Microservices Overview

Introduction to Microservices

Lesson Objectives

At the end of this module you will be able to:

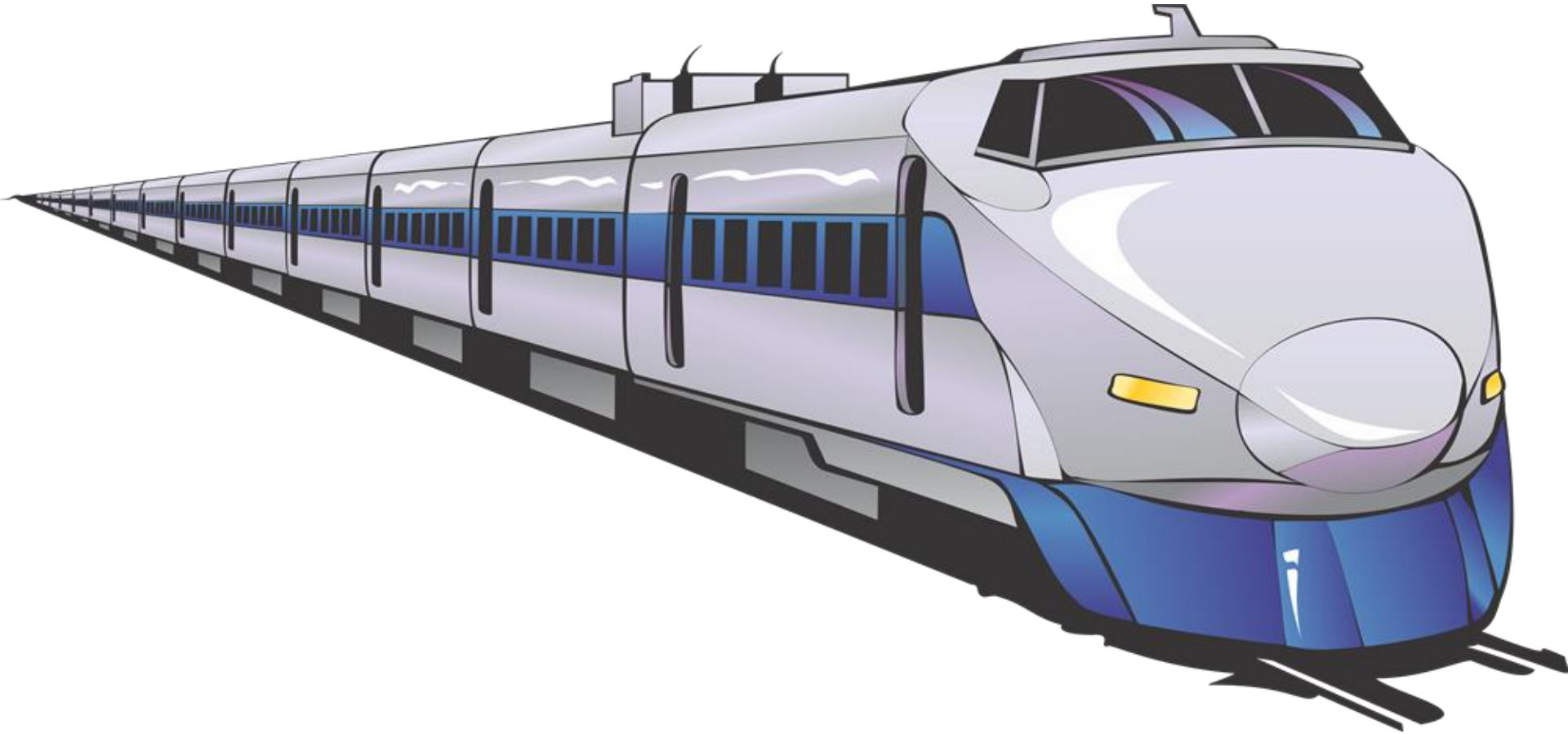
- Understand limitations of Monolithic Systems
- Understand the need and importance of Microservices
- Design principles need to be followed in Microservices
- 12 factor application

Service Introduction

A Service is an application component which provides functionality to other components (web app, mobile app, desktop app, or even another service)

Service-Oriented Architecture is an architectural pattern in which components provide services to other components via a communications protocol over a network.

Monolithic System



Monolithic System

Monolithic system is a typical enterprise application.

There's always one package which basically contains everything which tends to end up with large code base.

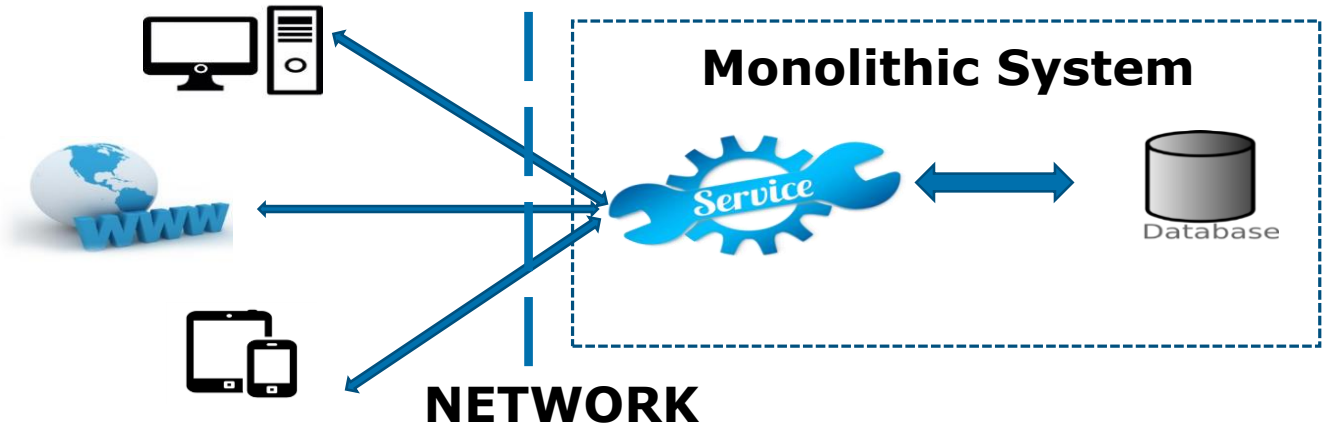
When application expands, It keeps grows with no physical division and no restriction in size.

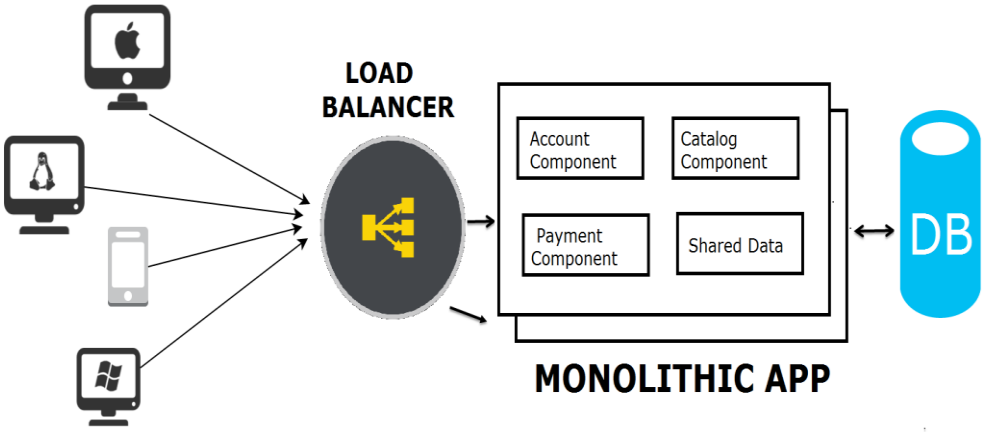
Developers takes longer time to develop new functionality with in application and it's difficult to make a change without affecting other parts of the system.

Deployment of a large system can also be challenging, because even for a small bug fix, a new version of the entire system need to be redeployed.

New technologies cannot be adopted for a specific functionality, since it is available in a overall package.

Monolithic Architecture





Monolithic System Advantages

Single codebase

- Easy to develop/debug/deploy
- Good IDE support

Easy to scale horizontally (but can only scale in an “un-differentiated” manner)

A Central Ops team can efficiently handle

Monolithic System Limitations

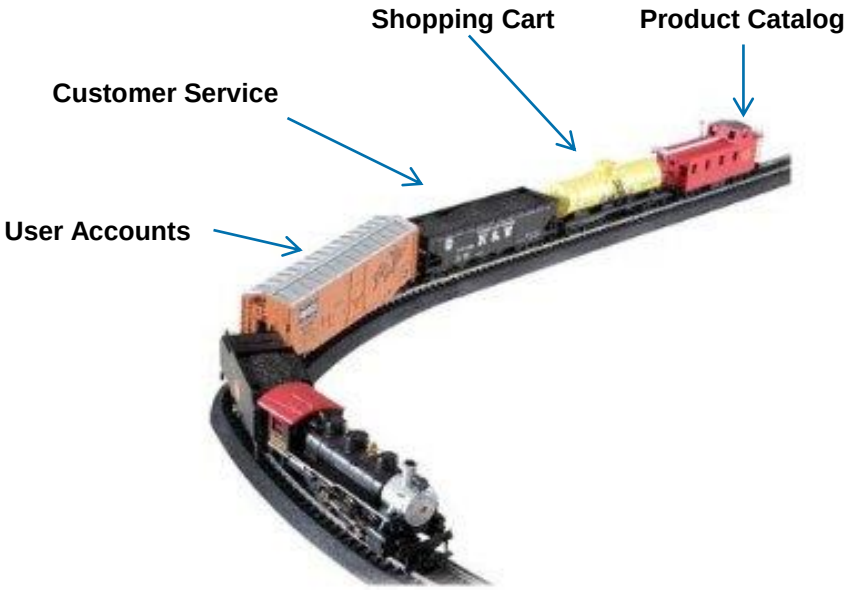
As codebase increases …

- Tends to increase "tight coupling" between components (system stops working, if one component stop working)
- Just like the cars of a train

All components have to be coded in the same language

Scaling is "undifferentiated"

- Cant scale "Product Catalog" differently from "Customer Service"



Microservices Introduction

Microservices are small software components that are specialized in one task and work together to achieve a higher-level task.

A microservice is an independent unit of work that can execute one task without interfering with other parts of the system.

Micro-service normally has a single focus. i.e. It does one thing and it does it well.

It is basically an improved version of service-oriented architecture.

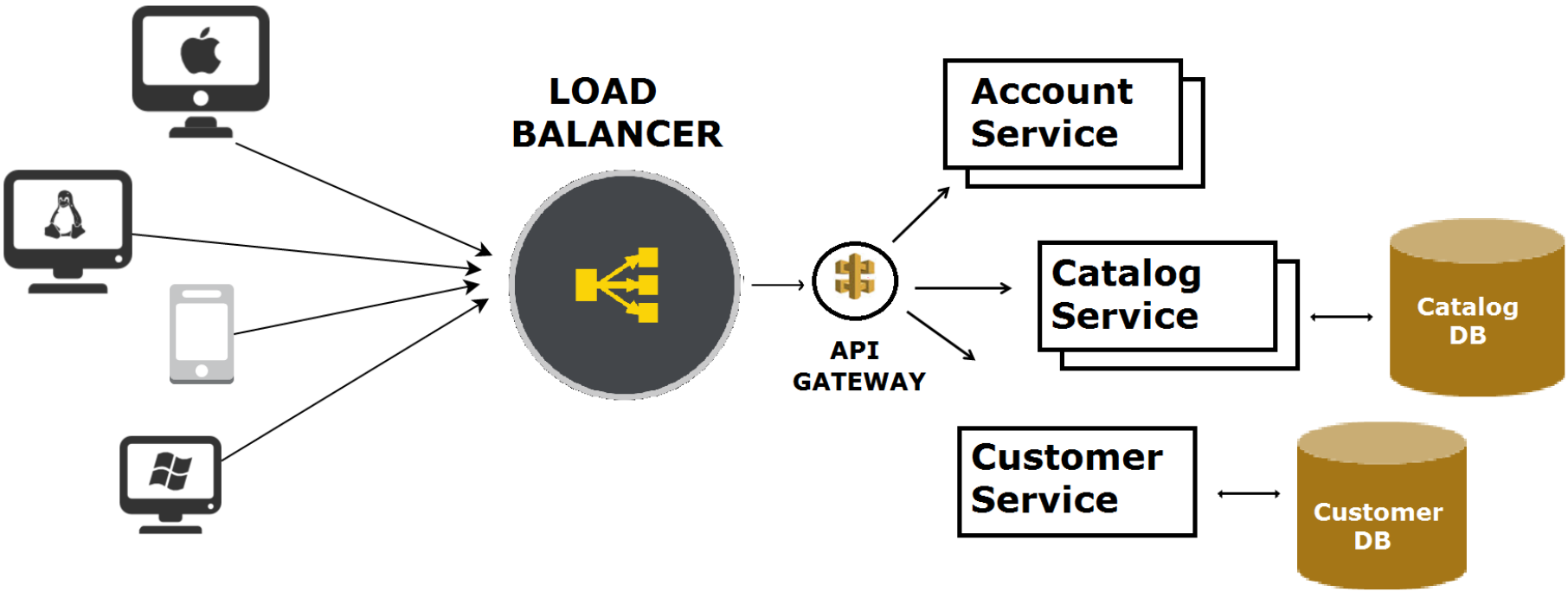
Microservices are business units that model the company processes.

They are smart endpoints that contain the business logic and communicate using simple channels and protocols

Microservices Advantages

- Individual parts can be enhanced based on market needs.
- Microservices can be tested using automated testing tools
- Deployment is super easy with the release and deployment tools.
- Virtual machines can host microservices on-demand
- Embraces new technology
- Supports Asynchronous Communication technology
- Simpler server side and client side technology
- Shorter development times
- Highly scalable and better performance
- Enables frequent updates and fast issue resolution

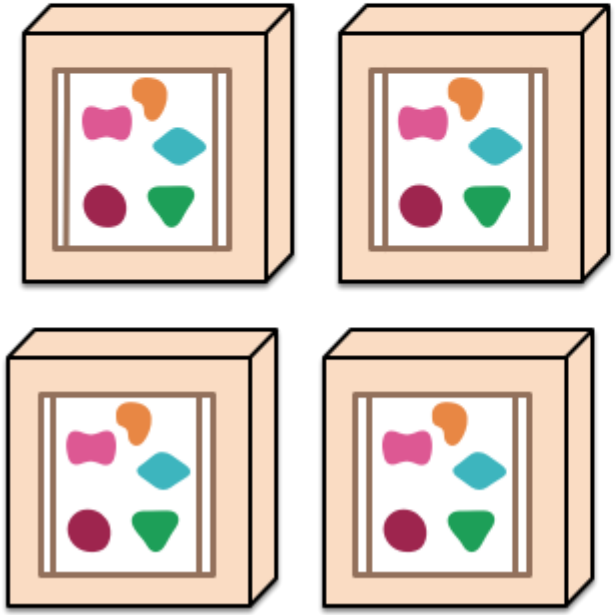
Microservice Architecture



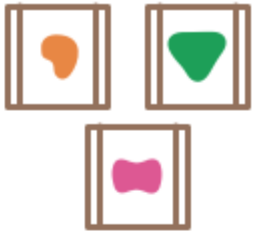
Monolithic vs Microservices

A monolithic application puts all its functionality into a single process...

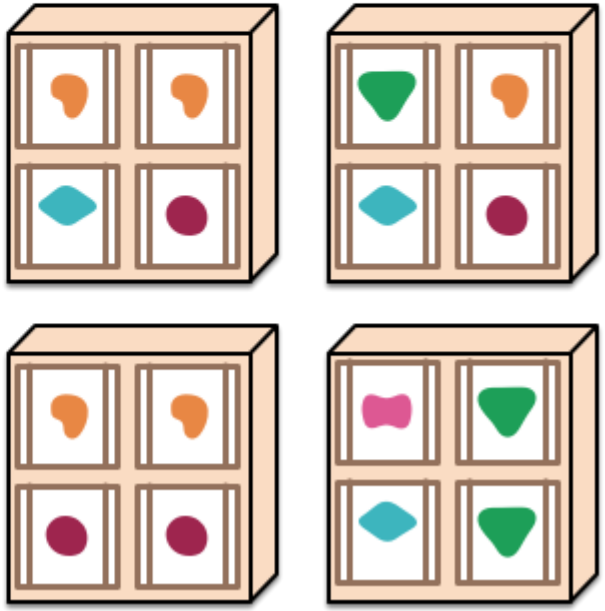
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.



Microservices Challenges

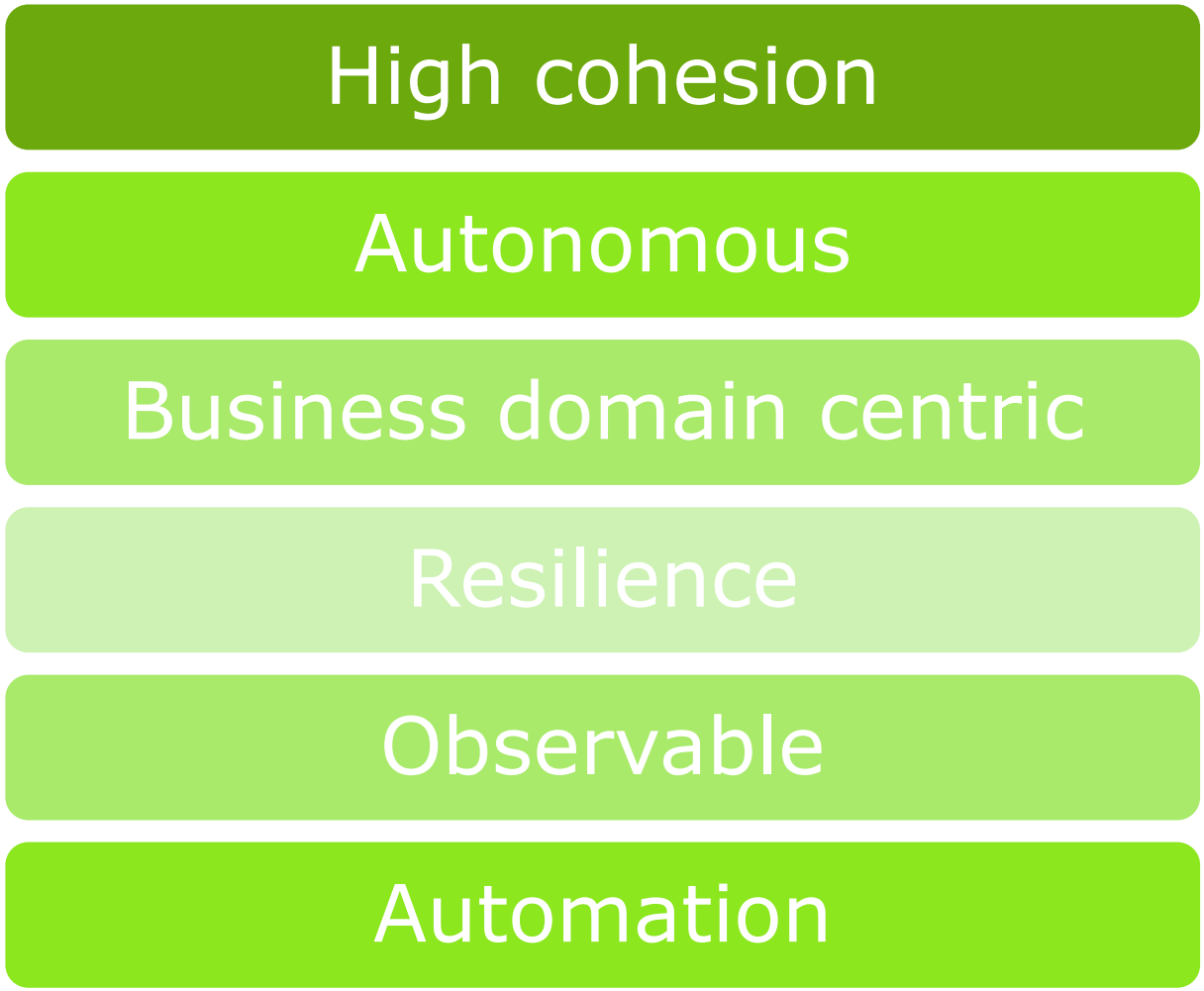
Can lead to chaos if not designed right

Distributed Systems are inherently Complex

Operational Overhead(DevOps Model required)

MicroServices does not automatically mean better Availability, unless Fault Tolerant Architecture is followed.

Microservices Key design principles



High Cohesion

High cohesion principle controls the size of the microservice and the scope of the contents of the microservice

- Microservices content and functionality in terms of input and output must be comprehensible.
- Must have a single focus.
- It makes overall system highly scalable, flexible, and reliable

Autonomous

Microservices should also be autonomous

- A microservice should not be subject to change because of an external system it interacts with or an external system that interacts with it. i.e. It should be loosely coupled.

Microservice should also be stateless

Microservice should support backwards compatibility

Microservice can be independently changeable and independently deployable

Contracts between the services must be clearly defined

- Concurrently developed by several teams

Business Domain Centric

A microservice should be a business domain centric

- The overall idea is to have a microservice represent a business function or a business domain
- This helps to scope the service and control the size of the service
- When business changes or the organization changes or the functions within the business change, microservices should change in the same way because the system is broken up into individual parts which are business domain centric

Resilience

Microservice need to embrace failure when it happens.

- Microservice needs to embrace the failure by degrading the functionality within it or by using default functionality.
- If one component of a system fails, the failure should not cascade, the problem should be isolated and the rest of the system should work.
- Microservices can be more resilient by having multiple instances which registers themselves on start up, and deregister themselves on failure to have an awareness of fully functioning microservices.
- Microservices should validate incoming data, and it shouldn't fall over because they've received something in an incorrect format.

Observable

System built with Microservice architecture must be observerable.

- System's health must be observed in terms of logs, system status and errors.
- Logging must be centralized and monitored from a centralized place. Which help us to achieve
  - Solving the problem quickly
  - Monitor business data
  - Data used for scaling and capacity planning.
  - Distributed transaction details

Automation

Microservice architecture should support the feature of automation with the help of automation tools.

- Automated testing will reduce the amount of time required for manual regression testing, and the time taken to test integration between services and clients, and also the time taken to set up test environments.
- Continuous Integration tools

Microservices Best Practices

Isolate your services (Loosely Coupled)

Use Client Side Smart LoadBalancers

Dependency Calls

- Guard your dependency calls
- Cache your dependency call results
- Consider Batching your dependency calls
- Increase throughput via Async/ReactiveX patterns

Test Services for Resiliency

- Latency/Error tests
- Dependency Service Unavailability
- Network Errors

Microservice and Docker

Docker is an excellent tool for managing and deploying microservices.

Each microservice can be further broken down into processes running in separate Docker containers, which can be specified with Dockerfiles and Docker Compose configuration files.

Combined with a provisioning tool such as Kubernetes, each microservice can then be easily deployed, scaled, and collaborated on by a developer team.

Specifying an environment in this way also makes it easy to link microservices together to form a larger application.

Twelve-Factor App methodology

The Twelve-Factor App methodology is a methodology for building software as a service applications. These best practices are designed to enable applications to be built with portability and resilience when deployed to the web.

Twelve-Factor App methodology

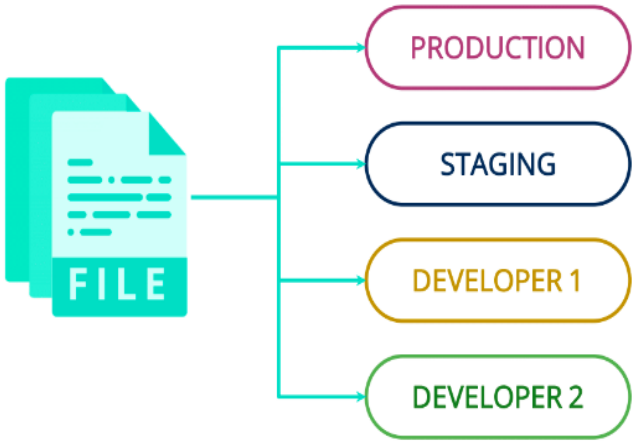
The Twelve Factors

#	Factor	Description
I	Codebase	There should be exactly one codebase for a deployed service with the codebase being used for many deployments.
II	Dependencies	All dependencies should be declared, with no implicit reliance on system tools or libraries.
III	Config	Configuration that varies between deployments should be stored in the environment.
IV	Backing services	All backing services are treated as attached resources and attached and detached by the execution environment.
V	Build, release, run	The delivery pipeline should strictly consist of build, release, run.
VI	Processes	Applications should be deployed as one or more stateless processes with persisted data stored on a backing service.
VII	Port binding	Self-contained services should make themselves available to other services by specified ports.
VIII	Concurrency	Concurrency is advocated by scaling individual processes.
IX	Disposability	Fast startup and shutdown are advocated for a more robust and resilient system.
X	Dev/Prod parity	All environments should be as similar as possible.
XI	Logs	Applications should produce logs as event streams and leave the execution environment to aggregate.
XII	Admin Processes	Any needed admin tasks should be kept in source control and packaged with the application.

Twelve-Factor App methodology

Codebase

- When building an application, we must use one *codebase* that ca be run in all *environments* .
- Some examples of using *codebases* are the use of *git* , *svn* or other *version controls* .



Twelve-Factor App methodology

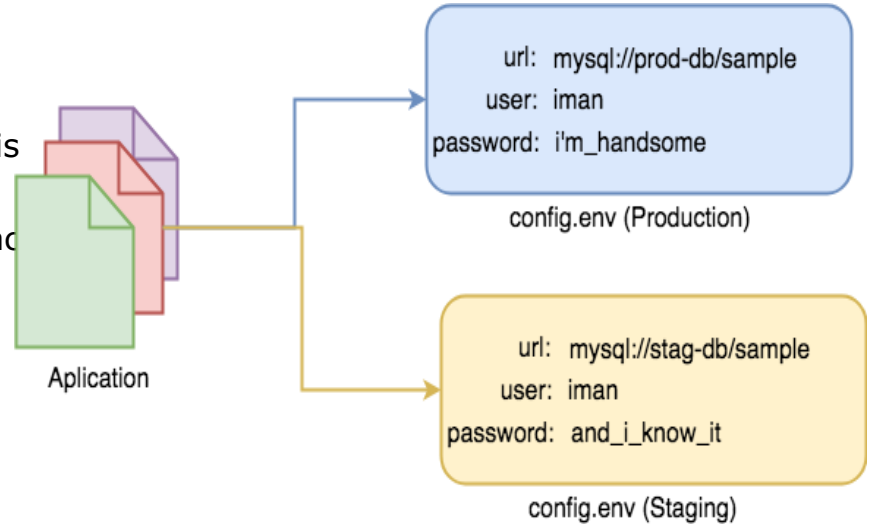
Dependencies

- Software good is software that is obvious dependencies management of his.
- So that when the process of deploying to any server we are not overwhelmed.
- For example, like packagee.json in NPMs in NodeJs or Gemfilein Ruby. So during the deployment process , we only have to run one command npm install and all dependencies will be automatically installed.

Twelve-Factor App methodology

Configuration

- How can the application *run* in different *environments* .
- Because during the *software development* process , all applications and *source code* used are the same, different is the configuration such as *Database* , *Server* , etc.
- Therefore, during the *development* process , we cannot plac our application configuration *hardcodes* in our application *source code* , but separate them into different files based on the *environment.*



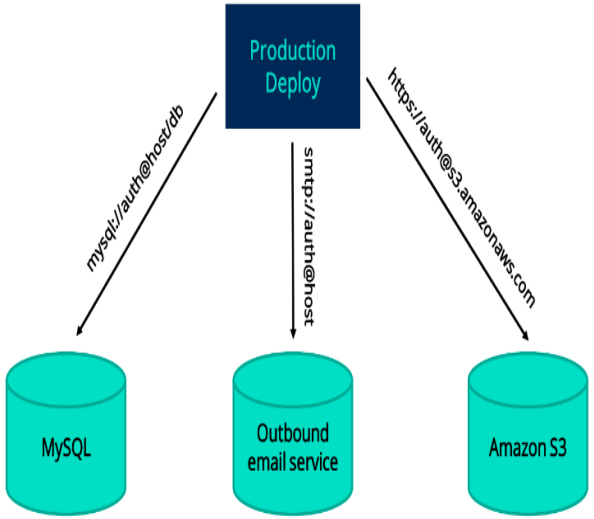
Twelve-Factor App methodology

Backing Service

- *Threat Backing Service as Attached Source*

The point here is, when the *development* process the application of all *services* connected to our application must be treated as a *source* that can be replaced with similar *services* .

So it is not *tighly-coupled* and depends only on one *source* , but can be replaced with a different *source* .



Twelve-Factor App methodology

Dev / Prod Parity

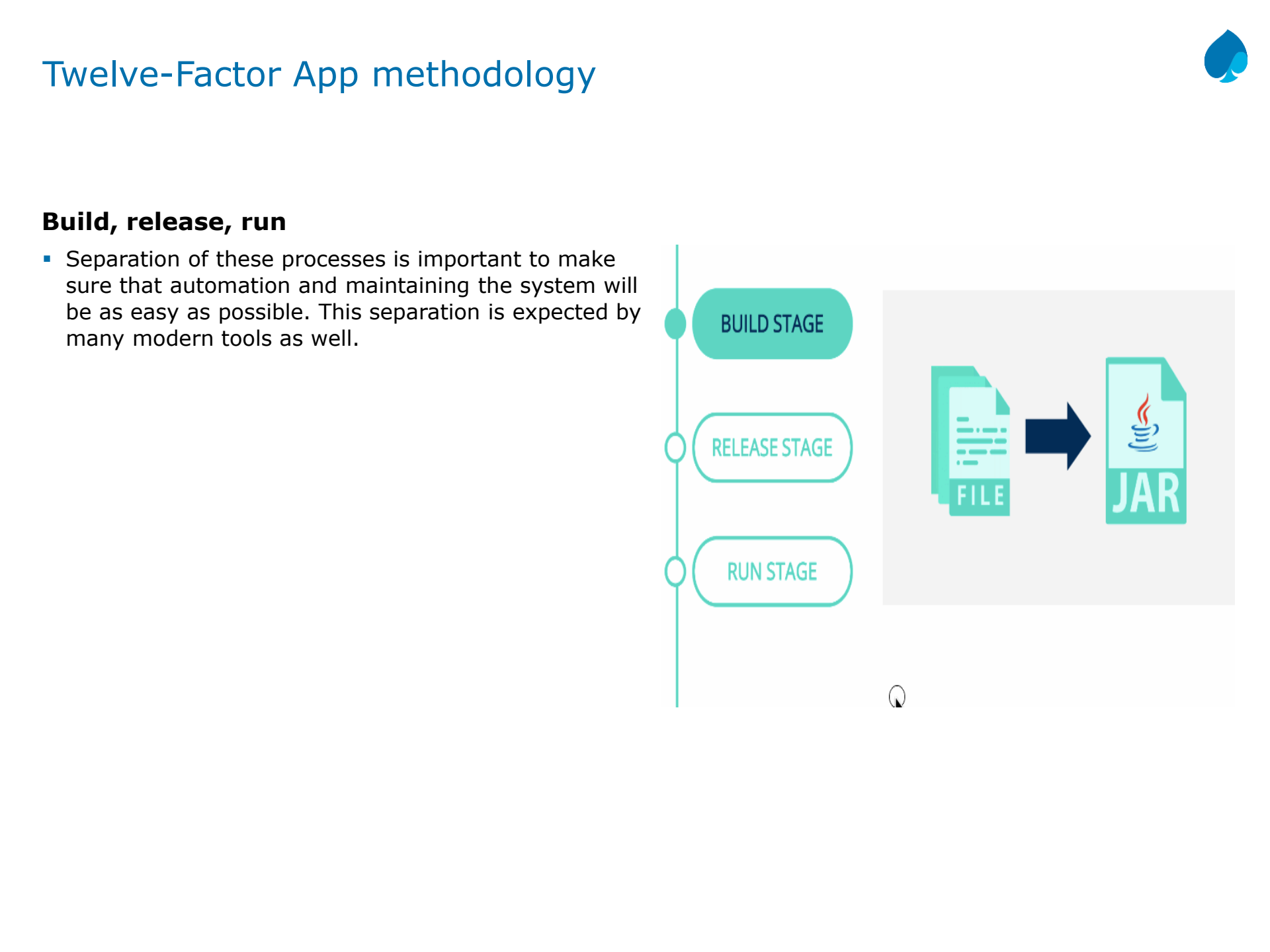
- *Keep development, staging, and production as similar as possible*
- During the application *development* process , we must reduce / minimize the Gap, such as *Time Gap, Personnel Gap,* and *Gap Tools* .
- *Well,* in the development of *modern software engineering* , this problem has been resolved with the *Continues Deployment* .
- With *continuous deployment* , we can reduce the gap. When the *code is* pushed to the *codebase*(git), *running tests* are automatically carried out and *deployed on everywhere*

Twelve-Factor App methodology

Processes

- *Execute the app as one or more stateless processes*
- The point is that the application we develop must be *stateless* (not storing *state*).
- All data must be stored in the *backing service* , not in the application.
- Including *cache* , files, etc. that are on the application should be stored in the *backing service* .
- We have to make sure when the application is *restarted* , then all the processes that have previously occurred are *clear* and not left on the *disk* or in the application memory.

Twelve-Factor App methodology



Build, release, run

- Separation of these processes is important to make sure that automation and maintaining the system will be as easy as possible. This separation is expected by many modern tools as well.

Twelve-Factor App methodology

Port binding

- Export services via port binding.
- This is quite simple- make sure that your service is visible to others via port binding (if you need it visible at all, possibly it just consumer from a queue). If you built a service, make sure that other services can treat this as a resource if they wish.

Concurrency

- Scale out via the process model
- Tools such as kubernetes can really help you here. The idea is that, as you need to scale, you should be deploying more copies of your application (processes) rather than trying to make your application larger (by running a single instance on the most powerful machine available).

Twelve-Factor App methodology

Disposability

- Maximize robustness with fast startup and graceful shutdown
- Fast startup is based on our ideas about scalability and the fact that we decided on microservices. It is important that they can go up and down quickly. Without this, automatic scaling and ease of deployment, development- are being diminished.
- Graceful shutdowns are arguably more important, as this is about leaving the system in a correct state. The port that was used should be freed so that the new server can be started, any unfinished task should be returned to the queue etc.
- One last thing… Crashes also need to be handled. These will be the responsibility of the whole system rather than just the service- just don’t forget about it.

Twelve-Factor App methodology

Logs

- Treat logs as event streams
- Bringing Splunk or Logstash/ELK Stack to help with your logs, can bring dramatic gains.
- Trends, alerts, heuristic, monitoring- all of these can come from well design logs, treated as event streams and captured by one of these technologies. Don’t miss out on that!

Admin processes

- Run admin/management tasks as one-off processes
- Admin tasks should be run from the relevant servers- possibly production servers.
- This is easiest done by shipping admin code with application code to provide these capabilities. The tools should be there even if they are not part of the standard execution of the service.

Summary

Service-oriented architecture (SOA) is a design approach where multiple services collaborate to provide some set of capabilities.

A monolithic application puts all its functionality in a single process where as microservices architecture puts each functionality into a separate service.

Monolithic apps – good for small organizations

MicroServices have its challenges, but the benefits are many