Architecture design document

OFS Platform

By Mohammed Al Harbi

Contents

1. Introduction

enterprise software it is important to design the software such that it can handle many users while ensuring that does not shuts down your application. In this research I will clarify the reason why such applications do not fit the typical monolith design. In addition, we shall highlight various different architecture approaches and pick the one we see fit

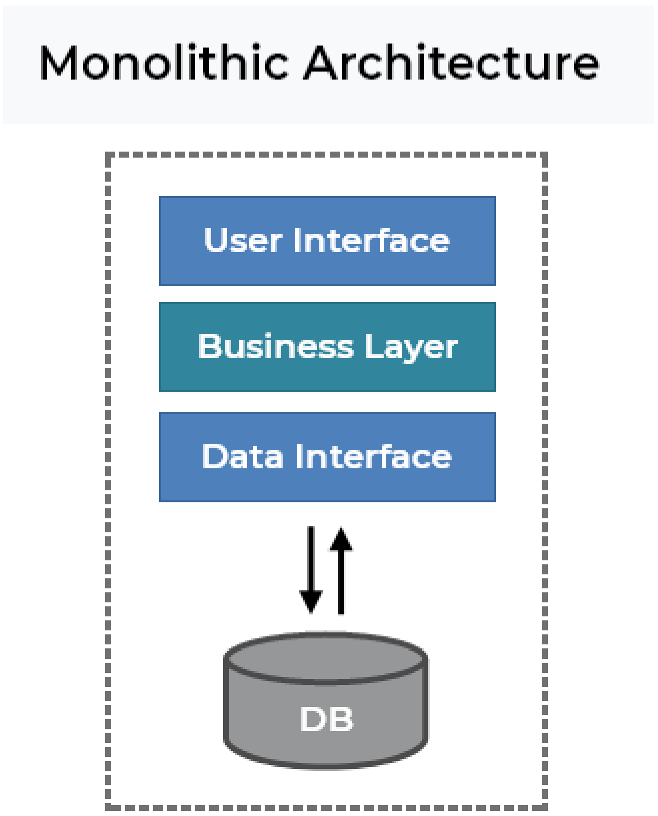
2. Context summary:

NOTE: The context and non-functional requirements is present on detail in the requirements folder.

OFS is a social application focused for Oman citizens which solves certain problems citizens are facing such as having to look up manual or local services to repair air conditions, fill up water tank and other common services . OFS platform shall facilitate the accessibility of certain local services based on user location and provides chance to promote freelancing service .

3. What is monolith architecture and why it does not fit?

Monolithic applications can surprisingly fit most of the typical application requirements and such application are very prevalent in the internet. Monolithic architecture consist of basic 3 system components as the following picture



1. User Interface – This is also known as 'front-end' and usually is one interface per application although it is possible to have multiple user interfaces which depend on software context.

2. Business Layer – This is the most important layer among them all as the entire application logic happen in this layer, this is also known as 'backend' in a monolithic software design this is **always** one entity that contains all business logic and expose

the possibility of applying business logic via protocols such as HTTP as most common example and also ws (WebSocket) and Soap(XML).

3. Data Interface – This layer is responsible for data persistence and the provision of entities and how they should look like in the database.

4. DB – This is the database of an application and this can be local or on a server and applications can switch between databases by changing configurations and usually have multiple database is recommend to be fault tolerant.

Now that we have defined what a monolith is, it is important to understand the good and bad aspects of such design:

- Good Aspects:
    1. All code in one repository, thus DevOPS are quite easy.
    2. Easy to trace errors and bugs as you have one code base.
    3. Great to start with.
    4. Such architecture is very common among applications and thus prevalent free resources to learn from that support monolithic design.
    5. Deployment is not difficult.
- Bad Aspects:
    1. Can become too big and complex after some years of maintaining and developing.
    2. As logic grows, different libraries and modules may require more RAM & Memory which increases the cost of the deployment.
    3. Stuck on one programming language, and cannot leverage booming technologies which may facilitate the development and is a better choice in general.
    4. In some cases successful Java applications can reach up to millions line of code and compiling such projects might take up to 30 minutes which reduces productivity.

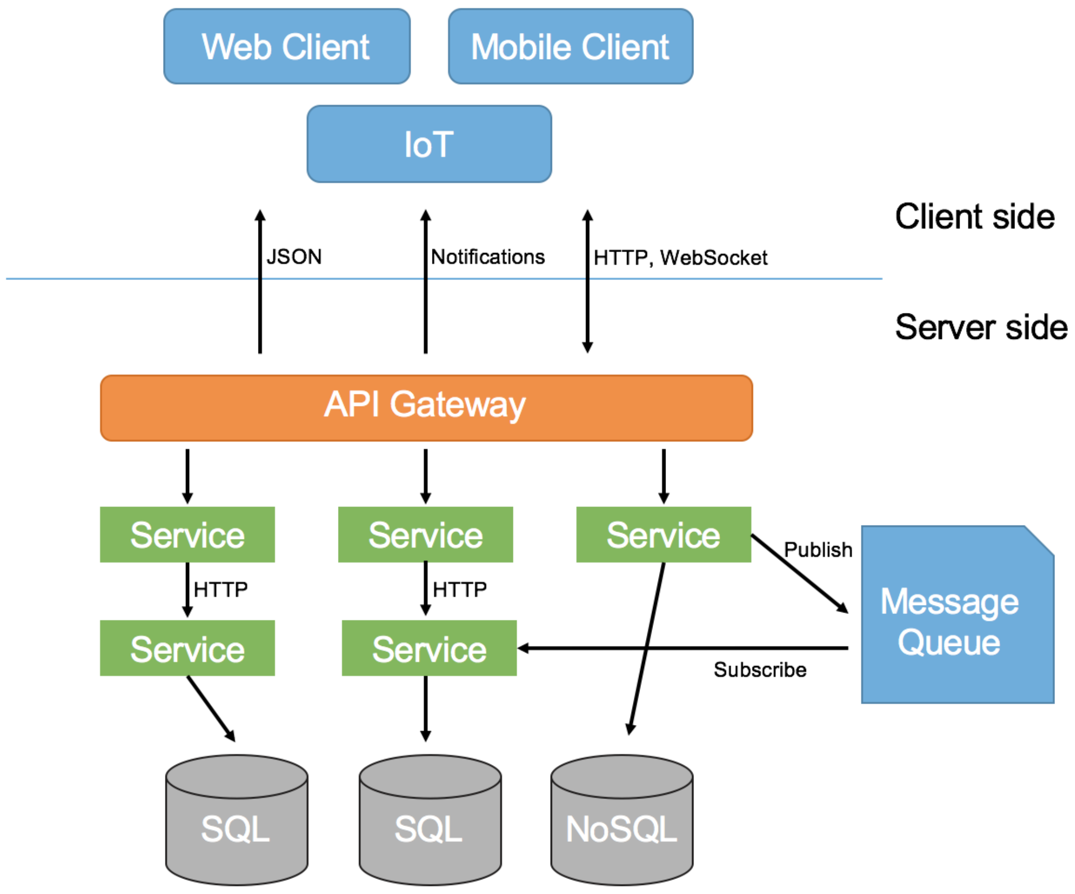
5. Hard to work in parallel with a team as you need to make sure the same code area are not being developed by multi developers simultaneously , very slow development to work solo.

6. Horizontal scaling of such architecture is very inefficient.

In the context of enterprise software scaling is the key, and the problem with scaling in monolithic applications is not cost-efficient as high loads usually connected to some features that are heavily used while other features might not and scaling monolithic software brings the entire application to load the balance instead of a company from the heavily used part of the application.

4. What is microservices and why they fit?

Microservice applications can fit enterprise context by decoupling the core business logic into small manageable services and providing the entire business functionality in one service known as API Gateway. The following is the architecture of a microservice:



1. Client – This is also known as 'front-end' and usually is one interface per application although it is possible to have multiple user interfaces which depend on software context.

2. API Gateway – This is the entry point of clients to access the entire microservices functionality. It provides routing, security and protocol transition in the application which makes it a must have in any microservices application.

3. Services – This is the decoupled business logic core which usually take one known entity business logic and takes care of all its functionality and expose them, each one of those is also known as 'microservice'.

4. Message Queue – This is a communication form that occurs asynchronously between services (service-to-service). Message queues reduce the processing work load between services by storing a service message. It is recommended to have messaging queues between services as this lowers the workload.

Now that we have defined what a microservice is and looked at its typical components, it is important to understand the good and bad aspects of such design:

- Good Aspects:

1. Easy to understand as a team of developers can handle the size of a microservice or one developer per microservice and this allow working in parallel and fast developing pace.

2. Cheaper in deployment and production as the hosted machines have to take care of one business logic entity at a time and thus need Less Memory and RAM in comparison to typical monolith deployment.

3. Opens possibilities for introducing new booming technologies as each individual microservice does not have to stick to one programming language. (For example, Python in combination with Java can be possible.)

4. Since each microservice contain less code and solely focused on one entity a microservice typically load faster than a monolith even if the number lines of code is the same and thus more productivity from the developers side.

5. Very cost-efficient when scaling horizontal as the load will be distributed among copies of the heavily used features and not the idle parts of the application.

- Bad Aspects:

1. Complex architecture design in comparison to the typical monolith architecture.

2. Data is not consistent by default, solutions to achieve data consistency in microservices vary and picking inappropriate solution we risk possibility of inconsistent data and in turn this will affect user experience badly.

3. Testing and monitoring is difficult initially, as you increase the number of microservices the possibility of having a runtime failure gets higher, each microservice has its own debugging log and solving this issue require custom configurations in a unified debugging and monitoring area.

4. Difficult development operations, as you increase the number of microservices DevOps become harder to maintain especially when it is possible to develop a microservice in whatever programming language.

In the context of enterprise software microservices architecture solves the problem of scaling and opens possibilities for distributed systems, the key in scaling in such architecture shines in the fact that when more users utilize one part of the system this architecture ensures to have instances of that highly used decoupled microservice while not waste resources by instantiating idle microservices and this is why it is a good fit.

That being said, there is nothing perfect and microservices does come with challenges which we need to find solutions to.

5. Proposed architecture design

Before going into the architecture design we need to recall the non-functional requirements as the alignment between architecture design decisions and the requirements system needs to be established.

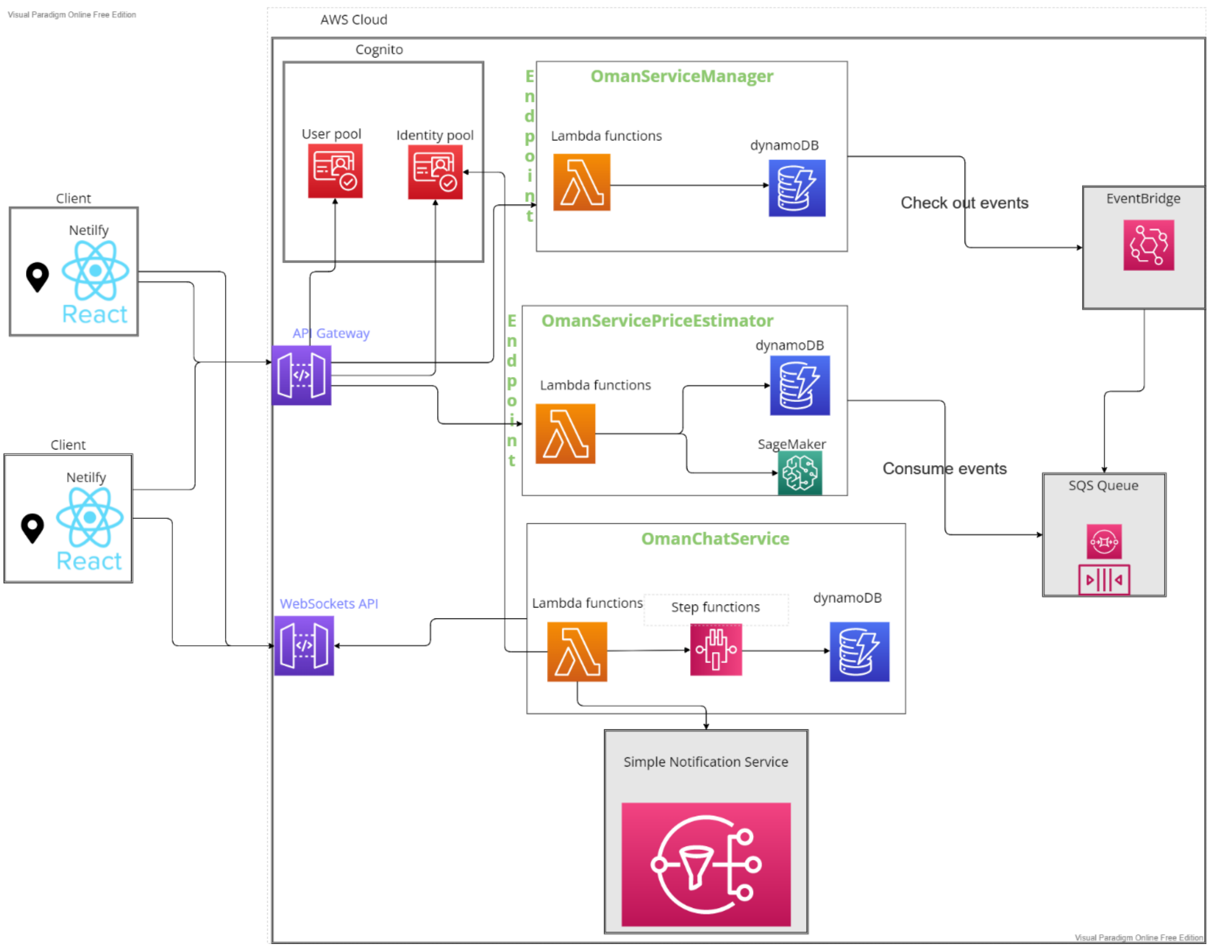
The system must be:

- **Scalable***:* *It should handle over 200,000 people simultaneously with the same latency.*
- **Available***:* *It should have a maximum downtime of 5 minutes.*
- **Cost-efficient:** *The total cost of all services must equal each service usage cost.*
- **Portable:** *It has different user interfaces that aligns with the screen size of the device.*

5.1. Design Architecture

The following is my proposal architecture design diagram for OFS Platform: -

Type of architecture: Event-driven serverless microservices architecture.



First we will define each key word in the architecture to understand what we are working with.

Event driven:

It is an architectural blueprint which advocates building systems that contain decoupled services as this design solves the problem of data consistency that typical microservices have by default.

The main key in this architecture is that decoupled services can produce an event to indicate some kind of action has occurred and it does not need to know at all about who this event will be used (or consumed) by. On the other hand, different decoupled services can see events and based on them perform actions which means they do not need to know who produced those events. This approach unlocks true potential of decoupled microservices as they can perform actions asynchronously.

One of the benefits of using Event-driven is making the system resilient and therefore combining that with scalability we can achieve a responsive system.

Serverless:

It is a model where the cloud provider completely manages servers, including traffic optimization, resource allocations and configurations. Thus, it is fast to develop with serverless and since we mentioned 'fast' latency is metric we care about in any application and serverless allocation servers near users location rather than one fixed IP address that never changes and thus result in a very responsive application.

Serverless AWS offers 'pay-for-value' model which aligns with achieve the non-functional requirement **cost-efficient** and applications built on using serverless on AWS are **highly scalable** and **available** by nature.

Microservices architecture:

NOTE: We have defined microservices on the document, thus this will be a short definition.

Microservices architecture is an architecture that decouples the entire business logic into business entities each is managed by a small, maintainable and deployable service each has its own database, then all represented by an API Gateway which connects to all those small services.

Secondly we will go over the architectural design choices

Client:

The client-side is the user interface and here is where the non-functional requirement **portability** comes in play. I decided to choose React/React Native as a JavaScript framework since it allows me create web applications, Android and IOS applications. Building the interface using JavaScript also allow me to shift between frameworks if need but with the downside of refactor JavaScript framework related code.

API-Gateway:

Utilizing AWS Services when we build serverless applications makes since and this is no exception for API Gateway. It is definitely a must have in any microservice architecture to expose the entire functionality of decoupled microservices to clients.

I believe that I need to ensure that the API gateway is available all the time which is another topic I need to research on and have not yet figured out what could ensure this.

Cognito:

It is authentication/authorization service which completely manages user data. One of the core system components of OFS Platform is the login components where we have different roles mostly importantly the service freelancing provider and the service seeker, instead of implementing this part ourselves utilizing what the cloud offers to perform the same operation can be save time and effort which we can put in the other system components.

Lambda:

It is a highly available service and falls under the category *FAAS* (Function-as-a-service), which allows developers to reliably run their run while the cloud provide completely manages the server infrastructure. Lambda is a core service which can align with about 200 AWS services, and we will develop the main microservices logic with in serverless framework.

EventBridge:

It is an AWS service which acts as an event bus. Microservices such as OmanServiceManager can perform CRUD actions such as creating a freelancing service and EventBridge will keep that event (action of creating) when other microservices can consume it and perform certain action about it. This allows asynchronous communication among microservices, for example between OmanServiceManager and OmanServicePriceEstimator where we utilize AI and data produce by events to estimate the price of a service based on freelancers data and utilizing this AWS Service allows for Event driven architecture.

SQS Queue:

It is an AWS service which acts as a messaging queue, that is connected to the EventBridge. The reason why we want to connect the SQS Queue with the Eventbridge and then the microservices with SQS to able first in first out behaviour. In addition to be event driven, this can be beneficial as the service of freelancers prices can differ from time to time. Thus, can interesting insights in SageMaker AI FreelancingServicePriceEstimator such understanding one service category average price changes over time.

WebSockets API:

It is in fact an API Gate but behaves in a WebSocket manner, looking the API Gateway AWS service official it is not possible to have a gateway that is both REST and Websockets which results in me adding another gateway specifically for Websockets functionality to work in the system. We want to able chatting functionality and rest communication type does not fit this role since RESTful communication is request-response model whereas websockets is a bidirectional communication that allows for sending requests and receiving requests between client and server quickly. (We don’t want to refresh page to see a new sent message, nor refresh interval.)

SNS Simple Notification Service:

It is notification system that is built-in AWS manages and provides which is one of the core functionalities we want to implement in OFS Platform and thus it makes since to utilize that service.