Architecting Microservices for Cloud

A handbook of best practices, design considerations and practical trade-offs

Indrajit Nadgir

TABLE OF CONTENTS

TO AMBIKA

She is the inspiration behind this book.

Author information

Indrajit Nadgir
email: indrajitnadgir7@gmail.com

Indrajit is a software product architect with decades of expertise in designing and developing enterprise software applications. He has led complex cloud modernisation and digital transformation initiatives for products with million(s) $ size, that impact thousands of customers and millions of end users.

INTRODUCTION

I have been a software product architect for over a decade now. The technological evolution has been phenomenal over the past few years. When internet came around in 1990s it marked the beginning of web applications with static web pages. Then came the client-server model with dynamic web applications. Those were largely developed using what is now called monolithic architecture. Over the last couple of decades the technology ecosystem has evolved to create Simple Object Access Protocol (SOAP), Service Oriented Architecture (SoA), Representation State Transfer (REST) APIs, and then eventually microservices on cloud. Of course microservices have been around for a very long time and so has cloud. But the significance they have gained in the last decade is just mind blowing.

If you are a product architect who wants to understand all the moving parts of a microservices application or know the best practices while designing your microservices this book is for you.

In this book we start with what cloud strategy suits your needs and then evaluate best practices, trade-offs & important considerations in microservices API design, transaction management, data distribution strategies and security.

This is not an introductory level book. For example, this book does not explain what a REST API is or what a transaction is. It is assumed you already know all that stuff. If you don't may be a refresher on basic concepts like REST APIs, transactions, data bases and security is a good idea before you start this book.

CLOUD STRATEGY FOR YOUR APPLICATIONS

Cloud and microservices seem to be the buzz words everywhere in the world of application development. Microservices and cloud have revolutionalised the way enterprises approach their digital transformation strategy. There is a paradigm shift in the technology ecosystem with cloud that brings so many benefits with it like high scalability, high availability, fault tolerance and better time to market. But the most important benefit the cloud offers is that it changes CAPEX to OPEX. It is one of the most important advantage that cloud offers and it is a driving force for many CTOs to embrace cloud.

So how does cloud convert CAPEX to OPEX?

When you run your applications on-premises, you first invest in setting up a data center, servers, network and buy software licenses. Then you hire the administrators that will manage your data center. All of this is upfront cost and what counts as Capital Expenditure (CAPEX).

For example – Let us say you have an e-commerce business application and you set up a data center with all the hardware and software leased with a certain load in mind. But if your business is far more successful and far more quickly that you imagined , you need to provision more infrastructure. But it could be seasonal or a spike which does not sustain. After a point you have all this additional infrastructure which you do not need anymore. You see the problem? You are to predict the demand in advance and invest before you see any revenue from that demand. The chances that you predicted wrong are far higher that you think.

However, when you move to cloud all this cost is reduced. Cloud providers will provide you the required infrastructure on the go. If you need more you can provision more. This converts the cost model to Operational Expenditure (OPEX). You end up paying for what you use. Thus it gives you more control over the costs and brings down the overall cost – Total Cost of Ownership (TCO).

Cloud takes a pay-as-you-go approach. So as your demand increases you can provision more (scale) and if the demand reduces you can reduce your provisions accordingly.

Now that cloud offers all these advantages, should we move all our applications to cloud?

Let us consider the following criteria for a moment:

- You have a small business application created using a fixed technology stack which is unlikely to change.
- The demand for your business/ application is stable and predictable. It does not increase/ decrease drastically or suddenly.
- The teams creating the application are fixed in size and unlikely to grow exponentially.

- It is okay to have a long time to market for your application and frequent releases are not required.
- The applications are not mission critical and some downtime can be accommodated.
- The network traffic is stable and predictable
- You already have a data centre set up with required infrastructure to run your application.

If you see that you tick most of the boxes above then you must consider using or continuing to use a monolithic architecture application. There is nothing wrong with a monolith as long as it is a deliberate architectural choice.

So when do you need your applications to be the cloud?

It is essential to take a pragmatic approach when choosing between architectures. The purpose of the architectural choices is to enable a solution to a business problem considering all requirements like time to market, cost, availability, performance, fault tolerance, expected growth or expected increase in load, etc.

The problem is – when you have a hammer everything looks like a nail. And microservices architecture is no different. We have to be careful because microservices architecture will not necessarily solve the problem and some times it will create many.

So when do you choose to develop your application using microservices?

1. Your business is scaling really fast. This is typically observed with fast growth start ups where there is a sudden explosion of user base and the application must keep up with the sudden increase in the traffic. Microservices architecture is built to do that. Applications built using microservices deployed on cloud infrastructure can be easily orchestrated to scale on demand.
2. Scaling: You have a business where the number of users can suddenly increase at a given time of the day or in a given month of the year. For example: you have a tear end clearance sale on your e-commerce website every year in say December, right before Christmas. You expect that the traffic will increate 4-5 folds during that time. You want your application to be able to handle such load automatically.
3. Time to market: When your application is a monolith, you build the entire application even when you make a small change to the application. It takes time to build and especially to test. Because now you have to run the whole battery of tests to deploy on production even though your change is small. Develop -> Staging -> Production testing.

If it is a product that must be released to an end consumer via a delivery portal, then there is the whole process to get the product to GA (General Availability).

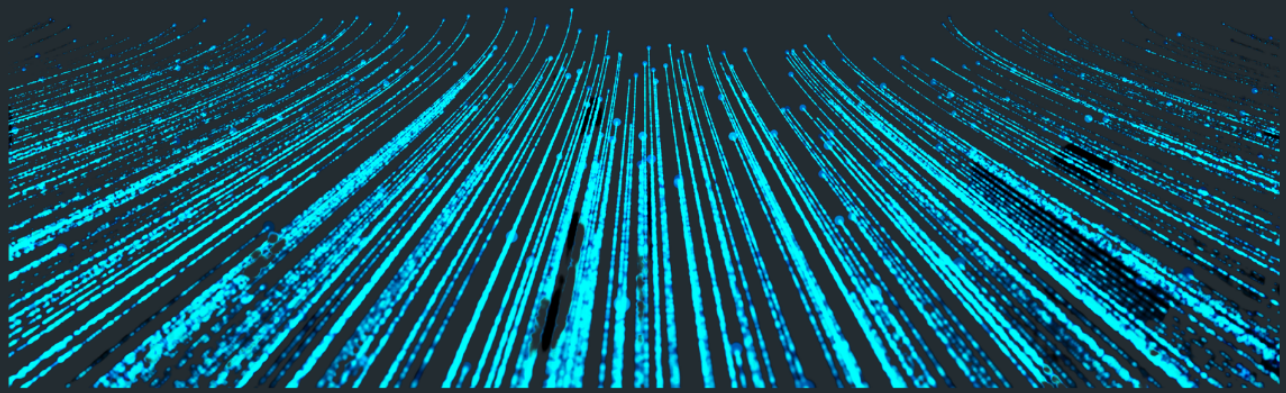
But with microservices you can build, test and deploy/ deliver each microservice independently. Microservices are designed to be light weight and thus integrate very well with Continuous Integration and Continuous Delivery (CI/CD) platforms. So,the time to market reduces considerably. This is especially an advantage for productswhich require to make rapid changes to their design during early stages.

4. Fault tolerance: With a monolith a failure in one part of the application brings down the whole application. But it is unacceptable when you have a mission critical application where you need fault tolerance to be built in to the application. Microservices are by design fault tolerant since they are independent of each other.
5. Zero downtime (High Availability): Another requirement for a mission critical application is to be highly available. A near zero downtime is expected not only during the runtime of the application but also during upgrades and patches. Microservices architecture can ensure high runtime availability. They can be delivered easily with advanced devops like using canary releases to ensure smooth upgrades.
6. Different parts of the application use different languages or different databases.
7. Yours is a very successful product which has evolved beyond recognition. The teams have become large and unmanageable. The source code keeps bloating up and maintaining it is no longer possible in the traditional way. You need to move to microservices architecture immediately. Microservices architecture offers to modularise the application into small self contained units which can be led by independent teams.Each microservice is designed to be developed, tested and deployed or delivered independently so individual team can choose to do what is best for them thus allowing them to scale.

If your application needs any of the above criteria satisfied then consider microservices architecture. In this book we will discuss many challenges that will arise with microservices architecture and how we can solve them. What the different trade-offs, considerations and best practices are.

As we begin designing a microservices application for cloud, one of the most important area to start with is the API design. Microservices, most of the times, are made of APIs. That is how they expose their functionality to outside world. In the next section we explore some of the best practices to design our APIs.

API DESIGN FOR YOUR CLOUD APPLICATIONS



A good API design is a prerequisite to a good microservices application. A well thought out API strategy must be put in place before starting any development work so that the resulting APIs are easy to use and consume, well documented, secure and performant. REST APIs are the most common web interfaces that are available today which can be consumed by various clients like a mobile application or from a web browser. The beauty of REST APIs is the ease with which they can be developed and consumed. It is very easy to develop a REST API but difficult to do it right. Below are a few best practices that will help you to design your APIs so that they can be robust.

Use JSON in HTTP requests and responses

JavaScript Object Notation (JSON) has gained immense popularity over the years because of its simplicity and readability. REST APIs can consume and respond in both XML and JSON formats, but using XML needs a lot of parsing and processing on the client side before consuming it and it introduces considerable overhead. Use of JSON makes it straight forward to consume data especially since there are in-built JavaScript libraries to parse JSON objects to text and vice-versa on both client and server side.

Nouns in the url

Use nouns instead of verbs to name the endpoint URLs. HTTP requests already have verbs in them like GET, POST, PUT and DELETE. Do not add those actions again in the url like

```
/v1/@JaneDoe/get_articles


/v1/@JaneDoe/create_article
```

This will make the urls verbose and can be quite confusing as the APIs get more complex.

Instead use nouns to indicate the resources on which the action is intended and let the standard HTTP verbs indicate the action.

For example:

```
GET /v1/@JaneDoe/articles (to get all articles)
```

```
GET /v1/@JaneDoe/articles/:article_id (to read an article with an id)

POST /v1/@JaneDoe/articles (to create an article)

PUT /v1/@JaneDoe/articles/:article_id (to update an article with an id)

DELETE /v1/@JaneDoe/articles/:article_id (to delete an article with an id)
```

Hypermedia As The Engine Of Application State (HATEOAS)

HATEOAS is a REST architectural constraint and a design best practice. With HATEOAS we can send the available actions and resource links embedded in the HTTP response dynamically. The client sending the request does not need to hard code the URLs and thus it can help prevent broken URLs as the endpoints evolve over time. The clients can navigate and change the application state based on the hypermedia links in the responses thus driving the workflow forward.

The server can decide what further resources should be made available to the user based on the request (and other parameters like authorisation, business logic, etc) and add appropriate links.

The client can build menus, buttons, or navigation dynamically based on the response from the server.

For example consider a GET request to fetch the list of articles by a given author.

```
GET /v1/@JaneDoe/articles?page=3 HTTP/1.1
```

The server can just return the paginated list as requested. But then the client does not know if it can go to the next page or what other actions it can perform from this point onwards after displaying the requested page. And the client has to build the URLs on its own to navigate and perform those actions. The client gets involved in driving the application state and any API changes on the server-side must result in changes on the client-side. It leads to broken links and functionality more often that expected.

HATEOAS can help handle this problem with embedded links which tell the client what actions it can perform next and how.

For example a HATEOAS response to the GET request above would look like:

```
HTTP/1.1 200 OK
Content-Type: application/hal+json
{
        “meta”:
                {
                        “page”: 1
                        “total_pages”: 5
                }
        “author_name”: Jane Doe,
        “email”: jane_doe@myamazingwebsite.com,
        “data”: [
                {
                        “article_id”: 1,
                        “article_title”: Example Title 1,
"links": {
"self": "/v1/@Jane_Doe/articles/1"
}
                },
                {
                        “article_id”: 2,
                        “article_title”: Example Title 2,
"links": {
"self": "/v1/@Jane_Doe/articles/2"
}
                }
        ],
"links": {
"self": "/v1/@Jane_Doe/articles?page=3",
"first": "/v1/@Jane_Doe/articles?page=1",
"prev": "/v1/@Jane_Doe/articles?page=2",
"next": "/v1/@Jane_Doe/articles?page=4",
"last": "/v1/@Jane_Doe/articles?page=5"
}
}
```

The client knows from the response that it can navigate to the next page, previous page, first page and the last page — all dynamically linked to the navigation buttons in the UI. Further, the users can also click on the individual articles to display the contents of the article.

Although HATEOAS relieves the client from driving the application state, it does not mean the client can be oblivious to the application state. The client still needs to know what different actions are possible from a given state. But HATEOAS helps the client to determine what among those actions are valid and how to handle them — dynamically.

However, on the downside it can make responses quite verbose and many developers find it impractical to handle workflows entirely using HATEOAS. So, due trade-offs need to considered during design.

Secure your APIs

Security is a cross cutting concern in enterprises. So, there will usually be an organisational security policy that guides the application security. It is an industry-wide practice to adhere to ‘***Defence in Depth’*** with many layers of security while application & data security being just two layers in it. Security is an elaborate topic per se. I have explained in detail about it in a dedicated section on the security best practices for cloud deployment of APIs in this book.

It is a common practice to have a separate dedicated service for authentication and authorization across the enterprise and use an API gateway to handle incoming requests before they even hit the individual services.

However, it is vital for the service to assume nothing and practice a ***zero trust policy***. What it means is that while it is not its job to authenticate or authorize requests per se, the service must validate the incoming request token by delegating it to the authorization server and only allow request if the token is valid and has the required set of permissions to access the endpoints.

Many frameworks have provisions to additionally secure methods/ routines/ functions based on the role of the incoming request user.

For example, Spring Security for Java based applications has support for @Secured annotation over a service method that can be used to secure the method to calls from a user with a given role only.

All communication from clients to services must happen over TLS encrypted channel. Set up mTLS for secure communication internally between services. Consider using an API Gateway or a service mesh to address a lot of cross-cutting security concerns.

Either by using the organizational security infrastructure or adding your own to secure your APIs, a security strategy is a must.

Caching

Every request usually hits the database. Every time you fetch something from the DB it involves some latency. To pack better performance in your APIs you can cache the data from previous responses. On the downside the returned data could be out-of-date. So, it is important to have an expiry time set for each object cached after which it is evicted from cache.

Caching can be supported at different levels:

If you use an ORM framework like Hibernate it offers a first-level cache and a second-level cache that can reduce DB hits.

There are in-memory caches like Redis and Apicache which can cache data and return from the service layer of the application without even hitting the DAO layer. API gateways can be configured to cache and return data without even hitting the application.

Browsers cache webpage content in a local or private cache for a single user for faster load time.

CDN (Content Delivery Network) servers cache images, videos and webpages to reduce latency.

Be sure to use the Cache-Control header to specify how the clients should handle caching of data.

Pagination, Sorting and Filtering

The data returned by a GET request can be very huge. If all of this data is handled all at once from querying the DB to building a JSON response to displaying the data on the UI it can be a performance bottleneck and slow down the application considerably. Especially if it is a popular workflow and has a lot of users performing the same query in the same time period it can be a very bad user experience.

Pagination allows the user to define the page size and the page number they are looking for, so that the server can return the results only for that page. Databases often have a way to specify OFFSET (page number) and LIMIT (page size) on the query so that only the expected subset of data is fetched.

For example:

```
GET /v1/@JaneDoe/articles?page=3
```

displays the page 3 of the list of articles by author Jane Doe.

```
GET /v1/@JaneDoe/articles?page=3&page_size=5
```

limits the number of articles per page to be 5.

Support users to specify how they want the data returned to be sorted.

For example:

```
GET /v1/@JaneDoe/articles?sort=+likes&-published_date
```

displays the list of articles by author Jane Doe sorted by number of likes and then by date of publishing the article. '+' and '-' are used to indicate the ascending and descending orders.

Allow users to filter the content returned based on additional criteria.

For example let us say a user wants to search for articles by Jane Doe but only if the article contains the words 'cloud' and 'containers'.

```
GET /v1/@JaneDoe/articles?search=cloud,containers
```

or say we want to search for an article with a given title

```
GET /v1/@JaneDoe/articles?title=That%20Wonderful%20Book
```

Documentation

It is very easy to miss this one. Far too many wonderful APIs have been rendered useless because of lack of proper documentations. I cannot stress this enough — ***document, document, document***. If the consumers cannot understand what the API does, what requests are expected or what responses to expect, it cannot be impactful. There are many frameworks to make a developer's life easy while creating meaningful documentation for APIs.

For example — Open API Documentation (Swagger) is a very popular and simple framework to create highly consumable documentation for your APIs. It offers excellent integration into Spring applications and easy documentation using annotations.

Proper request validation and Exception handling: Return standard HTTP error codes

You are working on a critical application and the server responds with a generic 500: Internal Server Error. You try to find out from the error message if there is anything you can do to get it working but the application has suddenly decided that you are its mortal enemy and does not want to talk to you. There is no meaningful detail in the error message that you can use to work around or even understand the problem. And you hear yourself exclaiming '***Et tu Brute!***'

You do not want the users of your application to go through this experience.

The trick is to validate all incoming requests to see if they are inline with what is expected by the service. Do not assume the users to always send the right kind of requests. There could be null values or invalid values in the request parameters. Be sure to validate it all.

The second part of this equation is to handle all exceptions and converting them into appropriate HTTP Status codes with details.

Do not return the exception trace as it is. Most users cannot read it or cannot make sense out of it. Do not send just the HTTP Status Code and generic error message with it. It is insufficient.

Offer details about why such an error occurred and if there is anything a user can do to work around it.

For example: Let us say a user tries to login without creating an account first. This obviously will lead to a UserNotFoundException from the service. Sending a response with empty body or some cryptic error message will cause more harm than good. Also, returning a response with HTTP Status code 500 with just the exception stack trace does not help much either. To the response add a proper message that says "*The user id which you are using to log in is not yet created. First create your account using 'Register User' link.*". Add the link to 'Register User' page using HATEOAS and send it along with the response. Now the user has a way to understand what went wrong and what the user can do about it.

Logging and Monitoring

Add a logger with different levels of logging to the APIs. In production, log levels are set to ERROR (there should not be many, anyway :)) and in staging logs typically are a little more verbose at DEBUG level. This should help investigate any problems quickly and address them.

Monitoring and collecting operational statistics of our running application is vital to keep track of how the application is performing and behaving in production. There are several frameworks that help achieve this at an application level and at an infrastructural level.

For example — Spring Actuator provides several endpoints like /health, /metrics, /info, etc that provide details about the application and integrate well with Spring applications so well that monitoring becomes a trivial problem.

Prometheus provides monitoring and alerting using time series data at a container runtime level. Actuator provides Prometheus integration as well which makes it very easy to visualise all data using a tool like Grafana.

Testing

Goes without saying. Unit test each API endpoint thoroughly with all combinations of possible request data including all null values. Be sure to design and develop the endpoints with Inversion of Control (IoC) and Dependency Injection (DI) rather than hard-wiring the instances.

APIs must be tested with Unit tests, Performance tests, static security analysis tests, dynamic security analysis tests (runtime), container security tests, Integration tests, Functional Tests, User Acceptance Testing (UAT), Smoke tests and Regression tests. Automate and ensure good code coverage.

Versioning

Support at least two versions of the API at any point of time. Typically versions can be specified in the endpoint URL like so:

```
GET /v1/articles


GET /v2/articles
```

API changes from vCurrent to vNext must be announced in advance. If any functionality is getting removed, do not remove it in vNext. First deprecate it in vNext and then remove it in vNext + 1. Any deprecations from vCurrent must be included in the announcement.

Usually the release cycle repeats as follows: announce the End of Service (EoS) for vPrevious, then announce vNext General Availability (GA), then GA vNext, then announce the EoS for vCurrent, then announce vNext + 1 GA and so on.

Use canary releases to deploy new versions. It involves deploying the vNext with all requests still routed to vCurrent. Once all testing is completed, route only 5% of the traffic to vNext and then slowly increase to a 100%.

Backward compatibility

Ensure the API changes are done in a way that is backward compatible. Do not remove information from the return data in a new version but add information as required by the change.

For example

Let us say in v1 of the API, we are displaying the author name for an article.

So typical response for a GET request like

```
GET /v1/articles?author_name=abc HTTP/1.1
```

would be

```
HTTP/1.1 200 OK

Content-Type: application/hal+json

{

    ...

      "author_name": Jane Doe

    ...

}
```

And then we create a new v2 for the API where we want to split the author name into first name and last name. Let us say v2 also offers additional information with HATEOAS links. So, we can replace the author_name with first name and last name in v2. But what about the consumers of the v1 of API? They cannot upgrade to v2 even though they might want to consume the new links added in v2 because it breaks the previous functionality.

However, adding the newly created fields along with author_name will help retain functionality.

```
HTTP/1.1 200 OK

Content-Type: application/hal+json

{

    ...
```

```
        "author_name": Jane Doe,

      "first_name": Jane,

      "last_name": Doe

      ...

}
```

Now the new consumers who want to use first_name and last_name can use those fields and the consumers moving from v1 to v2 who want to continue using author_name but want to use links can safely use v2 without breaking functionality. Obviously this example is very simple (for brevity) and the real world problems and use cases tend to be much more complex, but the driving principle remains the same.

Return a single object in response body

When sending response to a REST request many a times it is quite common to make the mistake of sending a list.

For example: It is very tempting to send a list of articles matching the search query for a GET request like /articles?author_name= abc. But consider the implications for a moment. If the endpoint changes and we no longer want to send just a list of articles but also some author info along with it, it becomes annoyingly difficult to do that. The API needs to change in a way that is no longer backward compatible.

So, send an object as a response that wraps the list. This way we ensure that when we need to change the response to include additional information it can be handles elegantly.

Rate limiting, Circuit breakers, Throttling

There are design patterns which help build a robust API that can handle failure elegantly.

Rate limiting an API defines the number of requests a user can send to your API in a given period of time. Any requests beyond the limit will return with HTTP status code 429: Too many requests.

Most API Gateways provide this functionality out of the box.

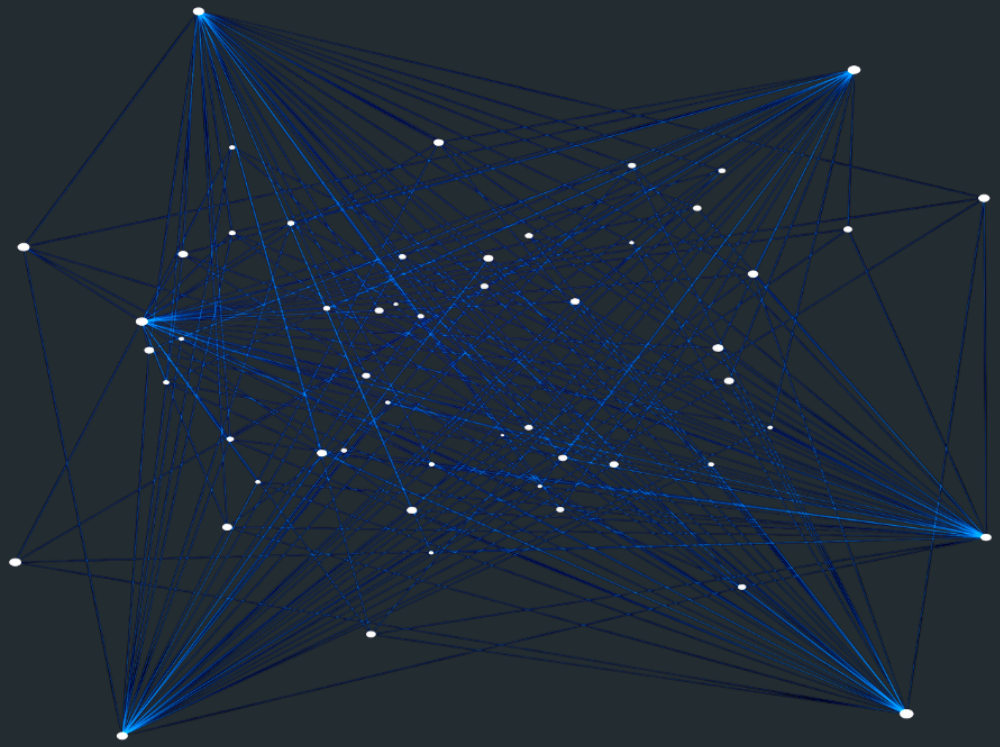
Circuit breaker is a design pattern to prevent an API from calling a failing operation over and over expecting a success, when it is likely that the operation will continue to fail. Many frameworks provide this functionality out of the box. Netflix Hystrix is a very popular example.

User loads vary on an API depending upon time of day or business hours, etc. This can put overload on the resources and cause them to fail or partially unavailable. One way to handle this is via autoscaling. But autoscaling involves provisioning of additional resources and that takes some time. The services might be unavailable temporarily during that period. Alternatively, when the resource consumption by APIs reaches a certain level, the requests can be throttled to ensure the APIs can continue to work. One strategy could be to suspend low priority operations until the high priority ones are completed or additional resources are provisioned.

Parting thoughts

Like I said when we began this section , it is very easy to design an API but very difficult to do it right. In this chapter I have outlined a few best practices that will help you consider some important design aspects while architecting your REST APIs for your microservices. Once you have your API designs ready the next problem to solve will be to decide how these APIs will work together to solve a problem. We head into the next section to understand microservices communications and the several trade-offs involved in it.

MICROSERVICES COMMUNICATION & DISTRIBUTED TRANSACTIONS



One of the very important areas to consider during architecture design phase is - how the microservices are going to communicate with each other. Have a strategy in place that covers all microservice communications.

1. **Do you need synchronous / asynchronous communication between services?**
2. **Security considerations**

   Secure the communication channel between microservices using mutual TLS (mTLS). It can be done using a service mesh that secures the east-west traffic.

3. **Performance implications**

   Inter service communication is inevitable in microservice architecture. However, it is a good idea to limit the chatter over the east-west traffic. During design do consider if the services will become very chatty and what is the performance penalty involved. Maybe it is a case where two services are so tightly coupled in the business context that they are part of the same use case. It might be prudent to merge them into a single service. There is nothing wrong in designing and developing a monolith or a macro-service after due diligence and if it is deliberate. It is not bad architecture. You are just prioritising strong consistency and performance over availability and scalability. Likewise it is absolutely fine if your business needs you to favour higher availability and be eventually consistent.

Synchronous communication between microservices

Microservices can issue synchronous calls between each other when the use case justifies it. But one needs to be careful about long running operations in another microservice if a synchronous call is made to it from your microservice. Do consider other alternatives like modular monoliths or eventual consistency.

If you choose to use synchronous communication it is pretty straight-forward to issue a call from one microservice to another. It can be via a REST call, for example. However, it is quite expensive to do it and considered an anti-pattern because it increases coupling and reduces availability.

Asynchronous communication between microservices

Microservices are built for asynchronous communication. It makes the design robust and loose coupling provides high availability. Asynchronous communication is typically achieved with a distributed event store acting as an event bus between microservices.

Although asynchronous calls provide better performance they can create consistency issues. The design leans in favour of eventual consistency. This is discussed in detail in the next section.

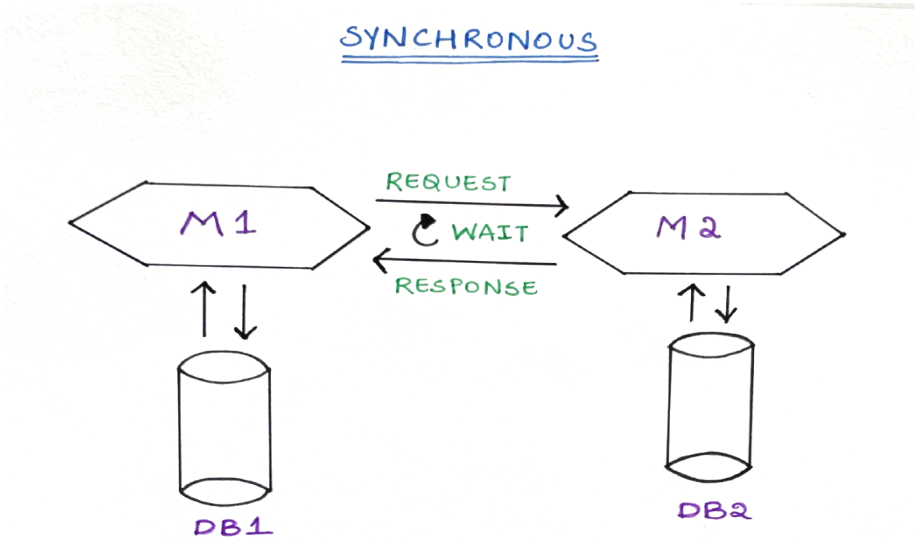


Figure 1: Synchronous communication between microservices

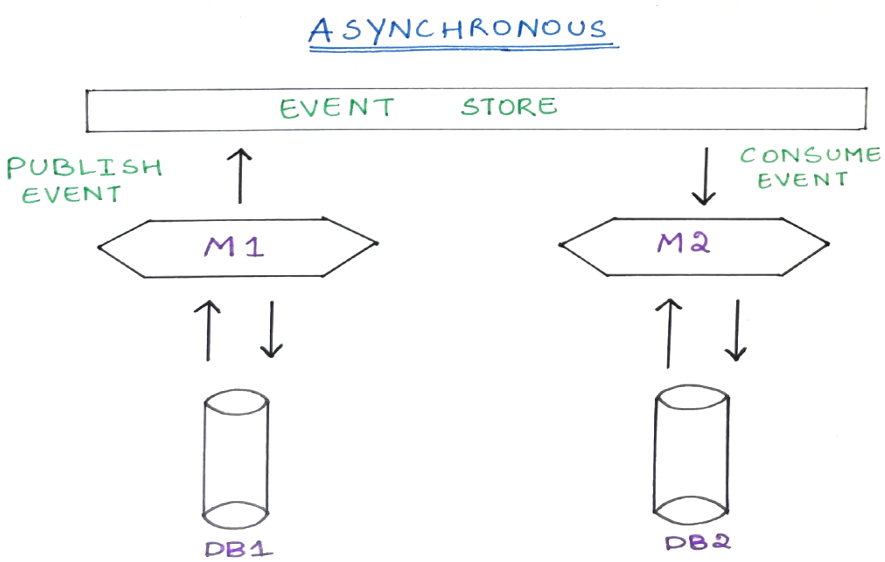


Figure 2: Asynchronous communication between microservices

Transaction Management

In a monolith transactions are ACID (Atomic, Consistent, Isolated, Durable) compliant. Since monoliths run as individual processes sharing the same runtime and using a single database instance, transactions can be handled seamlessly. But that is hardly the case with microservices.

Microservices are designed to use their own databases and run as separate processes.

That could mean some part of data will be handled by microservice M1 and committed to database DB1 and some other part of data will be handled by M2 and committed to DB2, as part of the same transaction. Or it could mean that a microservice must write to a combination of multiple databases or message queues/ event stores.

Trade offs need to be drawn between consistency and modularity. When you split the functionality into multiple microservices you give away strong consistency between operations. You are trying to reduce the affinity between services to make them independent and therefore they operate in an asynchronous fashion. Asynchronous operations tend to be eventually consistent.

If an operation needs to be highly consistent and you are considering the operation to be spread across two or more microservices, then reconsider. It will be prudent to use a modular monolith instead of microservices in this scenario.

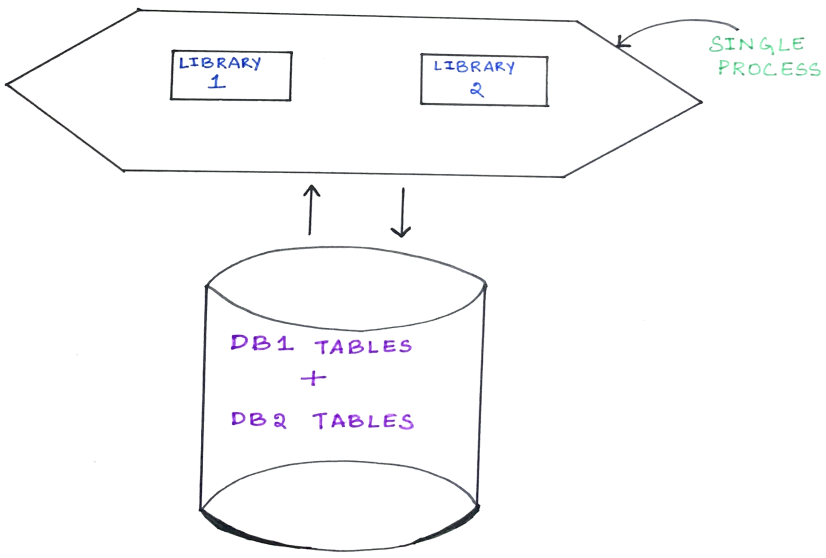


Figure 3: Modular monolith running as a single process with a single database

However, let us say that there is a business case with an operation O1 consisting of 3 steps:

1. Commit to DB1 //needs to be highly consistent
2. Commit to DB2 // needs to be highly consistent
3. Commit to DB3 // book keeping. Can be eventually consistent

The first two steps have very high consistency requirements while the third step is okay to be eventually consistent since it is just logging some information to DB3.

So, we can handle the first two steps in microservice M1 and then put out a message to microservice M2 to handle step 3.

This means that there must be distributed transactions in M1 & M2. Distributed transactions have higher performance penalties and can get quite complex to manage. It is recommended to avoid distributed transactions, ideally. But we don't live in an ideal world, do we?

Two Phase Commit (2PC XA transactions)

One way to handle a distributed transaction spanning across multiple services is to use XA (eXtended Architecture). XA is a 2 phase commit (2PC) protocol by X/Open group standard.

XA transactions are designed to have a transaction manager which controls the commits across resources. Each local transaction has a resource manager attached to the resources.

The resource managers are enrolled into the transaction manager. So, when the transaction manager commits, each resource manager commits its own local transaction. If any resource manager reports a failure, the transaction manager rolls back all the commits.

Using XA, the scenario that we described earlier can be handled as below:

in Microservice M1

```
XATransactionManager tm = new XATransactionManager (XAResourceManager rm1, XAResourceManager rm2, XAResourceManager rm3);
  try {
        tm.begin();
        // Some SQL statements for relational database DB1 (mapped to rm1);
        //Some statements for database DB2 (mapped to rm2);
        // Some messages produced and sent to an event store ES (mapped to rm3);
        tm.commit()
      }
  catch(Exception ex){
        tm.rollback();
       //set your house in order
  }
```

Separately in microservice M2

```
{
  try {
      // Init transaction tx
      tx.begin();
      // Consume the message from the event store ES
      // Process data
      // Execute changes to DB3
      // Produce messages to ES
      tx.commit();
      }
   catch(Exception ex){
        tx.rollback();
   }
}
```

M2 can choose to create its own 2PC transactions if there is a need or it can handle the operation as a set of local transactions with idempotent consumer pattern.

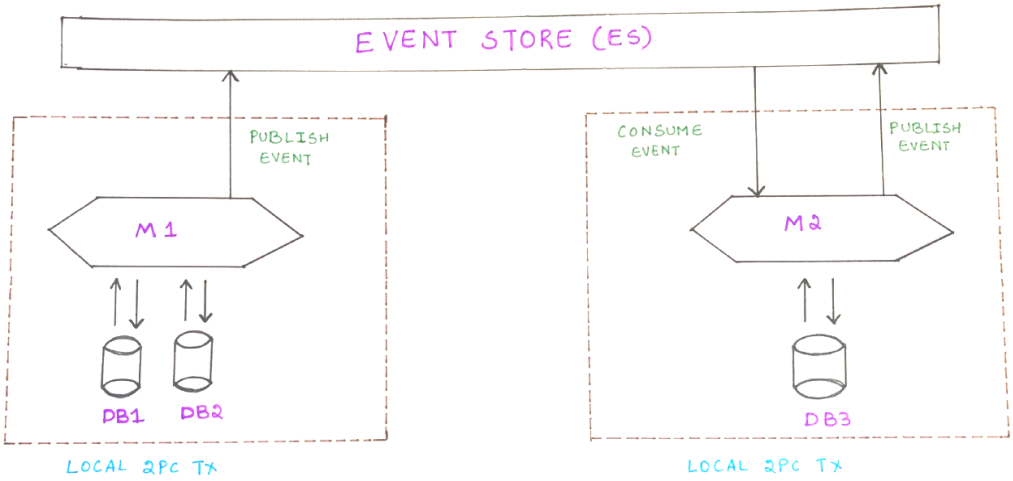


Figure 4: Local 2PC transactions in microservices

M1 is synchronous and highly consistent in its commit to DB1, DB2 and ES. M2 is synchronous and highly consistent (if 2PC is used) in its commit to DB3 and ES. However, the overall system spanning across M1 and M2 is asynchronous and if M2 fails in its local transaction there is no way for M1 to rollback its changes.

But what if the commit to DB3 is required to be strongly consistent?

Let us say the requirement is to commit to all the databases DB1, DB2 and DB3 at once or all must fail. There are use cases like a banking transaction where eventually consistent transactions can create confusion and bad user experience. In such use cases it is better to use a good monolith to design solutions.

If you want synchronous and highly consistent distributed transactions within a micro service you can use 2PC. But if you need it across microservices then consider a redesign. When you try

to achieve consistency across microservices using 2PC, you are using a tool that was not quite meant to do that. Different microservices run as separate processes and thus they cannot guarantee that the local transactions inside each microservice can be globally atomic.

There are frameworks that support distributed transactions across REST APIs. But the associated performance penalty is very high and it impacts availability of the overall system. Add to this the complex corner cases like '*what if the transaction coordinator goes down for a very long period*', you get the picture. It often leads to strong coupling, unwanted dependencies and the architecture starts to get a little more complex than you would like it.

Figure below shows the best case success scenario in a 2PC transaction across microservices.

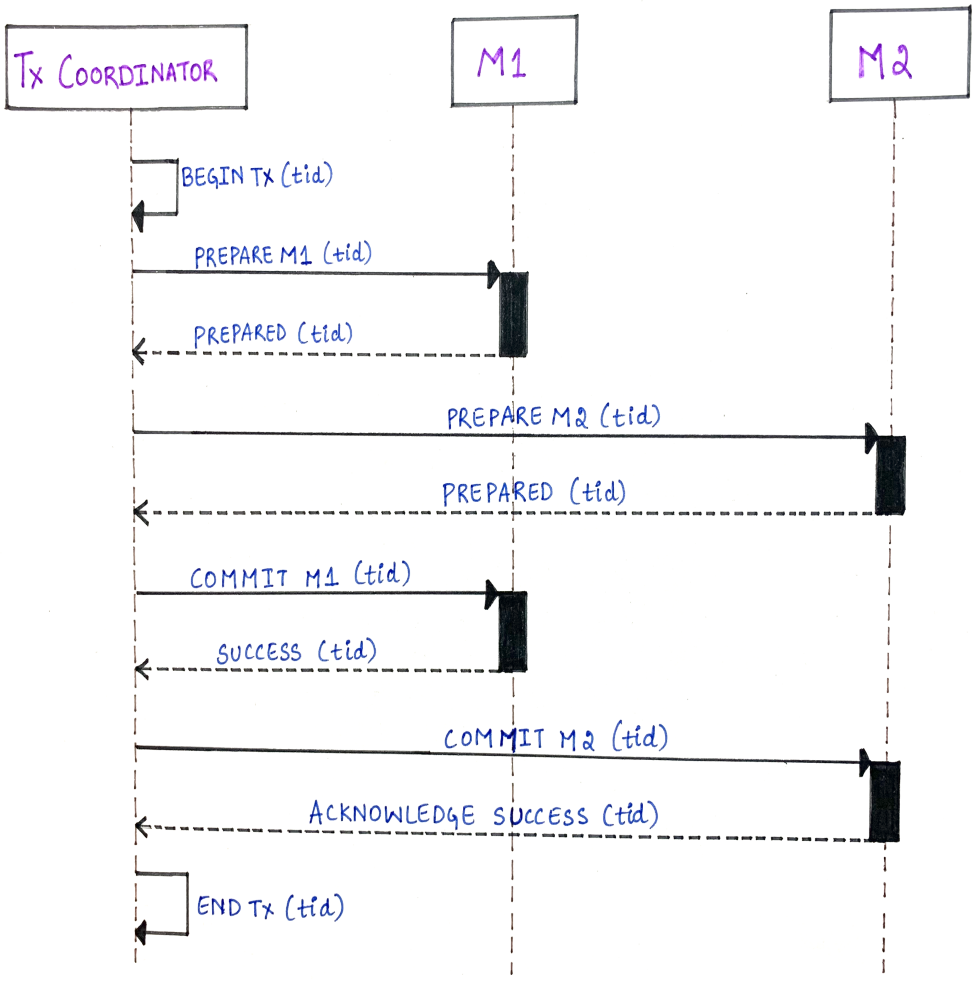


Figure 5: 2PC transaction between microservices – success scenario

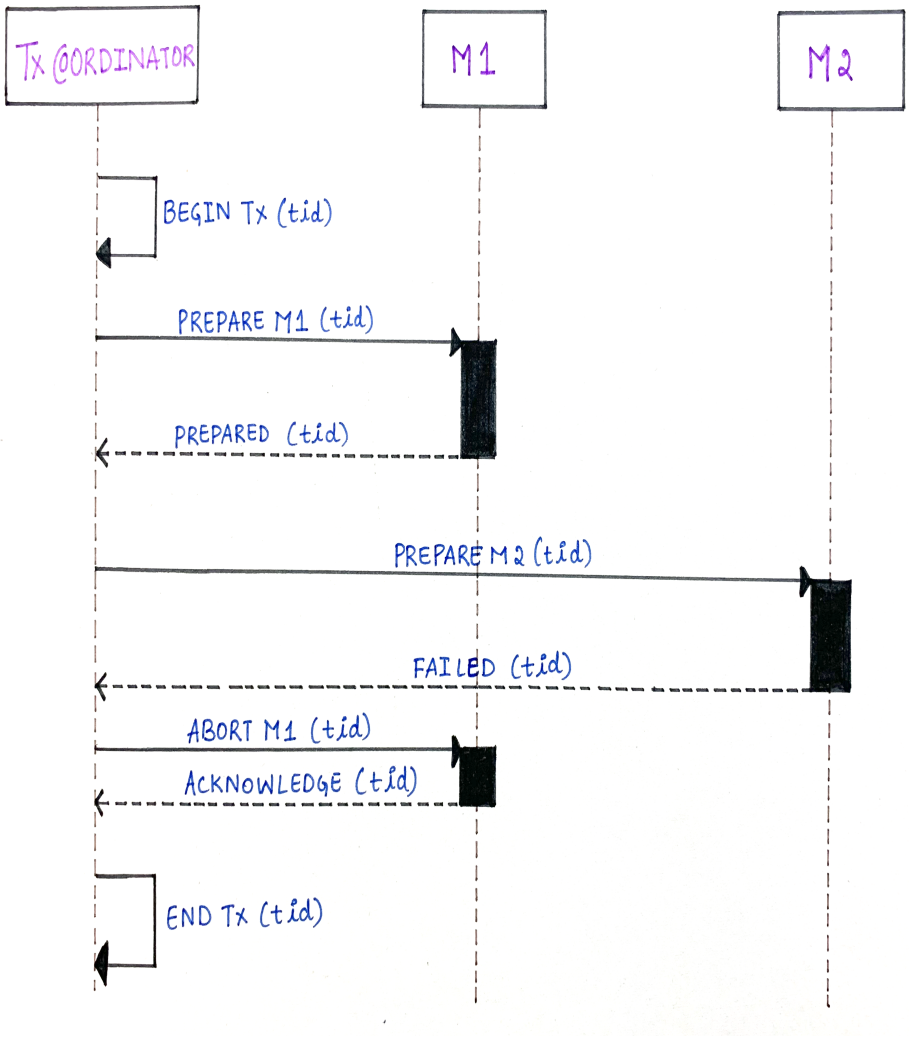


Figure 6: 2PC transaction between microservices – failure scenario

2PC is not a very good fit with microservices architecture for the scenario we just discussed. So, what are the alternatives? In most cases it is acceptable for the system to be eventually consistent. These cases can be supported with microservices executing their individual local transactions and communicating to other microservices about the outcome. And then the services involved in the work flow coordinate with each other to either reach a desired end goal of a successful distributed transaction or in case of a failure in one or more local transactions

compensate the earlier local transactions by issuing a set of transactions to eventually bring the system to a consistent state.

There are a couple of ways to go about it.

Orchestration

One of the services can take up the responsibility of being an orchestrator who leads the interactions between services during the transaction work flow. The orchestrator issues requests to other microservices to execute their end of the transaction or issue requests to compensate in case of a failure.

From the example with microservice M1 and M2 earlier,

Let us modify it a little bit. Microservice M1 commits to DB1, M2 commits to DB2 and M3 to DB3.

Microservice M1 can act as an orchestrator with the work flow as below:

1. M1 instantiates the orchestrator O1.
2. O1 issues commit to DB1.
3. O1 sends a command to M2. This can be, for example, using a message broker with M2 acting as an idempotent consumer with ability to retry its operation.
4. M2 commits to DB2.
5. O1 acts as a consumer to the result of the operation on M2 (direct response or via a message broker).
6. M2 responds with the result of its local transaction.
7. If M2 is successful, O1 continues its happy flow. If M2 reports a failure, O1 issues a set of compensating transactions to commit to DB1. That ends the saga.
8. If successful, then O1 now issues command to M3 to commit to DB3.
9. M3 responds back with a status. If successful that completes the transaction. If not, then O1 issues a set of compensating transactions to commit to DB1. Then issues a compensating command to M2 to compensate the earlier DB2 commit. That ends the saga.

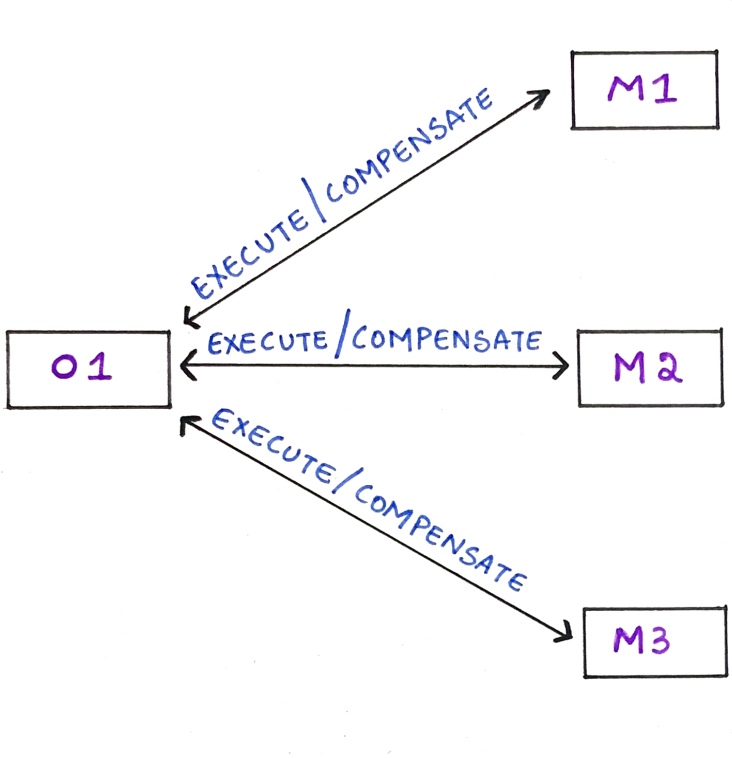


Figure 7: Orchestration with three microservices

Choreography

Choreography is a saga transaction pattern where each microservice executes a local transaction and updates its data source and passes on a relevant message to the event store. Other microservices pick this up and execute their local transactions and the process continues until all individual microservices participating in a work flow complete their respective pieces of work.

If one microservice fails with an unexpected error then it issues a relevant message to the event store and other microservices issue compensating transactions locally to cancel their earlier commits. There is no centralized control over the entire transaction.

There are a couple of ways to achieve choreography for the same scenario described in orchestration.

Using an event store with 2PC

The participating microservices can generate events to reflect the results of their local transactions using an event store. Other participants can consume those events to either execute their local transactions or compensate.

However, each microservice must handle its own local transaction as a 2PC XA transaction since it involves two steps internally – commit to its own database and then publish an event to the event store.

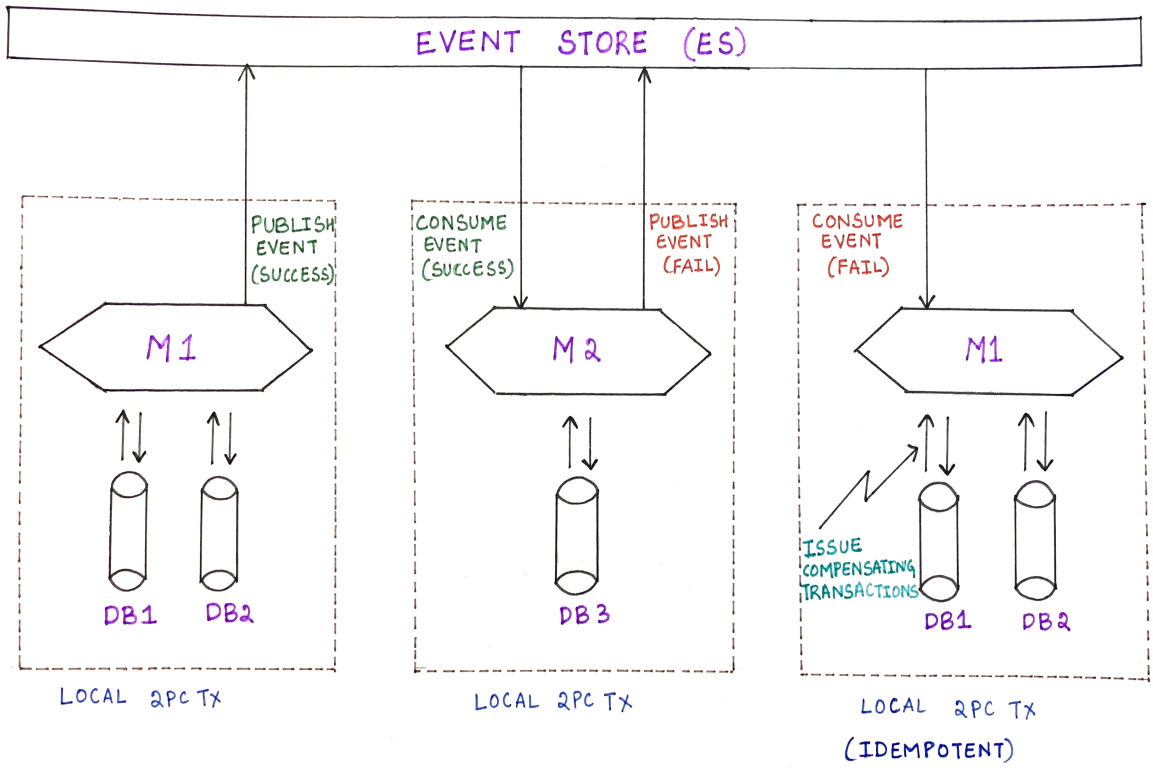


Figure 8: Choreography between two microservices M1 and M2

Using a Change-Data-Capture (CDC) tool like Debezium with an event store

2PC is not very convenient for reasons already discussed previously. You can choose to use a change data capture tool like Debezium to capture the database changes and send events to an event store. Other participants can consume those events and execute their local transactions or compensate.

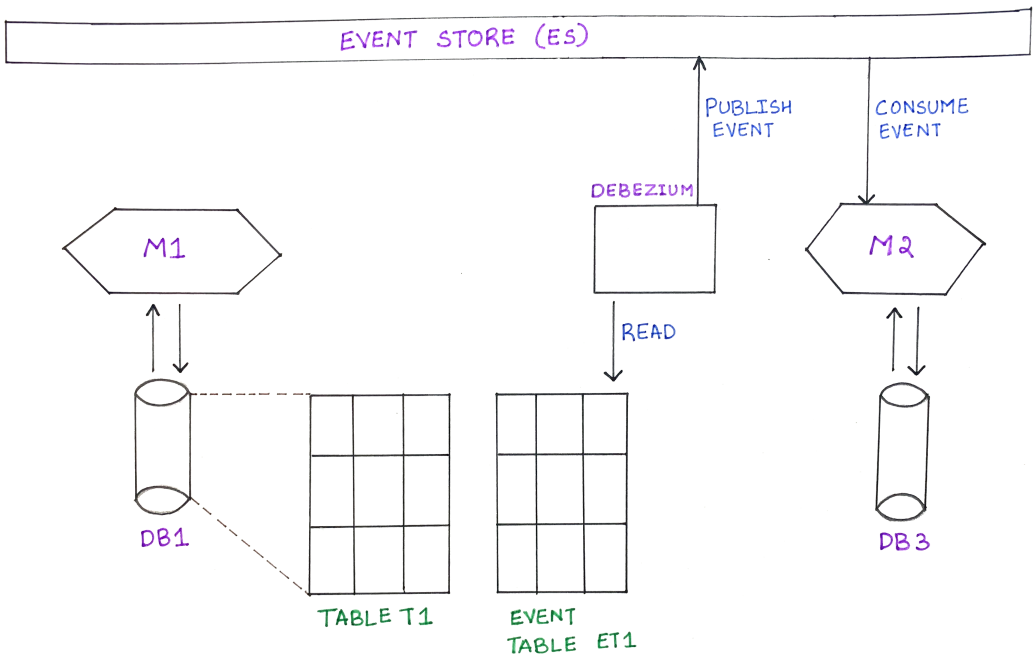


Figure 9: Choreography between microservices M1 and M2 using Debezium instead of 2PC.

Parting Thoughts

Although saga transactions seem very simple at first, there are cases which will come to light during implementation that will make you reconsider your strategy and revise it.

For example: What happens when some other transaction over-writes the data when microservices M1 and M2 are executing a saga. M1 commits to DB1. M2 fails. Meanwhile someone else has over-written the data in DB1. M1 is unable to issue a compensating transaction. Boom! Data corruption! This might not be a big deal in social media websites because it represents less than 1% of cases. But for a banking application consistency is everything, even if it is eventual consistency.

Rest assured there are mechanisms to counter such issues and corner cases like for example using semantic locking.

Now that we have handled transactions in microservices we move to the next section to design our data strategy. Data in a microservices application is distributed because each micro service has its own database. It requires novel strategies to be developed to scale the data while making the system fault tolerant and highly available.

HANDLING DISTRIBUTED DATA FOR YOUR CLOUD APPLICATIONS

Data is the most critical aspect of enterprise application software. Be it a monolith or a microservices architecture there could be a need to distribute data. However distributed data strategy is non-trivial and needs to consider several trade-offs.

Especially if you are using microservices there is a chance that you will need to distribute data at some point. It could be because the application has grown exponentially and now the data needs to be split across multiple data sources or you may have a Command Query Responsibility Segregation (CQRS) design pattern implemented where the writes are handled by a different service and the reads are handled by a different service (and hence data is written to one node and read from multiple different nodes).

If you are using a single data source and the data is growing then there is an option to vertically scale it. It does not require you to distribute the data across multiple nodes of data source or horizontally scale it. If you have a traditional application it is likely using a relational data base. Relational databases (RDBMS) have been around for a while now. They form the data backbone in most enterprises. The data is structured and expressed as a schema with assured ACID compliance.

With vertical scaling, as the data grows it needs stronger and bigger servers (more CPUs, more RAM, etc) to handle the kind of load that comes with volume. However, there is a limit to how much you can scale vertically and it presents performance problems after a threshold. Thus arises the need to scale horizontally.

Also, to improve availability and fault tolerance you need to consider replicating the data.

When you develop a strategy to distribute the data, do consider the trade offs involved.

The most basic questions we normally start with -

- What is the volume of the data?
- Is the data structured or unstructured?
- What is the nature of data? Is it relational or non-relational? Does it need to be highly consistent?
- What is the performance penalty involved in distributing my data? Is it acceptable?

To handle high growth data (for example consider a social media app where the user data just keeps piling up) or unstructured data, it can be done using a NoSQL database.

The data is stored like a self-contained document without the need for joins. So, it is easier to scale horizontally than in a relational database. NoSQL databases favour high availability over

high consistency.If horizontal scalability is your design preference and eventual consistency is an acceptable trade-off consider a NoSQL database.

Distributed data trade-offs: CAP Theorem

For a distributed datastore it is necessary for the system to make trade-offs between Consistency, Availability and Partition Tolerance (CAP). CAP Theorem formalises these trade offs.

Consistency:

For a distributed data system to be consistent all queries should return the same data irrespective of the node which is queried. It means that the data is instantly replicated and all nodes have the same copy of data.

Obviously it takes finite time for data to replicate across all nodes. So perfect consistency is impractical. But for the purpose of this discussion we assume the time taken to replicate is zero.

Availability:

A distributed data system is called highly available if all requests get a valid response even if one or more nodes fail. So, a request can be sent to an alternative node to get a response if a given node is unavailable thus making the system highly available.

Partition Tolerance:

A partition is a scenario when there is a communication failure between nodes. Thus the nodes are not able to replicate data between themselves until the communication is restored. A partition tolerant distributed system is able to continue to operate even when there is a partition.

Partitions are inevitable in a distributed system. So it is impractical to design a distributed system that favours consistency and availability and trades off partition tolerance.

CAP theorem states that, in a distributed data system, in the presence of a partition you need to trade-off between consistency and availability.

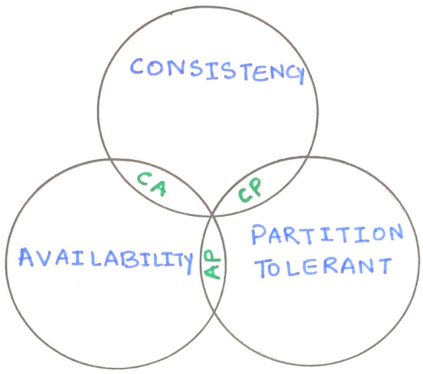


Figure 10: CAP Theorem

So we predominantly see two kinds of systems:

CP systems:

In a partition tolerant distributed data system, if you favour consistency over availability it is called a CP system. In a CP system if there is a partition then the nodes which are unreachable are removed from the system until the communication is restored. There are CP systems which make the entire cluster go 'down' or some which prevent writes or some which allow read and write operations only to main node which is inside partition. All are reasonable approaches.

For example: Let us say there are 3 nodes N1, N2 and N3.

An operation writes to N1. The system immediately replicates the data to N2 and N3. So whenever there is a read operation on any node the same data is returned there by ensuring consistency.

But let us say there is a partition with N3. A CP system will remove the node N3 from servicing any requests until the partition is resolved. This is done because N3 might be holding stale data and will not be able to update its copy of data until it is able to communicate with other nodes.

N3 is made unavailable thus trading off availability in favour of consistency.

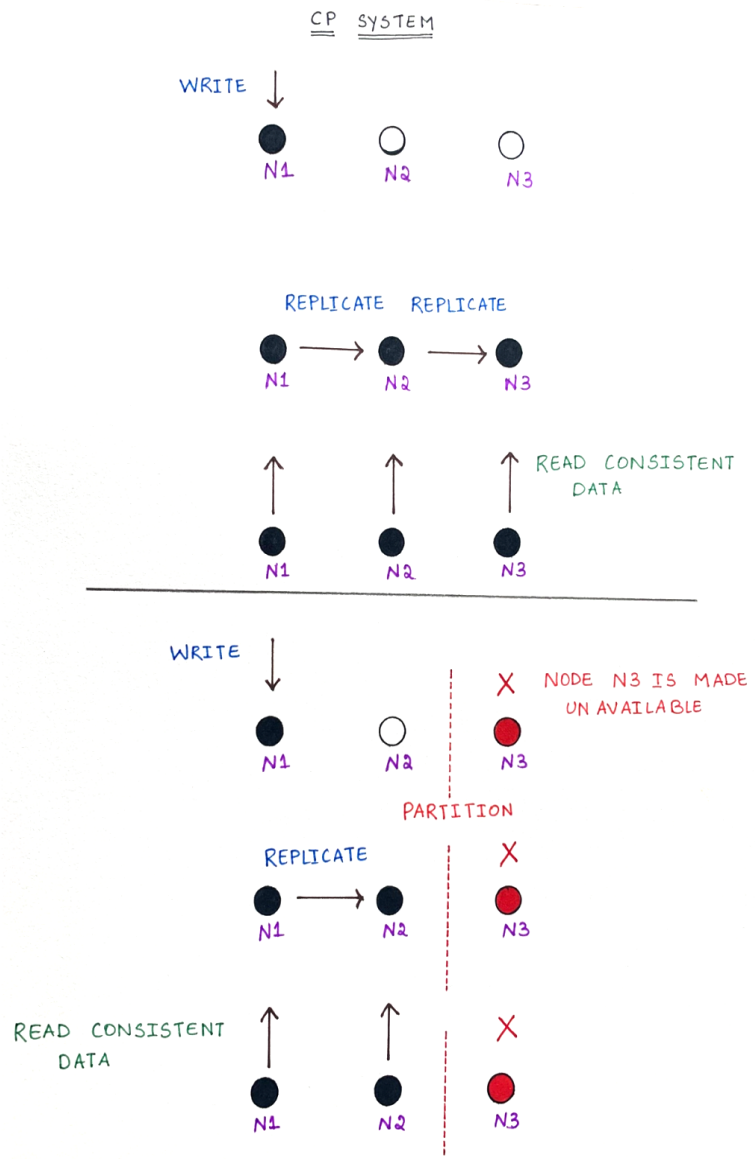


Figure 11: CP system scenario

AP systems:

In a partition tolerant distributed data system, if you favour availability over consistency it is called a AP system. In an AP system if there is a partition then the nodes which are not reachable continue to operate with potentially stale data. The data might be outdated since there there is no communication with other nodes and hence the responses might return inconsistent data. But the system is highly available since none of the nodes is taken out of service.

Let us consider our earlier example with three nodes N1, N2 and N3.

When there is a partition with N3, instead of removing N3 from the system, we let N3 continue to service requests with potentially inconsistent data thus favouring availability over consistency.

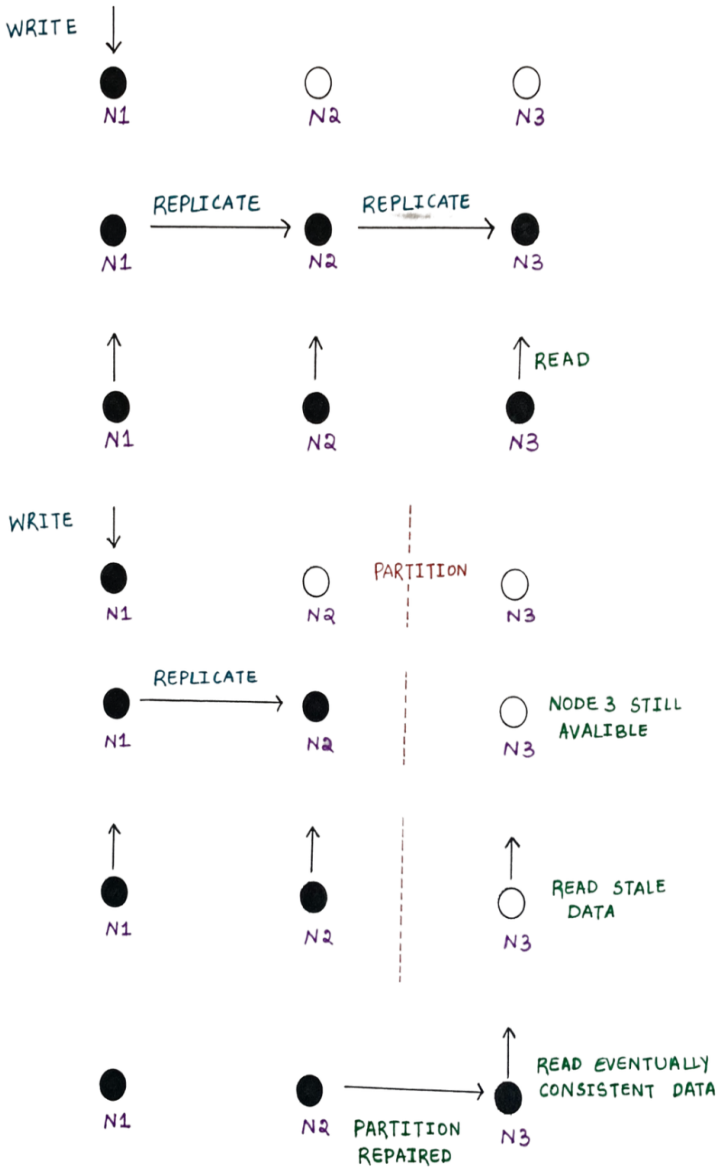


Figure 12: AP system scenario

While CAP theorem talks about the trade-offs between consistency, availability and partition tolerance, there are a few other data distribution strategies worth noting to help your application achieve better scalability, fault tolerance and availability while lowering the latency.

Read Replication

In a typical application the reads are more than the writes to the database. It is called the read write ratio. For applications with a high read write ratio, it is important to ensure that the application instances are able to read seamlessly without latency introduced by writes or other reads.

This can be achieved with read replication of data with the creation of additional nodes which act as replicas for read operation.

The data replication can be done synchronously or asynchronously.

- Synchronous replication

  During a write operation the data is written to a primary node. The success response is sent only after it gets replicated to secondary nodes. This obviously introduces higher latency but ensures high consistency.

- Asynchronous replication

  Once the data is written to the primary node the success response is sent. The replication happens asynchronously between main node and replicas. The latency is lower but it comes at a cost of lower consistency. There is a chance that the secondary node responds with stale data during a read operation because it has not yet received the update from primary node.

There are three ways to achieve read replication:

- Single primary node

  There is a primary node which takes in writes and there are multiple secondary nodes for reads. The data gets replicated from the primary node to the secondary nodes. This strategy ensures better fault tolerance but low throughput because there is just one primary node.

  If primary node fails, then one of the secondary nodes is elected to be a primary node and thus ensuring better availability.

- Multiple primary nodes

  To improve throughput, you can set up the system so that there are multiple primary nodes spread across multiple data centers with replicas in each data center.

- No primary node

  This is a strategy where there are no primary nodes and all nodes are primary nodes. It means all nodes are replicas that can take write operations. This is an asynchronous replication strategy thus providing very high fault tolerance, availability and low latency but compromising on consistency.

Sharding

With read replication you can ensure the system can offer better availability, fault tolerance and low latency. But what about scale? What if in your application the write volume increases due to business expansion and more customers are onboarded?

For example: Let us hypothetically say you are using a relational database and you have a 100,000 customers. That corresponds to a 100,000 rows in the Customer table. Now, the business grows exponentially and very soon you have a 1 million customers and then 10 million. So, 10 million rows in the same table does not perform as well as a 100,000 rows. As discussed before, you do some vertical scaling with better servers but that hits a ceiling at some point. You need other options.

Sharding offers a way to horizontally scale the database. You create a shard key and write the data to multiple nodes based on the shard key.

Fo example: A very trivial shard key for the above Customer example could be customer_id % 10. This divides the customers into 10 shards with 1 million customers each for a total 10 million customers. There are many advanced ways to create a shard key that can help split the data writes evenly across shards in an optimal way.

In NoSQL databases sharding is far easier than in relational databases because of the way data is structured. In case of relational databases, it can get quite tricky to shard data. You must be careful about the table joins and query strategy. You can split the tables to colocate the related data into one shard and perform joins internally to the shard. You can also perform a cross-shard join or a distributed join but be aware of the performance penalty you pay with each trade-off.

Many relational databases offer excellent out-of-the-box support for sharding.

Sharding + Replication

You can use a combination of sharding and read replication to achieve a good trade-off between high availability, high consistency and low latency while ensuring good fault tolerance.

While designing your system for all the trade-offs discussed so far it is important to address security, monitoring and back-up & recovery.

- Security is at the core of all enterprises. All data at rest should be secured using strong encryption. All data in transit should use encrypted channel via TLS.
- Monitor file systems, CPU usage, memory usage, read/ write to disk, etc using standard metrics offered by all enterprise DB.
- Continuously back-up all data for disaster recovery.

Parting thoughts

So far in this book we have designed APIs for our microservices, managed inter-service communications and transactions. In this section we saw various strategies to handle distributed data systems. But the most important aspect of your application is its security which is what we cover in the next section.

SECURING YOUR MICROSERVICES APPLICATIONS

With monoliths, applications and data typically reside in an internal data center and businesses could secure it with one perimeter and have peace of mind. But with cloud all that has changed. But with it comes a change in the way you secure your critical assets as there are a lot more variables involved with cloud than with an in-house data center. The attack surface is much larger on cloud. Below we about some of the best practices to reduce and control the surface area for potential attacks to secure APIs, applications, services and data.

Encrypt all data exchanged

HTTP transfers data in plain text. It means anyone listening in over the wire can extract all the information that gets exchanged as request and response between the client and the server. This is obviously a major security concern. All information exchanged over the wire must be encrypted. This is achieved with HTTPS and TLS. TLS handshake is used to exchange SSL certificates and establish a secure channel for information exchange. All data transfers only via this secure channel thus making it unreadable to any malicious 3rd party. It is imperative to enforce usage of HTTPS for all API endpoints.

HTTP Strict-Transport-Security response header can also be used to inform the client that all communication to the server must be using HTTPS.

Avoid using GET requests to send sensitive information.

Information sent in GET requests gets cached by browsers and gets logged by servers. Sending passwords or any other sensitive information in the GET requests can lead to security leaks. Always use POST requests to send such information in the request body.

Authorisation and Authentication

Authentication is used to verify the user identity. Authorisation is used to verify if the authenticated user is allowed to access the said resource on the server to perform a given operation.

Moving from Session based security to Token based security

HTTP and REST APIs are stateless by design. Every request is treated as a new request. This poses a problem. How will the server know if the user is legitimate for every request without storing any state? If the application is monolithic there is a good chance that the user log in details are maintained in a session on the server and a session id is used to manage it. If there is load balancing involved then the sessions are possibly managed via a shared session cache or using sticky sessions.

It works fine until a certain threshold after which it does not scale. As the number of users increases exponentially the server must maintain more and more states which becomes unsustainable after a point. Also session based security does not work well with microservices architecture whose primary goal is to scale.

Token based security solves this problem by using tokens to pass information between client and server to enable scaling and not storing the user information on the server-side.

OAuth 2.0 (Open Authorisation) is an industry standard protocol for authentication and authorisation widely used by many enterprise applications. JSON Web Tokens (JWT) are used to encode and send claims as OAuth Bearer tokens in request headers. OAuth 2.0 and JWT form the basis of token based security.

OAuth 2.0 supports authorization flows that address several scenarios for server-side (web server applications), mobile, device, desktop, client-side (JavaScript), machine-to-machine applications.

Authorisation Code Flow is used with the server side web applications where the source code is not exposed to general public.

Steps involved in the flow are detailed in figure below.

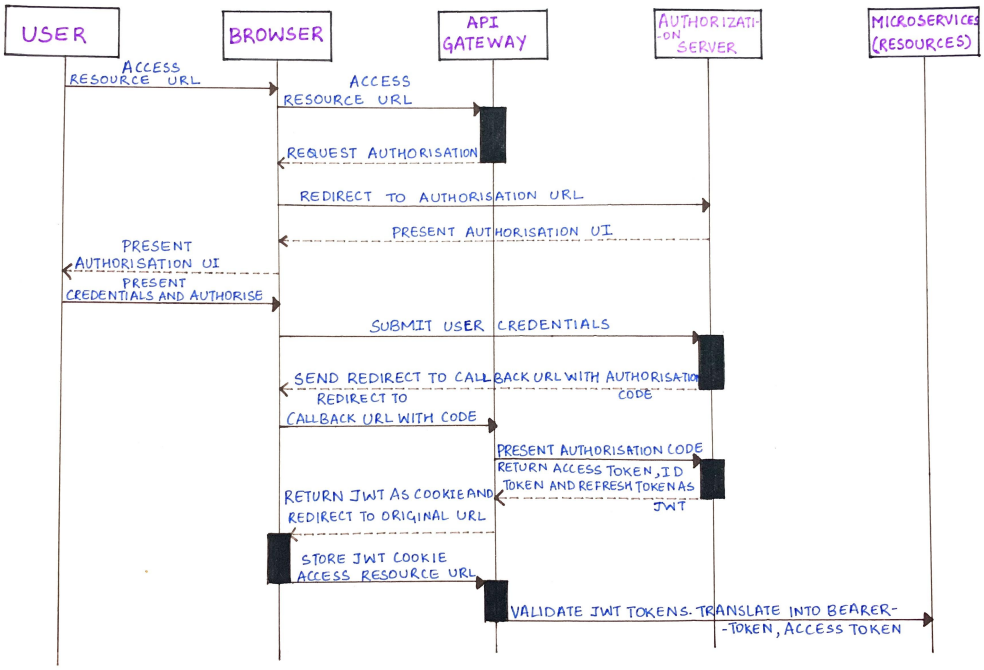


Figure 14: Authorization Code Flow

Apart from the Authorisation Code flow there are other flows supported to cover other scenarios.

Implicit Flow is used with client applications residing on user devices which cannot store client secret securely.

Resource Owner Password Flow is used when the applications are trusted and want to access each others' resources.

Client Credentials Flow is used in machine to machine authorisation where the application is authorised rather than the user.

JWT Tokens

JSON Web Tokens (JWT) are extensively used during the OAuth process and Single Sign On (SSO). They are a compact and secure way to transfer data between parties.

A JWT consists of three components – a header, a payload and a signature.

The header contains metadata information about the algorithm used to sign the token, type of the token, etc.

The payload contains the claims, user id, user name, token expiry,issued at, etc

The signature is a one way hash using a hashing algorithm like SHA-256, header, payload and a secret key. So the signature could be verified by only the server that issued the token with the secret key that only the server possesses.

There is no way to change the payload of the token since the signatures will not match when the server verifies the token.

Multi-factor authentication

Use multi factor authentication for additional security. It protects the application and the infrastructure in case of a password breach.

Dedicated Auth Server

Use a dedicated and trusted Authorisation Server to handle all authentication and authorization. Do not allow individual services or API gateways to create or validate tokens.

Zero Trust Policy

Microservices must not trust anyone else apart from themselves. Each microservice must validate tokens of incoming requests. Requests must have minimal scopes associated only with the target microservice and access tokens from user requests should not be forward chained to other services. Instead a new token must be created for the next request in the chain with a defined scope.

This approach helps a fine grained control over the security of individual services.

API Gateways

API gateways act as entry points to the application. They provide an added layer of security between the client and the services and protect the resources from outside world.

The requests hit the API gateways and then they are transformed and transferred to the actual microservices. So, malicious requests can be spotted and filtered.

API gateways can protect the microservices from DoS (Denial of Service) attacks, SQL injections, and other such security threats because they are designed to add rate limiting, throttling, circuit breaker and handle incoming traffic elegantly.

They can also act as a intermediary to exchange incoming JWT tokens to get the bearer tokens with the Authorisation Server and forward them to the microservices. It is a good practice not to forward the incoming tokens directly to the services.

Service Mesh

While API gateways handle the north-south traffic, service meshes are designed to take care of the east-west traffic (the traffic between services internal to the infrastructure).

Service mesh abstracts a lot of cross cutting concerns and offer capabilities like observability, traffic management and security without having to manage them in individual services. Some of the benefits include mTLS (mutual TLS) to encrypt the traffic between services, routing, access control, canary deployments, monitoring, logging and even authentication.

Services mesh frameworks are evolving to act as API gateways too. In the future it would not be surprising to see a unified solution.

Audit, Log and Monitor

Audit everything regularly with a expert group guiding the security infrastructure. Monitoring and logging are offered by cloud infrastructure, API gateways, service meshes and independent open source products.

Container security

Container runtimes can lead to security breaches if deployed with a wide surface area for attack. Use the following strategies to reduce the potential attack surface and to put checks to ensure safety.

- Use a thin host operating system.

- Always run your containers in the no-new-privilege mode.
- Create containers with non-root users.
- Run containers with read-only, where applicable.
- Do not pack in credentials in the image. Use container orchestration volumes or environmental variables to pass such information to container runtime.
- Use a thin base image for creating containers.
- Check containers for any violations in the common security best practices before deploying.
- Scan container images regularly for any vulnerabilities.
- Use a trusted base image to build containers.
- Set up security gates at every step of DevOps like develop, build, test, package, deploy and operate to check of any vulnerabilities.

Defence in Depth

So far we have covered several best practices to secure the APIs, applications and services running in cloud. But what about securing the cloud itself? How do we do that? Do we set up a strong firewall at the entry point and negotiate all traffic from there? What if there is a leak? Should we not diversify like what we do with our financial investments to mitigate risks? Defence in Depth is a layered security defence strategy to avoid single points of failure. It originates from military strategy where the most critical assets are protected with concentric layers of security. Defence in Depth employs several layers of security mechanisms to safeguard cloud resources.

- Physical Layer – Physically secure the cloud data centers by controlling who has physical access to the data center facility premises, the floors and the data center server labs. Most Cloud Service Providers (CSPs) have well defined processes to guarantee robust security and compliance to international standards.
- Access Policy Layer – Define the access policy to the cloud resources with Role Based Access Control (RBAC). Use deny by default approach and review access privileges regularly for both the cloud administrative teams accessing cloud resources and the users of applications accessing data and services of the application.
- Perimeter Layer – It is sometimes also called the edge because it is the boundary that separates the cloud from the outside world. It is here where you set up your firewall, Intrusion Prevention System (IPS), Intrusion Detection System (IDS). Most CSPs provide

built in protection against DDoS attacks, SQL injections and popular attack patterns with the help of artificial intelligence.

- Network Layer - Secure the open ports and IP addresses accessible to the external internet. Set up logical boundaries to your network and set up network security groups.
- Compute Layer – Compute layer covers the VMs or the nodes. Secure it with antivirus and anti-malware protections. CSPs provide extensive options to secure the compute layer.
- Application Layer – Build security as an essential part of Software Development Life Cycle (SDLC). Integrate security gates at every step – vulnerability testing, static analysis, dynamic analysis and vulnerability scans. Security context should be built into each microservice with a Zero Trust Policy and Deny by Default Policy. Runtime instances should be monitored, logged and audited.
- Data Layer – It is the most important asset that you are securing and therefore it is at the centre of the concentric layers of security we build. Secure all data at rest and in flight. Encrypt data persisted on the file system. Encrypt data sent from and to users and between services. Use data masking for sensitive data like credit card information.

With these layers acting together as concentric circles of defence layers, we create a formidable deterrent against any attacks and protect our key assets.

Parting thoughts

We have covered several security best practices for your microservices in this section. With this we come to the end of our book. In this book I have tried to address some of the design challenges you will face while you architect your microservices applications and the trade-offs involved. I have detailed the best practices and important considerations through various scenarios covering cloud strategy, microservices APIs design, transaction management, distributed data strategies and finally security of your microservices.

I hope you find the book useful and I wish you all the very best!