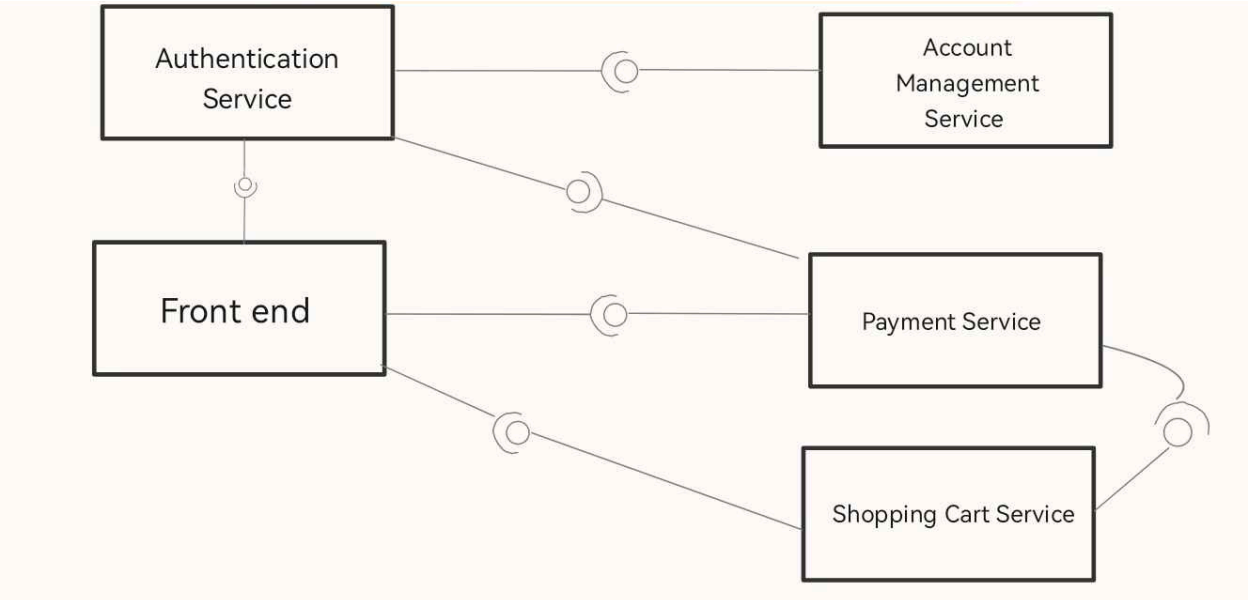
Decentralised Online Shopping

1. Compose a system with a maximum of 6 services/components which meet the security requirement via a structural diagram. Elaborate why your composition can achieve the requirement. (Hint: Component diagram is structural diagram).



Where security concerns are needed

- Authentication Service (Login)
- Payment Service

Why This Composition Meets Security Requirements:

- **Peer-to-Peer** Communication reduces the risk of centralised attacks by ensuring that data is only shared directly between the necessary services. Ex. Front-end will communicate directly to Authentication Service without going through the mediator. Increased development time and complexity compared to a mediator based communication.
- Using secure authentication mechanisms like OAuth tokens when authenticating users for secure message exchange, ensures that only authenticated and authorised users can access sensitive information.
- Users will have to get authenticated once again during payment (Via mail verification)
- Peer to Peer is also well suited for implementing End-to-End Encryption between each service such as authentication -> user management. By encrypting the messages before they are sent, even if an attacker manages to intercept the communication between services (e.g., between the Authentication Service and User Management Service), they won't be able to decipher the content of those messages.

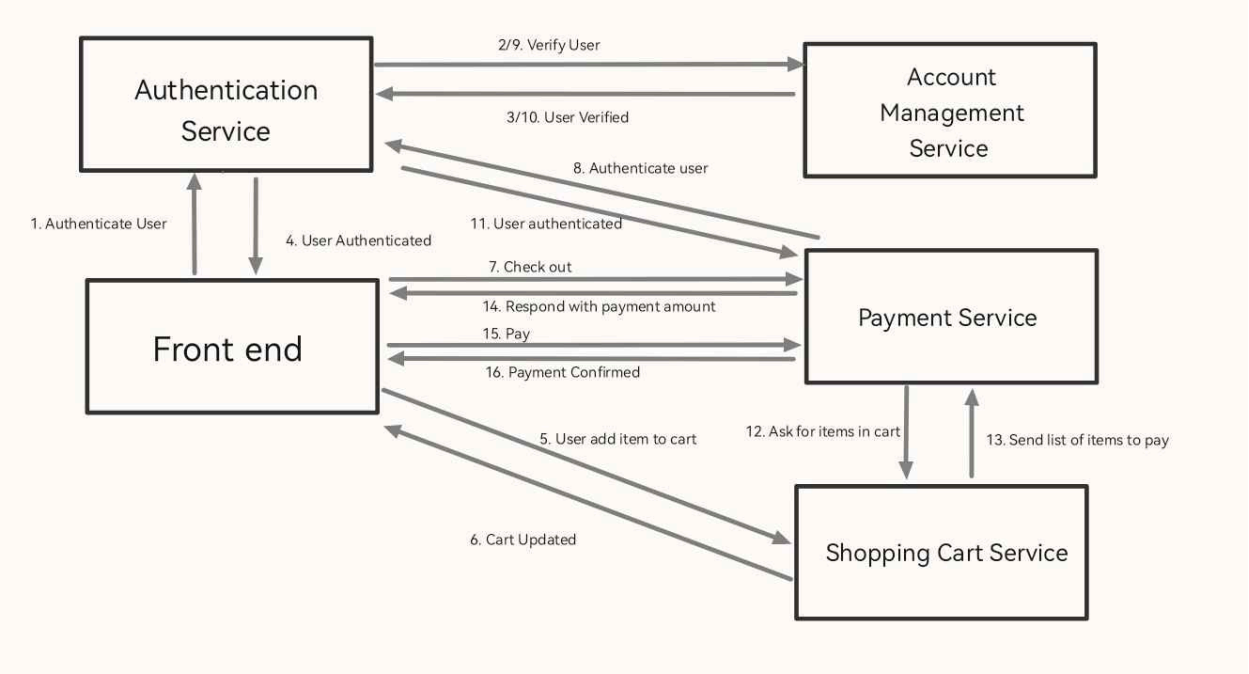
2. Rationalise whether the system should use REST, SOAP, ESB or any combination of them to help in achieving the above security requirement. Illustrate a case where the above communication techniques can help in the requirement with a behavioural diagram. (Hint: Communication diagram and sequence diagram are considered behavioural diagrams)

    SOAP will be used on the payment service since security is our major concern. SOAP offers better security than REST, it provides built-in security standards called WS-Security, which enables the usage of security tokens such as X.509 certificates, SAML (Security Assertion Markup Language), and Kerberos tokens. SOAP messages can be encrypted and signed so that only the intended receivers can read them and verify their integrity.

    SOAP is slower than REST due to sending communications in XML-based formats. Each item of data is surrounded in tags, resulting in a larger message size than more compact forms such as JSON utilised by REST. XML parsers must also handle sophisticated data structures, namespaces, and possibly enormous documents, necessitating more processing power and time. The security will justify the overhead since it is our major concern.

    At first I wanted to use REST for communication on services other than payment due to REST having better performance and is faster since there's no XML parsing and no complex standards. However, REST is stateless and may not keep the information on the previous request. This is bad for the shopping app that needs to keep the user's preference, being stateless means requiring the client to send all necessary information with each request, complicating the management of shopping carts and user sessions.

    So I'm going to use SOAP on all services and also won't use an ESB. Since my system only has SOAP and no REST, all services will send requests in SOAP and the services will have an XML parser to handle and interpret SOAP messages, enabling seamless interaction between services. This is to avoid the security vulnerability and overheads that comes with an ESB due to its centralised nature, no single point of failure. The downside of not using an ESB is the complexity in designing the system and when integrating in new components.



3. Elaborate your deployment decisions and testing strategies which support the above security requirement.

Deployment decisions

- Container :
  - The system will be deployed using Kubernetes to host all containers. All services will be containerized for the following **security reasons**:
    - **Isolation:** Containers create a lightweight, standalone, executable package that can be easily run and executed. Each service runs in its own container, which limits the potential impact of a **security breach** to just one container.
    - **Immutable Infrastructure:** Containers are immutable by nature. Once a container is deployed, it remains unchanged, which **reduces the attack surface** as there are no configuration changing
    - **Role-Based Access Control (RBAC):** Kubernetes provides RBAC, which allows you to define fine-grained access controls for different users and services. This helps in limiting permissions to only what is necessary, **reducing the risk of unauthorised access.**

Testing Strategies

System Testing(Security Testing) :

- Evaluating if the system has **incorrect WS-Security configurations** in SOAP services.
- Example, using tools like Burp Suite to craft and send malicious XML requests. With correct configurations from WS-Security, the system should rejects unauthorised or malicious requests and be safe from the attack

Integration Testing:

- During integration testing, check the **security of interactions** between services and that the messages are properly delivered and not intercepted by Man-in-the-middle attacks. Tested by using tools like bettercap to simulate MITM attack and ensure that the outcome

Unit Testing:

- Focus on individual components in isolation. For example, testing if the function that hashes the user's password(Like werkzeug library used on my group's project) properly hashes the password by checking the result password stored in the database.