**TABLE OF CONTENTS:**

1. [Microservices](#Introduction)

2. [History of Microservices](#HistoryOfMicroservices)

3. [Problems with Monolith & SOA[Service Oriented Architecture]](#ProblemWithMonolithAndSOA)

4. [Microservices Architecture](#MicroservicesArchitecture)

5. [Problems solved by Microservices](#ProblemsSolvedByMicroservices)

6. [Designing Microservices Architecture](#DesignMicroServices)

7. [Deploying Microservices [deep dive in CONTAINERS & Kubernetes concepts]](#DeployMicroservices)

8. [Testing Microservices](#TestingMicroservices)

9. [Service Mesh [Make Communication between services]](#ServiceMesh)

10. [Logging & Monitoring](#LoggingAndMonitoring)

11. [When we should not use Microservices](#WhenWeShouldnotUseMicroServices)

12. [Microservices & the organization](#MicroservicesAndOrganisation)

13. [Anti-Patterns & Common Mistakes](#AntiPatternCommonMistakes)

14. [Breaking Monolith to Microservices](#BreakingMonolithToMicroservices)

15. [Case Study](#CaseStudy)

16. [Conclusions](#Conclusions)

**MICROSERVICES:**

One of the popular architecture in Software World.

**Agenda:**

1. History of Microservices

2. Problems with Monolith & SOA[Service Oriented Architecture]

3. Microservices Architecture

4. Problems solved by Microservices

5. Designing Microservices Architecture

6. Deploying Microservices [deep dive in CONTAINERS & Kubernetes concepts]

7. Testing Microservices

8. Service Mesh [Make Communication between services]

9. Logging & Monitoring

10. When we should not use Microservices

11. Microservices & the organization

12. Anti-Patterns & Common Mistakes

13. Breaking Monolith to Microservices

14. Case Study

15. Conclusions

**History:**

Microservices are introduced to overcome the problems faced in 2 different architectures...

i. Monolith

ii. SOA (Service Oriented Architecture)

*MONOLITH Architecture:*

Original Architecture

All the software components are executes as Single Process.

-> Thread

-> Memory Resource

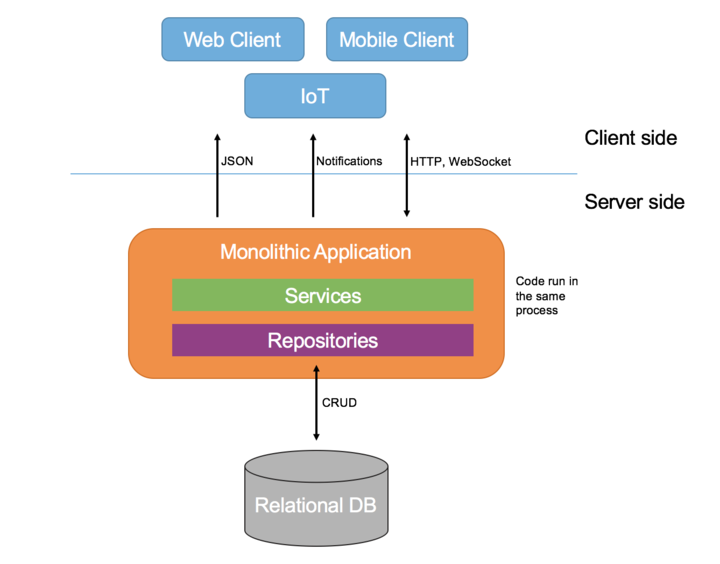
-> Compute Power

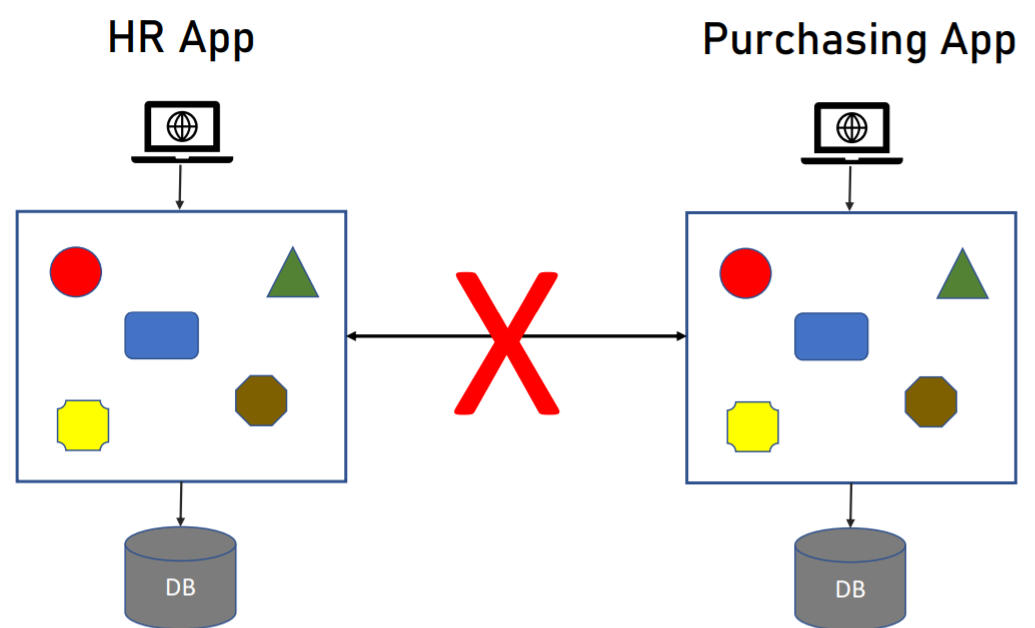
Strong Coupling

Implemented as SILO (Standalone Apps). It will not export anything. Nothing can Go Out.

**Pros:**

Easier to design [Because, there is no *Networking, Message transfer, Cross Debugging*]





*SOA: [Service Oriented Architecture]*

Coined in 1998

Apps are exposing the functionality to outside world

Usually implemented using **SOAP, WSDL**. SOAP & WSDL are in XML formats that contains all the things that are exposed by the Apps.

**ESB (Enterprise Service Bus)** is used to make communication between 2 services or else between 1 service & 1 client.

-> It is the very important component in SOA.

-> It takes of routing, aggregation, validation, authentication etc.,

-> ESB engines are basically developed by ORACLE or IBM.

Things which was overcome from Monolith by SOA:

We allowing the set of information of the application into outside world. If I want to access the azure application by referring & implementing its WSDL we can access it.

**PROBLEMS THAT FOUND IN *MONOLITH* AND *SOA:***

**Problem in MONOLITH:**

*1.Single Technology*

Finally, *Monolith* is going to execute as a SINGLE Process. So, all the components are needs to be developed using the same development platform.

Upgrading is also a problem. We can’t able to upgrade single component.

*2.Inflexible Deployment*

We should deploy all the components every time. If we have single line change we should deploy all the components.

*3.Inefficient Compute Resources: [CPU & RAM]*

If one specific component needs more *CPU or RAM* resources. We can’t able to allocate it. Because, whole *CPU or RAM* was allocated to entire Monolith Process.

*4.Large & Complex:*

Codebase is LARGE and Complex. Which contains lot of dependencies. So, small changes will affect the other components.

Testing the application also difficult.

**Problems in SOA [Service Oriented Architecture]:**

*1.Complicated & Expensive ESB (Enterprise Service BUS):*

ESB engines are basically provided by companies like ORACLE or IBM. So, it also seems to be expensive.

Because of all the important tasks are handled by ESBs most of the companies which are using this ESBs are focusing more on ESBs rather than the actual services in the SOA.

*2.Lack of Tooling:*

SOA should achieve the Short Development Cycle. When achieving this *SHORT DEVELOPMENT CYCLE,* **testing and deployment** takes place manually by the developers there is *no tool* to provide the testing and deployment functionality.

**Characteristics of Microservices: [1,2,5,6,7 are the Most Important Attributes]**

*1.Componentization via Services*

Creating a modular Software is the Best Practice.

-> Using Libraries

-> Using Services (Web APIs, RPC)

In Microservices we are using the Services

*2.Organized Around Business Capabilities*

UI 🡪 API 🡪 Logic 🡪 DB (All these things are handled by a SINGLE Team)

Motivation:

-> Quicker Development

-> Well defined Boundaries

*3.Products not projects*

Traditionally, developers don’t have a interaction with customers when developing the PROJECTS. But, Customer interaction will present in developing the PRODUCTS.

Motivation:

-> Increase the customer satisfaction

-> Changes he developer’s mindset.

*4.Smart End-Points and Dump pipes*

In SOA, communication between 2 services are handled using ESBs(Enterprise Service Bus). This ESBs are more complex & expensive.

Microservices systems use ‘dump pipes’ – simple protocols[HTTP]

Services can communicate each other by using simple HTTP Request.

Motivation:

->Accelerate Development

->Make the app easier to maintain

*5.Decentralised Governance*

We can select any type of DBs and any type of languages based on the technology requirements. We no need to follow any centralised approach.

Motivation:

Enables making the optimal decisions for the specific service.

*6.Decentralised Data Management*

Each services no need to use the Single DB. Each services can have its own. But, it is not for all the cases it may change based on the requirements.

Motivation:

-> Having the right DB is important.

->Encourages Isolation.

*7.Infrastructure Automation*

To add the AUTOMATION in TESTING and DEPLOYMENT. Short Deployment Cycles is must in Microservices. Because, it is essential for the Microservices.

Most used Automation Tools:

1.Azure Devops

2.GitLab

3.Jenkins

Motivation:

->Short Deployment Cycle

*8.Design for failure*

In Microservices, there are *lot of processes* and *Network Traffic* will present.

We should handle all those things by implementing proper *Logging & Monitoring* Mechanisms.

Monitoring Tools:

1.Azure Monitor

2.Application Insights

3.Kubernetes

Motivation:

->Increase the System’s Reliability

*9.Evolutionary Design*

Moving from Monolith to Microservices should be GRADUAL. No need the break the whole part.

**Problems solved in Microservices:**

1.Single Technology Platform (using *Decentralized Governance*)

2.Inflexible Deployment (using *Componentization using Services, Decentralized Data Management*)

3.Inefficient compute resources (CPU & RAM) (using *Componentization using Services*)

4.Large and complex code base (using *Componentization using Services*)

5.Complicated and Expensive ESBs in SOA (using *Smart End Points & Dump Pipes*)

6.Lack of Tooling (using *Infrastructure Automation*)

**Designing the Microservices Architecture:**

“ Plan More, Code Less ”

Architecture Process Flow: (3,4,5 are the most important things)

1. Understand the System Requirements.
2. Understand the Non-Functional Requirements.
3. Map the Components.
4. Select the Technology Stack.
5. Design the Architecture.
6. Write Architecture Document. (like a *help document*)
7. Support the team.

**MAP THE COMPONENTS:**

**Mapping the Components** (= Services):

**VERY IMPORTANT STEP**

Should define the various components based on the below factors:

* Business Requirements

Collection of requirements around specific business capability.

Eg: *In ORDER MANAGEMENT,*

Add, remove, update, calculate amount

* Functional Autonomy (Keep the suitable functions only)

For *ORDER MANAGEMENT,*

Getting all the ORDER Details [**ACCEPTED**]

Get all the users whose age is between 35 and 40[**DECLINED**]

* Data Entities

Services are build around the Entities.

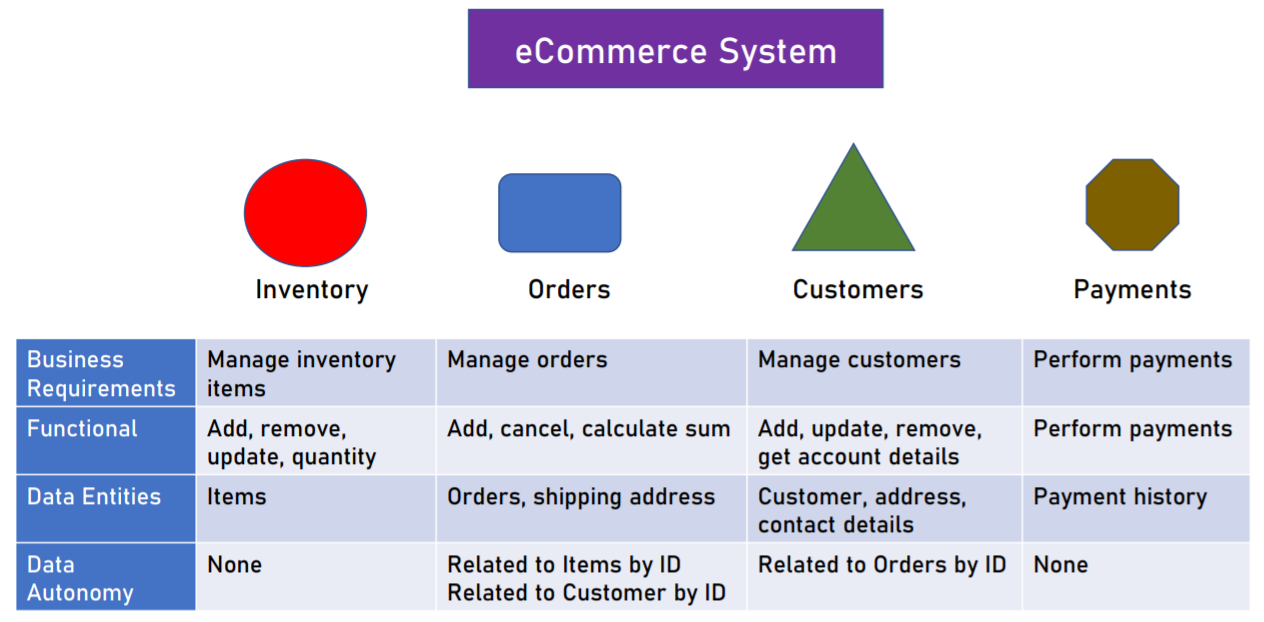
Eg: Order, Items

* Data Autonomy

Should define all the needed fields in proper DBs.

Should maintain proper relationships between different entities.

Eg: Both Customer Name and Address should be in the same table



**Edge Cases of eCommerce System:**

#1 Retrieve customers names with their orders count.

#2 Getting all the data. (If the amount of data is too large. We should maintain the Report Engine)

**Defining Communication Patterns:**

Efficient communication between the services is crucial.

Main Patterns:

1. 1 to1 Sync
2. 1 to 1 Async
3. Pub-Sub / Event Driven

* **1 to 1 Sync**

One service waits for another service to respond.

It can be implemented in following ways:

Direct Connection

**SERVICE 1 🡪 SERVICE 2**

*Cons:*

Single Changes in a any one of the service will impact more.

Service Discovery

Yellow Page Service [Service which contains the URLs for all the services. It will provide the appropriate URL for the request.]

Service 1 needs to know the *Yellow Page Service* URL alone.

**SERVICE 1 🡪 YELLOW PAGE SERVICE 🡪 SERVICE 2**

*Tool:* **Consul**

Gateway Service

More over similar to Service Discovery. In addition to that, we can provide some authentication service here.

Current Industry trend.

* **1 to 1 Async**

One service cannot wait for the response from other service.

**Service 1** just send the message to the **Service 2.** Service 2 will takes care of the remaining process including error handling process also.

*Cons:*

Sometimes we need *3 party tools.*

Error Tracking is difficult.

We can implement it by following way:

**SERVICE 1 🡪 *Queue* 🡪 SERVICE 2**

Tool (Queue): **RabbitMQ**

* **Publisher-Subscriber / Event Driven**

**SERVICE 1** doesn’t wait for the response from other one.

Service wants to send messages to more than one services (subscribed services). **Service 1** doesn’t know about how many services going to receive the message.

**SERVICE 1 🡪 Publisher-Subscriber Engine 🡪 SERVICE 2**

**🡪 SERVICE 3**

**🡪 SERVICE 4**

Note: Once the communication patterns are implemented. It cannot be modifiable.

**SELECTING THE TECHNOLOGY STACK:**

In *MONOLITH,* this process is one which has high weightage. Because, whole application is going to developed using the SINGLE PLATFORM/ LANGUAGE.

But, in *MICROSERVICES* we can choose the technology stack based on the requirements.

We should consider the below things to select the technology stack:

1. Development Platform
2. Data Store

*Development Platform:*

*Data Store:*

Relational DB:

For Structural data storage & for complex querying.

Popular DBs: **SQL Server, MySQL, PostgreSQL**

No SQL DB:

For large amount of data & without having any complex querying process.

Popular DBs: **MongoDB, Amazon Dynamo DB, Couchbase, Azure Cosmos DB**

Cache:

For frequently accessing data & the data will not change frequently like country names.

Most Popular: **Redis**

Object Store:

To store unstructured large data Documents, Images, Files etc.,

Why not to use the original File System?

Deleting & Modifying the data can be easily done.

Popular Tool: **Microsoft Azure Blob Storage, Amazon s3**

**DESIGN THE ARCHITECTURE:**

Layer based architecture is needed to achieve the MODULAR application:

UI (User Interface) / SI (Service Interface) Layers.

Business Logic Layer

DAL (Data Access Layer)

|  |
| --- |
| UI / SI Layer |
| Business Logic Layer |
| Data Access Layer |

We should not skip the Business logic

We should not reverse the process. i.e. Data Access Layer must not the Business Logic Layer

**DEPLOYMENT OF MICROSERVICES:**

Deployment of an Microservices should be fast and effective.

In case of *Monolith,* we are going to deploy the single application. But, in case of *Microservices* we need to deploy multiple applications building, testing, deploying all the application by manually is very difficult. So, **Infrastructure Automation(One of the Property of Microservices)** places important role here.

**CI/ CD:[Continuous Integration – Continuous Delivery/ Deployment]**

To do CI & CD process we need a **CI/ CD Engine.** This Engines can be provided by *Azure Devops* or else we can use *JENKINS* in place of it.

**Container:**

Deploying the application using a Containers is very efficient deployment when compare with deploy using the VMs.

**Docker: (developed in 2013)**

Most popular Container based platform to deploy the applications in efficient manner.

Problems arise when using the DOCKER:

Deployment

Scalability

Monitoring

Routing

High-Availability

**Kubernetes: (developed by GOOGLE in 2014)**

To solve all the above listed problems in DOCKER we are using Kubernetes.

For Container Management.

**TESTING THE MICROSERVICES:**

Following 3 types of testing are the most common test which was happening in every software test cycle:

Unit Testing.

Integration Testing.

End To End Testing.

In case of Microservices, we should follow all the testing proper way. Because, one small change in a single application reflects in multiple dependents applications (Services).

*Unit Testing:*

Testing the newly implemented function in a specific application using any automation frameworks **(Java – Junit).**

*Integration Testing:*

Testing the applications integrated to the services. For eg: validate the DB is integrated properly or not.

Unit Testing framework also supports *integration testing.*

3 ways of integration testing:

Fake:

Testing the DB integration by using the FAKE DB inside the service itself.

Stub:

Validated using the Hard coded data.

Mock:

Here, DB doesn’t holds any data. We just check the access is made or not.

*End to End Testing:*

To validate entire flow of the services. From the SERVICE – START to Last SERVICE – END.

**SERVICE MESH:**

**Software Application** that manages the *Service-Service* Communication.

Communication between services can cause lot of problems.

Service Mesh deals with below problems in Microservices:

Timeouts

Security

Retries

Monitoring

Services provided by ***Service Mesh;***

Protocol conversion

Communication Security.

Routing.

Circuit Breakers (to avoid the Cascade Failure of Services)

Load Balancing etc.,

Types of Service Mesh:

In-Process.

Pros:

* Performance is better. (Since, we didn’t need any network calls to communicate with **Service Mesh**)

Side Car (Most Popular). [Recommended]

Pros:

* Platform independent (we can develop it using any language. We no need follow the service’s language.)
* No code changes needed in Service side.

**No need to waste our time to handle the** [**above listed Problems.**](#ProblemsInMicroservicesAddrByServiceMesh) **We can use the Service Mesh products instead.**

**Sidecar Products: [Istio is most preferred. If not works move to next ones.]**

****

**In-Process Product:**

****

**Should You Use Service Mesh?**

Only when we have more number of services and we want to communicate each other much more.

Or else complex communication requirement with various protocols.

**LOGGING AND MONITORING:**

It is the extremely important process in Microservices.

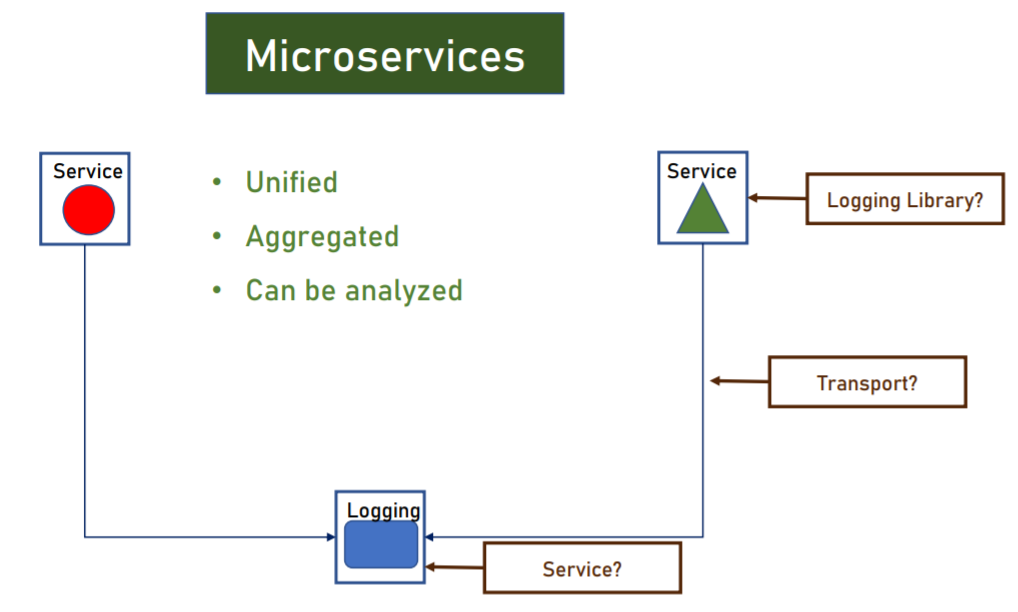
Flow of the Microservices was very big. So, identifying the errors is difficult

***Logging vs Monitoring***

To get the Systems Activity. Systems Metrics.

Documenting Errors. Generate Alert whenever the system CPU usage > 90%, Response Time > 5secs

**Implementing Logging:**

****

**Important things** we can log: Timestamp, User, Severity etc.,

**Logging Services:**

****

**Implementing Monitoring:**

**🡪** Infrastructure Monitoring

🡪 Application Monitoring (Source: LOGS)

Most of the monitoring products includes both of them. It is capable to capture the underlying servers performance and also can able to get the details from the Logged details of the applications.

There are huge number of monitoring products are available. So, we need to waste our time on it.

**Monitoring Products:**

****

**WHEN WE SHOULDN’T USE MICROSERVICES:**

When we are having the below characteristics we should avoid using the Microservices:

* Small Systems
* Intermingled Functionality or data.
* Performance Sensitive. (Like Military application may need more performance.)
* Quick & Dirty Systems. (Whereas, Microservices needs more time plan and design. Without proper planning and design we are meshed with more problems.)
* No planned updates (Microservice’s is most important functionality **Short Update Cycle.** If our application doesn’t have anymore updates no need to go for microservices).

**MICROSERVICES & THE ORGANIZATION:**

Moving to Microservices make more changes in an organization. Without adapting to the changes it is difficult to move with Microservices.

*Conway’s Law:*

*“Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization’s communication pattern”*

*Problem in Traditional System:* Each and every tech team do their own work. Team work misses there. So, we end up with a Project not a Product.

*Ideal Team* should interact more to develop a good product.

*When you are in Organization that is not using the Microservices how do you change the mindset of the organization?*

Give sessions on success stories of Microservices and Show them as POC (Proof Of Concept) to change the Mindset of the organization’s approach.

**ANTI-PATTERNS & COMMON MISTAKES:**

Some of the anti-patterns we shouldn’t follow and some common mistakes we should avoid:

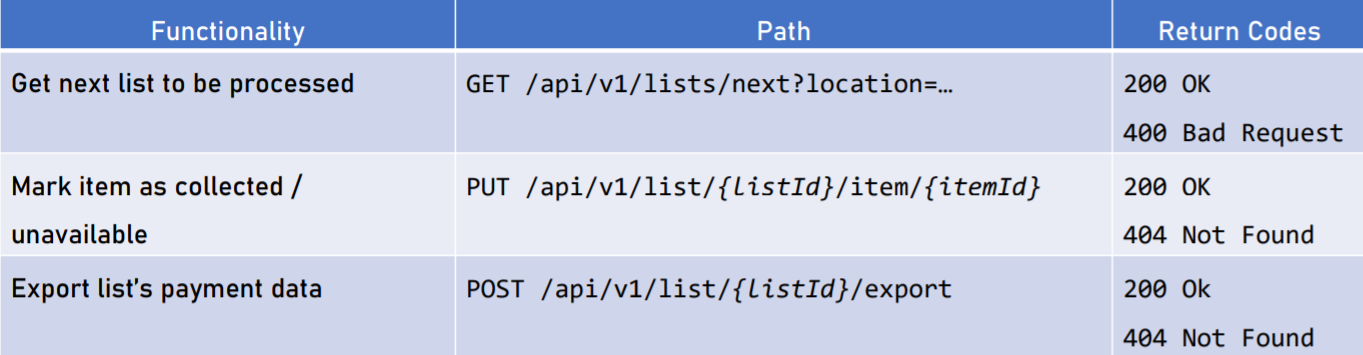
*No Well-Defined Services:*

We should do a proper component mapping. We should not tell our teammates we are going to develop a Microservices. Should start with proper design.

*No Well-Defined API:*

API plays vital role in Microservices. Without APIs services will not communicate effectively.

During, the API developed following the below table is good practice.



*Implementing Cross-Cutting Last:*

*Expanding Service Boundaries:*

**BREAKING MONOLITH TO MICROSERVICES:**

What is the Need?

Small Update Cycle.

Modular approach.

Platform Independent.

3 different strategies:

1. New modules as Services
2. Split the existing modules to Services
3. Developing the entire application from a Scratch. [Complete Rewrite]

**CASE STUDY: (To compare all the theoretical concepts with Real World Scenario.)**

Refer PPT downloaded in **Resources.**

Mapping the Components:

Usage of the Service.

User who is going to use the service.

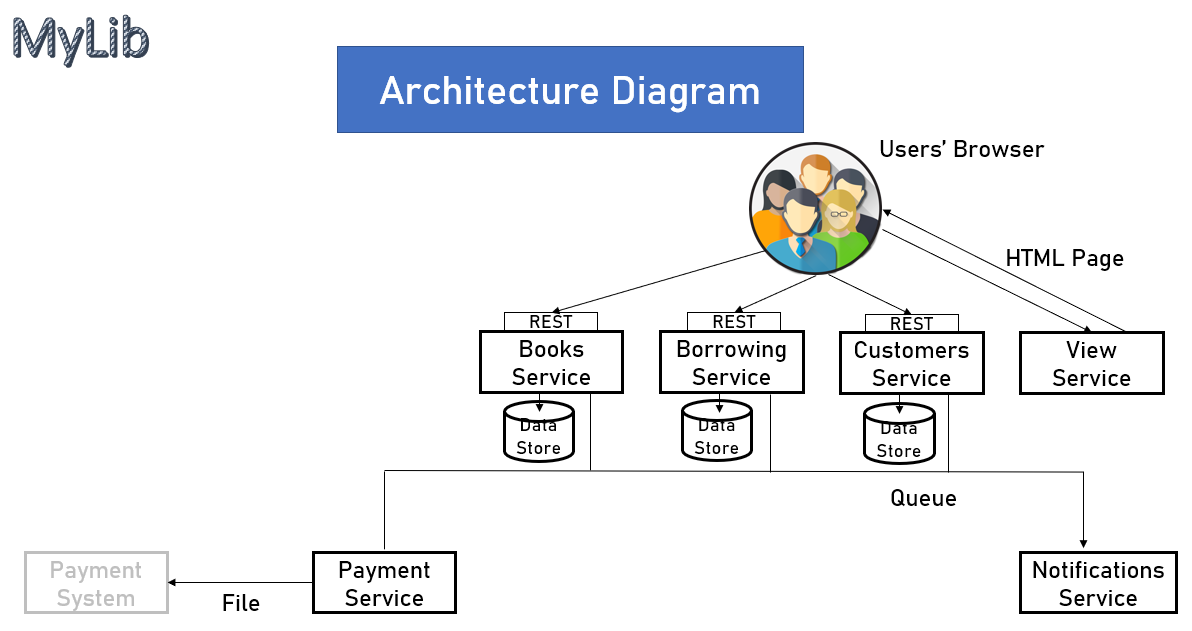
Service needs storage or not. (Because, some services depend upon other services. So, that kind of services will not need storage space anymore.)

List out all the References.

Select an API Type. [REST API is used for the immediate responses.]

And finally, follow the API template. [Functionality, Path, Return Code]

**Sample Architecture Diagram for Library Management Microservices: [it will help the customer to know about what exactly going there.]**

****

**CONCLUSIONS:**

Summary.