Problem Statement for a Mock Exam: Hybrid Digital Wallet

Following reports that Donald Trump vows to stockpile Bitcoin if he becomes President of the United States again, the bitcoin's value has increased by 5% toward its record high. This situation also attracts the concept of "stablecoin," which had waned since Facebook proposed "Libra" a few years ago.

In your role as a junior developer at a FinTech firm, you have been assigned to design a digital-wallet prototype that can handle both traditional and cryptocurrency transactions, similar to TrueMoney Wallet. The main requirement is reachability; it is essential to ensure that the withdrawal and deposit systems are consistently available to users. Any downtime could result in missed opportunities and financial losses due to the commission percentage from each transaction.

Based on the above information, your tasks are the following:

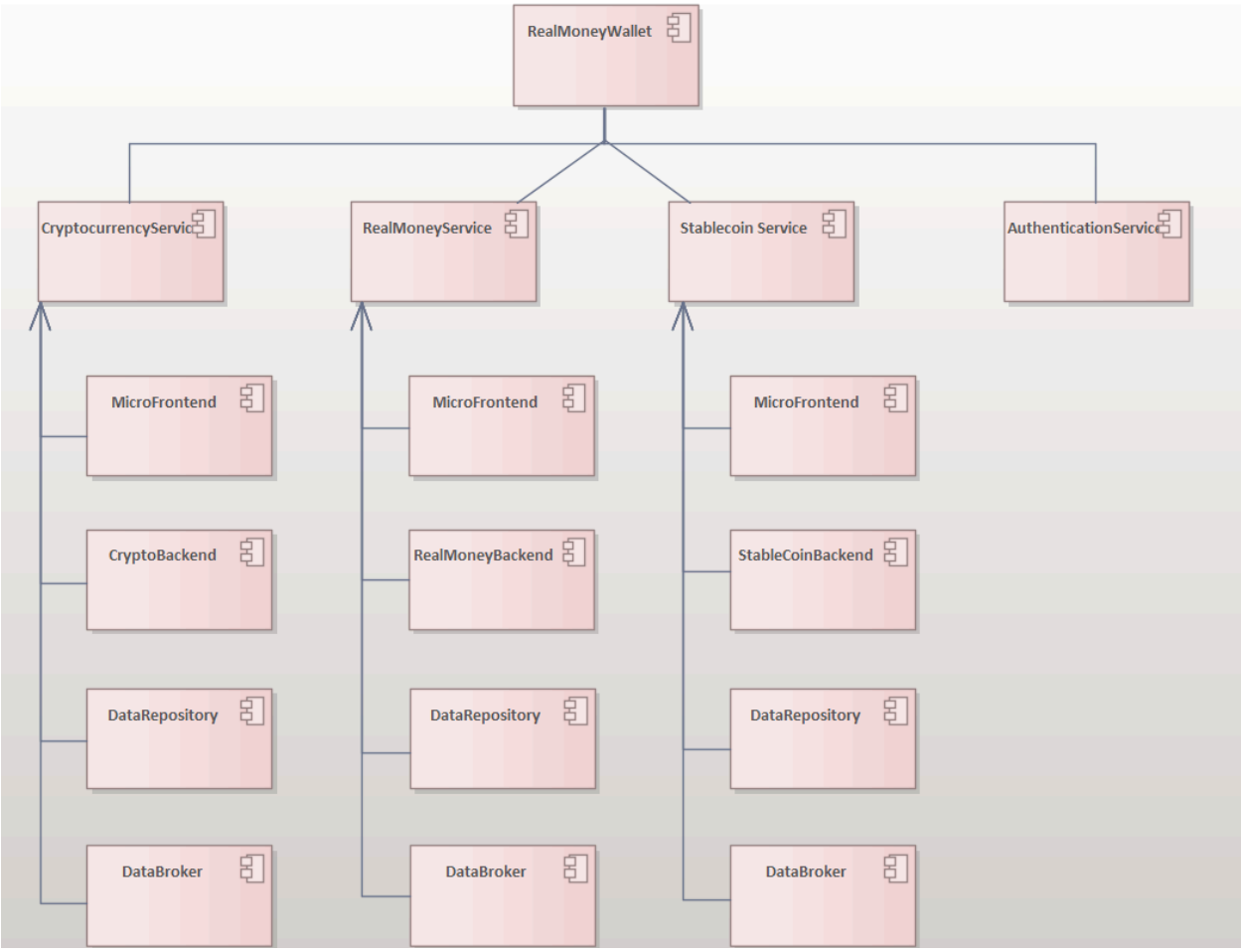
1. Compose a system with a maximum of 10 microservices/components via a structural diagram (e.g. component diagram). Elaborate how and why your composition can achieve the reachability requirement above.

2. Elaborate which microservices communication pattern(s) help achieve the above reachability requirement. Illustrate a case where one of the selected communication pattern(s) can help with the above reachability requirement via a behavioral diagram *(e.g. communication diagram or sequence diagram).*

3. Elaborate on your deployment decisions and testing strategies that support the above reachability requirement.

4. Explain how and why your cloud migration strategies support the above reachability requirement.

1. Decomposition of RealMoneyWallet System :



The composition for RealMoneyWallet System is designed to support the born to grow and born to die principles. The services are separated into several components that handle each money type and balances. This will allow the components to be developed and scale independently, this also means that they do not depend on each other. If developers(me) want to add new components, they can easily connect them as a new component, making the system growable. They have a separate data repository, separate frontend, separate backend to handle each service individually. Also without esb or mediator, the system reduces the dependency on a centralized component.

Since the services are independent, the failure of one service shouldn't affect other services. For example, users will be able to do a transaction on cryptocurrency separated from real money transactions. If the cryptocurrency service is not functioning properly, users will still be able to reach / use the other two kinds of transactions. This aids the reachability problem statement that requires the services to stay operating consistently even if the other services are down.

The system is also designed to use redundant data repositories or data replication for restorability in the system. Each post request will ask the service to save the data into several databases that are stored on separate containers. In case of transaction failure, the other databases are still available. The multiple databases come with space related trade-offs. The replicates consume lots of space. The system will have to waste spaces instead of being able to use this space to expand(scale) to a bigger space. When the user base expand, it will be costly to upgrade this space, because the database size needed to increase will be * number of databases on the system. There's also a trade-off when migrating databases. The replicated databases may be from different database providers so when there are platform specific errors, the others will have chances of surviving. But again, this will also complicate things more when scaling or migrating the data repositories

The system's frontend will be constructed using flutter because their provider class made it easy to decompose the frontend and is event driven by nature, so it will be suitable with communication style in Q2 too.

2. Communication Style : Event Driven

I chose **Event Driven** because it's most suitable for the reachability problem statement. The asynchronous nature of Event Driven ensures that services remain functional even if others fail. The example of how event driven patterns may be used in the system is on the micro frontend. The provider class has a notify function that will notify all micro service components that are listeners to the provider class when there's an api call. The api calls are async so if one of the requests fails, the other will do their thing and not get interrupted. Flutter also made each frontend component separate from each other, so that it won't affect other components on the page. The main tradeoff of event-driven is the potential data inconsistency, or concurrency control problem that comes with the asynchronous calls if not managed properly.

There will still be request & response communication between the provider and the backend. The system has a separated backend to help support reachability, failure of one service backend won't drag others down. But the system will need to have some health check measures to make sure the failed backend won't stay in that state for long, since it's a system that is related to money and transactions. In the next question, I will use Docker, it provides a built-in heath check instruction to automatically monitor the health of a container, these can be configured in the dockerfiles of the container. The health check will happen in the specified intervals

I simply won't use through data because managing the database logic is gonna be hell and there is lots of overhead from having to store multiple databases already. If a transaction fails, rollback time is going to be the end of the system.

3. Deployment Type

The system is going to be containerized in a docker container then orchestrated with k8. The isolation nature of a container makes services remain reachable, when there's high demand on a particular container, other services are unaffected by sudden load increases. If I have high resources for the system, I can create many replicas(instances of each service that is going to be deployed) to lessen the traffic in each service type. Kubernetes supports automatic load balancing, distributing traffic across instances of each of the money service types. This will prevent the overload problem mentioned earlier and will decrease the chance of the system not being reachable. The trade-off is the complexity in creating a dockerfile for each container. There are going to be a lot of containers for each replicated database, each frontend and backends.

I'm using a multi repository with branching to make the git recoverable and if someone messed up, it won't affect the whole organization. Multi repo helps developer side reachability. Imagine in a mono repo, if someone who is responsible for job A pushes a broken code into the repo, the others(ones who have nothing to do with job A) will have problems running the system. So multi repo will help with updating codes in batches, while other teams will still be able to run the code in peace without interruptions from other teams. Letting each team member create branches and merging later will also make the commits not affect others. Also use git blame so you know when someone pushes a bad commit, so you can curse them. Also, when testing just your team's code, you won't have to pull other team's commits, which will be annoying. The trade-off is that each team has to meet regularly to discuss the work progress and plans, so that each batch of big updates will be able to work with the other team's code.

Integration Testing:

Test Case 1: Test database reachability when one database is disabled.
Type: Mock
Goal: Ensure that the application can still access the remaining databases when one database is down.
Input: Requests to read data from the databases, with one database simulated as disabled.
Output: Successful read response from one of the available databases.

Test Case 2: Test if the backend is still working when the other backend is down.
Type: Stub
Goal: Ensure that the service can still call apis when one of the other services is down.
Input: Try to send a postman predefined request(may be login, or making transaction) to a backend that should still be working.
Output: The backend is functioning normally.

Unit Testing:

Test Case 1: Test micro frontend reachability on component X in a flutter page when one of the other components is not working.
Type: Mock
Goal: Ensure that the component can still function when one of the other components is down.
Input: Simulated requests to component X while component Y is mocked to cause an error.
Output: Component X functioning normally, while component Y displays errors.

Test Case 2: Test if the service tries to recover when getting bad health check results.
Type: Mock
Goal: Ensure that the service starts to recover when it receives a bad health check result.
Input: Simulate a bad health check response in a service, track if recovery plan is working.
Output: Print debug when recovery plan starts running.

4. Cloud Migration Strategies

I could replatform to using a multi cloud instead of having all the services in the same cloud. Relying on different service providers will help distribute risks and aid reachability when one provider is down, the other services on other clouds will still work.

I would replatform my services to these four PaaS, Heroku, AWS Elastic Beanstalk, Google App Engine (GAE), and Microsoft Azure App Service.

I chose PaaS instead of IaaS or SaaS because we don't have to write our own backend. I can just use Firebase Authentication for authentication and Firestore to store data. PaaS also provides features for database replication, so I won't have to deal with that. The problem of set up complexity mentioned above will be taken care of too. The trade-off is the expensiveness of PaaS and being dependent on the service provider. If something happens to the provider's physical location, like earthquakes, or a situation like that time that Alibaba Cloud data center is burnt in Singapore, your system will also be down.