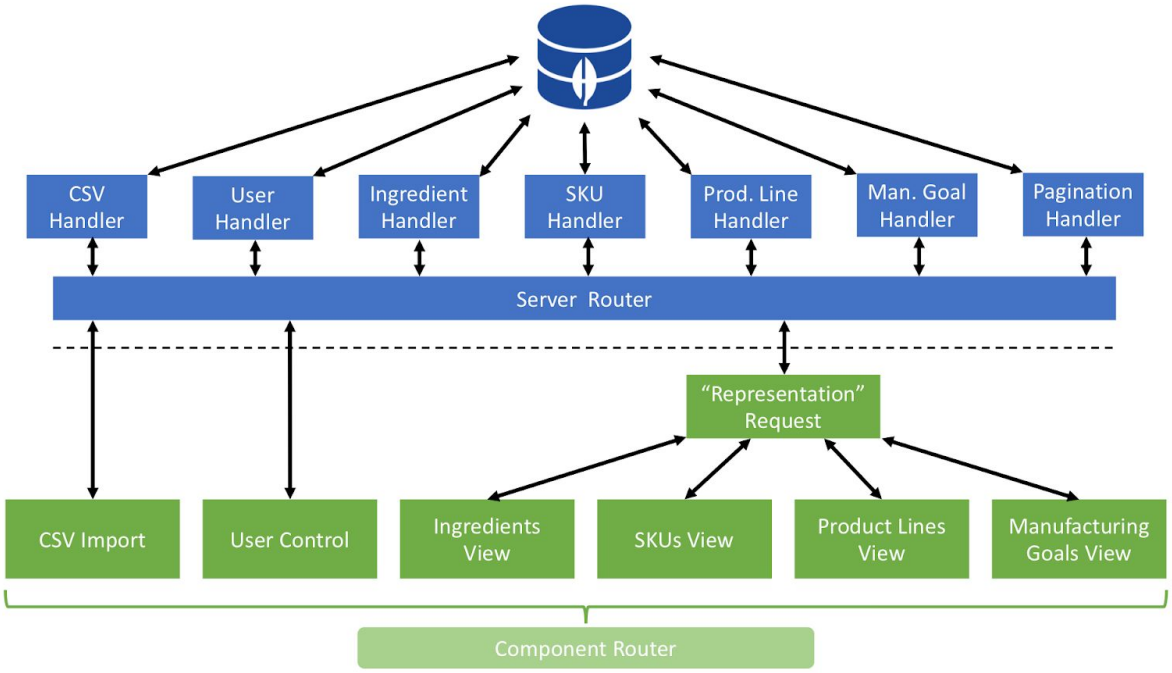
Evolution 1 -- Team Meta

Design Plan

To begin our design plan, we first had a meeting to discuss each of the features and core elements that would be present in the final project. Additionally, we discussed in general terms what aspects of each element would need to communicate with the backend, and how we might utilize a controller of sorts in our design in order to separate the frontend and database/backend of the project. Our ultimate design plan can be seen in the figure below.



Analyzing the above image from the top to bottom we can see that we have a variety of handlers from which to make calls to the database. Each of these handlers has a specific purpose and handles the api calls for specific requests. Additionally, all our API calls are made through a specific Server Router module which enables us to easily organize and edit our API calls if needed. The Server Router serves as the initial point at which the frontend, or the 'Representation' Request makes contact with the backend. In the frontend of the project we have a 'Representation' Request module (often referred to as a SubmitRequest Controller) that determines what response the backend should return given various frontend calls. This SubmitRequest Controller acts as the controller of the project and for almost all sections of the project (except for User Control and CSV Import). It dictates the response needed given a frontend action. For example, if a change is made in the Ingredients View, the SubmitRequest will determine what action needs to be taken given that change and then appropriately retrieves the backend data information and serves it up to the front end of the project. Finally, we use a Component Router to determine which views and components should be displayed at which

time. The Component Router also uses the privileges of the current user to determine if the specified components are able to be viewed by the current user.

Key Features

Key features of our design include the use of the Server Router, the SubmitRequest Controller, and the Component Router. The Server Router was a very important part of our initial design plan as one task we did as a group was mapping out what potential API calls we would need. The Server Router then ensured that we had a single module in which to place all our API calls and therefore ensure consistency across making the calls and then transitioning to the handler. This helps in our development as it makes the organization and the flow of the project easy to understand and continuously replicate.

The SubmitRequest Controller is also a key part of our design as it helps to create a controller for almost all of the views (apart from CSV and User Control). The SubmitRequest Controller was actually a later addition to our design as we had initially had each separate module (i.e. Ingredients view, SKUS view, etc.) interacting directly with the server router, but we felt that it would be important to have a unified controller rather than a miniature version of a controller in each of the separate more frontend modules. This helped our design stay consistent and also enabled us to modify calls and functions in order to work with multiple modules rather than, for example, having to create similar functions for SKUs and Ingredients that virtually served the same purpose.

Finally, our Component Router is also a key aspect of our design as it enables the project to logically determine which component needs to be displayed and then serve that component up to the frontend. Additionally, this allowed us to include an important aspect of User Control and have this permeated throughout the different views in order to ensure that specific users were seeing the correct views and not seeing the views which they did not have permission to view.

Strengths

Many of the key features discussed above are important strengths in our design. The SubmitRequest, Service Router, and Component Router in our design enabled us to be consistent in the project and also modular in our implementation. The separation of backend and frontend became much cleaner and easier to implement when we included the SubmitRequest controller. Additionally, the organization of the API calls has ensured that tweaks to our flow and the API calls and handlers is quickly done and easy to understand.

A major benefit that we hadn't truly understood until talking to other groups is the framework we chose. Many groups had different approaches to which technologies they used for this project, but it's very clear to us that the MERN/MEAN stack is very well equipped to handle this type of project. The extensive documentation and pre-built functionality really made the core of our project come alive quickly. Moreover, React has a great deal of "best practices" that encouraged the team to have similar approaches to implementation. As well, they're effective practices for a modular and scalable web app, so it helped us structure our application well.

Weaknesses

A weakness in our design, which will be discussed more in the implementation discussion is the break up of multiple pages and views. For example, the SKUS component and the Ingredient component had many similar functions and components that could have been abstracted out in our final design.

Another weakness in our design was the use of Mongoose versus generating SQL queries. The fact that we used Mongoose allowed us to very quickly get information in our database, but the lack of a rigid schema led to bugs that took forever for us to find. One specific example of this happened when we changed a field in one of our collections from a String to a Number. Our previous code for conflict checking queried said field using a String key. With this schema change, our code continued to query the field with a String key but since the field was now a Number, it never caught a conflict even if one existed. When we finally caught this bug, we had to go through all our handlers and make appropriate changes to the design. If we had a more rigid schema, we would have caught this bug as soon as we changed the field type.

Implementation

The implementation of our design had benefits and negatives which will be discussed further below. However, an important part of our implementation is how we chose to approach our project and design and how we delegated the work.

Full Stack for Features vs. Frontend/Backend for Core

During the implementation of the design plan discussed above, we found that it was important to ensure that all members of the team had a comprehensive understanding of the full stack. However, we first assigned core components to team members. Therefore, our approach was to delegate the core components, such as the database calls and the table implementations, to specific people which meant that the database person was working entirely in the backend and the person working on the tables was working entirely on the front end. In this way, we split the core components of the project by frontend vs. backend. Then, we chose to split the “feature” components such as the manufacturing goals, user stack, and CSV importing by person. The person assigned to a feature then completed that feature’s full stack in order to gain a complete understanding of the full stack of the feature and project as a whole. We felt that this was an adequate split of our attention (however at times certain people might have had more work, which will be further discussed below). The ability to then own a feature front to back was important when we then had to help our teammates in parts of their feature because then a specific location in the stack was fairly easily understood.

Implementation Benefits

We felt that the way in which we delegated these features/aspects of the core design and our approach of sometimes splitting between frontend and backend and sometimes splitting by feature was a benefit. This enabled our group members to take ownership of features and also made them responsible for testing their stacks front to back. As discussed below, the testing of specific features was fairly effective, however the integration tests were where our

team lacked testing. However, as a whole, owning a stack seemed to be an effective way to code and also allowed teammates to work on their own rather than need to constantly communicate with a different teammate that was designing the front or backend of a feature, etc. This also increased our code velocity because whenever someone on the team needed functionality from a certain stack, all he/she had to do was ask the person in charge of that stack to quickly implement it.

Not only that, but the splitting of front-end vs back-end for the core aspects of our project forced every individual to understand the project as a whole. This is because everyone had to contribute in at least some way to the core design. Ultimately, the fact that we organically allowed splits to be made in dividing up labor (i.e. not restricting every type of split to be between features or between front-end and back-end), we felt that every individual on the team had the opportunity to take ownership of a feature which allowed them to become familiar with a full-stack and also had the opportunity to contribute significantly to the core functionality which forced us all to become familiar with the entire project.

Implementation Negatives

There are many implementation negatives that our team encountered that, had they been avoided, would have made for a more successful first evolution of our project. The following items were huge implementation negatives for our team: poor scheduling, features before core, refactoring late in the evolution, introducing bugs last minute, and no time for testing.

One of the biggest implementation challenges we faced as a team was poor scheduling and task assignment. As a team, we did not set concrete timelines at the beginning of our evolution, aside from a code freeze that we did not meet, and this severely disadvantaged our group. Poor scheduling can lead to so many different negative consequences, likely even more that we have yet to fully understand or realize. Poor scheduling affected all aspects of our evolution. It affected our time management, our task assignment, the number of times we had to meet, and ultimately the amount of time we had to integrate our project. Because of our poor scheduling, our group had to constantly meet to ensure we were on track, however this meant that if the group wasn't meeting, members didn't always know exactly what the deadlines of their tasks were which perhaps stalled productivity. Additionally, the lack of schedule meant we had a very difficult time gauging how effective we were in our implementation and whether or not we were on track to meet our code freeze, which we were not. Had we more formally scheduled and assigned tasks with deadlines, we would have been much more effective working individually as we were working as a group.

Additionally, an important downside of our implementation was the way in which we assigned the aspects of the project and how we went about completing them. More specifically, we were simultaneously building out the "features" of our project, such as the manufacturing goals, before we had completed the basic view of SKUs and Ingredients that included filtering. This proved very difficult as we ran into issues where certain features were dependent on the core of the project, but the core of the project was not yet built or had not yet been completed. What likely would have been better is if we had all worked together to complete the core of the project before splitting off into features. This issue and the previously mentioned issue of

scheduling could have been avoided if we properly roadmapped. This would have let us see early in the implementation that certain features depended on core functionality and also have a better gauge on how much time we actually had left. We definitely realized it's very easy to overestimate how much time you have when you don't schedule it out.

The act of completing features before core also meant that in our implementation we had to do multiple refactors during the middle of our evolution. While some of these refactors, such as the refactoring to create the SubmitRequest Controller were not as time intensive or disruptive, some of the other refactors were more complex. For example, the way in which we retrieve data from the database completely changed midway through the evolution when we finally began to incorporate filtering into our project. It can be easily argued that filtering was perhaps a core part of the project that should have been built out before moving onto other parts of the project. The filtering is especially important in the SKUS/Ingredients view, and therefore, when planning our API calls at the beginning, we should have planned for them. Because our API calls are used throughout our project, this resulted in many files needing to be changed, when in hindsight we could have simply designed this core element at the beginning and avoided the need for refactoring.

An additional part of our implementation that caused a flip-flop in our design plan, and some subsequent inefficiencies, was our refactoring of the ListPage view component. In our initial design, we had intended to have a single ListPage view that could be used by both the SKU Page View and the Ingredient Page view with different properties in order to populate the ListPage accordingly. However, half way through the evolution, we decided that the few aspects of SKU page and Ingredient Page that were different meant that it was easier (at the time) to separate the two out and simply flesh out the differences within the different components. However, later on this resulted in a lot of duplicated code and also, if a bug was found in one version of the ListPage, it would need to be corrected in both versions which could sometimes be tedious or easily forgotten.

Time management, attributed to our lack of scheduling, contributed to certain team members taking on large parts of the work. For example, we had one team member complete the entirety of the styling which, though looked great in the end, was an immense amount of work that should have been divided up. However, given that it was the last 24-hours before our evolution was due, we chose to have the person with the most styling experience take on this task while the other team members continued to work on integration. This was a poor choice in dividing the work and stemmed from our lack of scheduling and time management. It led to one individual taking on a massive part of the project with so little time left that it would actually be less efficient to explain it to everyone else. If we had planned this "explaining time" into our schedule, we could have avoided this.

Finally, because of our lack of planning and poor scheduling of tasks, we did not meet our code freeze. While this alone caused many implementation errors, the biggest of all was our inability to test, which will be further discussed below. As many had advised, the integration of our projects was more time intensive than we had anticipated and because we did not meet our code freeze, our testing time turned into integration time. This meant that in the last few hours before our project was due, we introduced two lines of code that disabled an entire part of our project. This implementation negative, and perhaps our most disappointing implementation

failure, can be traced back to our poor planning and lack of scheduling which led to us working on integration right up until the deadline when we should have been testing and removing bugs.

Design Plan Benefits for Next Evolution

As discussed, there are many key aspects of our design such as the Server Router, SubmitRequest, and Component Router. We believe that these key aspects of our design will enable us to continue building a powerful project and that our design is flexible and scalable. Additionally, many aspects of our design and the way in which we created certain components (aside from ListPage discussed above) makes it easy for us to reuse the components for new pages and views that have similar characteristics to the ones we previously implemented. For example, we have a common page table that is used across our views that will be easily repurposed for the Formulas Page and many other components such as the detailed view pages will be easily implemented because of the modularity and abstractions we did with the first implementation of these elements.

The organization of our API calls and the overall flow of our project from back to front and vice versa will also benefit us in the upcoming evolution as it will enables us to integrate the new changes consistently and ensure that a pattern is continuously used.

Design Plan Downsides for Next Evolution

While aspects of our design will be beneficial for the next evolution, there are also aspects that will perhaps create challenges as we complete the next evolution. One example of this is the use of Redux in our project. In our first evolution, we implemented Redux, a javascript library for managing application state, in the User Control stack, but not anywhere else throughout our project. This then created some inconsistencies in the project which could prove to be difficult in the future evolution as we determine where it might be appropriate to have Redux implemented and where we will not want have redux implemented.

Additionally, the ListPage that was split into IngredientsPage and SKUsPage will potentially need to be refactored back into a single component in order to make the project more scalable and extendable moving forward.

Another downside was our lack of planning for integration with CSV parsing. Since we decided that CSV parsing should be a whole different stack then the rest of the project, we failed to see similarities that could have made its implementation easier. The current state of our CSV parser is 4 files, 2 of which are 500+ lines. This disgusting design is a product of us planning how to integrate something without being fully aware of what its functionality should be. Refactoring this code will definitely be an arduous task for evolution 2.

How could design have been better?

While our design plan, ultimately came together, the initial plan certainly could have been more concrete. While developing our project, we were able to follow a rough idea of how we wanted our design to be implemented based on the basic plan we had created, however, it seemed that the finer details were written into the plan **after** we had actually implemented the specific part of the plan in our code. If we familiarized ourselves more with the desired

functionality, we could have made a better plan. Instead we just made a very vague front-end/back-end split where we hoped to have some sort of central controller to end up with a modular design. The thing is, the previous sentence could be applied to 90+% of web servers. We now realize that being complacent with our overgeneralization led to loads of difficulties when trying to integrate our project. The worst part is the design we came up with was after multiple hours of discussing. Had we known more about the project during that discussion time, our design could have been much more specific to our project.

A concrete way in which we could have had a better design is with a more thought out and detailed testing plan which we will discuss below.

Testing Plan

Though the testing plan is an important part of the design plan, and some would say one of the most essential parts of the design plan, one of our largest design flaws was our lack of a detailed testing plan. More specifically, because we did not have a concrete testing plan, we did not have an estimate around how much time the testing would have required before we started to make our loose schedule.

Our rough testing plan was to include unit tests on various parts of the project which would be incorporated into the project as we went and then ultimately design integration tests to ensure that we hadn't broken any features when we went to integrate all our features. This brings us back to something we mentioned in the previous section about becoming complacent with design. Because we had mentioned that we planned to create unit tests, no one actually implemented any. We all just knew that we could get that done eventually, which ended up becoming never. We kept pushing it off until we decided our code freeze would be our testing period. But, since we were testing all throughout, we were unable to meet our code freeze. And given that we made our code freeze too late and did not even meet the code freeze date, we were never able to fully execute the plan of writing integration tests and simply manually tested our features as we went along instead of writing individual unit tests.

In the next evolution, we are making testing a priority and are moving our code freeze up by two days in order to use those two days to do testing on our project. We hope that this will allow us to catch any bugs that we might introduce upon integration.

Moving Forward

Going forward, the most important things relate to scheduling, modularizing, thinking in the short term and testing.

There was a great deal to learn about in terms of scheduling. It's a category of management and thinking that has a large "barrier to entry", but has exponential benefits once implemented correctly. The first part of scheduling that we'll need to work on is setting down a rough schedule. We say this again and again, but at the end of the day we always felt like it would be more work than it would help us, so we kept pushing it off. This is a huge fallacy and it caused us to not comprehend how much work we had ahead of us, which in turn gave the group unnecessary stress and ultimately buggy code. More than simply scheduling, we need to plan in buffers and extra events (such as code freezes and time for integration). The final touch-ups are

equally as important as writing the code in the first place, so we need to give it the time is deserves.

Similarly, modularizing and parallelizing our work flows would have allowed us to work far more efficiently and would have reduced the time that people spent “blocked” on their coding. To accomplish this effectively, we need to turn everything into a task, assign it to someone, and give it a deadline. Obviously, one person should get tasks that are dependent on each other and largely independent of others’ work. By doing this, we allow people to set their own schedules and complete their work as they have time, instead of waiting on others to complete that. We don’t need to sit in a room and code 5 times a week, we can instead just have several stand-ups a week and work on our own otherwise. With this tasking, we also need to focus on our prioritization of tasks. As discussed, last evolution we failed to put enough resources towards building out the core of the project and instead put people on ancillary and disconnected features like User Control and CSV parsing very early on. If there is no concept of what the core and the database is going to look like, there should not be anyone trying to build features dependent on that. To solve all of these issues, we’ve decided on using Asana as a task-tracker and we’ve put more emphasis on our planning meetings throughout the week to ensure that we stay on track to meet our deadlines.

Thinking in the short term was a concept that didn’t come about until more recently. Belala mentioned during a standup that he gets frustrated when we plan so far out in the future (who’s going to build what) that we lose sight of the things that are actually important. It’s extremely easy to fall into the trap of wanting to plan everything out as early as possible. The more technical side of this issue is that when we’re designing and implementing, we have a concept that we want to build components with many use cases, API calls that can handle everything and a design methodology that can mold to every use case. The issue is that for a system to really have these concepts it must be large enough to have repeated components, API calls and use cases. We were planning as if we’re scaling up to the next big Social Media app, but in reality this is a very minor project. We should absolutely incorporate good design fundamentals, but expecting that a table component that we build is going to be able to be used for every table in our system is short-sighted and will only cause headaches in the future. It’s far more important to build a component to be perfect for the use case that we’re worried about than to be a hugely extensible object that takes many refactors to fit to other use cases.

Finally, as many groups have said and we’ve reiterated several times in this report, we need wide-ranging testing. We need unit tests at every level, system-wide tests for every stack, and a test plan to ensure everything works at every integration. This is a huge undertaking and we are hopeful that we will be able to do this all, but aware that certain aspects might be too large to tackle. However this type of testing is what we need to be able to maintain this project over a long period of time. Even with minor features, we’ve merged branches and had completely unrelated sections break simply because of a weird variable name change. This would take stress out of our integration periods and allow ourselves to spend more time building out new features as opposed to maintaining old ones.