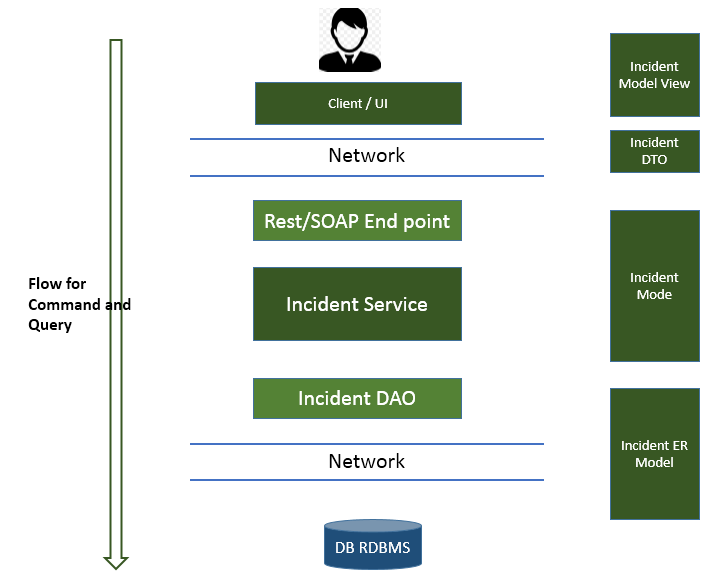
# Micro-service design Event Sourcing

## Command Query Responsibility Separation (CQRS) pattern

Here we move from traditional approach of using the same architecture for writing (command) and reading (query) towards event sourcing and why we should separate the read and write and benefits of achieving it and also the challenges.

### Evolving towards event sourcing

Tradition approach of read and write in a typical monolithic application



### Observations:

Below could be some drawbacks to the above architecture

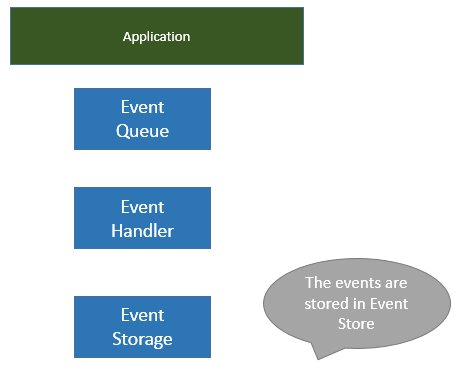
1. We read and write data from same layer/flow.
2. Same model, business service or DAO/DTOs are used for read and write (which might not be always fit to)
3. We use the same deployment model. I.e. reader and writer deployed at the same time.
4. We use the same data store to read and write data
5. We always get the final state of data (after a multiple updates). But we will never know how we reached there. Or what was the state of that particular record on some past business date. (Because we don’t have transaction log always).
6. We can’t scale read and write independently. If read needs more resource that write.
7. No data history or snapshot.
8. Because of using the same data base we need to do read and write in the same data object. Where to improve read performance we may put multiple indexes on database tables where the same index will kill the write performance.
9. This approach tends to create huge monolithic architecture.
10. When we write we write single entity but when we read we read list of entities.

One architecture which approaches these kind of challenges is **Event Sourcing.**

## Event Sourcing

Event sourcing is an architectural pattern where the state of and application is being determined by **series of events**. State is computed by series of events.

### Building blocks of event sourcing



The sequence of events in the event queue is called event stream.

Names of the event should be very expressive and should be in **past tense** as these happened in the past.

Ex:

* ShipmentDeliveredEvent
* CartCheckoutEvent
* CustomerVerifiedEvent

Event should be always **immutable.** We never **delete** event from **event store.**

Usually the event bus or event queue is implemented by a **message broker** architecture. (Kafka, Active MQ, Rabbit MQ).

Well known examples for event sourced systems are:

* Version control systems
* Database transaction logs

The event store has a very high business value, because it consists the complete state of our data from beginning. How it got changed. This could be used for analytics.

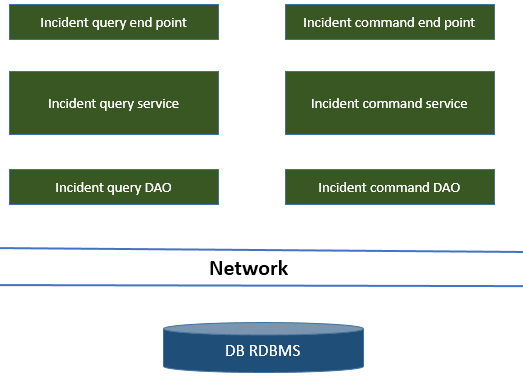
### Performance problems

If we do temporary queries on event store to determine the state of the application we may run into performance problems.

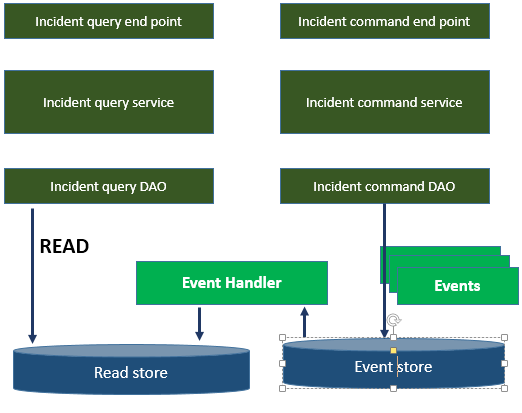
We can pre-compute some application state. A way to create pre-processed application state is **CQRS pattern.**

## First refactoring step towards CQRS event sourcing

In our previous application design we can separate the read and write steps as below



Here the issue we still have is we are running against same data storage, so in above we don’t get too much advantage and insufficient. So we shouldn’t step there. We should design as below.



**Read Store** could be highly optimized, aggregated from performance optimization. Could be a non-relational data base.

### Benefits with above model

We can individually scale read and write based on need. Or auto scale at a particular time of a business day when write is more or read is more.

We have high degree of technology choice freedom to design read and write. But don’t make it a **technology** **soup** unnecessarily. Else management will be very difficult down the line in long run.

This kind of architecture very fit for **Bounded Context** in **Domain Driven design**.Always let the business domain drive your software architecture not the technology.

### Challenges in event sourcing architecture

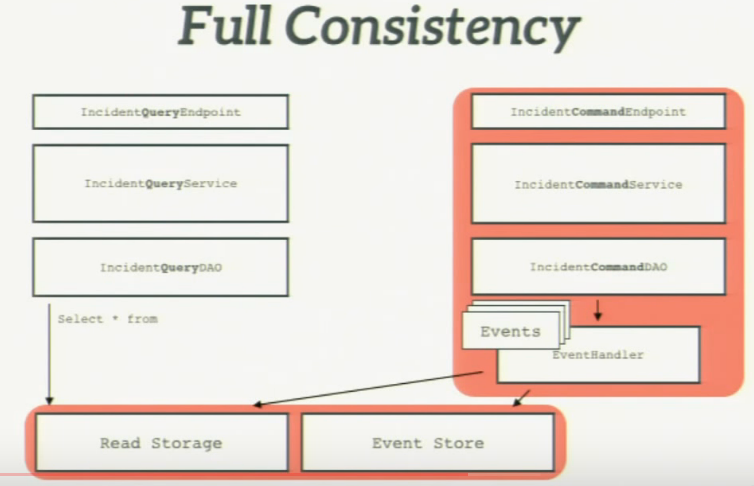
* Consistency
* Validation (on aggregation from multiple domain)
* Parallel updates, parallel processing of events

### Consistency

Based on **CAP** theorem, system either could be highly consistent or highly available at a particular point of time but not both.

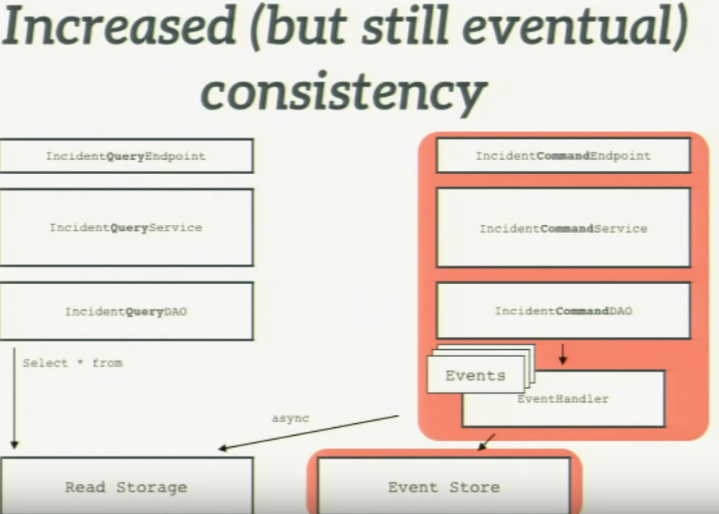
Below design could make the system highly consistent. If we make read and event/write store in a relational database in same node and can update to event store and read store under a single transactional boundary.

But this kind of system will not **scale.**



**Increased but eventual consistency.**

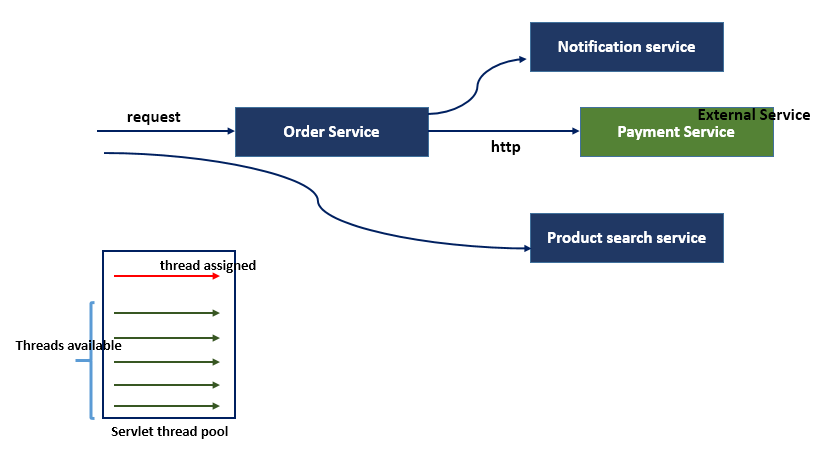
Here we can make the inserts into write store extremely transactional, But read store will be updated asynchronously. Which may contain stale data to read for certain amount of time



# Circuit Breaker pattern

Circuit breaker is a pattern for **fault tolerant** micro service communication.

## Evolving towards Circuit breaker pattern



When we calls to external service and the service is slow then the problem stats, how to handle it in my application.

### Cascading slowdown

On external service slowness, it may also impact with (make slow) other service communication in my micro-service architecture and the reason for it is **Servlet threadpool.** Because may be because of slow response from the external microservice, at a particular time all the threads of my servlet thread pool could be occupied and waiting for response and hence other service communication may not get the thread allocated and leads to slowness and the same behavior will propagated to other services making it slow too cause cascading slowdown.

Which in above case may slowdown **Notification service** and **Product search service**

### Implementing Timeout

Implementing **Timeout** is a proven and also reliable way to some extent to deal with this kind of scenario. Where we can configure a timeout interval (in time) and once that breached request will timeout and thread will be free.

The problem with timeout is, if the request receive rate is much higher than timeout configured rate, then eventually at a particular time all the server thread pool will be blocked and waiting for response or timeout to complete and again lead to the same problem i.e. slowness.

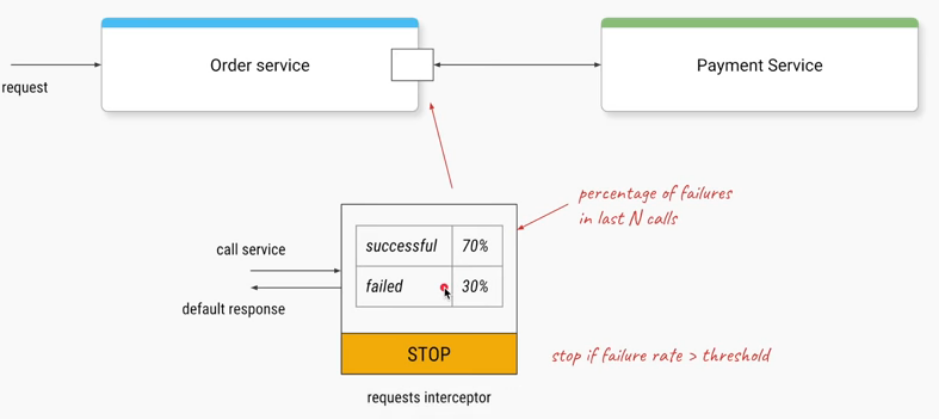
### Circuit breaker pattern

The intelligent way to deal with this kind of problem is, if a communication getting timeout or slow response for maximum number of request or maximum percentage of request. In that case it’s better to break that circuit for certain period of time then again try connecting after certain time.

When the circuit is open/break. During that time we can send default response or slightly intelligent way we can cache the previous equivalent response and add to response.

Moreover with circuit breaker we can configure threadpool capacity to multiple services so at a particular time there always be some thread available to use the request to that service.

Here we will add an **interceptor** close to the service which will track the number of success and failed request and based on that will decide if to **STOP** or **ALLOW** request further to the service.



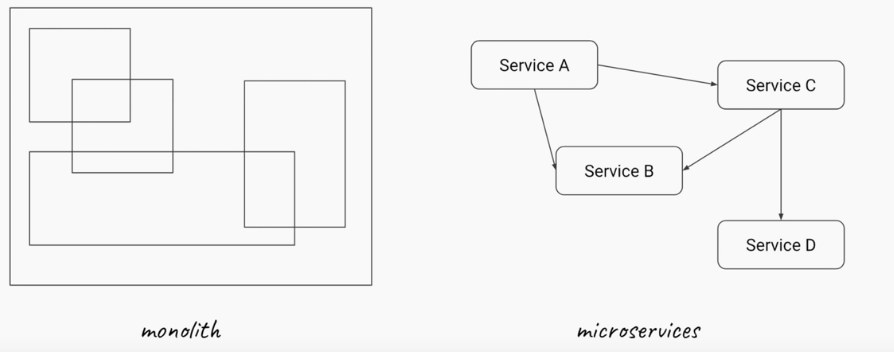
### Framework provides circuit breaker pattern

Netflix **Hystrix** which currently is not in development and in maintenance mode.

# Service Mesh Designing

Service mesh is a concept applies to microservices.

## Moving to microservice



## Benefits and challenges

**Benefits:**

* Faster delivery: Smaller size encourages frequent releases with minimal risk
* Isolation: Even if one service is down, entire application won’t be down. The other components will operate as usual.
* Scaling: We can scale independent services based on our need.
* Flexibility: Flexible to choose best technology or data store
* Culture: Well defined ownership as each service will be owned by a Pizza size team

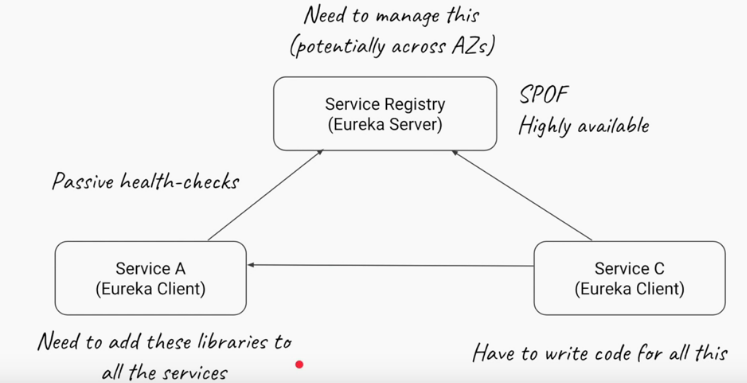
**Challenges:**

* Service discovery
* Load balancing
* Fault tolerant
* Distributed tracing
* Telemetric
* Security (mTLS, policies, patches)

### Service discovery-challenges:

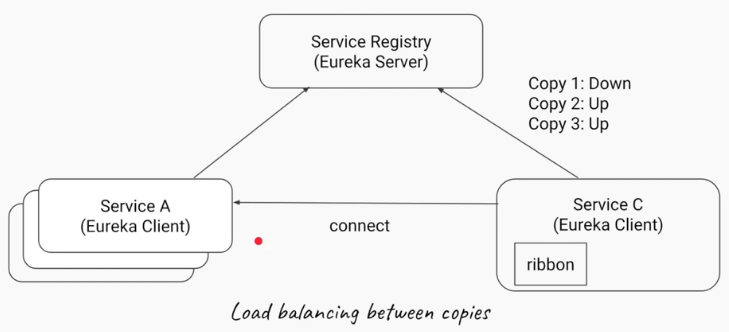
Each service copy based on deployment will have different IP address. So we need some kind of management like service registry (ex. spring cloud eureka)

Service registry is a single point of contact which leads to single point **failure.** So peer awareness and clustering concept available in Eureka server where we can deploy multiple copies of Eureka server.

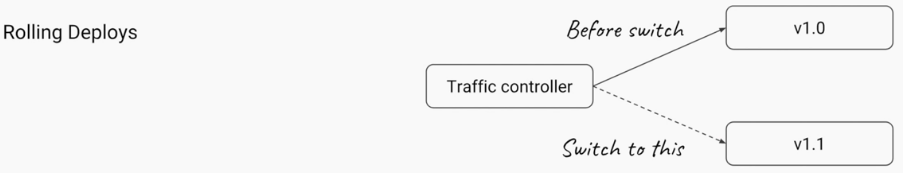


The other problem is we need to add library and code to each and every service we have to have this entire things to work.

If our service has scaled (means has multiple copies deployed) and all are running here we need to add library called **Ribbon,** it will get all the addresses of all the copies from **Service registry** and internally will decide how to do **Load balancing** between copies. First time it might connect to copy-1 next time copy-2 and so on.



### Canary / Rolling Deploys



If we have **v1.0** of the service and all the traffics are going to that. We want to start a version **v1.1** a new update to our service and on right time we want to switch all the traffic from v1.0 to v1.1. And while doing this I want to ensure service/business is not impacted and in the background service will be updated from previous to next version.

Same concept also applies to Canary deploys.



When we have a new feature and we want first to test it with a **small crowd** before deploying it into wider audience. Gradually we want to migrate more and more traffic to new version and eventually all the traffic once confirmed stable in prod.

### Distributed tracing

Now the flow of my request will go through multiple microservices. Then how we can debug in this case.

So when a request came in we need to assign a unique **trace-id** and **span-id** and **parent-span-id.**

**Spring Cloud Sleuth** does this for us.

### Security

* **Manual TLS between services:** How to ensure if two services are taking with each other they use **https.** So we need to install the certificate on all the service and also **certificate rotation** needs to be done in every 30 or 60 days.
* **Networking policies/whitelisting:** How to ensure set of rules so Service-A should only talk with Service-B and Service-C but not Service-D.

Having all these challenges how **Service Mesh** is intend to solve all these challenges.

## Service Mesh

The motto of service mesh is don’t burden my service with all these **infrastructure related decisions** or **communication related** but not on the business code the service is responsible for.

The idea is to take all these challenges and give it to some other component to deal with.

### Sidecars

Sidecars are the components which takes the heavy lifting from service code and instead Service-A communicate with Service-B, the **sidecar** will communicate with each other.



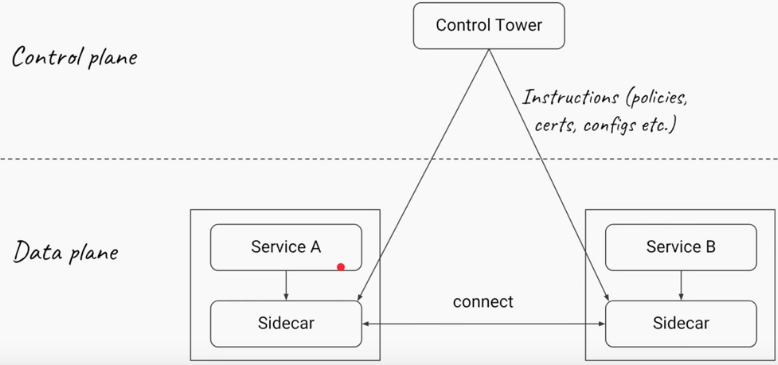
**sidecar** is an application/program will run at same place/container where microservice is running.

Here the sidecar will act as **proxy** for service. We will offload all the infrastructure and other challenges we talked from service to sidecar.

### Data plane vs Control plane

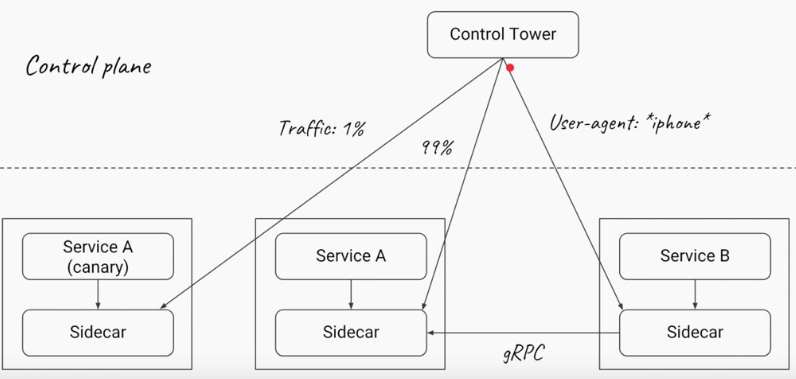
We also need some program or **control tower** which manages the **sidecar.** Which means we don’t want to put code into sidecar it supposed to manage by control tower.

This control tower will be responsible for pushing **service discovery, network policies, traffic management, load balancing and certificate for mutual TLS authentication** so on and so forth.



Now as we have **control tower** we can configure traffic management for Canary and Rolling deployment.

We can check or add headers and could manage the traffic.



Since services are not talking to each other directly so now **sidecar** can decide which communication protocol to use. It can move from HTTP1 to HTTP2 or GRPC.

The major benefit when we have a huge number of microservices and we don’t have to write a single line of code to manage all these as all will be managed from **control tower.**

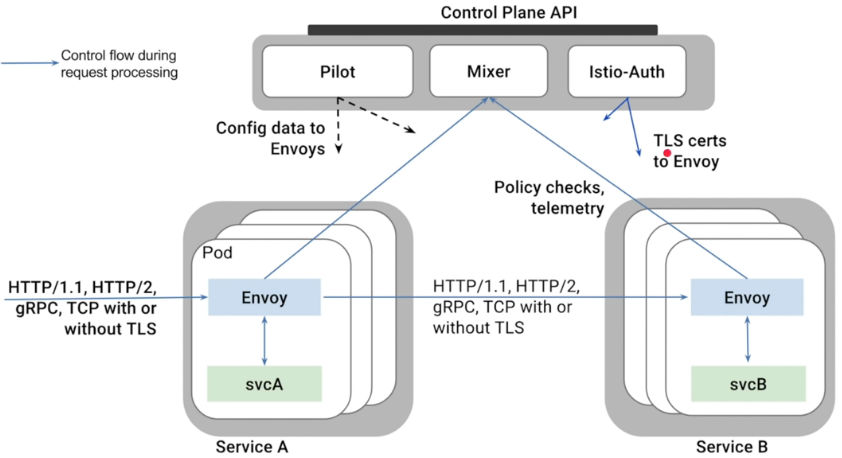
And this whole concept of **sidecar** and **control tower** is called **service mesh.**

So **service mesh** allows us to offload all the heavy burden relates to infrastructure, communication, network related challenges from our code to **control plane** and driven by **sidecar.**

## Istio + Envoy

One of the project which implements this service mesh is **Istio + Envoy.**

**Envoy** acts as a **sidecar** where **Istio** as **control plane**



## Responsibilities of service mesh

* Service discovery
* Load balancing
* Multi-protocol support (HTTP2 / gRPC)
* Fault tolerant (Circuit breaking, Rate limiting, Auto retries)
* Scaling
* Telemetric (including wire level like MongoDB, DynamoDB)
* Distributed tracing
* Security (mTLS, policies)

# API Gateway – STATELESS Authorization

## Spring cloud gateway for stateless authorization

### Authentication in monolithic application

* Security module authenticates the identity of the user.
* Session is created for the user and session id returned to the client.
* Client records the session id and sends it to the application in subsequent request (so no need to authenticate each time)

This is a state-full way of authentication. Where application remembers who the user is.

### What happens in microservice architecture

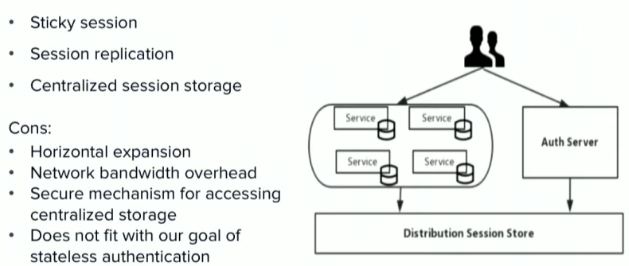
In microservice application we have many and many services and we may need to secure many services from client, external application and even from service to service communication.

So here each of the microservice needs to implement security from scratch for all the different clients calling it. Which does not make sense.

### Different ways to solve the problem

* Distributed session management
* Sidecar proxy
* Shared library
* API gateway

### Distributed session management



Here we need to put a load balancer in front and needs to mark the **sticky** flag to the session. So every time the request will go to the same service node were the session might have cached or put in related DB.

The problem will occur when that particular node is down and once next request will go to a different service node, it will be completely new for it.

We can manage this by putting a **Distributed session store** to which each and every service will connect and will get the session information.

Again here the problem will come on scaling which we need to do and manage which is increase the infrastructure complexity and communication where each service needs to make.

All this solution is a **Stateful** paradigm where we needs to move towards **Stateless** architecture.

### Sidecar proxy

**Sidecar** happens to be another service running in the same container or VM where is our microservice and this handles some of the cross cutting things which we don’t want to deal with every time.

* Adds more complexity to application.
* Every time we deploy microservice a sidecar needs to go with it.
* Management of all sidecars/data plane and control tower/control plane is hectic.

### Shared library

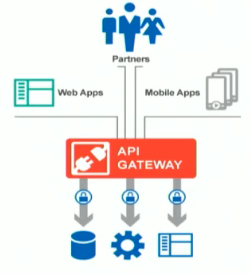
* Authentication and authorization rules are at certain level of isolation.
* No single point failure in the architecture.

**Cons:**

* If we write that application in a particular language our other application or components should also understand that language or run in the same JVM.
* If we have v1.1 of the library we need to worry about how do we push this to all of our application.

The best solutions from here comes out to be **API gateway.**

## Solving with API Gateway



* Runs in front of the microservices.
* Security is a global at this layer.
* Microservices controls there authorization of their content based on the permissions that are passed to.
* Facilitates microservices to focus on their business logic.

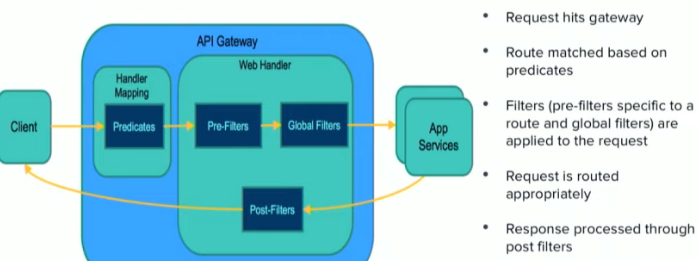
### Why API Gateway

API Gateway will be the common layer of cross cutting concerns like:

* Routing
* Security
* Rate limiting
* Circuit breakers
* Metrics
* Analytics
* Service discovery
* Monolithic staging
* Resiliency
* Canary and rolling deployment

One of the best option is **Spring cloud gateway.**

### Spring cloud gateway flow



**Predicates** are the set of rules to be applied.

We can add/remove header from request in **pre-filter.**

Authentication/authorization goes in **Global filters.**

In **Post filter** before response being sent we can manipulate like if we want to strip of any header.

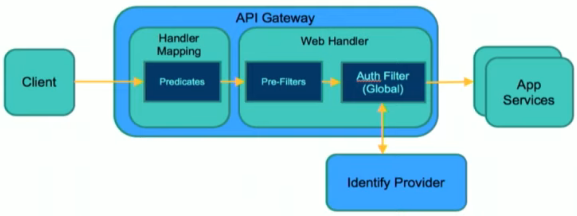
## Secure the gateway and beyond

### Security – spring cloud gateway + Identity provider

**Identity provider** will act as an authorization server will be deployed as a separate component and will be key to application to provide authorization.

Spring cloud gateway works in conjunction with IDP for security handling.

* Securing the application
* User identity propagation
* Stateless authorization



Identity propagation will solve the **STATEFUL** authorization needs and makes the authorization process completely **STATELESS.**

