Low Level Design

Team No Stress

12/10/2021

Daniel Ramirez
Hunter Lewis
Luke Sunaoka
Matthew Quinn (Team Leader)
Michelle Vo
Tyler Thorin

Introduction

1.1 Purpose:

Provide consumers with easy access to understandable nutrition facts about the products they consume. Help consumers gain back transparency and allow for educated choices in an industry that tries so hard to deceive consumers.

The system is designed for the average citizen living in the contiguous United States of America. The food industry has made it too difficult for consumers to understand what is in their food and to be able to make educated choices. This application will increase transparency and help consumers regain control over their products.

1.2 Scope of document

Documentation about the component level of the various modules that will be implemented and maintained throughout the development life cycle. The document is segmented based on the module. Each section on a given module covers the requirements of that module, implementation details, and how that module interacts with other modules.

1.3 System overview

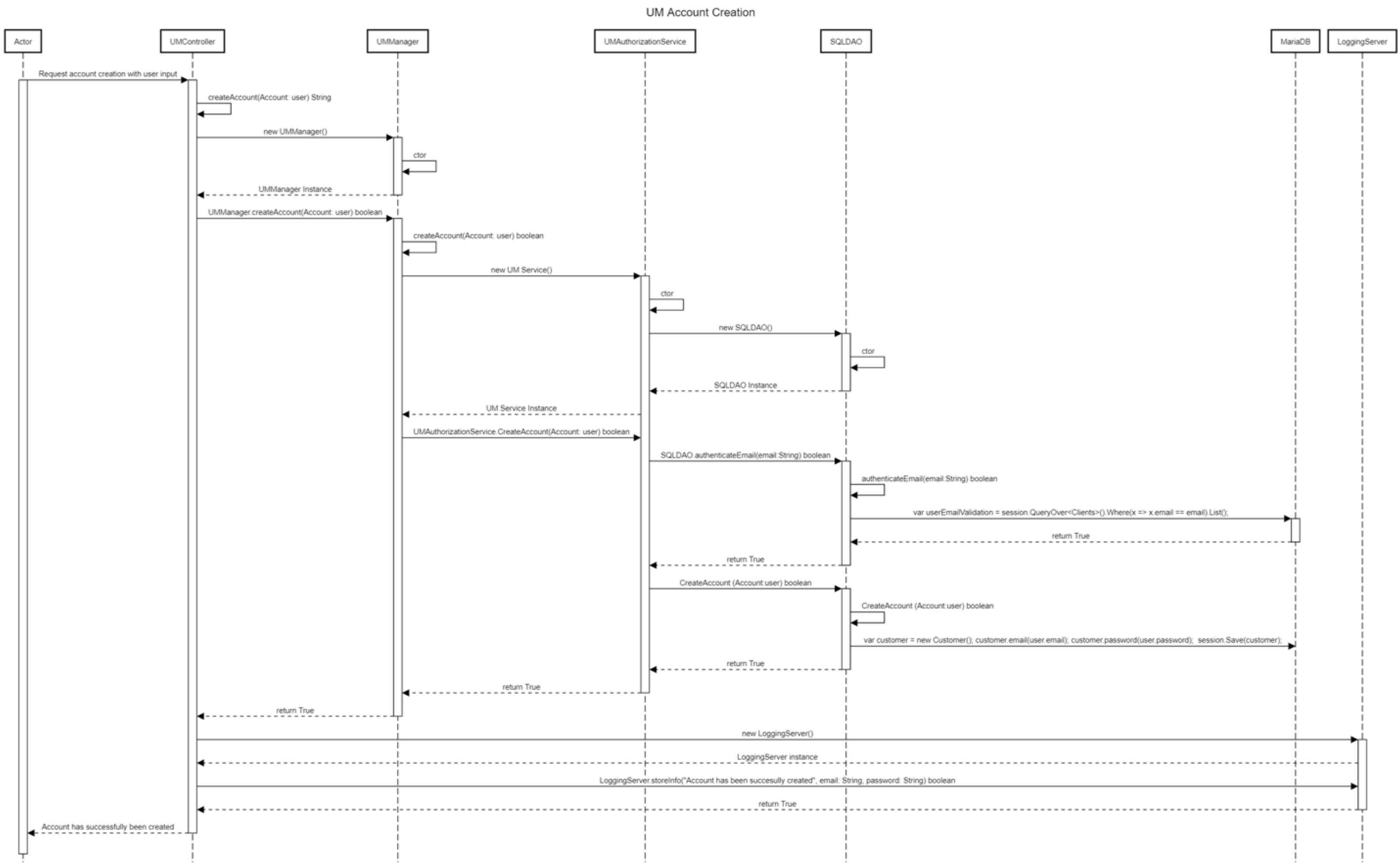
This application is being designed as a PWA(Progressive Web Application) with support for Google Chrome version 94.0.4606.61 for android/windows devices and 94.0.4606.52 for IOS devices. Having this application as a PWA will allow it to be more accessible and easily maintainable compared to a mobile application. The application will take products users scan, decode the barcode, send that information to our database, the database will return a food ID related to a product which will then fetch all necessary information on a product. This will integrate other resources of the application such as personalized daily percentages in order to provide the best possible user experience.

Low Level System Design

2.1 Sequence Diagram

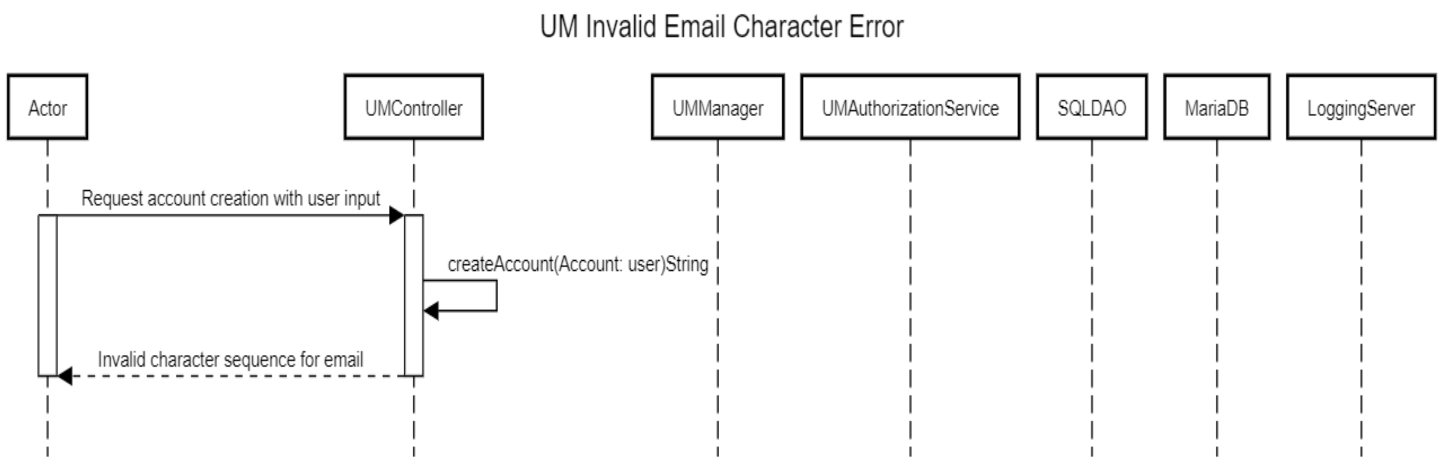
- Diagram capturing flow of information between all application layers

2.1.1 UM Account Creation


UM Account Creation
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account creation with user input
createAccount(Account: user) String
new UMManager()
ctor
UMManager Instance
UMManager.createAccount(Account: user) boolean
createAccount(Account: user) boolean
new UM Service()
ctor
new SQLDAO()
ctor
SQLDAO Instance
UM Service Instance
UMAuthorizationService.CreateAccount(Account: user) boolean
SQLDAO.authenticateEmail(email:String) boolean
authenticateEmail(email:String) boolean
var userEmailValidation = session.QueryOver<Clients>().Where(x => x.email == email).List();
return True
return True
CreateAccount (Account:user) boolean
CreateAccount (Account:user) boolean
var customer = new Customer(); customer.email(user.email); customer.password(user.password); session.Save(customer);
return True
return True
return True
new LoggingServer()
LoggingServer instance
LoggingServer.storeInfo("Account has been succesully created", email: String, password: String) boolean
return True
Account has successfully been created

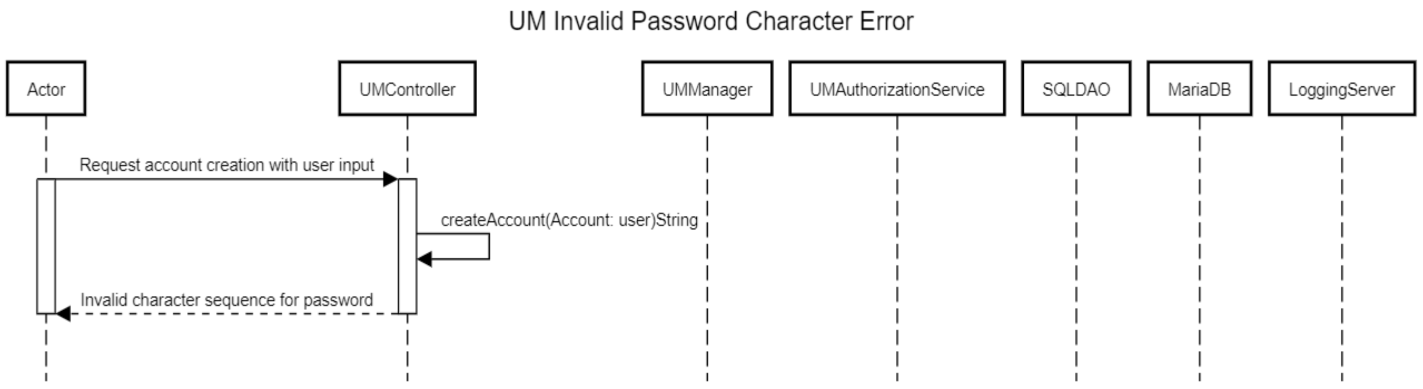
UM Account creation illustrates the process of a user creating an account for What the Food! The user will provide an email address and a password where this request will be passed onto the UMManager to the UMAuthorizationService to the SQLDAO. The SQLDAO will then store the newly made information in the User server. Upon creation of a successful account from the user server, the SQLDAO will notify the UMAuthorization Service in which the UMAuthorizationService will notify the UMManager. From there the UMManager will notify the UMController in which the UMController will request the Logging server to record this event. Upon successful logging, the UMController will notify the user that they have successfully created an account.

2.1.2 UM Invalid Email Character Error


UM Invalid Email Character Error
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account creation with user input
createAccount(Account: user)String
Invalid character sequence for email

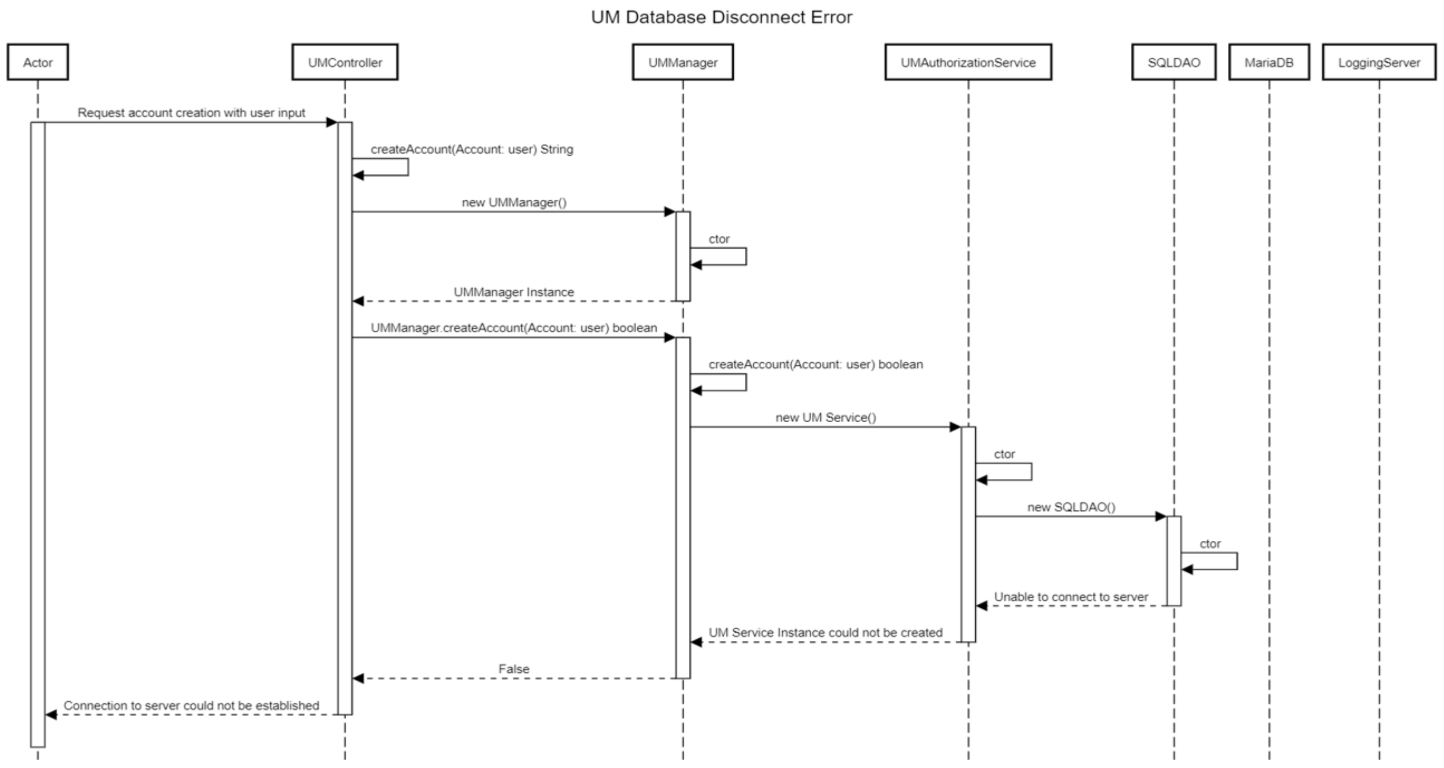
UM Invalid Username Character Error illustrates the case in which a user inputs a invalid sequence of characters when creating their account. The user will be notified that they have inserted an invalid character sequence and prompted to start over.

2.1.3 UM Invalid Password Character Error


UM Invalid Password Character Error
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account creation with user input
createAccount(Account: user)String
Invalid character sequence for password

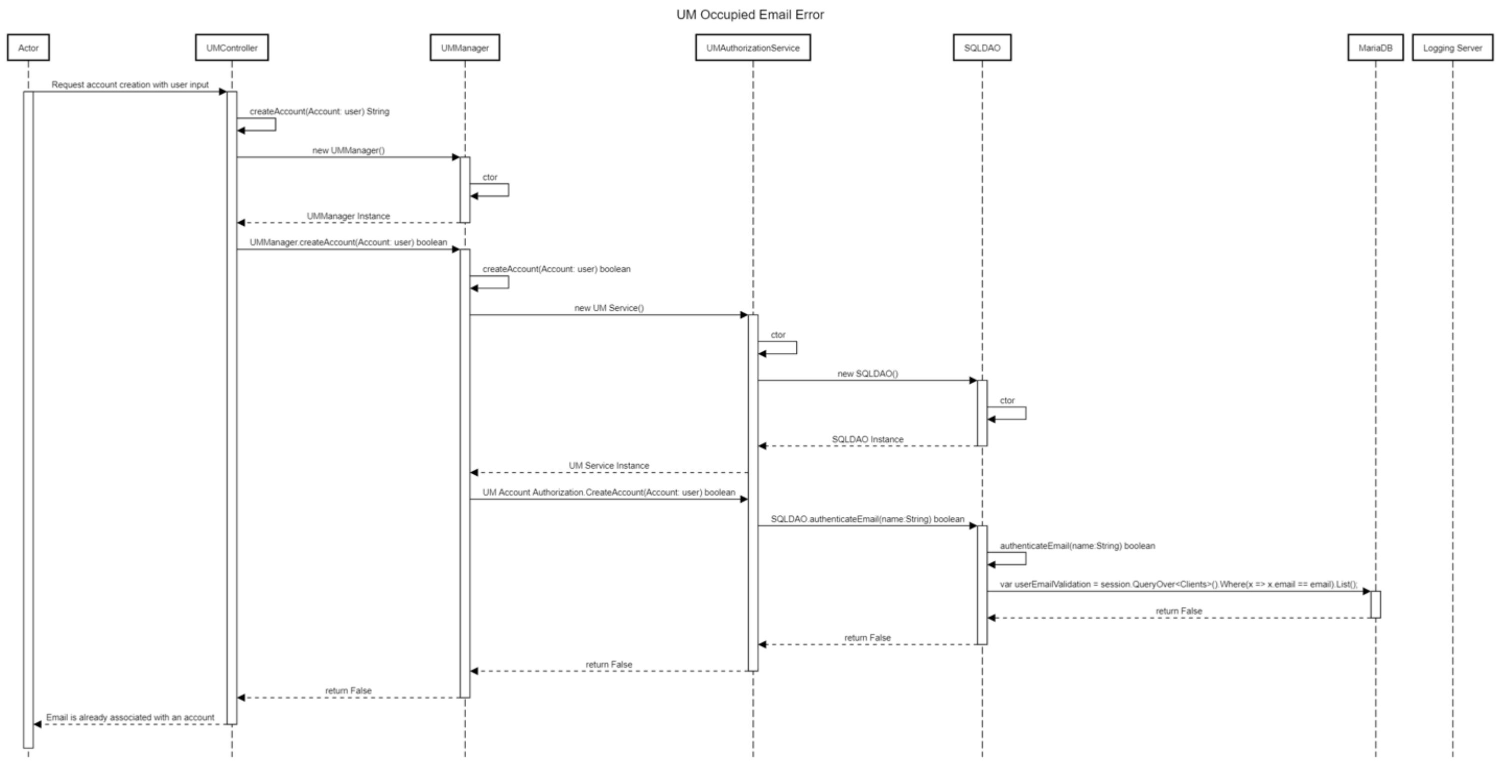
UM Invalid Password Character Error illustrates the case in which a user inputs invalid sequence of characters when setting their password. Invalid inputs include weak passwords or any string that does not adhere to our password guidelines prompted to the user. The user will be notified on what the minimum requirements are for a password when their input does not meet the requirement.

2.1.4 UM Database Disconnect Error


UM Database Disconnect Error
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account creation with user input
createAccount(Account: user) String
new UMManager()
ctor
UMManager Instance
UMManager.createAccount(Account: user) boolean
createAccount(Account: user) boolean
new UM Service()
ctor
new SQLDAO()
ctor
Unable to connect to server
UM Service Instance could not be created
False
Connection to server could not be established

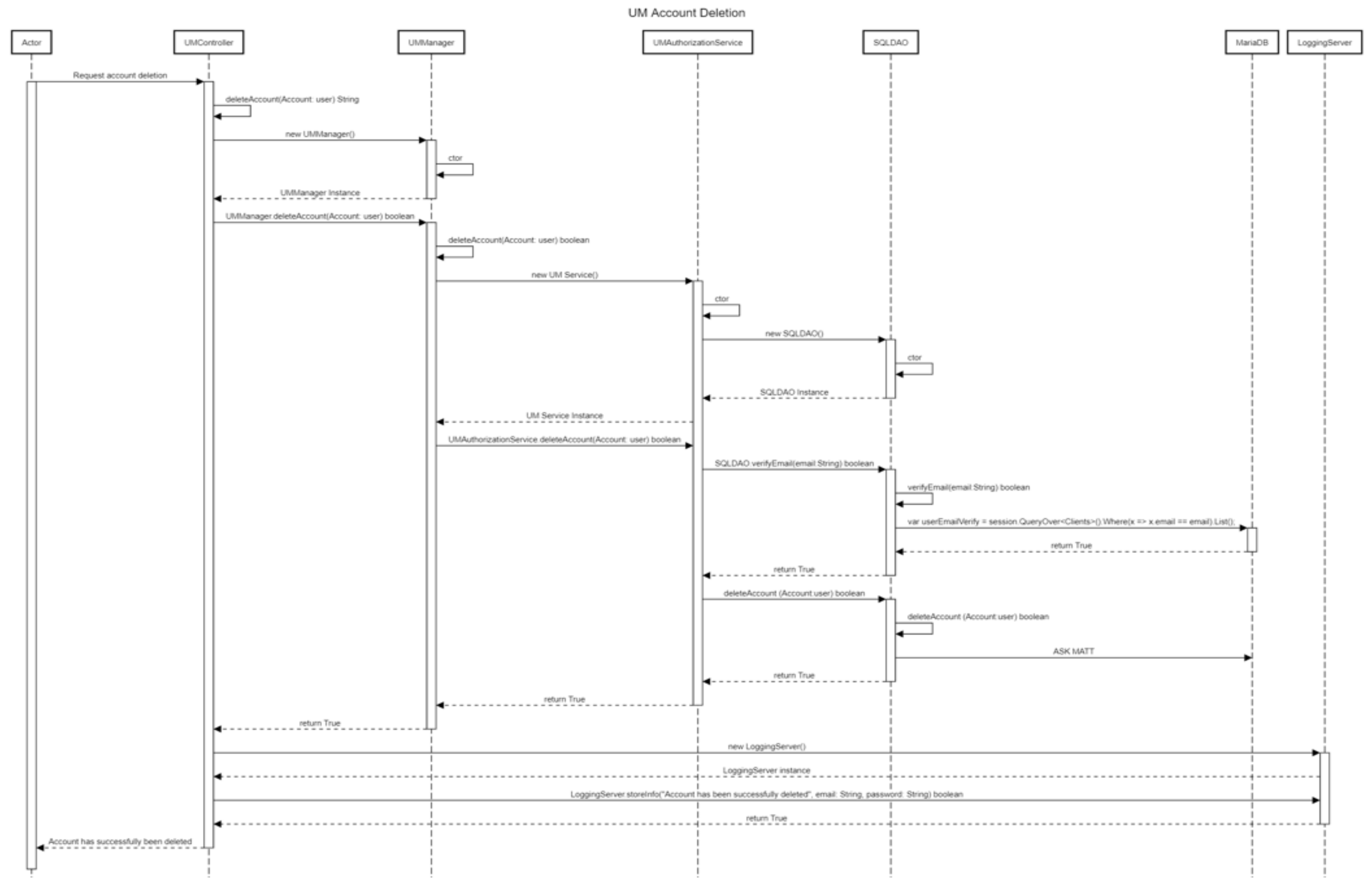
UM Database Disconnect Error illustrates the case in which an account creation fails due to the SQLDAO being unable to access the Client Server. If the SQLDAO is unable to access the MariaDB for any reason, the SQLDAO will notify UMAuthorizationService that server access has failed and therefore account creation must be canceled. The UMAuthorizationService will then notify the UMManager. The UMManager will then notify the UMController that access has failed. THe UMController will notify the user that access to the server was unsuccessful and they are currently unable to create an account.

2.1.5 UM Occupied Email Error


UM Occupied Email Error
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
Logging Server
Request account creation with user input
createAccount(Account: user) String
new UMManager()
ctor
UMManager Instance
UMManager.createAccount(Account: user) boolean
createAccount(Account: user) boolean
new UM Service()
ctor
new SQLDAO()
ctor
SQLDAO Instance
UM Service Instance
UM Account Authorization.CreateAccount(Account: user) boolean
SQLDAO.authenticateEmail(name:String) boolean
authenticateEmail(name:String) boolean
var userEmailValidation = session.QueryOver<Clients>().Where(x => x.email == email).List();
return False
return False
return False
return False
Email is already associated with an account

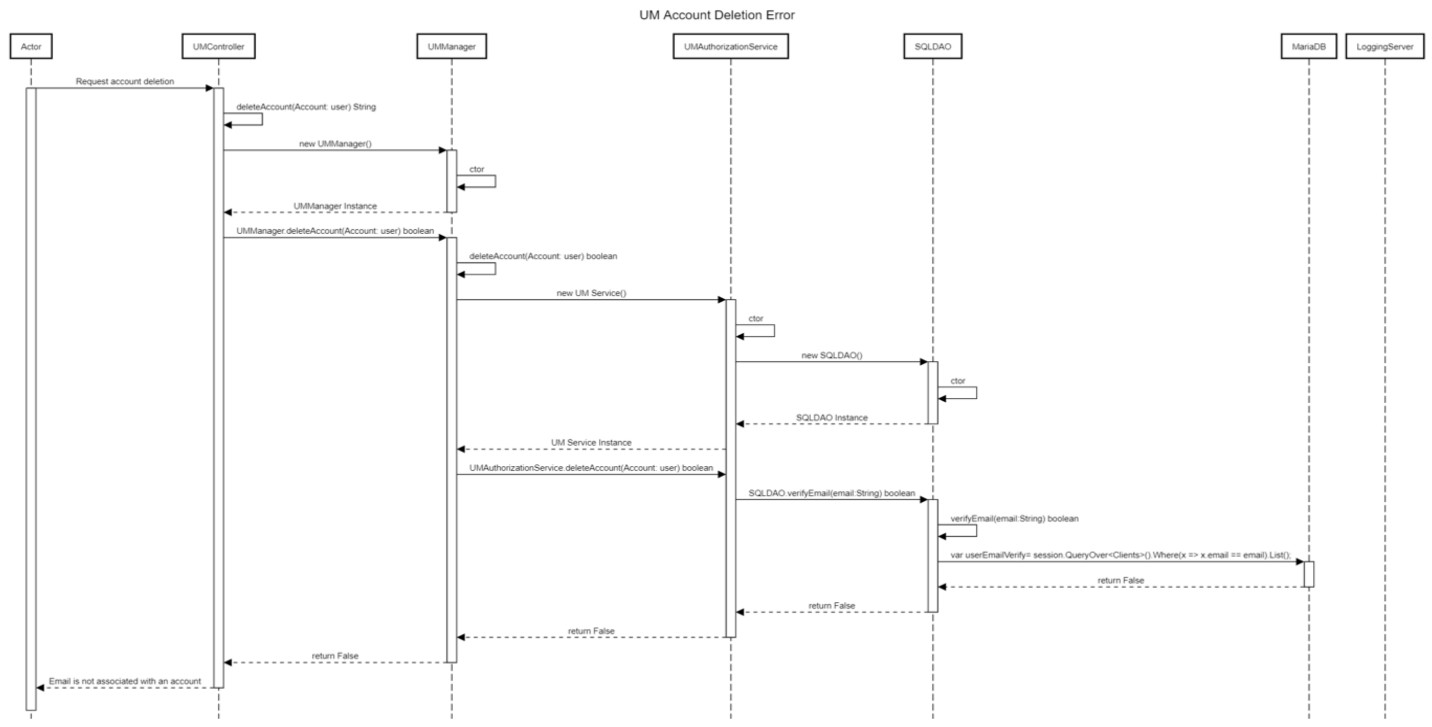
UM Occupied Email error illustrates the case in which a user attempts to create a new account with an already registered email. The user will input an email address and a password to register an account. From there, SQLDAO will confirm with the server to authenticate whether the email is available or not. If the email is already in use, the DAO will stop the account creation process and notify UMAuthorizationService of this failed case. The UMAuthorizationService will then notify the UMManager. From there the UMManager will notify the UMController. The UMController will then notify the user that the email account is already associated with an account.

2.1.6 UM Account Deletion


UM Account Deletion
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account deletion
deleteAccount(Account: user) String
new UMManager()
ctor
UMManager Instance
UMManager.deleteAccount(Account: user) boolean
deleteAccount(Account: user) boolean
new UM Service()
ctor
new SQLDAO()
ctor
SQLDAO Instance
UM Service Instance
UMAuthorizationService.deleteAccount(Account: user) boolean
SQLDAO.verifyEmail(email:String) boolean
verifyEmail(email:String) boolean
var userEmailVerify = session.QueryOver<Clients>().Where(x => x.email == email).List();
return True
return True
deleteAccount (Account:user) boolean
deleteAccount (Account:user) boolean
ASK MATT
return True
return True
return True
new LoggingServer()
LoggingServer instance
LoggingServer.storeInfo("Account has been successfully deleted", email: String, password: String) boolean
return True
Account has successfully been deleted

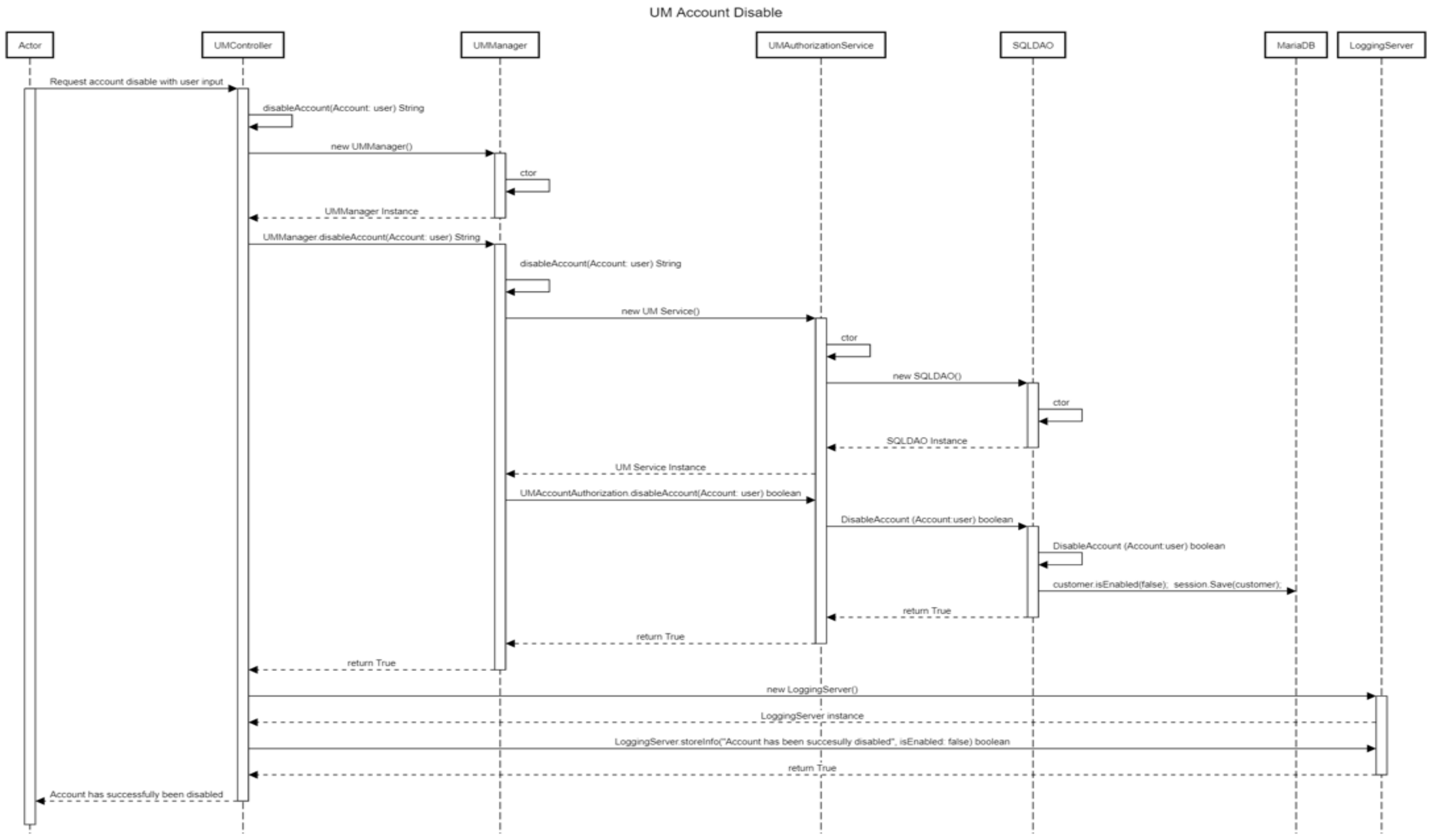
UM Account Deletion illustrates the scenario in which a user wants to delete their account. Ther user will first enter their email and password they wish to delete. From there request will be passed onto the UMManager to the UMAuthorizationService to the SQLDAO. The SQLDAO will then confirm with the DQL Client Server and confirm whether email is associated with an account. Once this is verified, SQLDAO will delete the user account from MariaDB. Once this is successfully done. The SQLDAO will parse back through the UMAuthorizationService, UMManager, and UMController . Once the client receives this news, the UMController will log this event by accessing the loggingServer. Once this event is successfully logged, the UMController will notify the user that their account has successfully been deleted.

2.1.7 Account Deletion Error


UM Account Deletion Error
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account deletion
deleteAccount(Account: user) String
new UMManager()
ctor
UMManager Instance
UMManager.deleteAccount(Account: user) boolean
deleteAccount(Account: user) boolean
new UM Service()
ctor
new SQLDAO()
ctor
SQLDAO Instance
UM Service Instance
UMAuthorizationService.deleteAccount(Account: user) boolean
SQLDAO.verifyEmail(email:String) boolean
verifyEmail(email:String) boolean
var userEmailVerify= session.QueryOver<Clients>().Where(x => x.email == email).List();
return False
return False
return False
return False
Email is not associated with an account

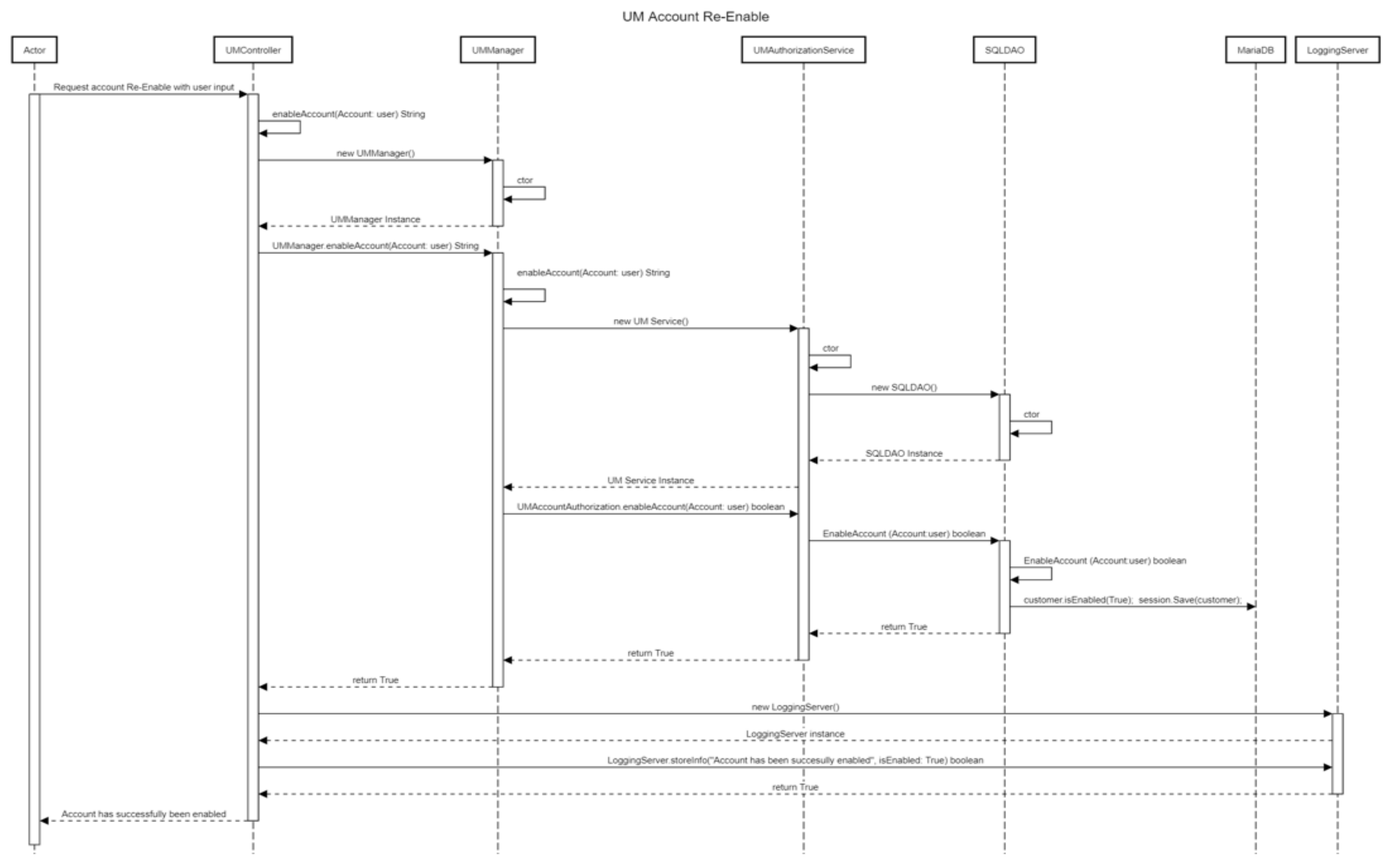
UM Account Deletion Error illustrates the scenario in which an account could not be deleted. This error is the result if the user requests deletion of an account that does not have an associated account. If there are no associated accounts to the email requested, the SQLDAO will notify the UMAuthorizationService which will notify the UMManager which will notify the UMController. The UMController will then notify the user that the email requested to be deleted does not have any associated account within the database.

2.1.8 Account Disable


UM Account Disable
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account disable with user input
disableAccount(Account: user) String
new UMManager()
ctor
UMManager Instance
UMManager.disableAccount(Account: user) String
disableAccount(Account: user) String
new UM Service()
ctor
new SQLDAO()
ctor
SQLDAO Instance
UM Service Instance
UMAccountAuthorization.disableAccount(Account: user) boolean
DisableAccount (Account:user) boolean
DisableAccount (Account:user) boolean
customer.isEnabled(false); session.Save(customer);
return True
return True
return True
new LoggingServer()
LoggingServer instance
LoggingServer.storeInfo("Account has been succesully disabled", isEnabled: false) boolean
return True
Account has successfully been disabled

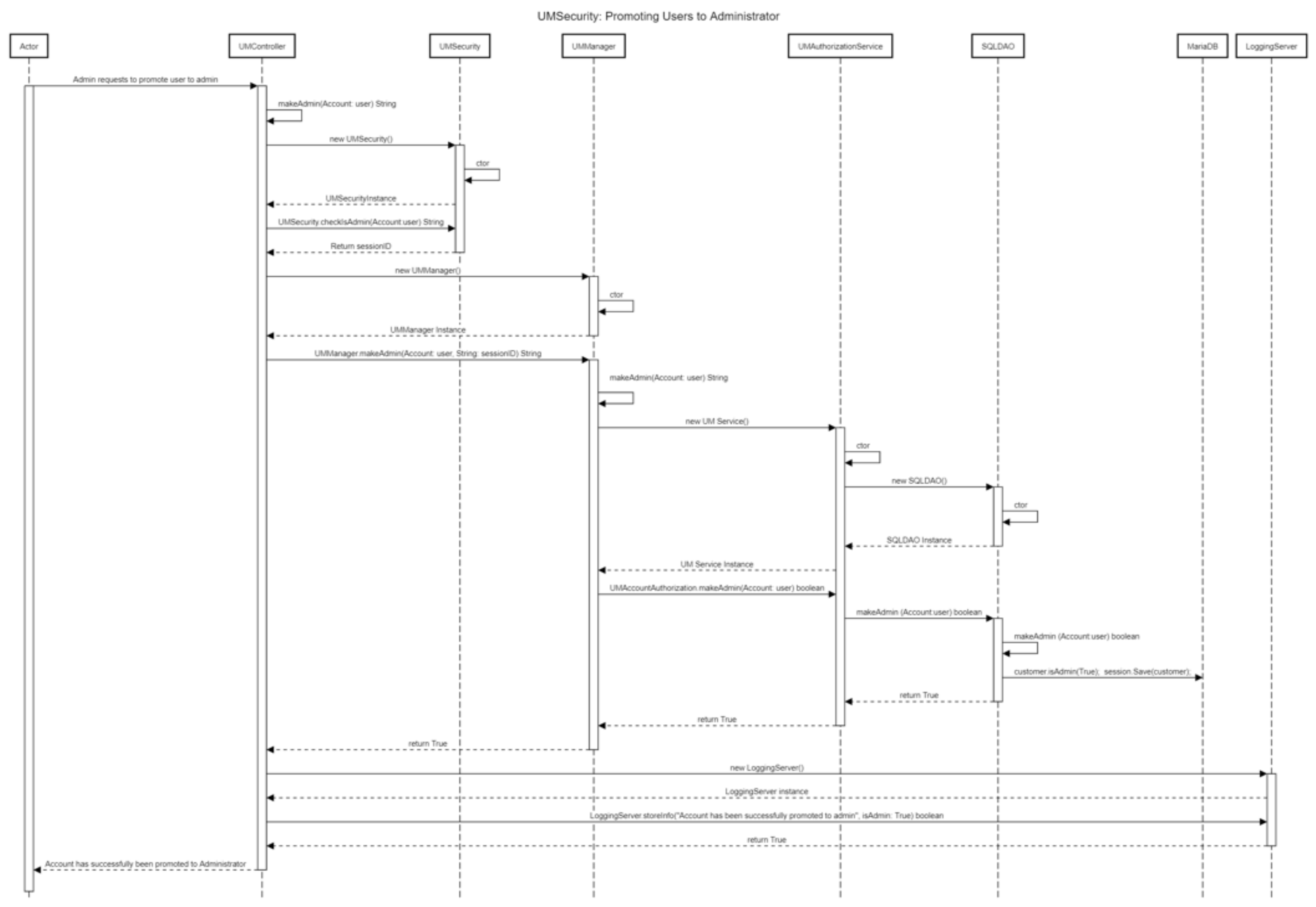
UM Account disable covers the scenario where a user would like to disable their account and stop services, but doesn't want to delete the account as a whole. A boolean value associated with the account isEnabled would be set to false after sending a request. The transaction would then be logged.

2.1.9 Account Reenable


UM Account Re-Enable
Actor
UMController
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Request account Re-Enable with user input
enableAccount(Account: user) String
new UMManager()
ctor
UMManager Instance
UMManager.enableAccount(Account: user) String
enableAccount(Account: user) String
new UM Service()
ctor
new SQLDAO()
ctor
SQLDAO Instance
UM Service Instance
UMAccountAuthorization.enableAccount(Account: user) boolean
EnableAccount (Account:user) boolean
EnableAccount (Account:user) boolean
customer.isEnabled(True); session.Save(customer);
return True
return True
return True
new LoggingServer()
LoggingServer instance
LoggingServer.storeInfo("Account has been succesully enabled", isEnabled: True) boolean
return True
Account has successfully been enabled

UM Account Re-Enable focuses on accounts that need to let users use the application after disabling their accounts. A boolean value associated with the account isEnabled would be set to true after sending a request. The transaction would then be logged.

2.1.10 Promote Account as Administrator


UMSecurity: Promoting Users to Administrator
Actor
UMController
UMSecurity
UMManager
UMAuthorizationService
SQLDAO
MariaDB
LoggingServer
Admin requests to promote user to admin
makeAdmin(Account: user) String
new UMSecurity()
ctor
UMSecurityInstance
UMSecurity.checkIsAdmin(Account user) String
Return sessionID
new UMManager()
ctor
UMManager Instance
UMManager.makeAdmin(Account: user, String: sessionID) String
makeAdmin(Account: user) String
new UM Service()
ctor
new SQLDAO()
ctor
SQLDAO Instance
UM Service Instance
UMAccountAuthorization.makeAdmin(Account: user) boolean
makeAdmin (Account user) boolean
makeAdmin (Account user) boolean
customer.isAdmin(True); session.Save(customer);
return True
return True
return True
new LoggingServer()
LoggingServer instance
LoggingServer.storeInfo("Account has been successfully promoted to admin", isAdmin: True) boolean
return True
Account has successfully been promoted to Administrator

UM Administrator promotion covers an admin requesting another user to become an admin as well. The request would be taken to the Security service. By checking if the user is an administrator, a hashed auto-generated ID would be created. It would then be used to call the manager layer to change the requested user to promote to an administrator.

2.2 Client side validation

- Account Creation
    - Checks if the email address meets legal parameters. If not, it will prompt the user to enter a valid email address. If no email is provided will prompt users to add an email.

    - Checks that the user provides both first and last name. If this place is blank will alert the user that they must fill this out before proceeding
    - Account password must contain one capital letter and symbol. This section is required. If account password does not contain the required parameters the user will be notified of its shortcomings and be given the password requirements
    - Confirm Password, the user will be prompted to re enter their password. If both entries of the password are not the same the webpage will alert the user that one of the passwords does not match
- Barcode Scan
    - If the barcode can not be found in the live video stream the page will inform the user to ensure the barcode is as close to the center of the screen as possible and to retry.
    - If a user manually enters a barcode, it must be the correct number of digits. If it is too short or too long the system will highlight the entered code and inform the user that it is of incorrect length
- AMR calculation
    - Checks that the user enters non negative values for weight and height. If the user enters an invalid number, highlight the section red and inform the user that they must enter positive values only

2.3 Server side validation

- Log In
    - Checks that the user's submitted password matches the ID they are attempting to log in with. If no login fails and informs the user then invalid credentials were submitted. If the Username and password where correct, logging will be successful and redirect the user to the main menu of the application
- Barcode Scan
    - Checks if the submitted barcode exists within our database. If it does move onto returning the food information data. If not, inform the user the product does not exist and ask if they would like to add it to our database
- Food Information
    - Reviews: Validate user review and make sure there is no use of profanity or inappropriate language.

Data Design

3.1 Keys and Tables

- Table: Account
  - Account will contain information about the user’s account, however it will not hold any information about the user’s interactions with our services functions.
  - Primary Key: aEmail
  - Associations:
    - AMR: Each Account is tied to one and only one AMR
    - FoodFlags: Each account can have many FoodFlags
    - History: Each account can have many History
    - Reviews: Each account can have many Reviews
- Table: AMR
  - AMR will hold account information needed to calculate a user’s personal AMR as well as their own custom AMR if stated. This has been separated from Account since not all user accounts will use/have to use the AMR feature.
  - Primary Key: aEmail
  - Foreign Key: aEmail
  - Associations:
    - Account: Each AMR is tied to one and only one Account
- Table: FoodFlags
  - FoodFlags will hold all ingredients that a given user has ‘flagged’ to be warned about if found in a given food item.
  - Primary Key: aEmail, ingredient
  - Foreign Key: aEmail
  - Associations:
    - Account: Each FoodFlag can have at most one Account
- Table: History
  - History will hold the scan history of our users.
  - Primary Key: aEmail, foodID, date, time
  - Foreign Key: aEmail
  - Associations:
    - Account: Each History can have at most one Account
- Table: Reviews
  - Reviews will hold all the reviews made by our users.
  - Primary Key: aEmail, barcode
  - Foreign Key: aEmail (Account), barcode (Food Item)
  - Associations:
    - Account: Each Review can have at most one Account
    - FoodItem: Each Review can have at most one FoodItem
- Table: FoodItem

    - FoodItem will contain product information of a given food item.
    - Primary Key: barcode
    - Foreign Key: labelID
    - Associations:
        - Reviews: Each FoodItem can have many Reviews
        - Label_FoodItem: Each FoodItem has at least one Label_FoodItem
- Table: Label_FoodItem
    - Label_FoodItem is the joining table between Label and FoodItem.
    - Primary Key: joinID, labelID, barcode
    - Foreign Key: labelID (Label), barcode (FoodItem)
    - Associations:
        - Label: Each Label_FoodItem can have at most one Label
        - FoodItem: Each Label_FoodItem can have at most one FoodItem
- Table: Label
    - Label will hold more specific information about a given food item, specifically the nutritional values of that item. This has been separated from Food Item, since there will be cases where a single Food Item may have more than one Label.
    - Primary Key: labelID
    - Associations:
        - Label_FoodItem: Each Label has at least one Label_FoodItem
        - Ingredients: Each Label has at least one Ingredients
        - Vitamins: Each Label can have many Vitamins
- Table: Ingredients
    - Ingredients will hold all of the ingredients listed on a Label
    - Primary Key: labelID, ingredient
    - Foreign Key: labelID
    - Associations:
        - Label: Each Ingredients can have at most one Label
- Table: Vitamins
    - Vitamins will hold all of the vitamins that are listed on a label.
    - Primary Key: labelID, vitamin
    - Foreign Key: labelID
    - Associations:
        - Label: Each Vitamins can have at most one Label

3.2 User Management

User management will detail how much accessibility a given user has to our application. Furthermore, this section will detail how users can interact with certain aspects within our application.

- Scalability
    - What the Food! Will account for scalability by preemptively setting our max user amount. Because our app will launch locally, we will limit our server capacity to 30 users because we are not expected to pass this capacity. If we are to pass this amount, we will limit the number of users accessing the application at a given time.
- Administrative Users
    - Administrative users will be able to access all features provided within the application.
    - Administrative users will be able to access the user server and be given the ability to access and adjust, such as adding and deleting, information within the server accordingly.
    - Administrative users will be able to access the food products server and be given the ability to access and adjust, such as adding and deleting information, within the server accordingly.
    - Administrative users will be able to promote a basic register user account to an administrative user account as well as demote an admin to a basic user.
    - Administrative users will be able to give users any status accordingly as well as strip them of any title.
    - Only one administrative user may occupy the food products server or users server at a time.
- Login Page
    - All users are able to access the sign up page which can be created by an unused, verifiable email.
    - All users are given the ability to login using an authenticated email and a correlated password.
    - All users will be given the ability to login as a guest but will be limited on their accessibility.
    - All users are not able to access login information of other users as well as the user server which stores all information of every user within our application.
- Home Page
    - All users will be able to access the homepage of our application.
    - Accounted users will be provided aggregated news catered specifically towards them on the homepage.
    - Guests will be provided general news that has been recently posted on the Internet.
- Scan

    - All users will have the ability to scan a product, upload a picture, as well as manually input a barcode number.
- Personal Calorie Count
    - Accounted users will be provided a personal calorie count based on their history's associated account.
    - History of a user will be obtained through the user server. No user will be able to access the user server.
    - Guests are unable to access this feature.
- History
    - Accounted users will be able to access their history associated with their account.
    - User history will be obtained through the user server. No user will have access to this server.
- Food Alternatives
    - All users will be able to upload, scan, or manually input a product in which the application will return multiple food alternatives.
    - All information about food products will be withdrawn from the Food Products server. No users will have access to this server.
- Food Information
    - Accounted users will be able to obtain reviews about a product as well as leave a review on the product.
    - The account user that left the review will be associated with the product under User server while the review itself will be associated with the food products server. The user will not be able to access both the food product server and the user server.
    - Guests will not be able to access this feature.
- Settings
    - All users will be able to access the settings within the application
- Food Flags
    - All users will be able to put filters and flags on specific nutritional labels

3.3 Key Design Considerations

- Primary Keys
    - Account: "aEmail", or account email, is representative of the email that the account is tied to, and is chosen to be the primary key for this table since this value will be able to uniquely identify a tuple. This decision is made under the assumption that there cannot be multiple accounts with the same email.

- AMR: “aEmail”, is chosen for this table’s primary key. This is because just like the account above “aEmail” can be the sole unique attribute for a given tuple.
- FoodFlags: “aEmail” and “ingredient” are chosen as the primary key for this table. Just the ingredient alone is not enough to make a given tuple unique, therefore we need to add another attribute, and for this table the only option is “aEmail”.
- History: “aEmail”, “foodID”, “date”, and “time”, are all chosen as this table’s primary key. If one were to take away any of these attributes then there is a possibility that another tuple would have the same values, therefore all attributes are needed to make this a proper primary key.
- Reviews: “aEmail” and “barcode”, are chosen as this table's primary key. These two were chosen since out of all attributes these two would be able to uniquely identify a given tuple. This is made under the assumption that a user can only have one review per “barcode” (food item).
- FoodItem: “Barcode” is chosen as this table’s primary key. This is because a food item’s barcode will always uniquely identify it, so for purposes of our database it will also uniquely identify a tuple in this table.
- Label: “labelID” is chosen as this table’s primary key. Upon creation of a label object a unique “labelID” will be created and assigned to that object. This is because to uniquely identify a label tuple we would need to include every attribute in the label, and even then there is a chance (although infinitesimally small) that there will be another label with the exact same nutritional values. Therefore a unique ID will be created to avoid copying large amounts of data to the Label Table’s children.
- Label_FoodItem: “labelID” and “barcode” are chosen as the primary key, since to achieve a many to many relationship table, the joining table’s primary key must be the primary keys of both parents.
- Ingredients: “labelID” and “ingredients”, are chosen as this table’s primary key. These two values are the only ones in the table, but are needed to provide a tuple’s uniqueness.
- Vitamins: “labelID” and “vitamin”, are chosen as this table’s primary key. These two values are the only ones in the table, but are needed to provide a tuple’s uniqueness.

● Foreign Keys
- Account’s Children: FoodFlags, History, AMR, and Reviews tables are all children of Account, and all use the same foreign key, “aEmail”. AMR is tied to Account because it stores additional information of that user, if the user decides to give that information. FoodFlags, History, and Reviews are

    all multivariable attributes of Account, so they are separated from Account, but are still tied to it as Account's children.
  - FoodItem's Children: Reviews and Label_FoodItem are children of FoodItem, and it is a shared child of Account. Review has a foreign key from Account (stated above to be "aEmail"), and also from Food Item. That foreign key being "barcode". Review was chosen as a child of FoodItem since a review needs to be tied to a product, in this case a certain food product. FoodItem's foreign key for Label_FoodItem's is also "barcode".
  - Label's Children: Label_FoodItem, Ingredients, and Vitamins are all children of Label. Ingredients and Vitamins are both multivariable attributes of Label, so they are separated from Label, but tied to Label as its children. Label_FoodItem also shares the same foreign key.
- Label and FoodLabel's Mulitvariable Issue
  - The decision to separate FoodItem and Label arose from the issue where some food items can contain multiple labels (for example a variety bag of chips). This means that a given FoodItem can have multiple labels, as well as each Label being able to have multiple food items. These two are multivariable attributes of one another, however we cannot denote a many to many relationship between two tables. We must have a joining table, in this case "Label_FoodItem" creating a many to many relationship between our two tables, while also acting as a list of multivariable attributes for one another.

4.1 Session Management

- When a user logs in with their account a session begins.
- If the user has no interaction with the application for 30 minutes, the user will be logged out automatically and the session will be terminated
- The session will be terminated upon user log out if the user manually logs out
- During a user's session personalized settings will be loaded upon login.

5.1 Unit Testing

- *5.1.1 Create Account*
  - Arrange:
    - Initialize a user object with the new account information
  - Act:
    - If the account information does not exist in the database, then
    - Query the database with the user object to add a new account
  - Assert:
    - Success Case:

            - A new account is created
        - Failure Case:
            - A new account is not created

- *5.1.2 Login*
    - Arrange:
        - Initialize a user object with the user’s account information
    - Act:
        - If the email/password matches one found in the user database, then
        - Query the database with the user object to log the user in
    - Assert:
        - Success Case
            - The user is successfully logged in
        - Failure Case
            - The user is not logged in

- *5.1.3 Scan Barcode*
    - Arrange:
        - Declare an empty barcode object
    - Act:
        - Call the scanBarcode() function and assign the returned value to the barcode object
    - Assert:
        - Success Case:
            - The barcode is captured using the camera and returned by the function
        - Failure Case:
            - The barcode is not captured or returned by the function

- *5.1.4 Manually Enter Product*
    - Arrange:
        - Declare an empty product object to store the product
        - Declare a userInput object
    - Act:
        - Prompt the user to enter a barcode as text input
        - Initialize the userInput object with the text input
        - Query the product database with the userInput object
        - Assign the empty product object to the returned product from the database
    - Assert:
        - Success Case:
            - The user has successfully entered a barcode as text
            - The product database returns the product
            - The product object contains the product
        - Failure Case:

        - Barcode is unable to be entered manually
        - The product database returns the wrong product or a null object
        - The product object does not contain the product

- *5.1.5 Define Chemical Ingredients*
  - Arrange:
    - Declare a map to store the product's chemical ingredients along with their definitions
    - Get the list of ingredients from the product object
  - Act:
    - Query the ingredients database to obtain definitions for each ingredient
    - Assign the definitions to their corresponding ingredients in the map
  - Assert:
    - Success Case:
      - The ingredients database returns the ingredient's definition
      - The map contains the ingredients along with their definitions
    - Failure Case:
      - The ingredients database returns an incorrect definition or a null object
      - The map contains the wrong definitions for the ingredients

- *5.1.6 Articles Based on Currently Scanned Product*
  - Arrange:
    - Get the currently scanned product as an object
    - Declare an empty list object to store the articles
  - Act:
    - Query the articles database with the currently scanned product to obtain articles for the product
    - Initialize the list with the articles returned from the database
  - Assert:
    - Success Case:
      - The articles database returns articles based on the product
      - The list contains the articles returned from the database
    - Failure Case:
      - The articles database returns the wrong articles or a null object
      - The list does not contain the articles returned from the database

- *5.1.7 Food Alternatives*
    - Arrange:
        - Get the currently scanned product as an object
        - Declare an empty list to store the food alternatives for the product
    - Act:
        - Call the foodAlternatives() function to generate food alternatives
        - Initialize the list with the returned food alternatives from the function
    - Assert:
        - Success Case:
            - The foodAlternatives() function generates similar food alternatives for the product
            - The list contains the food alternatives returned from the function
        - Failure Case:
            - The foodAlternatives() function does not generate similar food alternatives for the product
            - The list does not contain food alternatives returned from the function or is null

- *5.1.8 Food Flags*
    - Arrange:
        - Get a list of products to display
        - Get a list of the user's food flags
        - Declare a new list of flagged products to display
    - Act:
        - Initialize the list of flagged products
        - Iterate through the product list
            - If the product or the product's ingredients contain any of the food flags, then
            - Add the product to the list of flagged products
    - Assert:
        - Success Case:
            - The list of flagged products only contains products that match the user's food flags
        - Failure Case:
            - The list of flagged products contains products that don't match the user's food flags

- *5.1.9 Food Filter*
    - Arrange:

        - Get a list of products to display
        - Get a list of the user’s food filters
        - Declare a new list of filtered products to display
    - Act:
        - Initialize the list of filtered products
        - Iterate through the product list
            - If the product and the product’s ingredients do not contain any of the food filters, then
            - Add the product to the list of filtered products
    - Assert:
        - Success Case:
            - The list of filtered products does not contain any products that match the user’s food filters
        - Failure Case:
            - The list of filtered products contains products that don’t match the user’s food filters

- *5.1.10 Personalized Daily Percentage Values (AMR)*
    - Arrange:
        - Create user
    - Act:
        - Call AMR function with required parameters(Age, Gender, Height, Weight, activity level)
    - Assert:
        - Success Case: returns AMR within range (400, 10,000)
        - Failure Case: returns value outside of legal range

- *5.1.11 Calorie Frame of Reference*
    - Arrange:
        - Create user with an AMR
    - Act:
        - Run calorie frame of reference function
    - Assert:
        - Success Case: Function returns a list of ints of length 5
        - Failure Case: Function does not return a list of ints with of length 5

- *5.1.12 In-House Product Reviews*
    - Arrange:
        - Create user
        - Create test review

- Act:
  - Have user leave test review
- Assert:
  - Success Case: if a review from the test user exists in the database
  - Failure Case: No reviews from the test user exist in the database

- *5.1.13 Feedback on Application Recommendations*
  - Arrange:
    - Make user
  - Act:
    - Make user look at food alternatives until prompted to give feedback or until 1000 iterations
  - Assert:
    - Success Case: User was prompted to give feedback
    - Failure Case: User was never prompted to give feedback

- *5.1.14 User Data Collection*
  - Arrange:
    - Create a new user
  - Act:
    - Opt the user out of data collection
    - Make the user go through an operation that would gather information
  - Assert:
    - Success Case: Data was not stored
    - Failure Case: Data was stored

- *5.1.15 History*
  - Arrange:
    - Create a new user
  - Act:
    - Add an item to the users history
    - Access the food item through the history
  - Assert:
    - Success Case: Correct item was added and then retrieved from database
    - Failure Case: Item was not added or not retrieved from database

6.1 Exception handling

- Create User Account

    - Create Account Failure
        - The account information already exist and should not be created
        - Prompts the user to try again
    - Connect to Database Failure
        - The user’s information does not get stored in the database
        - The system tries again three times before alerting the administrator.
    - Secure Password Verification Failure
        - The user’s password is accepted when it’s too weak
        - It should give the user an error when the password does not meet the requirements, and prompt the user to try again.
    - Valid Input Failure
        - notavalidemail$yahoo.com is an invalid email address
        - invalid.com is an invalid email address
        - Prompts the user to try again until successful

- Login Feature
    - Logging into Account Failure
        - Username/Email does not match the one in the user database.
        - Incorrect password
        - Prompts the user to try again
    - Retrieving Account Information Failure
        - Email does not match the one in the user database
        - System prompts user to make a new account
    - Saving Account Information Failure
        - The user must login again after every session
        - System displays below login to remember user account after sign in

- Scan Barcode
    - Camera Opens on Multiple Platform Failure
        - The camera does not work
        - The system prompts the user to reopen the camera
    - Barcode Scanning Failure
        - The barcode is not captured
        - Prompts the user to recapture the barcode

- Define Chemical Ingredients
    - Chemical Ingredients Definitions Failure
        - A definition is not found for calcium phosphate monobasic
        - System will retry three times before alerting administrator
    - Categorizing Ingredients Failure
        - Calcium phosphate monobasic and ammonium bicarbonate cannot be classified

        - System will retry three times before alerting administrator

- SQL Database Access
    - Accessible Product Information Failure
        - The scanned product is inaccessible through the database
        - System will prompt the user to try again
    - Rapid Performance Failure
        - The database return results too slowly
        - After retrying several times the system will alert the administrator
    - Connect to Database Failure
        - The database is inaccessible within the application
        - After retrying several times the system will alert the administrator

- Food Flags
    - Flagging High Fructose Corn Syrup Failure
        - The user is not alerted when a product contains high fructose corn syrup
        - After retrying several times the system will alert the administrator
    - Visual Indicator Failure
        - The product is not flagged with a checkmark
        - The system will prompt the user to try again

- Average Product Rating
    - Product Rating Failure
        - The product rating is not calculated
        - After retrying several times the system will alert the administrator
    - Star Rating Failure
        - The average rating is not shown via stars
        - After retrying several times the system will alert the administrator
    - Connecting to External Sources Failure
        - The website cannot be accessed
        - After retrying several times the system will alert the administrator

- In-House Product Reviews
    - Internal Reviews Failure
        - The review cannot be submitted
        - System will prompt the user to try again
    - Explicit Filtering Failure
        - Explicit reviews are not discarded
        - After retrying several times the system will alert the administrator
    - Character Limit Failure

        - Character input exceeds character limit of 300
        - System will alert the administrator

- Feedback on Application Recommendations
    - Recommendation of Related Products
        - The user is prompted while scanning a new product with the camera
        - If error continues to persist the system will alert the administrator

- User Data Collection
    - Opt Opt Failure
        - The user cannot opt out of user data collection
        - After retrying several time the application will force quit

6.2 Logging

Logs will be made on the following actions

- User attempts to log in
    - Date, Time, Email, IP, Success/Fail, Fail Case, Page Name
- Account (Date, Time, Email, Success/Fail, Fail Case, Page Name)
    - Creation
        - Date, Time, Email, Success/Fail, Fail Case, Page Name
    - Account deletion
        - Date, Time, Email, Success/Fail, Fail Case , Page Name
    - Password Change
        - Date, Time, Email, Success/Fail, Fail Case, Page Name
    - User Name Change
        - Date, Time, New Name, Previous Name, Success/Fail, Fail Case, Page Name
    - Added Or removed Food flags
        - Date, Time, Email, Flag Removed or Added, Ingredient, Success/Fail, Fail Case, Page Name
    - Request Data
        - Date, Time, Email, Success/Fail, Fail Case, Page Name
    - AMR
        - Date, Time, Email, Values Changed, Changed To, Changed From, Success/Fail, Fail Case, Page Name
    - Review

        - Date, Time, Email, Product, Review Rating, Review Text, Success/Fail, Fail Case, Page Name
    - History
        - Date, Time, Email, Product, Index in history, Success/Fail, Fail Case, Page Name
        - Added item to history
        - Accessed item in history

- Scan
    - Add item request
        - Date, Time, Email, Product, Success/Fail, Fail Case
        - Success
            - A new item was added to the database
        - Failure
            - An item add was attempted, but required information was unavailable

- Etc
    - Date, Time, Success/Fail, Fail Case, Page Name
    - All pages will invoke a log if they fail to load within 5 seconds.

6.2 Archiving

Basics

- The system will archive all logged entries made within the system.
- Archiving will automatically execute at the 1st of every month on 00:00:00 AM (PST).
- The archival process must complete within 90 seconds after request.
- All logs older than 30 days are grouped and stored to a specified archival drive.
- Archival drive should always be able to accept any incoming archived memory. Notify the administrative users if the archival drive is nearing its capacity.
- If the archiving process fails, it will retry at 02:00:00 AM (PST) at the 1st of every month.
- If archiving fails after the 2nd attempt, notify the administrator and put the archival process on halt.
- Any failure that may result during archiving will be notified to the administrative users. Furthermore, all failures will not disrupt the system's functionality.
- Persistent data storage must be active and accessible by the server at all times to account for any potential loss of memory that may result during the archival process.

Success Case

- The system will request an archiving process to begin on 00:00:00 AM (PST) of the 1st of the month. Within 90 seconds, all logs older than 30 days are then grouped and stored to a specified archival drive which will accept the incoming log. In the case of a failure, the system will repeat the steps mentioned earlier at 02:00:00 AM (PST) on the 1st of the month. If this step is to fail as well we the archival process will be put on a temporary halt without affecting the system’s state and notify all administrative users of the issue.

Fail Cases

- Archiving does not begin on the 1st of every month.
- Archiving does not begin at 00:00:00 AM (PST).
- Unable to complete the archiving process within 90 seconds after request is made.
- Unable to move and group logs that are older than 30 days and store them to a specified drive.
- Unable to retry archival process for a second attempt on 02:00:00 AM (PST) on the 1st of every month.
- Unable to notify administrative users of failed archiving after a second attempt on 02:00:00 AM (PST) on the 1st of every month was unsuccessful.