Module 1: Pipe and Filter Questions:

1. For what kinds of applications and in what scenarios is the pipe and filter architecture a suitable choice?
The pipe and filter architecture is useful when a series of distinct transformations need to be applied to a specific starting input in order to reach a certain output format (e.g applying transformations on an image, shell commands, etc...)

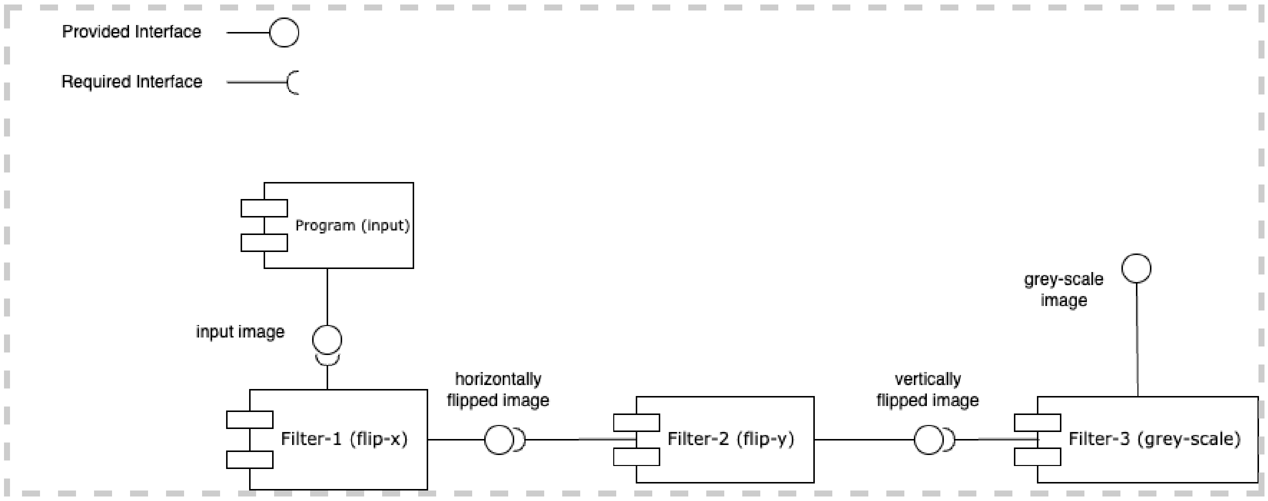
2. How does the Pipe and Filter architecture promote modularity and reusability in software design?
In the Pipe and Filter architecture each data operation is broken down into a self-contained filter (separation of concerns), this separation of components promotes high modularity. In addition, filters expect a well-defined format of input and have no dependencies upon other filters; this highly cohesive and decoupled nature of filters allows for high reusability within the system.

3. In the context of Pipe and Filter, what are "filters" and what purpose do they serve?
Filters are self-contained operations that operate on an input in order to produce a specific output which can then be used with further filters (e.g flip an image). It is worth noting however that filters are independent from each other and should only perform one task.

4. How are "pipes" used to connect filters in the Pipe and Filter architecture?
Pipes act as the channels that connect communicating filters together. A pipe contains data (preferably light payload) that can be transmitted onto another filter for further transformations. It is worth noting that pipes are usually unidirectional for performance purposes.

5. What are some advantages of using the Pipe and Filter architecture compared to other architectures?
The Pipe and filter architecture usually achieves high performance while also maintaining a high sense of modularity due to separation of concerns. In addition the pipe and filter architecture is monolithic in nature which means it can be simpler and easier to understand as opposed to other architectures. Furthermore, the simplicity and lack of complexity of this architecture can result in lower maintenance and build costs.

Component Diagram:



Module 2: Web Services Questions:

1. In what scenarios is Web Services architecture a suitable choice for building applications?

First, we want to acknowledge that there are three main roles that constitute the baseline for the architecture's interactions: service provider, service registry and service requester. The web service provides functionality to the client while letting the server to host by sending responses to the client- Although this layered architecture is generally slower than monolithic architectures, it enhances modularity in the sense of grouping components in modules while incorporating the principle of separation of concerns (presentation tier, logic tier, database tier). This entails that the web-service architecture is tailored for systems whose specifications are heavily centered around the feature of interactions manifested in the form of commands sent from one end to the other. Despite the slower execution that comes with this architecture, it allows for dealing with larger quantities of data with respect to the data-management tiers (logic and database tiers).

2. How does the web service architecture contribute to better scalability and maintainability of applications?

- **Maintainability:** As stated in the question above, the enhanced modularity allows for grouping of components in modules, which makes the code more organized, and organized code implies an existing ease of updating as well as developing it for prospective unpredictable changes. To be more specific, the separation of concerns of the tiers makes it easier to identify exactly where in the code errors occur, speeding up the process of debugging (the tiers serves as a deconstruction of the main tasks that satisfies the overall operational objectives of the architectural system)

- **Scalability:** Once again, the outcome of the satisfactory modularity is to be capitalized upon. Thanks to the separation between client and server, data migration from one server to another is straightforward. Hence, the performance of the application is not relying on one single server. Consequently, the ability to cope with growing numbers of users in the absence of deterioration of performance is based on the fundamental concept of data migration of high availability

3. Elaborate on the security aspects of Web Services architecture. How can it ensure data protection and maintain the integrity of sensitive information in a distributed environment?
This architecture makes it easy to distinguish between the users, by applying an authentication system that serves as an identifier that confirms that the 'correct' individual is logged in on the associated account (on the client). Through the lens of the developer, the web-services architecture promotes a simplicity in restraining the available features for all clients (this includes creating and storing data through the logic-tier in the database, but also accessing it). With that said, developers possess the ability to modify the conditions that needs to hold true in order to access a specific piece of data (in most cases this check is done by reading the identifier of the client or comparing hierarchical ranks tied to the account)

4. How do RESTful APIs enhance the effectiveness of Web Services architecture and enable stateless communication between clients and servers? Provide a real-world example of RESTful API implementations and the advantages/disadvantages of being stateless.

RESTful APIs enhance the effectiveness by simplifying the communication between web applications while synchronizing and integrating with other applications to attain interoperability. The stateless technology is to be observed in the communicative methods where the server completes requests from the client in an asynchronous fashion that disregards the states of the previous ones. This quality favors the quantity of manageable tasks that can be completed within a short period of time. On the other hand, the disadvantage that comes with this is that web services need to get additional redundant information in each request to identify the client's state (each call contains the data necessary to complete itself, since the foundational building block for stateless technology is that each call can be executed and completed independently).

A real-world example of a RESTful API is Twitter API.

5. If you compare Web Services architectures to monolithic architectures, in what scenarios might Web Services be a preferred choice?
Monolithic architectures aren't sufficient when it comes to the attainment of the following qualities:

- Interactive system
- Expansive/Comprehensive systems involving a large quantity of components

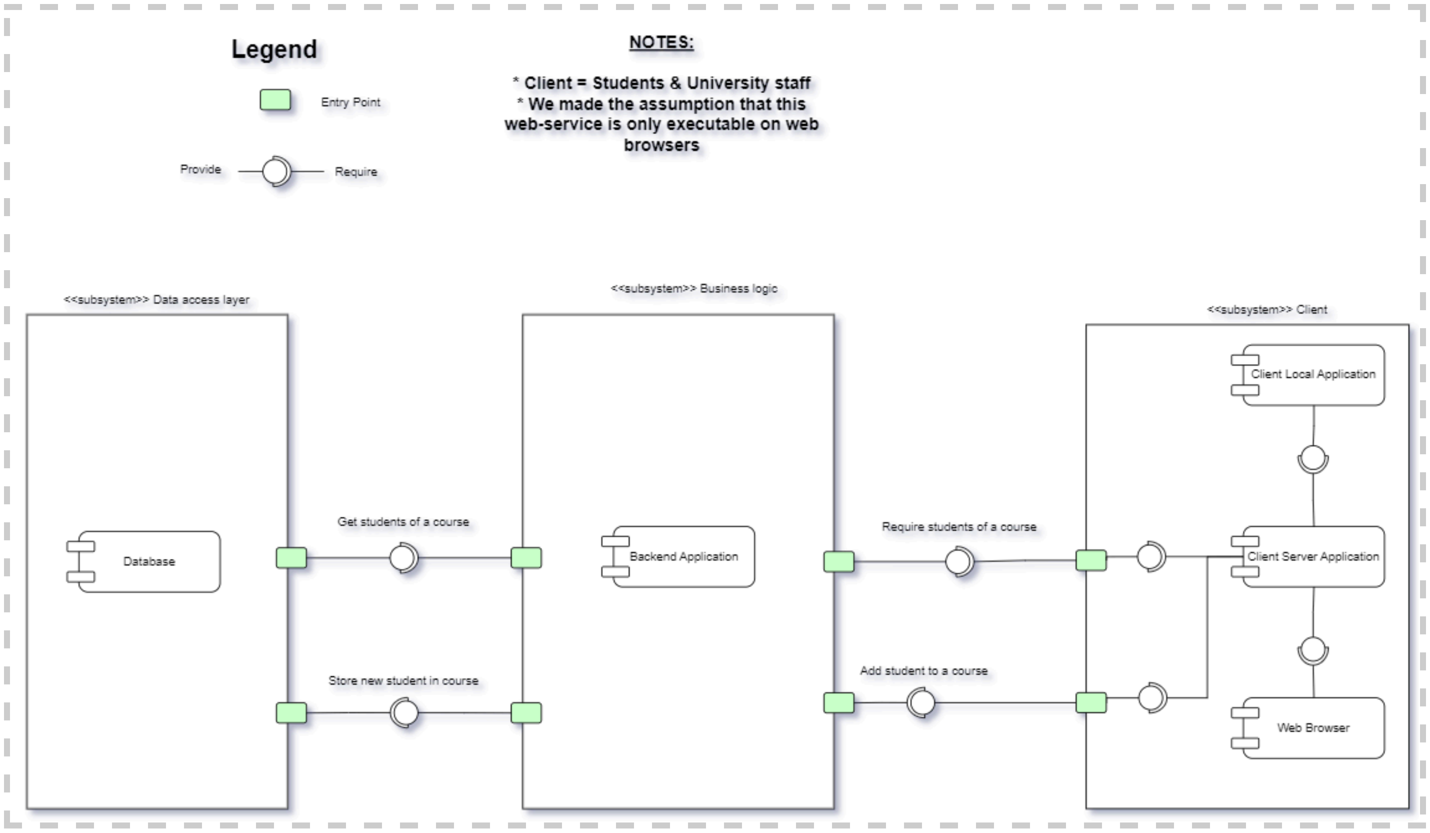
Monolithic is good for simplicity, and when many components require continuous monitoring, the system finds itself in a state of computational overhead. Consequently, we find it reasonable to assert that if there is potential for the system to expand and where there is an uncertainty in the sequence of execution of requested operations (interactions), the web-service architecture is better suited. In the case where you want to develop and deploy the architecture in a short period of time, monolithic architecture is the way to go.

6. Compare Web Services architectures to service-oriented architectures. (How) do they encourage the creation of reusable components and services across different applications?

- **Service-oriented:** A collection of distributed components that provide and consume services, where the components consume services from their shared user interfaces. This means that when intending to create extended functionality, we instantiate a realization for that component to the user interface

- **Web services:** Similar to the reusability of the prior architecture, we can create shared user interfaces and apply them to a multitude of components. However, the thing that stands out for this architecture is that the separation of concerns contributes to modularity and modifiability. Each specific tiers' area of functionality or missions greatly diverge, meaning that there is a structure to be followed with predefined principles for each tier. In that way, we cannot generally reuse a component across the entire architecture unlike service-oriented, but rather constrain it to one tier.

Component Diagram:


Legend
Entry Point
Provide
Require
NOTES:
* Client = Students & University staff
* We made the assumption that this web-service is only executable on web browsers
<<subsystem>> Data access layer
<<subsystem>> Business logic
<<subsystem>> Client
Database
Get students of a course
Store new student in course
Backend Application
Require students of a course
Add student to a course
Client Local Application
Client Server Application
Web Browser

Module 3: Microservices Questions:

Part 2 - Data Collection

Feature #1: Delete order by id

- The number of modules affected are 2. OrdersUI and MSClint are existing modules that were affected by addition of the deleting services.
- In order to enable DeleteServices we had to create two modules. Those modules are DeleteServices.java and DeleteServicesAI.java.
- Time to have deleteServices working was 3 hours.
- For the DeleteServices to work we had to access the orders collection (table) so that the users can delete an order with specific order_id from the Collection.

Feature #2: User Registration and Authentication

- The number of modules affected are 2. OrdersUI and MSClient are existing modules that were affected.
- In order to enable AuthorizationServices we had to create two modules. Those modules are AuthorizationServices.java and AuthorizationServicesAI.java
- Time to have Authentication services working was 3 hours.
- For the AuthorizationServices to work we had to create a new collection (table) so that the users can be added or identified to and from the collection.

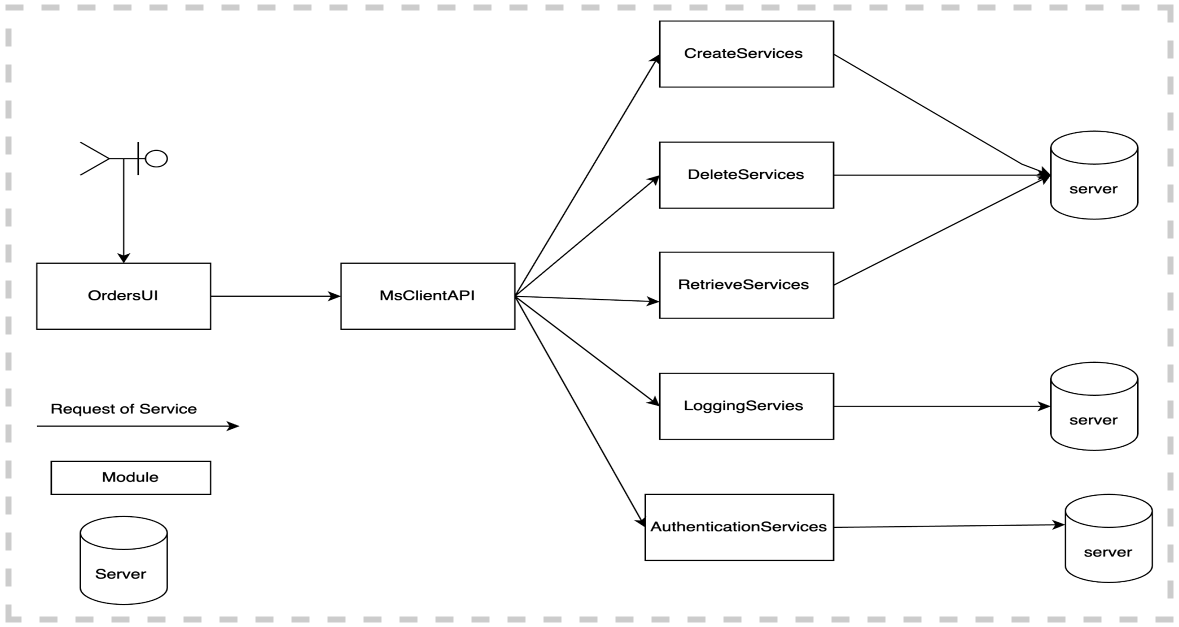
Feature #3: Logging Activity and Faults

- The number of modules (class) affected are 2. OrdersUI and MSClint are existing modules that were affected by addition of the deleting services.
- In order to enable LoggingServices we had to create two modules. Those modules areLoggingServices.java and LoggingServicesAI.java
- Time to have deleteServices working was 3 hours.
- For LoggingServices to work we had to create a collection(table) in the database that stores all activities and time when a service was utilized and if an error happened we had to specify the error and the affected part of the system. This will allow developers to know if there is any issue in the system when and where it happened.

Part 3 - Architectural Discussion

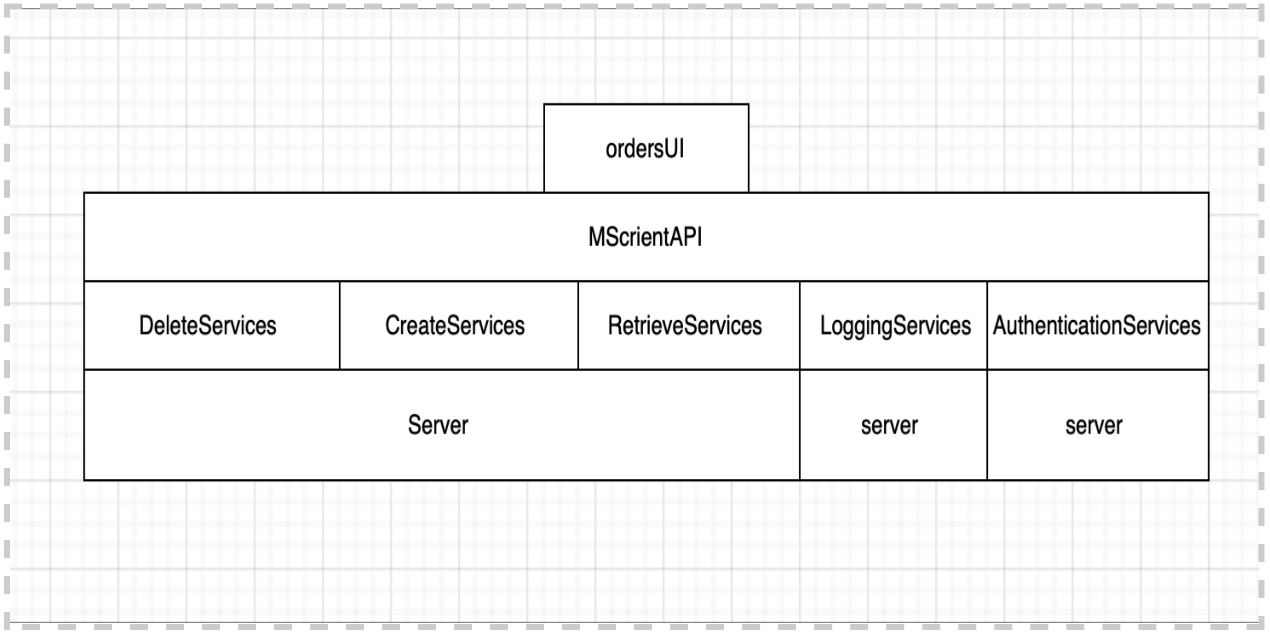
<u>Module View of microservices architecture of the system.</u>

The diagram below shows a module view of one of the chosen architectures for the system (Microservices architecture). The view shows a user who requests different services using OrdersUI. Then OrdersUI calls MSClientAPI that requests the specific service.

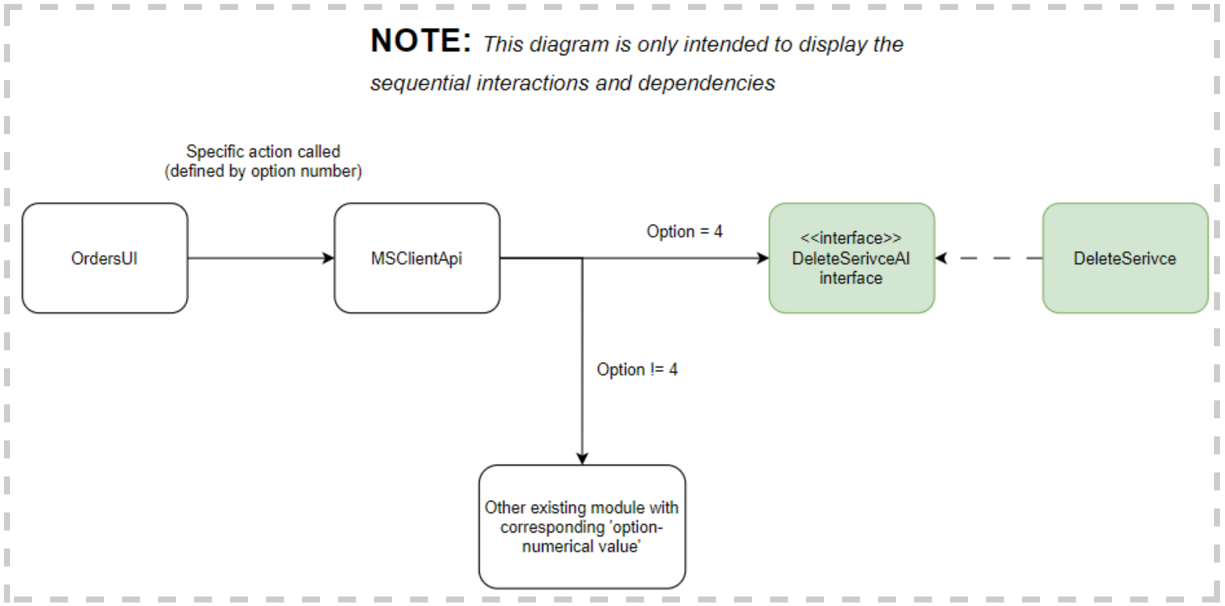


Module view of the layered architecture structure of the system.

The diagram below shows the module view for the chosen architecture of the system (layered architecture). The module can only communicate with a module below or above it



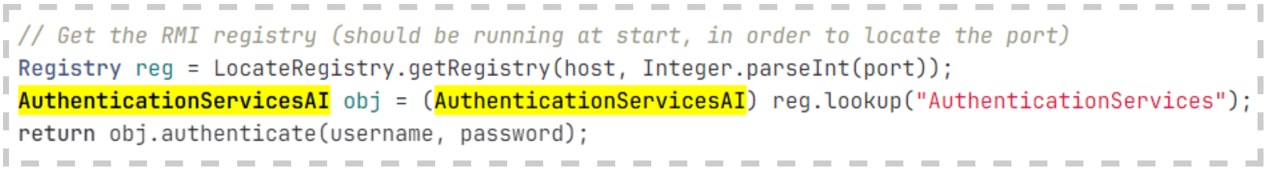
Feature 1:
The design decisions that followed with this feature are as demonstrated below, where the green boxes represents the creations of classes/interfaces and their interaction with the other modules in the architecture:


NOTE: This diagram is only intended to display the sequential interactions and dependencies
Specific action called (defined by option number)
OrdersUI
MSClientApi
Option = 4
<<interface>> DeleteSerivceAI interface
DeleteSerivce
Option != 4
Other existing module with corresponding 'option-numerical value'

The picture above entails that the architectural impact when adding this feature is an interaction between the MSClientApi, the new interface and the independent service itself. In order to make this executable in the coding department, an if-statement for the selected option (in OrdersUI.java) must connect it to the MSClient, which in turn forwards a connection to the service where the code for the new feature is located. With that said, the scalability is favorable, as we can scale services independently. Since all retail stores have the ambition to grow, That is the ambition of the system because it will be supporting a business that aims at growth. These two architecture styles allow decoupling of the system which decreases dependence hence increasing modifiability quality attribute. This quality attribute allows easy expansion of the system in future.

Feature 2:
The modifications required to implement this feature can be seen in OrdersUI, MSClientAPI. In OrdersUI we connect the intermediary MSClientAPI object ‘api’ in option 1 for login and option 2 for register. Inside MSClientAPI we changed ‘authenticate()’, ‘makeUser()’ and ‘reportLog()’. In accordance with the defined template in the interface AuthenictationServicesAI, we coded functionality for ‘authenticate()’ and ‘register()’, which yields great returns in the metric of modifiability, with polymorphism as the tactic (AuthenticationServices realizes AuthenticationServicesAI), where possibilities for adding classes of similar functionality are made possible. To be more concrete, this

can be observed in MSClientAPI, where we use the highlighted interface rather than the specific class in ‘authenticate()’:



```java
// Get the RMI registry (should be running at start, in order to locate the port)
Registry reg = LocateRegistry.getRegistry(host, Integer.parseInt(port));
AuthenticationServicesAI obj = (AuthenticationServicesAI) reg.lookup("AuthenticationServices");
return obj.authenticate(username, password);
```

Feature 3:
We added option 5 in OrdersUI, which we connect with LoggingServicesAI and LoggingServices in MSClientAPI. LoggingServices is an independent and decoupled service in the sense that its contained content can be executed on its own. Due to the decoupled code, the quality attribute of availability emerges. The degree to which the system is in an operable state and can function according to its intended specifications is elevated as a result of the overall structure of the architecture, where each of the other services fulfills the same requirements concerning independence. Thus, the entire system doesn’t shutdown if one service fails to perform its given tasks (fault isolation), meaning that if a crash occurs, the multitude of it is not that serious, inducing a less severe state of downtime that is easier to recover from.