CS 361 Software Engineering I

HW3: Architecture Assignment

Meal planner to promote healthy meals with low waste

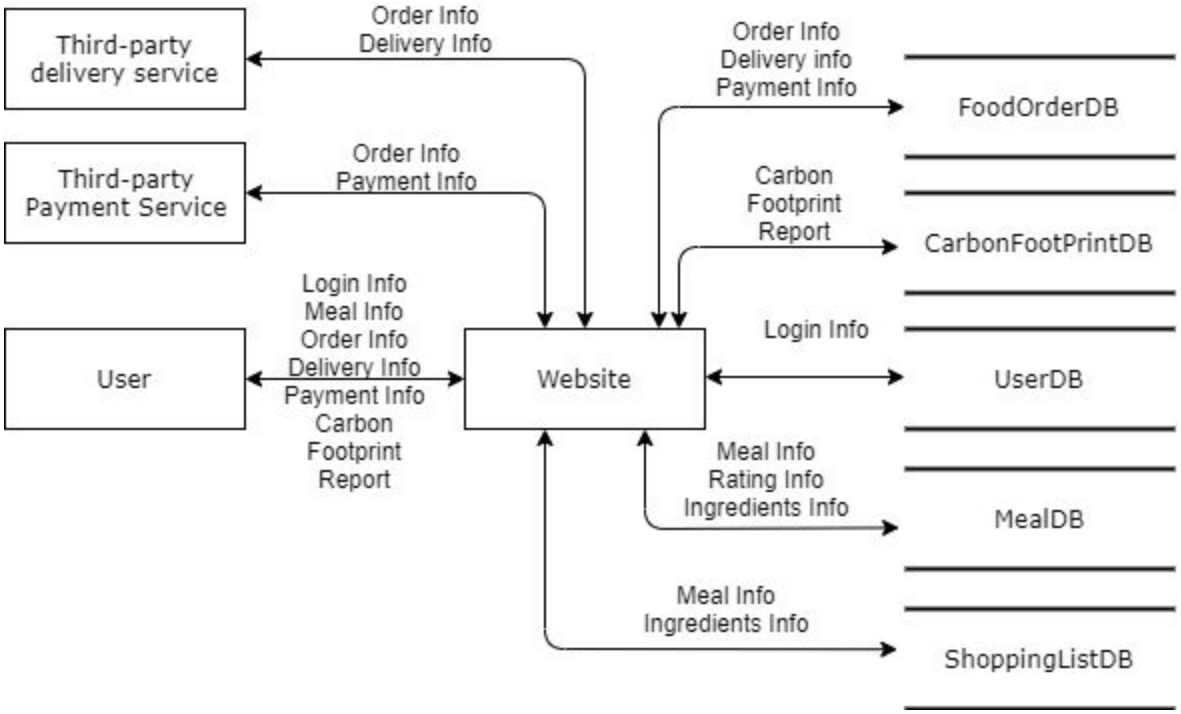
10/27/2019

Group 24

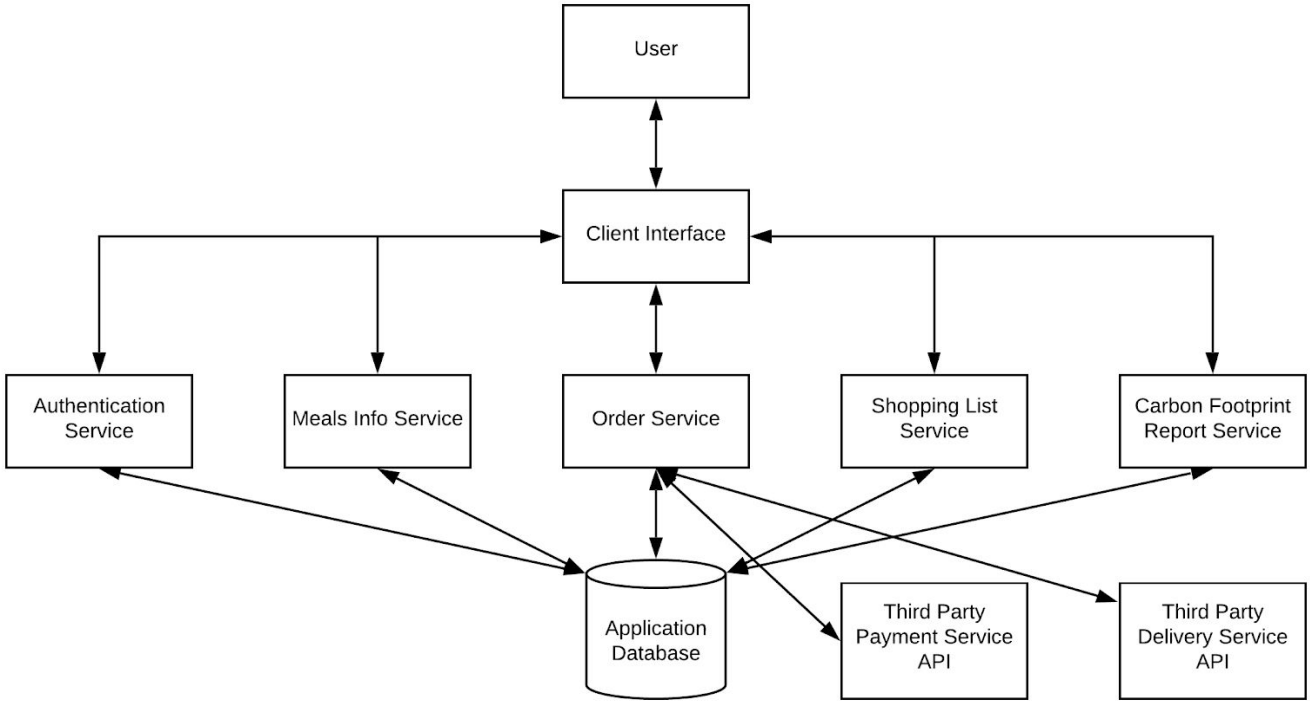
John Casey III
Raymond Lieu
Todd Radin
NianJun Shi
Brian Shim

System Architectures

Web-Application Architecture (server-heavy, monolithic)



Layered Architecture (client-heavy w/ microservices)



Key Quality Attributes

Reliability

- Architecture 1
    - This architecture is less reliable than Architecture 2, since all the work is done on the web server which interacts directly with all of the different components of the system. If the server is down, the web page is inaccessible to the user and no work will be completed on the back end.
- Architecture 2
    - This architecture is strong in regards to reliability. Since there are many microservices involved between the client and database, if one service is not working, it will not bring down the entire architecture, whereas Architecture 1 could pose this problem.

Efficiency

- Architecture 1
    - This architecture may be more efficient than Architecture 2 since there are less components and controllers that need to interact. In cases where the service is not being used by many users, this architecture might also be more performant than the second. In cases where the service is under heavy load, this might not be very efficient as you would need to allocate more resources to the whole server or create another entire server instance, when it might only be one small part of the program that needs additional resources.
- Architecture 2
    - The system design is much more complicated than Architecture 1. The overhead introduced by this complexity might make our application less efficient and performant when not being used by many users. However, since this architecture contains many isolated microservices, each individual component should be more efficient than Architecture 1 since they will have less to do. Also, in situations where the service is under heavy demand, this architecture will be more efficient as the specific services under heavy load can be scaled up to use more resources independently of the others. Likewise, those not being used very much can be scaled down.

Integrity

- Architecture 1
    - Both architectures provide similar levels of security. The main difference is the level of abstractions. With this architecture, users are authenticated directly against the backend database by the web server.
- Architecture 2
    - Both architectures provide similar levels of security. The main difference is the level of abstractions. With this architecture, we make use of a separate authentication service. This increases separation of responsibilities and might mean slightly higher security overall. Users would be issued a token by the authentication service, which would be later validated by the other services to determine authorization.

Usability

- Architecture 1 and Architecture 2
    - Both architectures provide the same level of usability to the client, as the user experience will remain unchanged. The differences are seen solely on the back-end.

Maintainability

- Architecture 1
    - This architecture might be simpler to build out at first as there are fewer components overall. This architecture might be quicker to build for some developers who are not accustomed to microservices. As the application increases in size and complexity, we might start running into issues as monolithic applications are typically not very welcoming to new developers.
- Architecture 2
    - While there are additional components in this architecture, since the components are separated out by role, it should make updating the system easier as we can build out functionality and features on the components themselves independently of the others. Additionally, unit testing might be easier as we will have individual services with clearly defined roles and APIs.

Flexibility

- Architecture 1
    - This architecture can adapt to changes, albeit more slowly than Architecture 2.
- Architecture 2

    - This architecture provides much more flexibility than Architecture 1 since the components are built as microservices. As business needs change, we can alter the services accordingly. As long as the microservice API endpoints stay the same, we are free to alter the implementation of each service however we see fit without modifying the client or any other microservice at all.

Portability

- Architecture 1 and Architecture 2
    - Both architectures provide the same level of portability for the user since this is a web application.

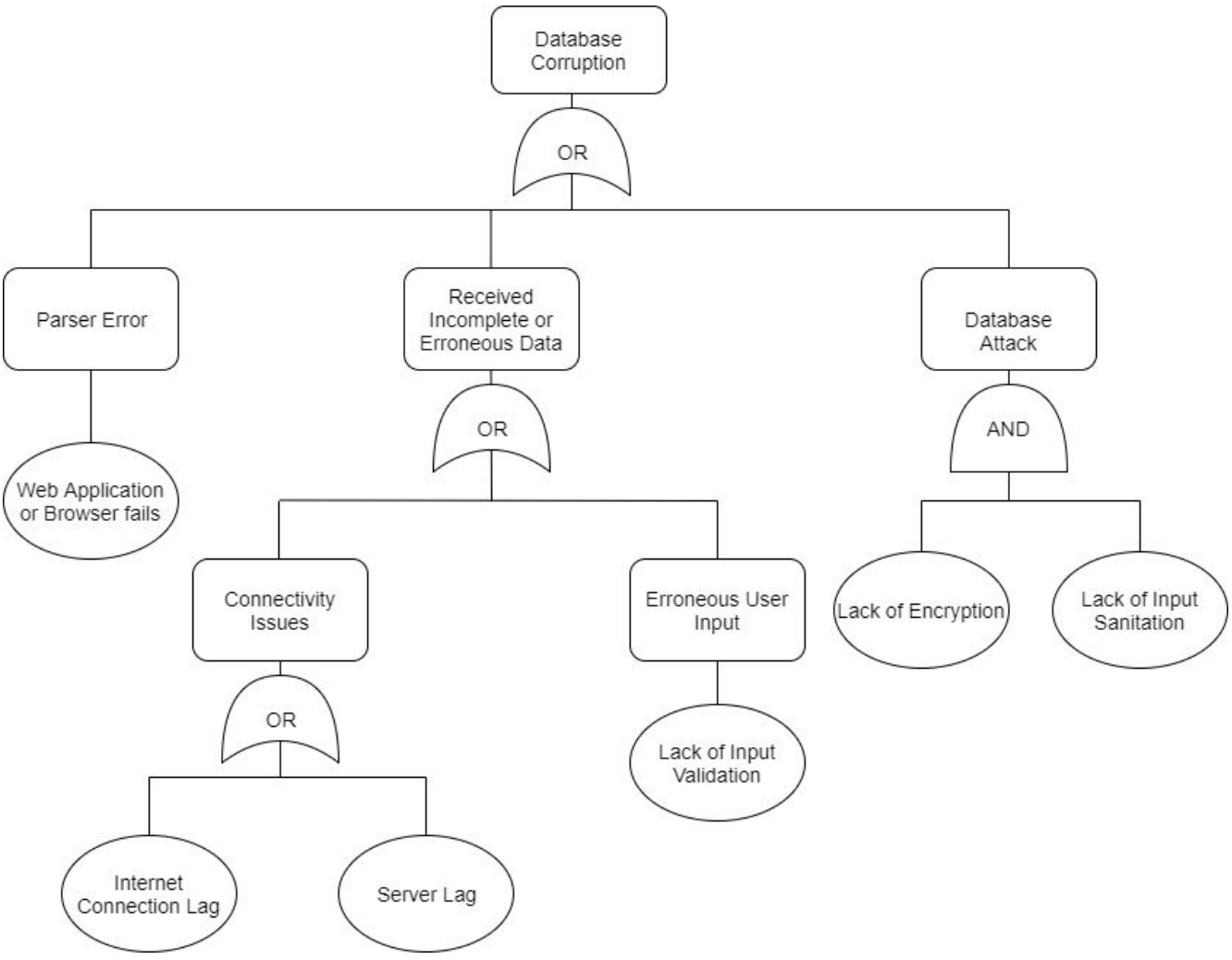
Reusability

- Architecture 1 and Architecture 2
    - Both architectures are very strong in regards to reusability. Since the product is a web application that allows purchasing items and shipping them, we could reuse the backend and replace the frontend for any number of e-commerce web applications.
- Architecture 2
    - As this architecture is made up of many microservices, Architecture 2 has an edge over Architecture 1 in that we could easily reuse individual microservices in other similar, but unrelated applications. For example, once we've developed an authentication service we could easily pull it into almost any other application. The order service could also be pulled out and put into a completely different e-commerce application.

Failure Modes

Database Corruption Failure

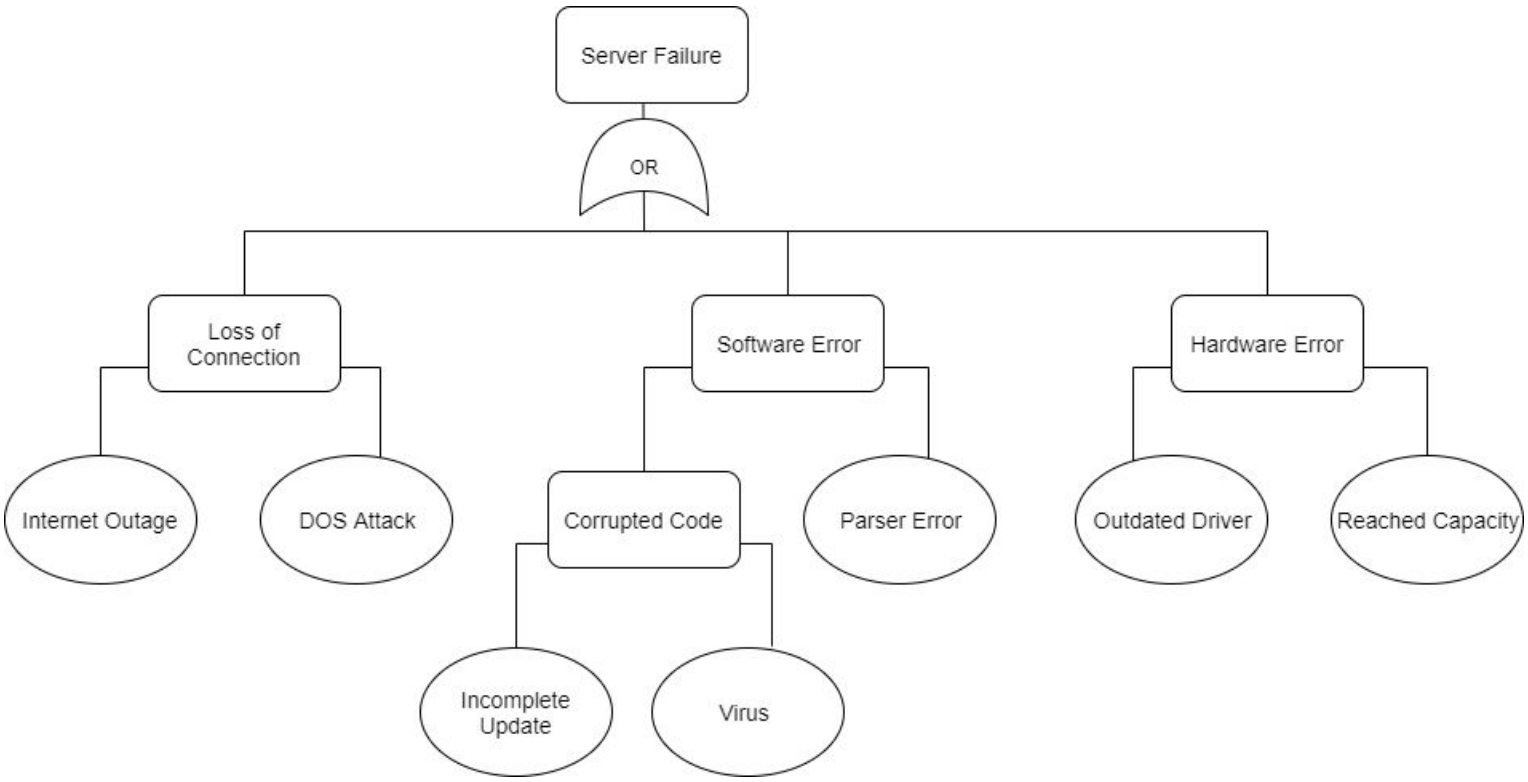
Fault Tree



The web-application architecture is more prone to this database corruption failure since the application frequently interacts with the different databases, sending or requesting data. The increase in information exchange opens up the possibility of sending inaccurate data (types) and opening up vulnerable channels for hackers or targeted attacks.

Server Failure

Fault Tree



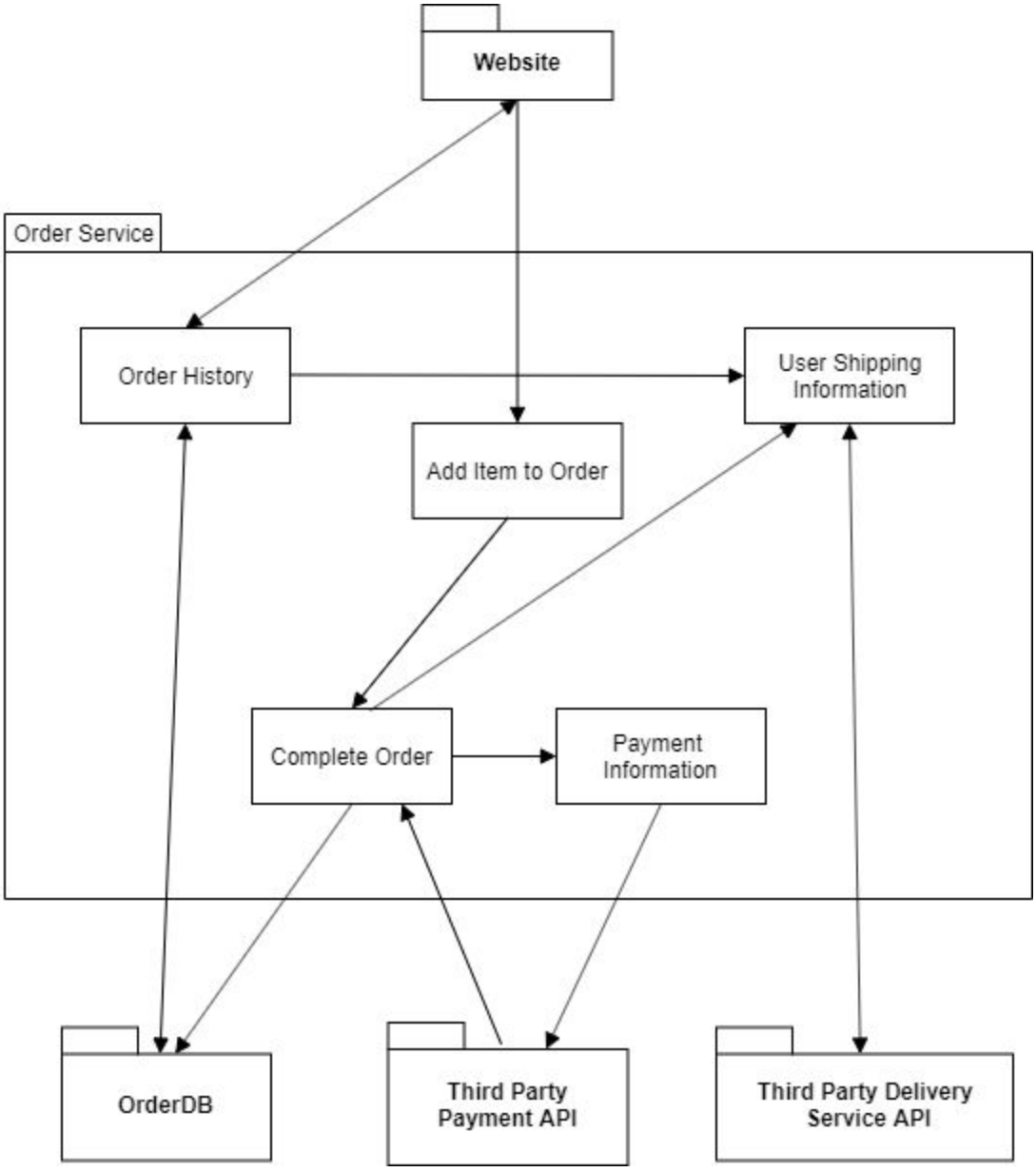
The web-application architecture is more prone to server failure. Since the web application frequently sends data requests to the server, when the server fails, the application would not be able to deliver any of the promised services to the user or be able to send any information to third-party. It would render the application useless until the server is restored.
The layered architecture with microservices is significantly more resilient to this sort of failure. If one service fails (e.g. the order service), clients would still be able to use other services and use the application with partial functionality (log in, view meals, add to shopping list, etc.).

Architecture Decomposition

We've chosen architecture #1, the server-heavy for further decomposition and validation.

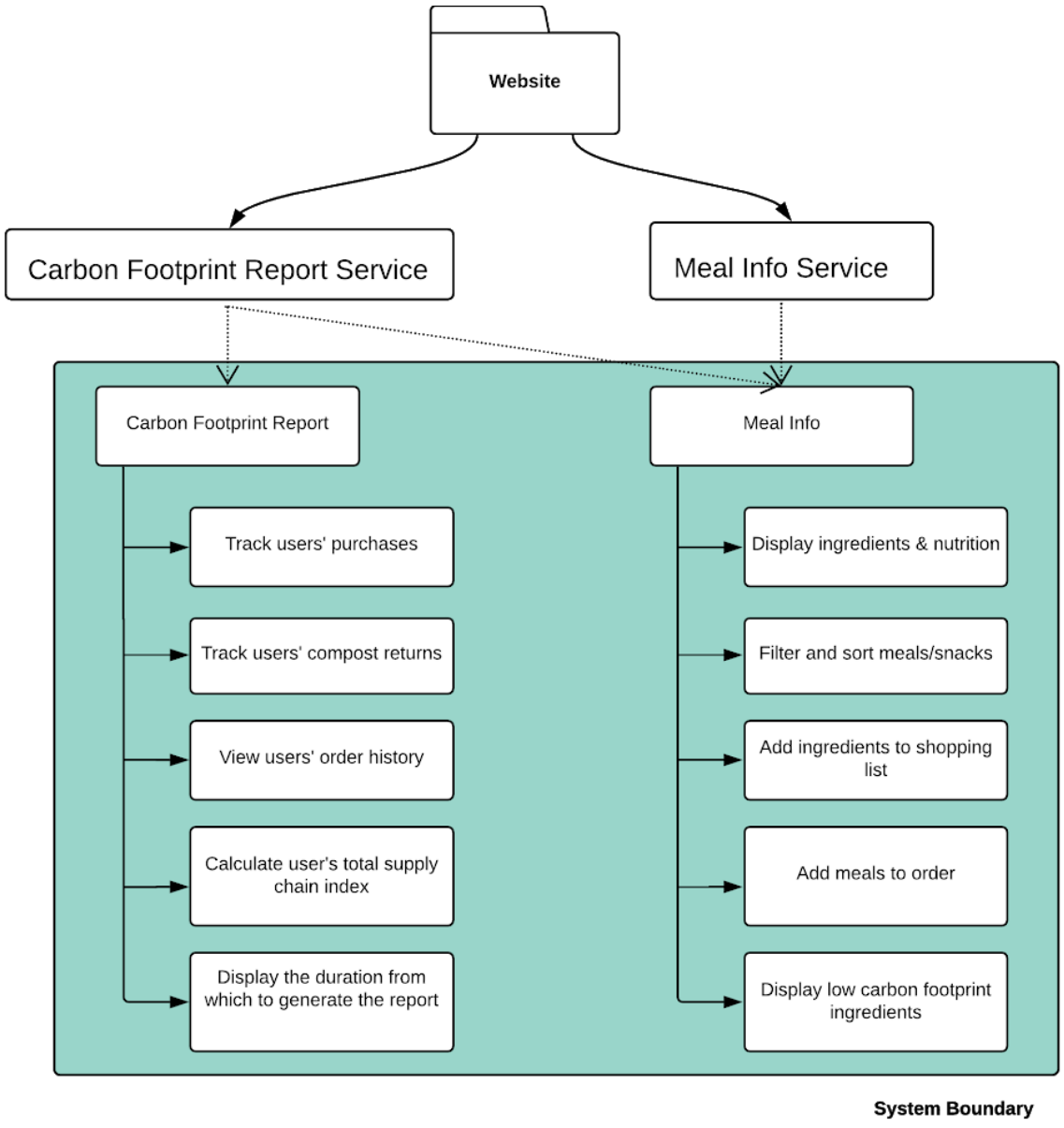
Important Element 1

Order Service Lower-Level Dataflow Diagram



Important Element 2

Carbon Footprint Lower-Level Dataflow Diagram



Architecture Validation

Use Case 1: Viewing and scheduling curated meals and snacks

- With the chosen architecture, authentication will take place when user credentials ("login info") are submitted through the client to the web server via the web site. The server will then validate those credentials against the UserDB. If the user credentials are correct, a session will be created on the server for that user and the session id will be stored as a cookie in the user's browser.
- The user submits a GET request for the meals page to view the list of meals. Authorization is performed by the web server on receipt of this request to verify that the user is logged in with a valid account by checking for a valid user session, created during the authentication step.
- The meals' name and rating info are retrieved from the MealDB by the web server. It is rendered as a web page to the user where they can browse and filter through all the available curated meals.
- The user selects the meal they want to view on the web page, submitting a request to the web server to get the page for that meal. The web server handles the request by querying the MealDB for the meal info, rating, and ingredients. This information is used to render the individual meal page for the user.
- The user selects the quantity of servings they wish to schedule and clicks the button to add the meal to their order, or the button to add the meal ingredients to their shopping list.
- The request to schedule the meal is sent to the web server.
    - If the request was to add the meal to the shopping list, the web server populates the ShoppingDB with an entry for the user with the corresponding meal and ingredient info.
    - If the request was to add the meal to their order, the meal is added to the user's "cart" in their session by the web server that will later be used to create an order, as specified in use case 2.

Use Case 2: Ordering ingredients to prepare scheduled meals

- As discussed in the use case 1 walkthrough, the web server performs authentication, and authorization is performed with any request to ensure the user has a valid session.
- From the user's "cart" page, the user will enter delivery, billing and payment information, and click the button to "Place Order". This submits a request with the order, payment, and delivery information to the web server.
- Since our chosen architecture is server-centric, the web server will proceed to process all parts of the order using the information from the user request:

    - Payment information and the order amount from the user request is sent by the server to the third-party payment service API for processing.
    - The third-party payment service returns a response indicating whether or not the transaction was successful. If the payment was approved, an entry with the order information is persisted in OrderDB by the server. The entry in OrderDB is updated with the success response from the payment processor, and the web site shows a completion screen to the user.
    - Order information and delivery information is sent to the third-party delivery service API for shipment. The server receives a tracking number from the third-party-delivery service.
    - The entry in OrderDB is updated with the tracking number from the third-party delivery service. After the order is shipped, the server will be able to use this tracking number to periodically query the third-party delivery service API for updates. If there are any updates, the server will send notifications to the user through the website or application.

Use Case 3: Track supply chain waste index and carbon footprint

- As discussed in the use case 1 walkthrough, the web server performs authentication, and authorization is performed with any request to ensure the user has a valid session.
- Since our architecture is server-centric, the server is able to communicate with the OrderDB, MealDB, and the CarbonFootprintDB without any additional interaction from the client.
- After an order is placed (use case 2), the server will add an entry for the user in the CarbonFootprintDB. To do this, the server will query the OrderDB for the meals that were a part of that order. Then, it will query the MealDB for the carbon footprint values associated with those meal ingredients. Finally, it will populate entries in the CarbonFootprintDB with the date, the meals ordered, and the carbon footprint and supply chain waste index values associated with those meals.
- From their account page on the website, the user selects to view their “Personal Waste and Carbon Footprint Report.” The next page displayed requests the user to specify the duration from which to generate the report. This submits a request to the server to generate the report for that date range.
- The server takes the request and queries the CarbonFootprintDB for all entries for that user within the given date range. It uses them to calculate the user’s total carbon footprint and waste index, and renders a web page with this value as well as all the meals the user has ordered with their individual associated carbon footprint and supply chain waste index values.

Implications

- As seen in the use-case walkthroughs for our server-centric web application architecture, the most significant benefit is that the server has access to all components of the application. This allows it to directly query the databases for all the data it needs to perform any computation, render any websites, and it is a single point of communication for both the user and for any third-party services. Also, being a web application, the user does not need to install any client to their computer or mobile device, and they can access the service from anywhere.

  That said, this introduces a serious risk. If our server is unavailable for any reason, including loss of network connectivity and server failure, the entire application is down. This includes not only all the back-end services, but the web site itself will also be unavailable to the user so they will not be able to see their scheduled meals, recipes, shopping list, and more.

  One potential solution for this would be to introduce a client application that has some sort of caching functionality for offline use. The client application will cache information about the user's scheduled meals and shopping list. When the web server is offline, the user will not be able to view new meals or make any orders; however, they will be able to see their shopping list so they can still go out and buy ingredients on their own, and they will be able to see their scheduled meal information so they can prepare the meals. From the user perspective, this enhances the usability and reliability quality attributes as the core service will still be available to them in the event of some major back-end system failure.

- Going through the use case 3 walkthrough, it was apparent that our server has access to all the information it needs to generate the Food Waste and Carbon Footprint Report by querying OrderDB and MealsDB. Adding records to the CarbonFootprintDB at the time of order might improve performance when user's request to generate the report, but it might also decrease storage efficiency as we are effectively duplicating information we already have in a separate database. A solution might be to just have the web server query the OrderDB and MealsDB after receiving the user request and use that to generate the Waste and Carbon Footprint report real-time. This will also simplify our architecture as we will not need a separate CarbonFootprintDB.

Team Member Contributions

Here is a list of team member contributions for HW3:
All - Communicated on Slack throughout the week and added contributions to Google doc
Brian Shim - Layered Architecture Data Flow Diagram and Carbon Footprint Decomposition Dataflow Diagram
John Casey III - Layered Architecture Data Flow Diagram, Web Application Architecture Data Flow Diagram, Architecture Validation, Implications, Key Quality Attributes, Server Failure description
NianJun Shi - Web Application Architecture Data Flow Diagram, Failure Mode Diagrams and Descriptions
Raymond Lieu - Web Application Architecture Data Flow Diagram
Todd Radin - Key Quality Attributes, Order Service Decomposition Dataflow Diagram