In Progress" orders and the API queries the database.
4. The orders are returned to the cook interface with all necessary details.
5. The cook clicks to mark an order that is finished as "complete".
6. A request is sent to the API to update the status of that order.
7. The cook clicks on the "completed" tab to see which orders have been sent out.
8. Another request is sent to the API to get all "Completed" orders by querying the database.
9. The orders are returned to the cook interface including the order that was just marked as complete.

Post-Conditions:

The order marked as complete is now in the "completed" tab as opposed to the "in progress" tab and the orders are up to date.

Variation 5A: Instead of marking the order as complete, the cook changes the order from complete to to in progress as the order was mistakenly marked as complete.

1. The cook logs into the website to access the cook interface.
2. The cook clicks on the orders tab.
3. A request is sent to the API to get all "complete" orders and the API queries the database.
4. The orders are returned to the cook interface with all necessary details.
5. The cook clicks to mark an order that is finished as "in progress".
6. A request is sent to the API to update the status of that order.
7. The cook clicks on the "completed" tab to see which orders have been sent out.
8. Another request is sent to the API to get all "in progress" orders by querying the database.
9. The orders are returned to the cook interface including the order that was just marked as complete.

Retrieving Pending Orders

Last Modified: May 16, 2020

Use Case Description:

An Alexa needs to retrieve the pending orders that are associated with the specific Alexa.

Actor and actor description:

- The Alexa
    - The Alexa at a restaurant.

Pre-Conditions:

Before this use case can begin, the manager must have already set up a restaurant, menu, and registered the Alexa in question to be associated with their restaurant and a table. Also, a customer has started but not finished an order on the Alexa.

Triggers:

A customer wishes to access the order they started but have not yet finished.

Main Success Scenario:

1. The customer wakes the Alexa up again.
2. The Alexa sends a request through the API to check for pending orders.
3. The API queries the database and checks to see if there is a pending order with the id of the Alexa.
4. If there is, the Alexa asks the customer if they would like to continue the order
5. The customer tells the Alexa to continue the order and finishes the order.

Post-Conditions:

The Alexa continues its standard procedure.

Variation 4A: There are no pending orders to be displayed. In this variation, step 5 is disregarded.

1. The customer wakes the Alexa up again.

2. The Alexa sends a request through the API to check for pending orders.
3. The API queries the database and checks to see if there is a pending order with the id of the Alexa.
4. If there is not a pending order, the Alexa waits for the customer to prompt an action.

Variation 5A: The customer does not wish to continue the order. Steps are added.

1. The customer wakes the Alexa up again.
2. The Alexa sends a request through the API to check for pending orders.
3. The API queries the database and checks to see if there is a pending order with the id of the Alexa.
4. If there is, the Alexa asks the customer if they would like to continue the order
5. The customer tells the Alexa they wish to start a new order.
6. The Alexa sends a request to the API to get rid of the pending order from the database.
7. The customer starts ordering with a blank order.

Retrieving Acceptable Fonts to Use

Last Modified: May 14, 2020

Use Case Description:

A manager is setting up their restaurant with AutoGarcon and needs to decide which font to use for their customized version of the app.

Actor and actor description:

- The Manager
    - The Manager of a restaurant.

Pre-Conditions:

The manager already has registered with AutoGarcon and made an account.

Triggers:

The manager wants to fully set up their restaurant and thus needs to choose the font their restaurant will display on the app.

Main Success Scenario:

1. The manager logs in.
2. The manager accesses the customization screen.
3. A request is sent to the API to retrieve the usable fonts.
4. A drop-down menu is created with those fonts.
5. The manager clicks the drop-down menu and chooses the font they want.
6. The manager enters in all remaining data and clicks submit.

Post-Conditions:

Upon submitting, a request is sent to the API to store the data entered into the "restaurants" table. And the info is saved.

Variation A: Every step is the same. These steps are also used if the trigger is the manager wishes to change the font instead of initially choosing a font.

Requesting Help on the App

Last Modified: May 13, 2020

Use Case Description:

A customer is dining at a restaurant and needs help from a staff member at the restaurant.

Actor and actor description:

- The Customer
  - The customer is dining at a restaurant.

Pre-Conditions:

A customer is eating at a restaurant who uses AutoGarcon. The customer has scanned the QR code.

Triggers:

The customer encountered a problem during their dining experience that they need a real person to help them with.

Main Success Scenario:

1. The customer places their order.
2. The customer needs a refill of the soft drink.
3. The customer presses the "help" button to get a staff member to refill their drink.
4. A request is sent to the API.
5. The cook interface continuously makes requests to get orders and receives the request for help.
6. A staff member clears the help request and goes to the table which sends a request to the API to set the customer's status to "good".
7. At the end of the meal, the customer needs the check to pay and leave so the customer requests "bill".
8. A request is sent to the API to set the service to "bill".
9. The cook interface requests to get the services needed.
10. A staff member sees the customer needs the bill so they print the bill out and sets the service back to "good".

Post-Conditions:

The customer pays and leaves the restaurant and the service status for that table is set at "good" again.

Variation 1A: The customer has not yet placed their order yet due to a question they would like to ask a staff member so they first request help and then place their order. Step 1 is now step 6.

1. The customer has a question the app cannot answer.
2. The customer presses the "help" button to get a staff member to help them.
3. A request is sent to the API.

4. The cook interface continuously makes requests to get orders and receives the request for help.
5. A staff member clears the help request and goes to the table which sends a request to the API to set the customer's status to "good".
6. The customer places their order.
7. At the end of the meal, the customer needs the check to pay and leave so the customer requests "bill".
8. A request is sent to the API to set the service to "bill".
9. The cook interface requests to get the services needed.
10. A staff member sees the customer needs the bill so they print the bill out and sets the service back to "good".

Requesting Help on an Alexa

Last Modified: May 13, 2020

Use Case Description:

A customer is dining at a restaurant and needs help from a staff member at the restaurant. The customer is using an Alexa for their experience at the restaurant.

Actor and actor description:

- The Customer
  - The customer is dining at a restaurant.

Pre-Conditions:

A customer is eating at a restaurant that uses AutoGarcon. The customer has started the Alexa up.

Triggers:

The customer encountered a problem during their dining experience that they need a real person to help them with.

Main Success Scenario:

1. The customer tells the Alexa to place their order.
2. The customer needs a refill of the soft drink.
3. The customer asks the Alexa for "help" in order to get a staff member to refill their drink.
4. A request is sent to the API.
5. The cook interface continuously makes requests to get orders and receives the request for help.
6. A staff member clears the help request and goes to the table which sends a request to the API to set the customer's status to "good".
7. At the end of the meal, the customer needs the check to pay and leave so the customer asks the Alexa for the bill.
8. A request is sent to the API to set the service to "bill".
9. The cook interface requests to get the services needed.

10. A staff member sees the customer needs the bill so they print the bill out and sets the service back to “good”.

Post-Conditions:

The customer pays and leaves the restaurant and the service status for that table is set at “good” again.

Variation 1A: The customer has not yet placed their order yet due to a question they would like to ask a staff member so they first request help and then place their order. Step 1 is now step 6.

1. The customer has a question the Alexa cannot answer.
2. The customer asks the Alexa for help to get a staff member to help them.
3. A request is sent to the API.
4. The cook interface continuously makes requests to get orders and receives the request for help.
5. A staff member clears the help request and goes to the table which sends a request to the API to set the customer’s status to “good”.
6. The customer asks the Alexa to place their order.
7. At the end of the meal, the customer needs the check to pay and leave so the customer asks the Alexa for the bill.
8. A request is sent to the API to set the service to “bill”.
9. The cook interface requests to get the services needed.
10. A staff member sees the customer needs the bill so they print the bill out and sets the service back to “good”.

Retrieving A User’s Favorite Restaurants

Last Modified: May 15, 2020

Use Case Description:

This use case is when a customer is accessing their favorite restaurants.

Actor and actor description:

- User
  - The user of the app. Also is the customer at the restaurant in question.

Pre-Conditions:

Before the use case starts, the customer has already added at least one restaurant to their favorites so it is not empty.

Triggers:

The user wants to access their favorite restaurants.

Main Success Scenario:

1. The user logs in (see user login case for more information on this).
2. The user clicks on their favorites in order to access a restaurant on the list.
3. A request is sent to the database through the API.

4. The API makes a query to access the customer's favorite restaurants using the customer_id input.
5. Favorite restaurants appear on the user's app.

Post-Conditions:

The results are sent back to the app so the customer can see which restaurants they have marked as their favorites.

Variation 5A: The user has no favorites, thus no favorites can be displayed.

1. The user logs in (see user login case for more information on this).
2. The user clicks on their favorites in order to access a restaurant on the list.
3. A request is sent to the database through the API.
4. The API makes a query to access the customer's favorite restaurants using the customer_id input.
5. The customer has no favorites so instead of displaying favorites, "No favorites" is displayed.

Deleting A Restaurant from a User's Favorites

Last Modified: May 13, 2020

Use Case Description:

A customer wishes to remove a restaurant from their favorites list.

Actor and actor description:

- A customer
  - The customer who uses the app.

Pre-Conditions:

The customer must already have an account with AutoGarcon and have favorite restaurants already set up.

Triggers:

A restaurant is no longer one of the customer's favorites after a bad experience so they would like to remove it from their favorites list.

Main Success Scenario:

1. The customer logs in.
2. The customer clicks on their favorite restaurants.
3. The customer clicks to delete a restaurant.
4. A request is sent to the API to delete the favorite with an input of the customer's ID and the restaurant ID of the restaurant the customer wants to remove.
5. The restaurant is successfully deleted and the customer sees "Favorite successfully deleted".

Post-Conditions:

The customer now does not see that specific restaurant in their favorites.

Variation 5A: The restaurant is successfully deleted but an error occurs due to the request being sent twice.

1. The customer logs in.
2. The customer clicks on their favorite restaurants.
3. The customer clicks to delete a restaurant.
4. A request is sent to the API to delete the favorite with an input of the customer's ID and the restaurant ID of the restaurant the customer wants to remove.
5. The restaurant was deleted but an error shows up due to the request being submitted multiple times.

Retrieving Order Statistics

Last Modified: May 12, 2020

Use Case Description:

A manager wants to know statistics at their restaurant.

Actor and actor description:

- Restaurant Manager

Pre-Conditions:

The restaurant has already been set up and operating with AutoGarcon.

Triggers:

The manager wants to know some statistics about what customers are ordering at their restaurant.

Main Success Scenario:

1. The manager logs in.
2. The manager clicks on the statistics page.
3. A request is sent to the API.
4. The API tells the database to run a stored procedure to retrieve the statistics of the restaurant.
5. The stored procedure queries the database to find statistics like the item at the restaurant that is ordered the most.
6. The API sends the statistics back to the website.

Post-Conditions:

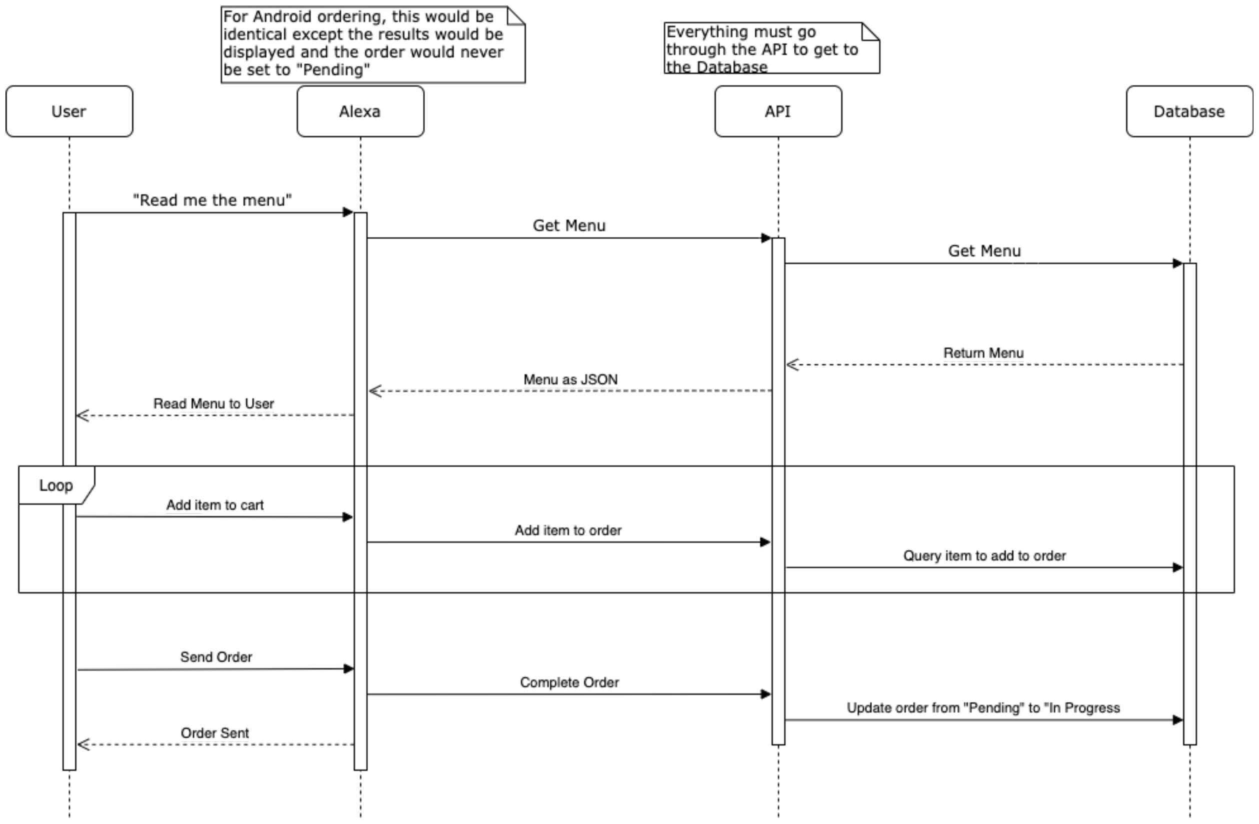
The manager takes the information to promote and do specials on certain items to make a bigger profit.

Variation 5A: The stored procedures cannot be used as there is no data to query due to no orders that have been placed yet.

1. The manager logs in.
2. The manager clicks on the statistics page.
3. A request is sent to the API.

4. The API tells the database to run a stored procedure to retrieve the statistics of the restaurant.
5. The API recognizes that no orders have been placed and thus there is no data to query over.
6. The API sends “No statistics yet” back to the website.

System Sequence Diagram:


For Android ordering, this would be identical except the results would be displayed and the order would never be set to "Pending"
Everything must go through the API to get to the Database
User
Alexa
API
Database
"Read me the menu"
Get Menu
Get Menu
Return Menu
Menu as JSON
Read Menu to User
Loop
Add item to cart
Add item to order
Query item to add to order
Send Order
Complete Order
Update order from "Pending" to "In Progress
Order Sent

This diagram depicts the “Creating a New Order” use case. It is specifically for ordering from an Alexa, however, the process is similar to ordering off the app. The user first prompts the Alexa to tell them the menu. Then the Alexa sends a request to the API with the request for a menu. Here, the API constructs a query to get the menu of the restaurant. The database executes the query and the results from the query are sent back to the API. Then the API sends the result to Alexa where the Alexa will read the results out loud. Now the user will add items to their order. The user tells the Alexa what item to add and the Alexa sends a request to the API. The API will make a query to the database to create an order (if this is the first item being ordered) or add an item to an order. This step will loop for all the items the user is ordering. Once the user is done ordering, they will tell Alexa to send the order where the order’s status will get updated from “Pending” to “In Progress” and Alexa will tell the user that the order is sent.

Section 4: Nonfunctional Requirements

Usability

Our database is used by customers, Amazon Alexas, as well as restaurant management and staff. We are working on the back end in order to facilitate the transfer of data between different users in real-time as well as to store personal information. We will handle storing the data from creating new restaurants, new users, and new orders.

Reliability

Our system is to be running at all times and we have backups stored on AWS that will allow us to revert to a previous version that did work even if it may not have all of the functionality of the version that crashed. However, no system failures should happen because of newly implemented functions as we test each on a test server before incorporating them with our actual server.

It is important to note that we are hosting our server on AWS through AWS Educate which gives account holders $100 worth of credit. Our current policy is that when our current running account has $10 remaining, we would go through our installation procedures on a different account and move all the data and code over to the new server. This would be accompanied by an email detailing the switch to the other teams in order to allow them to easily transition to sending requests to the new server.

Performance

We want the requests to our endpoints as well as the queries to the database to match industry standards in terms of response time. For the database queries, we look to MySQL's Query Response Time Index, which lays out acceptable response times. Based on these requirements, our database queries will respond within 100ms. On the API side, we expect to match the industry standards set by Google for back-end servers, which is to have all requests complete within 200ms.

Supportability

We will back up everything on an external hard drive in case of complete failure and data loss. AWS will also back up the database every week. Both ensure that the last working version of the database is held in case of failure. Also, before adding anything to the server, we will put it on a test server to ensure that it won't crash our working server. When the addition has been proven to work, we will then add it to the actual server. In terms of the tables in the database, we will maintain all tables in Boyce-Codd normal form (see algorithms section) in order to reduce redundant data. For security, we will be using the SHA-512 hash function to hash all user passwords using a salt with a length of 512 bits. This ensures that if passwords from the database were exposed, only the hashes of the passwords can be seen, not the actual passwords. When hashing passwords, we will perform 50,000 iterations of the hash function both before storing the passwords and when checking a password supplied for logging in. This, along with salting, is to increase the time that it takes for login attempts, reducing the effectiveness of brute force attacks. For the API, our web server will use encryption through TLS over HTTPS. This is to ensure end-to-end encryption of sensitive data such as passwords, preventing man-in-the-middle attacks. For authorization, we will be using JSON Web Tokens to ensure that certain endpoints are restricted to authorized users.

Section 5: Implementation

Style-Guide

SQL

1. Naming Conventions
    a. General:
        i. In general, every name should be unique and should not match with any of the reserved keywords in SQL.
        ii. Names should be limited to 30 characters.
        iii. Names must begin with a letter and can only contain letters and underscores.
        iv. Names should only have a single underscore if needed.
        v. Names should avoid using abbreviations and should be commonly understood.
    b. Tables
        i. When creating tables for our database, use a collective name such as staff or people, or the plural form of a noun.
        ii. For table names, it is acceptable to use a collection of two words with no underscore as the name. This improves clarity of the tables as well as prevents confusion with two worded attributes.
    c. Attributes
        i. The attributes of each table should be a singular noun that does not have the same name as a table.
        ii. Attribute names should be separated with an underscore if they are a collection of two words.
        iii. A collection of suffixes that have universal meaning that should be used if they are applicable:
            1. id—a unique identifier such as a column that is a primary key.
            2. status—flag value or some other status of any type such as publication status.
            3. total—the total or sum of a collection of values.
            4. num—denotes the field contains any kind of number.
            5. name—signifies a name such as firstname.
            6. seq—contains a contiguous sequence of values.
            7. date—denotes a column that contains the date of something.
            8. tally—a count.
            9. size—the size of something such as a file size or clothing.

10. addr—an address for the record could be physical or intangible such as ip_addr.

   d. Stored procedures
      i. Stored procedures are similar to functions and as such must include a verb and to briefly describe the results of the procedure
      ii. Stored procedures must be in Pascal Case where each word in the name of the procedure is upper case and there are no hyphens, spaces, underscores in between words.

2. Querying the Database
   a. Keywords
      i. All reserved keywords, such as SELECT and WHERE, should be in all upper case.
      ii. The query should be aligned to the left of a keyword as shown below

```
SELECT *
FROM table
WHERE table.table_id = 24;
```

      iii. Every keyword should appear on a separate line except for the AS keyword.
   b. Whitespace
      i. Spaces should be included between keywords, before and after the equals sign =, after commas and surround apostrophes that are not within parentheses.
      ii. New lines should be applied before any AND or OR clause, after a semicolon to separate queries, after each keyword definition, after a comma when separating multiple columns into logical groups.
   c. Nested Subqueries
      i. Nested subqueries follow a similar style to nested scope in standard programming where the nested query is indented in line with the start of the query
      ii. Parentheses are used to encompass the nested query and should be renamed with the AS keyword in order to specific clarity.

<u>JavaScript</u>

1. Variables
   a. String literals must be denoted with single quotes.
   b. Using multi-line string literals is advised for readability. When creating a multi-line string, it should be done using concatenation instead of escaping the new line. We will not be using the ES6 template literal to preserve backward compatibility.
2. Naming Conventions

        a. Variables and functions must use camelCase.
            i. Note that any variables storing values directly to and from attributes to the database should match the naming convention in the database for consistency.
        b. CONSTANTS must be all capitalized.
3. Conditionals
    a. All conditionals should be as simple as possible.
    b. All equality logical comparisons must use ===/!==.
    c. Ternary operators are not allowed.
4. Functions
    a. Avoid nested conditionals.
    b. The curly brace should appear on the same line as the function header.
    c. Each closing curly brace should appear on their own line. They should be indented to match the indentation of the function header.
5. Comments
    a. Comments should appear in the line above the code that they are explaining.
    b. There should be a comment for every function in the code.
6. Other
    a. Always use semicolons when appropriate.
    b. Scope must be defined with curly braces.

Install documentation

Creation of database and API server

1. To create our database, we start with these steps.
    a. Create a Virtual Private Cloud
    b. Create a web server with an EC2 instance
    c. Create a database instance using RDS
    d. Allocate an Elastic IP address and associate it with the EC2 instance
    e. The documentation for creating these three can be found in the following github folder: https://github.com/bricao16/AutoGarcon/tree/database/Database/Documentation/DatabaseCreationAWS
2. Extra requirements:
    a. Use PuTTY to connect to the EC2 instance from windows.
    b. Install Node on EC2
        i. Install nvm: curl -o- https://raw.githubusercontent.com/nvm-sh/nvm/v0.35.3/install.sh | bash
        ii. Activate nvm: ~/.nvm/nvm.sh

        iii. Install nodejs: nvm install node
        iv. Make sure node installed correctly: node -e "console.log('Running Node.js ' + process.version)"
    c. Create a folder for the server and install Server.js and package.json from the github listed below to this folder
        i. https://github.com/bricao16/AutoGarcon/tree/database/Database
    d. Install all dependencies from package.json using npm: npm install
    e. Generate a key and cert for HTTPS
        i. While in the server folder, enter this command: openssl req -x509 -newkey rsa:4096 -keyout key.pem -out cert.pem -days 365
            1. This will prompt you with several questions that you need to answer. Remember the passphrase you enter for the next step.
    f. Create a file named '.env' in the server folder and input the following information:

```
SSL_PASSPHRASE = <passphrase used in last step>
DB_HOST = <endpoint of RDS server>
DB_PORT = <port of RDS server>
DB_USER = <username for RDS user>
DB_PASS = <password for RDS user>
JWT_SECRET = <secret that will be used for signing JWTs>
```

    g. Use pm2 to start the server and ensure it stays running
        i. First, stop all running node processes by entering this command: killall -9 node
        ii. Next, install pm2 with this command: npm i -g pm2
        iii. Start the server using pm2 with this command pm2 start <path to server code>
            1. Example: pm2 start /home/ec2-user/server/Server.js
        iv. Make pm2 start on server restarts: pm2 startup
        v. Save pm2 settings: pm2 save
    h. Link to github file with the links to each tutorial from above: https://github.com/bricao16/AutoGarcon/blob/database/Database/Documentation/LinuxInstallation/LinuxLinksforInstallation15Apr.md

Installation and setup of MySQL Workbench

1. Install MySQL Workbench from this link: https://dev.mysql.com/downloads/workbench/
    a. Select the download button in the other downloads section
2. After the install finishes, open Workbench.
3. To add the connection to the database, click Database > Manage Connections

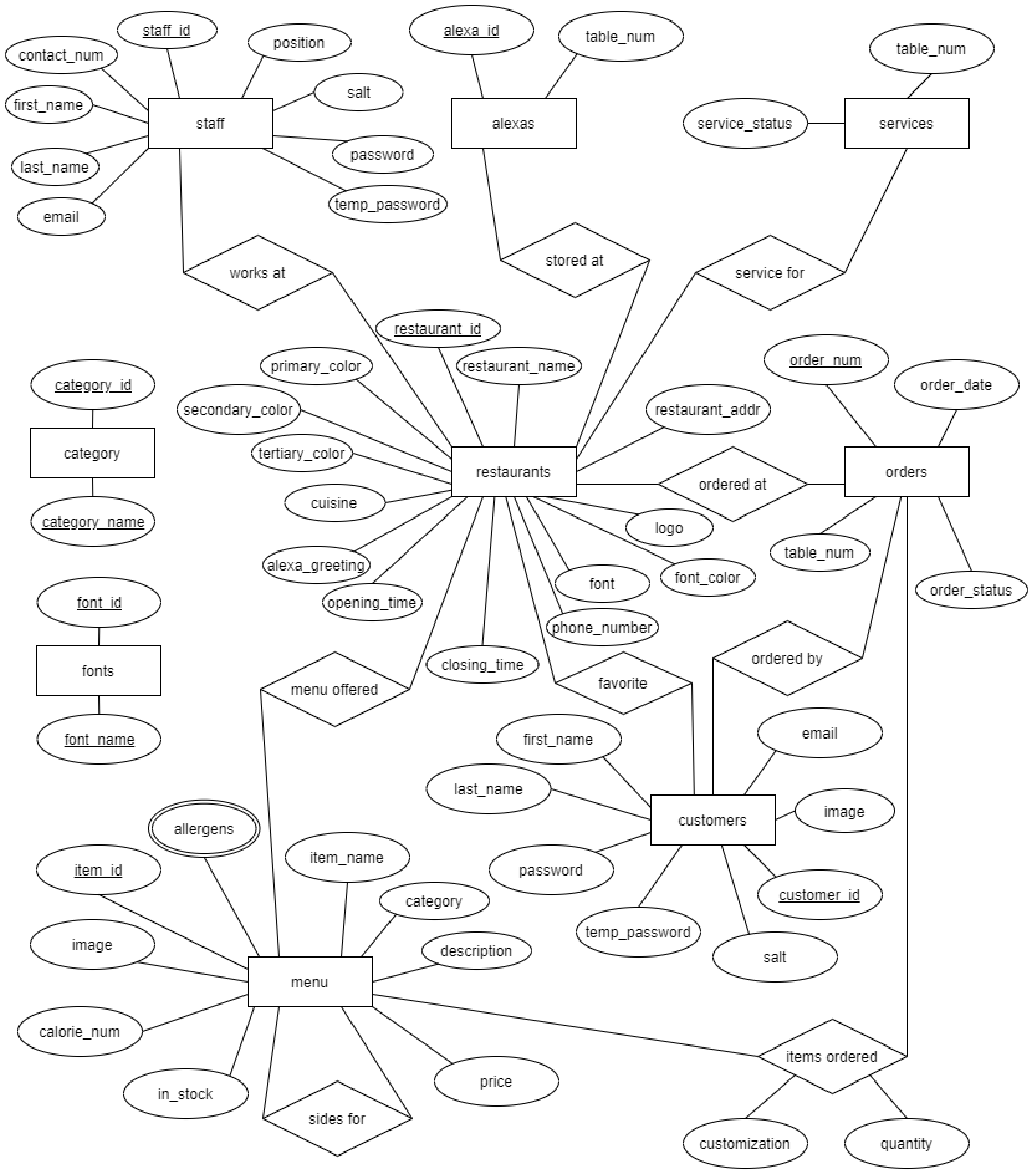
4. Add a new connection by clicking new in the bottom left corner.
5. Name the connection whatever you want, then change the Connection Method to Standard TCP/IP over SSH.
6. Enter the information as follows:
    a. SSH Hostname: <public DNS of EC2 server>
    b. SSH Username: <name of EC2 user>
    c. SSH Key File: <path to SSH key file created for EC2 server>
    d. MySQL Hostname: <endpoint of RDS server>
    e. MySQL Server Port: <port of RDS server>
    f. Username: <username for RDS user>
    g. Password: Store in Vault > Password: <password for RDS user>
7. Test Connection in the bottom right corner.
8. Close in the bottom right corner.
9. The connection is now saved and you can connect to the database from the homepage of Workbench.

Class Diagram

Database Schema

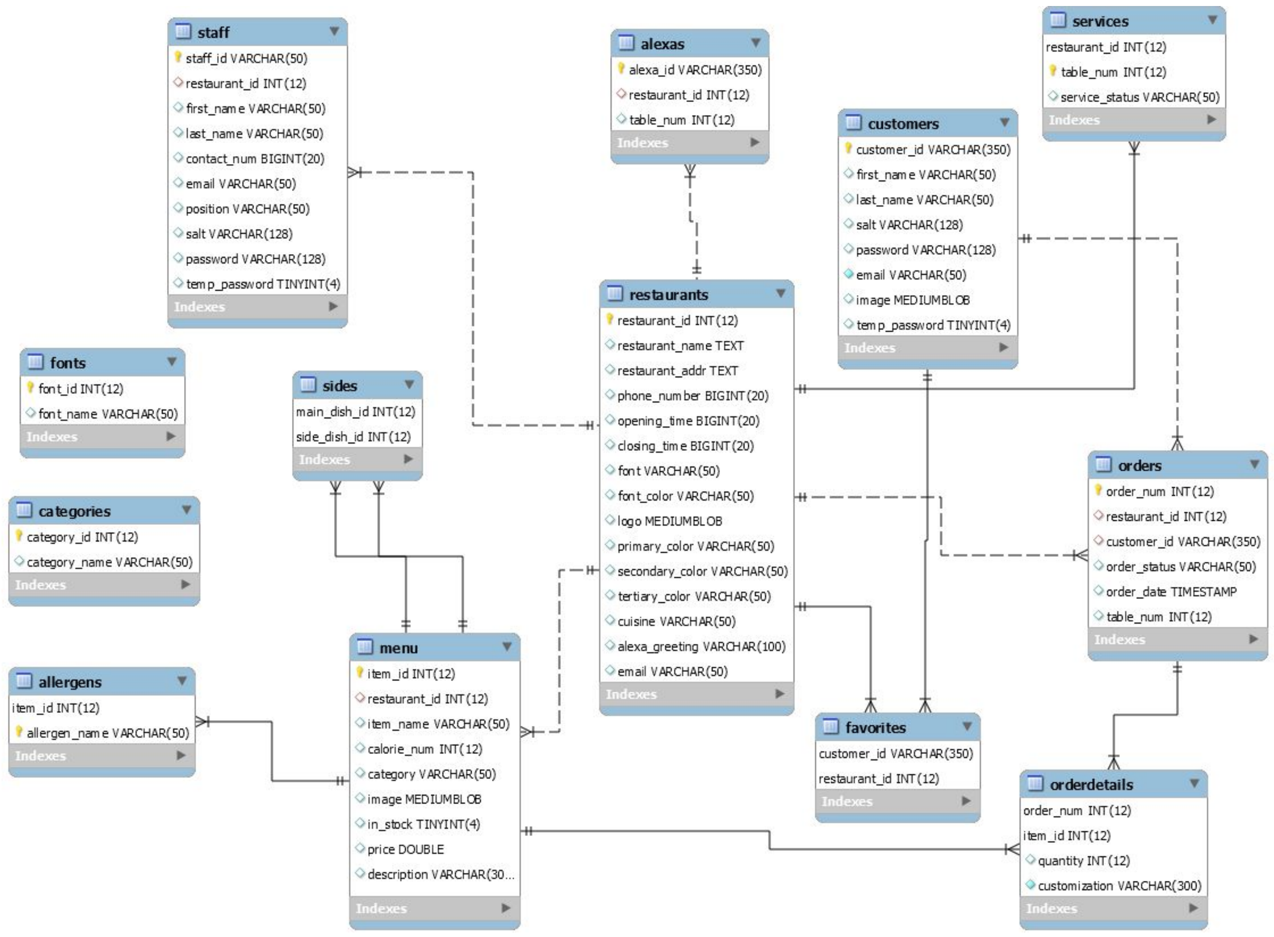
```
alexas
  alexa_id          KEY             VARCHAR(350)
  restaurant_id     FOREIGN KEY     INT
  table_num                         INT(120)

allergens
  item_id           FOREIGN KEY     INT
  allergen_name     KEY             VARCHAR(50)

categories
  category_id       KEY             INT
  category_name                     VARCHAR(50)

customers
  customer_id       KEY             VARCHAR(350)
  first_name                        VARCHAR(50)
  last_name                         VARCHAR(50)
  salt                              VARCHAR(128)
  password                          VARCHAR(128)
  email                             VARCHAR(50)
  image                                 MEDIUMBLOB
  temp_password                         TINYINT

favorites
  customer_id        FOREIGN KEY        VARCHAR(350)
  restaurant_id      FOREGIN KEY        INT

fonts
  font_id            KEY                INT
  font_name                             VARCHAR(50)

menu
  item_id            KEY                INT
  restaurant_id      FOREIGN KEY        INT
  item_name                             VARCHAR(50)
  calorie_num                           INT
  category                              VARCHAR(50)
  image                                 MEDIUMBLOB
  in_stock                              TINYINT
  price                                 DOUBLE
  description                           VARCHAR(300)

orderdetails
  order_num          FOREIGN KEY        INT
  item_id            FOREIGN KEY        INT
  quantity                              INT
  customization                         VARCHAR(300)

orders
  order_num          KEY                INT
  restaurant_id      FOREIGN KEY        INT
  customer_id        FOREIGN KEY        VARCHAR(350)
  order_status                          VARCHAR(50)
  order_date                            TIMESTAMP
  table_num                             INT

restaurants
  restaurant_id      KEY                INT
  restaurant_name                       TEXT
  restaurant_addr                        TEXT
  phone_number                           BIGINT
  opening_time                           BIGINT
  closing_time                           BIGINT
  font                                   VARCHAR(50)
  font_color                             VARCHAR(50)
  logo                                   MEDIUMBLOB
  primary_color                          VARCHAR(50)
  secondary_color                        VARCHAR(50)
  tertiary_color                         VARCHAR(50)
  cuisine                                VARCHAR(50)
  alexa_greeting                         VARCHAR(100)
  email                                  VARCHAR(50)

services
  restaurant_id     FOREIGN KEY          INT
  table_num         KEY                  INT
  service_status                         VARCHAR(50)

sides
  main_dish_id      FOREIGN KEY          INT
  side_dish_id      FOREIGN KEY          INT

staff
  staff_id          KEY                  VARCHAR(50)
  restaurant_id     FOREIGN KEY          INT
  first_name                             VARCHAR(50)
  last_name                              VARCHAR(50)
  contact_num                            BIGINT
  email                                  VARCHAR(50)
  position                               VARCHAR(50)
  salt                                   VARCHAR(128)
  password                               VARCHAR(128)
  temp_password                          TINYINT
```

ER Diagram


staff_id
contact_num
position
first_name
salt
staff
last_name
password
email
temp_password
works at
alexa_id
table_num
alexas
stored at
table_num
service_status
services
service for
restaurant_id
primary_color
restaurant_name
category_id
secondary_color
restaurant_addr
order_num
order_date
category
tertiary_color
restaurants
ordered at
orders
cuisine
logo
category_name
alexa_greeting
font
font_color
table_num
order_status
font_id
opening_time
phone_number
fonts
closing_time
ordered by
menu offered
favorite
font_name
first_name
email
last_name
customers
image
allergens
item_name
password
item_id
category
customer_id
temp_password
image
description
salt
menu
calorie_num
items ordered
in_stock
price
sides for
customization
quantity

Database Diagram

Created from reverse engineering the Database in MySQL Workbench.



Algorithms, Structures, & Patterns

Boyce-Codd Normal Form (BCNF)

1. Introduction: In relational database theory , a relation is said to be in Boyce-Codd Normal Form (BCNF) if all the determinants in the relation are keys. A set of relations is called a lossless decomposition of a given relation if the join of the relations gives back the original relation.
2. Algorithm: The algorithm for the decomposition of a relation into BCNF can be stated as follows.
    a. Step 1. From the given dependencies, write the dependency function.
    b. Step 2. Add to the dependency function the term ABC... H and get the horn function.
    c. Step 3. Search for a smallest term with a complemented literal.

d. Step 4. If Step 3 is successful, get the projection corresponding to the term.
e. Step 5. Delete the smallest term in Step 3 from the horn function along with all the terms containing the dependent attribute.
f. Step 6. Repeat Step 3 to 5 until the horn function reduces to a unate function.
g. Step 7. Form the projection corresponding to the literals in the unate function appearing in Step 6 and the literals that might have disappeared while going from Step 1 to Step 2.
h. The set of projections collected while carrying out the steps gives a lossless decomposition in which all the relations are in BCNF.

3. Implementation of BCNF: In the beginning we worked to ensure our database tables and columns within the table were adhering to BCNF. As time went on, it became more difficult to do this as requirements increased. As we added more tables, we ensured that redundant data and null values were minimized without redesigning the structure of the entire database.

Auto Increment

1. Introduction: Auto Increment is a function that operates on numeric data types. It automatically generates sequential numeric values every time that a record is inserted into a table for a field defined as auto increment. Implementation of auto increment requires normalization and unique identifiers which was touched on in the section on Boyce-Codd Normal Form. Auto increment can ensure that the primary key is always unique.
2. Implementation: For essential INT values in our database where many new entries are being created, we use the auto increment feature in MySQL to ensure the INT id values are unique.

Foreign Keys

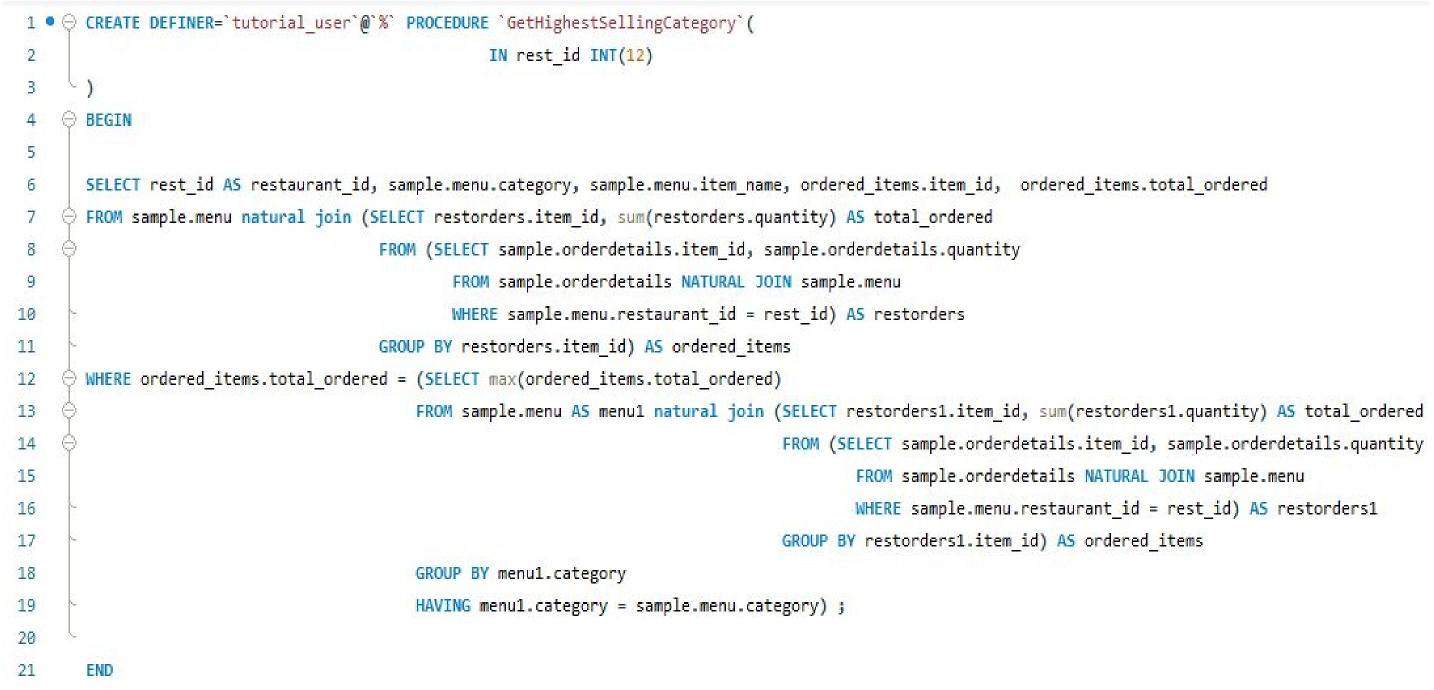
1. Introduction: A FOREIGN KEY is used to link two tables together. A FOREIGN KEY is a field (or collection of fields) in one table that refers to the PRIMARY KEY in another table. The table containing the foreign key is called the child table, and the table containing the candidate key is called the referenced or parent table.
2. Implementation: Most tables have links to other tables via foreign keys. There are only two tables, fonts and categories, that are not linked to any other tables because they are utilized for website and app dropdown menus.

Stored Procedures

1. Introduction: A stored procedure is a prepared SQL code that you can save, so the code can be reused over and over again. A stored procedure can be called in order to be executed. You can also pass parameters to a stored procedure and return values from the

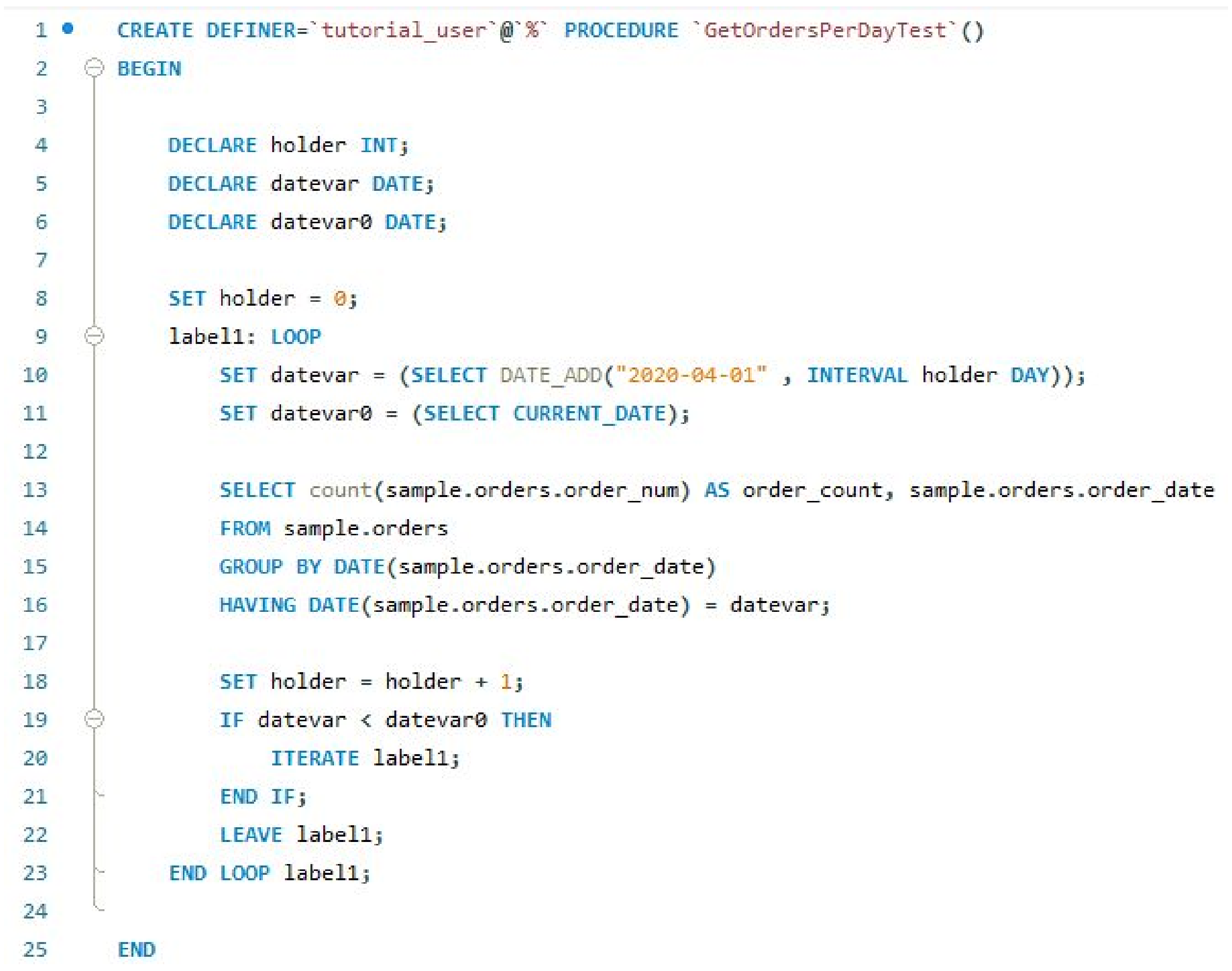
stored procedure. In essence, the stored procedure can act like a method would in some other coding languages.

2. Implementation: In our database, we use stored procedures where the queries would be too long or complex to include in the server code. We use stored procedures to get statistics for display on a restaurant's page. We use passed parameters and in some of our test stored procedures we use loops.
3. Images:
    a. Stored procedure to get highest selling item by category per restaurant



```
CREATE DEFINER=`tutorial_user`@`%` PROCEDURE `GetHighestSellingCategory`(
                                        IN rest_id INT(12)
)
BEGIN

SELECT rest_id AS restaurant_id, sample.menu.category, sample.menu.item_name, ordered_items.item_id,  ordered_items.total_ordered
FROM sample.menu natural join (SELECT restorders.item_id, sum(restorders.quantity) AS total_ordered
                                FROM (SELECT sample.orderdetails.item_id, sample.orderdetails.quantity
                                        FROM sample.orderdetails NATURAL JOIN sample.menu
                                        WHERE sample.menu.restaurant_id = rest_id) AS restorders
                                GROUP BY restorders.item_id) AS ordered_items
WHERE ordered_items.total_ordered = (SELECT max(ordered_items.total_ordered)
                                    FROM sample.menu AS menu1 natural join (SELECT restorders1.item_id, sum(restorders1.quantity) AS total_ordered
                                                                            FROM (SELECT sample.orderdetails.item_id, sample.orderdetails.quantity
                                                                                    FROM sample.orderdetails NATURAL JOIN sample.menu
                                                                                    WHERE sample.menu.restaurant_id = rest_id) AS restorders1
                                                                            GROUP BY restorders1.item_id) AS ordered_items
                                    GROUP BY menu1.category
                                    HAVING menu1.category = sample.menu.category) ;

END
```

    b. Test stored procedure to return number of orders every day even if nothing was ordered for certain days.



```sql
CREATE DEFINER=`tutorial_user`@`%` PROCEDURE `GetOrdersPerDayTest`()
BEGIN

    DECLARE holder INT;
    DECLARE datevar DATE;
    DECLARE datevar0 DATE;

    SET holder = 0;
    label1: LOOP
        SET datevar = (SELECT DATE_ADD("2020-04-01" , INTERVAL holder DAY));
        SET datevar0 = (SELECT CURRENT_DATE);

        SELECT count(sample.orders.order_num) AS order_count, sample.orders.order_date
        FROM sample.orders
        GROUP BY DATE(sample.orders.order_date)
        HAVING DATE(sample.orders.order_date) = datevar;

        SET holder = holder + 1;
        IF datevar < datevar0 THEN
            ITERATE label1;
        END IF;
        LEAVE label1;
    END LOOP label1;

END
```

Data Storage

The most important information we store is the menu for each restaurant and the restaurant information. All of the information is stored on separate tables that have keys to make each input unique. Examples of secondary information we store are orders, order details, Alexa information, staff, and customer information. This data makes it possible for orders to be placed and completed. It also makes it possible for the managers to make changes to their menu. Stored tertiary data is what services tables need (bill, help, or all good), when a menu item is offered (i.e. all day, dinner, lunch, etc.), and customer's favorite restaurants. All of this is stored on a MySQL database hosted on AWS.

Besides the data stored in the database, we also store sensitive information about the database and backups of the database. For sensitive information, we use a .env file on the EC2 server. This is a file that stores information such as passwords that are needed for connecting to the database. We are using this method because it provides a secure method for storing the information and it prevents data leakage in the server code. For the backups, we are using an external hard drive that we back up to every week or when there are significant changes made to the database. The physical backup stores all of the tables and data in our database so we can recover from a failure if need be. Additionally, AWS provides backups for the database and backs it up every seven days.

Testing and Verification

1. Server.js:
    a. After an endpoint is created, it is added to a second server which is used for testing inputs, outputs, and functionality. The endpoint is then manually tested using Postman, which is a tool used for crafting and sending requests to APIs. In order to verify that endpoints that alter data are working as intended, MySQL Workbench is used to view the data changes in the database. After verifying that the endpoint is working as intended, it is then added to the live server where other teams test it and notify our team of any bugs that are found.
2. MySQL
    a. Workbench: Most of the essential functions with the tables and database management were done through a user interface. Altering the table, setting the keys, data insertion, setting conditions for the columns, and column renames were done through the user interface. Workbench will then take your input, convert it into SQL, error check, and then allow application to the database if there are no errors.
    b. SQL Code:
        i. General code: A list of queries that output all the data for each table is run every time the database is opened to ensure the table data is all correct. Each of the queries was tested to ensure they generated all the data from each table. Table creation SQL was used to create the availability, main_menu, orders, orderdetails, services, and users tables but all other tables were created with the table import wizard available in Workbench.
        ii. Server SQL: The more advanced queries that would be put directly into the endpoints were tested in a generic tab in MySQL workbench before insertion into the server code for use in endpoints.
        iii. Stored procedures: The queries used in the stored procedures were tested in a generic SQL tab to ensure they were done correctly. Once ensured that the correct outputs were given, the MySQL Workbench Stored Procedure creation program was used to fill in the SQL code needed to generate a new Stored Procedure. The SQL code that was tested in a generic tab was then inserted into the Stored Procedure and tested in the generic tab as well. Once the code was ensured to be correct, it was inserted into the server code for use in endpoints.

Section 6: Reflection

Challenges

One challenge that we came across was communicating with the other teams. In the beginning, each of the teams were creating isolated designs for their portion of the project and we did not communicate with one another on a unified design. It was difficult for our team to assume what the other teams needed. To combat these struggles, we were very transparent with our database design as well as provided an interface of how we envision the API endpoints to look like so the other teams can begin development around these endpoints. We also talked with the other teams face-to-face and walked through their use-cases in order to figure out the logic behind the features.

Another aspect of this communication was the leadership of all the teams. When it came to getting everyone on the same page, it was difficult to get consensus or participation at times. Some managers were better than others when it came to communicating with other teams. Whether it was meeting presentations, communicating needs, delegating assignments, or just getting responses, communication was more difficult when all the teams had to work together for company presentations or demonstrations. Having the manager emails was a good tool but using something more instantaneous and holding the managers more accountable would have been helpful.

Security has been another difficulty that we faced. In the beginning, we had set up little in terms of user privacy or authentication for accessing the database. To overcome these challenges, we implemented salting and hashing passwords as well as implemented Javascript Web Tokens over all of the endpoints on our database to prevent those without proper authentication from modifying sensitive portions of the database.

Another aspect of security was HTTP vs HTTPS. In order to use HTTPS, we had to get a certificate in order to verify the transfer of data. Because it costs money to have a vendor actually sign our certificate, we had to self-sign our own certificates in order to use HTTPS. While this is more secure than HTTP, it caused issues for browsers and the other teams in accepting the self-signed certificate. We were able to have all our endpoints and server code working in HTTPS but the other teams were not able to get their programs to fully work with the HTTPS self-signed certificates.

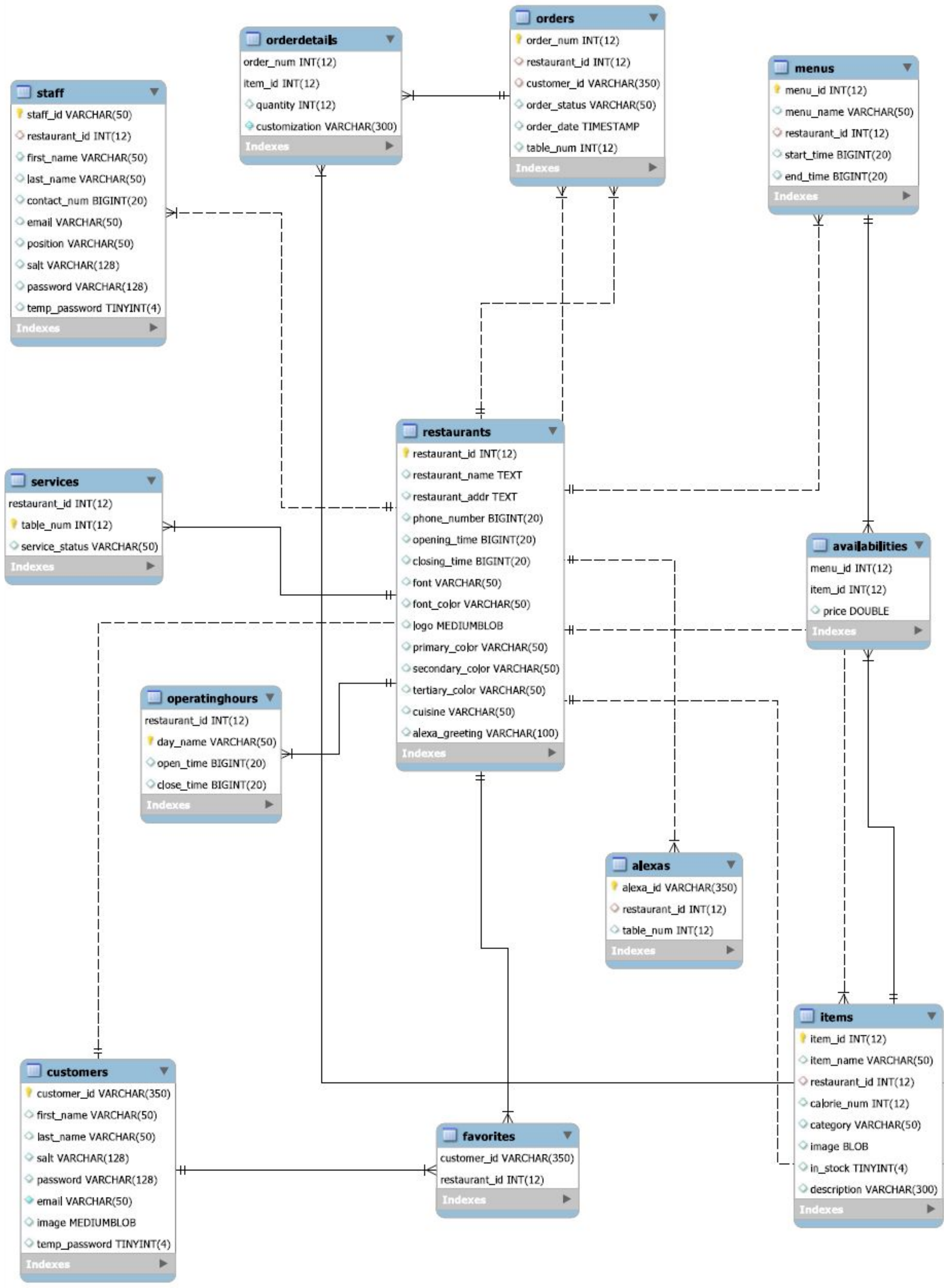
Done Differently

As the semester winded down, we realized that our design was flawed as it limited the ability to add more features. It also does not completely reflect the way restaurants would structure their data. If we could do things differently, we would have designed a different design by creating a new entity called menus which would link restaurants to their items, which is provided below. Having this menus entity allows for restaurants to build standard menus such as breakfast, lunch, and dinner. This would also allow for unique menus such as happy hour and specials. Implementing data this way also allows for items to be connected to different prices through the menu relation in addition to being able to control the availability of all the items through the use of the menus.

We would also like to change the way we handle users. Instead of keeping all the user information in our database, we could outsource the user login process to be handled through google by setting things up using Google accounts. There is also the challenge posed by how we treat each of the Alexas when it comes to ordering from them and how we should store that information in the database.

Another thing we could have implemented was payment. It would be difficult to set up our own secure way of doing this but if we used Google accounts as stated above, there is a functionality where one can pay with the security of PayPal or some other third party trusted vendor.

Communication is another area that we would like to have done differently. Many times other teams would come to the manager with a problem with one of our API endpoints. There would be some back and forth between the manager about the details of the issue until the problem is resolved. The issue with this system is that the back and forth is time consuming and can be completely avoided if the issue is clearly laid out in the beginning. It would have been better if there was a template request for other teams to use to make things run more efficiently.


orderdetails
order_num INT(12)
item_id INT(12)
quantity INT(12)
customization VARCHAR(300)
Indexes
orders
order_num INT(12)
restaurant_id INT(12)
customer_id VARCHAR(350)
order_status VARCHAR(50)
order_date TIMESTAMP
table_num INT(12)
Indexes
menus
menu_id INT(12)
menu_name VARCHAR(50)
restaurant_id INT(12)
start_time BIGINT(20)
end_time BIGINT(20)
Indexes
staff
staff_id VARCHAR(50)
restaurant_id INT(12)
first_name VARCHAR(50)
last_name VARCHAR(50)
contact_num BIGINT(20)
email VARCHAR(50)
position VARCHAR(50)
salt VARCHAR(128)
password VARCHAR(128)
temp_password TINYINT(4)
Indexes
restaurants
restaurant_id INT(12)
restaurant_name TEXT
restaurant_addr TEXT
phone_number BIGINT(20)
opening_time BIGINT(20)
closing_time BIGINT(20)
font VARCHAR(50)
font_color VARCHAR(50)
logo MEDIUMBLOB
primary_color VARCHAR(50)
secondary_color VARCHAR(50)
tertiary_color VARCHAR(50)
cuisine VARCHAR(50)
alexa_greeting VARCHAR(100)
Indexes
services
restaurant_id INT(12)
table_num INT(12)
service_status VARCHAR(50)
Indexes
availabilities
menu_id INT(12)
item_id INT(12)
price DOUBLE
Indexes
operatinghours
restaurant_id INT(12)
day_name VARCHAR(50)
open_time BIGINT(20)
close_time BIGINT(20)
Indexes
alexas
alexa_id VARCHAR(350)
restaurant_id INT(12)
table_num INT(12)
Indexes
items
item_id INT(12)
item_name VARCHAR(50)
restaurant_id INT(12)
calorie_num INT(12)
category VARCHAR(50)
image BLOB
in_stock TINYINT(4)
description VARCHAR(300)
Indexes
customers
customer_id VARCHAR(350)
first_name VARCHAR(50)
last_name VARCHAR(50)
salt VARCHAR(128)
password VARCHAR(128)
email VARCHAR(50)
image MEDIUMBLOB
temp_password TINYINT(4)
Indexes
favorites
customer_id VARCHAR(350)
restaurant_id INT(12)
Indexes

If Production Continued

If we continue production with our current database design, we would like to include more functionality regarding our menu items. Then, we would like to complete our set of endpoints for sides, so that we can add new sides to this table in the database. We would also like to implement a specialties table that allows for restaurants to have the ability to have a set of items that they would consider their specialties and incorporate different prices for these items. We would like endpoints to handle different sizes of an item, such as different amounts of side dishes like fries or maybe drinks.

We would also like to continue creating more queries to generate more statistics over our database. One specific statistic we would like to implement would be calculating the taxes for each restaurant based on the orders of our restaurant. This would require additional attributes set up for knowing tax rates for specific items and having those rates stored in our database. We would also like to create statistics for differentiating between our app customers and our customers ordering through Alexa.

Another aspect of our product we would like to improve would be security. We would also like to be able to handle user login failures on the server side, where if a user signs in too many times, we would lock them out of signing in similar to other services. We would also like to match all of the parameter checking that is done client side to be reflected on the server side.

Performance is another aspect of our design that we would like to explore deeper. We would like to bulk up the data in our database to reflect the product actually running. From here we could set up an index for our database to keep query times to acceptable levels. We would also like to look into sending timeout responses on the server for the times where our server does not respond in time.

Section 7: References

Tools

1. **Github:** Hosting source code and documenting details such as schemas, API endpoints, and important SQL queries. Link: https://github.com/bricao16/AutoGarcon
2. **MySQL Workbench**: GUI for viewing and editing the database. Link: https://www.mysql.com/products/workbench/
3. **AWS EC2**: Cloud computing services through Amazon Web Services that we are using to host our REST API on. Link: https://aws.amazon.com/ec2/
4. **AWS Educate**: Free Amazon Web Services account with $100 dollars of credit to use the resources offered. Link: https://aws.amazon.com/education/awseducate/
5. **AWS RDS**: Cloud computing services through Amazon Web Services that we are using to host our MySQL database on. Link: https://aws.amazon.com/rds/
6. **External Hard Drive**: Used for storing offline backups of the database.
7. **PuTTY**: SSH client used to connect to the EC2 machine that is hosting our REST API. Link: https://www.chiark.greenend.org.uk/~sgtatha m/putty/
8. **WinSCP**: SFTP client for Microsoft that is used to access Linux files in PuTTY. Link: https://winscp.net/eng/index.php
9. **Node.js**: Javascript runtime environment we are using to build our REST API. Link: https://nodejs.org/en/
10. **npm**: Package manager for Node.js that we are using to manage the modules we are using for development of our REST API. Link: https://www.npmjs.com/
11. **Microsoft OneNote**: Collaborative notebook used for taking notes during our meeting times. Link: https://products.office.com/en-us/onenote/digi tal-note-taking-app?rtc=1
12. **Microsoft Teams**: Platform used for remote communication between groups. Link: https://products.office.com/en-us/microsoft-te ams/group-chat-software
13. **Google Sheets**: Collaborative spreadsheet program that we are using to draft tables for the database. Link: https://www.google.com/sheets/about/
14. **Google Docs**: Collaborative text editor program that we use to modify shared documents such as this report and other plans. Link: https://www.google.com/docs/about/
15. **Gmail**: Email software used to communicate between different teams. Link: https://www.google.com/gmail/about/#
16. **Draw.io**: Online diagram software that we are using to create the ER diagram for the database design. Link: https://www.draw.io/
17. **Discord**: Online video chatting software that we are using for team meetings and chat function is used to upload documents and share information throughout. Link: https://discordapp.com/

18. **Zoom**: Online video chatting software that is used for company meetings and weekly team meetings. Link: https://www.zoom.us/
19. **Postman**: Collaboration platform for API development. Link: https://www.postman.com/api-platform/?utm_source=www&utm_medium=home_hero&utm_campaign=button

Sources

Amazon Web Services. (2020). *Setting Up with Amazon EC2.* Retrieved from *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/get-set-up-for-amazon-ec2.html#create-a-base-security-group*

Amazon Web Services. (2020). *Creating a Virtual Private Cloud (VPC).* Retrieved from *https://docs.aws.amazon.com/AmazonElastiCache/latest/mem-ug/VPCs.CreatingVPC.html*

Amazon Web Services. (2020). *Elastic IP address.* Retrieved from *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/elastic-ip-addresses-eip.html*

Amazon Web Services. (2020). *Create and Connect to a MySQL Database.* Retrieved from *https://aws.amazon.com/getting-started/hands-on/create-mysql-db/*

Amazon Web Services. (2020). *Connecting to Your Linux Instance from Windows Using PuTTY.* Retrieved from *https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/putty.html#putty-private-key*

Arnold, Jason. (2017, Apr 11). *Using the dotenv package to create environment variables*. Retrieved from https://medium.com/@thejasonfile/using-dotenv-package-to-create-environment-variables-33da4ac4ea8f

Bell, D. (2003). *Uml basics: An introduction to the Unified ModelingLanguage*. IBM.

Bell, D. (2004). *Uml basics: The sequence diagram*. IBM.

Heumann, J. (2008). *Tips for writing good use cases.* IBM.

Holywell, S. n.d. *SQL Style Guide.*

MySQL Enterprise Monitor 8.0 Manual :: 31.2 Query Response Time index (QRTi). (n.d.). Retrieved May 16, 2020, from https://dev.mysql.com/doc/mysql-monitor/8.0/en/mem-features-qrti.html

PSPlus. (2017, October 3). *Think you've got your requirements defined? Think FURPS!* Retrieved from *http://www.psplus.ca/articles/think-youve-got-your-requirements-defined-think-furps/*

Samad, Abdus. (2019, Jan 15). *Set-up SSL in NodeJS and Express using OpenSSL.* Retrieved from https://hackernoon.com/set-up-ssl-in-nodejs-and-express-using-openssl-f2529eab5bb

Simic, S. (2020, April 30). 7 Ways to Reduce Server Response Time {Improve Your Server Speed}. Retrieved May 16, 2020, from https://phoenixnap.com/kb/reduce-server-response-time

Tod, Robert. (2017, Jan 11). *Tutorial: Creating and managing a Node.js server on AWS, part 2*. Retrieved from https://medium.com/hackernoon/tutorial-creating-and-managing-a-node-js-server-on-aws-part-2-5fbdea95f8a1

Nambiar. Gopinath, Nagaraj, and Manjunath (1997) *Boyce-Codd Normal Form Decomposition* Retrieved from https://core.ac.uk/download/pdf/82674584.pdf

Guru99 (2020) *MySQL AUTO_INCREMENT with Examples* Retrieved from https://www.guru99.com/auto-increment.html

W3schools.com (2020) *SQL FOREIGN KEY Constraint* Retrieved from https://www.w3schools.com/sql/sql_foreignkey.asp

W3schools.com (2020) *SQL Stored Procedures for SQL Server* Retrieved from https://www.w3schools.com/sql/sql_stored_procedures.asp