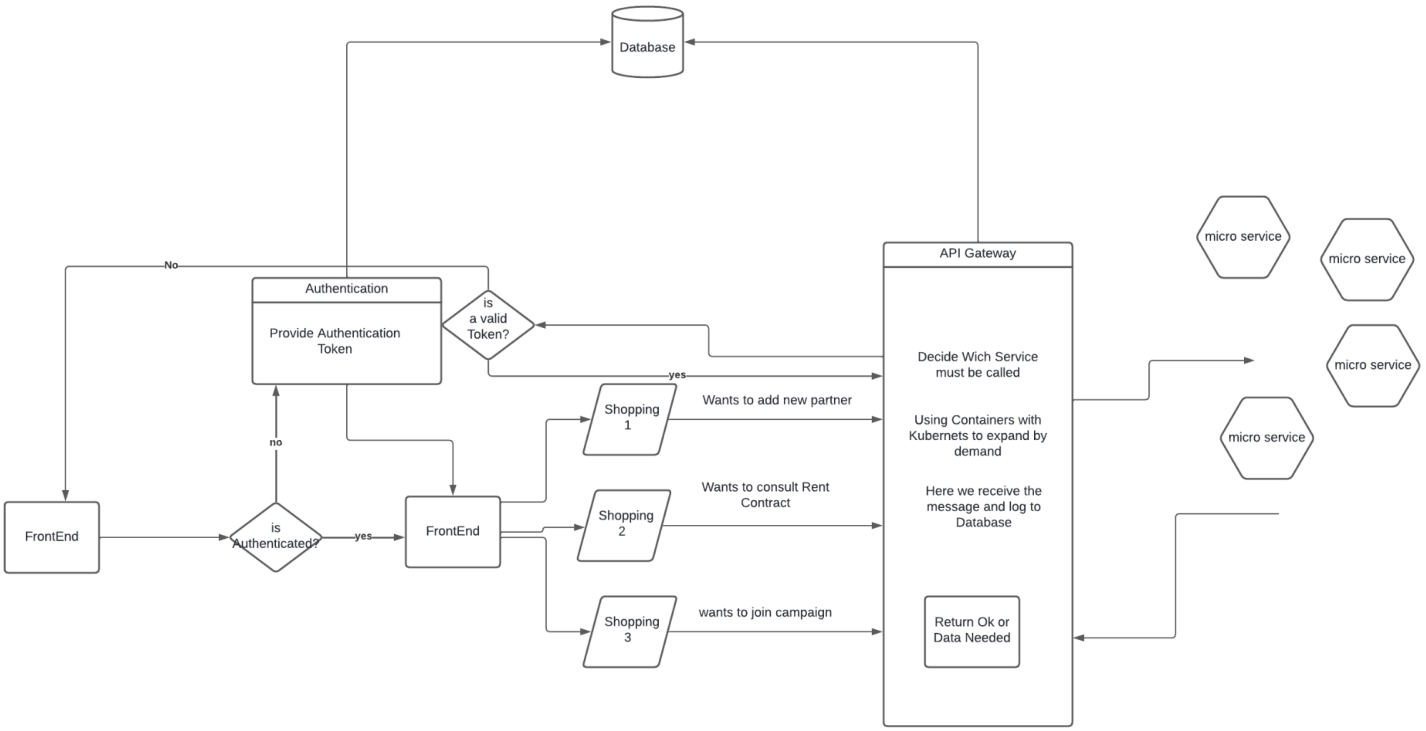
Architecture

After some research i found a standart microservices architecture



We have an authentication provider that works as a service to handle authentication and authorization (OAUTH2), providing a token to the frontend. With the token, we can access our API gateway, which receives every single message from the frontend. Based on the different types of submissions, it knows which service should be invoked next. It also logs the call in a log table.

This API must have a load balancer to handle the high demand of requests from the frontend. After delegating the request to the appropriate service,

Kubernetes can create containers when the services are invoked, reducing costs when the services are not being used.

Resilience Strategy

Circuit Breaker

The purpose of this pattern is to prevent the occurrence of network or service failures from cascading to other services, thereby improving the application's stability and resilience.

If we have a service B that is unavailable or responding with high latency, service A, which calls service B, must be able to maintain its availability in this failure scenario. To achieve this, it should perform one of the following actions:

Custom fallback: Service A should try to obtain the same data from another source, such as cached data. If this is not possible, a custom error message can be returned by the service.

Fail fast: If service A knows that service B is inoperative, it should not wait for the timeout, as this would lead to unnecessary consumption of infrastructure resources, degrading the application's performance. In this context, the application flow should be immediately diverted to a fallback routine.

https://learn.microsoft.com/en-us/azure/architecture/patterns/circuit-breaker

https://www.zup.com.br/blog/padroes-de-resiliencia-para-microsservicos

Strategies for Microservices Disaster Recovery

1. Synchronous Replication for Consistency: Implement synchronous replication for critical data to ensure consistency. While this might impact availability, it ensures that data across services remains coherent.
2. Asynchronous Replication for Availability: To maximize availability, use asynchronous replication. While this might result in some data inconsistency, it allows your services to remain operational during a disaster.
3. Monitor Workers for State Updates: Use monitor workers to periodically send state updates from each service to a master monitor. This approach provides a middle ground, allowing you to choose between consistency and availability based on your disaster recovery needs.