CSE 403 Course project: Requirements & Team Policies

Assignment: 02_requirements (washington.edu)

Team Info & Policies

Team Member Roles

By sub-group:

- Front-end design: Caleb, Ricky
- Back-end: Charles, Devika, Jason
- UI design: Charles, Devika, Jason

By person:

- Devika:
  - UI design, Back-end
  - Technical expertise: Java, Python, C, HTML, CSS, JS, SQL
  - Detailed roles:
    - Establish connection to frontend via RESTful API, help with logic for Business Layer
  - Set up weekly meetings

- Ricky:
  - Front-end
  - Technical expertise: Java, C#, C, JS, HTML
  - Detailed roles:
    - Displaying dynamic data retrieved from API, managing user input
- Caleb:
  - Front-end
  - Technical expertise: Java, C++, C, Python, React
  - Detailed roles:
    - Establishing access to backend API and implementing data retrieval and storage on frontend
- Jason
  - Back-end, UI design
  - Technical expertise: Java, SQL, C, HTML
  - Detailed roles:
    - Write tests for starter code, work on database management, and data accessibility
- Charles
  - Back-end, UI design
  - Technical expertise: Java, Python, and some C, C++, React, HTML
  - Detailed roles:

- Implementing the backend's Business Logic Layer, connecting Data Access Layer and Presentation Layer to it

Justification: Member roles chosen by each member's expertise/experience and interest. See team expertise section at the top of this document.

Software Toolset

Front-end: HTML, CSS, JavaScript, React
Back-end: Java, Mongo

Justification: These are key components of web development and technologies that the team members are familiar with. Together they can make a functional web application.

Schedules

Front-end	Architecture & Design (01/31):
	- Login page
	- API calls to backend
	- Displaying a page for a recipe (All milestones completed)
	Architecture & Design Midway point (2/08)
	- Recipe list from a search
	  - API call using authentication (Caleb - 2 days)
	  - Presentation of dynamic information (Ricky - 3 days)
	- Profile page
	  - Displaying saved recipes (Ricky - 1 day)
	  - Adding dietary preferences/restrictions (Caleb - 1 day)
	Beta Release (02/14):
	- Save recipes to profile (Caleb - 1 day)
	- Add/view current ingredients list (Ricky - 1 day)
	- Share recipe (Caleb - 2 days)
	- Tagging recipes (Caleb - 1 day)
	- Filtering recipes (Ricky - 1 day)
	Final Release (02/21):
	- Generation of shopping list from required/present ingredients (Ricky - 2 days)
	- General bug fixes (Caleb - 5 days)

Back-end	Architecture & Design (01/31): - Design/setup databases (for instance: login, recipes, ingredients, popularity, dietary restrictions, etc). Architecture & Design Midway point (2/08) - Set up Gradle for whole project (Charles - ~1 day) - Create starter code (no implementation) based on architecture and design (Charles, Jason - ~2 days) - Write tests for starter code (Charles, Jason - ~5 days) - Ability to log in based on email and password (Jason - ~ 2 days) - Allow search information retrieval from databases on dummy tests (i.e.

	set up database) (Devika - ~3 days) - Set up a server connected to the front end - (Devika - ~4 days) Beta Release (02/14): - Add implementation to starter code for business logic layer + connectors from it to presentation layer & data access layer (Charles - ~7 days) - Add implementation to starter code for presentation layer; make sure it properly connects to front end (Devika - ~7 days) - Add implementation to starter code for data access layer; make sure it properly connects to DB (Jason - ~7 days) These implementations will achieve these milestones: - Accessing relevant APIs and constructing basic recipe database - Ability to search for a recipe - Open recipe page - Create a recipe and share it - Viewing and adding ingredients to list - Updating personal shopping list Implementation and Documentation (02/21): - Implement connection to Mongo DB (Devika, Jason - ~ 7 days) - Implement and test the rest of the use cases (Charles - ~ 3 days) - Design the API routes for the rest of the use cases (Charles - ~ 1 day) - Implement connection between frontend and backend (Charles - ~ 3 days), using those API routes Final Product (03/07): - Additional use cases (Charles/Jason/Devika - 1~5 days for each new use case, depending on use case) - Obtaining recipes - Make web crawler (Devika/Charles/Jason - ~7 days) - Use web crawler, make sure it performs well (Charles - ~4 days) - Format web-crawled data for use in DB (Devika - ~1 day)
UI	Architecture & Design (01/31): - Designing the recipe page (Completed) Architecture & Design Midway point (2/08) - Designing other/menu pages (Devi - 1 day) Beta Release (02/14): - Designing components (Charles - 2 days) Final Release (03/07): - Implement accessibility into website (Jason - 2 days)

Artifacts

- Repository: [CSE 403 RecipeCart]

Communication Channels

- Slack channel
    - Primary mode of communication with TA Apollo
    - Policies:

        - Respond within a reasonable time when needed (i.e. within the day)
        - Be sure to stay updated with the channel (especially if Apollo requests something), so if something needs to be done it doesn't get forgotten
- Discord server
    - Policies:
        - Respond within a reasonable time (i.e. within the day)
        - "Stand-up meeting" Discord channel where text-based stand-up "meeting" is done. Every day, any time during the day, each member sends a message with:
            - What they did yesterday
            - What they'll do today
            - Any blockers
            - (it's okay if a member didn't do anything for the project on a particular day; they can send a message about it)

Product Description

RecipeCart will be an application that aims to integrate recipe searching/cooking and ingredient-shopping into a seamless two-in-one tool. Users can search, save, upload, and rate recipes, and the (remaining) required ingredients for these recipes can be added to a built-in shopping list. This app aims to fix a common problem, of getting the right amount of ingredients for a recipe, based on what the user already has in their shelves and fridge. Technology-wise, the application will be a mobile app, where the user can login and get recipes and other information from a server.

Major Features

- User-uploaded local storage of ingredients and amounts (manually)
    - Optional to the user; other features should work but may have less functionality
    - User would need to create an account/log in for this
- User-uploaded recipes (manually)
- Default database of recipes supplied to user
- Search recipes with toggle for already-acquired/needs-shopping
    - Filter recipes based on size, time, skill level, dietary requirements
- Grocery list that the user can add ingredients to
    - Add a recipe's (missing) ingredients directly to it

Stretch Goals

- Integration with Amazon/Walmart API for seamless ingredient ordering
- Automatically tracking user ingredients and amounts
- Web-crawl various recipe websites to add recipes to app
- Add further user functionality such as commenting on and recommending recipes

Use Cases (Functional Requirements)

Devika - Create Recipe

Actors	Content creators who want to share recipes.
Triggers	Content creator triggers a share button to distribute recipes on the web app.
Preconditions	Content Creator has accurately entered ingredients, recipe instructions, and preferences (diet, allergies, type of meal, etc.).
Postconditions	Content creator's ingredients and recipe is available and searchable for other users
List of Steps	1. Create an account on website 2. Log into the website by entering correct user/password 3. Enter ingredients and amounts 4. Enter the recipe 5. Tag with the dietary preferences 6. (stretch) enable/disable commenting 7. Click share
Variations of Success Scenario	1. If the user has inputted tags such as a cuisine, dietary restriction, etc. their recipe will be viewable under those search terms 2. Certain ingredients and names may be more searchable than others dependent on saturation of results 3. (Stretch goal) If user has enabled comments, other users can comment under the recipe
Exceptions	Content Creator: Does not have ingredients listed or unknown ingredients listed. No recipe instructions. Incorrect login.

Jason - Search for Recipes

Actors	A person at home with lots of ingredients at home and wants to try out a new recipe.
Triggers	User triggers some search for recipes by clicking a button.
Preconditions	User has filled out a list of ingredients that they have, the type of recipe (breakfast/lunch/dinner) they're looking for, and a range of how expensive the recipe is that they're looking for.
Postconditions	User is supplied with a variety of recipes to choose from, each showing the ingredients required (quantity included), the type

	of recipe it is, and how expensive the recipe is approximately. After finding the right recipe for them, they are able to view it in further detail by clicking on it. The ingredients used are then deducted from their list they filled out.
List of Steps	1. Have the list of ingredients filled out and preferences chosen (Precondition). 2. Click button to start a search for recipes that follow these preferences and ingredients. 3. User searches through the list until they find a recipe that they like and click on it to view how to create the dish in detail.
Variations of Success Scenario	● The desired recipe is shown in detail based off of what the user clicked on from the list of recipes. ● Variation based on what was filtered or preferred. ● User is able to choose how many servings they want. ● A section of recipes that have never been tried. ● A section of recipes that the user has favorited.
Exceptions	● User has not filled out what ingredients they have (if they chose the option to search with ingredients already owned). ● If user has not filled out preferences, then all recipes will be shown for those ingredients or send an error to actor for not filling it out. ● User is not logged in and tries to add ingredients to their list.

Charles - Add “Recipe” to Shopping List

Actors	A user wants to cook a (specific) recipe but doesn’t have sufficient ingredients.
Triggers	● User finds a recipe (such as via search) ● Ingredients the user has (entered onto the app) does not contain all ingredients the recipe requires
Preconditions	● User has already entered what ingredients they have ● User is logged in ● User is on a page with a recipe
Postconditions	● The ingredients the user doesn’t have is entered into their shopping list ● (Stretch goal) The ingredients are then ordered via Amazon/Walmart API

List of Steps	1. User does some input (such as clicking a button) to add the recipe's ingredients to their shopping list. 2. System presents an option to add all ingredients or only ingredients they're missing. They select the latter option. 3. System presents an option for how many servings of the recipe they want. They select 1. 4. System properly adds ingredients to their shopping list. 5. User does some input that takes them to their shopping list. The recipe's ingredients should be properly added to the list. 6. (stretch goal) User does some input that gets ingredients in list to be ordered via a service such as Amazon/Walmart API 7. Ingredients are cleared from the list
Variations of Success Scenario	2b. The user selects the option to add all of the recipe's ingredients to their shopping list. ● The ingredients they already have, not just missing ingredients, will also show up in the shopping list. 3b. The user selects multiple servings instead of just 1. ● The quantities for each ingredient are properly multiplied when added to the list. 5b. There is already ingredients in the shopping list (some of which being the same as the ones in the recipe) ● The list's existing ingredients will not be replaced: ingredients not in the recipe won't be deleted, and for ingredients in the recipe, the system will add the quantities for the recipe's ingredients to the quantities for the list's existing ingredients.
Exceptions	4'. User isn't logged in, or are somehow logged out ● Present them with the login screen. Once logged in from that page, the ingredients should be properly added to the list. 4''. Database that stores their ingredient inventory fails (to properly get their ingredients) (if variation 2b isn't used) ● Present them with an error message explaining that their ingredients couldn't be obtained. Give them the option to add all ingredients to the shopping list. 6'. System fails to connect to ordering service, or ordering service fails ● Present them with an error message saying that the service failed and to try ordering again later. Ingredients are not cleared from the list.

Ricky - Save Recipe

Actors	A user had saved a recipe to cook for later and would like to cook it now.
Triggers	● User finds a recipe (via search). ● User saves the recipe (via the save recipe button).
Preconditions	● User had saved the recipe. ● User is logged in. ● User is currently on the main page.
Postconditions	● The user is on the page for the saved recipe.
List of Steps	1. User navigates to the profile page. 2. System displays a list of user information and account options. The user chooses “Saved Recipes”. 3. System displays a list of saved recipes along with an ordering option which will order the list by different options. They sort the list by most recently added and scroll through the list to find the recipe. 4. The user clicks the recipe and the system displays the page for the recipe.
Extensions/Variations of Success Scenario	3b. The user chooses to not sort the list and keeps the default list. ● They will scroll through the list to find the recipe.
Exceptions	● The user was not logged in. ○ The System will display the login page. The user will log in which will redirect them to the main page. ● The recipe failed to save. ○ The user will need to find the recipe again (via search) and save the recipe.

Caleb - Add Ingredient to Shopping List

Actors	User wants to add an ingredient to the list of ingredients they already have.
Triggers	Actor triggers ingredient input by clicking some button
Preconditions	User is logged in

Postconditions	User now has list of ingredients stored on their account to be referenced for calculation of present/missing ingredients required by recipes
List of Steps	1. User navigates to profile page 2. System displays a list of user information and account options. The user chooses “Ingredient List”. 3. System displays a list of the user's current ingredients and quantities (empty list if nothing added yet.) User selects “Add Ingredients”. 4. System displays dialog to enter one/multiple ingredient(s) 5. User inputs ingredient name and quantity 6. Upon enter/confirm, the dialog disappears and the ingredient appears in the list
Variations of Success Scenario	● 3. ○ A. The user does not have any ingredients added to their supplies list - the list is empty ○ B. The user has ingredients already added to the supplies list - the list shows the current ingredients already in the list ● 5. ○ A. The user adds a single ingredient to the list ○ B. The user adds multiple ingredients to the list
Exceptions	● The user was not logged in. ○ The System will display the login page. The user will log in which will redirect them to the main page. ● Ingredient could not be added to list ○ The system will display an error and re-prompt the user to enter the ingredient again.

Non-functional Requirements

- Certain features need to have security measures:
  - The login system (such as storing passwords encrypted)
  - Anywhere where user’s text is processed (such as custom ingredients, recipe uploading, recipe searching) (so user can’t do any malicious text/spamming)
- Certain storage needs to be scalable: able to store many of them and efficiently search for them:
  - Recipes
  - Ingredients

    - User information (login details, owned ingredients, saved recipes)
- The website should have proper accessibility:
    - Compatible with screen readers
    - Navigate with keyboard

External Requirements

- When invalid user input happens such as filling out what ingredients they have or are looking for or incorrect login information, the website will output error messages to the user based on what was incorrect.
- The application will be deployed so that it will have a public URL that people can access.
- The repository should have instructions for someone else to set up their own server to run RecipeCart.
- The system should be well-documented.
- The scope for each major component of the project (frontend, backend, and UI) should match the available resources (the people assigned to the component, for each component).

Team process description

Software Toolset

Front-end: HTML, CSS, JavaScript, React
Back-end: Java, Mongo

Justification: These are key components of web development and technologies that the team members are familiar with. Together they can make a functional web application.

Risk Assessment

Major Risks

- Construction of initial database of recipes could prove difficult
- Secure way to add ingredients/recipes to database
    - Preventing injection, spamming, etc.
- Designing a way to track user's ingredients that isn't tedious to them
- Efficiency of accessing and retrieving data

Expanding on the 5 major risks we foresee:

- Struggling to construct the initial database of recipes

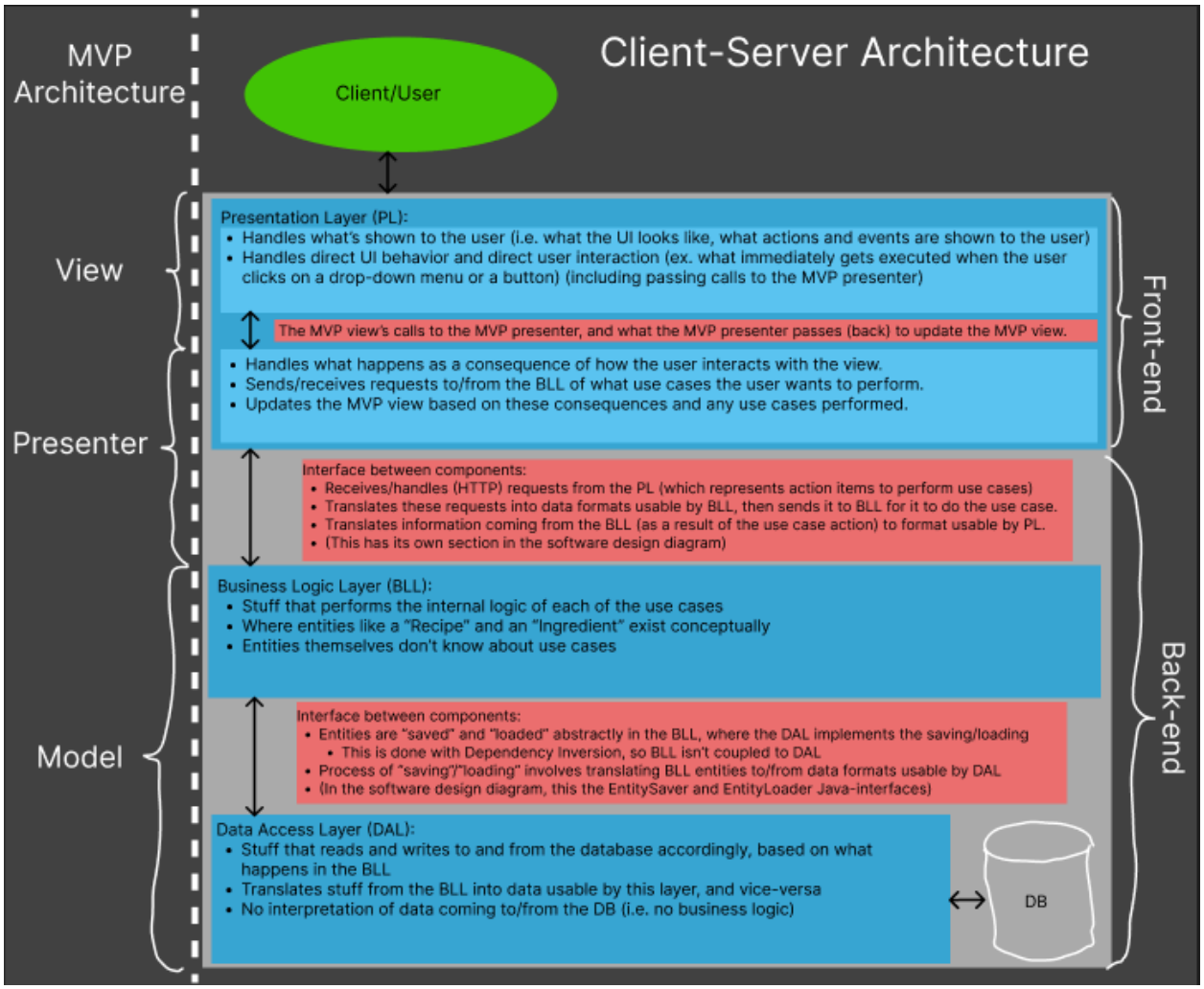
    - The likelihood of occurring would be high, as we're most likely to run into issues trying to put everything together so that the recipes would include all the correct ingredients, quantities, and other attributes that we might want connected to the recipes.
    - The impact of this would also be medium, because users would then get incorrect information on what would be required for the recipe. We would still have the directions to make the recipe, so the user wouldn't be completely lost.
    - Evidence: We have turned our ER diagram into a schema in MongoDB and we already have had a bit of a harder time setting up the collections, so I think that we might require a bit more time doing the database for recipes.
    - Steps to reduce the likelihood/impact would be focusing on getting the main parts of the recipe database up, so the parts that might not be complete won't be too damaging.
    - To detect the problem, we will have to create automated tests (most likely in Gradle) that we will be able to run as we're working on it to try and display all the attributes/directions for the recipe.
    - Mitigation plan for this failure would be to hide the recipe from the user. If the recipe is completely unusable, then it would be better to not show it as an option, than for the user to click on it and no/bad information
- Finding a secure way to add ingredients/recipes to database
    - Likelihood of this happening would be low. We plan to directly import the data into our database, so we believe that it should be accurate.
    - Impact if this occurred would be high, since we depend on the data in our database for the recipes.
    - Evidence: We have tried importing data into databases as an experiment, and we noticed that it was relatively straightforward and didn't have many complications.
    - Steps to reduce the likelihood/impact would be trying to directly import all of our data, which should prevent any misinformation.
    - To detect this problem, we could run tests to see if our database is equal to the imported data to see if anything was altered.
    - If this problem were to occur, then we would maybe have to manually create a simple database with some ingredients and recipes to our database if we are unable to securely add them.
- Secure Login and access to API endpoints
    - Likelihood of occurring: medium. Although we are using an external resource, auth0, and the JWT authentication system, we are self coding the access key authentication (which may lead to issues)
    - Impact: user information would be accessible to others. Opens opportunities to tamper with other's profile and post through someone else's account.
    - Evidence: Based on our research from their website, auth0 should take care of most of the implementation of the login validation tokens, so we would only have to worry about what happens after they are authenticated.

    - Steps: we are using auth0 and JWT, which are well established authentication systems. We are also doing a lot of research to ensure that we set up the key authentication appropriately and securely.
    - Detection: user reported invalid login. Auth0 resources for brute force detection, breached password protection, bots, and suspicious id for suspicious logins.
    - Mitigation: pass fake authentication tokens to ensure that the system is still running properly. Use the aforementioned auth0 resources to mitigate attacks. Potentially send a login email to users if they log in using a new device.
- Injection/spam attacks
    - Likelihood of occurring would be medium. We would have to have a system to prevent a user from sending too many requests.
    - Impact would be medium to high. Our web server might lag behind in requests and it might even be enough for the web server to crash depending on the severity of the spam/attack.
    - Evidence: From the auth0 website, we also see that they will be able to prevent most of the spam attacks on the login side of our webpage, so we believe that it should mainly be the recipe page of our webpage that we will have to worry about.
    - Steps to reduce the likelihood and impact of this problem would be using Auth0 to prevent login spams and coding a part to protect the rest of the web pages from being attacked by preventing a user from requesting too many things at once.
    - To detect this problem, we could attempt to keep track of each user's posts or requests for data and try to determine whether or not they're being malicious.
    - Mitigation for this problem would be preventing a user from doing some action over a certain number of times. Limiting a user from making many requests would also help prevent the web server from lagging behind and crashing.
- Not efficient at accessing and retrieving data
    - Likelihood of occurring would be medium. We would want an efficient script to access all the data required or else the whole system would be slow.
    - Impact would be medium. The less efficient our program is, then the slower the whole process would become, which might be an issue, but it would still function.
    - Evidence: From other courses, there are ways to effectively speed up the runtime of programs and we will attempt to use algorithms that will scale well.
    - To reduce the likelihood of this happening, we will focus more on effective coding patterns to reduce the algorithm times and brainstorm efficient methods of accessing/retrieving data.
    - Detection tracking API call times to figure out where the hold up in the call is. Keeping track of milliseconds
    - Mitigation plan would be to reduce the number of things that happen on the web server by taking out unnecessary components to help the runtime.

Since the submission of the requirements doc, we have begun research and implementation of methods to combat these risks. Most notably, the usage of a third party authentication service,

Auth0, has enabled us with tools to validate users by securely logging in. Using this validation, we can prevent spam attacks on the API, bot attacks on logins, and suspicious IP usage against our website. This also enables us to pass authentication tokens to the backend for secure access of the endpoints. In relation to the database risks, some preliminary research has guided us to use timings on our API endpoints to prevent bottle-necking our efficiency. Construction of the initial database still remains unclear, as that would require some amount of web-scraping or database copying to provide users with default recipes. As a medium impact risk with a high degree of uncertainty, it has taken a slightly lower priority than the other risks and thus has received the least amount of research.

Software Architecture:



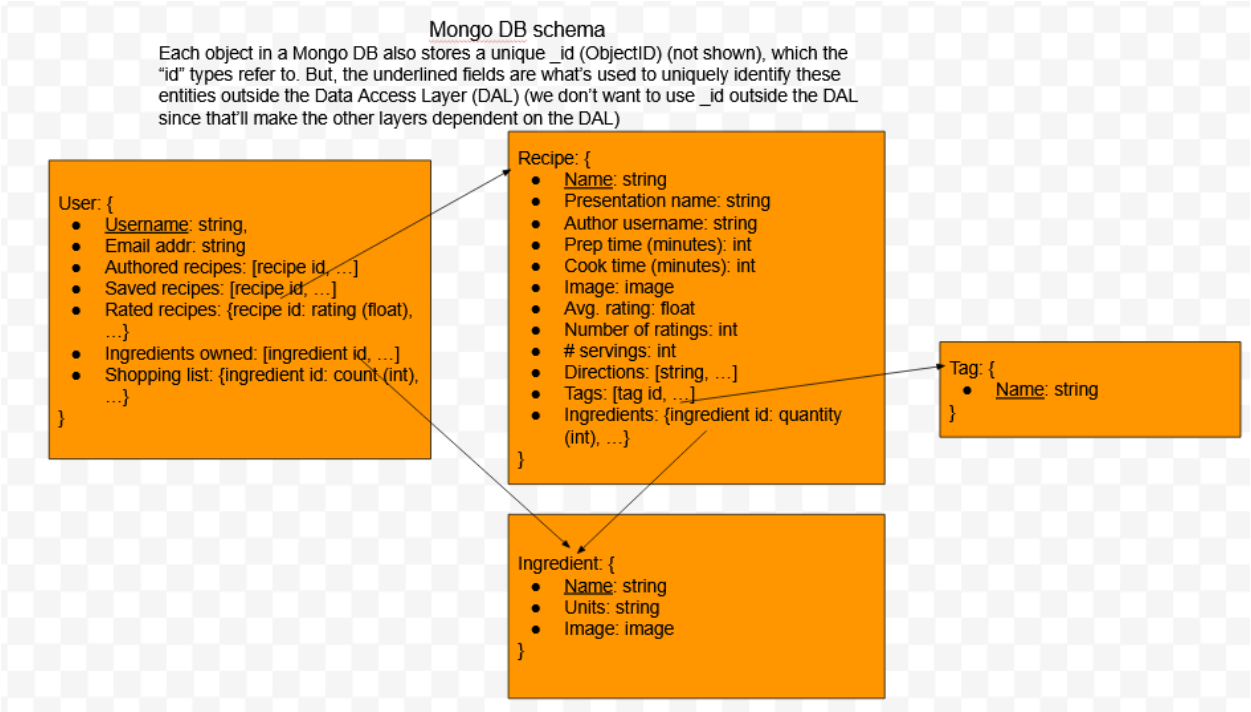
RecipeCart will use the Client-Server Architecture. The diagram above specifies the major software components (Presentation Layer, Business Logic Layer, Data Access Layer), what they do conceptually, and what interfaces exist between them. (Note that no interface exists between the Presentation Layer and the Data Access Layer, since they're concerned with completely different things and shouldn't know about each other.)

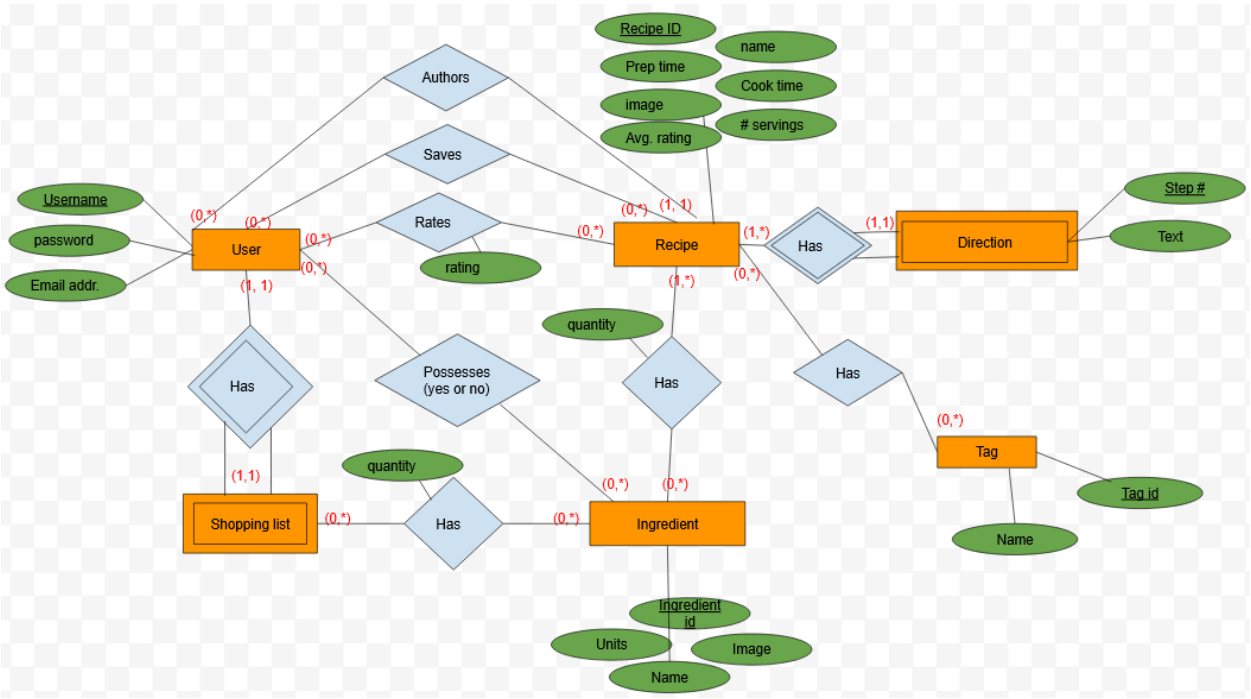
With RecipeCart following the Client-Server architecture, the back-end system will be considered part of the server within this architecture. It consists of the layers/interfaces as shown in the diagram. It will be responsible for retrieving and persisting user/recipe/etc. data, performing business logic based on this data, and communicating with the front-end about this data and what needs to be done regarding users, recipes, etc.

The front-end would be considered the client within this architecture, communicating with the server (i.e. sending the user the website, and the user sending their interactions), as well as communicating requests for the back-end to perform use cases. Communication to both the client and the back-end will be done through the HTTP protocol. The front-end would be considered the presentation layer since it will be responsible for presenting the website to the user and handling user interaction.

RecipeCart will also follow the industry standard architecture design of Model-View-Presenter (MVP). Parts of the architecture in the diagram are labeled with corresponding parts of the MVP. The model component is responsible for the storage/arrangement of data pertaining to entities such as users and recipes, as well as housing the logic for the behaviors of these entities. The view component is what the user sees and interacts with. Interacting with the view component calls the presenter, which serves as a go-between for the model and the view. It is responsible for handling user inputs from the view, calling the model based on this handling when needed, and updating the view as necessary.

The data stored by this system (aside from non-persistent data in the business logic layer (i.e. while doing use cases) and the presentation layer) is data about the different entities concerned by RecipeCart. This includes data about users, shopping lists, recipes, ingredients, and tags. The data access layer will be implemented using a (Mongo) database. The database schema, as well as the ER diagram it's based on, is given below:





Assumptions:

- We assume that Auth0 will be able to track which users have created accounts and the credentials linked to those accounts to properly allow the user to interact with our website. However, based on the way the architecture is set up, this only affects the presentation layer.
- We assume that the data access layer is gonna be implemented with a database, specifically a Mongo database, based on the schema we provided. However, based on the way the architecture is set up, this only affects the data access layer, and this implementation can be swapped out (e.g. for testing purposes).

Alternatives:

1. If Auth0 doesn't apply to our situation/doesn't work out for us, then we would have to use a different authentication framework (ex. FireBase Authentication) or create another database to keep track of this information ourselves.
   Pros:
   - Results in greater flexibility for us, in terms of how we can set up our authentication
   - Changes with the authentication system used only affects presentation layer

   Cons:
   - If we keep track of authentication information ourselves, it may not be as secure as using an existing authentication framework.
   - Making another DB for authentication would mean that some architecture/design has to change (ex. Should this new DB be in the data access layer or a new layer?)

2. If Mongo doesn't work out for us, then we can use a different type of database (ex. SQL) or a different way of storing data (ex. A file system)
   Pros:
   - This new way of storing data may be better than Mongo for us, which benefits us in the long run
   - Changes with how saved data is stored only affects data access layer
   - ER diagram stays the same

   Cons:
   - Would have to come up with a new schema, since the current one assumes Mongo is used

Software design

Definitions for each software component:

- Presentation Layer: this deals with everything about displaying the actual web page to the user (the "front-end") The libraries and packages that will be used for the presentation layer will include React, Material UI, and Webpack. React will be used to provide the infrastructure to build a single page application for our UI. MUI, a pre-packaged library containing useful components, will be used to build the individual pieces that will be used within our react application. Webpack will be used to host our application on a server. This layer also deals with receiving requests from the front-end, that are based on what the user wants to do. It passes details of these requests to the business logic layer (after performing login validation if needed) via RESTful API, and after the business logic layer is done with the request and passes back the results, these results are relayed back to the front-end where the updated information is displayed to the user. We have a variety of pages to display relevant information, and these pages can be found in the `front-end/app/components/pages/` directory. The relevant components are imported from the `front-end/app/components/` directory.
- Business Logic Layer: this layer deals with actually performing the use cases, and is primarily structured around doing these specific use cases. EntityCommands (representing action items for use cases) are executed whenever request details reach this layer (via Commander). To perform these use cases, entities (Recipe, etc.) are created and/or retrieved from the EntityStorage, manipulated, and then saved to EntityStorage as appropriate.
- Data Access Layer: this layer deals with providing data needed for the business logic layer to perform its use cases, as well as taking data from the business logic layer (as a result of whatever use cases are performed) and making sure it's saved appropriately. The former is done through giving the EntityStorage an EntityLoader, which it then uses to retrieve data. The latter is done through giving the EntityStorage a EntitySaver, which it then passes results of use cases to (and expects it to be saved appropriately). This layer provides implementations for EntityLoader and EntitySaver. The methods implemented are what read/write from/to whatever data repository is used (a database, in this case).

A pdf of the design diagram can be found here (it requires you to be in your UW Google account to access; if you can't access it, let us know on Slack right away): Design.pdf. This diagram details the classes that'll be used to form the components of the architecture, as well as their responsibilities and what classes they have dependencies on. There are also some abstractions to note:

- "EntityStorage": this class abstracts away any notion of a database or anything about the data access layer to the business logic layer. If an entity (e.g. an Ingredient) wants to be saved (not as in the use case, but for the data/whatever action happens to be saved), then a saving method is called from this EntityStorage. If an existing entity (that may not have an object currently associated with it) wants to be used, then a loading method is called from this EntityStorage. To do the actual saving and loading, the main method puts an EntitySaver and EntityLoader (from the data access layer) into the EntityStorage after it's created. But EntityStorage isn't concerned with how these interfaces are implemented–it just executes their methods when needed.
- "EntityCommand": classes implementing this interface are the do-ers of the various use cases. This way, the logic of the use cases are abstracted away from the rest of the business logic layer (except for the EntityCommander, which executes these Commands).
- "EntityCommander": this class abstracts away the business logic layer from the presentation layer. A RequestHandler passes in the EntityCommand it wants executed, and EntityCommander executes it (after doing some extra stuff to it, such as setting up the EntityStorage for the EntityCommand to use). EntityCommands are not allowed to be executed outside the business logic layer.

Some other notes:

- The classes in the diagram are not set in stone; minor changes can be made if needed.
- The design was created such that the business logic layer doesn't know anything about how the other layers are implemented. In the diagram, nothing in the business logic layer has a dependency on anything outside the layer. The data access layer could be implemented in a database, file system, etc., and the business logic layer wouldn't be able to tell the difference. It's a similar case with the presentation layer. This makes each layer easier to test individually.
- When making this design, I tried to follow the guidelines of Clean Architecture.
- In order to avoid coupling the Business Logic Layer and the front-end to Mongo (i.e., to make these layers not have to use Mongo ObjectIds), an invariant has been introduced:
    - (for a given EntityStorage's underlying (abstract) storage) each entity has a field whose value is unique to that entity
        - (i.e. other entities of the same type don't have the same value for that field)
        - These fields are:
            - "Username" for users
            - "Name" for ingredients
            - "(unique) Name" for recipes (NOT "Presentation Name")
            - "Name" for tags

        - These fields can be used to uniquely identify entities, outside of the Data Access Layer
        - The DB schema has been configured to adhere to this invariant (NOT vice-versa)

Coding Guidelines:

- For each of our programming languages, we plan to adhere to Google Style guidelines. This link is what we will be referencing: Google Style Guides. We decided on these style guidelines because Google is a successful company that has clear and consistent guidelines.
    - JavaScript Style Guide
    - Java Style Guide
    - HTML/CSS Style Guide
- We will enforce these guidelines through a linter that gets run along with our tests, so mistakes are caught early. For Java files, the linter will be attached to Gradle using the Spotless plugin. For JavaScript files, we will use ESlint.

Process Description

External Feedback Processing

- The primary instance where external feedback would be essential is immediately after the beta release. We would primarily expect feedback about the ease of use, and we would implement this to make our website intuitive and fluid to use.
- Additionally, external feedback on adding ingredients to the list of ingredients the user already has could help us refine the process to make it less tedious and cumbersome.
- To obtain this feedback, we are planning to use Google Forms with various prompts for the features we have implemented in our beta release.

Test Plan & Bugs

For testing, we plan to use unit tests and integration tests to ensure proper functionality of our code. For the frontend, we plan to use the Jest library to unit test a variety of our functions. For backend unit testing, we plan to use Gradle for our implementation. Additionally, after a feature is implemented, it will undergo thorough user acceptance testing by each of the developers. For integration testing, we will establish a Git pipeline to test each new merge for functionality before the changes are pushed to production. In terms of bugs, we will address each bug on a case-by-case basis to determine if the bug takes higher priority than implementing new features before our release deadlines. Currently, we have a text document for tracking known bugs located in `bug-tracking/bug-tracking.txt` where we add a short description about the bugs we encounter. More severe bugs are addressed immediately, and bugs are removed from the document as they are fixed.

Automated Testing and Continuous Integration

CI Service Research:

CI Service	Pros	Cons
GitHub Actions	- Integrated tightly with GitHub which makes it easier if your repository exists in GitHub. - GitHub Actions is built with the focus to be simple and easy to use. - Provides a large set of plugins from the GitHub Marketplace.	- Instantly deployed to GitHub and users (depending on branch), so if there is a bug it will be shown directly to them. - Compared with other CI tools, GitHub Actions is less scalable so it won't be a good option if you're working on a large scale project. - As a project scales up, since GitHub Actions is targeted towards small-medium sized projects, the project will have to be moved to another CI service.
CircleCI	- Has advanced features such as analytics, SSH debugging, etc. - Provides a large set of built-in plugins and reduces the operational overhead of maintaining and installing plugins. - Great security since all variables are encrypted - Includes GitHub repository support	- There's an upper limit on how much we can build, before we'd have to buy credits to build more. - The learning curve for CircleCI is greater compared to other CI tools since it provides a vast amount of customizable features.
Travis CI	- Allows block tests to be run in parallel, reducing the runtime significantly for larger projects. - Allows for direct connection to GitHub. - Easy to setup, does not require a dedicated server and no installation required.	- Build environments have to be rebuilt every time, which can take a long period of time depending on the project size (limited to open source plan as others cost money). - Difficult to switch between systems and limited customization. - Security risks of having public open source projects (Also previously had a mistake that exposed thousands of open source projects).

CI service:

- We plan to use GitHub Actions because it would allow us to implement automatic test building, testing, and using pipelines for testing. GitHub Actions can be enabled on GitHub and will allow us to configure/build automated workflows for our project. Steps to setup GitHub Actions can be found here: GitHub Actions Docs, which has everything on building automated tests, creating workflows, and deploying GitHub Actions.
- Since our repository is also on GitHub, GitHub Actions allows us to easily link our repository to it through GitHub internally, which is another reason we're using GitHub Actions. This allows us to add automated tests through `.yml` files stored in the `.GitHub/workflows/` directory.
    - We have set up our workflows such that there's a separate workflow for the front-end and a separate workflow for the back-end.
        - Back-end workflow: "gradle.yml". This uses the back-end build system (Gradle) to build the back-end files, run tests on them, lint, etc. to make sure everything works there. Only changes that affect the back-end or Gradle will cause this workflow to run (upon pull request/push).
        - Front-end workflow: "node.js.yml". This uses the front-end build system (npm) to build the front-end files, run tests on them, lint, etc. to make sure everything works there. Only changes that affect the front-end will cause this workflow to run (upon pull request/push).
- Due to the small-scale nature of our codebase, we are going to run all unit tests on each CI build.
- CI builds occur whenever there's a push to the remote repository or a pull request. We will ensure that every Pull Request passes all tests before merging, squashing commits if necessary.
- To add a new test to CI, there is a separate set of steps for front-end testing and back-end testing. More detailed steps are listed immediately below.

Test automation/build system infrastructure:

- Back-end: Our test-automation infrastructure will be using JUnit tests that are built by Gradle. We will use JUnit because we are doing Java in the backend and this is a testing structure that we have familiarity in. We plan to test each important method of our code so we know which parts are working and finally we would do a few tests on the overall system, to ensure all parts are working well together. We will have multiple test files so we can run tests on separate methods independently if we want to only check a few files at a time. Using JUnit would allow us to check values of variables through assert statements and JUnit/Gradle will tell us which parts of the test file failed.
    - How to add a new test:
        - Backend tests are located in the src/test/java folder of the repository.
        - You can create a new class and create new tests in that class, or you can create new tests in existing classes.

        - To actually create the test, create a new package-private (no prefix) void method annotated with “@Test” in a new or existing test class.
        - New tests are automatically run with existing ones when doing the Gradle build task.
    - JUnit assertions are needed to properly test functionality. If they’re not already imported in the class, you can import them with
        - `import static com.recipecart.testutil.TestUtils.*;`
    - There’s also @ParameterizedTest that’s often used, which allows the same test to be executed with different values for each. More info on this is here.
    - Conventionally, tests for a java class (that’s in the src folder and subfolders) are named <class name>Test.java, and generally, all tests for that class would be in <class name>Test.java. Conventionally, test method names begin with the word “test”.
- Also, linting will be part of the build process (i.e. the build will fail if there’s a code formatting error), and the Spotless plugin for Gradle will be used to enforce Google style guides for backend code. The build will fail if the linter catches formatting errors. The Gradle “spotlessApply” task will automatically fix formatting errors.
- Front-end: Our automated testing infrastructure for frontend Javascript files is built on the Jest framework. We chose this framework for its variety of features including: Error handling, various equality assertions, and the ability to render components and simulate user actions.
    - To add a new test to the code base:
        - Navigate to front-end/tests/ and create a file with extension `.test.js` and run `npm test` from the command line to execute your test.

Bug tracking

- Bugs are currently tracked in the “bug-tracking” folder of the repository. In the “bug-tracking.txt” file, bugs that are found are detailed, including steps to reproduce them.

Documentation Plan

By nature of the design of our repository, we have split the development process into frontend, backend, and database. This allows us to separate our documentation for local development into each category. Primarily, we will utilize README.md files containing detailed instructions for local development. Additionally, we will use tooltips on the website to make a user-friendly experience.

Back-end: In addition to README.md files, the Java code will be documented via Javadoc, and Gradle will be used to generate their documentation pages.