**What are Microservices?**

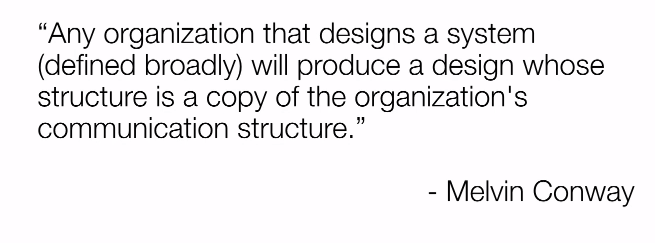
1. Small autonomous (change independently) services that work well together.
2. Functionally Decomposed

**Key characteristics**

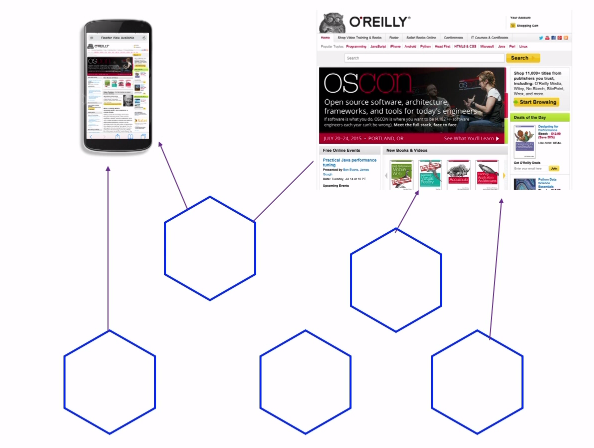
1. Independent processes
2. Communicate over APIs, Events or Messages
3. High degree of autonomy
4. Small, focused on doing one thing well.

**Advantages of Microservices**

1. **Better alignment with the organisation**: - With the large code base the teams fighting over the code base who gets to check in now? Who is in charge of this part of the code? This will be even worse if the teams are in different geographical locations. With Microservices architecture you can assign ownership of different services to independent teams.



1. **Ship feature faster and safer**: - In a Monolithic application, even for a one-line change all of the tests needs to be executed. It is a one big deployment. So deploying everything has a high degree of risk although it is a one-line change. This leads to less frequent changes going out and as a result of this there will be more back logs.
2. **Independent Scalability: -** Cost effectively scale out the Microservices as they are functionally decomposed.
3. **Target Security Concern: -** In a Monolithic system there is one security whereas for Microservices each services can have its own securities and securities can be made lighter or tighter eg for a Microservice which is taking care of payments could be enforced with expensive security whereas for Microservices performing some trivial operation could have less expensive security.
4. **Enable segregation model: -** I.e segregate a set of Mciroservices and place it in a separate network in order to give more protection etc
5. **Adopt technologies more easily: -** I.e each of the Microservices can be built in independent languages with a different technology stack.
6. **Embrace uncertainty in digital: -** We have Mobile, Ipad, apple watch, windows, tablet etc. If we have one Monolithic system, we have to alter it to work for all of these. Tomorrow a new device may come and we will have to change our Monolithic system to handle the new device. Whereas if we have Microservices we can mix and match the capabilities of each Microservices.



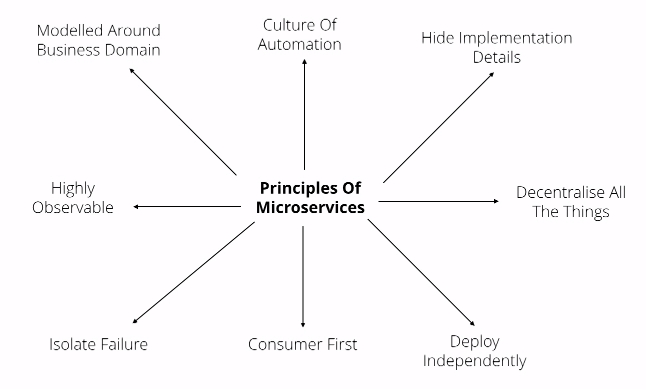
**Disadvantages of Microservices**

1. So many options
2. Take time to get there
3. Testing is more complex
4. Monitoring is more complex
5. Resilience is not free
6. Lots of boxes to manage
7. Distributed systems are hard

**Principles of Microservices**

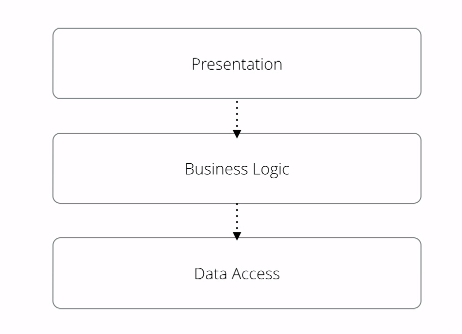
**Principle**: - Idea. This change less frequently

**Practices**: - Ways in which the idea can be implemented. This change more frequently



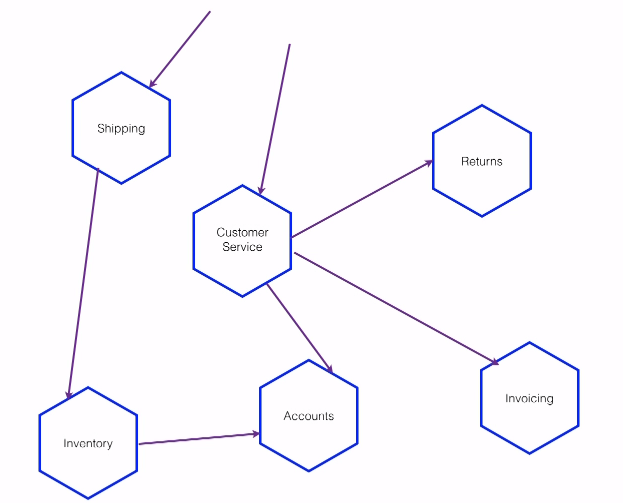
**Modelled around business domain**

The monolithic application usually will be designed as follows. As we moved to SOA each layer was sliced as an API.



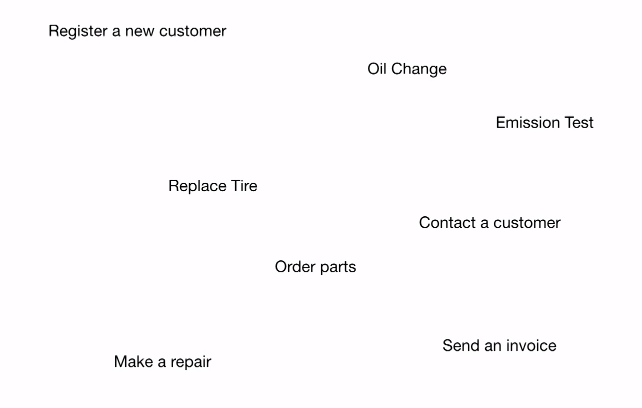
Each layer being owned by particular team. In this case a small change needs to be co-ordinated with different teams and the respective teams had to make changes and deploy. By slicing in the above manner team members were experts of UI, server side, DB etc

Instead of slicing horizontally if you slice vertically

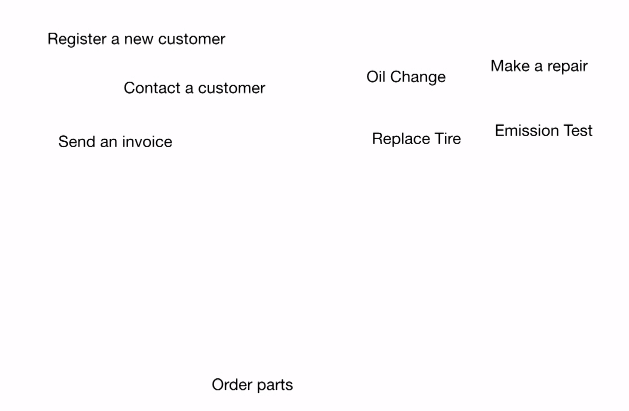


This leads to services having names that reflects functionality of our system. So any person technical, non-technical etc when looked at the above architecture will get an idea of what the system does. The team that owns the systems becomes expert on that part of the domain (functionality).

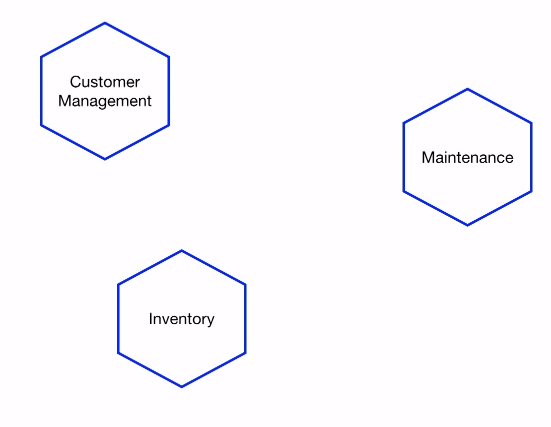
Design an application by listing down their **capabilities**



We can then group the capabilities



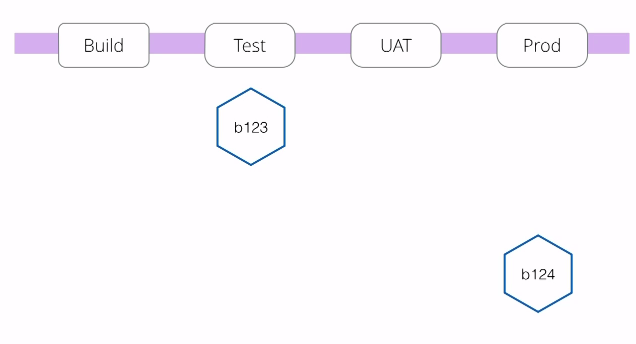
This leads to different Micrsoservices that does one thing.



**Culture of Automation**: -

1. Continuous delivery: -

Considering every check in as a release candidate (i.e when we commit a code we should be happy that it can go to production)



If we can automate the visibility whether or a not a software / change can go in to production embracing the idea of continuous delivery it is going to be easy to press the button and send the software to production

1. API Driven Machine Provisioning: -

Writing few lines of code to spin up a machine helps in spinning up production like machines in no time. Eg VM ware, AWS, DigitalOcean, Vagrant etc

1. API Driven OS Configuration: -

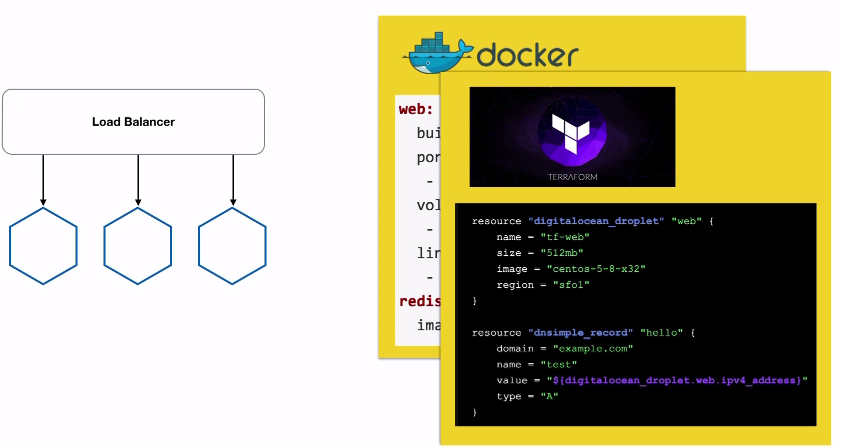
Once we have a machine from above, we need to configure it. Example tools Chef, Ansible, Puppet etc. This tells how a machine / OS should look like example this version of Apache, this version of Java etc.

1. Custom Image Creation: - #2 + #3. Tools like Packer can be used.
2. Platforms: -

Consider PAAS, IAAS and CAAS (is becoming populate i.e Container as service eg Core OS, Google’s Kubernetes etc )

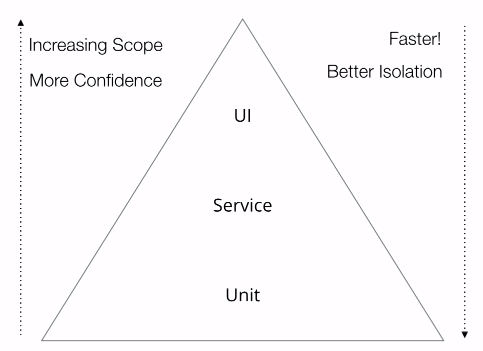
1. Declarative Environment Provisioning: -

One service might be deployed in different data centres and load balanced whereas for a developer Laptop it is just in one place i.e one copy. So encapsulating these differences and still making to provision these in automated way is challenging. Declarative environment provisioning is one such approach where it takes #1 and #2 or #3 and applies it to a group. Tools example Docker, Terraform



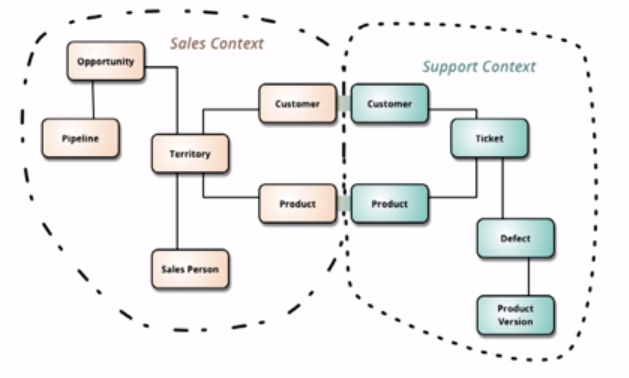
Here Terraform for example targets many different platforms i.e one config that deploys the app in docker and another config that deploys the same app in different way in AWS.

1. Automated testing: -



As you see integration test in the left hand side gives more confidence but they are not easy to be executed as we will have several challenges eg network failure, browser crashing for no reason etc whereas executing the unit tests is easy and faster. So with plenty of microservices we will see more of integration testing but we should avoid these end to end test and create more unit tests (I think by way of mocking)

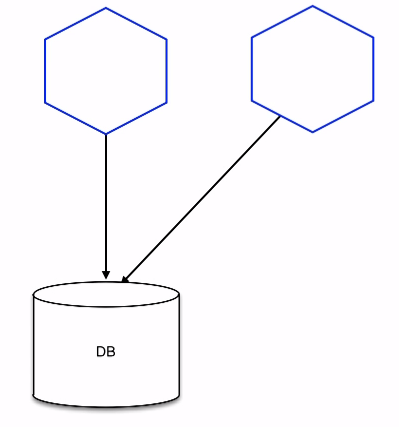
1. Hide Implementation Details
2. Bounded Context



Classify what should be exposed and what should be hidden. Example in the above bounded context / domain driven design customer and product are exposed whereas the others are not.

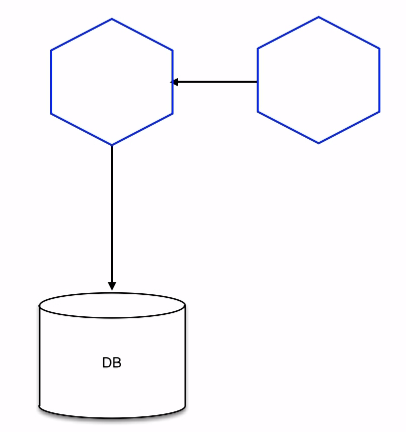
1. Hide your DB

Don’t do like below



As we cannot hide details because the other services can manipulate data. Moreover, we will not know the list of apps using the DB. The above model is not cohesive.

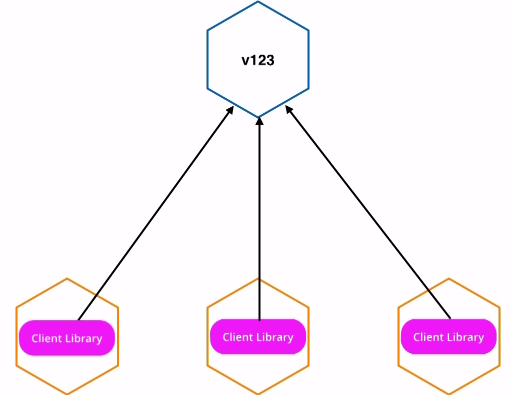
Instead of above do something like an API call

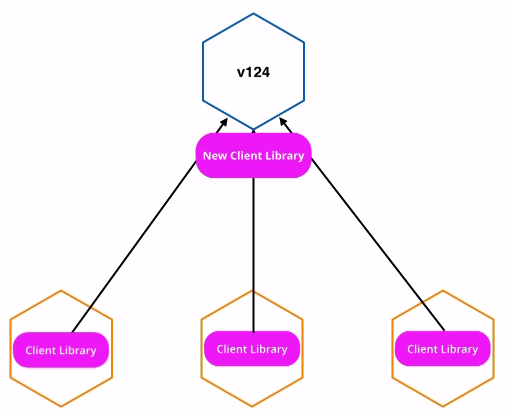


1. Think about the protocols

Eg RMI demands both server and client to be JVM based.

1. Be careful of client libraries





Client libraries are used to reduce the duplication of code. New client library is required to accommodate the changes that means the services using the old client libraries cannot use the service anymore.

1. Decentralize all the things
2. Self-service: -

Can I do what I want without co-ordinating like raise a ticket to get it done etc.

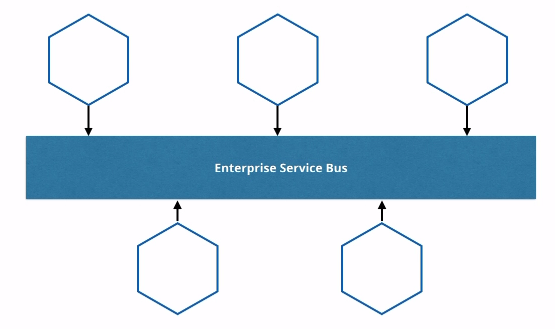
1. Owner Operator: -

You build it, you run it, you support it and you decommission it if required i.e no separate maintenance team, bug fix team, deployment team etc

1. Internal Open source: -

Instead of co-ordinating with the owner team to get them fix / enhance their app you can check out their code do changes and add a pull request. Obviously there should be gatekeepers of the app to vet the code changes etc and to decide on taking in the change to production.

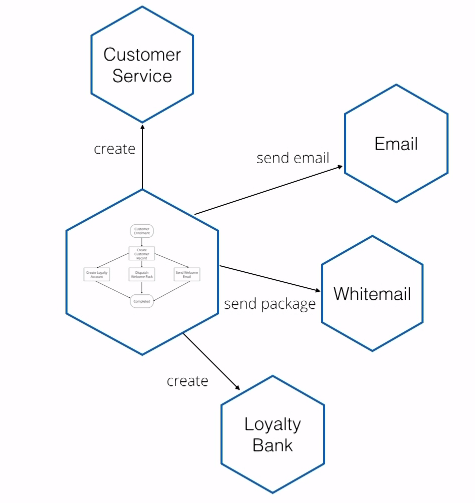
1. Beware Smart Middleware: -



Make your middleware (message bus) dump i.e it should just take a message from A and give it to B. It should not hold any business logic. If it has business logic, then you have to rely on the middleware team to get that part of the change done in order for your change to work.

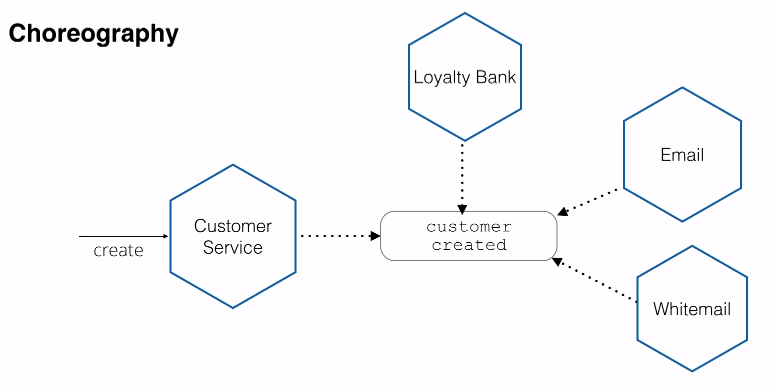
1. Orchestration vs Choreography

Orchestration

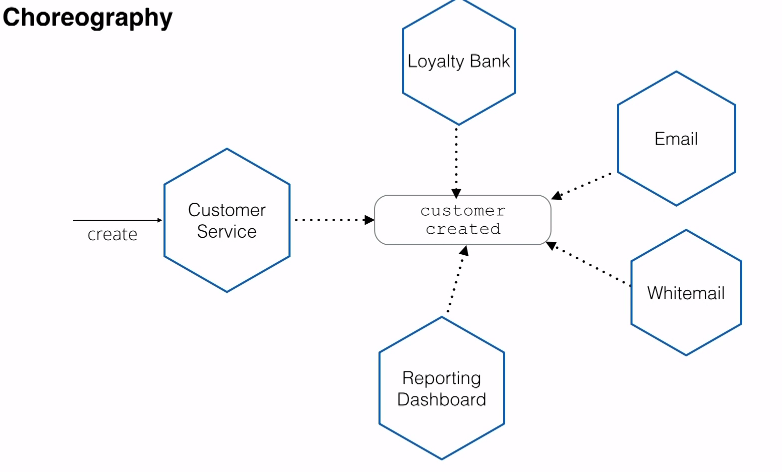


Here the central authority instructs other app

Choreography



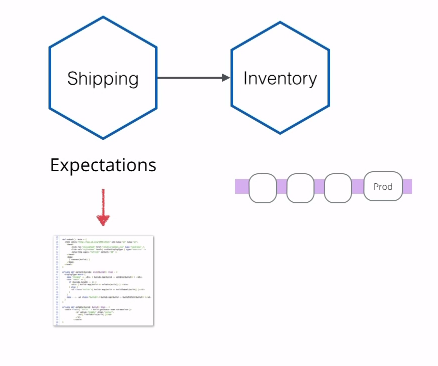
The above is event driven i.e as soon as the Customer service triggers a customer created event then all the other app listening to that event performs an operation. This gives a high degree of loose coupling I.e we can add new actors very easily like a reporting dashboard which will listen to the customer created event. None of the actors / app know anything about the other apps.



Downside of Choreography is we need to monitor and make sure the necessary sequence of events is triggered and processed (may be using splunk) example in a broadband company new customers created would not have received the broadband service or anybody who received broadband service might not have paid for so long.

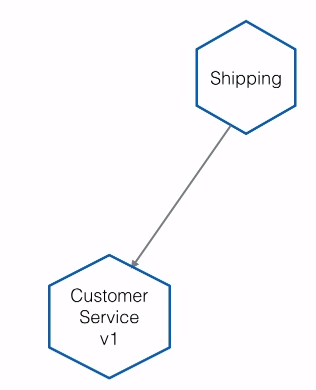
5. Deploy Independently: -

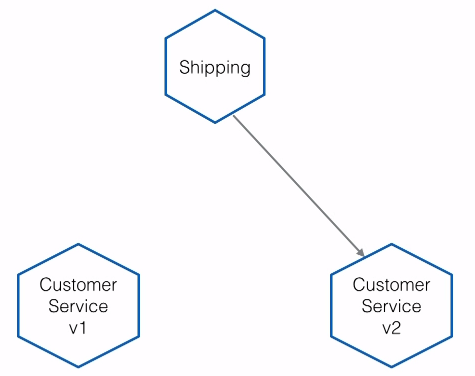
1. One Service per OS: - Docker uses the same OS but puts in in different container.
2. Consumer driven contracts: -



In the above the Inventory service is changed. In order to test the consumers, the consumers have an expectation tests that needs to be executed. If they all pass, then the inventory service can be moved to Prod. If they fail, then we have to co-ordinate with the Consumer service. We can consider the below two approach instead of this approach.

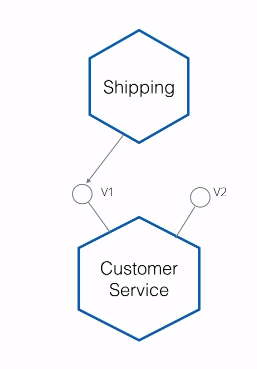
1. Co-existing service versions: -

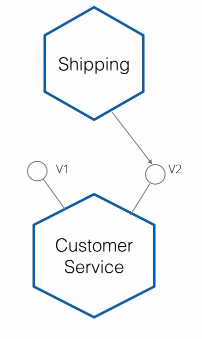


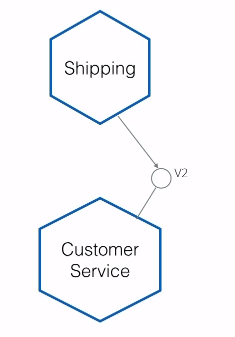


It is a dev / maintenance challenge if there are more than one version of service. The services somehow should know which version it should to talk to (i.e service discovery – eg DNS, Zookeeper, etcd, CONSUL).

1. Multiple End points: -







Again here the service needs to discover which version (i.e new end point) it has to talk to.

6. **Consumer First**

1. Conversation: -

Talk to the consumers regularly, may be have a meeting, forum or a DL

1. Consumer driven contracts: - Same as above but here it is highlighted that we will come to know what the consumer is expecting.
2. Actionable Standards: - Agree standards on exception handling etc and have it documented and published.
3. API Documentation: - Swagger can be used. You can think of it like a Java Doc for the APIs’.
4. API Gateways: - Refer <http://microservices.io/patterns/apigateway.html>

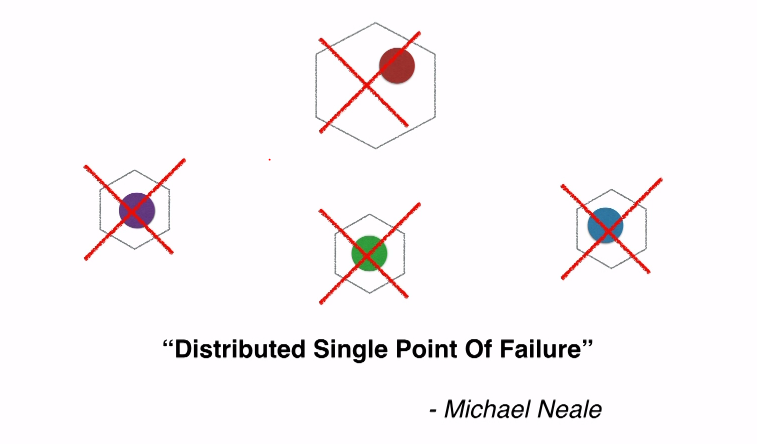
Implement an API gateway that is the single entry point for all clients. The API gateway handles requests in one of two ways. Some requests are simply proxied/routed to the appropriate service. It handles other requests by fanning out to multiple services. Rather than provide a one-size-fits-all style API, the API gateway can expose a different API for each client (mobile, web app etc). It can also be used to discover the service.

API gateways provides many but two important info

1. Developer portal that tells the available APIs’
2. API key management: - i.e with the help of the API key the caller can be identified there by knowing the consumers.
3. Service discovery: - Example DNS, Zookeeper, etcd, CONSUL etc.
4. Humane Registry: - A simple wiki page (or nice UI) containing basic info of the service, what it does, how to use it etc for the programmers.

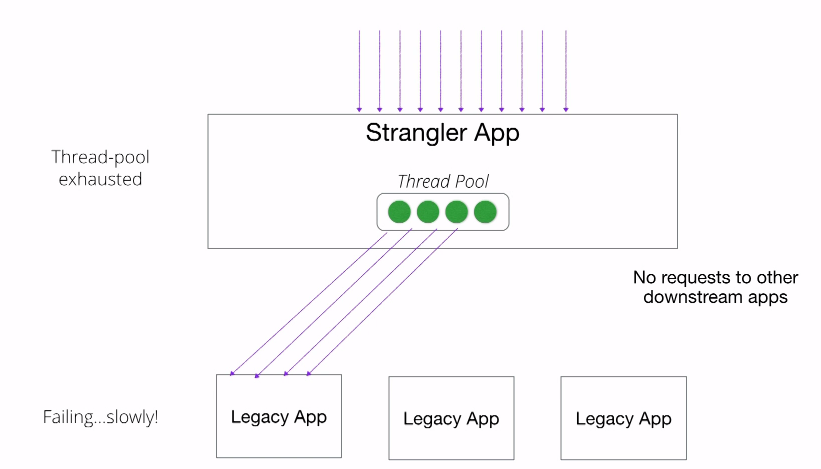
7. Isolating failure: -

1. Microservices are not reliable by default: - i.e we should avoid single point of failure. The following should be avoided.



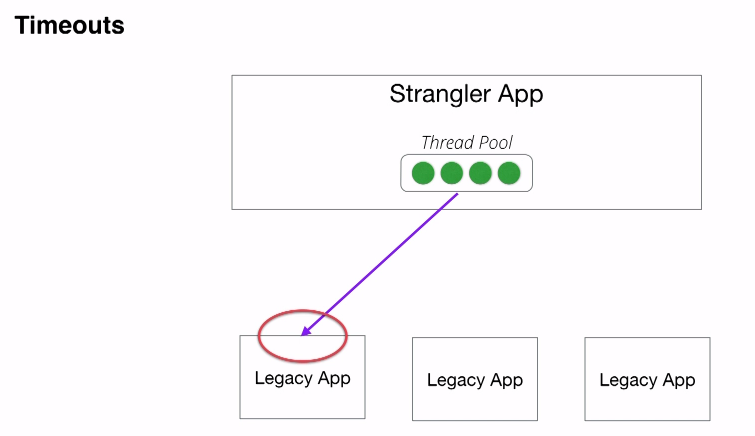
1. Cascading failure can hurt.

So consider the below the downstream app was failing slowly. The threads / requests to that app were waiting for the app to complete the process. As a result of that the threads exhausted and new requests were waiting and no other requests were routed to other apps.

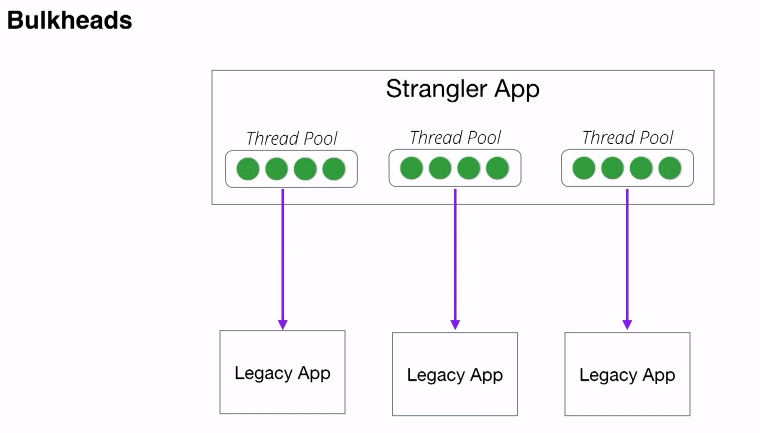


In order to fix the following measures were taken

1. Timeout: - Decreased the timeouts



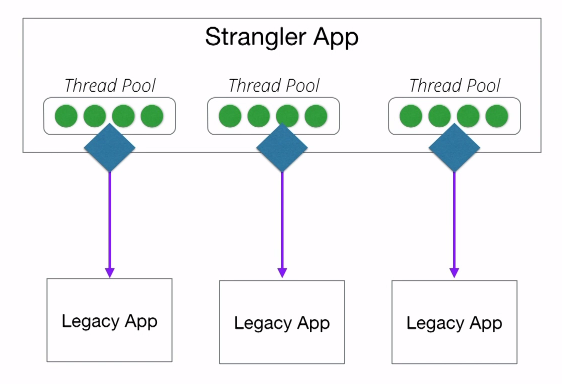
1. Bulkheads



More thread pools were put that means even if one is exhausted the rest of the other pools could operate.

Definition of bulkhead - is a dividing wall or barrier between separate compartments inside a ship, aircraft, or other vehicle. So if one part of the shit is hit the doors for that part can be closed and let it go away from the entire part of the ship

1. Circuit Breakers (Hystrix can be used for Circuit Breaker implementation)

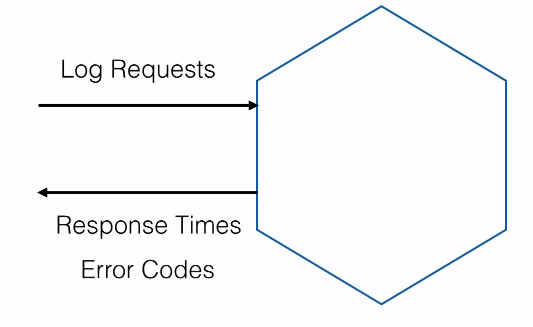


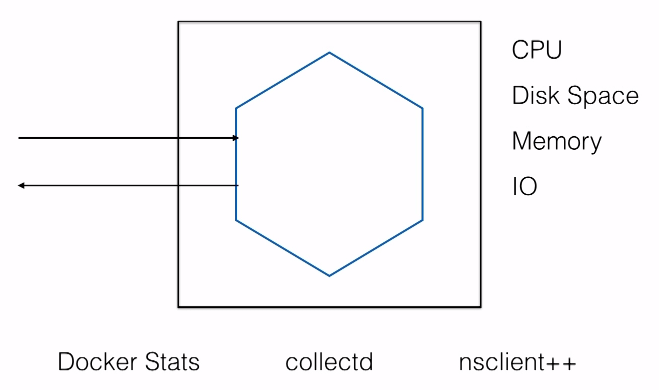


If a downstream errors (may be 500 errors in a short period of time or timeouts in a short period of time) then it is considered unhealthy and the circuit blows so the downstream app instead of failing slowly fails fast also that piece of functionality is brought down in the UI by displaying an error message.

8.Highly Observable

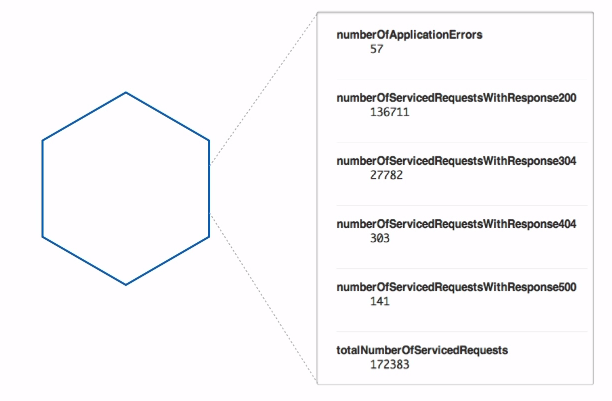
1. Standard Monitoring: - Across all the microservices app





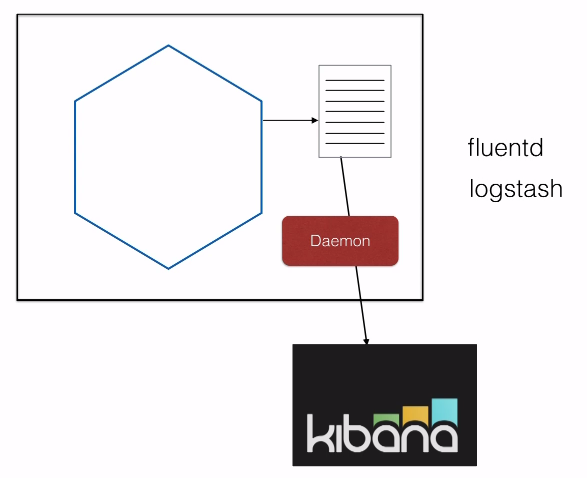
Docker stats gives the usage details about CPU, Disk space etc. In linix collectd can be used and in windows nsclient++ can be used to get the same info.

1. Health Check pages

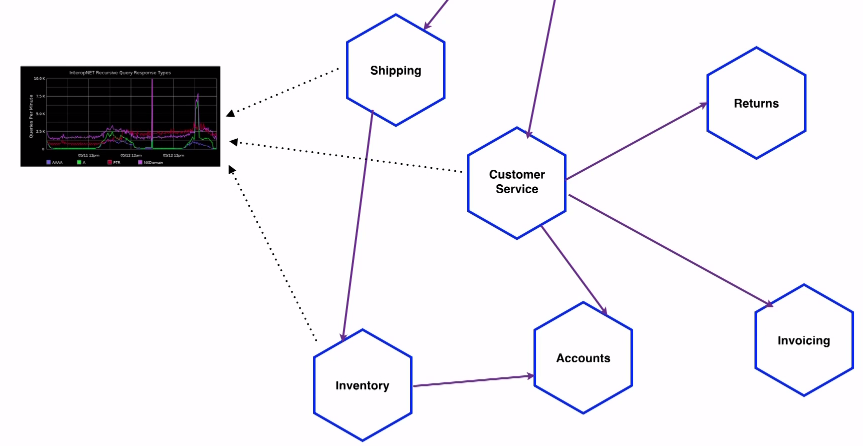


1. Logging Aggregation

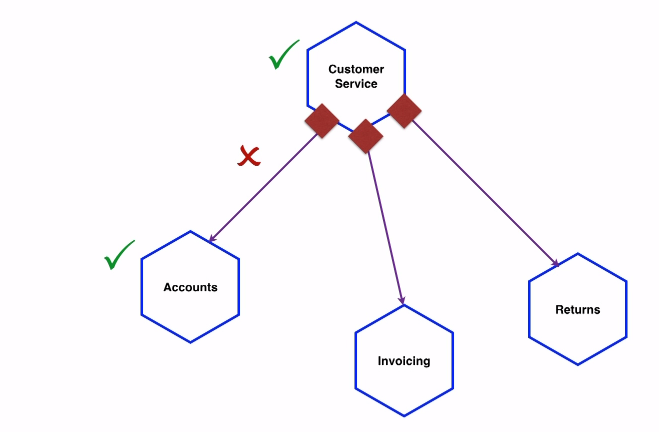
Logging Daemon (eg fluentd, logstash) can run in each node which can take the required logging details and print somewhere centrally so that we can monitor all of the services in one place.



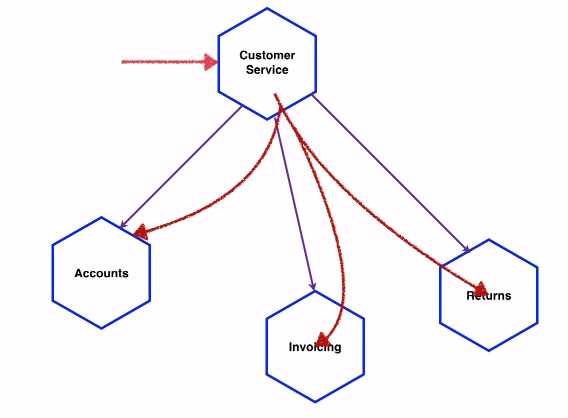
1. Stats Aggregation

CPC, IO, Memory etc usage of all the systems (aggregate) should be logged in one place. Famous tool used is Graphite. 

1. Business aggregated data can be logged if required.
2. Downstream Monitoring: - If the downstream is up and if the upstream says it is also up but if there are no communication happening then it might a network issue. So monitoring connections play a vital role.

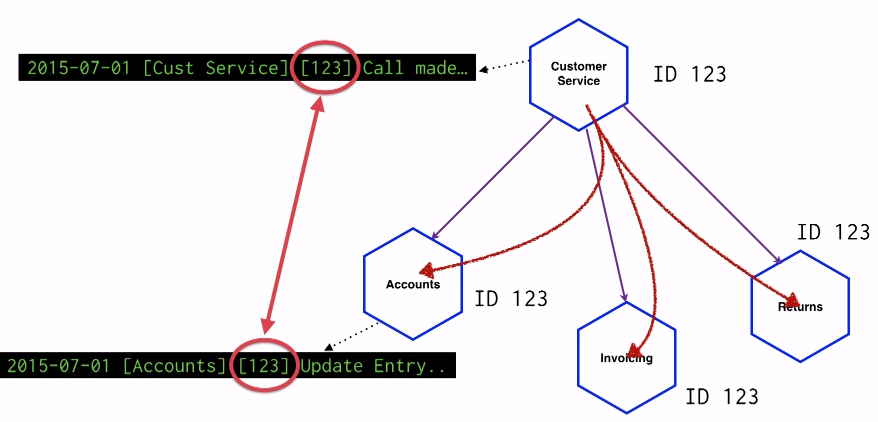


1. Semantic Monitoring



If a system is down or if more than one system is down should the on-call person be contacted? The answer to this is try a happy path testing in Prod and if it does not fail then the system is in the correct state. So nobody needs to be called. Make sure the system knows this is a test.

1. Correlation Ids: - This is useful in troubleshooting. It is unique per transaction / call eg trade id



1. Dashboard: - Now that we have logged so much information, many different users want to see the data in different view. Example tool is Dashing