ARAF KARSH HAMID
Co-Founder / CTO
MetaMagic Global Inc., NJ, USA

@arafkarsh
arafkarsh

$$\sum_{b=1}^{n} f(b) = \sqrt[3]{desi^3r^2e}\ 3D$$

1.21 GIGAWATTS!!

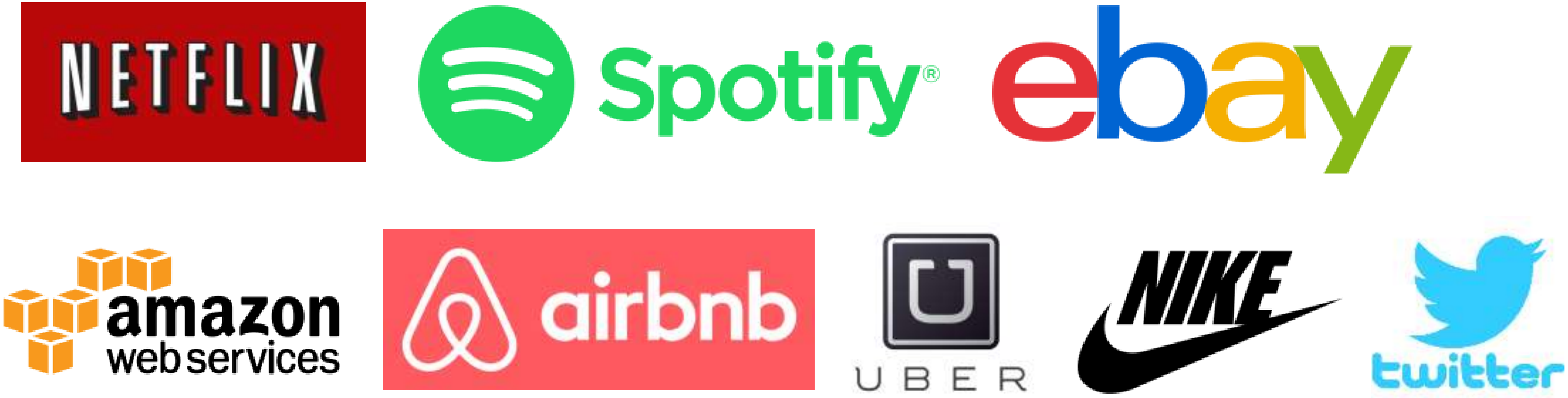
Microservices Architecture

World Agile Testing & Automation

Novotel Techpark, Bangalore, June 22, 2018

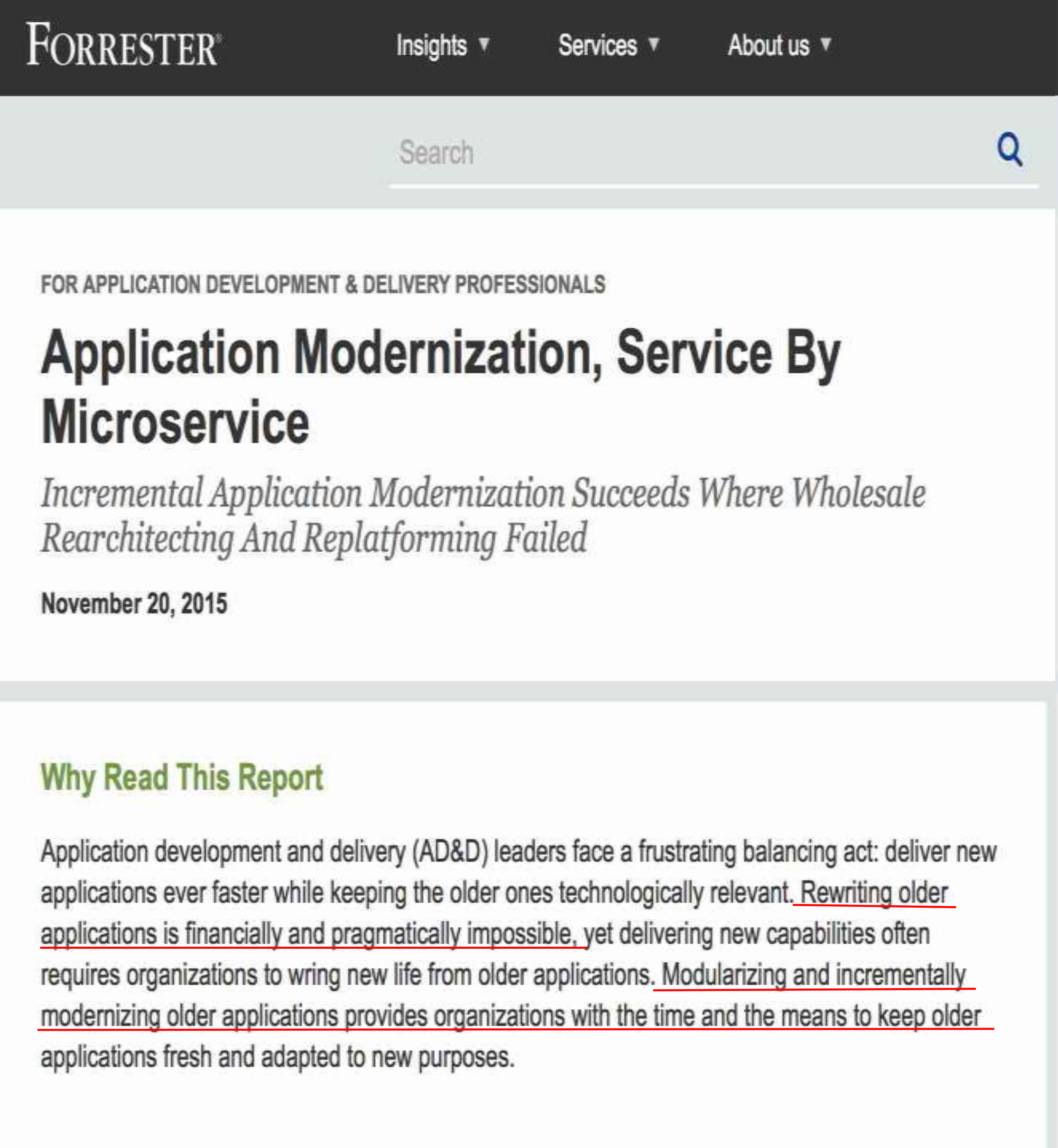
https://1point21gws.com/testingsummit/bangalore/

Pioneers in Microservices Implementation



New Entrants

Walmart
verizon



FOR APPLICATION DEVELOPMENT & DELIVERY PROFESSIONALS

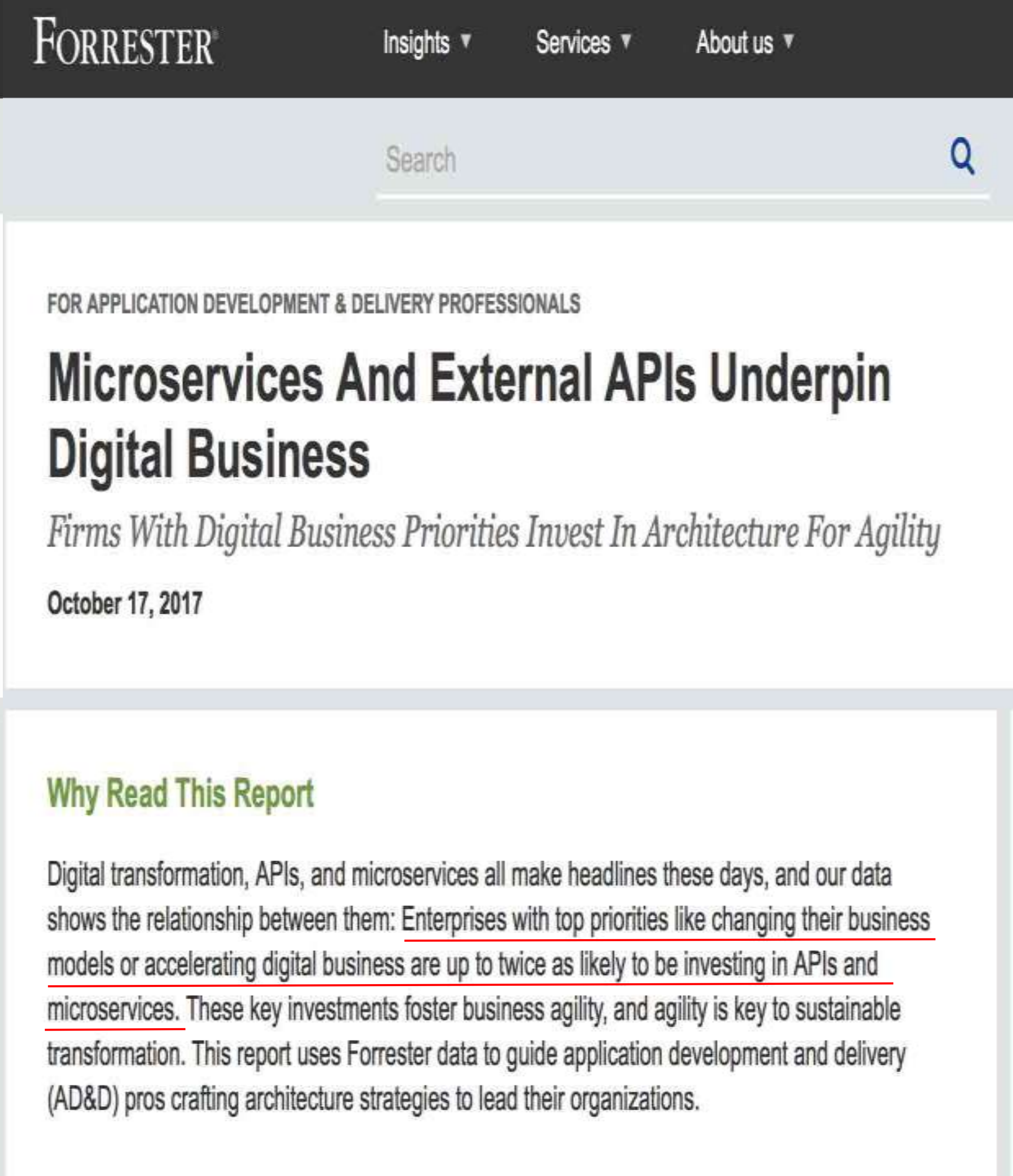
Application Modernization, Service By Microservice

Incremental Application Modernization Succeeds Where Wholesale Rearchitecting And Replatforming Failed

November 20, 2015

Why Read This Report

Application development and delivery (AD&D) leaders face a frustrating balancing act: deliver new applications ever faster while keeping the older ones technologically relevant. Rewriting older applications is financially and pragmatically impossible, yet delivering new capabilities often requires organizations to wring new life from older applications. Modularizing and incrementally modernizing older applications provides organizations with the time and the means to keep older applications fresh and adapted to new purposes.



FOR APPLICATION DEVELOPMENT & DELIVERY PROFESSIONALS

Microservices And External APIs Underpin Digital Business

Firms With Digital Business Priorities Invest In Architecture For Agility

October 17, 2017

Why Read This Report

Digital transformation, APIs, and microservices all make headlines these days, and our data shows the relationship between them: Enterprises with top priorities like changing their business models or accelerating digital business are up to twice as likely to be investing in APIs and microservices. These key investments foster business agility, and agility is key to sustainable transformation. This report uses Forrester data to guide application development and delivery (AD&D) pros crafting architecture strategies to lead their organizations.

Agenda

1

Architecture Styles

- Pros and Cons
- Micro Services Characteristics
- Monolithic Vs. Micro Services Architecture
- SOA Vs. Micro Services Architecture
- App Scalability Based on Micro Services
- Hexagonal Architecture

2

Design Styles

- Design Patterns
- Domain Driven Design
- Event Sourcing & CQRS
- Functional Reactive Programming

3

Scalability

- CAP Theorem
- Distributed Transactions : 2 Phase Commit
- SAGA Design Pattern
- Scalability Lessons from EBay
- Design Patterns

4

Microservices Testing

- Testing Strategy
- Category of Testing
- BDD Example
- BDD Features
- References

Pros and Cons

Pros

1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

Cons

1. Adds Complexity
2. Skillset shortage
3. Confusion on getting the right size
4. Team need to manage end-to-end of the Service (From UI to Backend to Running in Production).

Micro Services Characteristics

By James Lewis and Martin Fowler

Components via Services

Organized around **Business Capabilities**

Products NOT Projects

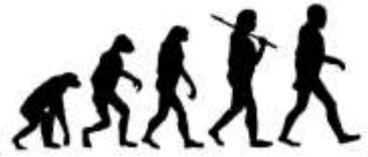
Smart Endpoints & Dumb Pipes

Decentralized Governance & Data Management

Infrastructure **Automation**

Design for **Failure**

Evolutionary Design



The key in making great and growable systems is much more to design how its modules communicate rather than what their internal properties and behaviors should be.
Alan Kay, 1998 email to the Squeak-dev list

Modularity ... is to a technological economy what the division of labor is to a manufacturing one.
W. Brian Arthur,
author of *e Nature of Technology*

We can scale our operation independently, maintain unparalleled system availability, and introduce new services quickly without the need for massive reconfiguration. —
Werner Vogels, CTO, Amazon Web Services

Micro Services System Design Model

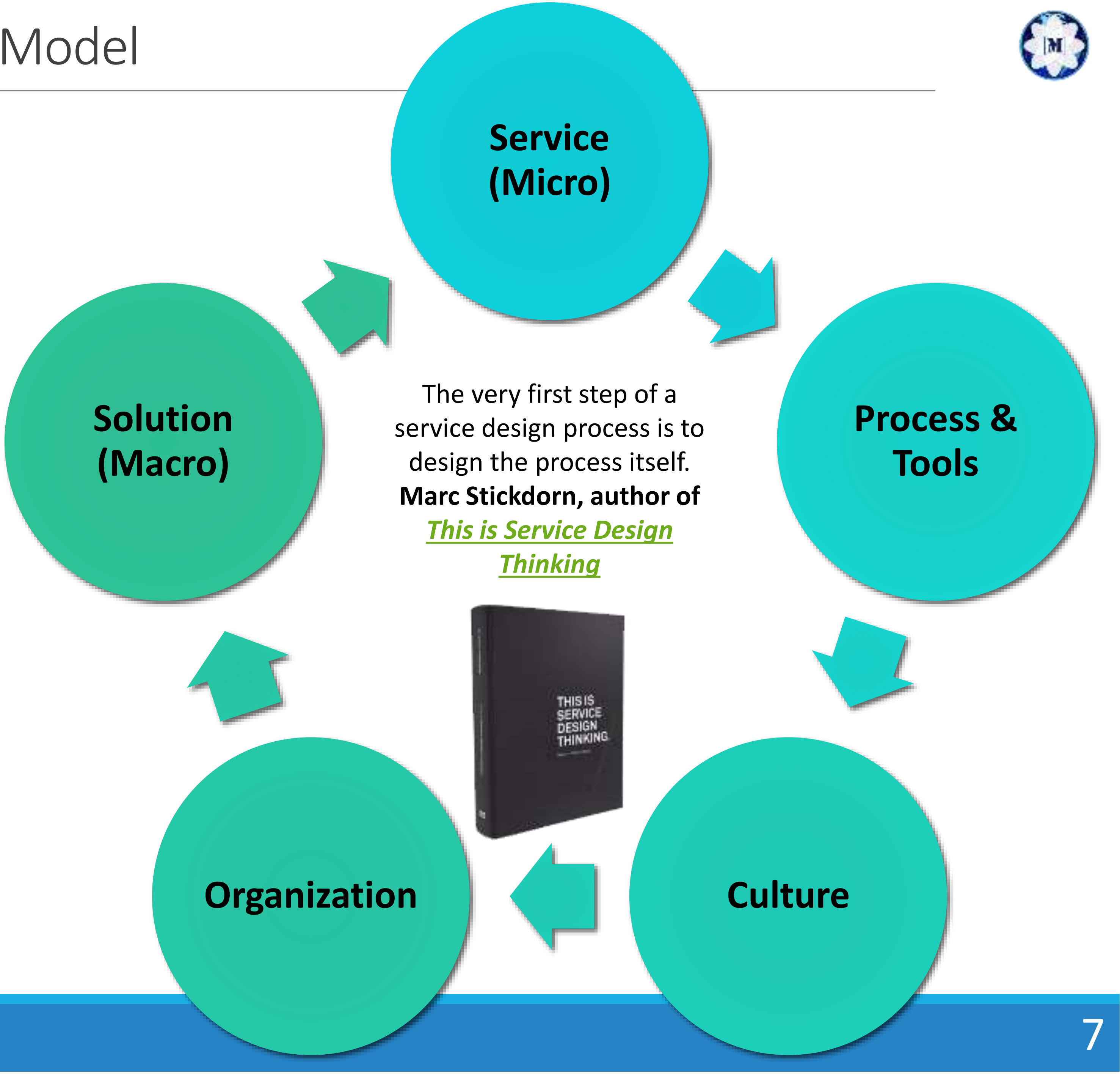
Service: Focuses on **a specific Business Capability**

Process & Tools: Development, Code Deployment, Maintenance and Product Management

Culture: A Shared set of values, beliefs by everyone. **Ubiquitous Language in DDD is an important aspect of Culture.**

Organization: Structure, Direction of Authority, Granularity, Composition of Teams.

Solution: Coordinate all inputs and outputs of multiple services. Macro level view of the system allows the designer to focus more on desirable system behavior.



The very first step of a service design process is to design the process itself.
Marc Stickdorn, author of
This is Service Design Thinking

Microservices Architecture

Infrastructure Architecture

- API Gateway, Service Discovery
- Event Bus / Streams
- Service Mesh

Software Design

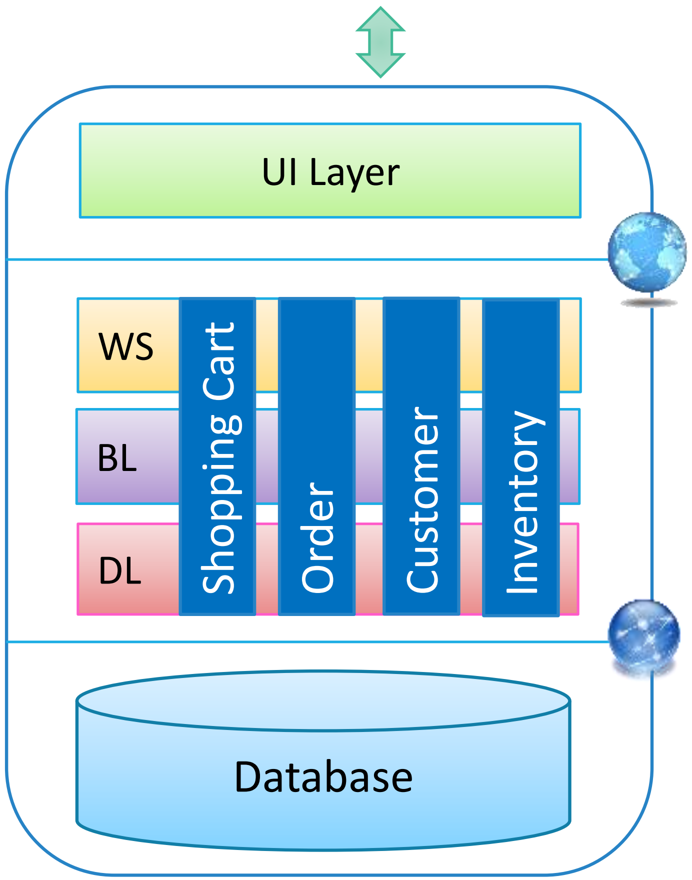
- Domain Driven Design
- Event Sourcing & CQRS
- Functional Reactive Programming

Monolithic vs. Micro Services Example

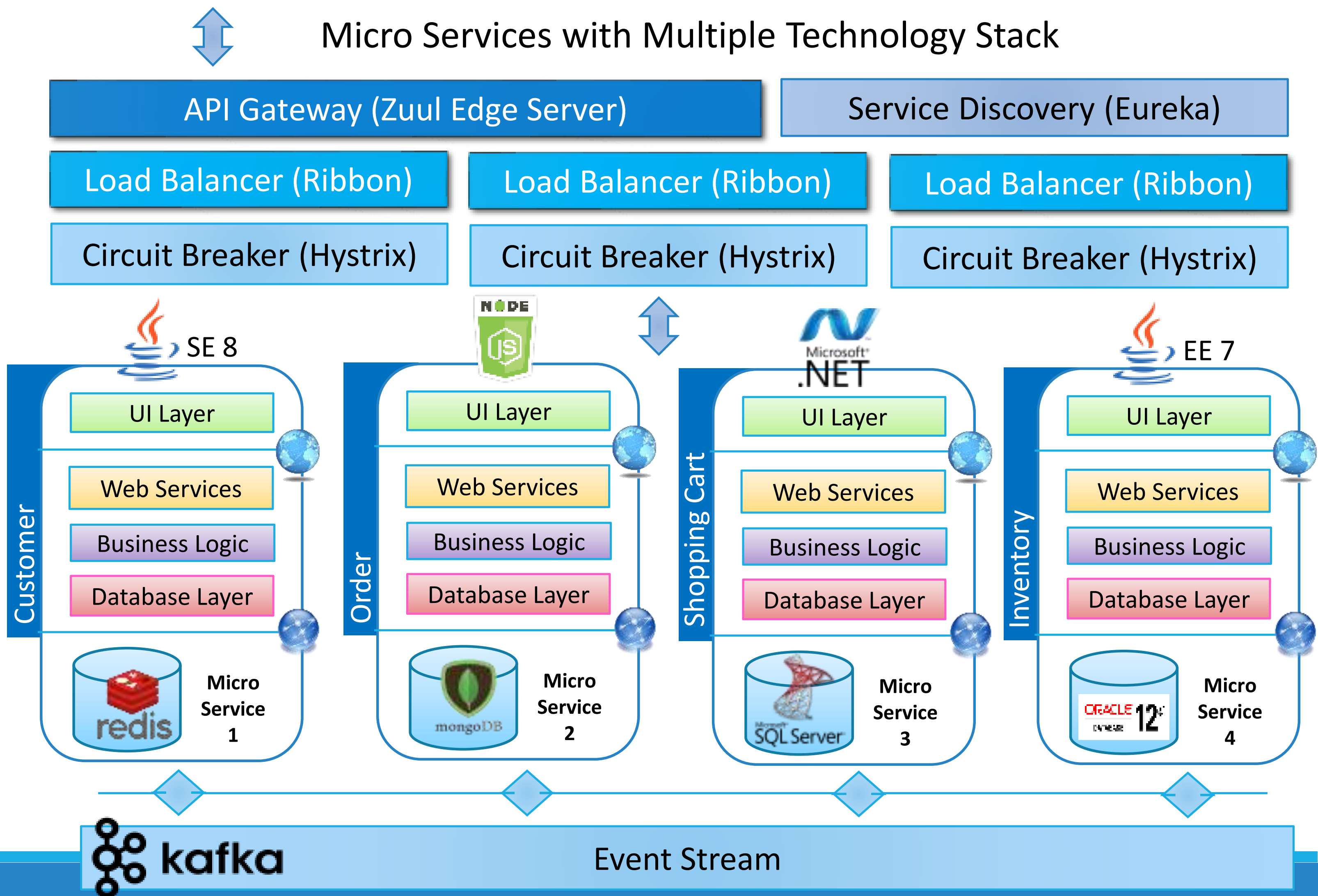
Existing aPaaS vendors creates Monolithic Apps.

This 3 tier model is obsolete now.

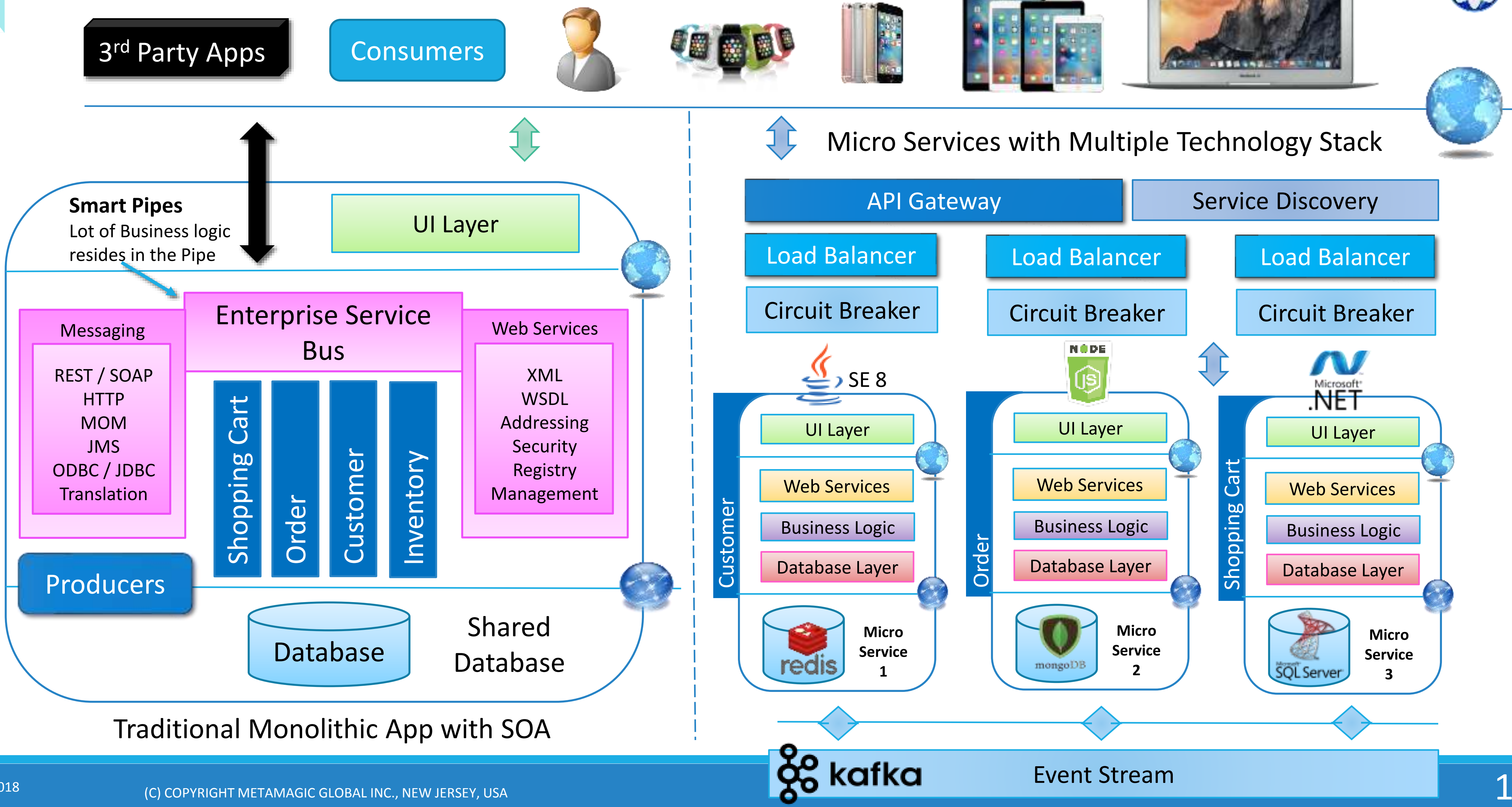
Source: Gartner Market Guide for Application Platforms Nov 23, 2016



Traditional Monolithic App using Single Technology Stack



SOA vs. Micro Services Example



3rd Party Apps

Consumers

Smart Pipes
Lot of Business logic
resides in the Pipe

UI Layer

Messaging

REST / SOAP
HTTP
MOM
JMS
ODBC / JDBC
Translation

Enterprise Service Bus

Shopping Cart

Order

Customer

Inventory

Web Services

XML
WSDL
Addressing
Security
Registry
Management

Producers

Database

Shared Database

Traditional Monolithic App with SOA

Micro Services with Multiple Technology Stack

API Gateway

Service Discovery

Load Balancer

Circuit Breaker

Load Balancer

Circuit Breaker

Load Balancer

Circuit Breaker

SE 8

Customer

UI Layer

Web Services

Business Logic

Database Layer

redis

Micro Service 1

Order

UI Layer

Web Services

Business Logic

Database Layer

mongoDB

Micro Service 2

Shopping Cart

UI Layer

Web Services

Business Logic

Database Layer

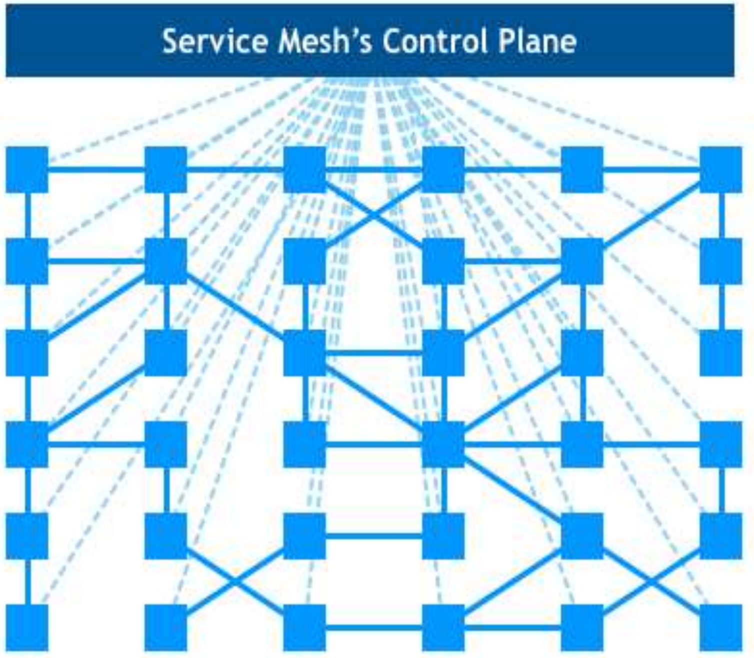
SQL Server

Micro Service 3

kafka

Event Stream

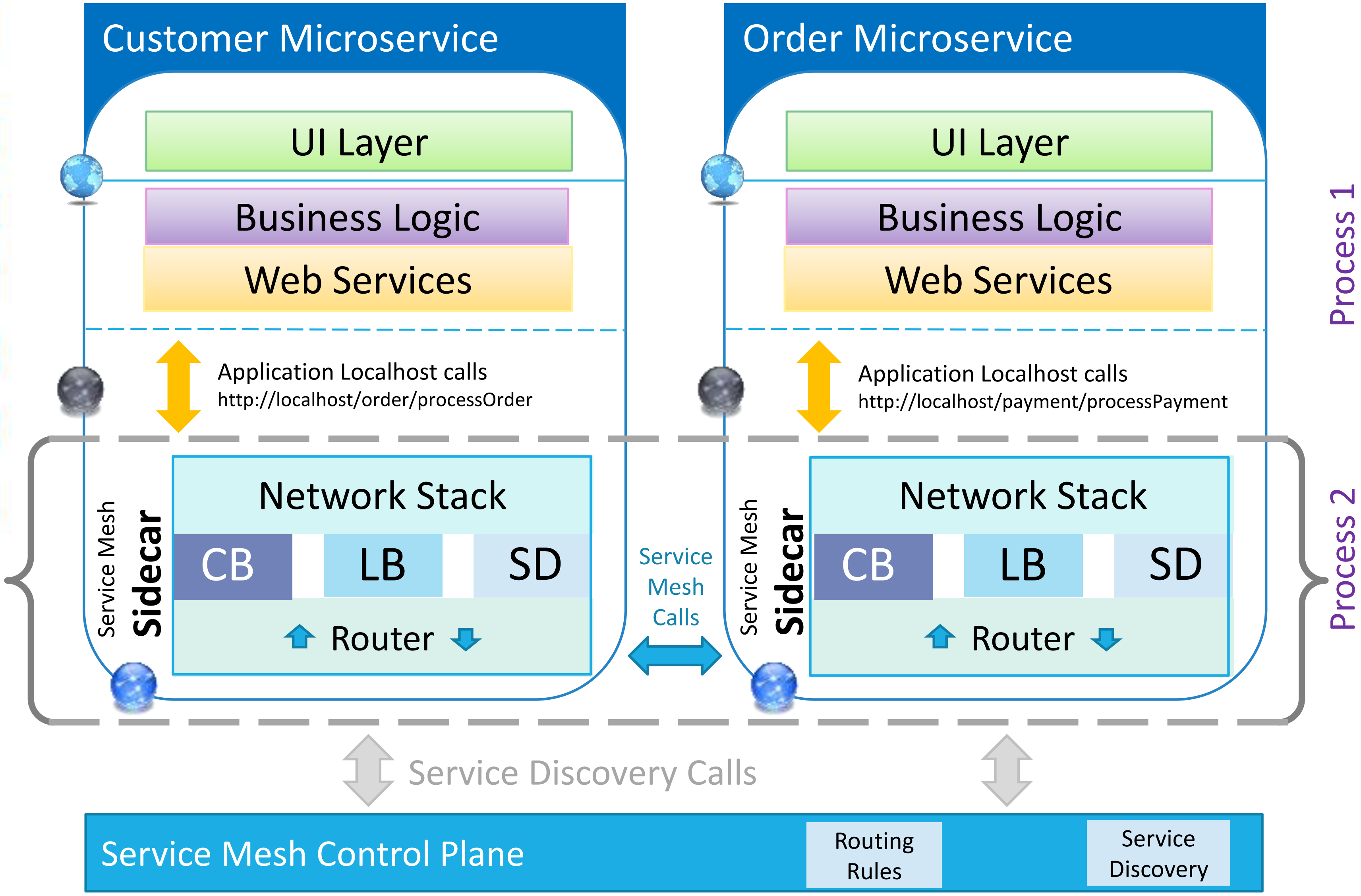
Service Mesh – Sidecar Design Pattern



🟦 Microservice

CB – Circuit Breaker
LB – Load Balancer
SD – Service Discovery

Data Plane



Customer Microservice

UI Layer

Business Logic

Web Services

Application Localhost calls
http://localhost/order/processOrder

Service Mesh **Sidecar**

Network Stack

CB | LB | SD

Router

Service Mesh Calls

Order Microservice

UI Layer

Business Logic

Web Services

Application Localhost calls
http://localhost/payment/processPayment

Service Mesh **Sidecar**

Network Stack

CB | LB | SD

Router

Process 1

Process 2

Service Discovery Calls

Service Mesh Control Plane

Routing Rules

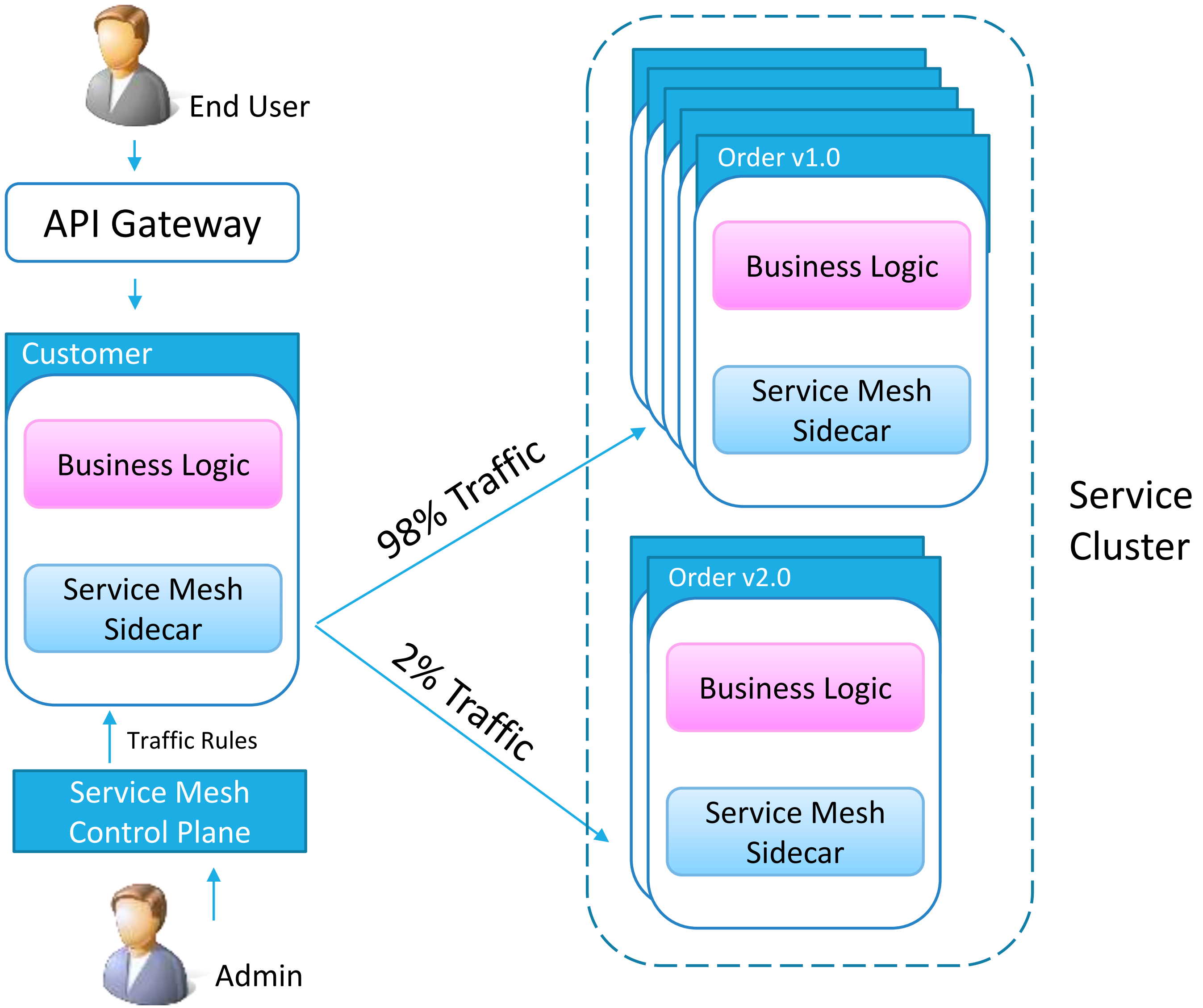
Service Discovery

Control Plane will have all the rules for Routing and Service Discovery. Local Service Mesh will download the rules from the Control pane will have a local copy.

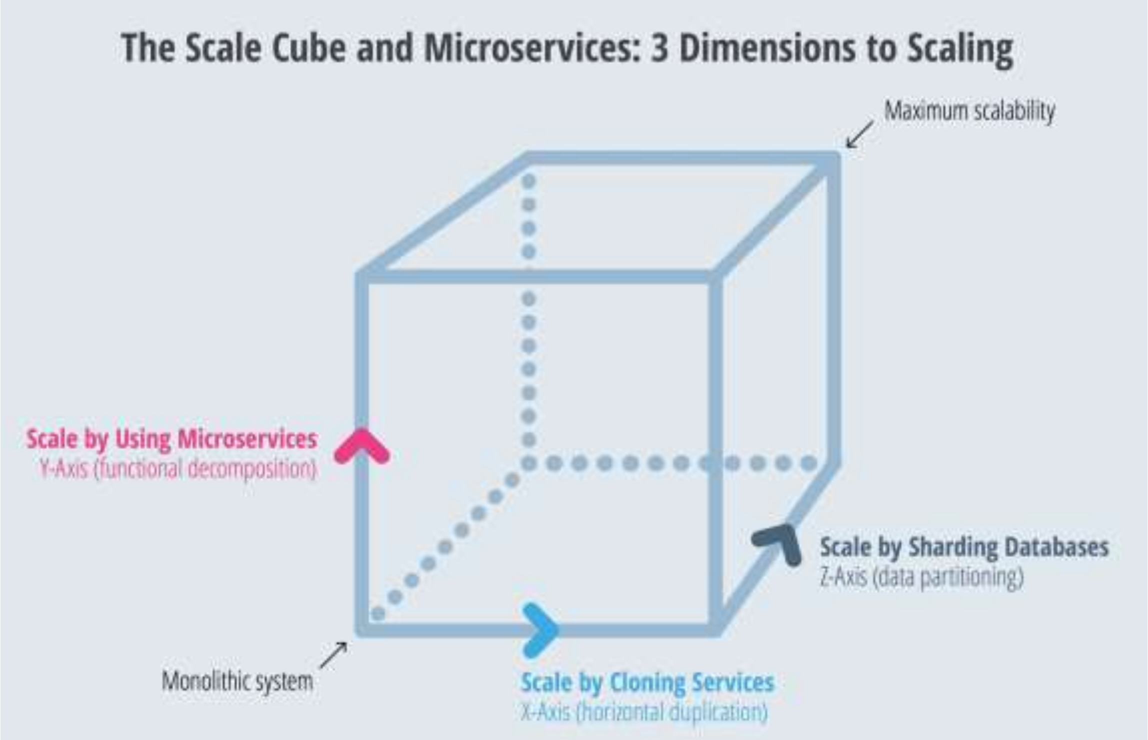
Service Mesh – Traffic Control

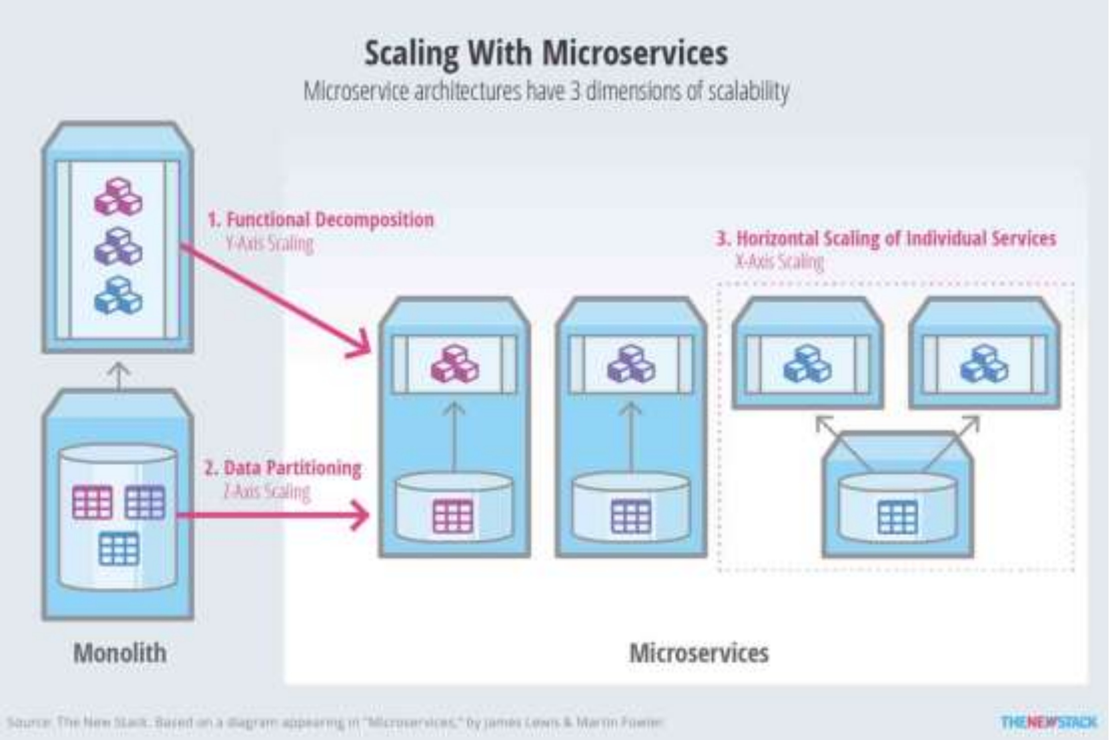
Traffic Control rules can be applied for

- different Microservices versions
- Re Routing the request to debugging system to analyze the problem in real time.
- Smooth migration path



Scale Cube and Micro Services


The Scale Cube and Microservices: 3 Dimensions to Scaling
Maximum scalability
Scale by Using Microservices
Y-Axis (functional decomposition)
Scale by Sharding Databases
Z-Axis (data partitioning)
Monolithic system
Scale by Cloning Services
X-Axis (horizontal duplication)


Scaling With Microservices
Microservice architectures have 3 dimensions of scalability
1. Functional Decomposition
Y-Axis Scaling
3. Horizontal Scaling of Individual Services
X-Axis Scaling
2. Data Partitioning
Z-Axis Scaling
Monolith
Microservices
THENEWSTACK

Source: The New Stack. Based on a diagram appearing in "Microservices," by James Lewis & Martin Fowler.

1. Y Axis Scaling – Functional Decomposition : Business Function as a Service
2. Z Axis Scaling – Database Partitioning : Avoid locks by Database Sharding
3. X Axis Scaling – Cloning of Individual Services for Specific Service Scalability

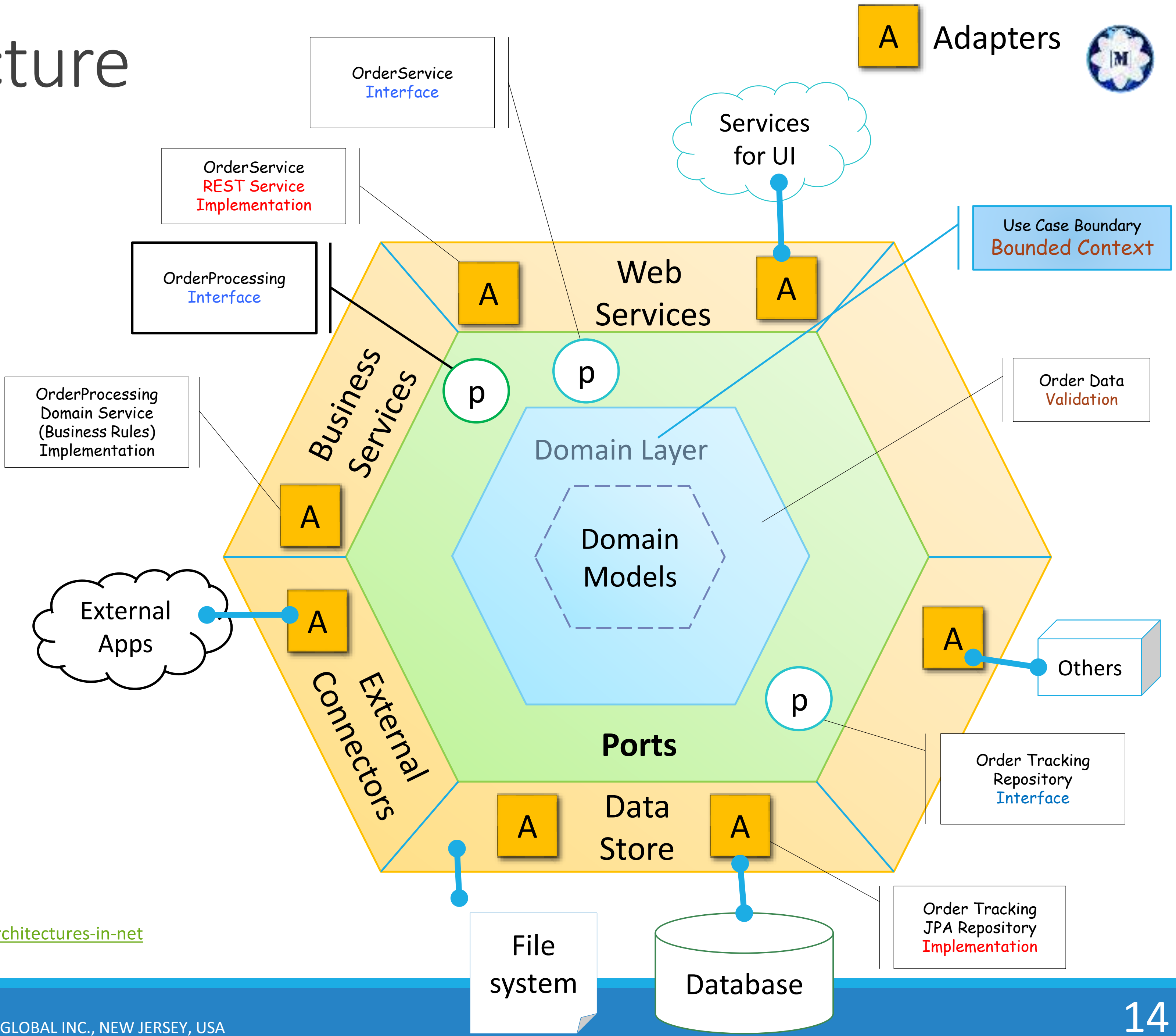
Hexagonal Architecture

Ports & Adapters

The layer between the **Adapter** and the **Domain** is identified as the **Ports** layer. The Domain is inside the port, adapters for external entities are on the outside of the port.

The notion of a “port” invokes the OS idea that any device that adheres to a known protocol can be plugged into a port. Similarly many adapters may use the Ports.

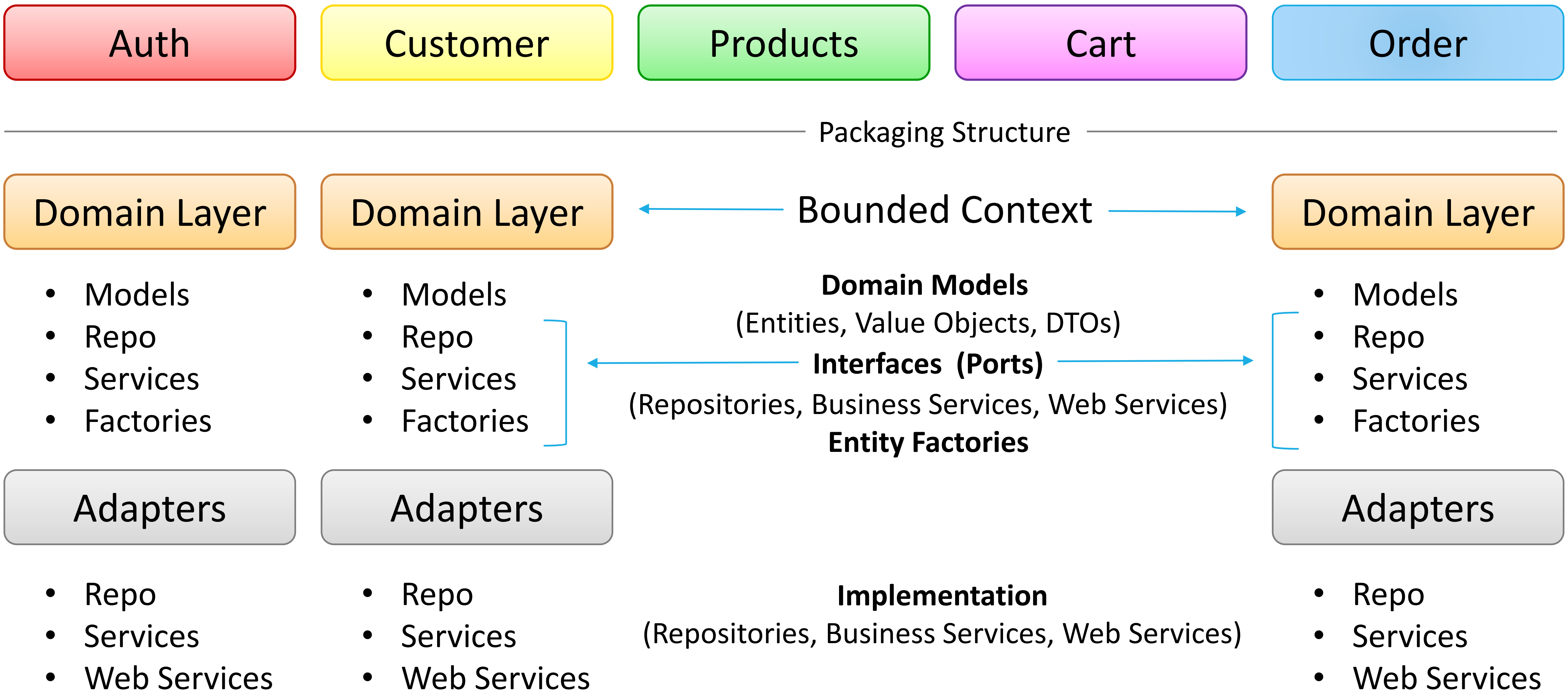
- Reduces Technical Debt
- Dependency Injection
- Auto Wiring



A Adapters
OrderService Interface
OrderService REST Service Implementation
OrderProcessing Interface
OrderProcessing Domain Service (Business Rules) Implementation
Services for UI
Use Case Boundary Bounded Context
Web Services
A
A
A
A
A
A
A
A
p
p
p
Business Services
Domain Layer
Domain Models
Order Data Validation
External Apps
External Connectors
Ports
Others
Order Tracking Repository Interface
Data Store
Order Tracking JPA Repository Implementation
File system
Database

Source : http://alistair.cockburn.us/Hexagonal+architecture
https://skillsmatter.com/skillscasts/5744-decoupling-from-asp-net-hexagonal-architectures-in-net

Shopping Portal Modules – Code Packaging



Auth

Customer

Products

Cart

Order

Packaging Structure

Domain Layer

- Models
- Repo
- Services
- Factories

Adapters

- Repo
- Services
- Web Services

Domain Layer

- Models
- Repo
- Services
- Factories

Adapters

- Repo
- Services
- Web Services

Bounded Context

Domain Models
(Entities, Value Objects, DTOs)
Interfaces (Ports)
(Repositories, Business Services, Web Services)
Entity Factories

Implementation
(Repositories, Business Services, Web Services)

Domain Layer

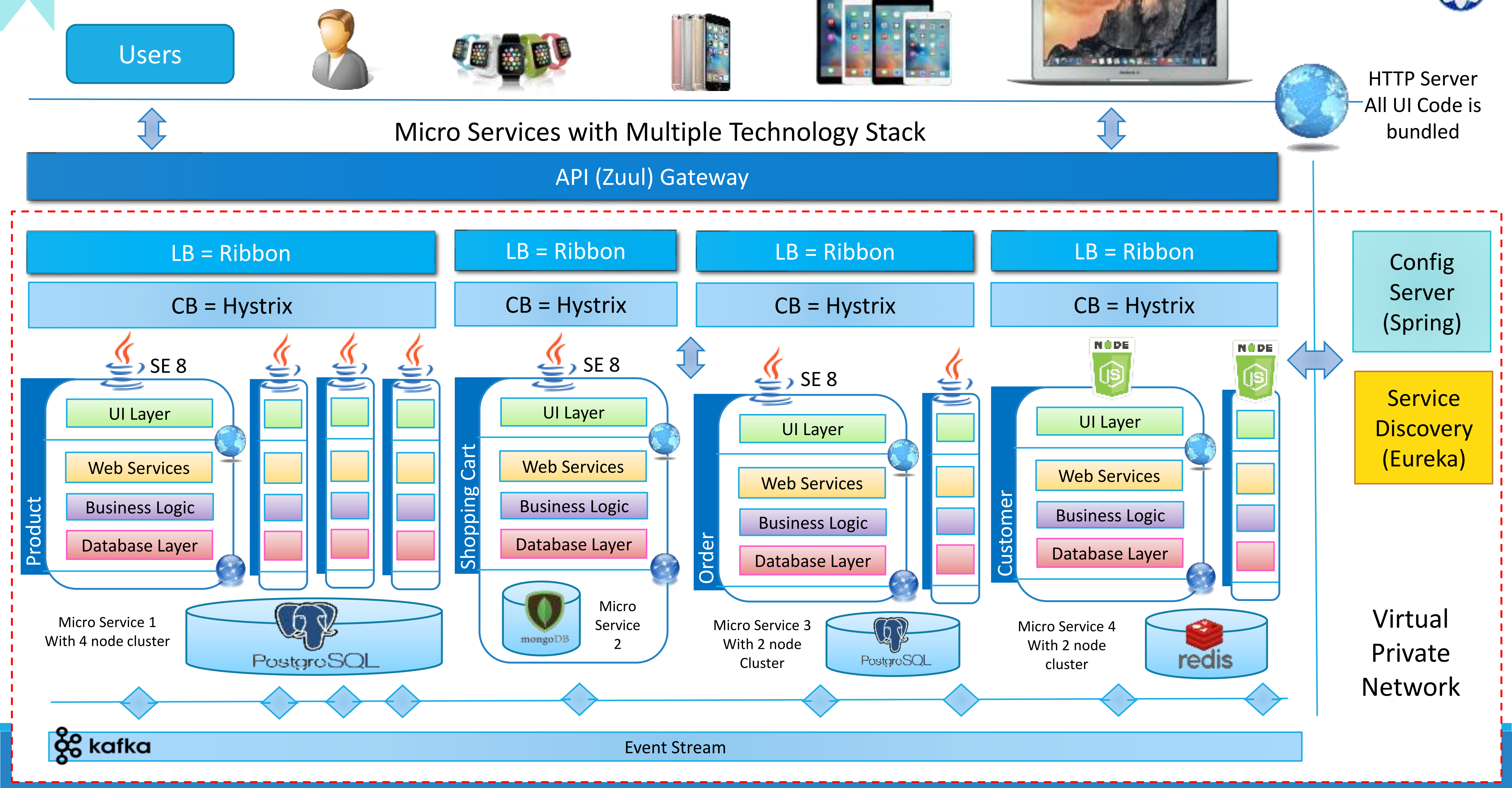
- Models
- Repo
- Services
- Factories

Adapters

- Repo
- Services
- Web Services

1

Micro Services Deployment Model



Users

HTTP Server
All UI Code is
bundled

Micro Services with Multiple Technology Stack

API (Zuul) Gateway

LB = Ribbon

CB = Hystrix

SE 8

Product

UI Layer

Web Services

Business Logic

Database Layer

Micro Service 1
With 4 node cluster

PostgreSQL

LB = Ribbon

CB = Hystrix

SE 8

Shopping Cart

UI Layer

Web Services

Business Logic

Database Layer

mongoDB

Micro
Service
2

LB = Ribbon

CB = Hystrix

SE 8

Order

UI Layer

Web Services

Business Logic

Database Layer

Micro Service 3
With 2 node
Cluster

PostgreSQL

LB = Ribbon

CB = Hystrix

NODE

Customer

UI Layer

Web Services

Business Logic

Database Layer

Micro Service 4
With 2 node
cluster

redis

Config
Server
(Spring)

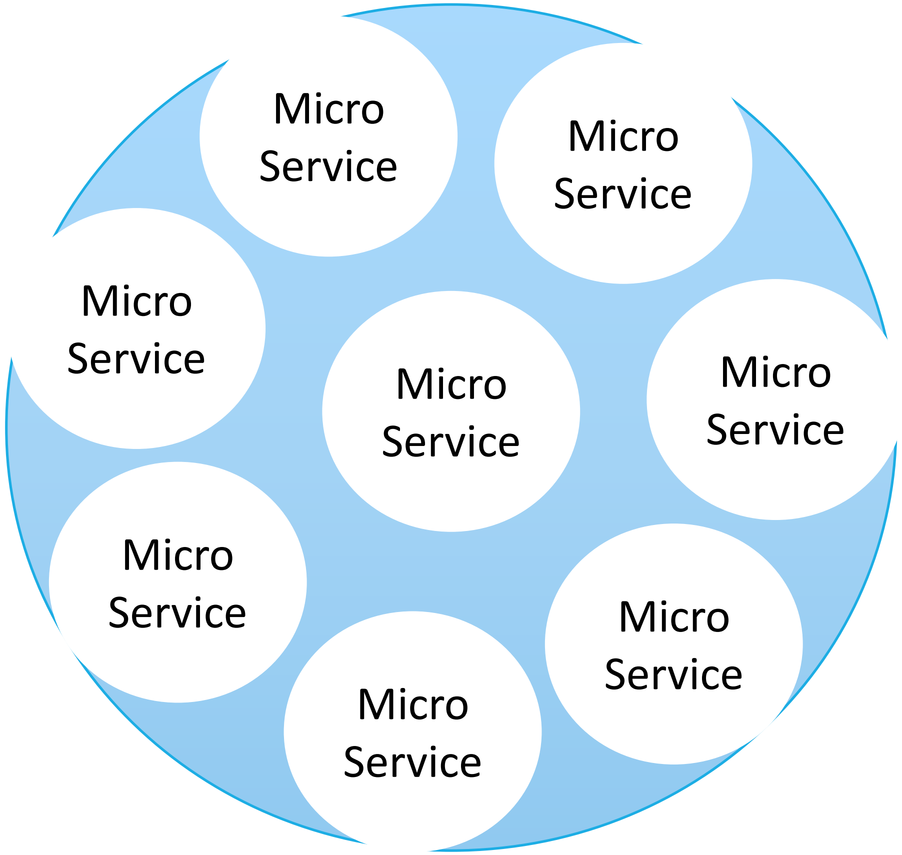
Service
Discovery
(Eureka)

Virtual
Private
Network

kafka

Event Stream

Summary – Micro Services Intro



Key Features

1. Small in size
2. Messaging–enabled
3. Bounded by contexts
4. Autonomously developed
5. Independently deployable
6. Decentralized
7. Language–agnostic
8. Built and released with automated processes

Benefits

1. Robust
2. Scalable
3. Testable (Local)
4. Easy to Change and Replace
5. Easy to Deploy
6. Technology Agnostic

Martin Fowler – Micro Services Architecture
https://martinfowler.com/articles/microservices.html
Dzone – SOA vs Micro Services : https://dzone.com/articles/microservices-vs-soa-2

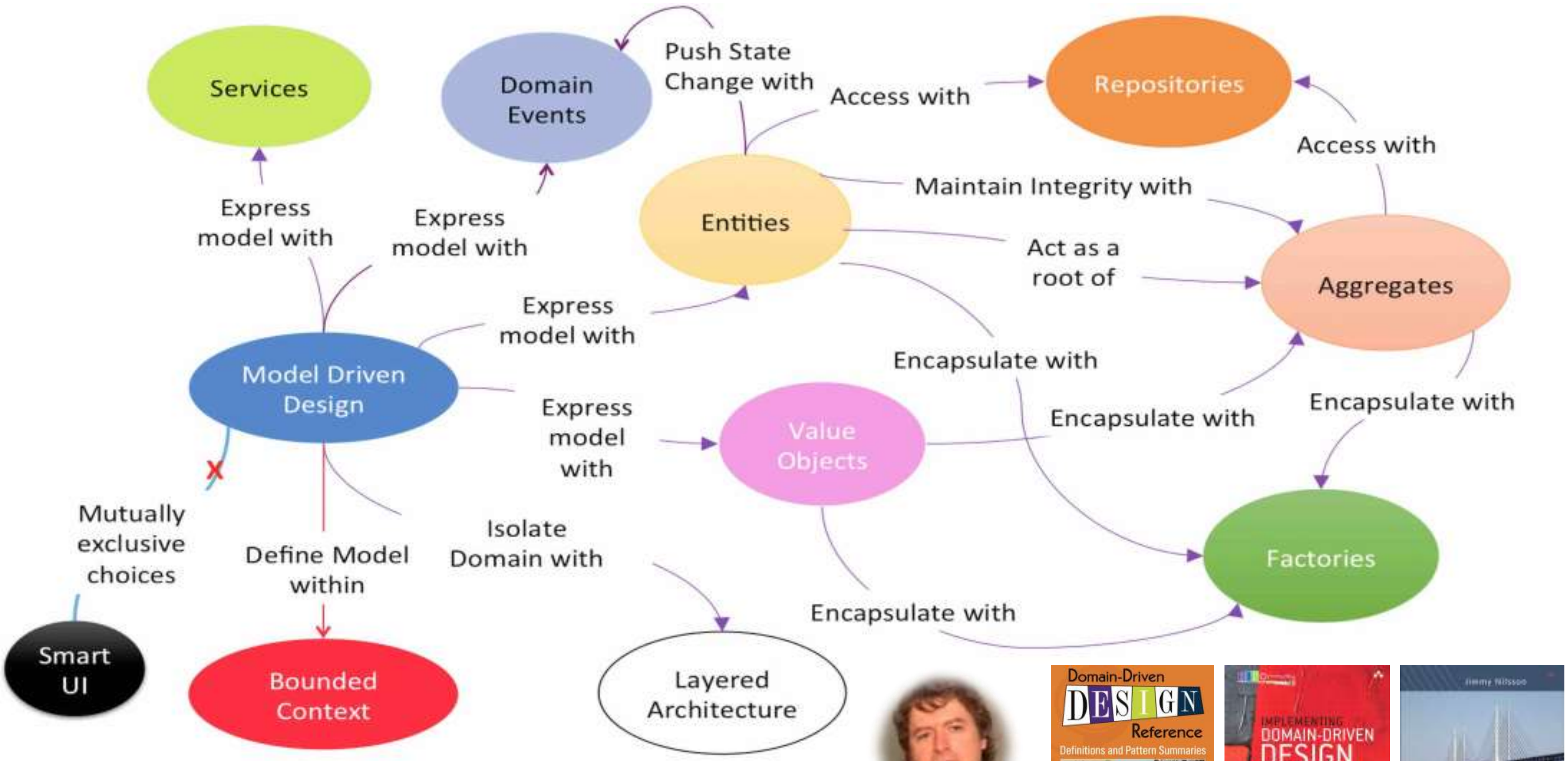
2

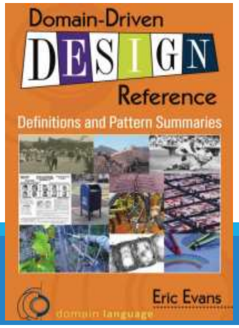
Design Styles

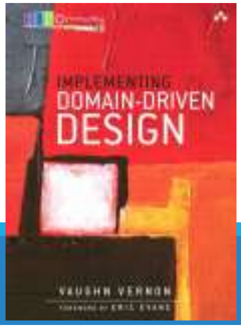
- Domain Driven Design
  - Understanding Requirement Analysis
  - Bounded Context
  - Context Map
  - Aggregate Root
- Event Sourcing & CQRS
  - CRUD
  - ES and CQRS
  - Event Sourcing Example
- Functional Reactive Programming
  - 4 Building Blocks of RxJava
  - Observable and Observer Design Pattern
  - Comparison : Iterable / Streams / Observable
  - Design Patterns : Let it Crash / SAGA

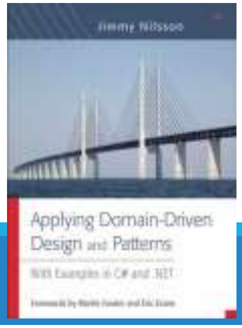
It's not necessary that you need to use all these patterns. You will be using these based on your technical requirement

Domain Driven Design




Domain-Driven DESIGN Reference
Definitions and Pattern Summaries
Eric Evans
domain language


IMPLEMENTING DOMAIN-DRIVEN DESIGN
VAUGHN VERNON


Jimmy Nilsson
Applying Domain-Driven Design and Patterns
With Examples in C# and .NET

Source: Domain-Driven Design Reference by Eric Evans

Ubiquitous Language : Understanding Requirement Analysis using DDD

Ubiquitous Language

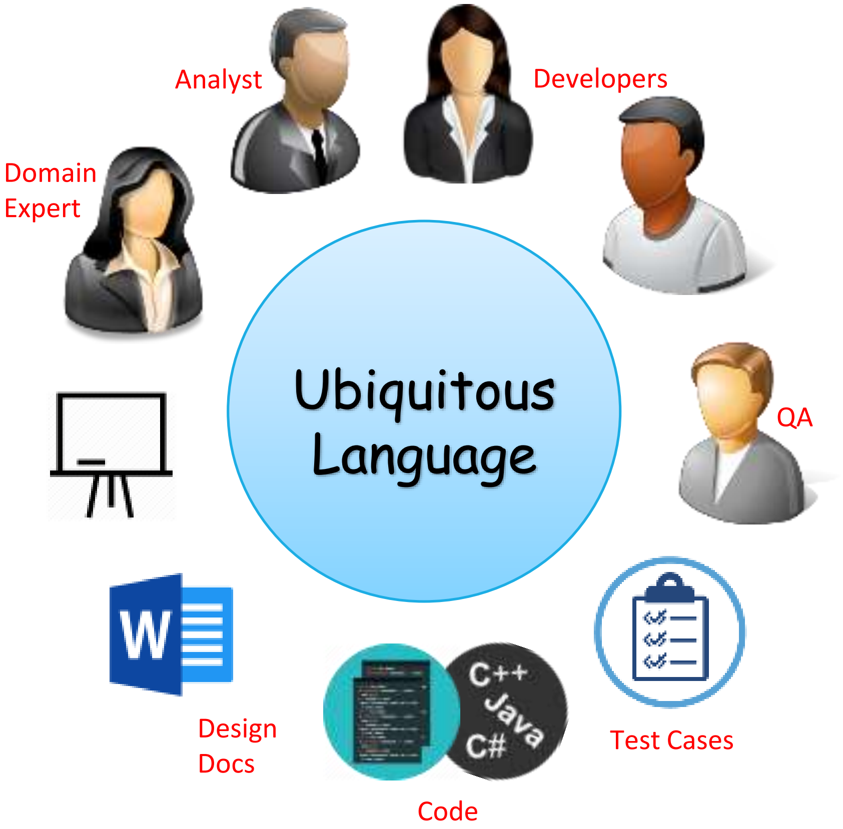
Vocabulary shared by all involved parties

Used in all forms of spoken / written communication

Restaurant Context – Food Item :

Eg. Food Item (Navrathnakurma) can have different meaning or properties depends on the context.

- In the Menu Context it's a Veg Dish.
- In the Kitchen Context it's is recipe.
- And in the Dining Context it will have more info related to user feed back etc.



Ubiquitous Language using BDD

As an insurance Broker
I want to know who my Gold Customers are
So that I sell more

Given	Customer John Doe exists
When	he buys insurance ABC for $1000 USD
Then	He becomes a Gold Customer

BDD – Behavior Driven Development

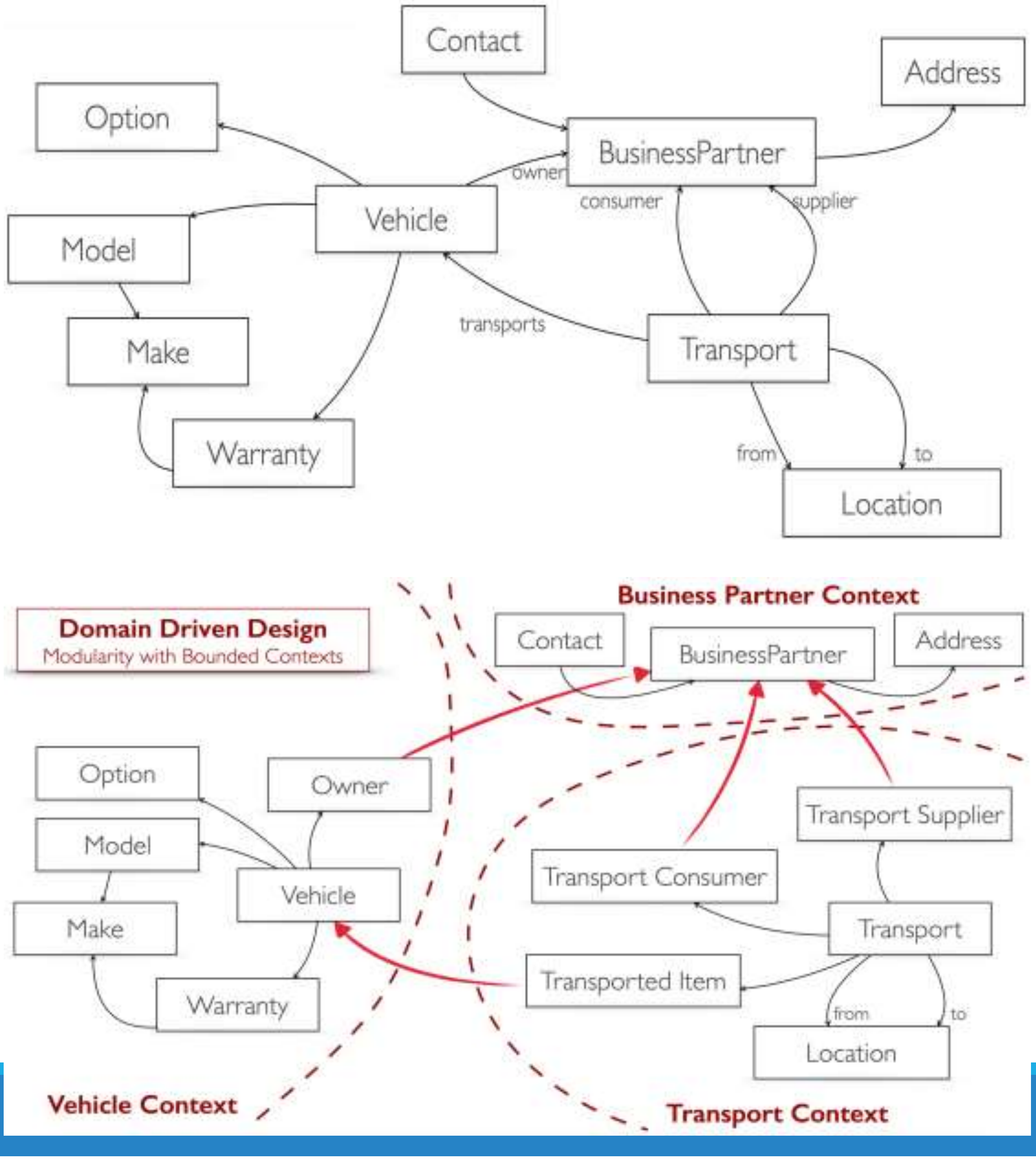
2

Understanding Requirement Analysis using DDD

Bounded Context

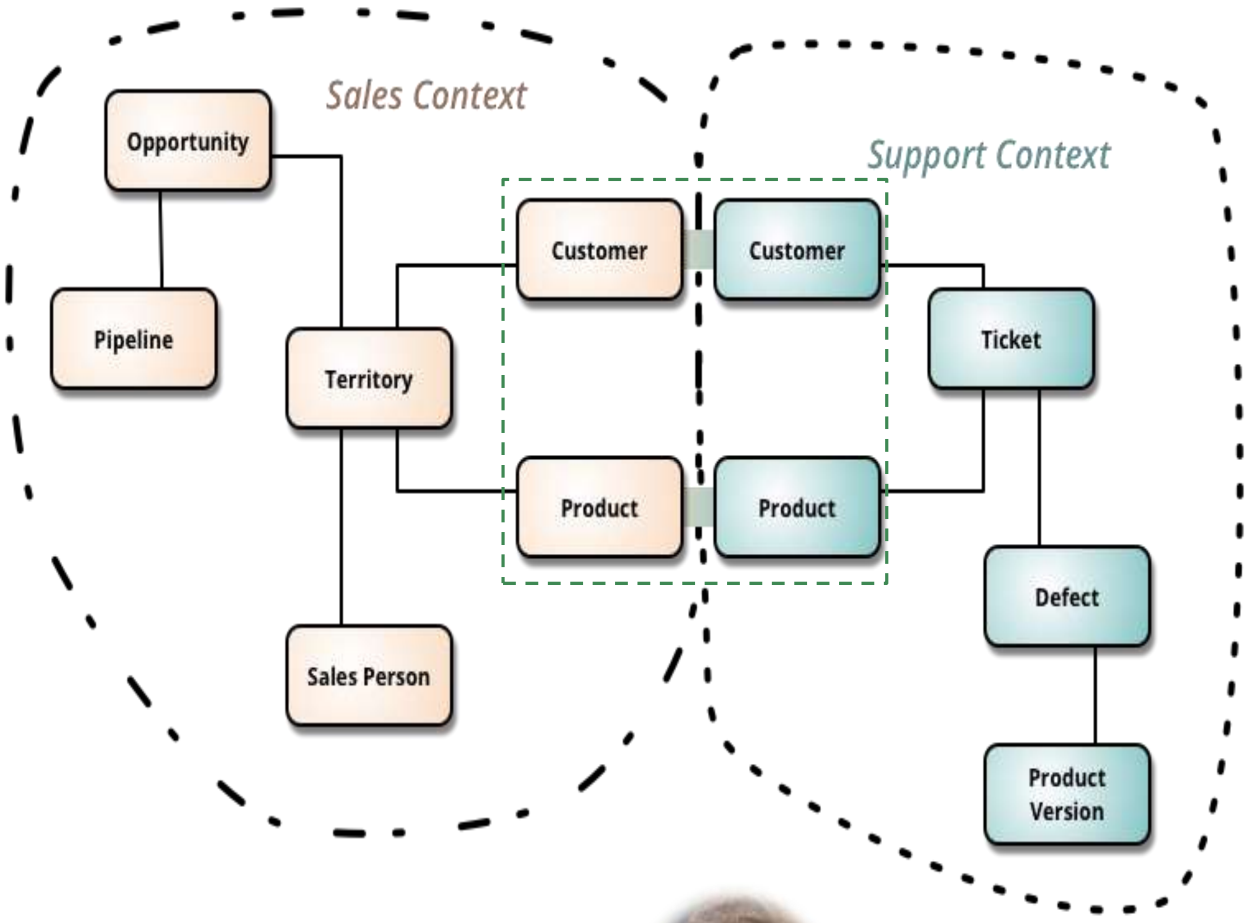
Areas of the domain treated independently

Discovered as you assess requirements and build language



DDD : Understanding Bounded Context

- DDD deals with large models by dividing them into different Bounded Contexts and being explicit about their interrelationships.
- Bounded Contexts have both unrelated concepts
  - Such as a support ticket only existing in a customer support context
  - But also share concepts such as products and customers.
- Different contexts may have completely different models of common concepts with mechanisms to map between these polysemic concepts for integration.



Source: BoundedContext By Martin Fowler : http://martinfowler.com/bliki/BoundedContext.html

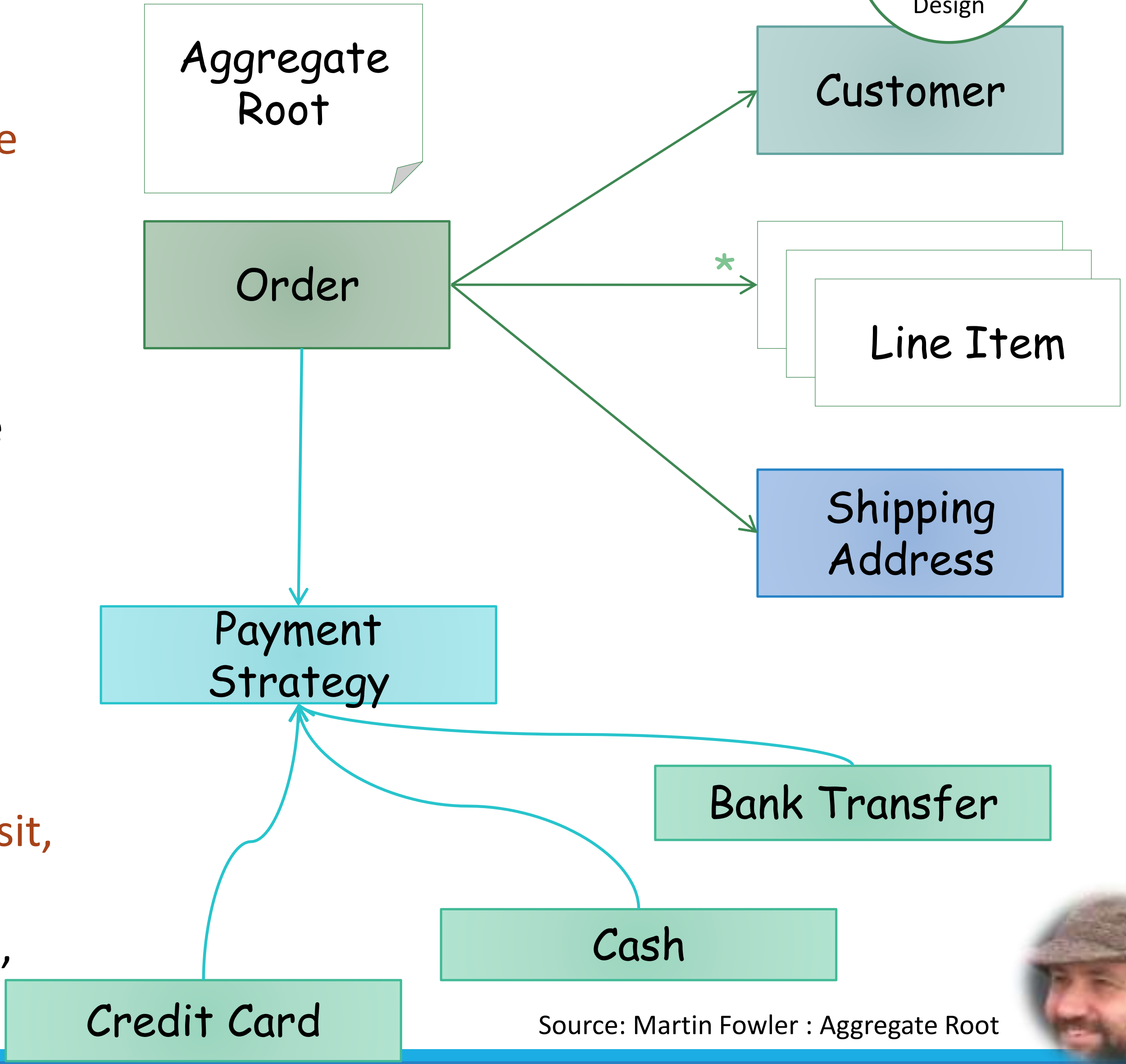
Understanding Aggregate Root

- An aggregate will have one of its component objects be the aggregate root. Any references from outside the aggregate should only go to the aggregate root. The root can thus ensure the integrity of the aggregate as a whole.

- Aggregates are the basic element of transfer of data storage - you request to load or save whole aggregates. Transactions should not cross aggregate boundaries.

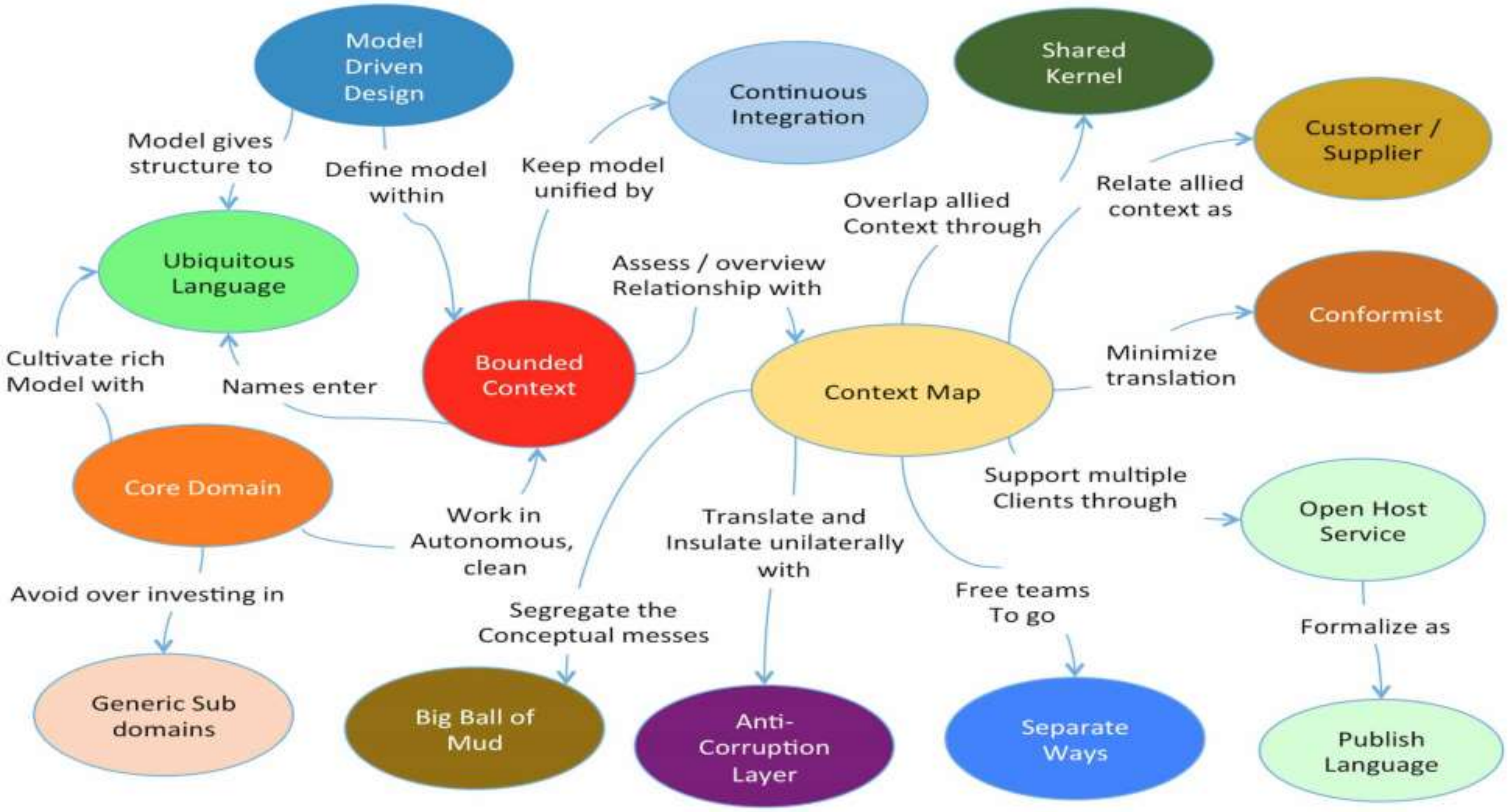
- Aggregates are sometimes confused with collection classes (lists, maps, etc.).

- Aggregates are domain concepts (order, clinic visit, playlist), while collections are generic. An aggregate will often contain multiple collections, together with simple fields.



Source: Martin Fowler : Aggregate Root

DDD : Context Map



Source: Domain-Driven Design Reference by Eric Evans

Shopping Portal

Order Module

Domain Layer

Models

Value Object

- Currency
- Item Value
- Order Status
- Payment Type
- Record State
- Audit Log

Entity

- **Order** (Aggregate Root)
- Order Item
- Shipping Address
- Payment

DTO

- Order
- Order Item
- Shipping Address
- Payment

Services / Ports

- Order Repository
- Order Service
- Order Web Service
- Order Query Web Service
- Shipping Address Web Service
- Payment Web Service

Utils

- Order Factory
- Order Status Converter
- Record State Converter

Adapters

- Order Repository
- Order Service
- Order Web Service
- Order Query Web Service
- Shipping Address Web Service
- Payment Web Service

Adapters Consists of Actual Implementation of the Ports like Database Access, Web Services API etc.

Converters are used to convert an Enum value to a proper Integer value in the Database. For Example Order Status Complete is mapped to integer value 100 in the database.

DDD – Summary

1. Ubiquitous Language
2. Aggregate Root
3. Value Object
4. Domain Events
5. Data Transfer Object
6. Repository Pattern
7. Context Map

More on this in Event Sourcing and CQRS Section.

CRUD / CQRS & Event Sourcing

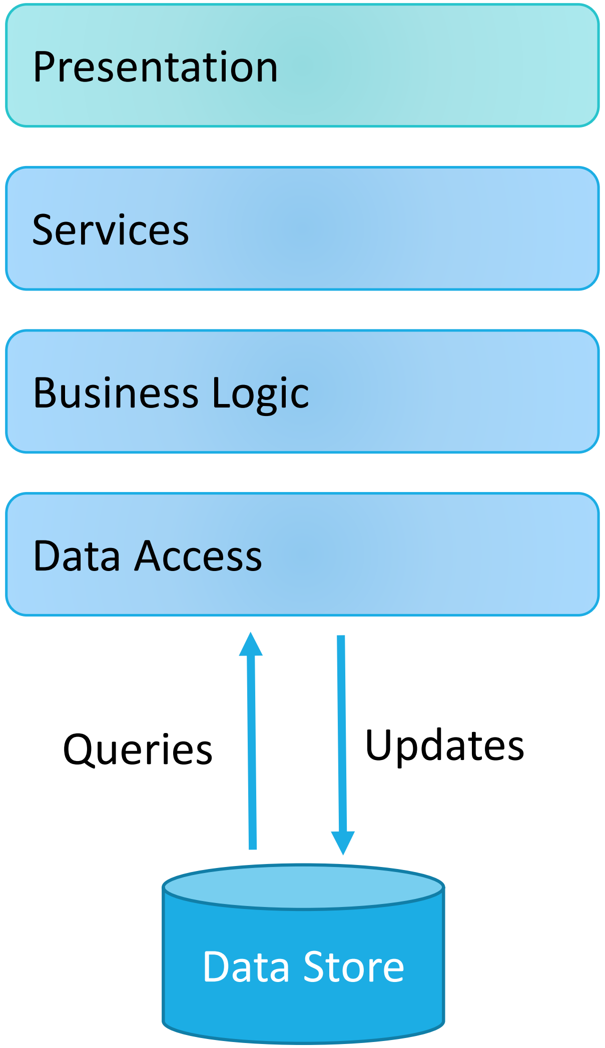
A brief introduction, more in Part 2 of the Series

Event Storming and SAGA

CRUD and CQRS

Source: https://docs.microsoft.com/en-us/azure/architecture/patterns/cqrs
https://blogs.msdn.microsoft.com/maarten_mullender/2004/07/23/crud-only-when-you-can-afford-it-revisite

Traditional CRUD Architecture



Presentation

Services

Business Logic

Data Access

Queries

Updates

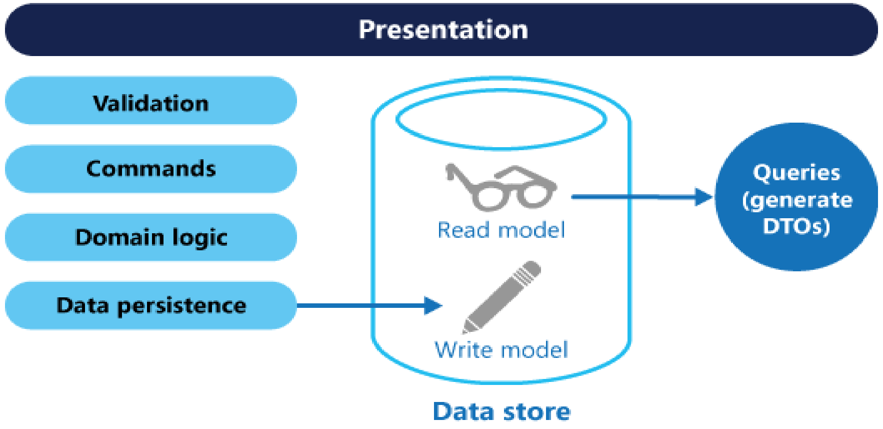
Data Store

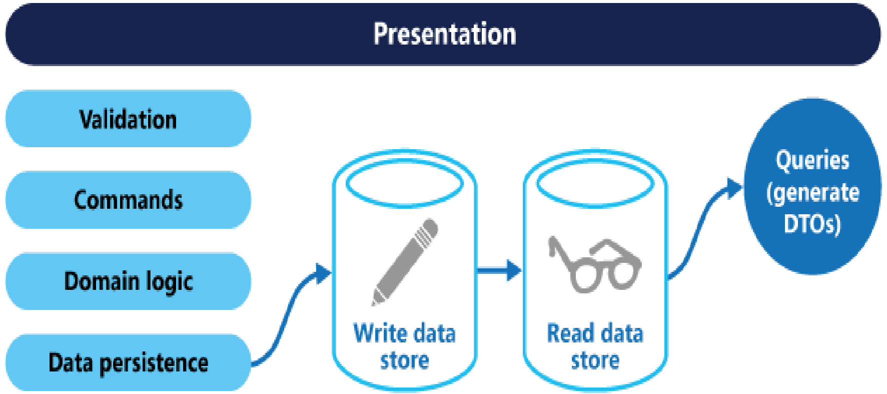
CRUD Disadvantages

- **A mismatch between the read and write** representations of the data.
- It **risks data contention when records are locked in the data store** in a collaborative domain, where multiple actors operate in parallel on the same set of data. These risks increase as the complexity and throughput of the system grows.
- It can make **managing security and permissions more complex** because each entity is subject to both read and write operations, which might expose data in the wrong context.

Event Sourcing & CQRS (Command and Query Responsibility Segregation)

- In traditional data management systems, both commands (updates to the data) and queries (requests for data) are executed against the same set of entities in a single data repository.
- CQRS is a pattern that segregates the operations that read data (Queries) from the operations that update data (Commands) by using separate interfaces.
- CQRS should only be used on specific portions of a system in Bounded Context (in DDD).
- CQRS should be used along with Event Sourcing.





Java Axon Framework Resource : http://www.axonframework.org

MSDN – Microsoft https://msdn.microsoft.com/en-us/library/dn568103.aspx | Martin Fowler : CQRS – http://martinfowler.com/bliki/CQRS.html

Axon Framework For Java

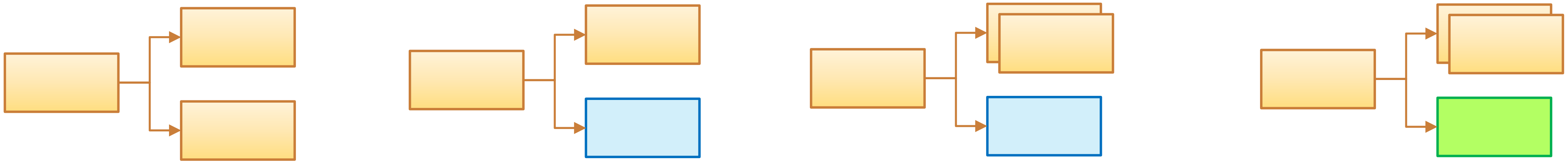
CQS : Bertrand Meyer

Greg Young

2

Event Sourcing Intro

Standard CRUD Operations – Customer Profile – Aggregate Root



Profile Created	Title Updated	New Address added	Notes Removed
Time T1	T2	T3	T4

Event Sourcing and Derived Aggregate Root

Commands

1. Create Profile
2. Update Title
3. Add Address
4. Delete Notes

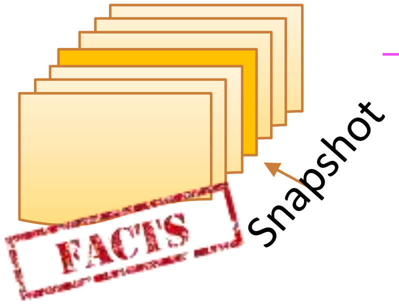
2

Events

1. Profile Created Event
2. Title Updated Event
3. Address Added Event
4. Notes Deleted Event

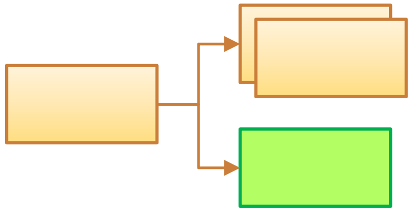
3

Event store



Single Source of Truth

Derived



Current State of the Customer Profile

4

Greg Young

2

Event Sourcing and CQRS Design Example

Domain

The example focus on a concept of a Café which tracks the visit of an individual or group to the café. When people arrive at the café and take a table, a tab is opened. They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served.

Events

- TabOpened
- DrinksOrdered
- FoodOrdered
- DrinksCancelled
- FoodCancelled
- DrinksServed
- FoodPrepared
- FoodServed
- TabClosed

An Event Stream which is an **immutable** collection of events up until a specific version of an **aggregate**.

The purpose of the version is to implement optimistic locking:

Commands

- OpenTab
- PlaceOrder
- AmendOrder
- MarkDrinksServed
- MarkFoodPrepared
- MarkFoodServed
- CloseTab

Commands are things that indicate **requests** to our domain. While an event states that something certainly happened, a command may be **accepted** or **rejected**.

An accepted command leads to zero or more events being emitted to incorporate new facts into the system. A rejected command leads to some kind of exception.

Aggregates

- A Single Object, which doesn't reference any others.

- An isolated Graph of objects, with One object designated as the Root of the Aggregate.

Exception

- CannotCancelServedItem
- TabHasUnservedItem
- MustPayEnough

An important part of the modeling process is thinking about the things that can cause a command to be refused.

Source: http://cqrs.nu/tutorial/cs/01-design

Event Storming : Restaurant Dining Example – Customer Journey

Processes 1

When people arrive at the Restaurant and take a table, a Table is opened. They may then order drinks and food. Drinks are served immediately by the table staff, however food must be cooked by a chef. Once the chef prepared the food it can then be served. Table is closed when the bill is prepared.

Customer Journey thru Dinning Processes

Commands 2

- Add Drinks
- Add Food
- Update Food

- **Open Table**
- Add Juice
- Add Soda
- Add Appetizer 1
- Add Appetizer 2
- Remove Soda
- Add Food 1
- Add Food 2
- Place Order
- **Close Table**

- Serve Drinks
- Prepare Food
- Serve Food

- **Prepare Bill**
- Process Payment

ES Aggregate 4

- Dinning Order
- Billable Order

Microservices

Food Menu

Dining

Kitchen

Order

Payment

Events 3

- Drinks Added
- Food Added
- Food Updated
- Food Discontinued

- Table Opened
- Juice Added
- Soda Added
- Appetizer 1 Added
- Appetizer 2 Added
- Remove Soda
- Food 1 Added
- Food 2 Added
- Order Placed
- Table Closed

- Juice Served
- Soda Served
- Appetizer Served
- Food Prepared
- Food Served

- Bill Prepared
- Payment Processed

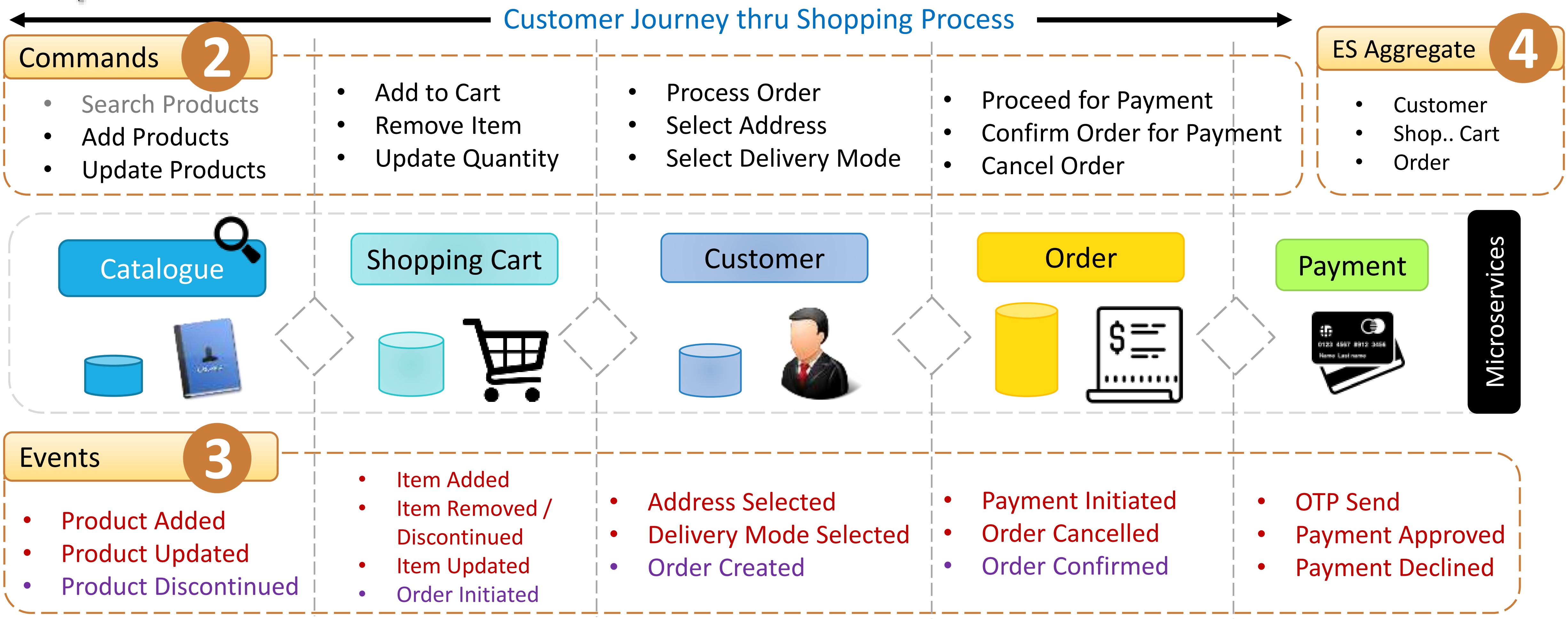
- Payment Approved
- Payment Declined
- Cash Paid

The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

2

Use Case : Shopping Portal – Event Sourcing / CQRS

Commands are End-User interaction with the App and based on the commands (Actions) Events are created. These Events includes both Domain Events and Integration Events. Event Sourced Aggregates will be derived using Domain Events. Each Micro Service will have its own separate Database. Depends on the scalability requirement each of the Micro Service can be scaled separately. For Example. Catalogue can be on a 50 node cluster compared to Customer Micro Service.



Customer Journey thru Shopping Process

Commands 2

- Search Products
- Add Products
- Update Products

- Add to Cart
- Remove Item
- Update Quantity

- Process Order
- Select Address
- Select Delivery Mode

- Proceed for Payment
- Confirm Order for Payment
- Cancel Order

ES Aggregate 4

- Customer
- Shop.. Cart
- Order

Events 3

- Product Added
- Product Updated
- Product Discontinued

- Item Added
- Item Removed / Discontinued
- Item Updated
- Order Initiated

- Address Selected
- Delivery Mode Selected
- Order Created

- Payment Initiated
- Order Cancelled
- Order Confirmed

- OTP Send
- Payment Approved
- Payment Declined

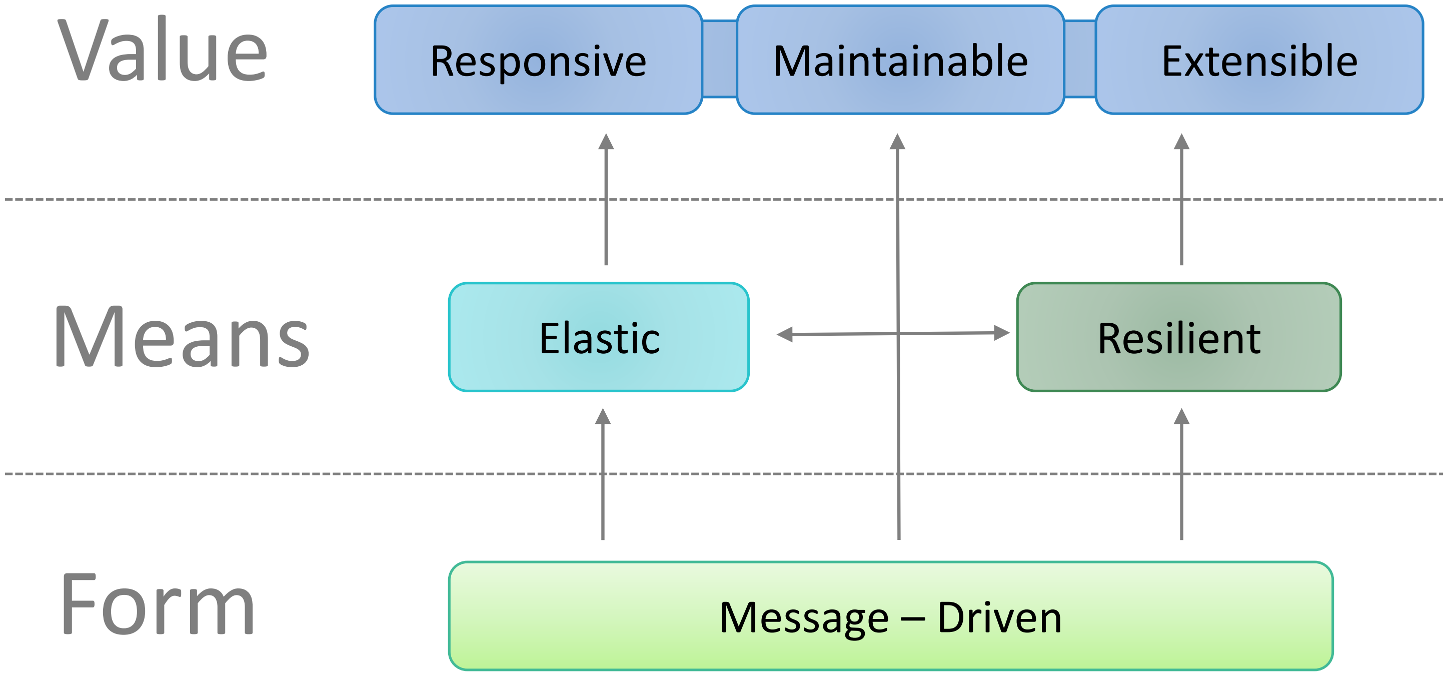
The purpose of this example is to demonstrate the concept of ES / CQRS thru Event Storming principles.

Summary – Event Sourcing and CQRS

1. Immutable Events
2. Events represents the state change in Aggregate Root
3. Aggregates are Derived from a Collection of Events.
4. Separate Read and Write Models
5. Commands (originated from user or systems) creates Events.
6. Commands and Queries are always separated and possibly reads and writes using different data models.

Functional Reactive Programming

Functional Reactive Programming



Principles		What it means?
Responsive	thus	React to users demand
Resilient	thus	React to errors and failures
Elastic	thus	React to load
Message-Driven	thus	React to events and messages

1. A ***responsive, maintainable & Extensible*** application is the goal.
2. A *responsive* application is both ***scalable (Elastic)*** and ***resilient***.
3. Responsiveness is impossible to achieve without both scalability and resilience.
4. A ***Message-Driven*** architecture is the foundation of scalable, resilient, and ultimately responsive systems.

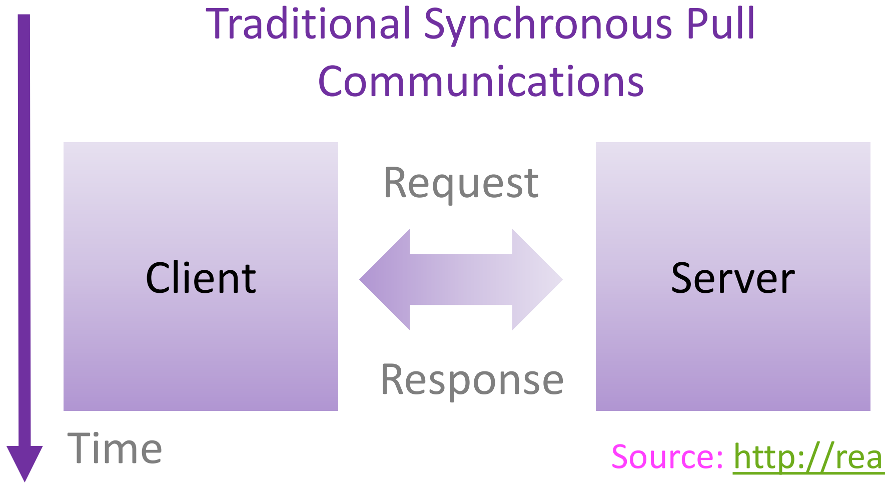
Reactive Extensions (Rx) Source: http://reactivex.io/

4 Building Blocks of RxJava

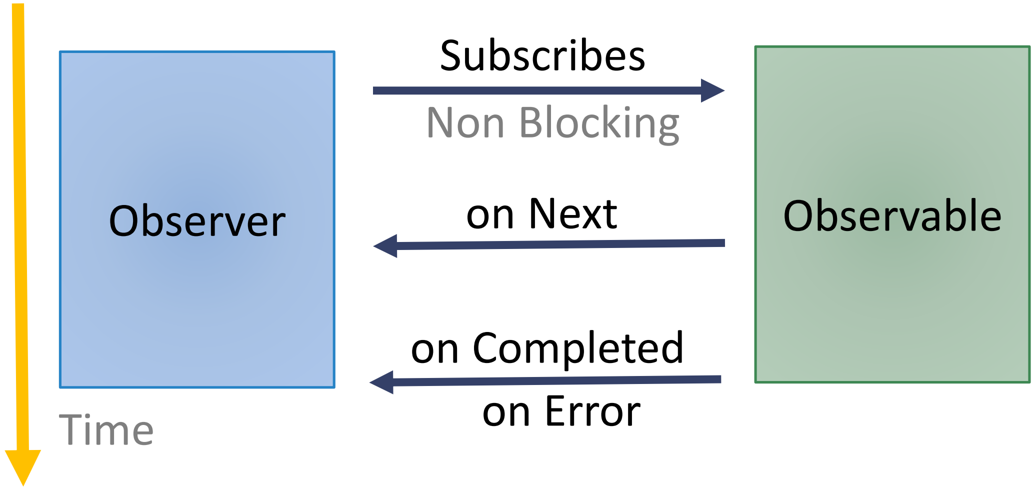
1 **Observable** Source of Data Stream [Sender]

2 **Observer** Listens for emitted values [Receiver]

1. The Observer subscribes (listens) to the Observable
2. Observer react to what ever item or sequence of items the Observable emits.
3. Many Observers can subscribe to the same Observable



Source: http://reactivex.io/



4 Building Blocks of RxJava

Reactive Extensions (Rx)

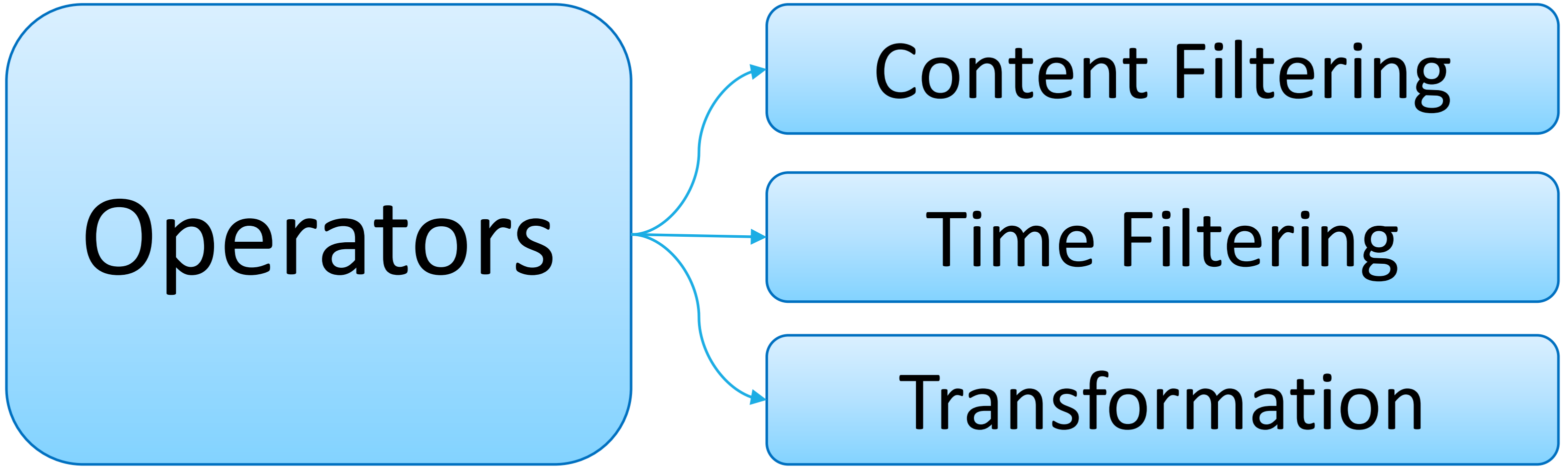
3

Schedulers

Schedulers are used to manage and control concurrency.

1. observeOn: Thread Observable is executed
2. subscribeOn: Thread subscribe is executed

4



Operators

Content Filtering

Time Filtering

Transformation

Operators that let you Transform, Combine, Manipulate, and work with the sequence of items emitted by Observables

Source: http://reactivex.io/

2

Observable / Observer Design Pattern

Building Block

1 & 2

- Allows for Concurrent Operations: the observer does not need to block while waiting for the observable to emit values
- Observer waits to receive values when the observable is ready to emit them
- Based on push rather than pull

Reactive Extensions (Rx)

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

What's missing from GOF Observer Pattern

Building Block

1

1. The ability for the producer to signal to the consumer that there is no more data available (a foreach loop on an Iterable completes and returns normally in such a case; an Observable calls its observer's onComplete method)
2. The ability for the producer to signal to the consumer that an error has occurred (an Iterable throws an exception if an error takes place during iteration; an Observable calls its observer's onError method)
3. Multiple Thread Implementations and hiding those details.
4. **Dozens of Operators to handle data**.

Source: http://reactivex.io/intro.html

Reactive Extensions (Rx)

Compare Iterable Vs. Observable

Observable is the asynchronous / push dual to the synchronous pull Iterable

Reactive Extensions (Rx)

Observables are:

- Composable: Easily chained together or combined
- Flexible: Can be used to emit:
  - A scalar value (network result)
  - Sequence (items in a list)
  - Infinite streams (weather sensor)
- Free from callback hell: Easy to transform one asynchronous stream into another

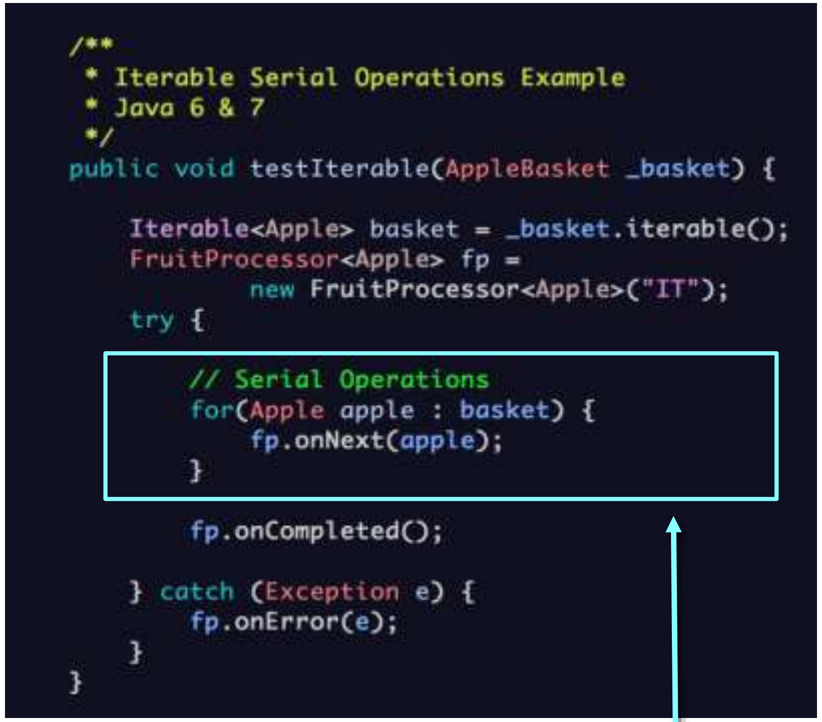
Event	Iterable (Pull)	Observable (Push)
Retrieve Data	T next()	onNext(T)
Discover Error	throws Exception	onError (Exception)
Complete	!hasNext()	onComplete()

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

Comparison : Iterable / Streams / Observable

Building Block 1

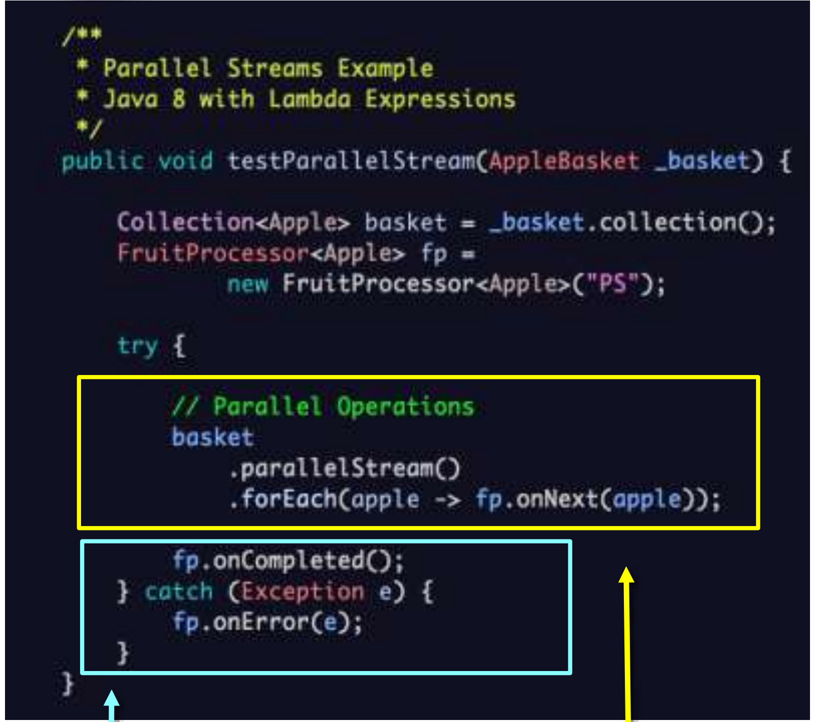
Java 6 – Blocking Call



```
/**
 * Iterable Serial Operations Example
 * Java 6 & 7
 */
public void testIterable(AppleBasket _basket) {

    Iterable<Apple> basket = _basket.iterable();
    FruitProcessor<Apple> fp =
            new FruitProcessor<Apple>("IT");
    try {

        // Serial Operations
        for(Apple apple : basket) {
            fp.onNext(apple);
        }

        fp.onCompleted();

    } catch (Exception e) {
        fp.onError(e);
    }
}
```

First Class Visitor (Consumer)
Serial Operations

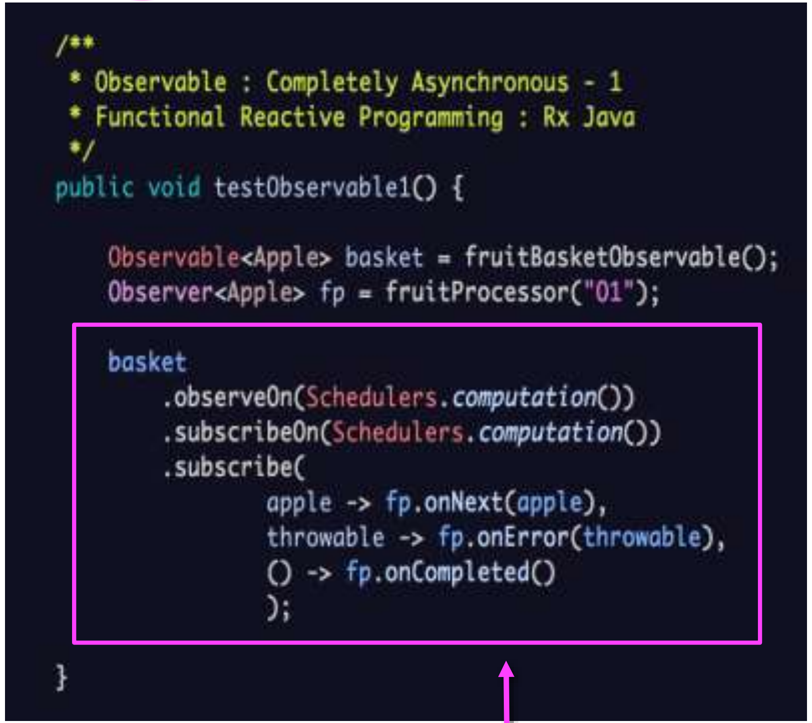
Java 8 – Blocking Call



```
/**
 * Parallel Streams Example
 * Java 8 with Lambda Expressions
 */
public void testParallelStream(AppleBasket _basket) {

    Collection<Apple> basket = _basket.collection();
    FruitProcessor<Apple> fp =
            new FruitProcessor<Apple>("PS");

    try {

        // Parallel Operations
        basket
            .parallelStream()
            .forEach(apple -> fp.onNext(apple));

        fp.onCompleted();
    } catch (Exception e) {
        fp.onError(e);
    }
}
```

Parallel Streams (10x Speed)
Still On Next, On Complete and
On Error are Serial Operations

Rx Java - Freedom



```
/**
 * Observable : Completely Asynchronous - 1
 * Functional Reactive Programming : Rx Java
 */
public void testObservable1() {

    Observable<Apple> basket = fruitBasketObservable();
    Observer<Apple> fp = fruitProcessor("01");

    basket
        .observeOn(Schedulers.computation())
        .subscribeOn(Schedulers.computation())
        .subscribe(
            apple -> fp.onNext(apple),
            throwable -> fp.onError(throwable),
            () -> fp.onCompleted()
        );

}
```

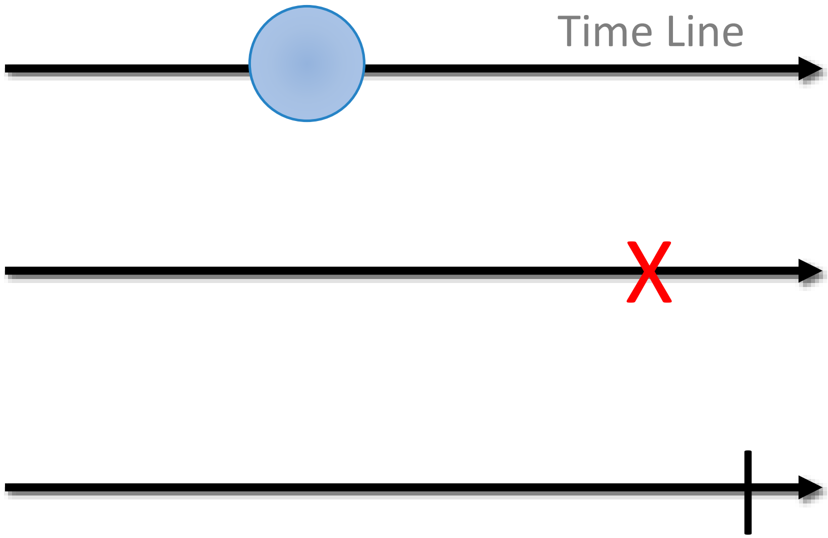
Completely Asynchronous
Operations

Source Code: https://github.com/meta-magic/rxjava

Observer<T> Contract

Methods:

- onNext(T)
- onError(Throwable T)
- onComplete()



onError / onComplete called exactly once

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html

Reactive Extensions (Rx)

2

Functional Reactive Programming : Design Patterns

Single Component Pattern

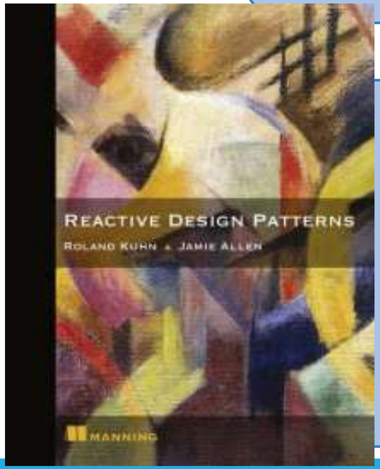
A Component shall do ONLY one thing, But do it in FULL.

Single Responsibility Principle By DeMarco : Structured Analysis & System Specification (Yourdon, New York, 1979)

Let-It-Crash Pattern

Prefer a FULL component restart to complex internal failure handling.

Candea & Fox: Crash-Only Software (USENIX HotOS IX, 2003)
Popularized by Netflix Chaos Monkey. Erlang Philosophy



Saga Pattern

Divide long-lived distributed transactions into quick local ones with compensating actions for recovery.

Pet Helland: Life Beyond Distributed Transactions CIDR 2007

Summary – Functional Reactive Programming

1. A ***responsive, maintainable & Extensible*** application is the goal.

2. A ***responsive*** application is both ***scalable*** ***(Elastic)*** and ***resilient***.

3. Responsiveness is impossible to achieve without both scalability and resilience.

4. A ***Message-Driven*** architecture is the foundation of scalable, resilient, and ultimately responsive systems.

3

Scalability

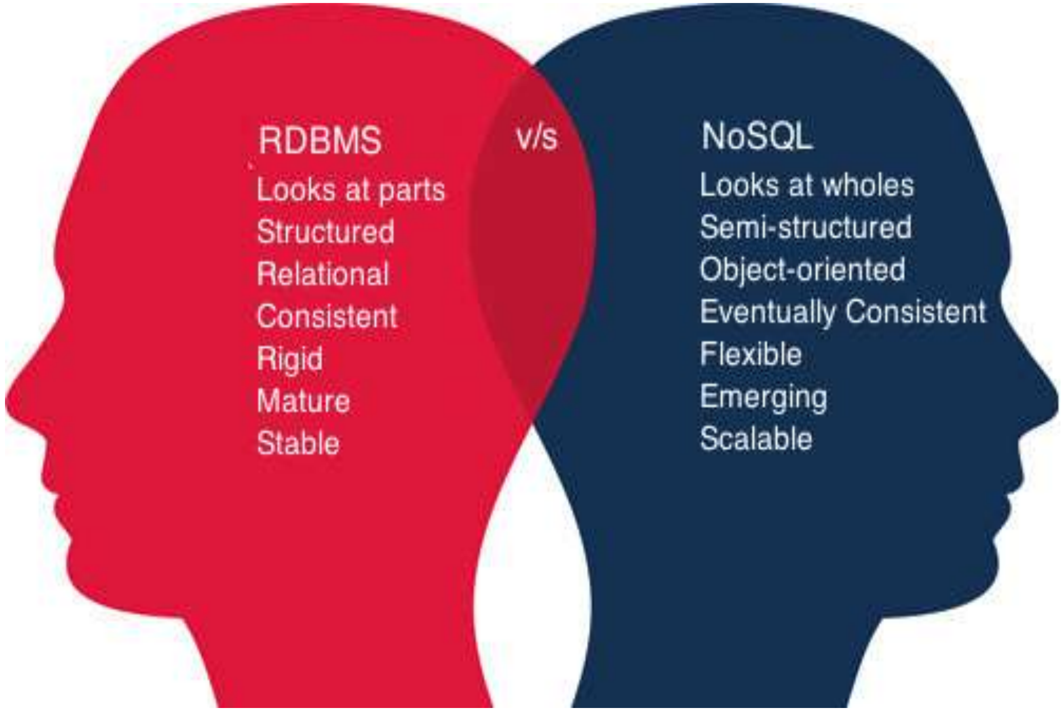
- CAP Theorem
- Distributed Transactions : 2 Phase Commit
- SAGA Design Pattern
- Scalability Lessons from EBay
- Design Patterns
- References

CAP Theorem

by Eric Allen Brewer

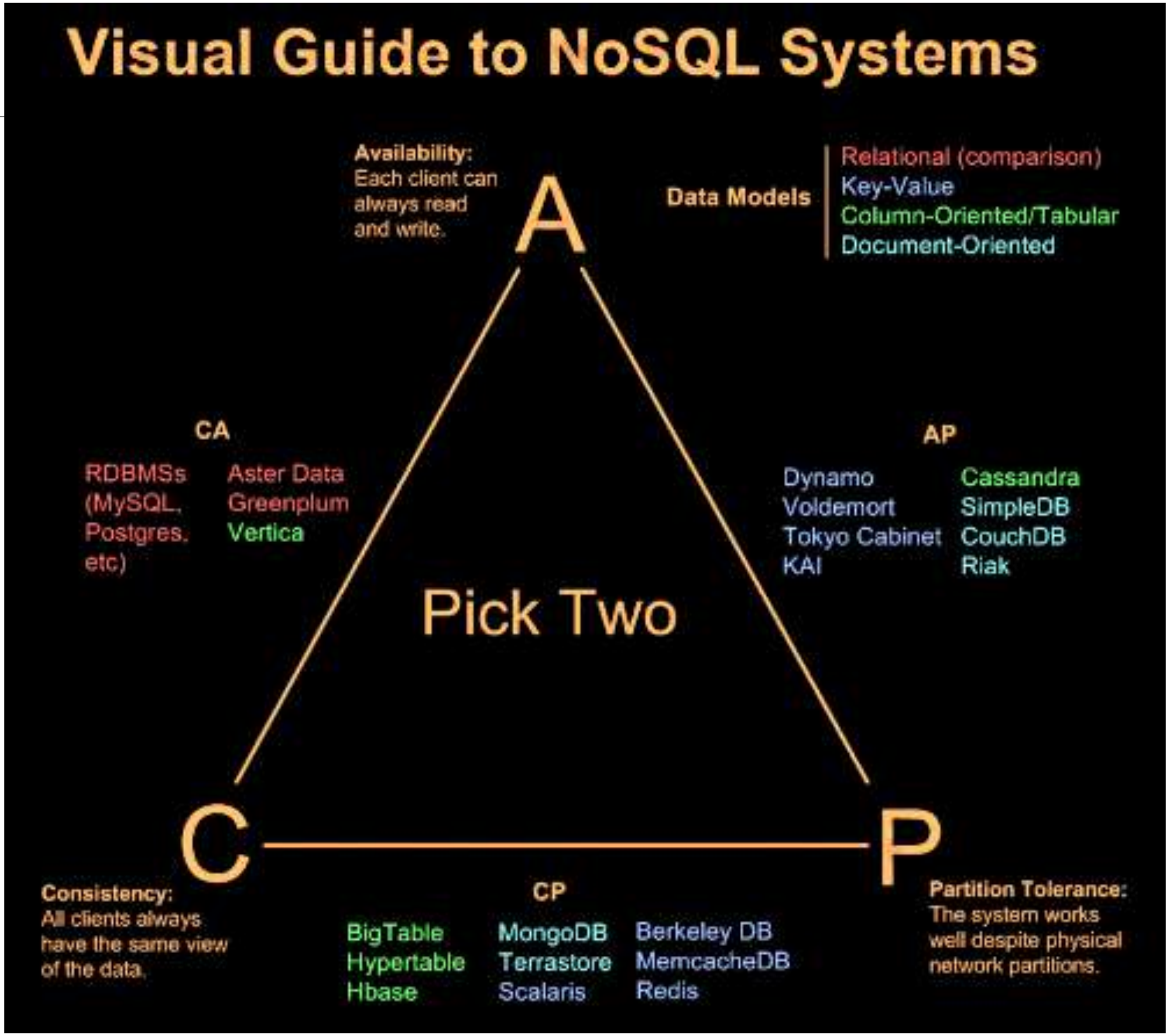
Pick Any 2!!

Say NO to 2 Phase Commit ☺



RDBMS v/s NoSQL

RDBMS	NoSQL
Looks at parts	Looks at wholes
Structured	Semi-structured
Relational	Object-oriented
Consistent	Eventually Consistent
Rigid	Flexible
Mature	Emerging
Stable	Scalable



Visual Guide to NoSQL Systems

Availability: Each client can always read and write.

Data Models: Relational (comparison), Key-Value, Column-Oriented/Tabular, Document-Oriented

CA: RDBMSs (MySQL, Postgres, etc), Aster Data, Greenplum, Vertica

AP: Dynamo, Voldemort, Tokyo Cabinet, KAI, Cassandra, SimpleDB, CouchDB, Riak

CP: BigTable, Hypertable, Hbase, MongoDB, Terrastore, Scalaris, Berkeley DB, MemcacheDB, Redis

Consistency: All clients always have the same view of the data.

Partition Tolerance: The system works well despite physical network partitions.

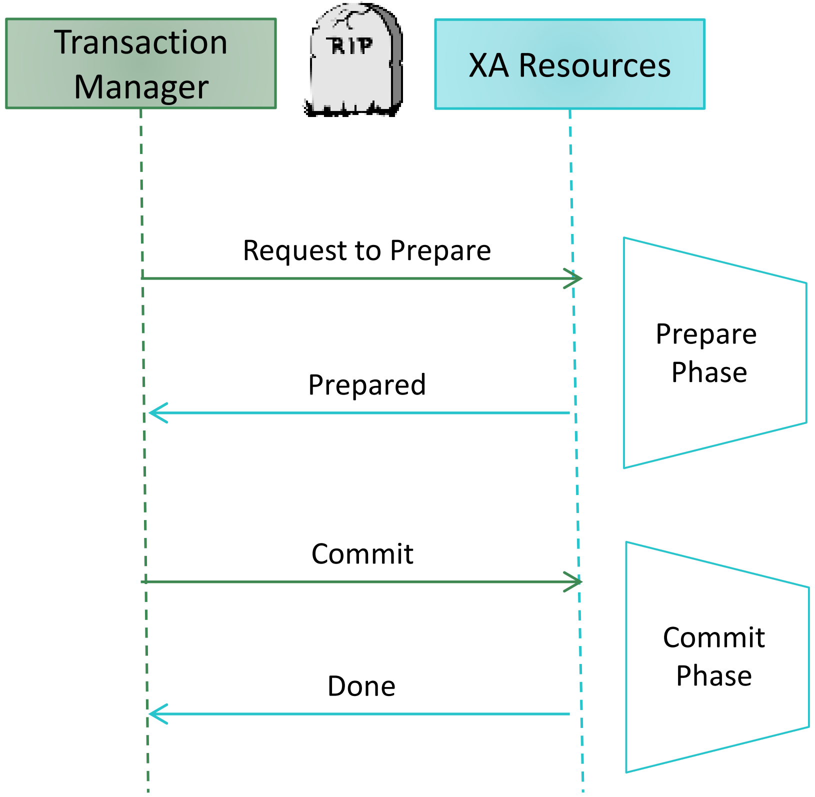
"In a network subject to communication failures, it is impossible for any web service to implement an atomic read / write shared memory that guarantees a response to every request."

Source: http://en.wikipedia.org/wiki/Eric_Brewer_(scientist)

CAP 12 years later: How the "Rules have changed" : http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed

Distributed Transactions : 2 Phase Commit

2 PC or not 2 PC, Wherefore Art Thou XA?



How does 2PC impact scalability?

- Transactions are committed in two phases.
- This involves communicating with every database (XA Resources) involved to determine if the transaction will commit in the first phase.
- During the second phase each database is asked to complete the commit.
- While all of this coordination is going on, locks in all of the data sources are being held.
- ***The longer duration locks create the risk of higher contention.***
- *Additionally, **the two phases require more database processing time than a single phase commit.***
- **The result is lower overall TPS in the system.**

Solution : Resilient System

- Event Based
- Design for failure
- Asynchronous Recovery
- Make all operations idempotent.
- Each DB operation is a 1 PC

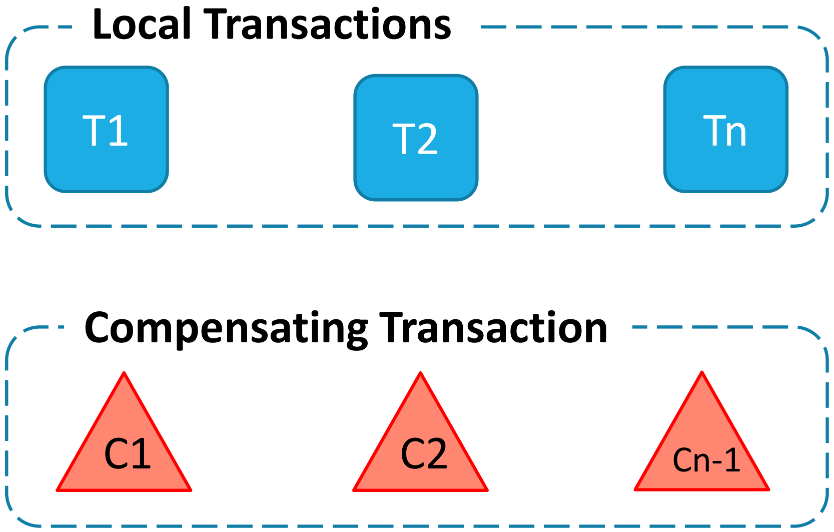
Source : Pat Helland (Amazon) : Life Beyond Distributed Transactions Distributed Computing : http://dancres.github.io/Pages/

3

SAGA Design Pattern instead of 2PC

***Long Lived Transactions (LLTs)** hold on to DB resources for relatively long periods of time, significantly delaying the termination of shorter and more common transactions.*

Divide long–lived, distributed transactions into quick local ones with compensating actions for recovery.



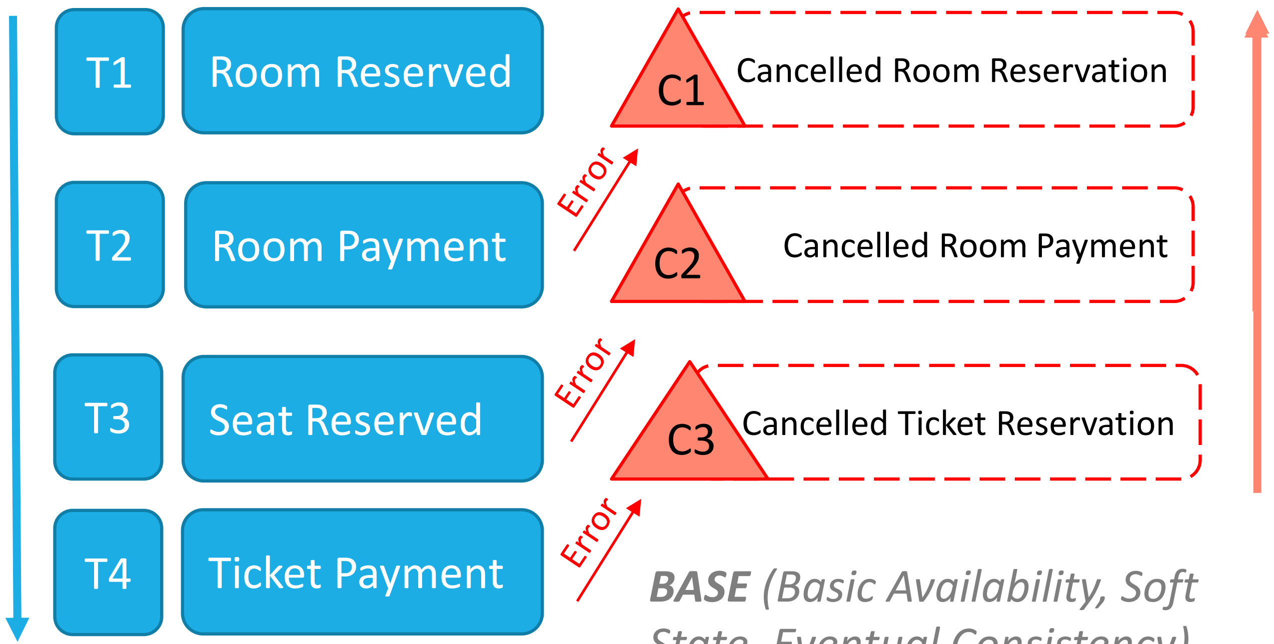
Local Transactions

T1 T2 Tn

Compensating Transaction

C1 C2 Cn-1

Travel : Flight Ticket & Hotel Booking Example



T1 Room Reserved

T2 Room Payment

T3 Seat Reserved

T4 Ticket Payment

Error

Error

Error

C1 Cancelled Room Reservation

C2 Cancelled Room Payment

C3 Cancelled Ticket Reservation

***BASE** (Basic Availability, Soft State, Eventual Consistency)*

Source: SAGAS (1987) Hector Garcia Molina / Kenneth Salem, Dept. of Computer Science, Princeton University, NJ, USA

Handling Invariants – Monolithic to Micro Services

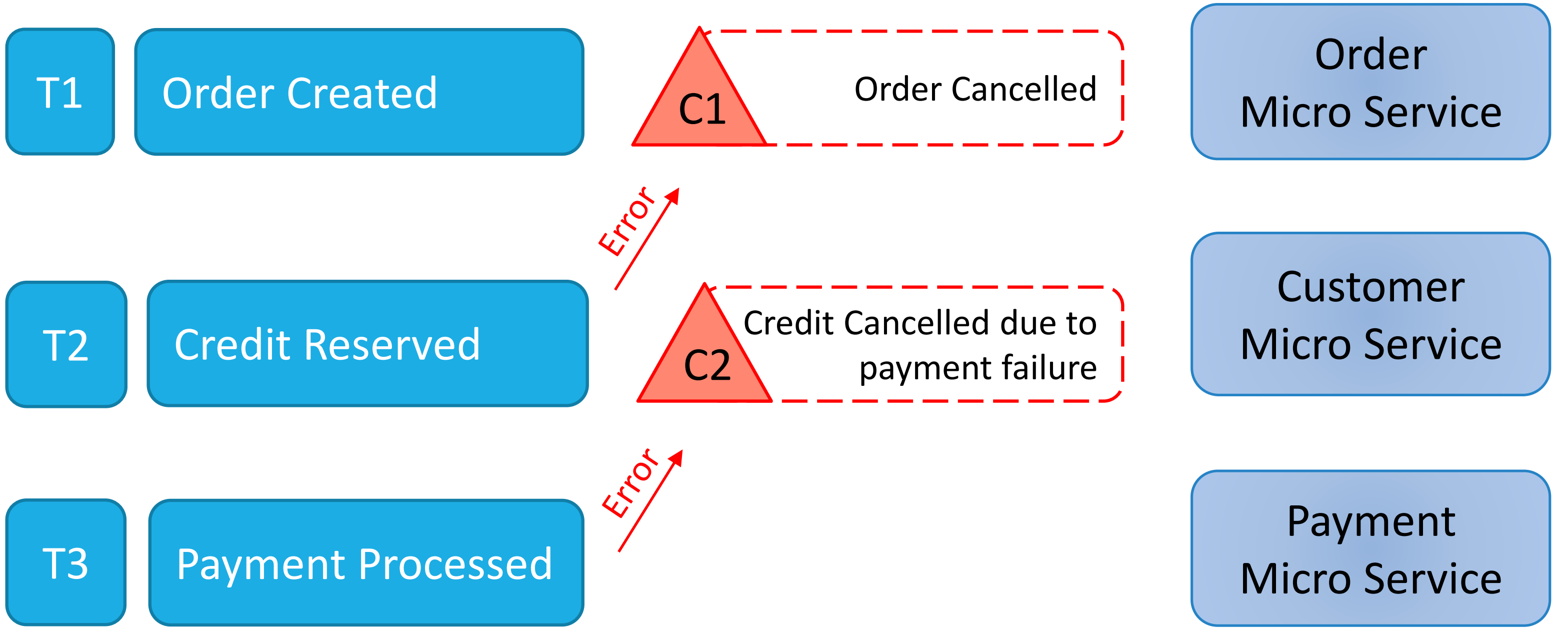
In a typical Monolithic App Customer Credit Limit info and the order processing is part of the same App. Following is a typical pseudo code.

In Micro Services world with Event Sourcing, it's a distributed environment. The order is cancelled if the Credit is NOT available. If the Payment Processing is failed then the Credit Reserved is cancelled.

Monolithic 2 Phase Commit

```
Begin Transaction
        If Order Value <= Available
        Credit
        Process Order
        Process Payments
End Transaction
```

https://en.wikipedia.org/wiki/Invariant_(computer_science)



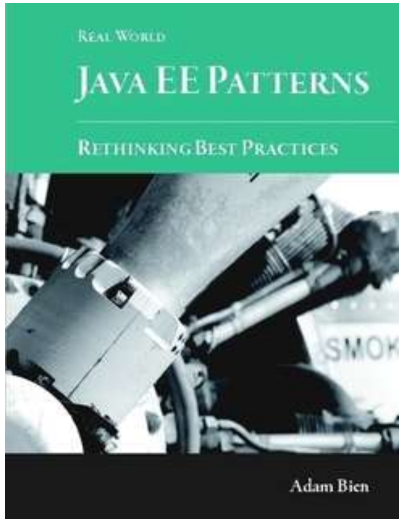
Scalability Best Practices : Lessons from ebay

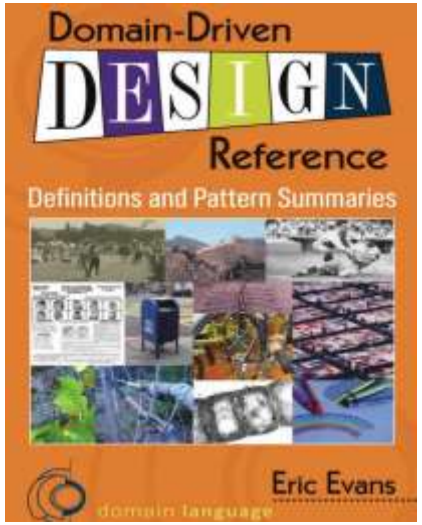
	Best Practices	Highlights
#1	Partition By Function	• Decouple the Unrelated Functionalities. • Selling functionality is served by one set of applications, bidding by another, search by yet another. • 16,000 App Servers in 220 different pools • 1000 logical databases, 400 physical hosts
#2	Split Horizontally	• Break the workload into manageable units. • eBay's interactions are stateless by design • All App Servers are treated equal and none retains any transactional state • Data Partitioning based on specific requirements
#3	Avoid Distributed Transactions	• 2 Phase Commit is a pessimistic approach comes with a big COST • **CAP Theorem** (Consistency, Availability, Partition Tolerance). Apply any two at any point in time. • @ eBay No Distributed Transactions of any kind and NO 2 Phase Commit.
#4	Decouple Functions Asynchronously	• If Component A calls component B synchronously, then they are tightly coupled. For such systems to scale A you need to scale B also. • If Asynchronous A can move forward irrespective of the state of B • SEDA (Staged Event Driven Architecture)
#5	Move Processing to Asynchronous Flow	• Move as much processing towards Asynchronous side • Anything that can wait should wait
#6	Virtualize at All Levels	• Virtualize everything. eBay created their on O/R layer for abstraction
#7	Cache Appropriately	• Cache Slow changing, read-mostly data, meta data, configuration and static data.

Source: http://www.infoq.com/articles/ebay-scalability-best-practices

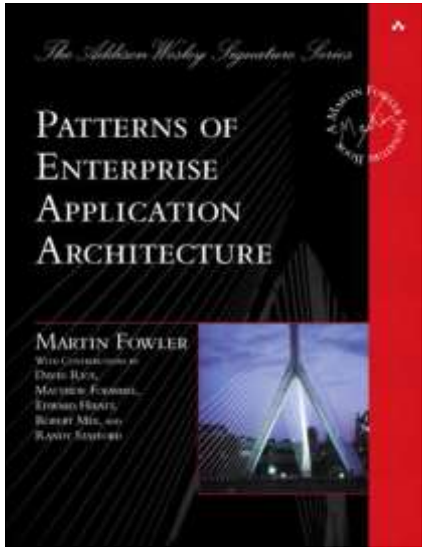
Design Patterns

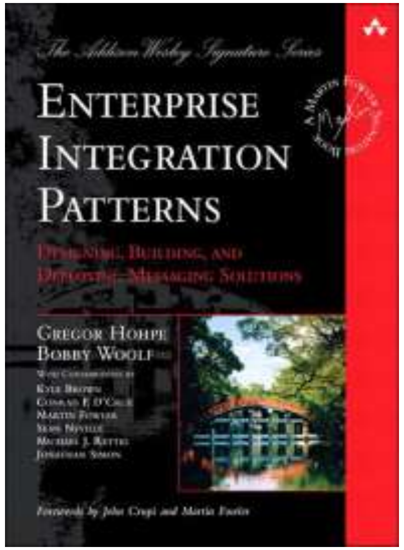
Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

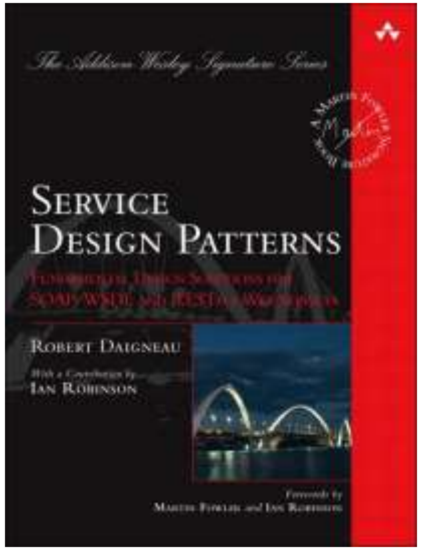


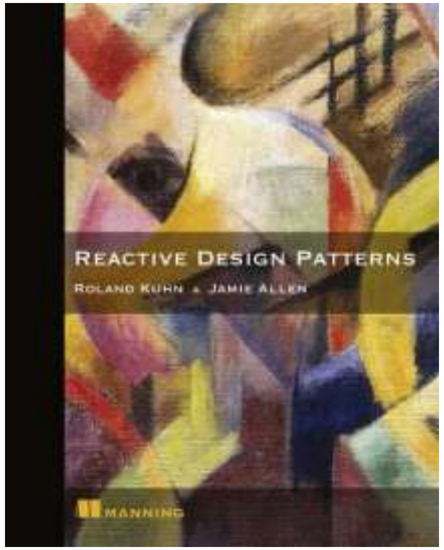


Design Patterns are solutions to general problems that software developers faced during software development.


The Addison-Wesley Signature Series
Patterns of Enterprise Application Architecture
Martin Fowler
With Contributions by
David Rice,
Matthew Foemmel,
Edward Hieatt,
Robert Mee, and
Randy Stafford
A Martin Fowler Signature Book


The Addison-Wesley Signature Series
Enterprise Integration Patterns
Designing, Building, and Deploying Messaging Solutions
Gregor Hohpe
Bobby Woolf
With Contributions by
Kyle Brown
Conrad F. D'Cruz
Martin Fowler
Sean Neville
Michael J. Rettig
Jonathan Simon
Forewords by John Crupi and Martin Fowler
A Martin Fowler Signature Book


The Addison-Wesley Signature Series
Service Design Patterns
Fundamental Design Solutions for SOAP/WSDL and RESTful Web Services
Robert Daigneau
With a Contribution by
Ian Robinson
Forewords by
Martin Fowler and Ian Robinson
A Martin Fowler Signature Book



Summary

1. Highly Scalable & Resilient Architecture
2. Technology Agnostic
3. Easy to Deploy
4. SAGA for Distributed Transaction
5. Faster Go To Market

In a Micro Service Architecture,

The **services tend to get simpler**, but the **architecture tends to get more complex**.

That **complexity** is often managed with **Tooling, Automation, and Process.**

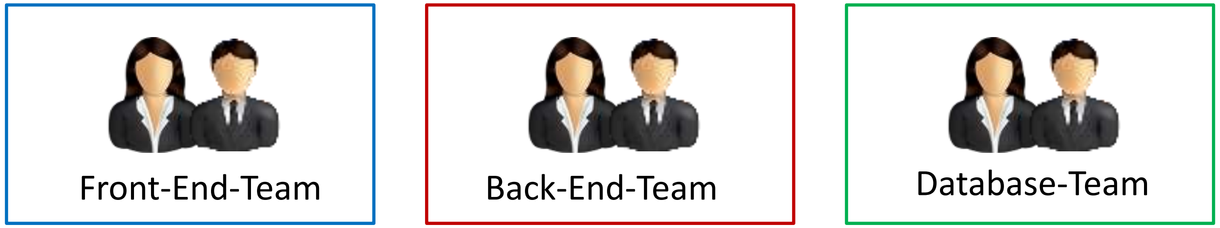
4

Design Styles

- Capability Centric Design
- Microservices Testing Strategies
- Behavior Driven Development
- Features of BDD

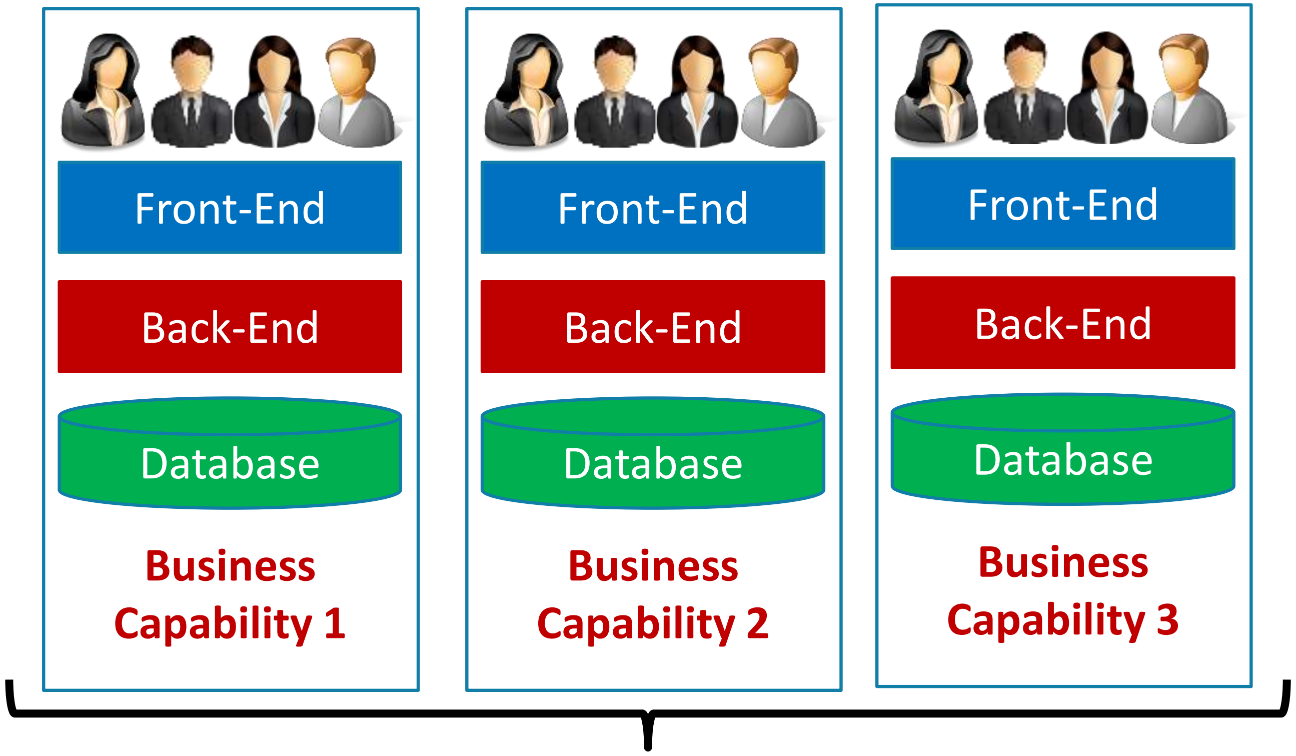
Capability Centric Design

In a typical Monolithic way the team is divided based on technology / skill set rather than business functions. This leads to not only bottlenecks but also lack of understanding of the Business Domain.



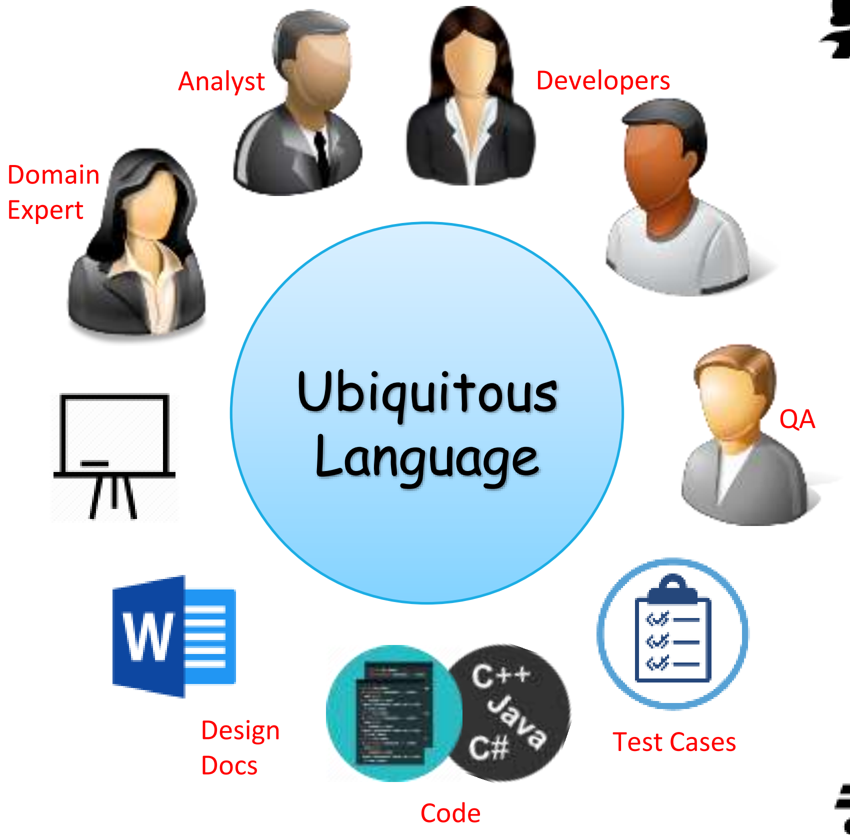
Business Centric Development

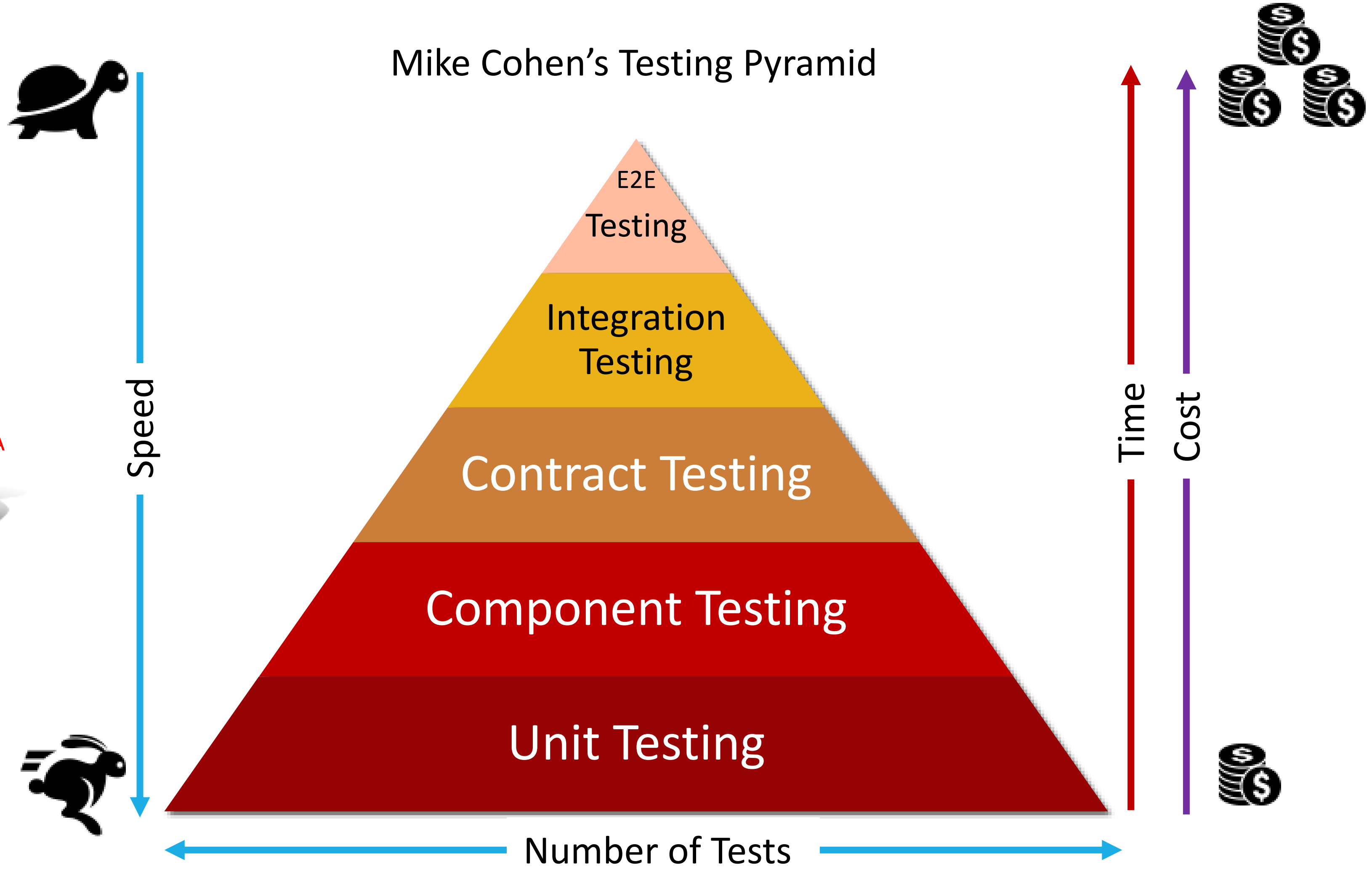
- Focus on Business Capabilities
- Entire team is aligned towards Business Capability.
- From Specs to Operations – The team handles the entire spectrum of Software development.
- Every vertical will have it's own Code Pipeline



Vertically sliced Product Team

Microservices Testing Strategies





Microservices Testing Strategy

Unit Testing

A unit test exercises the smallest piece of testable software in the application to determine whether it behaves as expected.

Integration Testing

An integration test verifies the communication paths and interactions between components to detect interface defects

Component Testing

A component test limits the scope of the exercised software to a portion of the system under test, manipulating the system through internal code interfaces and using test doubles to isolate the code under test from other components.

Contract Testing

An integration contract test is a test at the boundary of an external service verifying that it meets the contract expected by a consuming service.

End 2 End Testing

An end-to-end test verifies that a system meets external requirements and achieves its goals, testing the entire system, from end to end

Source: https://martinfowler.com/articles/microservice-testing/#agenda

Behavior Driven Development

Role-Feature-Reason Matrix

As an insurance Broker
I want to know who my Gold Customers are
So that I sell more

BDD Construct

Given	Customer John Doe exists
When	he buys insurance ABC for $1000 USD
Then	He becomes a Gold Customer

Source: https://dannorth.net/introducing-bdd/

Role-Feature-Reason Matrix

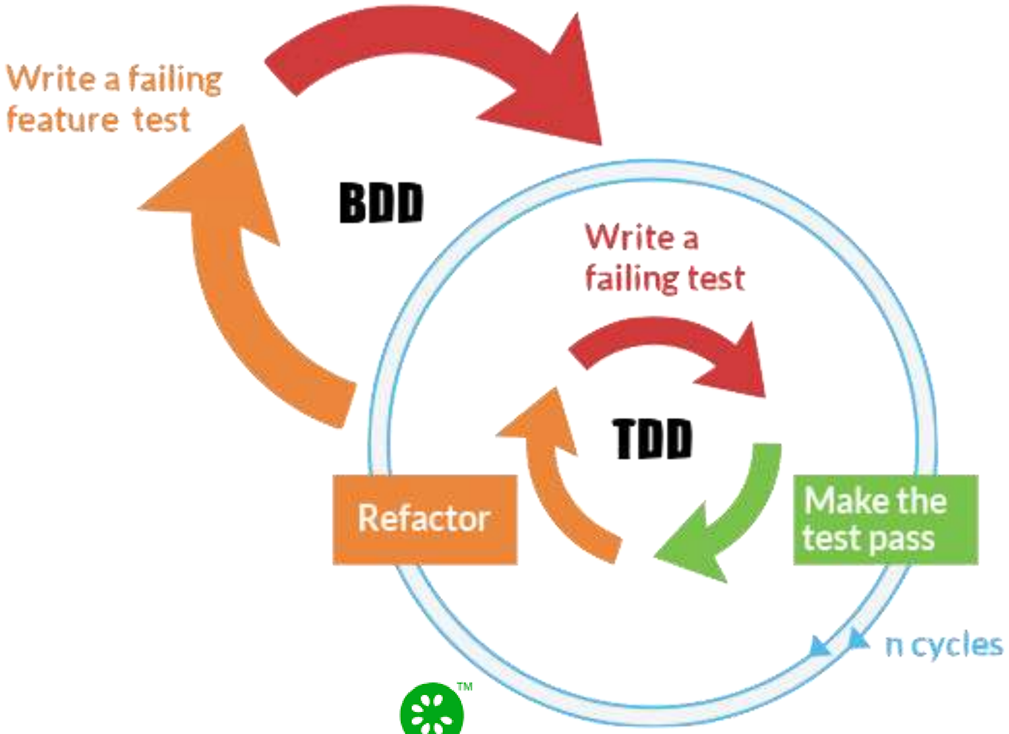
As a Customer
I want withdraw Cash from ATM
So that I don't have to wait in line at the bank

BDD Construct

Given	The account is in Credit AND the Card is Valid AND the dispenser contains Cash
When	The Customer requests Cash
Then	Ensure that the Account is debited AND Ensure cash is dispensed AND ensure that Card is returned.

Features of BDD

- Focus on **Behavior of the System** rather than tests.
- **Collaboration** between Business Stake holders, Analysts, Developers, QA.
- **Ubiquitous Language**
- Driven By **Business Value**
- Extends Test Driven Development



cucumber™

Cucumber merges specification and test documentation into one cohesive whole.

Source: https://cucumber.io/

References

1. Lewis, James, and Martin Fowler. "Microservices: A Definition of This New Architectural Term", March 25, 2014.
2. Miller, Matt. "Innovate or Die: The Rise of Microservices". *The Wall Street Journal*, October 5, 2015.
3. Newman, Sam. *Building Microservices*. O'Reilly Media, 2015.
4. Alagarasan, Vijay. "Seven Microservices Anti-patterns", August 24, 2015.
5. Cockcroft, Adrian. "State of the Art in Microservices", December 4, 2014.
6. Fowler, Martin. "Microservice Prerequisites", August 28, 2014.
7. Fowler, Martin. "Microservice Tradeoffs", July 1, 2015.
8. Humble, Jez. "Four Principles of Low-Risk Software Release", February 16, 2012.
9. Zuul Edge Server, Ketan Gote, May 22, 2017
10. Ribbon, Hysterix using Spring Feign, Ketan Gote, May 22, 2017
11. Eureka Server with Spring Cloud, Ketan Gote, May 22, 2017
12. Apache Kafka, A Distributed Streaming Platform, Ketan Gote, May 20, 2017
13. Functional Reactive Programming, Araf Karsh Hamid, August 7, 2016
14. Enterprise Software Architectures, Araf Karsh Hamid, July 30, 2016
15. Docker and Linux Containers, Araf Karsh Hamid, April 28, 2015

References

Domain Driven Design

16. Oct 27, 2012 What I have learned about DDD Since the book. By Eric Evans
17. Mar 19, 2013 Domain Driven Design By Eric Evans
18. May 16, 2015 Microsoft Ignite: Domain Driven Design for the Database Driven Mind
19. Jun 02, 2015 Applied DDD in Java EE 7 and Open Source World
20. Aug 23, 2016 Domain Driven Design the Good Parts By Jimmy Bogard
21. Sep 22, 2016 GOTO 2015 – DDD & REST Domain Driven API's for the Web. By Oliver Gierke
22. Jan 24, 2017 Spring Developer – Developing Micro Services with Aggregates. By Chris Richardson
23. May 17. 2017 DEVOXX – The Art of Discovering Bounded Contexts. By Nick Tune

Event Sourcing and CQRS

23. Nov 13, 2014 GOTO 2014 – Event Sourcing. By Greg Young
24. Mar 22, 2016 Spring Developer – Building Micro Services with Event Sourcing and CQRS
25. Apr 15, 2016 YOW! Nights – Event Sourcing. By Martin Fowler
26. May 08, 2017 When Micro Services Meet Event Sourcing. By Vinicius Gomes

References

27. MSDN – Microsoft https://msdn.microsoft.com/en-us/library/dn568103.aspx
28. Martin Fowler : CQRS – http://martinfowler.com/bliki/CQRS.html
29. Udi Dahan : CQRS – http://www.udidahan.com/2009/12/09/clarified-cqrs/
30. Greg Young : CQRS - https://www.youtube.com/watch?v=JHGkaShoyNs
31. Bertrand Meyer – CQS - http://en.wikipedia.org/wiki/Bertrand_Meyer
32. CQS : http://en.wikipedia.org/wiki/Command–query_separation
33. CAP Theorem : http://en.wikipedia.org/wiki/CAP_theorem
34. CAP Theorem : http://www.julianbrowne.com/article/viewer/brewers-cap-theorem
35. CAP 12 years how the rules have changed
36. EBay Scalability Best Practices : http://www.infoq.com/articles/ebay-scalability-best-practices
37. Pat Helland (Amazon) : Life beyond distributed transactions
38. Stanford University: Rx https://www.youtube.com/watch?v=y9xudo3C1Cw
39. Princeton University: SAGAS (1987) Hector Garcia Molina / Kenneth Salem
40. Rx Observable : https://dzone.com/articles/using-rx-java-observable

References – Micro Services – Videos

41. Martin Fowler – Micro Services : https://www.youtube.com/watch?v=2yko4TbC8cI&feature=youtu.be&t=15m53s
42. GOTO 2016 – Microservices at NetFlix Scale: Principles, Tradeoffs & Lessons Learned. By R Meshenberg
43. Mastering Chaos – A NetFlix Guide to Microservices. By Josh Evans
44. GOTO 2015 – Challenges Implementing Micro Services By Fred George
45. GOTO 2016 – From Monolith to Microservices at Zalando. By Rodrigue Scaefer
46. GOTO 2015 – Microservices @ Spotify. By Kevin Goldsmith
47. Modelling Microservices @ Spotify : https://www.youtube.com/watch?v=7XDA044tl8k
48. GOTO 2015 – DDD & Microservices: At last, Some Boundaries By Eric Evans
49. GOTO 2016 – What I wish I had known before Scaling Uber to 1000 Services. By Matt Ranney
50. DDD Europe – Tackling Complexity in the Heart of Software By Eric Evans, April 11, 2016
51. AWS re:Invent 2016 – From Monolithic to Microservices: Evolving Architecture Patterns. By Emerson L, Gilt D. Chiles
52. AWS 2017 – An overview of designing Microservices based Applications on AWS. By Peter Dalbhanjan
53. GOTO Jun, 2017 – Effective Microservices in a Data Centric World. By Randy Shoup.
54. GOTO July, 2017 – The Seven (more) Deadly Sins of Microservices. By Daniel Bryant
55. Sept, 2017 – Airbnb, From Monolith to Microservices: How to scale your Architecture. By Melanie Cubula
56. GOTO Sept, 2017 – Rethinking Microservices with Stateful Streams. By Ben Stopford.
57. GOTO 2017 – Microservices without Servers. By Glynn Bird.

Thank you

Araf Karsh Hamid : Co-Founder / CTO
araf.karsh@metamagic.in
USA: +1 (973) 969-2921
India: +91.999.545.8627
Skype / LinkedIn / Twitter / Slideshare : arafkarsh
http://www.slideshare.net/arafkarsh
https://www.linkedin.com/in/arafkarsh/

Enterprise Software Architectures

Hexagonal & Onion Architecture
Event Sourcing & CQRS
DDD – Domain Driven Design
Functional Reactive Programming

Araf Karsh Hamid

"It is NOT the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."
Charles Darwin

http://www.slideshare.net/arafkarsh/software-architecture-styles-64537120

Functional Reactive Programming

Araf Karsh Hamid

"It is NOT the strongest of the species that survives, nor the most intelligent that survives. It is the one that is the most adaptable to change."
Charles Darwin

http://www.slideshare.net/arafkarsh/functional-reactive-programming-64780160

Function Point Analysis

ISO/IEC 20926:2009

ARAF KARSH HAMID

http://www.slideshare.net/arafkarsh/function-point-analysis-65711721

API Gateway Features (Zuul)

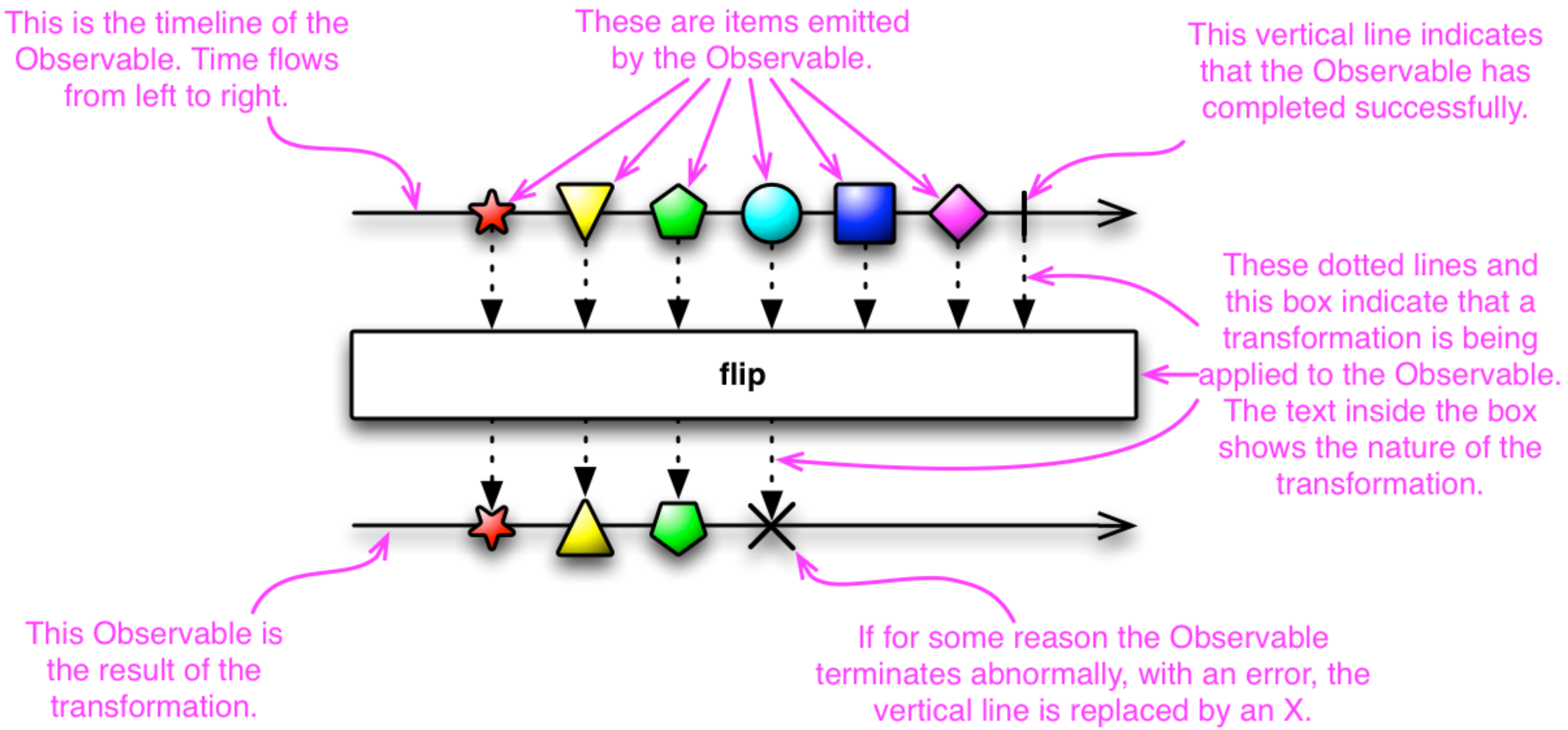
- **Authentication and Security:** identifying authentication requirements for each resource.
- **Insights and Monitoring**: tracking meaningful data and statistics.
- **Dynamic Routing:** dynamically routing requests to different backend..
- **Stress Testing:** gradually increasing the traffic.
- **Load Shedding:** allocating capacity for each type of request and dropping requests.
- **Static Response handling:** building some responses directly.
- **Multi region Resiliency:** routing requests across AWS regions.

- **Hystrix** is used to wrap calls to our origins, which allows us to shed and prioritize traffic when issues occur.
- **Ribbon** is our client for all outbound requests from Zuul, which provides detailed information into network performance and errors, as well as handles software load balancing for even load distribution.
- **Turbine** aggregates fine-grained metrics in real-time so that we can quickly observe and react to problems.
- **Archaius** handles configuration and gives the ability to dynamically change properties.

Source: https://dzone.com/articles/spring-cloud-netflix-zuul-edge-serverapi-gatewayga

Observable Design Pattern (Marble Diagram)

- An *Observer subscribes* to an *Observable*.
- Then that observer reacts to whatever item or sequence of items the Observable *emits*.



This is the timeline of the Observable. Time flows from left to right.

These are items emitted by the Observable.

This vertical line indicates that the Observable has completed successfully.

These dotted lines and this box indicate that a transformation is being applied to the Observable. The text inside the box shows the nature of the transformation.

This Observable is the result of the transformation.

If for some reason the Observable terminates abnormally, with an error, the vertical line is replaced by an X.

Reactive Extensions (Rx)

Source: http://reactivex.io/RxJava/javadoc/index.html?rx/Observable.html | http://rxmarbles.com

RxJava Scheduler Details

- If you want to introduce multithreading into your cascade of Observable operators, you can do so by instructing those operators (or particular Observables) to operate on particular *Schedulers*.

- By default, an Observable and the chain of operators that you apply to it will do its work, and will notify its observers, on the same thread on which its Subscribe method is called.

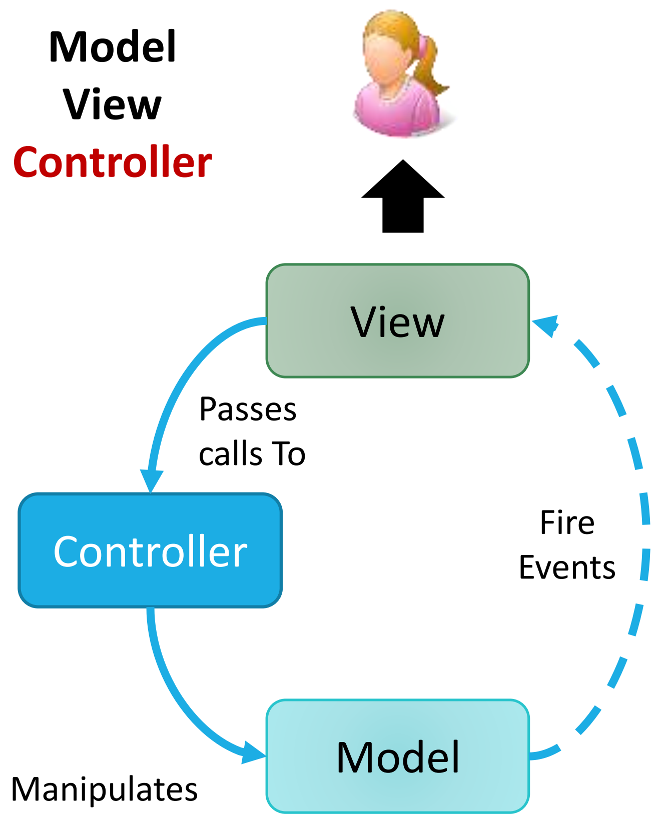
- The SubscribeOn operator changes this behavior by specifying a different Scheduler on which the Observable should operate. TheObserveOn operator specifies a different Scheduler that the Observable will use to send notifications to its observers.

Source: http://reactivex.io/documentation/scheduler.html

Reactive Extensions (Rx)

UI Design Patterns

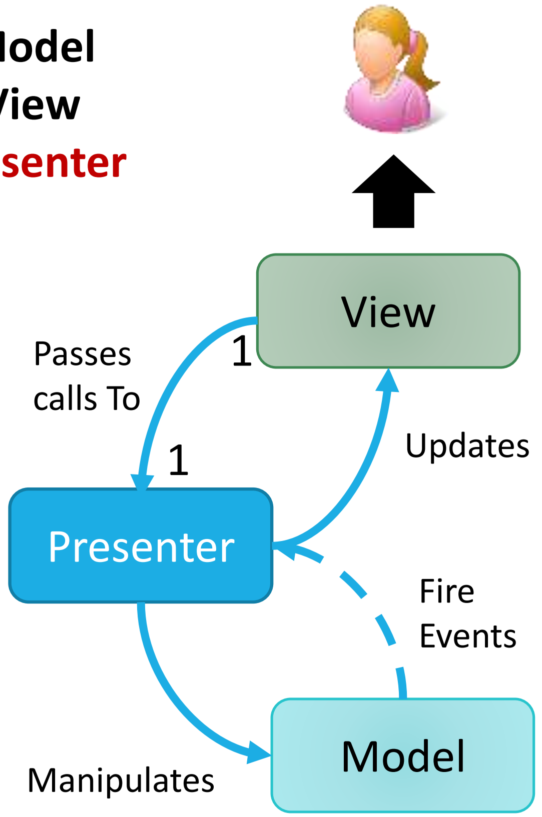
Model View **Controller**



- The **Controller** is responsible to process incoming requests. It receives input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Typically, it acts as the coordinator between the View and the Model.

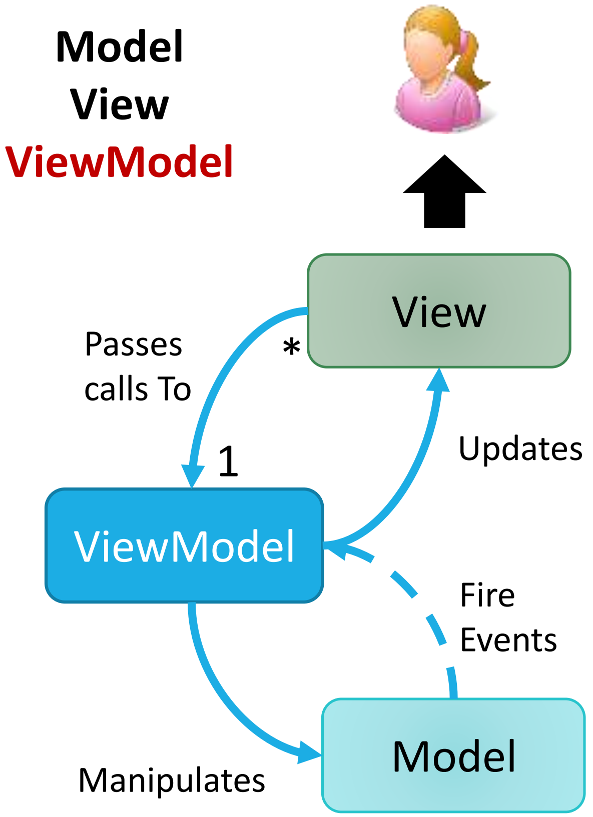
UI Design Patterns
MVC / MVP / MVVM

Model View **Presenter**



- The **Presenter** is responsible for handling all UI events on behalf of the view. This receive input from users via the View, then process the user's data with the help of Model and passing the results back to the View.
- Unlike view and controller, **view and presenter are completely decoupled from each other's and communicate to each other's by an interface**. Also, presenter does not manage the incoming request traffic as controller.
- **Supports two-way data binding between View and ViewModel.**

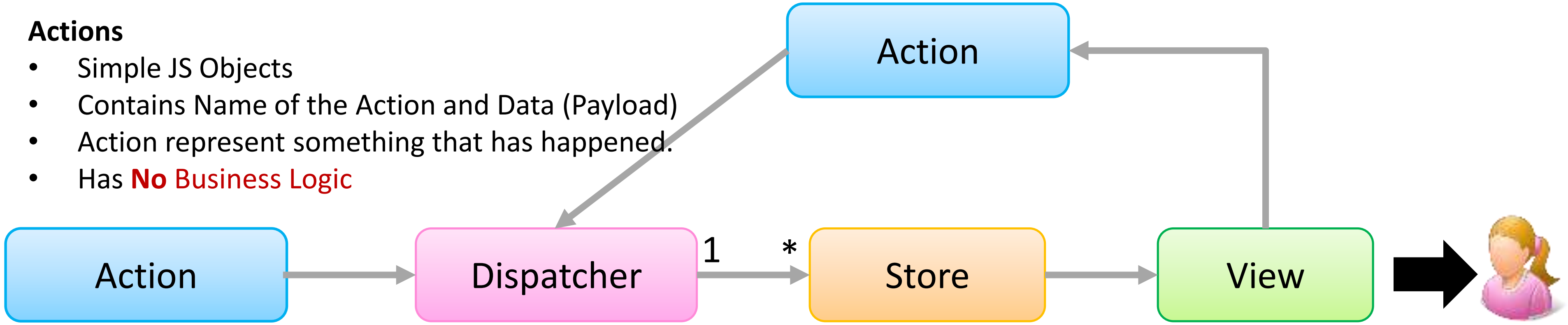
Model View **ViewModel**



- The View Model is responsible for exposing methods, commands, and other properties that helps to maintain the state of the view, manipulate the model as the result of actions on the view, and trigger events in the view itself.
- There is **many-to-one** relationship between View and ViewModel means many View can be mapped to one ViewModel.
- **Supports two-way data binding between View and ViewModel.**

Actions

- Simple JS Objects
- Contains Name of the Action and Data (Payload)
- Action represent something that has happened.
- Has **No** Business Logic



Action

Action

Dispatcher

1

*

Store

View

Every action is sent to all Stores via callbacks the stores register with the Dispatcher

Controller-Views

- Listens to Store changes
- Emit Actions to Dispatcher

Stores

- Contains state for a Domain (Vs. Specific Component)
- **In Charge** of modifying the Data
- Inform the views when the Data is changed by emitting the Changed Event.

Dispatcher

- Single Dispatcher per Application
- Manages the Data Flow View to Model
- Receives Actions and dispatch them to Stores

Flux Core Concepts

1. One way Data Flow
2. No Event Chaining
3. Entire App State is resolved in store before Views Update
4. Data Manipulation ONLY happen in one place (Store).

UI Design Patterns
Flux / Redux

Redux Core Concepts

1. One way Data Flow
2. No Dispatcher compared to Flux
3. Immutable Store

Available for React & Angular

Store

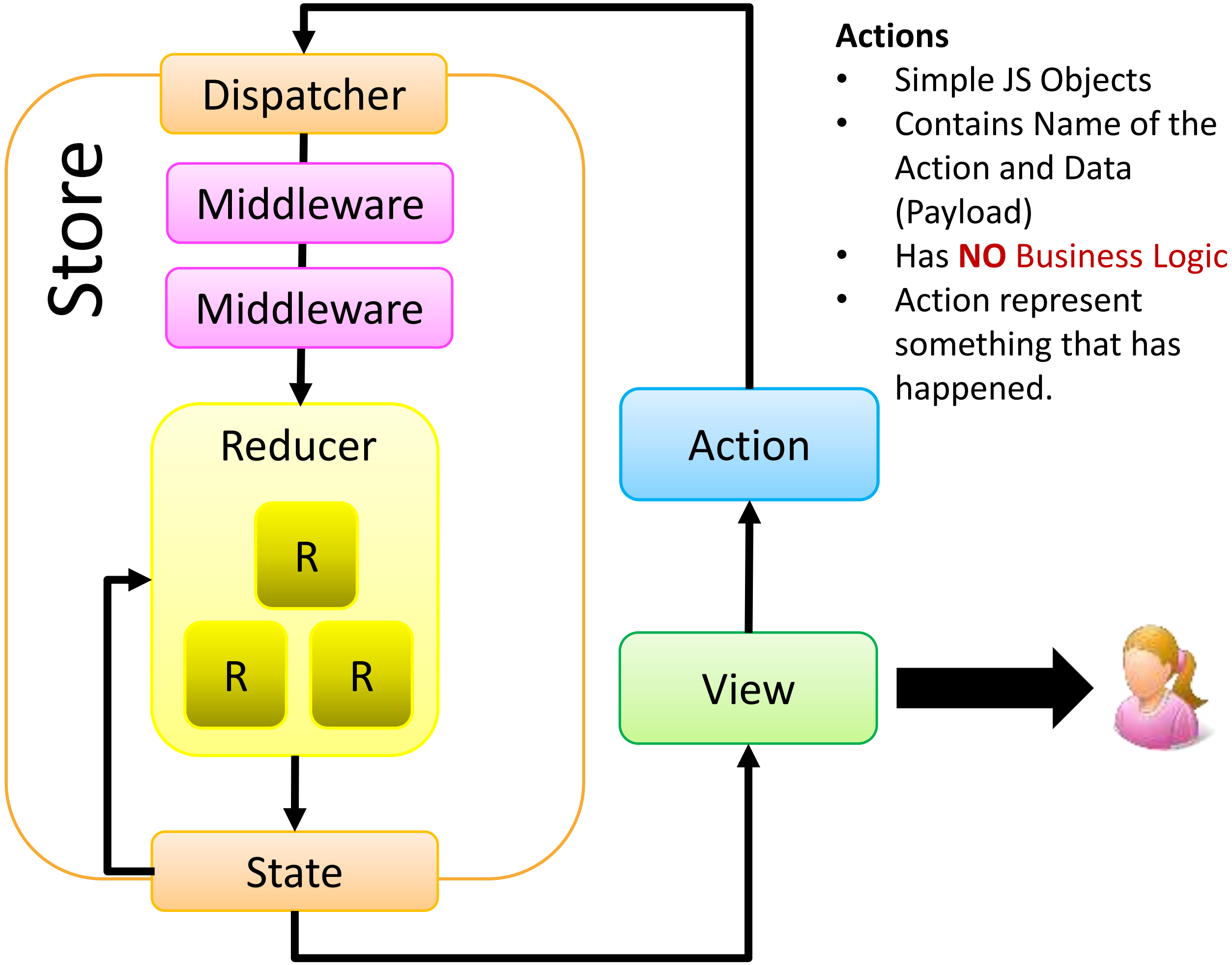
- Multiple View layers can **Subscribe**
- View layer to **Dispatch** actions
- Single Store for the Entire Application
- Data manipulation logic moves out of store to Reducers

Middleware

- Handles External calls
- Multiple Middleware's can be chained.

Reducer

- **Pure** JS Functions
- No External calls
- Can combine multiple reducers
- A function that specifies how the state changes in response to an Action.
- Reducer does **NOT modify** the state. It returns the **NEW State**.



Actions

- Simple JS Objects
- Contains Name of the Action and Data (Payload)
- Has **NO** Business Logic
- Action represent something that has happened.

UI Design Patterns Redux