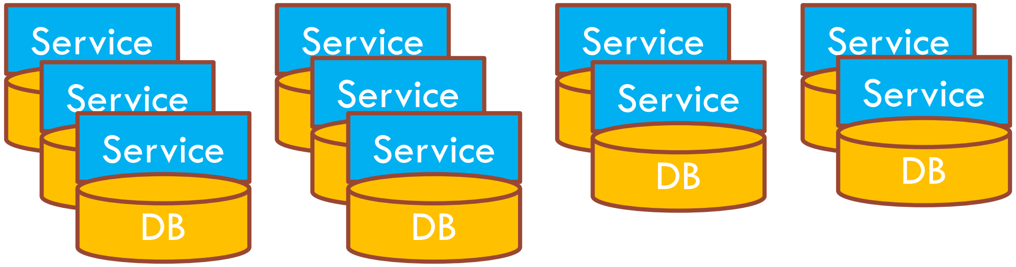
COMP 9322
Software Service Design and Engineering

Lecture 3 – Microservices Introduction

What Are Microservices

- Microservices are **small, autonomous** services that work together.
  - □ Architecture point of view: Distributed System
  - □ Organizational point of view: Mimics how the world works



Natura’s Point of View

Small

- How small?
  - Rule of thumb 1: something that could be rewritten in two weeks
  - Rule of thumb 2: if it is not too big, it is small enough
  - Rule of thumb 3: If the codebase is not too big to be managed by a small team
- They are focused on doing one thing
  - □ Cohesive

    To have all the related code together

    - Single responsibility principle (of Robert Martin):

    ‘Gather together those things that change for the same reason, and separate those things that change for different reasons’
  - □ Microservices apply cohesion to independent services:
    - Service boundaries = Business boundaries

Autonomus

- Isolate all things
  - An independent entity
    - Might be deployed as an isolated service
- Communicate with other services via network calls
  - Expose APIs – other services can communicate through APIs
- Loosely coupled
  - Can be changed independently and be deployed by themselves
- Encapsulate state as well as behavior
  - Own the state exclusively
    - Data is strongly consistent within each service
    - No more have a single consistent database for all services

Microservices – A better definition

- Services that are **loosely coupled, cohesive, act in isolation, autonomously** by **owning their state and behavior.**

Organizations and the Microservice

- Microservices are services modeled after a business domain
- Conwey's Principle:
  - Any organization that designs a system (defined more broadly here than just information systems) will inevitably produce a design whose structure is a copy of the organization's communication structure
- Information Systems Department of an Army:
  - How will the communication structure shape?
    - Command and control
  - Who will be the project manager?
    - The highest ranking officer
- A startup ? Will you give the same answers?

Why We Need Microservices

Applications of the Last decade	Applications of Today
Small number of concurrent users	Large number of concurrent users
Dedicated servers	Cloud based scalable servers
Slow networks	Fast networks
Deployed on a single machine	Deployed on a variety of environments
Seconds response time	Milliseconds response time
Small well defined data sets	Large unstructured data sets

- A single SQL database, consistent data on a single server is not suitable for today’s needs.
- Discrete services:
  - can be failed individually
  - can be scaled up/down easily
  - can be upgraded in isolation

Monoliths

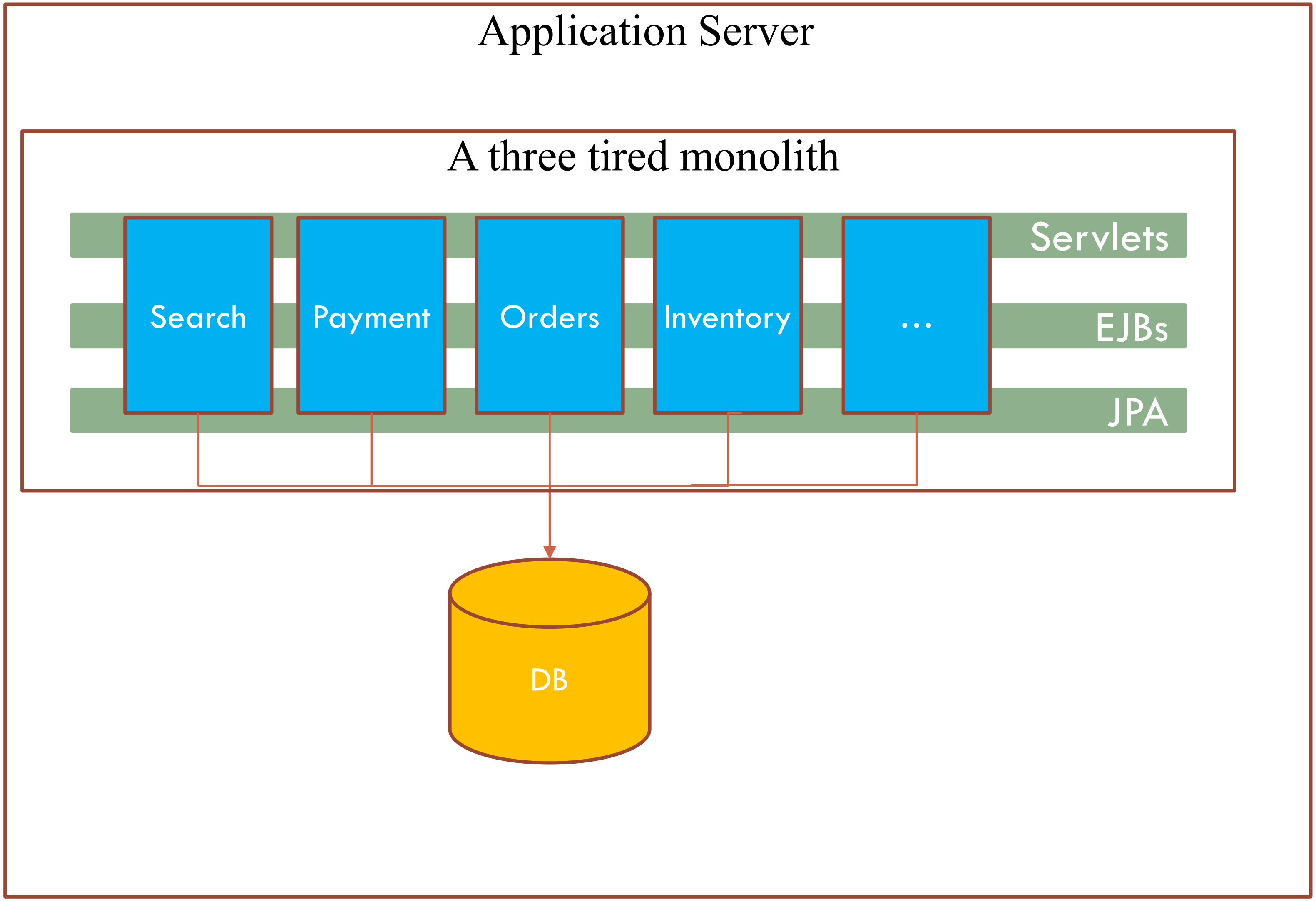
A geological feature consisting of a single massive stone or rock

Uluru is considered to be the biggest monolith on earth

We usually have Monoliths

- The term is used by Unix community to describe systems that get too big.
- They are built and maintained as a single unit.
- Difficult to scale up or down.
- Difficult to change.

Monolithic Application Server



What is Good with this Design

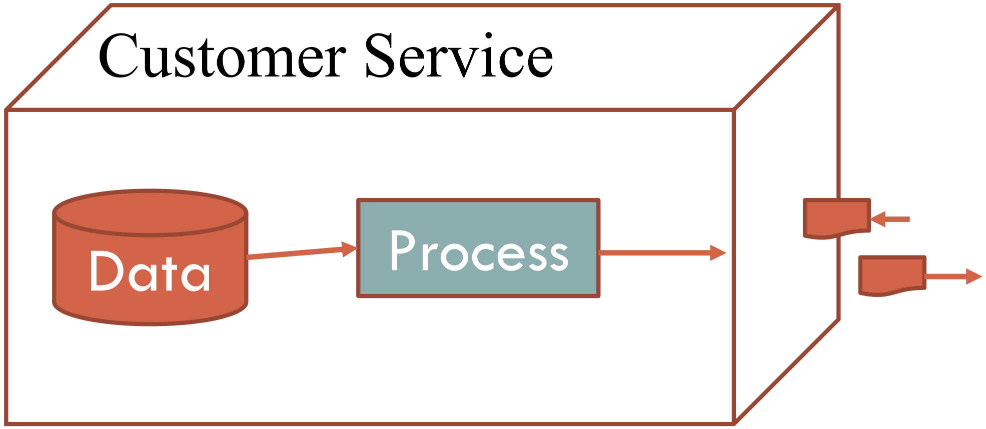
- All the logic for handling a request runs in a single process.
- You can decompose the application into classes and functions using the features of your programming language.
- The communication(network) overhead is minimal.
- You can run and test on your laptop.
- Interface, business logic and data base tiers can be changed independent from each other.
- You can horizontally scale up by running many instances of the whole process.

What is not so Good with this Design

- Integration through Database
  - Can not accommodate different data storage requirements of different services
- Have strong coupling
  - Workflow logic is implemented by synchronous method calls
- Tight coupling results:
  - All services are required to be upgraded together.
  - Failure in a service means failure of the system.
  - Scaling can not be done at service level.
    - You must scale the whole application with all the services.

Creating the Solution

- Creating a single Microservice is not a difficult task.



- Microservices need to communicate and collaborate to solve larger problems.
- The most critical aspect:
  - □ is managing the space between microservices

Communication Latency

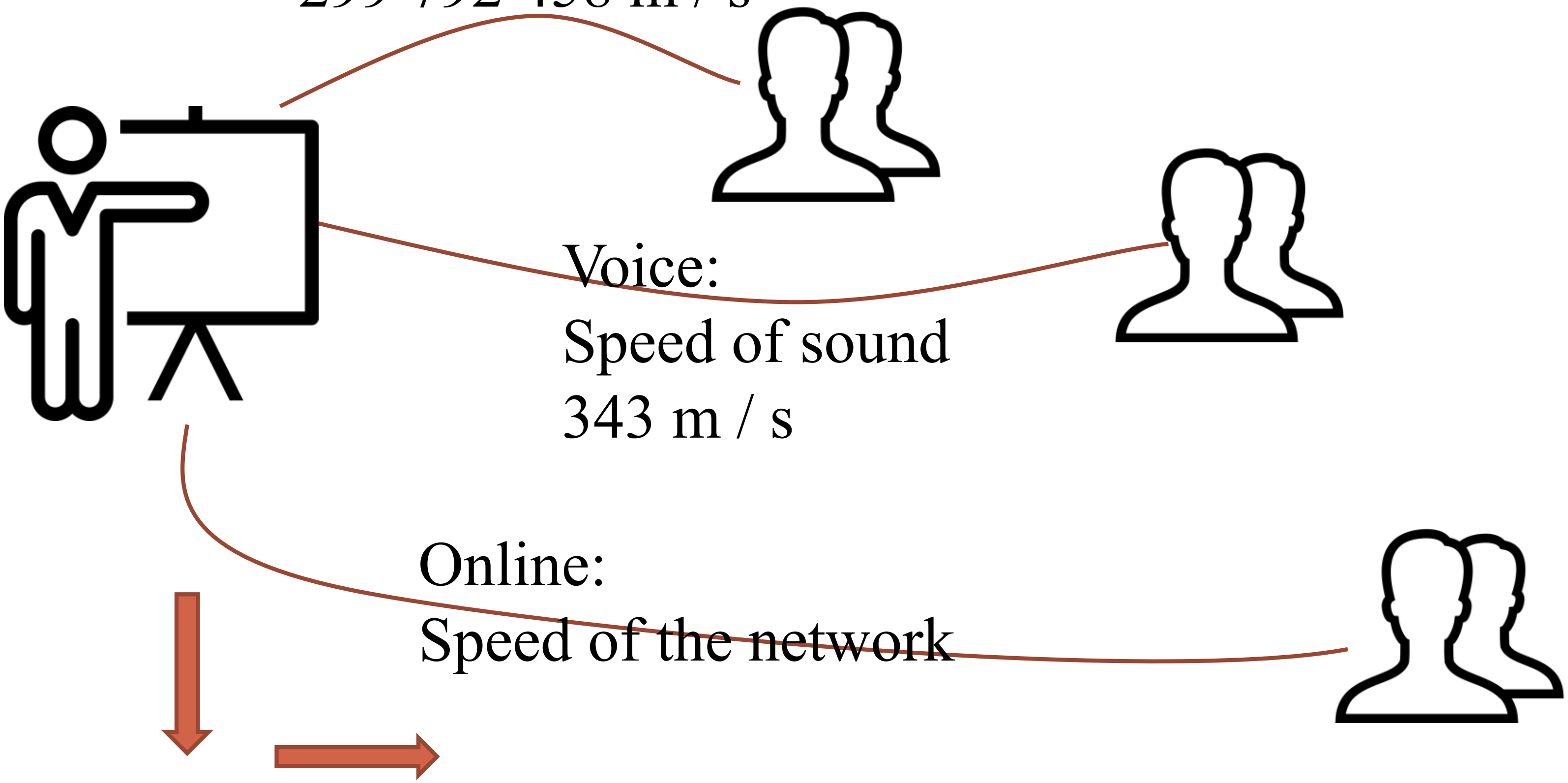


Image:
Speed of light
299 792 458 m / s

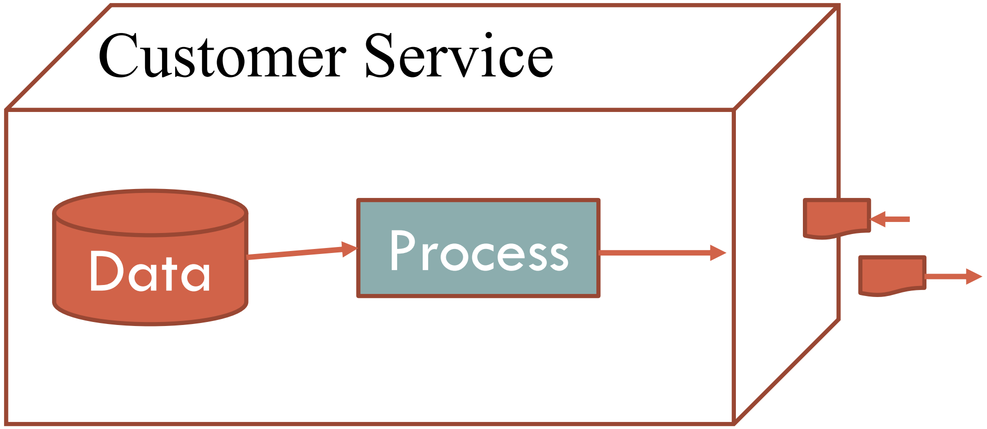
Voice:
Speed of sound
343 m / s

Online:
Speed of the network

Now

Then

Data Inside and Data Outside



Inside Data:

- Using ACID transactions (Atomic, Consistent, Isolated and Durable)
- Clear sense of ‘now’
- Data is in ‘Now’

Outside Data:

- Messages contain data extracted from ‘Customer Service’.
- By the time ‘Receive Order’ have seen it could have been changed.
- Data is from ‘Past’
- There is no notion of ‘Now’ between services
- Services should meet ‘Now’ with ‘Then’

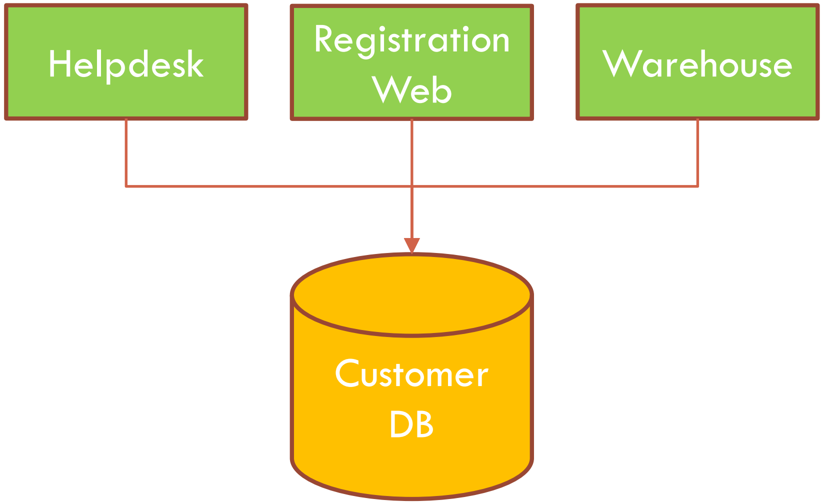
How Microservices Communicate and Collaborate

- The Shared Database
- Synchronous or Asynchronous
- Orchestration or Choreography
- Asynchronous Event Based Collaboration

A Sample Case

- Our organization would like to keep information related with customers that will be used by Helpdesk, Registration Website and Warehouse.
- Information related with customers will be created, updated, deleted and listed as needed.

The Shared Database

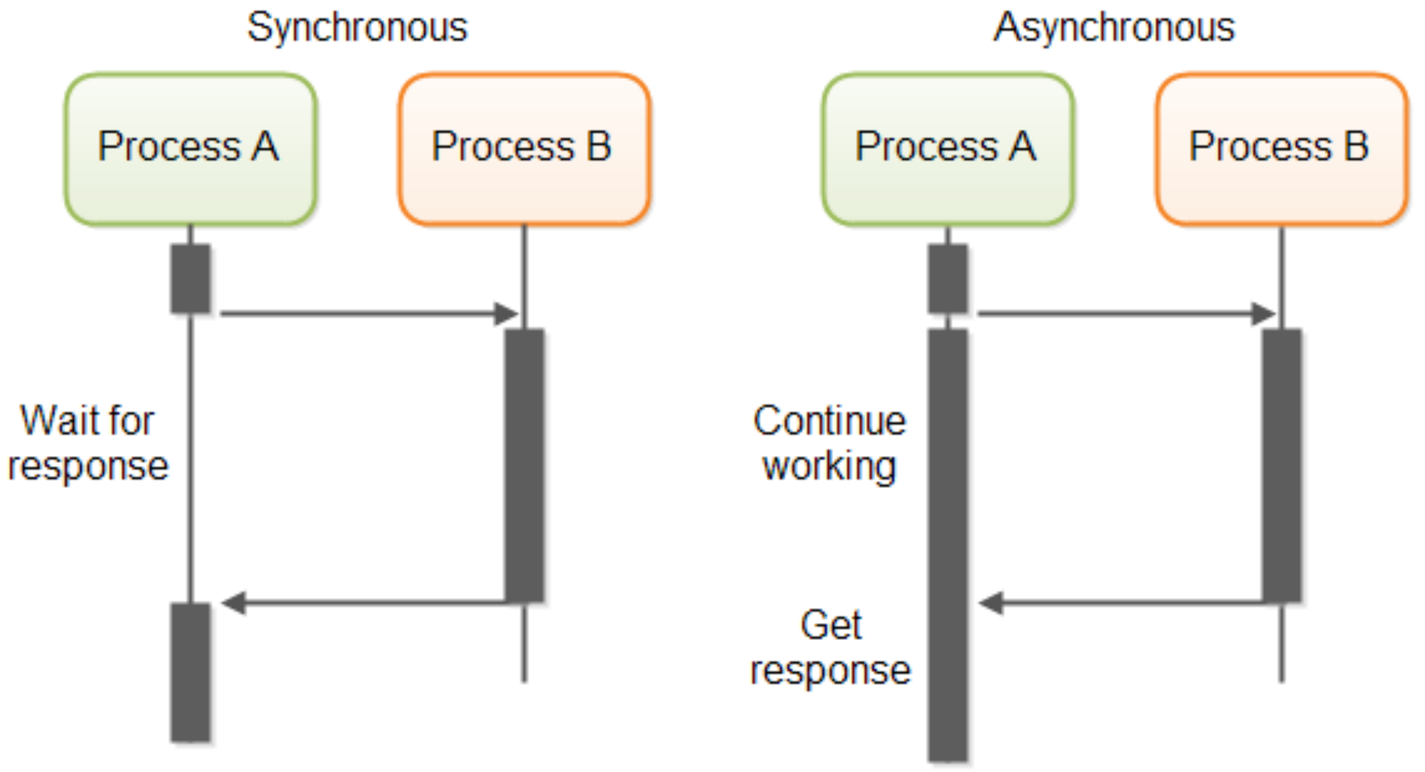


- In this solution:
  - □ The Registration UI creates customers using SQL.
  - □ Call center application views and edits customer info by SQL.
  - □ Warehouse updates customer orders by querying the DB.
- What is the problem ?

The Problems

- External parties bind to implementation details.
  - □ The data structure should be shared
  - □ If we change the structure to improve efficiency all users will be effected.
  - □ If warehouse would like to change its application it should need to make sure that it does not effect other applications.
  - □ We need extensive regression testing after all changes
- The applications are tied to selected technology.
  - □ Helpdesk by itself can not change from a relational DB technology to a different one.
  - □ It is not loosely coupled anymore.
- The business logic related to operations on Customers need to be spread.
  - □ If you like to change the way of new customer creation it should be changed in Registration Web and Helpdesk.
- The result: Avoid change in any costs !

Synchronous and Asynchronous Communication



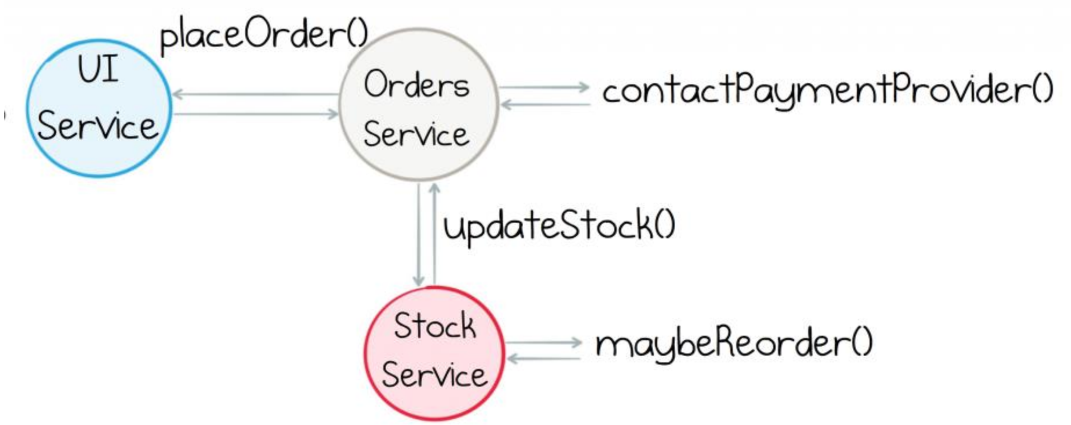
Synchronous

- More suitable:
  - where real-time interaction with minimal delays is needed,
  - where subsequent actions are dependent on the response received for the previous message transferred,
  - further actions need to be performed in sequential manner.
- Example:
  - ATM machine need to interact with the backend system to check the available balance.

Asynchronous

- More suitable:
  - □ where systems have long running jobs and there is no need of real-time responses.
  - □ when you need low latency – blocking a call may slow the system
- Example:
  - □ An ERP system needs to publish some information so that any interested parties can subscribe to that and get the updates.

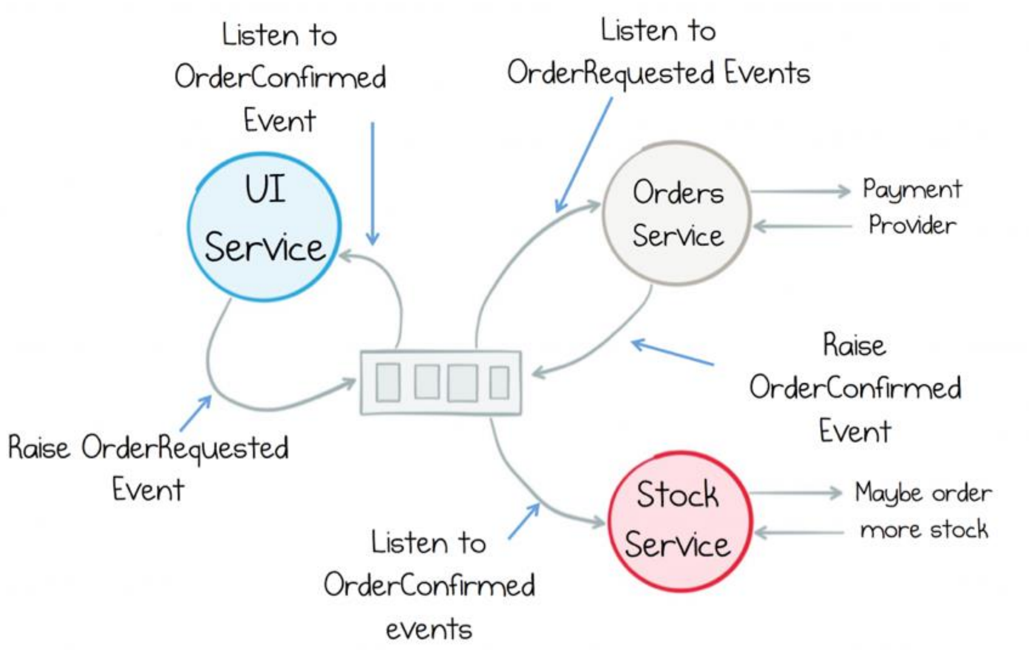
Request/Response Collaboration



1-Customer orders an item
2-Payment is processed
3-The system check the availability and the need for reorder

- Well aligned with synchronous communication
- For asynchronous applications adaptation is required:
  - □ Start the operation
  - □ Register a call back
    - ask server to notify when the operation complete

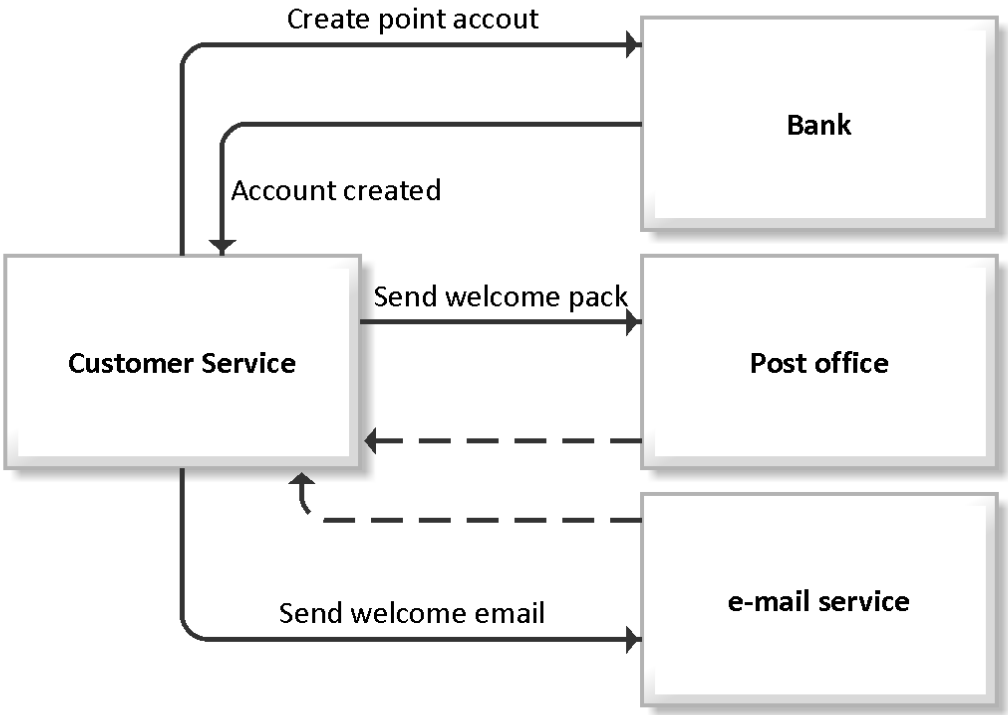
Event based collaboration


Listen to OrderConfirmed Event
Listen to OrderRequested Events
UI Service
Orders Service
Payment Provider
Raise OrderConfirmed Event
Raise OrderRequested Event
Stock Service
Maybe order more stock
Listen to OrderConfirmed events

- The UI Service raises Order-Requested event
- Orders Service and the Stock Service react to the raised event.
- Order service raise Order-Confirmed event
- UI Service reacts to Order-Confirmed

- Process announce what happened
- Other services decides what to do
- Business logic is distributed
- Highly decoupled – can add new services easily.

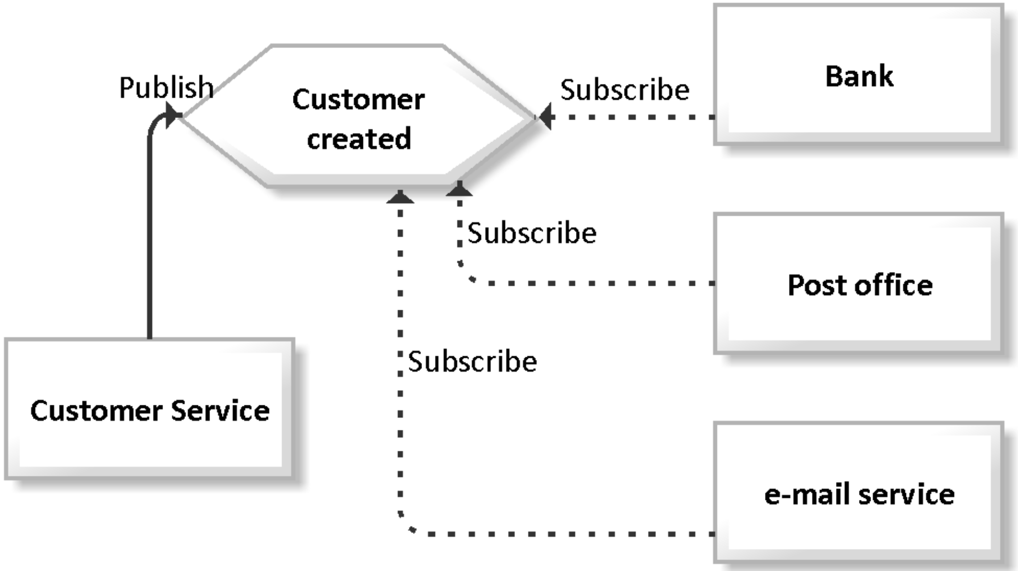
Orchestration



- □ Create a central control mechanism within:

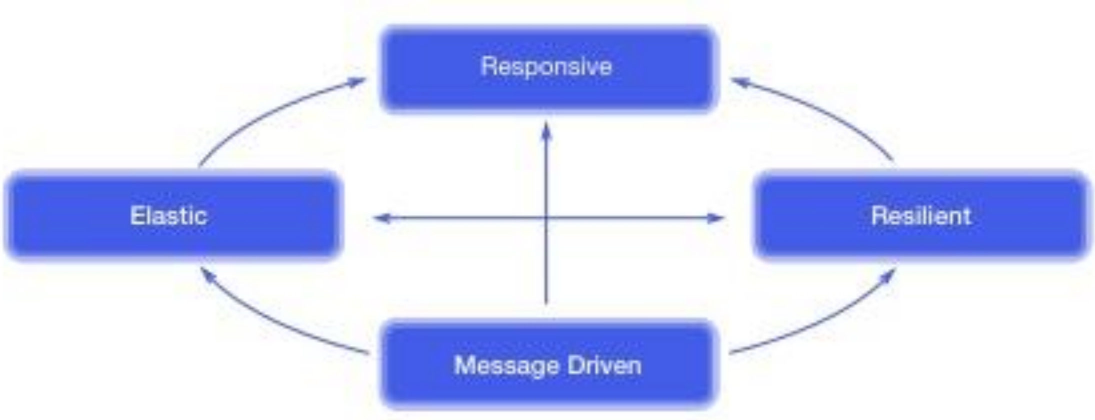
  CustomerService
- □ Once the process initiated CustomerService send request to other services.
- □ We can model into code or use BPM software.
- □ - Tightly coupled
- □ - High cost to change
- □ + Can monitor the status of the process.

Choreography



- Customer Service created the event.
- All services subscribe to this event react to it.
- + Loosely coupled
- + Easy to change
- - Additional work is needed to monitor the status of the process.

Reactive Systems



- Systems that are* :
  - □ Responsive,
  - □ Resilient,
  - □ Elastic and
  - □ **Message Driven**
    - Asynchronous, nonblocking message-passing that establish a boundary between components…
    - that ensures loose coupling, isolation and location transparency.

* The Reactive Manifesto

References

- Jonas Boner, Reactive Microsystems, 2017, Lightbend, Inc.
- Jonas Boner, Reactive Microservices Architecture, 2016, O’Reilly Media.
- Antifragile Software: Building Adaptable Software with Microservices, Russ Miles, 2016, Leanpub
- Newman, Sam, Building Microservices, 2015, O'Reilly Media.
- Vaughn Vernon, Domain Driven Design Distilled, 2106, Addison Wesley.

Useful Resources

- https://www.finextra.com/blogposting/16153/api-orchestration-or-choreography-whats-your-choice
- https://microservices.io/
- https://apifriends.com/api-creation/api-orchestration/