# Software Architectures

## Microservices

41492 – Engenharia de Software, Nuno Sá Couto e Rafael Direito

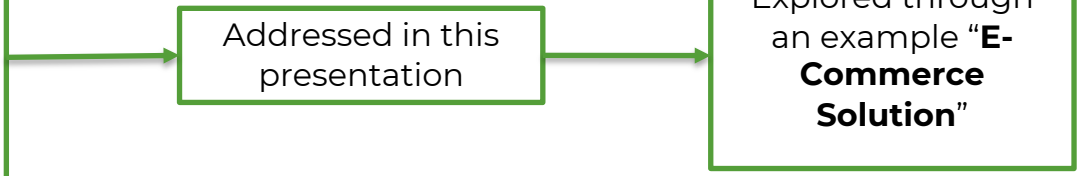
October 2nd 2023

# Agenda

01 Microservices – Architectural Process

# Microservices - Architectural Process

1) Understand functional Requirements
2) Understand the non-functional Requirements
3) **Map the Components**
   - **a) Components' Mapping**
   - **b) Communication Patters**




4) Select the Technology Stack
5) Designing the Services
6) Write Architecture Document
7) Create System Backlog and User Stories

# Components' Mapping

- The **most important** step of the whole process
- **Should be taken very SERIOUSLY**
- Once set, is difficult to change
- **Each component will be mapped to a Service**
- **Mapping should be based on:**
    1. Business Requirements
    2. Functional Autonomy
    3. Data Entities
    4. Data Autonomy

## Business Requirements

- **Collection of requirements around a specific business capability**
- **Examples:**
  - System must allow the adding, update, and removal of new items
  - Systems should allow the registering of new orders
  - System must provide the quantity of items in inventory
  - System must process payments

## Functional Autonomy

- **The maximum functionality that does not involve other business requirements/services**
- **Examples:**
  - The developed service **should** retrieve all orders placed in a certain week
  - The service **won't** get all orders placed by under-aged clients **(because it doesn't have the autonomy to do so)**
  - **Although, gray areas will always exist! It is almost impossible to design a system where all Services are fully isolated and autonomous!**

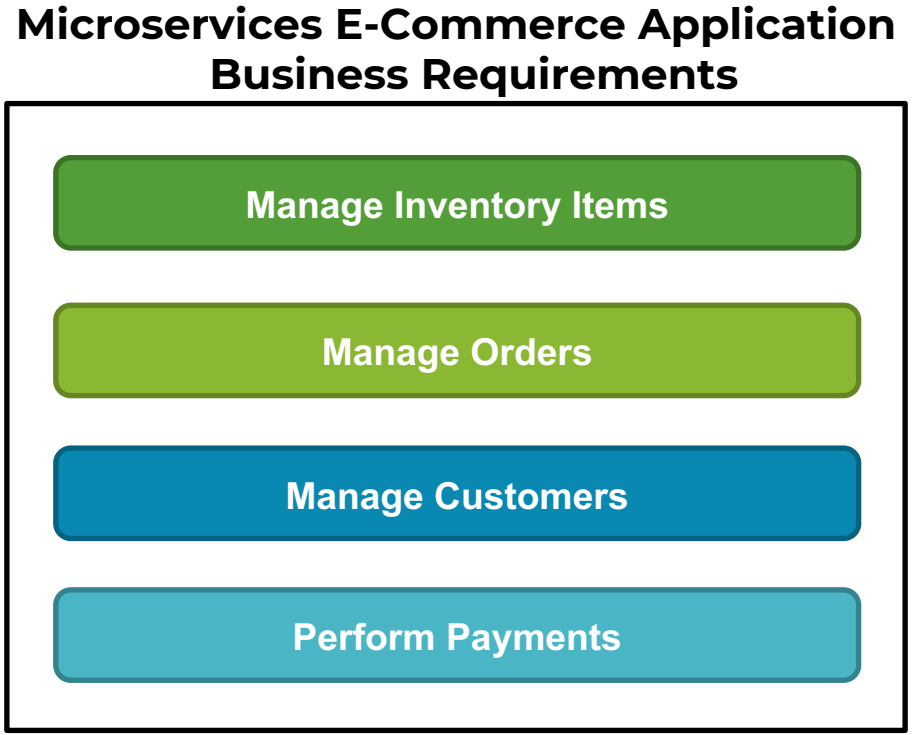
## Data Entities

- **A Service should be designed around a well-specified data entity**
- **Data can be related to other Services but just by ID**
- **Examples:**
  - We **identify the major entities** that will exist in the system, like orders and items, and **map all the components around them**
  - The “orders” component stores every information related to the orders
  - But… it needs to know which client placed which order. So, it also stores the client’s ID
- In real life there is almost always relationships between entities

## Data Autonomy

- **Each Service must be fully autonomous with only the data it manages. A Service should not depend on data from other Services**
- If our Service depends on data from other Services it is a sign that we missed the architecture design
- **Examples:**
  - **Employees Service that relies on Addresses Service to return employee’s data – Wrong!**
  - Include Address data in the Employees service – **Solution!**

# Components' Mapping - Example



# Components' Mapping - Example

| Business Requirements | Manage Inventory Items | Manage Orders | Manage Customers | Perform Payments |
|---|---|---|---|---|
| **Functional** | Add, Remove, Update, Quantity | Add, Cancel, Compute Sum | Add, Remove, Update, Get Account Details | Make Payment |

# Components' Mapping - Example

| Business Requirements | Manage Inventory Items | Manage Orders | Manage Customers | Perform Payments |
|---|---|---|---|---|
| **Functional** | Add, Remove, Update, Quantity | Add, Cancel, Compute Sum | Add, Remove, Update, Get Account Details | Make Payment |
| **Data Items** | Items | Orders, Shipping Address | Customer, Address, Contacts | Payments |

# Components' Mapping - Example

| Business Requirements | Manage Inventory Items | Manage Orders | Manage Customers | Perform Payments |
|---|---|---|---|---|
| **Functional** | Add, Remove, Update, Quantity | Add, Cancel, Compute Sum | Add, Remove, Update, Get Account Details | Make Payment |
| **Data Items** | Items | Orders, Shipping Address | Customers, Addresses, Contacts | Payments |
| **Data Dependencies** | None | Inventory Service: Items ID Customer Service: Costumers ID | Orders Service: Order ID | (Orders Service: Orders ID) |

# Components' Mapping - Example

| Business Requirements | Manage Inventory Items | Manage Orders | Manage Customers | Perform Payments |
|---|---|---|---|---|
| **Functional** | Add, Remove, Update, Quantity | Add, Cancel, Compute Sum | Add, Remove, Update, Get Account Details | Make Payment |
| **Data Items** | Items | Orders, Shipping Address | Customers, Addresses, Contacts | Payments |
| **Data Dependencies** | None | Inventory Service: Items ID Customer Service: Costumers ID | Orders Service: Order ID | (Orders Service: Orders ID) |

- **Real life cases are much more complicated than this**. You will face what is called "Edge Cases" that you must consider.

- We will discuss two common cases and how to deal with them!

# Components' Mapping - Example

- **Edge Case:** Retrieve all customers from Lisbon with the total number of orders for each customer
- **Data Dependencies:**
  - The **Customers Service** stores the address of the customers
  - The **Orders Service** stores all orders placed by a client

**How can we solve this ?**

# Components' Mapping - Example

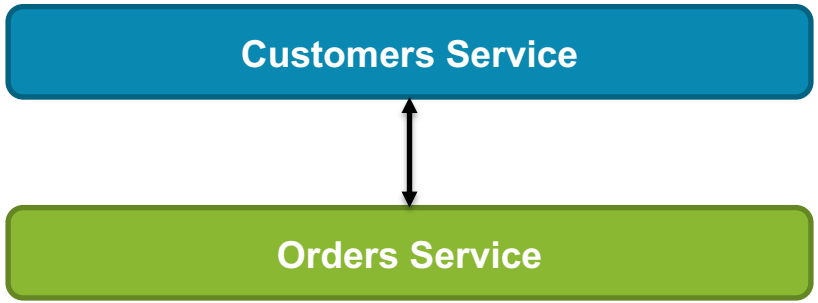
- **Edge Case:** Retrieve all customers from Lisbon with the total number of orders for each customer
- **3 Approaches:**
    1. Data Duplication
    2. Service Query
    3. Aggregation Service

# Components' Mapping – Example – Data Duplication

- **We could also store orders' information in the Customers Service Database**
  - Whenever an order is created or canceled this must be updated also on the Customer service database
  - The data is then duplicated and appears twice
  - **Problem:** Sync between databases although the update is quite straightforward

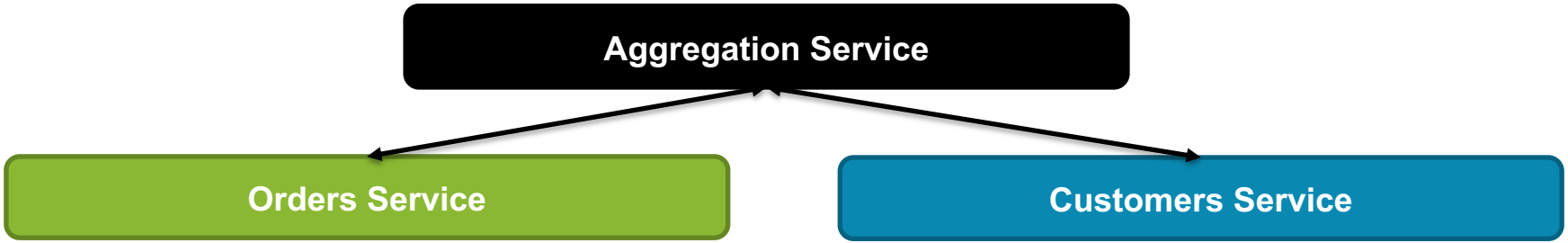
# Components' Mapping – Example – Service Query

- **We could have the Customers Service talking to the Orders Service**
  - Instead of creating the data on the two services, the services talk to each other
  - This means that when the Customer data is retrieved the Customer services goes to the order service and fetches all the order history (number of orders) related to that Customer ID
  - **Problem:** Loss of performance! Inter-Service communication takes time!



# Components' Mapping – Example – Aggregation Service

- **Another possibility is having another service that aggregates the results of each service query**
  - This Service queries the Customers Service and then queries the Orders Service with the result of the Customers query
  - The results merging is done by an additional external service
  - **Advantages:** Services remain fully isolated and don't talk to each other
  - **Problem:** Loss of performance! Inter-Service communication takes time!



## Components' Mapping - Example

# Pick Your Poison

- In **groups of 2 students**, discuss **what would be the best approach** to this edge case
- **Explain why** you have selected that approach!
- **You have 1 min and 30 seconds!**

# Components' Mapping – Example

- **Personal Recommendation: Data Duplication**
- It is **not a perfect solution** and **it has its own flaws** but it is better than the other two for this specific case:
  - Service Query has a serious potential for performance problems
  - Aggregation requires additional service and coding for a quite small requirement
  - We are dealing with very little data and data replicated into customers database in read only be this service (avoiding sync issues)

# Components' Mapping – Cross-Cutting Services

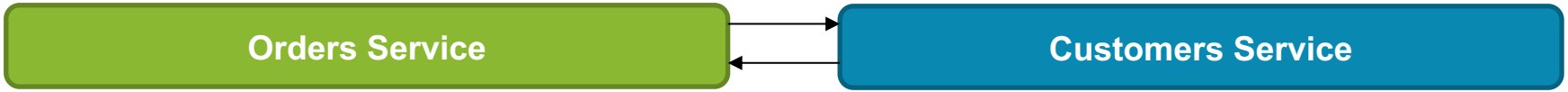
- **Services that provide system-wide utilities**
- Common examples
  - Logging
  - Caching
  - User Management (AAA)
- **MUST BE PART OF THE MAPPING!**
- Are the first Services being developed!

# Communication Patterns

- We need to establish communication between the different Services
- **It’s of utmost importance to choose the correct Communication Pattern**
- **Communication Patterns:**
    1. 1-to-1 Sync
    2. 1-to-1 Async
    3. Pub-Sub / Event Driven

# Communication Patterns – 1-to-1 Sync

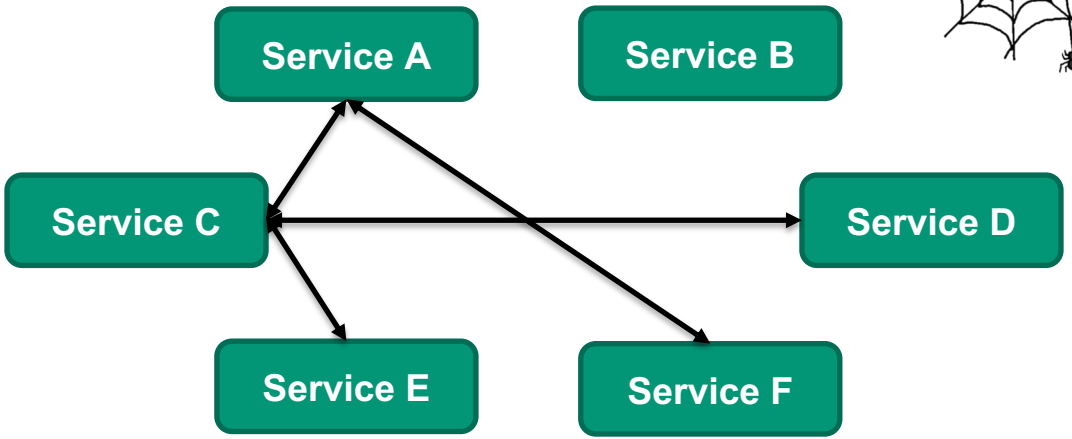
- **A Service calls another Service and waits for the response**
- Used mainly **when the first Service need the response** to continue processing




| Pros | Cons |
| --- | --- |
| • Immediate response<br>• Error handling<br>• Easy to implement | • Blocking request (it might take hours...)<br>• Loss of Performance<br>• May result in a spider web |

# Communication Patterns – 1-to-1 Sync

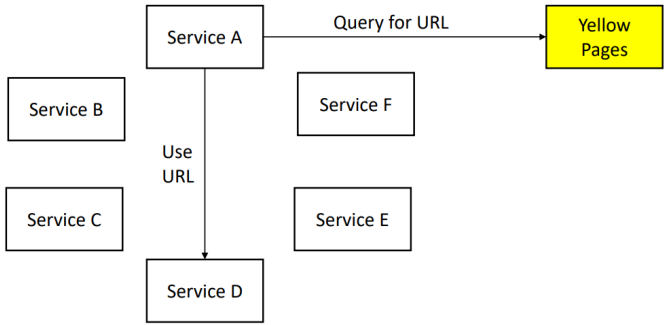
- **Problem: We may face a Spider Web!**




**If a Service's API has changed, all services will be affected. The same happens if we redeploy a Service in a new location!**

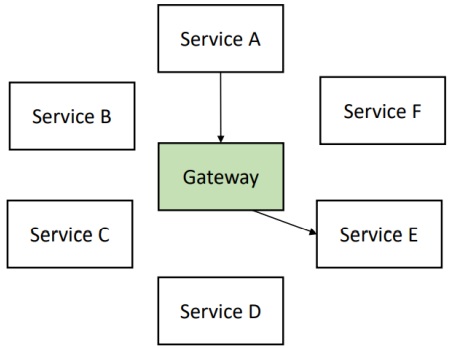
# How to Solve the Spider Web?

## Service Discovery




**Services only need to know the Service Registry URL**

## API Gateway




**Services interact through the API GW**

**Offers more services (Monitoring, AAA). Most popular choice nowadays**

# Communication Patterns – 1-to-1 Async

- **A Service calls another Service and continues working**
- Used mainly **when the first Service wants to pass a message** to the other Service


| Pros | Cons |
| --- | --- |
| • Non-blocking request<br>• **Better performance** | • More setup is required<br>• Difficult error handling |

# Communication Patterns – 1-to-1 Async


- The Orders service **sends the message to Queue**
- **Queue** then **relays the message** to Payments' service
- **The Order's Service flow finishes when the queue receives the message.** It does not care about what happened to the message afterwards: **Queue's Responsibility**!

# Communication Patterns – Pub-Sub / Event Driven

- **A Service notifies another Services (PLURAL!) and continues working**
- Used mainly **when the first Service wants to pass a message** to the other Services
- The service has **no idea how many services listen** / will receive msg
- Used mainly when the **first service wants to notify about an important event in the system**

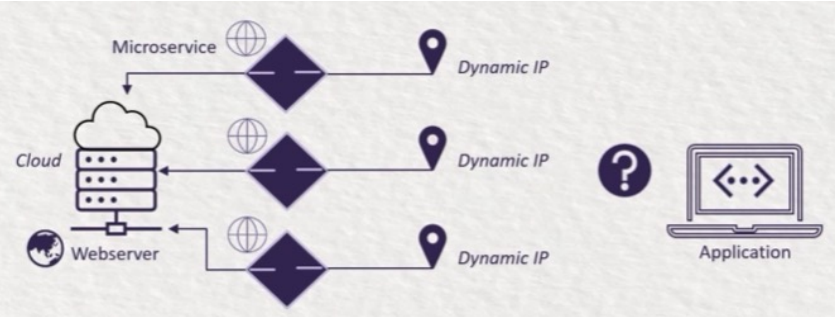
| Pros | Cons |
| --- | --- |
| • Non-blocking request<br>• **Better performance**<br>• **Notify multiple Services at once** | • More setup is required<br>• Difficult error handling<br>• May increase the system's load (CPU, RAM, Network) |

# Communication Patterns – Pub-Sub / Event Driven

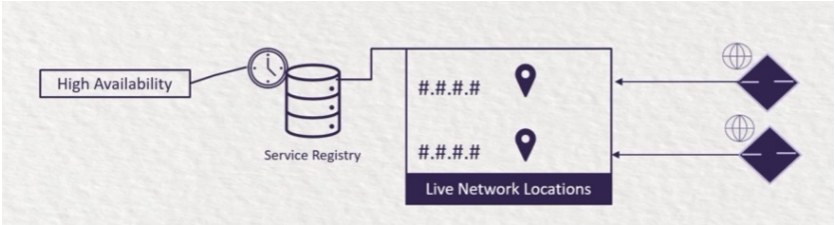

- **Like Async,** existence of a **middleman** agent between services
- **Order Service send message to Pub-Sub that publishes message to subscribers** (other services)
- **Order service does not have any idea about how many services subscribe to that message** (doesn't really care, it is the principle behind Async / Pub-Sub models)

# Communication Patterns

- **Service Registry and Discovery**
  - Key Concepts of Microservices
  - (majority) Cloud-based deployments have assigned network location details dynamically

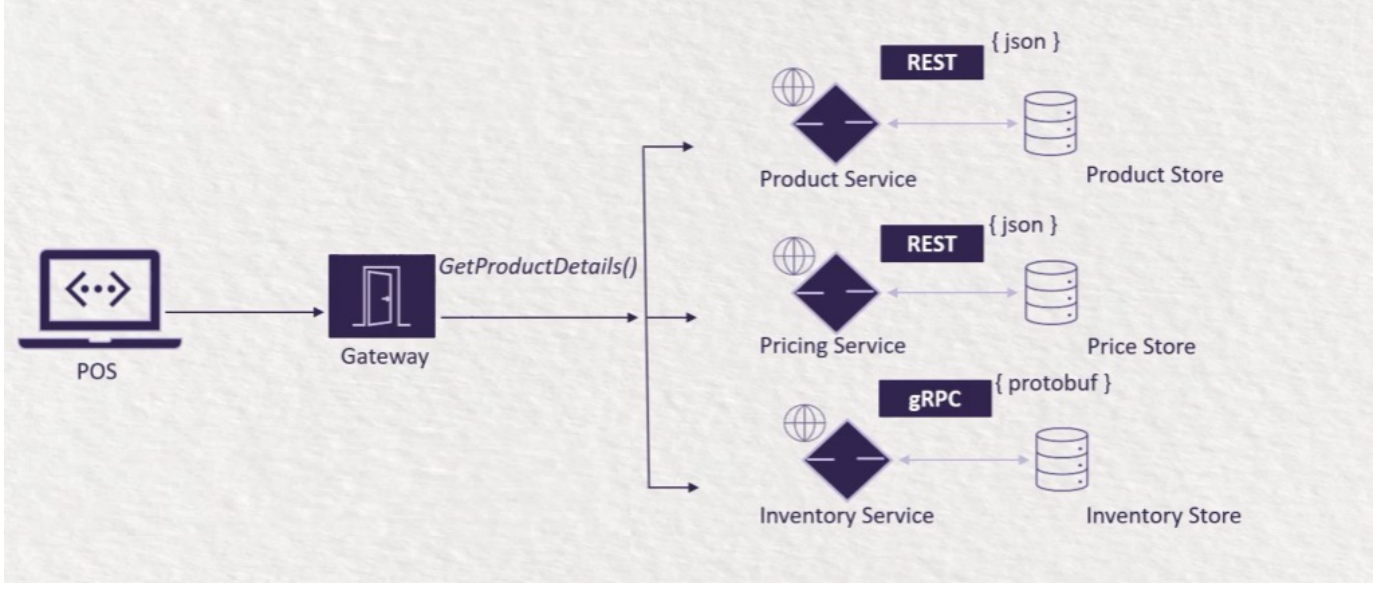



- **Service Registry is a solution for service Discovery**
  - Registry is auto-populated (registered) and
  - Updated (unregistered or state updated) when
  - Services start or terminate



# Communication Patterns

- **API Gateway**
  - API Gateways are light-weight middleware components that follow a consumer-centric model that focus more on end-client requirements.
  - It provides a single common entry point for all available microservices



# Communication Patterns

- **API Gateway - Features**
  - **Discovery**

    - Uses Service Discover patterns

    - Client and Service Side Discovery and make an effective use of Service Registry

  - **Protocol Conversion and translation**

    - Enables the establishment of communication between clients and microservices that use different protocols

    - Clients do not have to know which protocol a microservice uses

# Communication Patterns

- **API Gateway - Features**
  - **Client Specific API**

    - Application need vary by device type

    - Provides the best suited API for device type (web, mobile)

      *- Back-end for front-end pattern*

  - **API Composition for Distributed Transactions**

    - Handles data from multiple service calls using the concept of API composition

    - Aggregates the responses from microservices and the results to the client

# Communication Patterns

- **API Gateway – Identity and Access Management (IAM)**
  - Allow configuration of a variety of Identity Providers to perform Authentication & Authorization
  - Besides using internal IAM mechanisms, they also enable integrations with other Directory services and social identity providers (Amazon, Facebook) for authentication

    **- Identity Federation:** allow users to sign-in using their social media credentials.

    **- External Party:** Authentication
    **- Service Provider**: Authorization