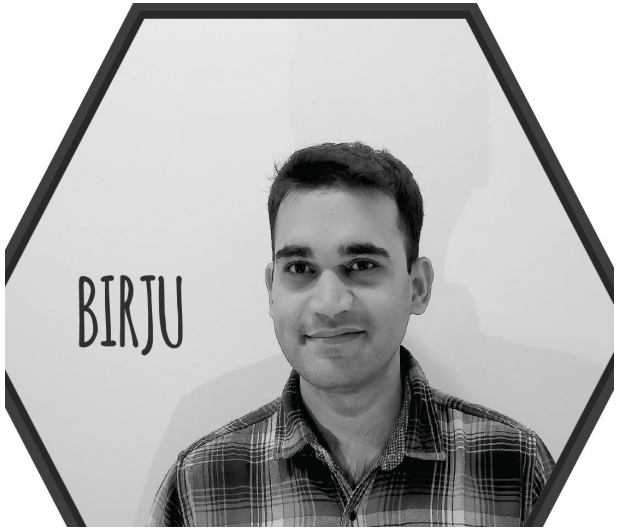
Spring Boot Microservices

Birju Shah - 17/09/2018 - 19/09/2019



Sr. Engineering Manager

Objectives

“By the end of three day training, you should have fair understanding about

What are microservices ?

When do you need microservices ?

Why there is a need of microservices architecture ?

How to design microservices architecture ?

Tools & Technologies to support microservices architecture with **Spring boot**.”

It’s a journey, not necessary a destination

Day 1

- Internet Basics
- Local Setup
- MVC
- Monolith Application
- Microservices

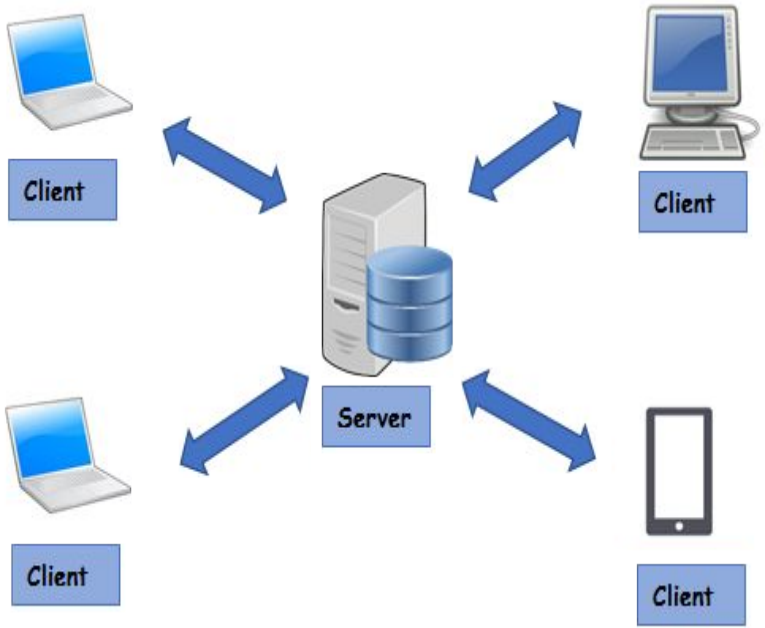
Day 2

- Principles
- Characteristics
- Architectural patterns
- Demo
- Communication

Day 3

- Security
- Data Mgmt.
- UI Mgmt.
- Deploy
- Q & A

Client Server Architecture



- **Server** acts as a **data provider**
- **Client** acts as a **data requestor**

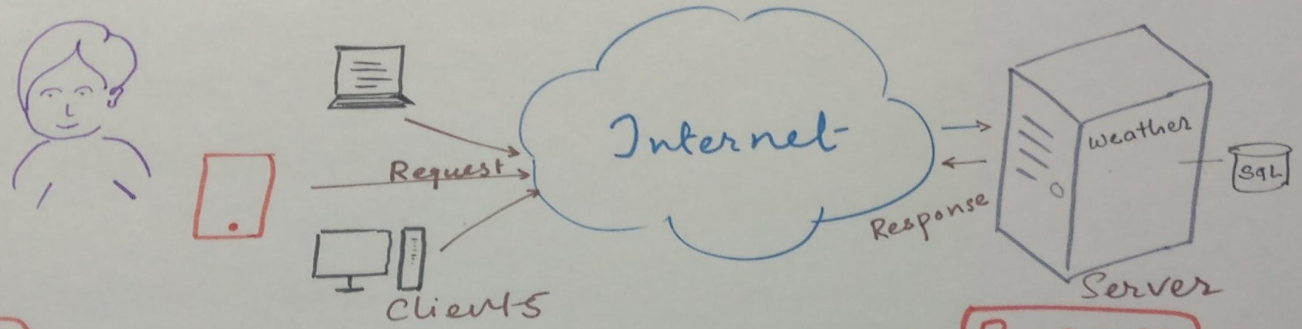
- **Client & Server** are assumed to be two computers separated by miles but connected via **web.**
- It is possible that both of them residing on the **same computer in two different processes**.
- It is also possible that both of them **residing in the same process**.

They communicate in their **local** language

- **A protocol : HTTP, RPC, TCP, UDP, FTP..**

Weather of Bangalore

http://weather.com/city/bangalore



Request

Request Line: GET /city/bangalore HTTP 1.1

Request Header:
- Request Method: GET
- " Time : 2018-09-17 13:30:00
- Accept Encoding: GZIP
- HOST Address : weather.com
- Client IP : 192.168.1.1
- Client Port : 42092
- User Agent : Mozilla....

Body → Request Body : {

Additional Content
- XML
- JSON

}

Response

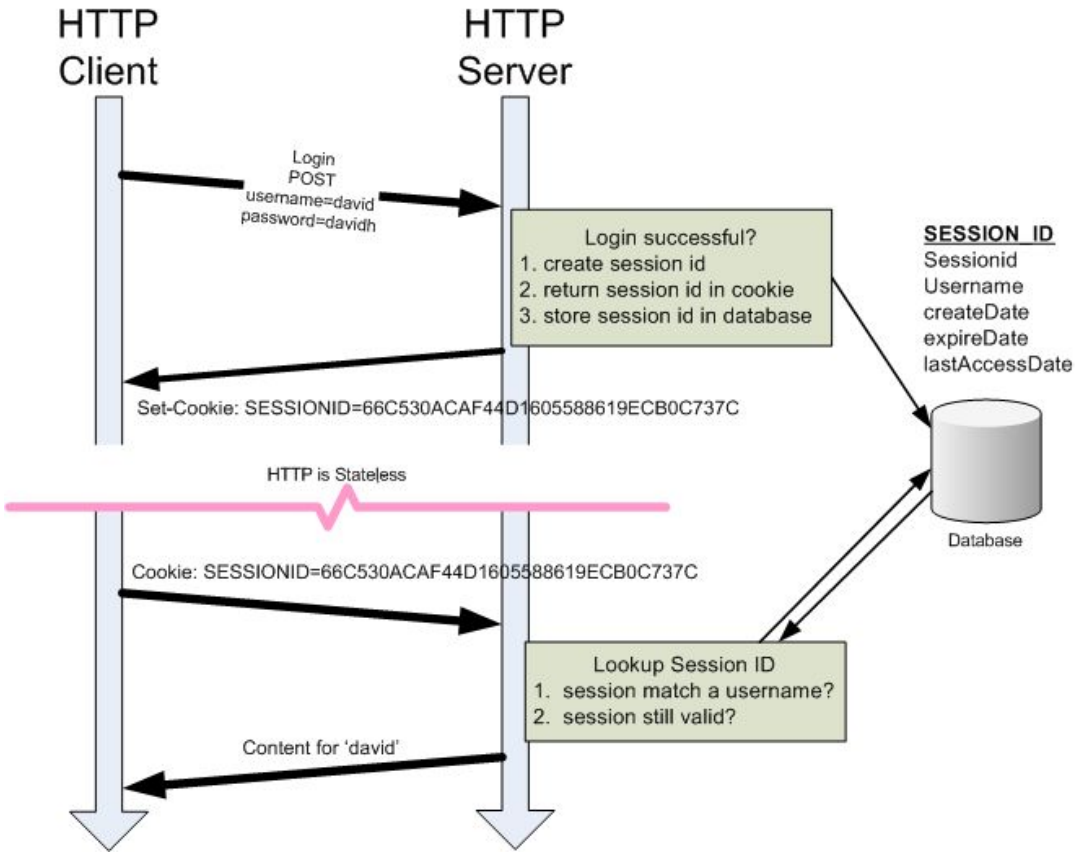
Response Header:
- Status : HTTP 1.0 200 Ok
- Server : Openresty
- Content-Type: application/JSON
- Content-Length: 536

Response Body:

```
{ "city": "Bangalore",
  "Temperature": "24.51 Degree Celsius",
  "Humidity": "94 percent",
  "Weather Desc": "Thunderstorm..."
  "wind-speed": "4.92 km. per hour",
  ....
}
```

HTTP Methods

Cookie & Session



HTTP is stateless

In order to map multiple requests from same client application server sets cookie value.

Client always shares cookie with every request to same host/ domain.

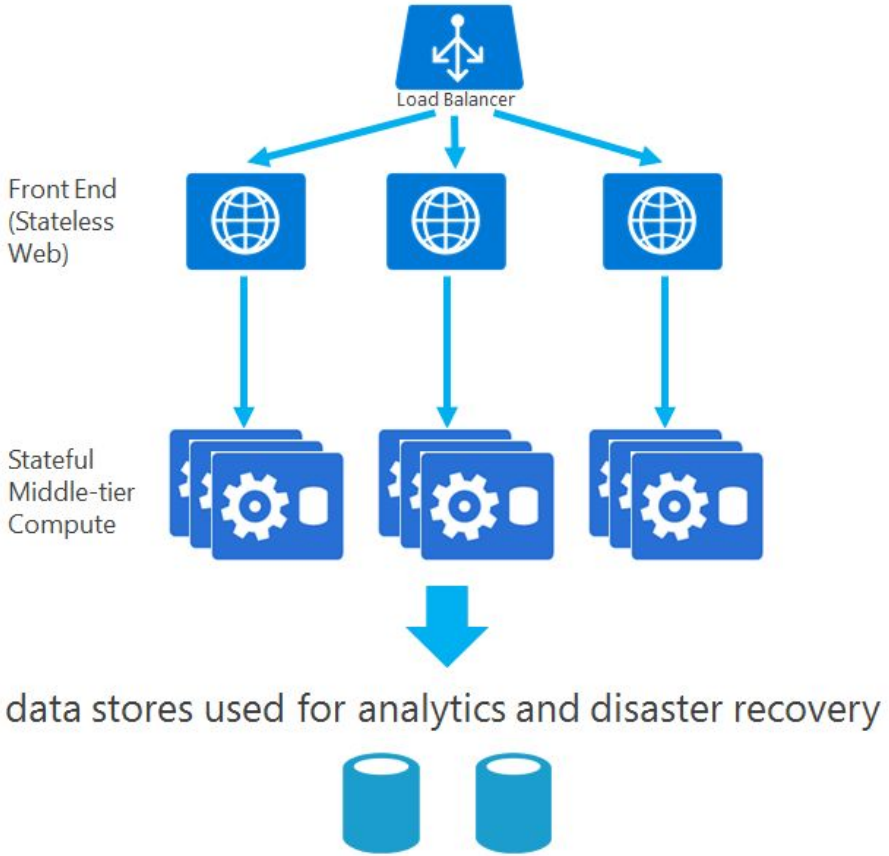
Cookie gets saved on client side
Session gets stored on server side (in file system / database)

Cookies are not shared across domains.

Sessions gets invalidated when client closes browser / session terminates after a specific time.

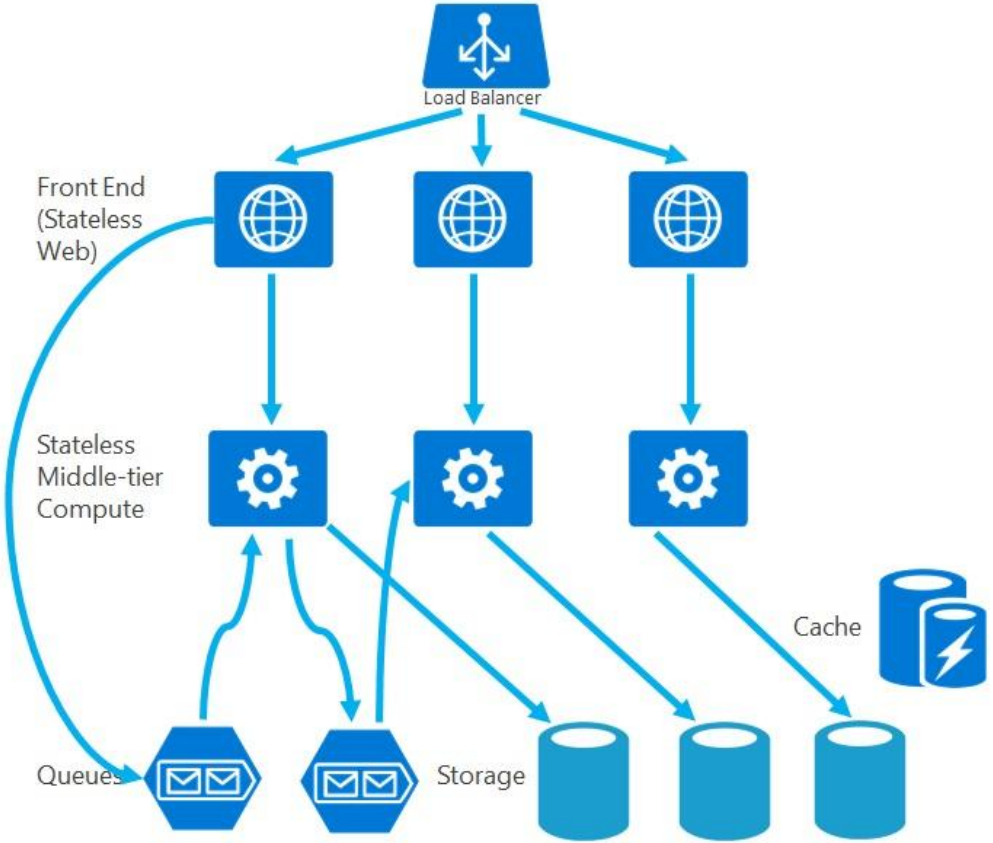
Stateful Web Application

- Application state lives in the compute tier
- Low Latency reads and writes
- Partitions are first class for scale-out
- Built in lock managers based on primary election
- Fewer moving parts



Stateless Web Application

- Scale with partitioned storage
- Increase reliability with queues
- Reduce read latency with caches
- Write your own lock managers for state consistency
- Many moving parts each managed differently



Introduction to groovy

- An agile dynamic **scripting** language for the Java Platform with many features that are inspired by languages like Python, Ruby and Smalltalk, making them available to Java developers using a **Java-like syntax**.

Scripting Language

- Productivity / Rapid Development / Agile
- Interpreted (No Compile Cycle)
- Expressive - Shorter, Less Verbose Syntax
- Feature Rich yet Simple and Productive
- Shell / System Integration / Systems “Glue”
- Dynamic
- Open Source – also code available by default
- Advanced languages features Closures / Mix Ins


```
println 'Hello Groovy'
```

Types

- Everything is an **object,** no primitive types.
- Primitive types are auto boxed
- Optional Typing – If not explicitly specified assumed to be java.lang.Object
- **Type safe** – Unlike some scripting languages, Groovy doesn't allow one type to be treated as another without a well defined conversion being available.

```
a = 1         // Implicit typing to Integer
b = 'howdy'   // Implicit typing to String
int c = 33    // Explicit typing to Integer
def d = 5.2f // def keyword means any type

println 'a is ' + a.class.name
println 'b is ' + b.class.name
println 'c is ' + c.class.name
println 'd is ' + d.class.name  // class name as
property
```

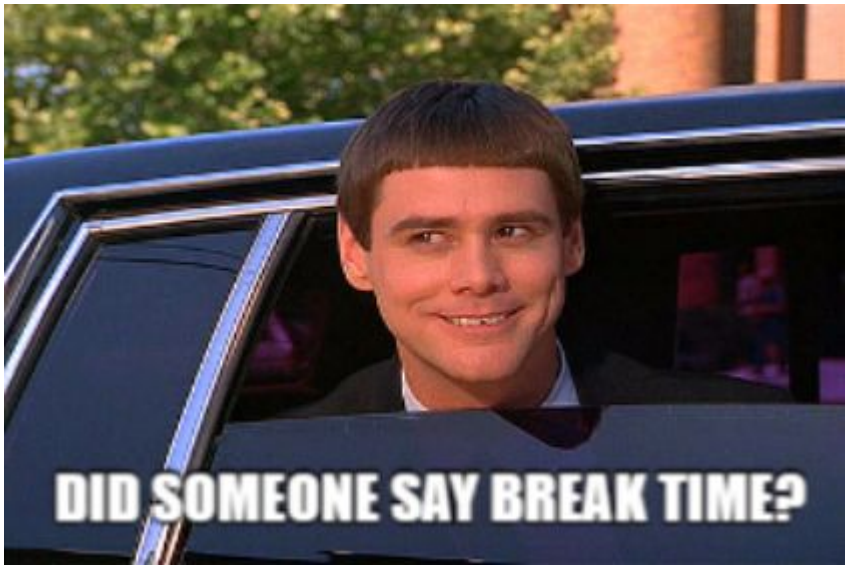
Advantages

- Lots of additional methods in GDK
- Support of CLOSURES
- Maps & List : helper methods
- **?** operator without null pointer
- Many ways to construct n object
- Default imports lang, util, io, net
- Default groovy imports : lang, util
- Auto generates accessor methods
- Syntax like invoices.items*.total() helps in readability

```
def list = [1,2,3]

// Closure to print contents of a list
def closure = { x -> println x }
list.each(closure)

// Simplify 1 - Create closure within each
list.each({ x -> println x })

// Simplify 2 - () not required if closure last param
list.each { x -> println x }

// Simplify 3 - 'it' is default parameter name if one param
list.each { println it }
```

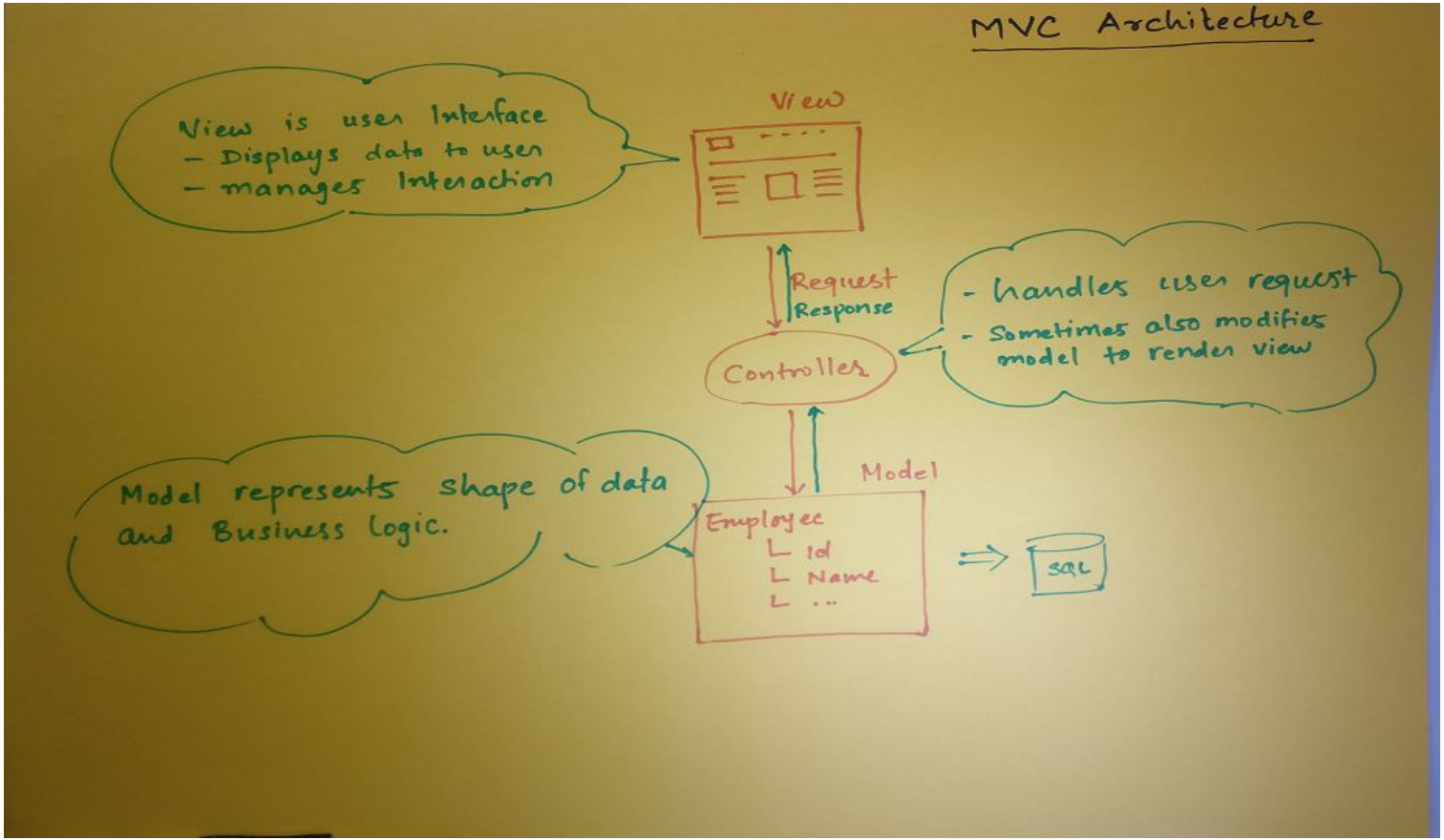
Spring Boot Initializer

- Don’t worry, This is just to get our hands dirty.
- Don’t worry if you do not understand anything.
- Install “Groovy” , “Java”, “Maven”
- Kindly visit : https://start.spring.io/
- Group : com.example, Artifact : demo, Dependencies : Web (select)
- Click on “Generate Project”

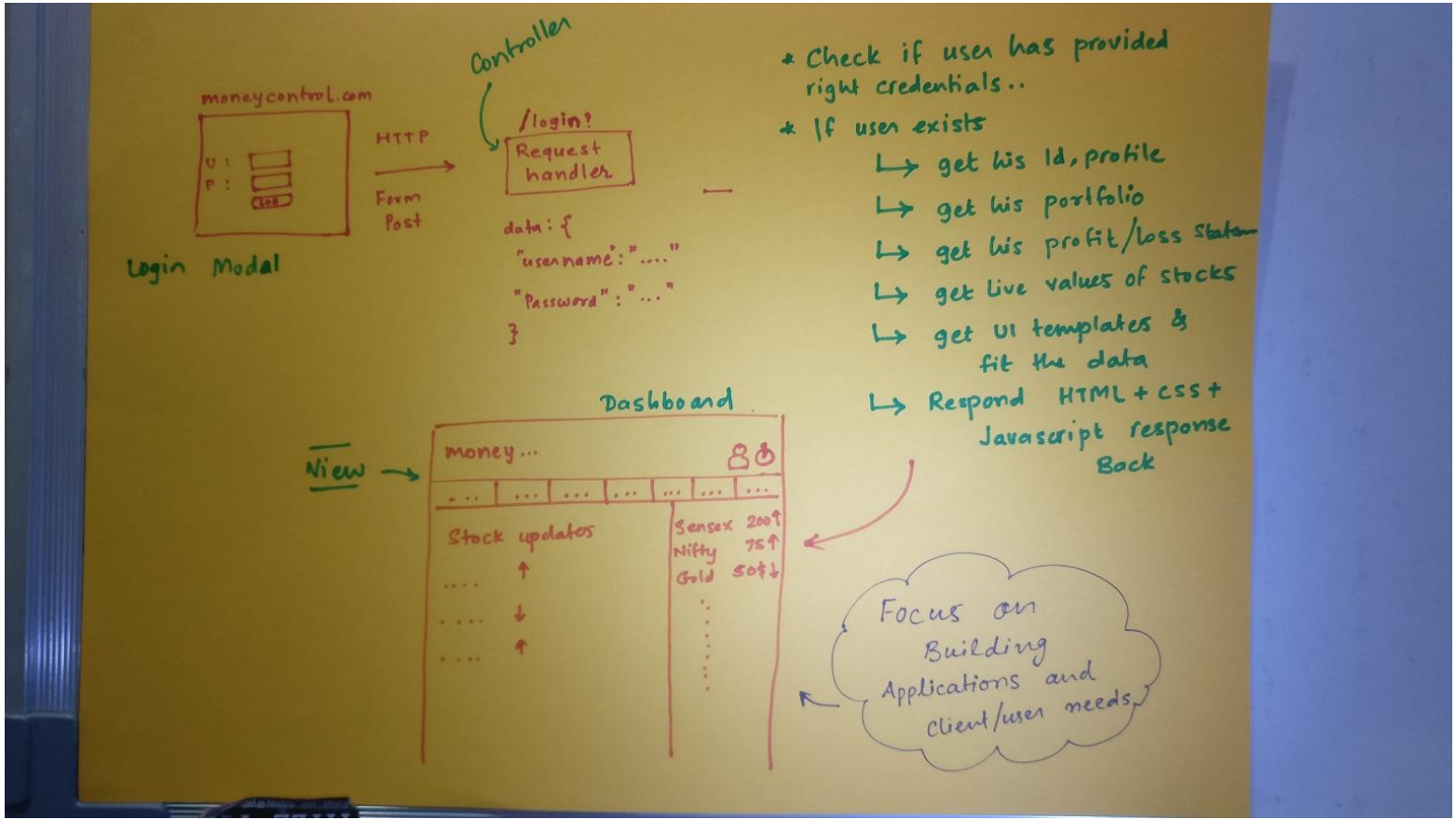


Shall regroup in 20 !

MVC Architecture



MVC Architecture Example



Designer Created the best design :)

User created his own shortcut !!!

WHY ??

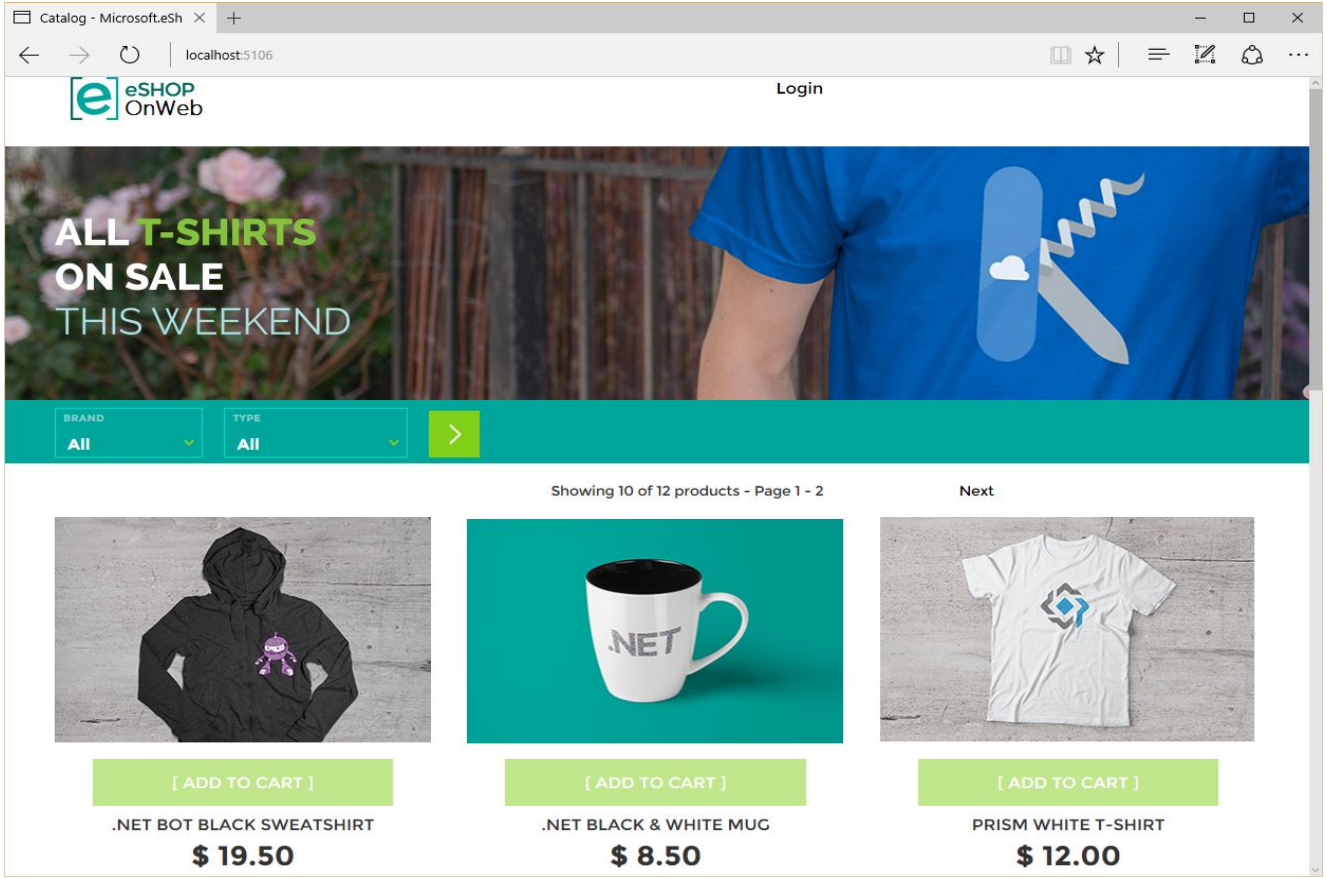
Mismatch

User Experience Demands Modern Web

UI

UX

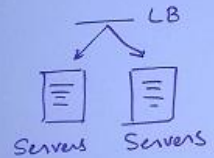
User Experience Demands Modern Web

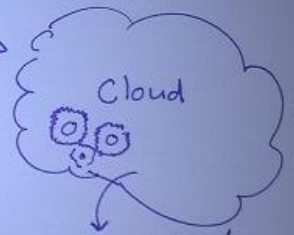


Engineers :- Functionality Focus

Product Managers :- Business needs
User Experience

Developers & Operations } devops :-





- autoscale
- geo distribution
- Operations as service
- Cost
- Low latency
- CDN

Traditional Web App Vs SPA

Factor	Traditional Web App	Single Page Application
Required Team Familiarity with JavaScript/TypeScript	Minimal	Required
Support Browsers without Scripting	Supported	Not Supported
Minimal Client-Side Application Behavior	Well-Suited	Overkill
Rich, Complex User Interface Requirements	Limited	Well-Suited

Tell me something about ..

Would like each one of you to tell me about..

- What is your role ?
- How the team is structured.
- What are the processes / standards you follow.

Design

“*If the fundamental design of the system doesn’t make it easy to make changes, then there are limits to what can be accomplished.*”

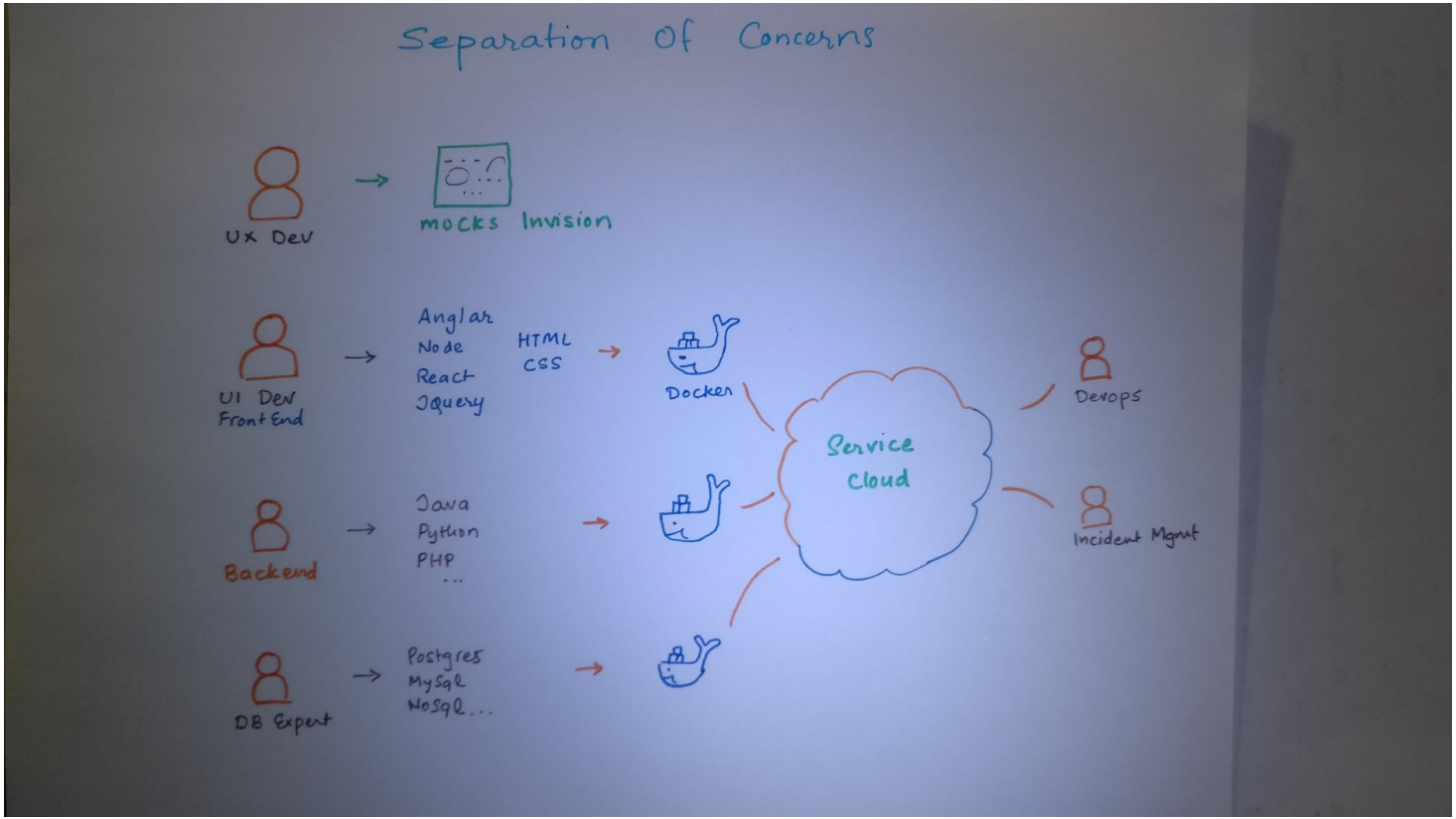
"If builders built buildings the way programmers wrote programs, then the first woodpecker that came along would destroy civilization."

- Gerald Weinberg

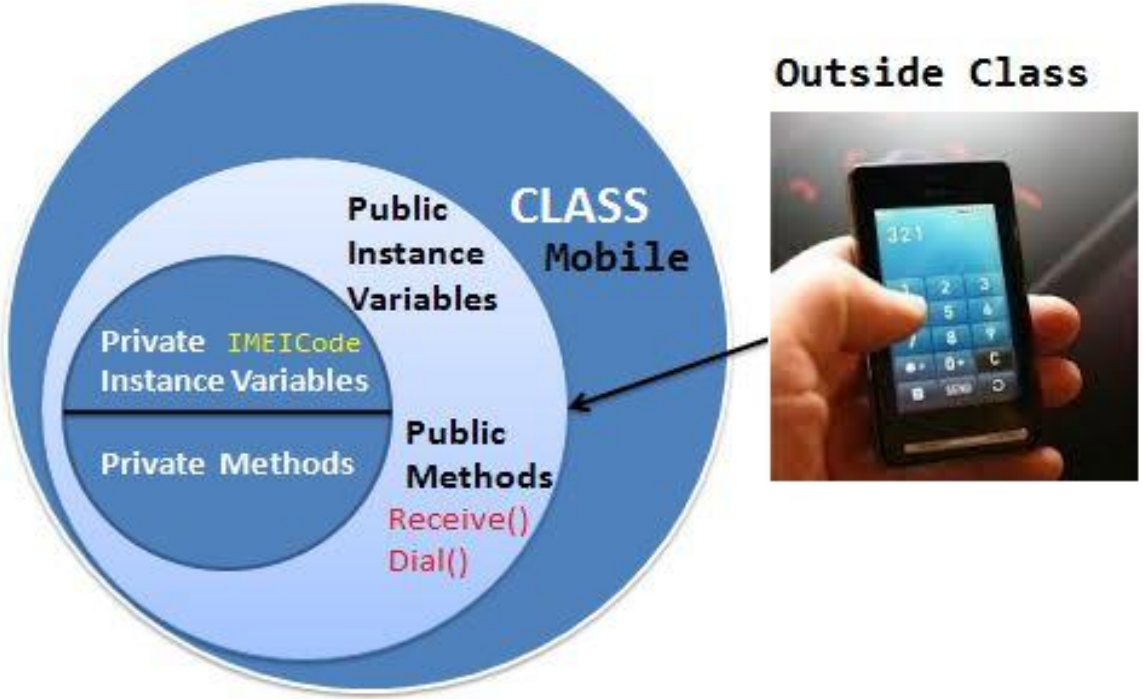
Application Design Principles

- Separation of concerns
- Encapsulation
- Dependency Inversion
- Single responsibility
- Don’t Repeat Yourself
- Bounded context
- Liskov’s substitution principle

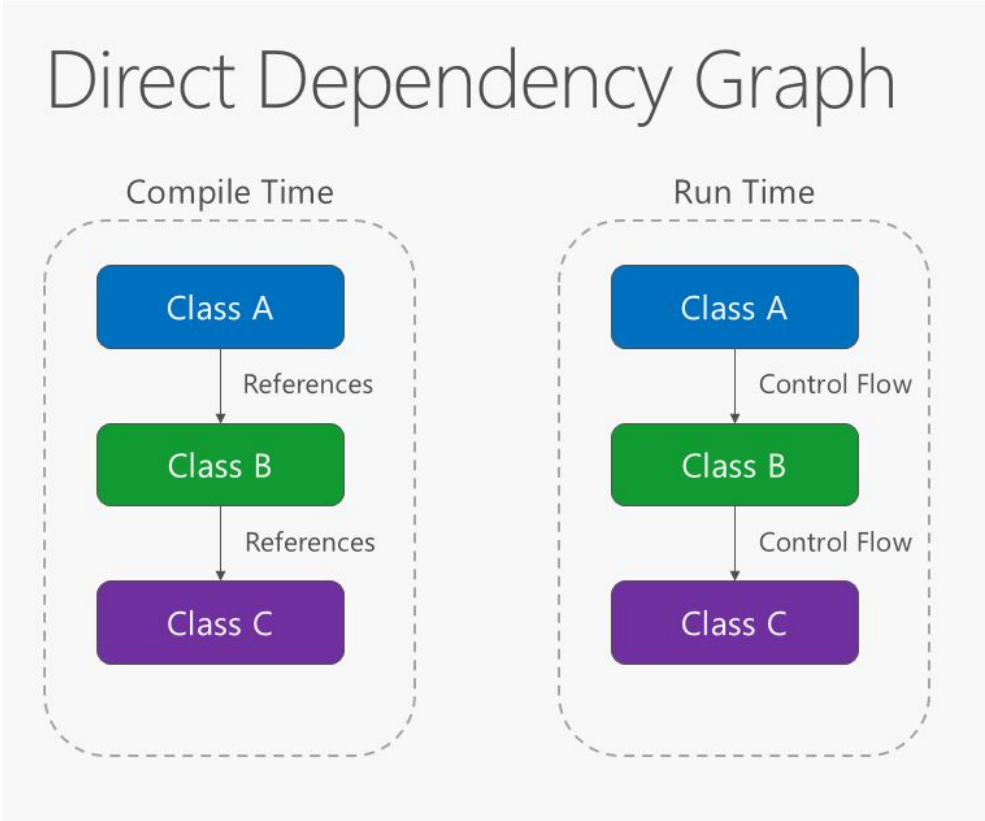
Separation of concerns



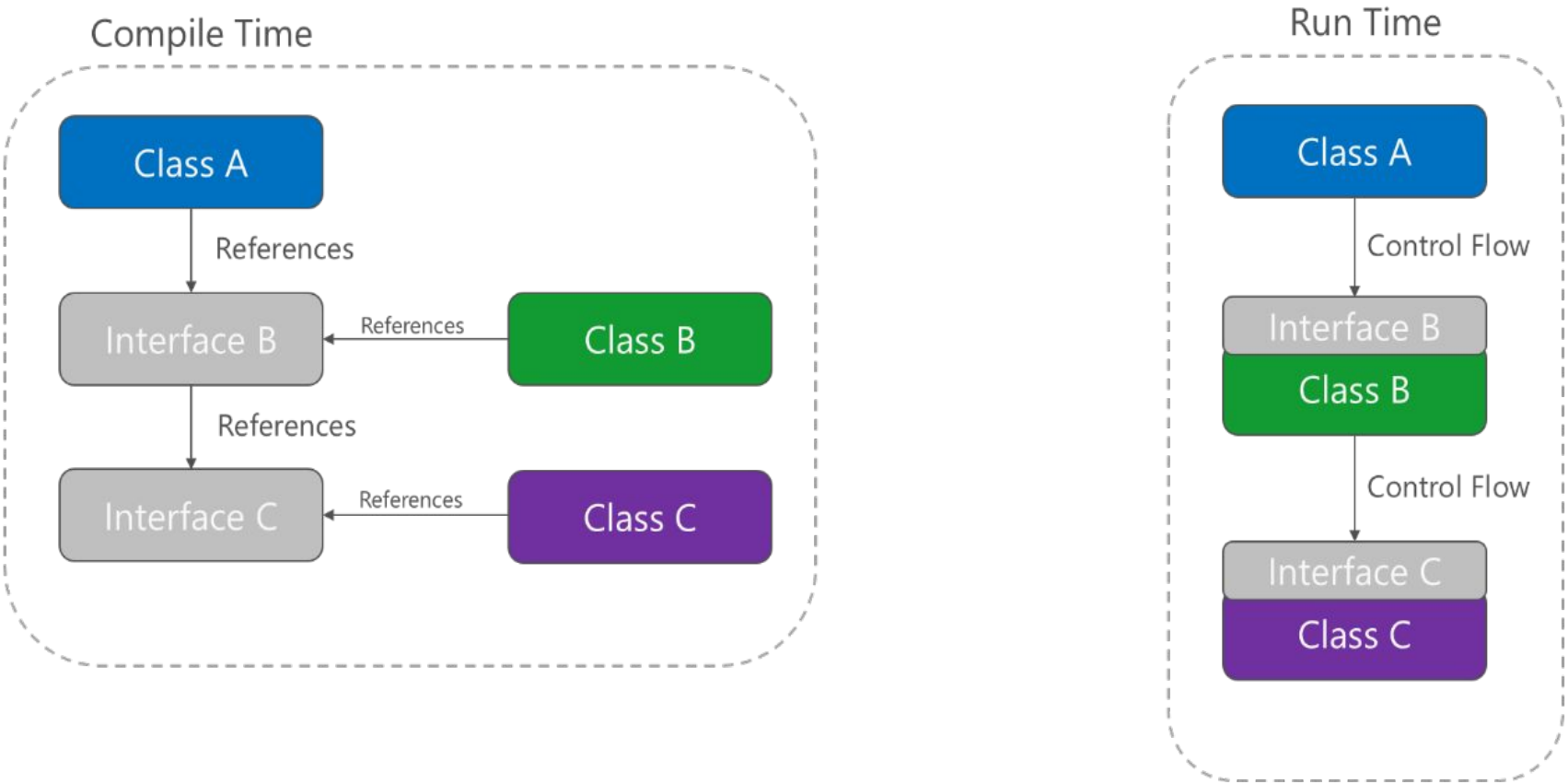
Encapsulation

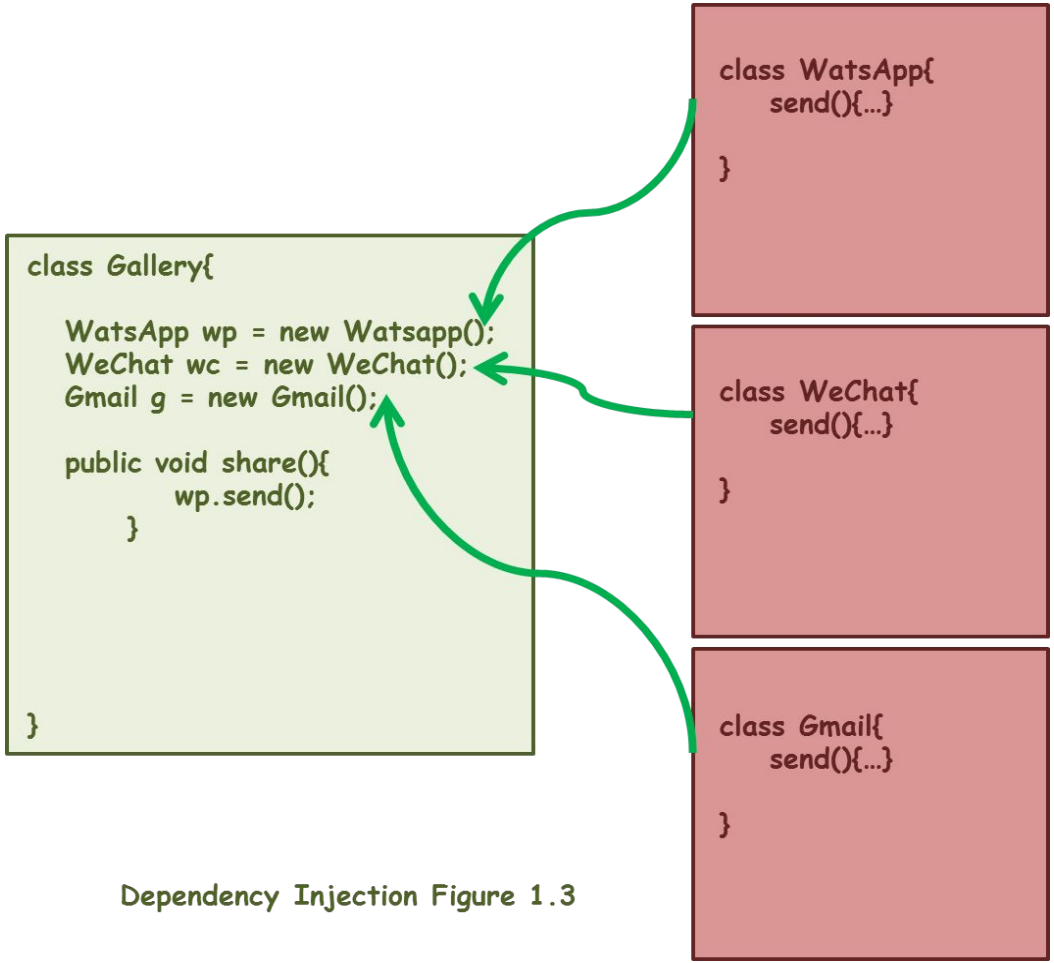


Dependency Inversion

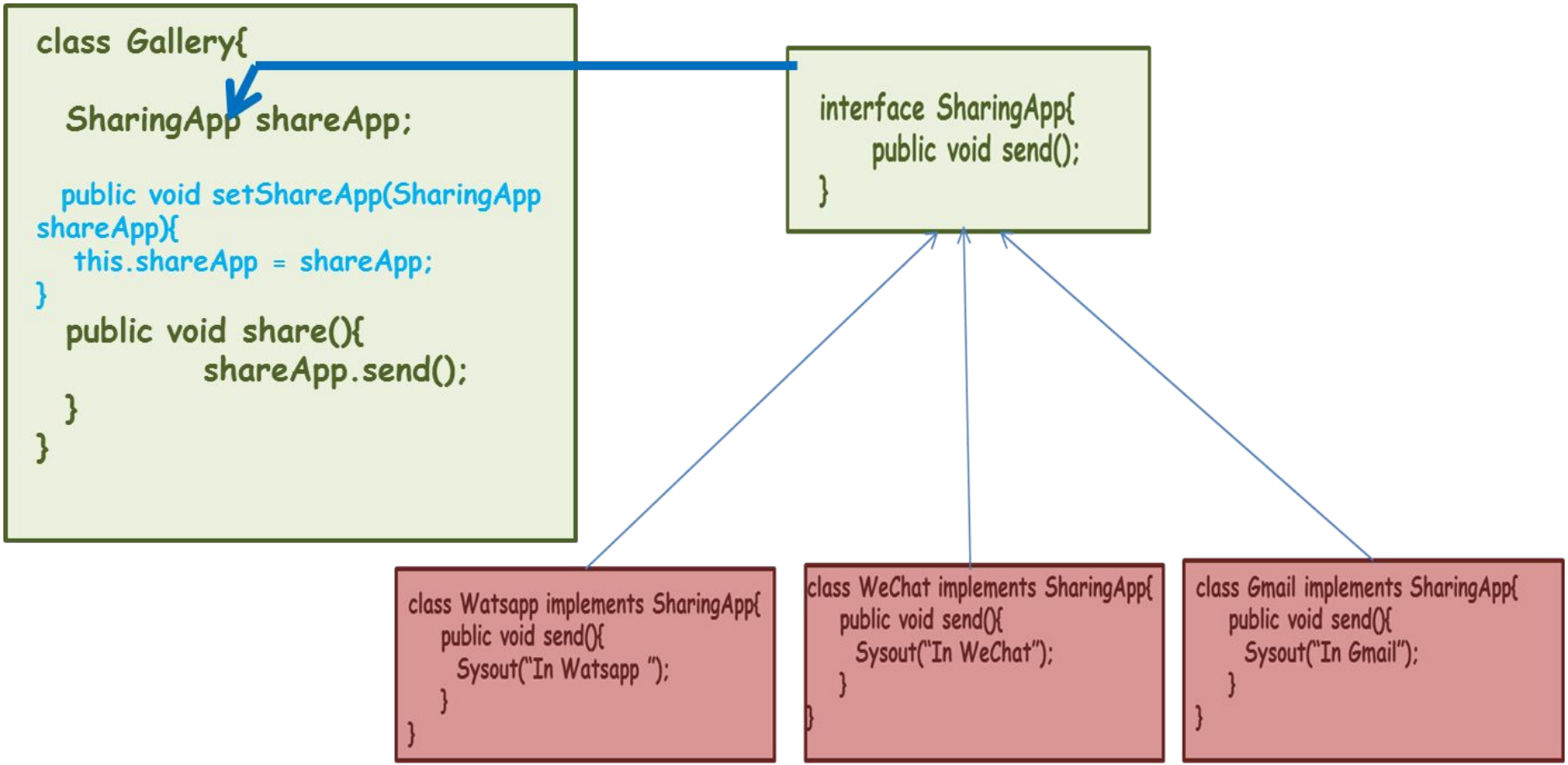


Inverted Dependency Graph





Dependency Injection Figure 1.3



Dependency Injection Figure 1.6

New is glue !

```
using (var client = new SmtpClient())
using (var message = new MailMessage(fromEmail, toEmail))
{
    message.Subject = subject;
    message.Body = bodyHtml;
    message.IsBodyHtml = true;
    client.Send(message);
}
```

Note the two new keywords here, gluing whatever else this class is doing to this implementation of message sending. This can be replaced with an interface like this one:

```
1 public interface IEmailClient
2 {
3     void SendHtmlEmail(string fromEmail, string toEmail,
4                        string subject, string bodyHtml);
5 }
```

Now the code that used to contain the first block of code can be rewritten to simply use the interface:

```
1  public class SomeService
2  {
3      private readonly IEmailClient _emailClient;
4      public SomeService(IEmailClient emailClient)
5      {
6          _emailClient = emailClient;
7      }
8      public void DoStuff(User user)
9      {
10         string subject = "Test Subject";
11         string bodyHtml = GetBody(user);
12         _emailClient.SendHtmlEmail("noreply@whatever.com", user.EmailAddress, subject, bod
13     }
14 }
```

- Only initialize with new keyword if you are sure and taking informed decision
- Favour contractors over employees analogy !
- Car mycar = new Audi( ); if we have this line at multiple places .. we are actually violating the next principle **DRY.**

Single Responsibility Principle

Cohesion : The drive to have related code grouped together.

Can be achieved by

“Single Responsibility Principle” By *Robert C. Martin*

“Gather together those things that change for the same reason”

And

“Separate those things that change for different reasons.”

Definition

Clip slide

"Every object should have a single responsibility, and that responsibility should be entirely encapsulated by the class."

*"Classes should have one responsibility - **one reason to change**."*

Information Control Corporation
Your Success. It's Our Business.

```
public class UserManager
{
    private readonly GetUserDbCommand _getUserDbCommand;
    private readonly AddUserDbCommand _addUserDbCommand;
    private readonly DeleteUserDbCommand _deleteUserDbCommand;
    private readonly SendEmailConfirmation _sendEmailConfirmation;

    public UserManager(GetUserDbCommand getUserDbCommand,
                       AddUserDbCommand addUserDbCommand,
                       DeleteUserDbCommand deleteUserDbCommand,
                       SendEmailConfirmation sendEmailConfirmation)
    {
        _getUserDbCommand = getUserDbCommand;
        _addUserDbCommand = addUserDbCommand;
        _deleteUserDbCommand = deleteUserDbCommand;
        _sendEmailConfirmation = sendEmailConfirmation;
    }

    public User GetUser(int userId)
    {
        return _getUserDbCommand.Execute(userId);
    }

    public void RegisterUser(string email, string password)
    {
        if (string.IsNullOrEmpty(email) || !email.Contains("@"))
            throw new ValidationException("The e-mail address is not valid");
        if (string.IsNullOrEmpty(password))
            throw new ValidationException("The password cannot be empty");

        _addUserDbCommand.Execute(email, password);
        _sendEmailConfirmation.Send(email);
    }

    public void DeleteUser(int userId)
    {
        _deleteUserDbCommand.Execute(userId);
    }
}
```

UserManager looks like a facade class

... Extract

Clip slide

Don't Repeat Yourself (DRY) Principle

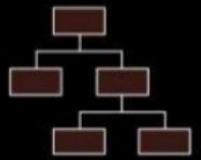
> *"Every piece of knowledge must have a single, unambiguous representation in the system."*
>
> **The Pragmatic Programmer**

> *"Repetition in logic calls for abstraction. Repetition in process calls for automation."*
>
> **97 Things Every Programmer Should Know**

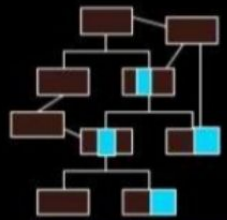
- DRY principle variations:
  - Once and Only Once (OOO)
  - Duplication Is Evil (DIE)

Open Closed Principle

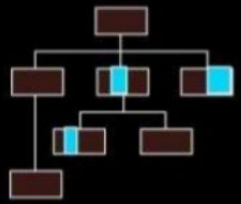
Usual way:



Starting code base

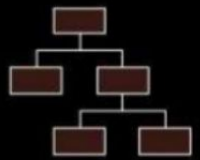


Changes implemented
blue == code changed

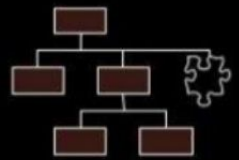


(Hopefully) Code cleaned up

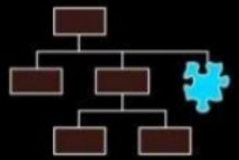
OCP:



Starting code base



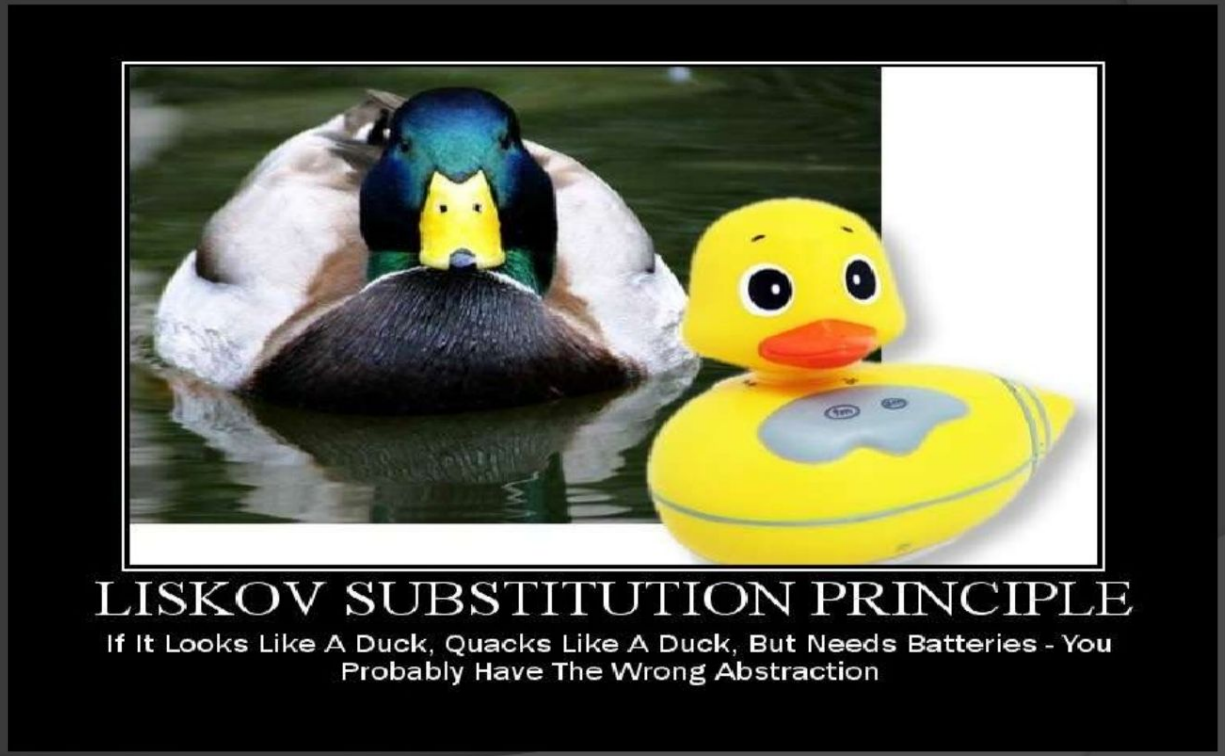
Change design to make
room for new feature



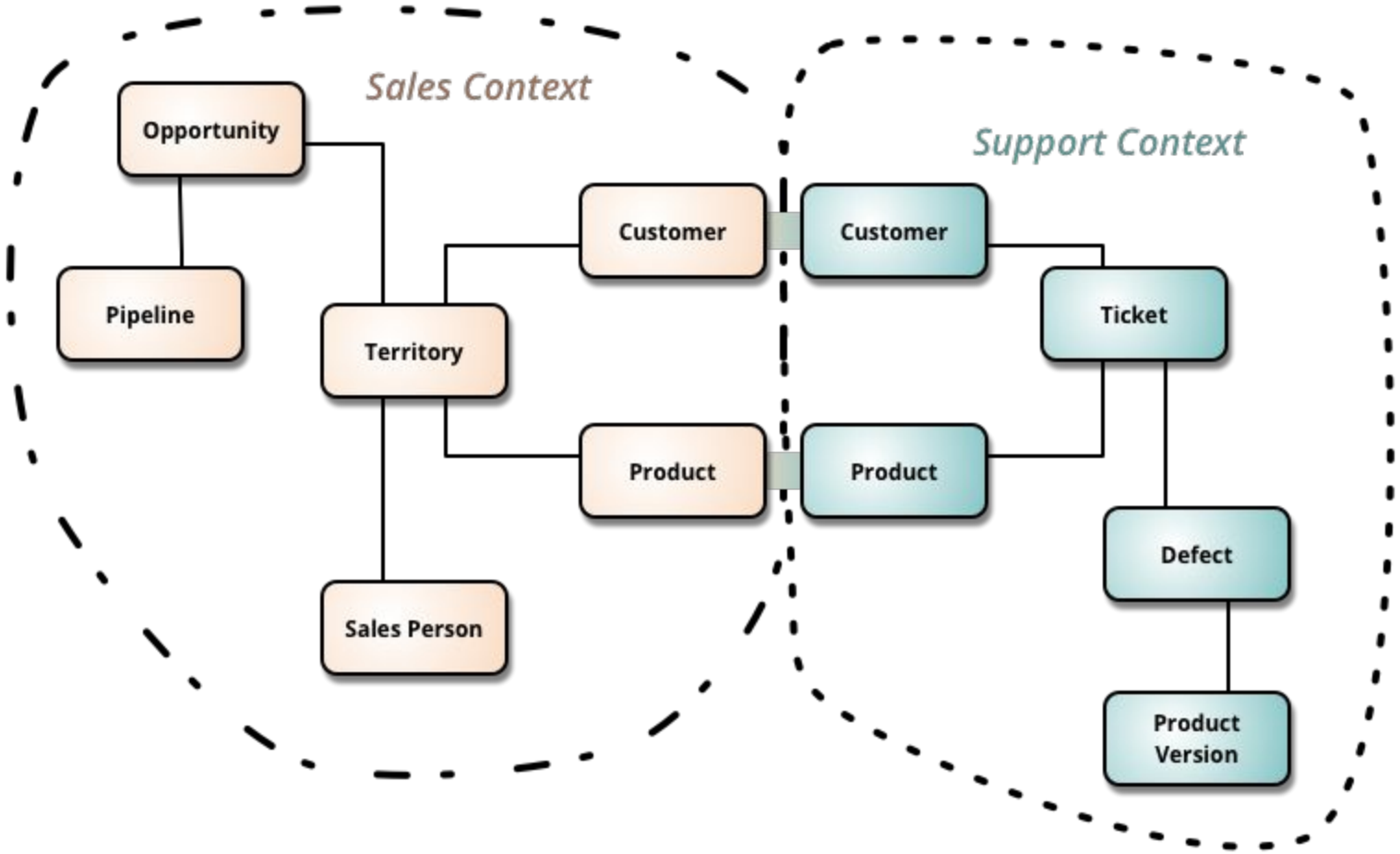
Implement feature

Clip slide

Likov's Substitution Principle



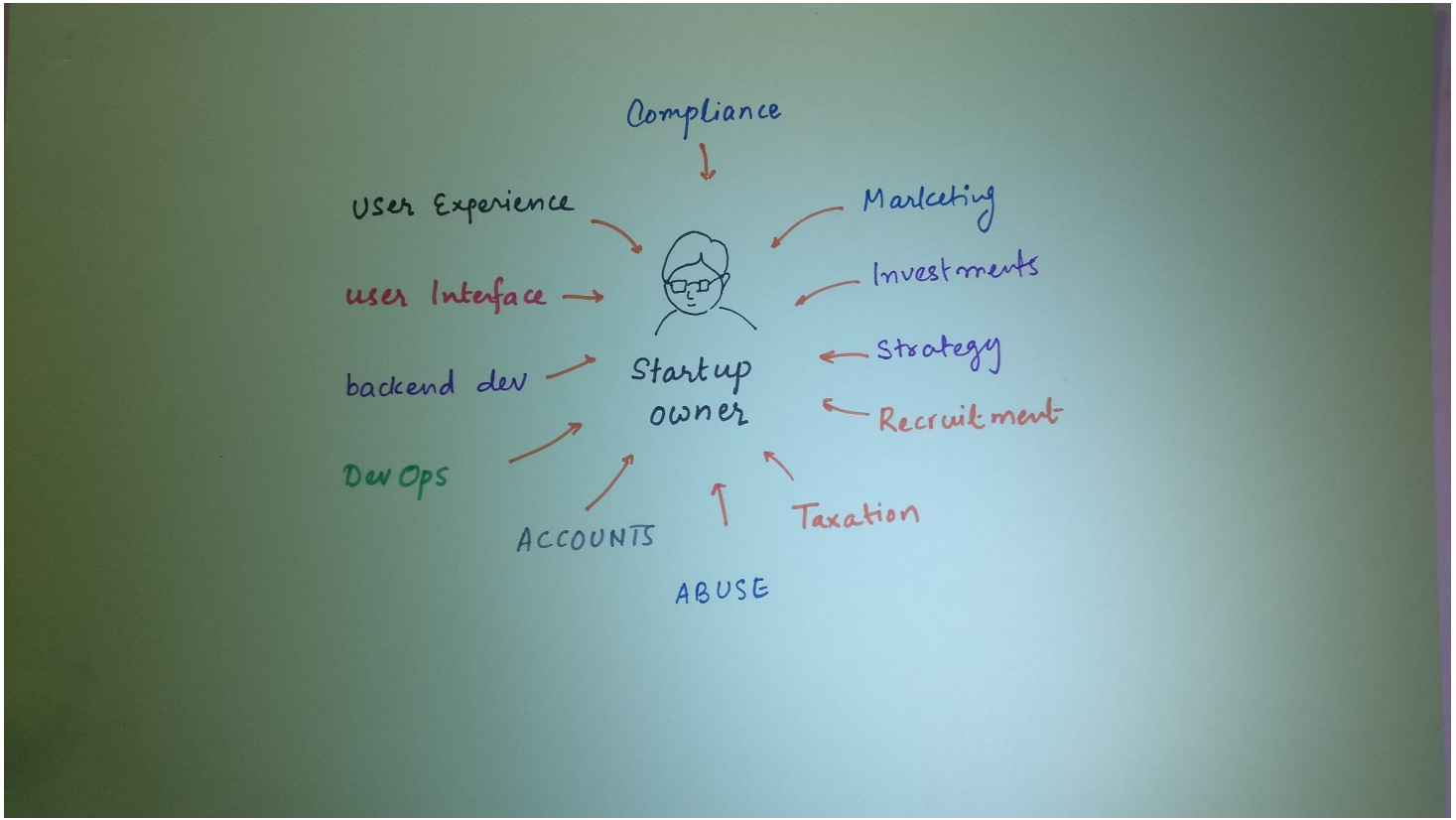
Bounded context !



Shall regroup after an hour !

Let’s start a company :)

The state of a startup owner :)



To support our business

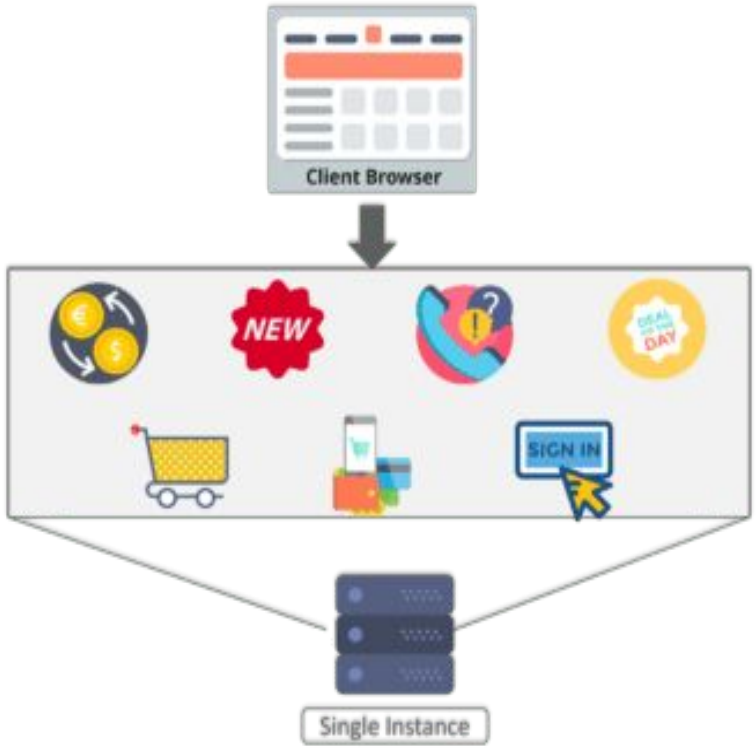
- We want to start selling products online. Rivals are amazon and flipkart

  In pair of 3, let’s try to design this application.

  #halfnhour for design, **#halfnhour** for understanding

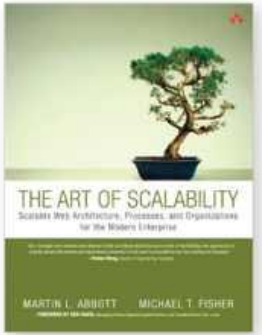
  Objective : We will take one example from your designed applications and might take it through our microservices journey.

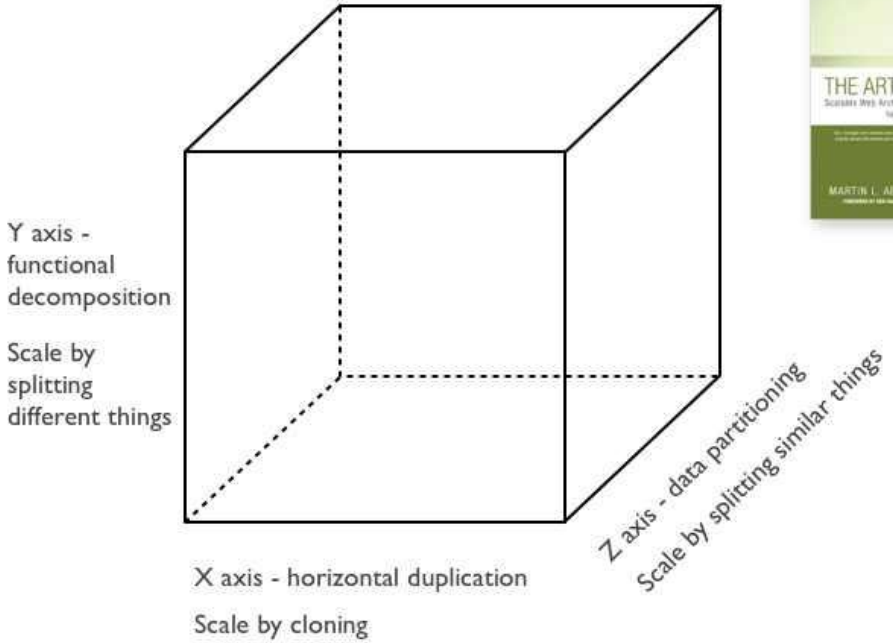
Monolith !



Scaling Monolith !

3 dimensions to scaling





Scalability

* Ability to handle Increased demands
* Ability of System
  - to perform efficiently
  - to be available, reliable & responsive
  - Without Increasing cost exponentially
* With Limited Resources... achieve maximum output.

"Itne paise mein itna hi milega".

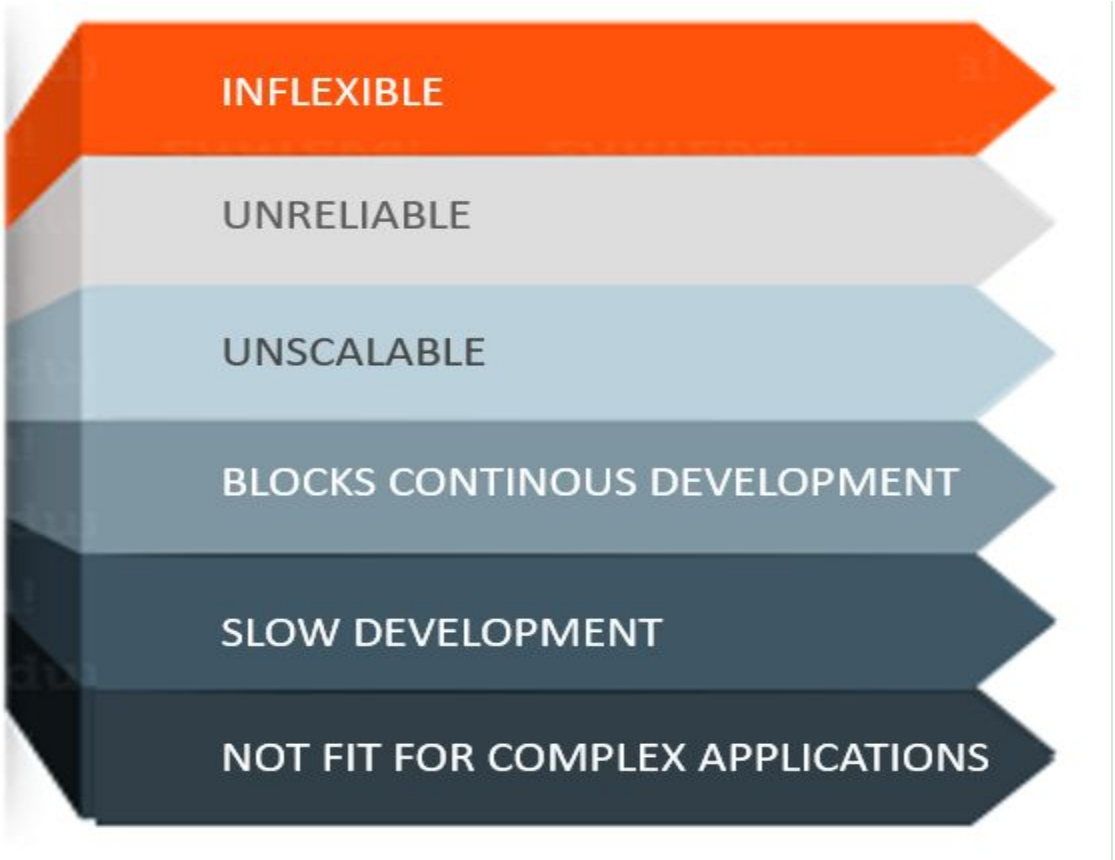
What needs to be done to achieve highest Scalability?

* Optimize Code
* Resource utilization CPU, Memory, Disc
* User Limits
* Process Limits
* Logs
* HTTP Static / Dynamic
* Database
* Load balancer
* Sessions
* Events
* Active - Active Setup
  master - Slave Setup

→

* Connection Pool
* Connection. Params
* Read / writes
* Partitioning
* Caching
* Persistant -
* Optimize queries
* Locks
* Indexes

Monolith !


INFLEXIBLE
UNRELIABLE
UNSCALABLE
BLOCKS CONTINOUS DEVELOPMENT
SLOW DEVELOPMENT
NOT FIT FOR COMPLEX APPLICATIONS

Shortcomings of monoliths

- **Inflexible –** Monolithic applications cannot be built using different technologies
- **Unreliable –** Even if one feature of the system does not work, then the entire system does not work
- **Unscalable –** Applications cannot be scaled easily since each time the application needs to be updated, the complete system has to be rebuilt
- **Blocks Continuous Development –** Many features of the applications cannot be built and deployed at the same time. Hard to manage **dependencies**.
- **Slow Development –** Development in monolithic applications take lot of time to be built since each and every feature has to be built one after the other
- **Not Fit For Complex Applications –** Features of complex applications have tightly coupled dependencies
- **Slow Deployment -** With multi million Line of code - it takes time to restart, deploy, manage dependencies (Jars, Versions)
- **Compromise -** Need to compromise at many levels during the development life cycle. We always tend to take calls for short term to deliver the features faster.

Shortcomings of monoliths

- Code becomes duplicate
- Difficult to understand by developer on such huge code
- More time required for bug fixing
- More time required for new enhancement
- High Chances of error to be occurred on unknown areas

It may happen then monoliths works fine.. Generates revenue.. But the threat and risk remains

Monolithic Apps: The good, the bad, the ugly…

Can be easier to test	Can be easier to develop	Can be easier to deploy	Can't deploy anything until you deploy everything
Harder to learn and understand the code	Easier to produce spaghetti code	Hard to adapt to new technologies	Have to scale everything to scale anything

Shall regroup in 20 !

Microservices

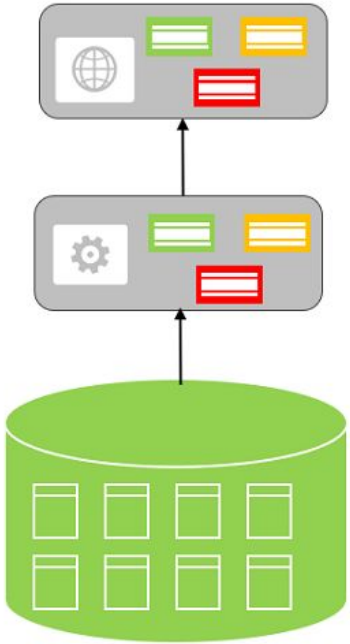
“ *Microservices are small autonomous services that work together.*”

“***At its simplest, the microservices design approach is about a decoupled federation of services, with independent changes to each, and agreed-upon standards for communication***.”

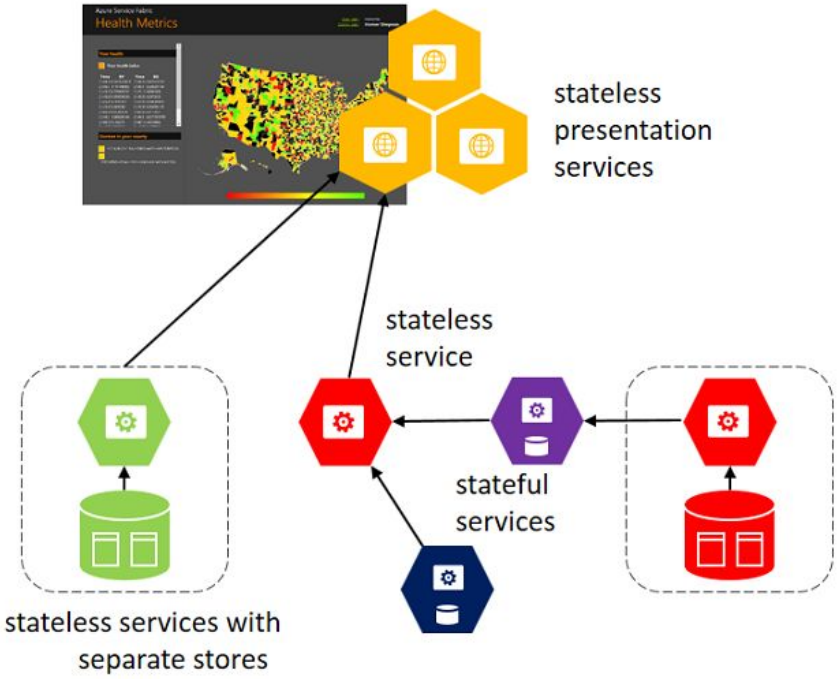
Let’s make sure... we are not doing this !! using microservices....

‘Good work guys, why don’t you take a break’ by Meme Binge

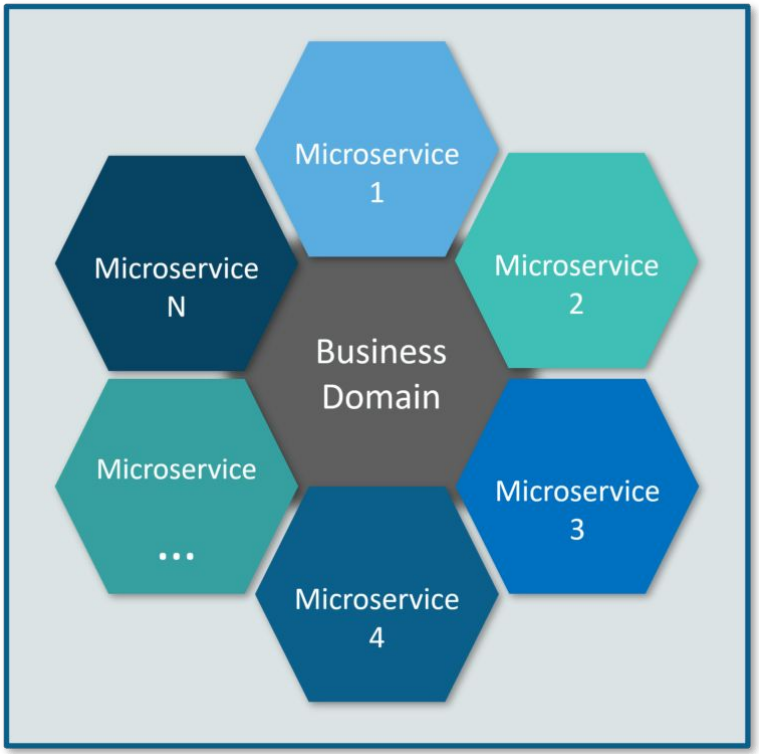
State in Monolithic approach

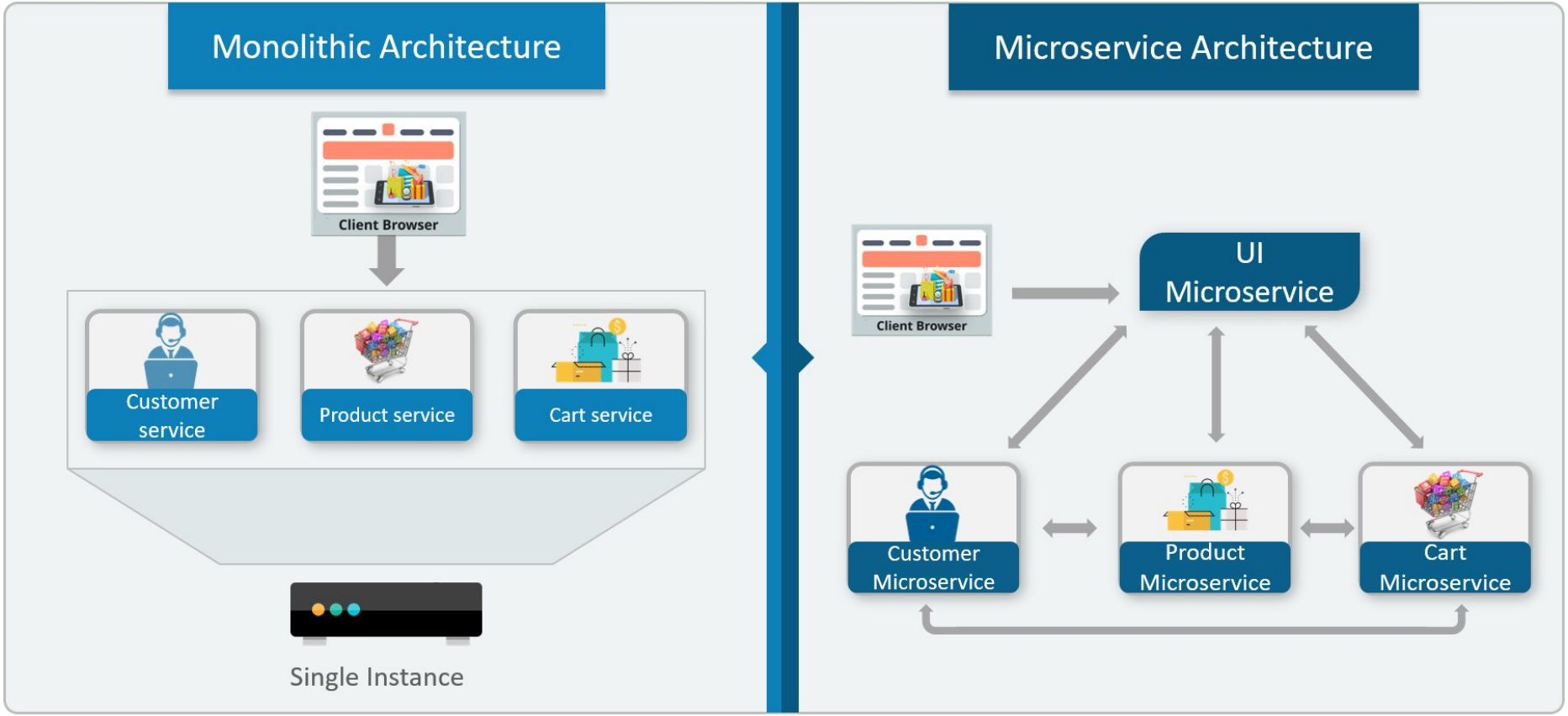


State in Microservices approach



Microservices, aka ***Microservice Architecture***, is an architectural style that structures an application as a collection of small autonomous services, modeled around a **business domain.**

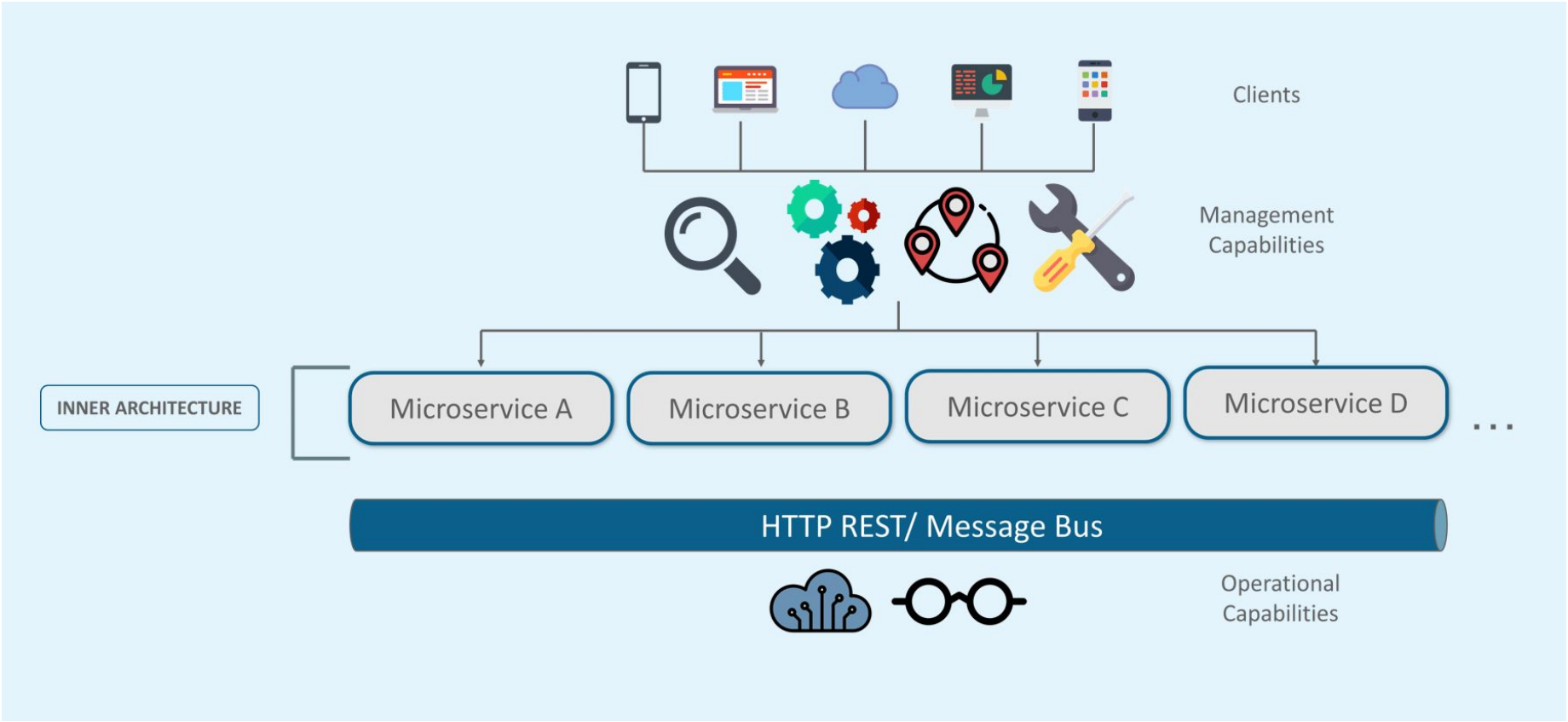



Monolithic Architecture
Client Browser
Customer service
Product service
Cart service
Single Instance
Microservice Architecture
Client Browser
UI Microservice
Customer Microservice
Product Microservice
Cart Microservice

Microservices Architecture

- Different clients (Different responses possible)
- Services are separated by domain / functionality
- Services have their own load balancer , execution environment & have independent store if required.
- Communicate with each other via synchronous / asynchronous method
- Each service is critical, needs to be monitored

Microservices Architecture



What do we get by using microservices ?


ARRE BHAI
AAKHIR KEHNA KYA CHAHTE HO
quickmeme.com

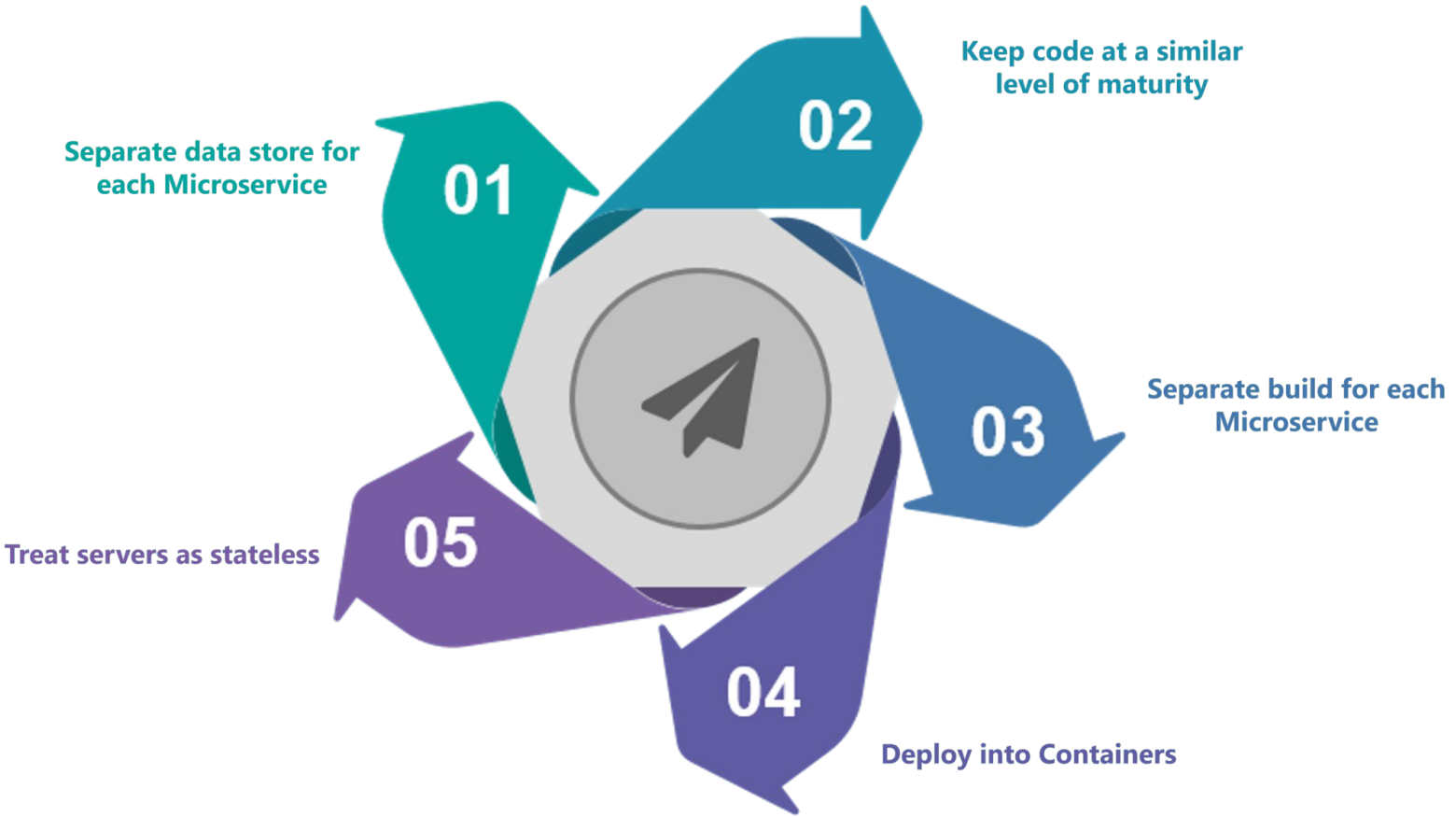
What do we get by using microservices ?

- **Decoupling** – Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
- **Componentization** – Microservices are treated as independent components that can be easily replaced and upgraded
- **Business Capabilities** – Microservices are very simple and focus on a single capability
- **Autonomy** – Developers and teams can work independently of each other, thus increasing speed
- **Continuous Delivery** – Allows frequent releases of software, through systematic automation of software creation, testing, and approval
- **Responsibility** – Microservices do not focus on applications as projects. Instead, they treat applications as products for which they are responsible
- **Decentralized Governance** – The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
- **Agility** – Microservices support agile development. Any new feature can be quickly developed and discarded again

Advantages

- **Independent Development** – All microservices can be easily developed based on their individual functionality
- **Independent Deployment** – Based on their services, they can be individually deployed in any application
- **Fault Isolation** – Even if one service of the application does not work, the system still continues to function
- **Mixed Technology Stack** – Different languages and technologies can be used to build different services of the same application
- **Granular Scaling** – Individual components can scale as per need, there is no need to scale all components together
- **Focused Team** - Focused team per service.
- **Replaceable -** Feasible to re-write and replace whole service.
- **Reusable** - If business is aligned to use that service across products. It's a win-win situation. For large enterprises its beneficial.

Best Practices



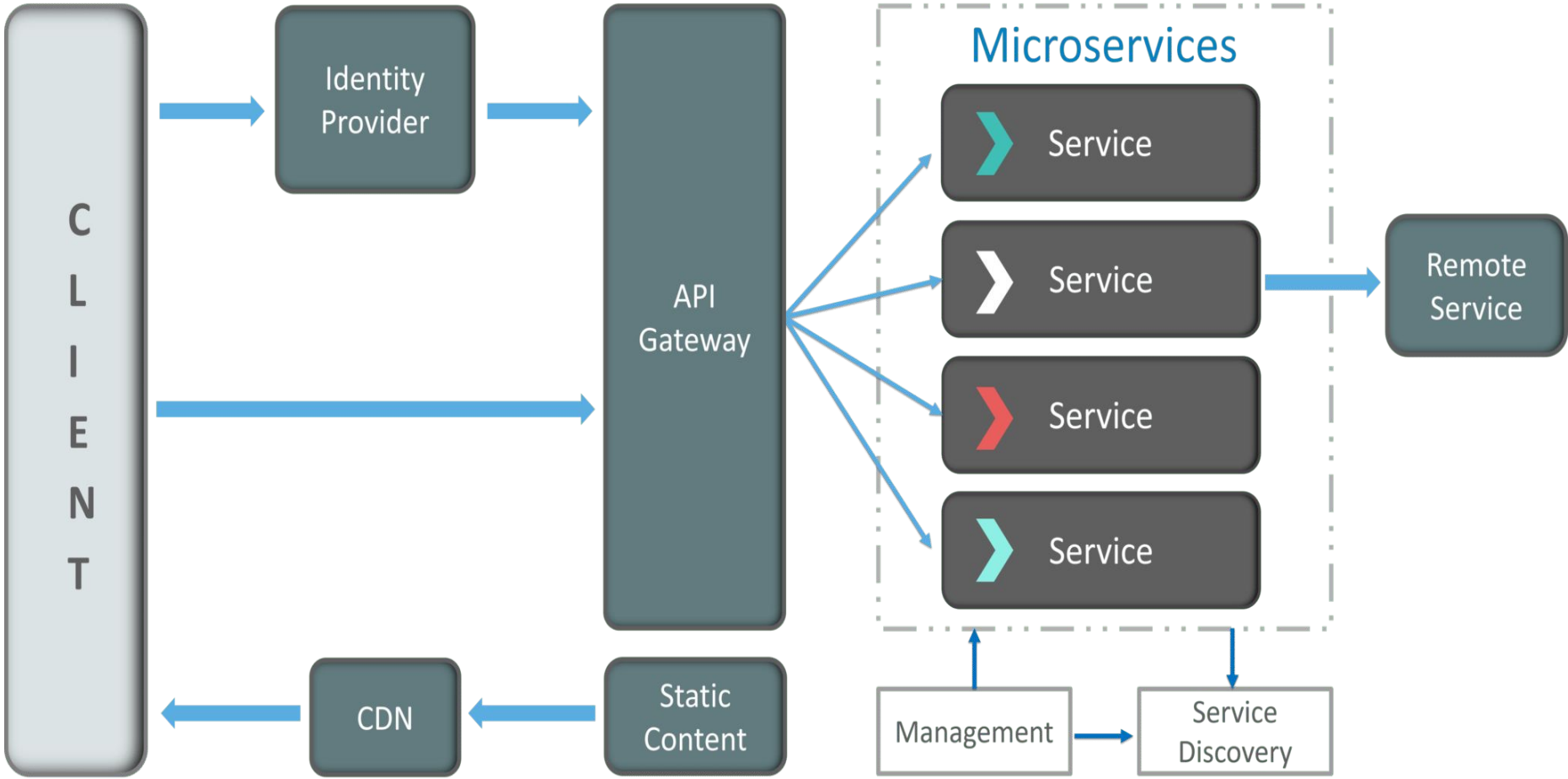
Best Practices

- By definition each microservice should be **independent of any other service**, can have its **own release & dev cycles.**
- Deploy API health monitoring and alters to identify the concerning point where multiple microservice within same product.
- Configure l**ess timeout for micro services** to avoid failing multiple services, and identify the **point of failure.**
- Maintain **versioning** for each microservice / API.
- Rightly decide on **deployment architecture** to manage requests from each service. Multiple microservices eventually creates traffic even no significant user traffic, as services interacting each other puts requests to server. This has been observed that internal services create more load than the actual user request.
- Prefer to have Single source of truth to every data. Generally it is suggested to have separate DB but this may increase chances of data inconsistencies. Hence take right decision after proper analysis in given situation.
- Implement **auto retires** wherever possible to mitigate failures, primarily in case of background jobs/queues. Or we can utilize **Hystrix**

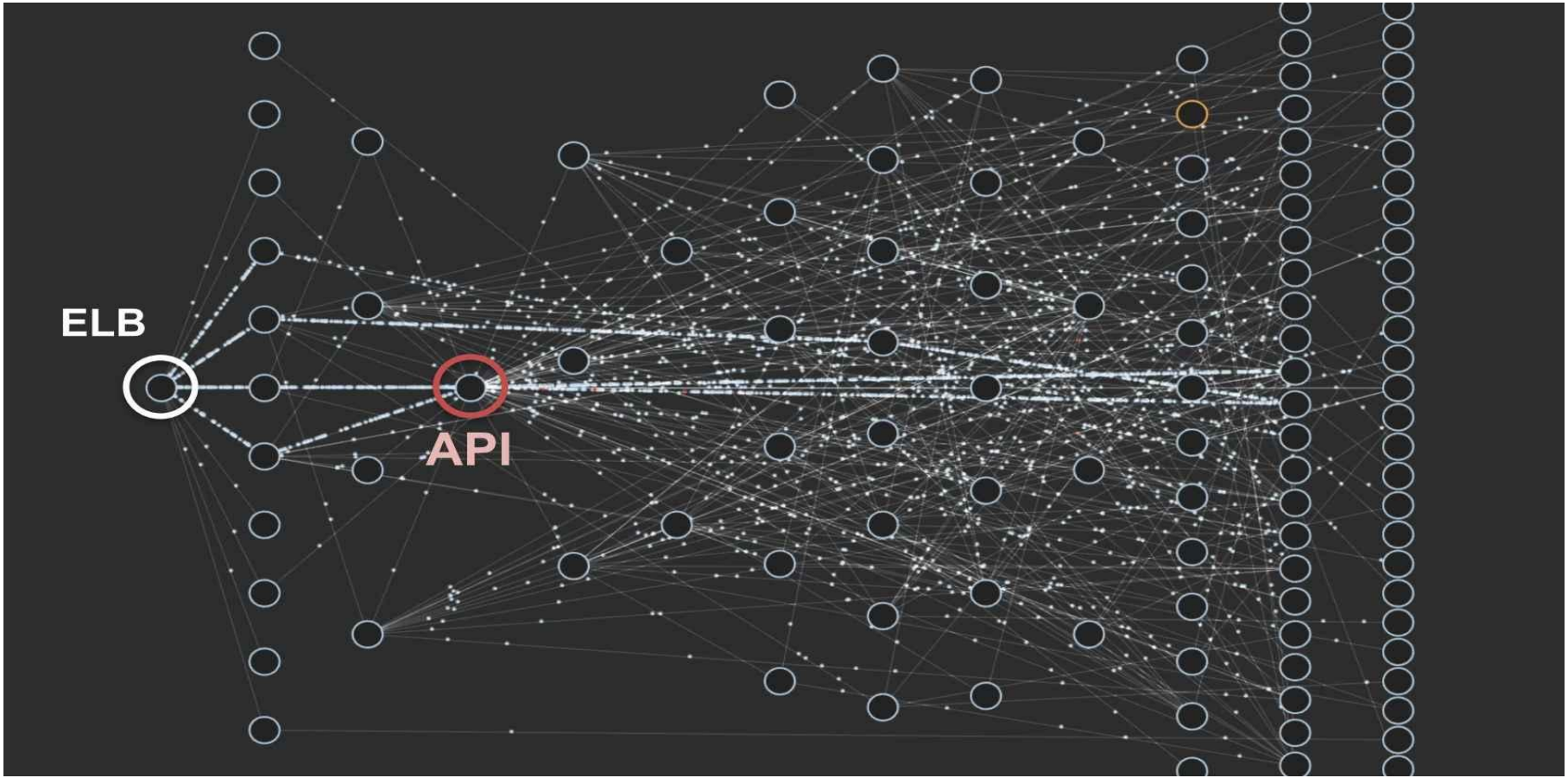
Components

A typical Microservice Architecture (MSA) should consist of the following components:

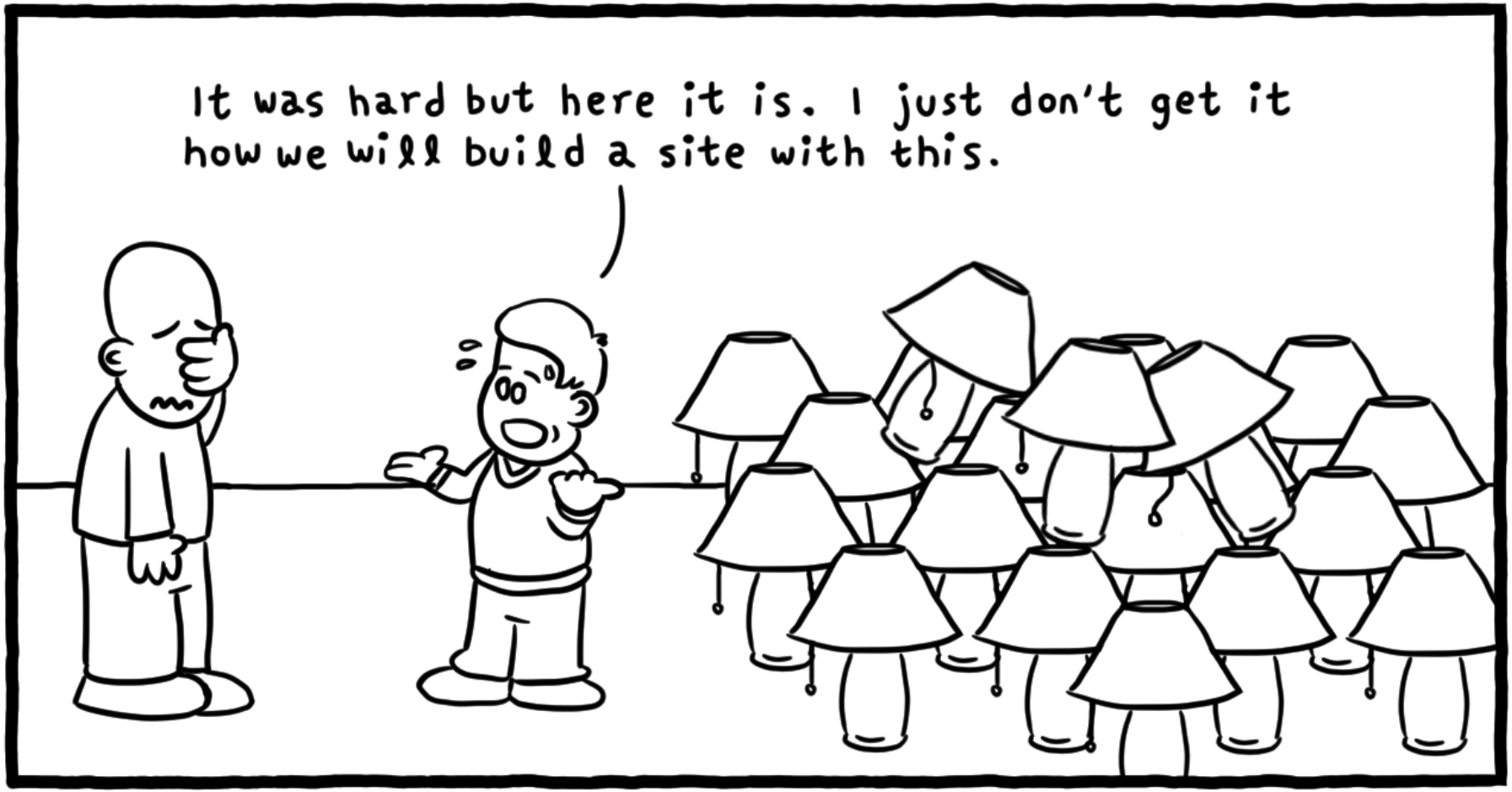
1. Clients
2. Identity Providers
3. API Gateway
4. Messaging Formats
5. Databases
6. Static Content
7. Management
8. Service Discovery
9. Monitoring
10. Fault tolerance techniques
11. Debugging & Tracing Errors


CLIENT
Identity Provider
API Gateway
Microservices
Service
Service
Service
Service
Remote Service
CDN
Static Content
Management
Service Discovery

Glimpse of NETFLIX API Mesh !



Cons Of Microservice Architecture	Pros Of Microservice Architecture
Increased complexity	Freedom to use different technologies
Each microservice focuses on a single business capability	Increased reliability due to isolation
Increased effort to maintain	Supports individual deployable units
Difficult to maintain transaction safety	Allows fine-grained security
Tough to track	Easier testing of each service
Multiple services need to be developed and deployed	Different services can be developed and deployed



Daniel Stori {turnoff.us}

DANIEL STORI {TURNOFF.US}

Thanks