

# Enterprise architecture patterns

UA.DETI.IES - 2022/23

# Recap

---

1. Software development process
  - Different models (sequential, incremental, evolutionary, ...)
2. Agile development methods
  - Agile principles and project management
3. DevOps Technical benefits
  - Continuous integration and software delivery



Team manager



Product owner



Architect



DevOps master

# Recap

---

1. Software development process
  - Different models (sequential, incremental, evolutionary, ...)
2. Agile development methods
  - Agile principles and project management
3. DevOps Technical benefits
  - Continuous integration and software delivery



Team manager



Product owner



Architect



DevOps master

# Software architecture?

---

- ❖ Architecture is the **highest-level concept** of the expert developers.

"In most successful software projects, the expert developers working on that project have a shared understanding of the **system design**.

This shared understanding is called 'architecture'.

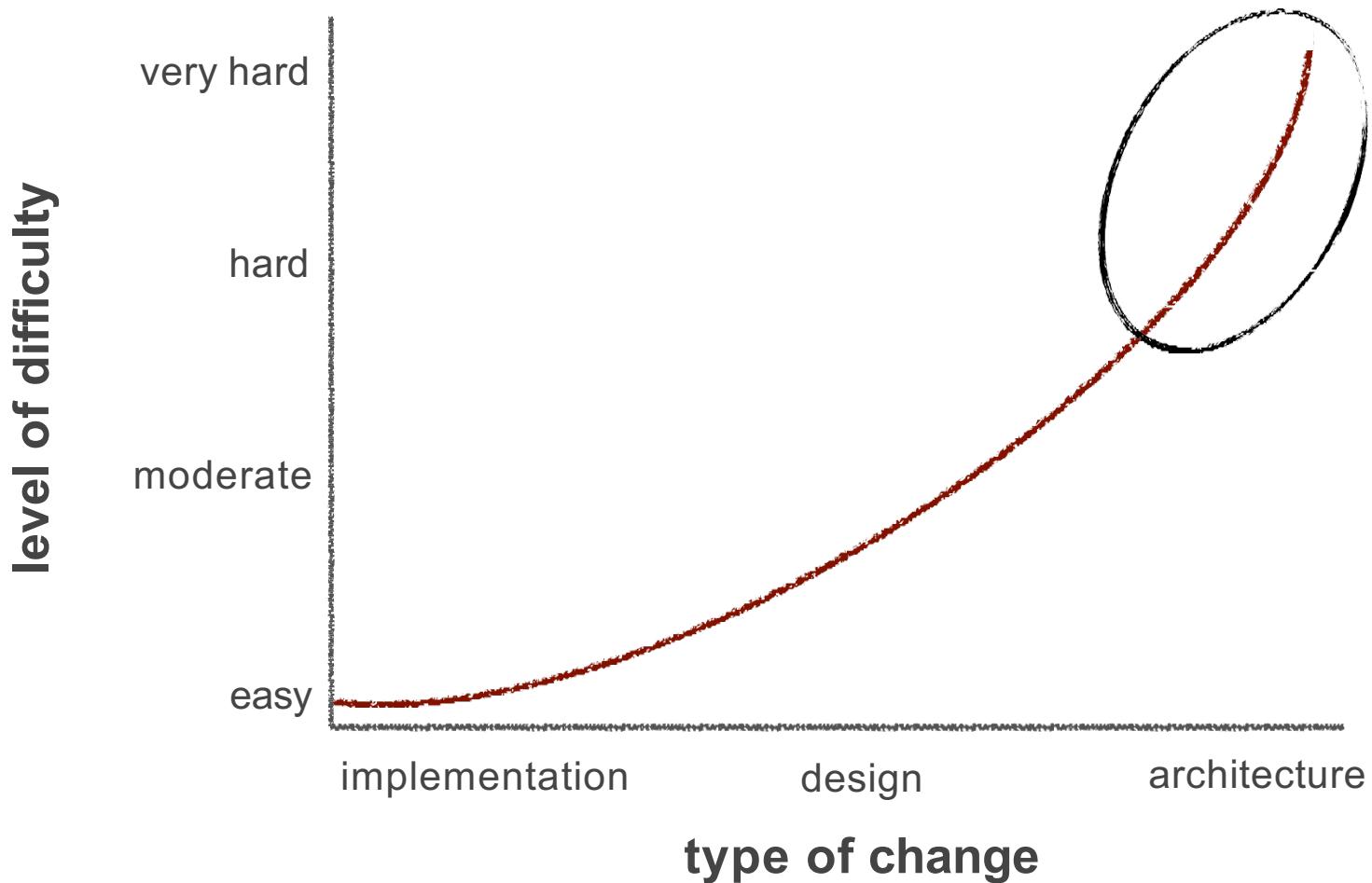
This understanding includes **how the system is divided** into components and how the components interact through interfaces.

These components are usually composed of smaller components, but the architecture only includes the components and interfaces that are understood by all the developers."

<http://martinfowler.com/ieeeSoftware/whoNeedsArchitect.pdf>

# Architecture decisions

---



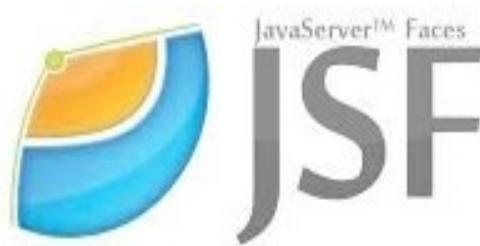
# Architecture decisions

---

- ❖ The decision to use a **web-based** user interface for your application

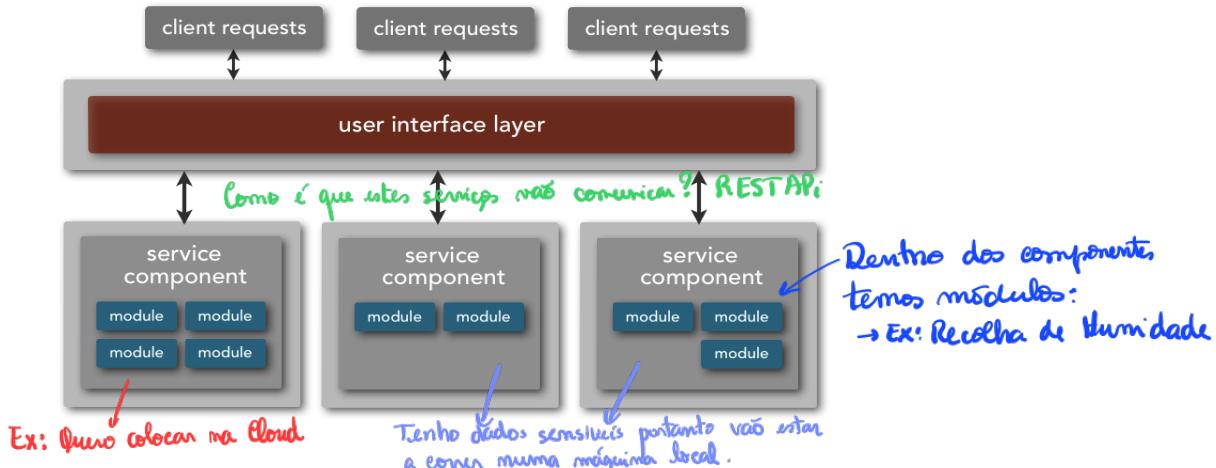


- ❖ The decision to use java server faces as your **web framework**



# Architecture decisions

- ❖ The decision that components should be **distributed** remotely for better scalability



- ❖ the decision to use rest to **communicate** between distributed components

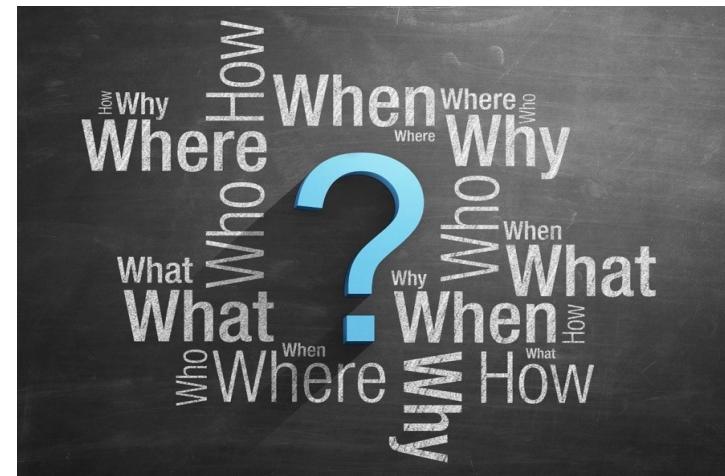


# Justifying architecture decisions

---

## ❖ **Groundhog day** anti-pattern

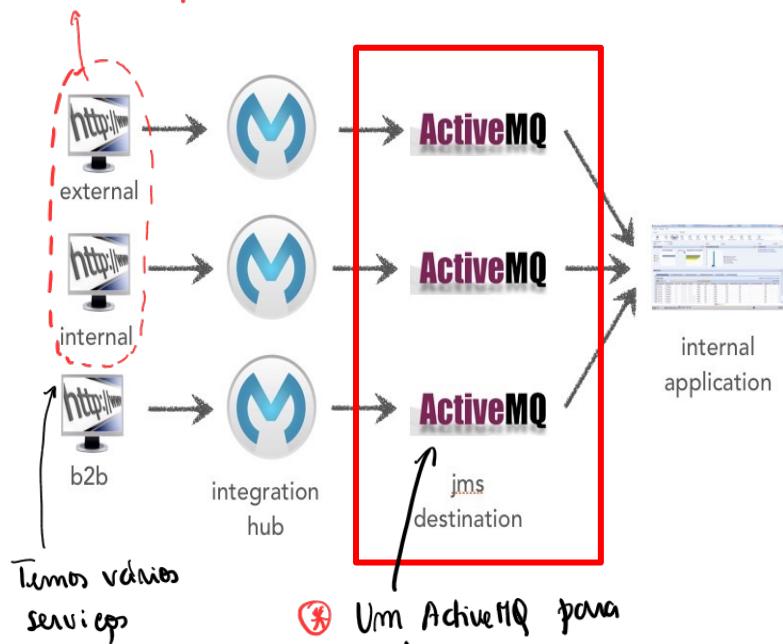
- Important architectural decisions that were once made get lost, forgotten or are not communicated effectively
- No one understands why a decision was made so it keeps getting discussed over and over and over...



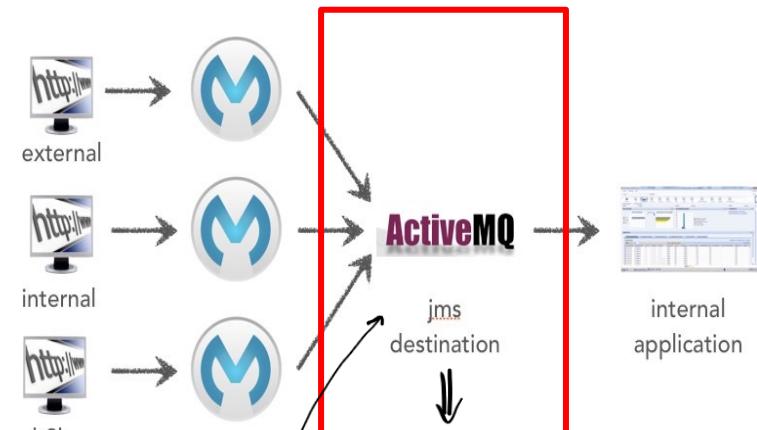
# Justifying decisions – Example

- ❖ Dedicated broker instances?

Ex: entre têm potencial para curar ponto óptimo por \*



- ❖ Or a centralized broker?



Ex: Sistema de Rega bastava isto  
(temos no máximo 20/30 válvulas)

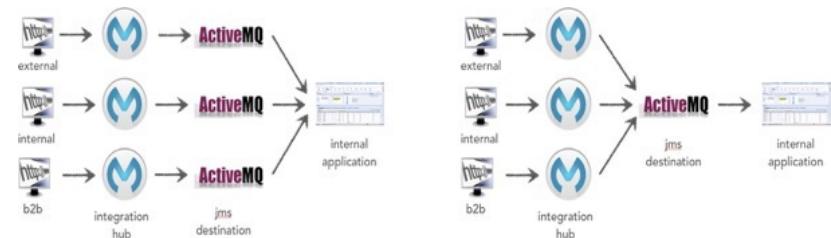
# Justifying decisions – Example

## ❖ Identify the conditions and constraints

- broker only used for hub access
- low transaction volumes expected
- application logic may be shared between different types of client applications (e.g., internal and external)

## ❖ Analyze each option based on conditions

- broker usage and purpose
- overall message throughput
- internal application coupling
- single point of failure
- performance bottleneck



## ❖ ... Solution?

# Justifying decisions – Example

---

## ❖ Architecture **decision**:

- centralized broker

## ❖ **Justification**:

- the internal applications should not have to know from which broker instance the request came from
- only a single broker connection is needed
  - allowing for the expansion of additional hub instances with no application changes.
- due to low request volumes the performance bottleneck is not an issue;
  - single point of failure can be addressed through failover nodes.



# Documenting and communicating

---

- ❖ Establish early on where your decisions will be documented and **make sure every team member knows** where to go to find them
  - In a central document or wiki, rather than multiple files spread throughout a crowded shared drive
- ❖ Email-driven solutions (... not ☺) *✓ Email não é um repositório*
  - people forget, lose, or don't know an architecture decision was made
  - therefore, don't implement the architecture correctly

# Avoid Witches brew architecture

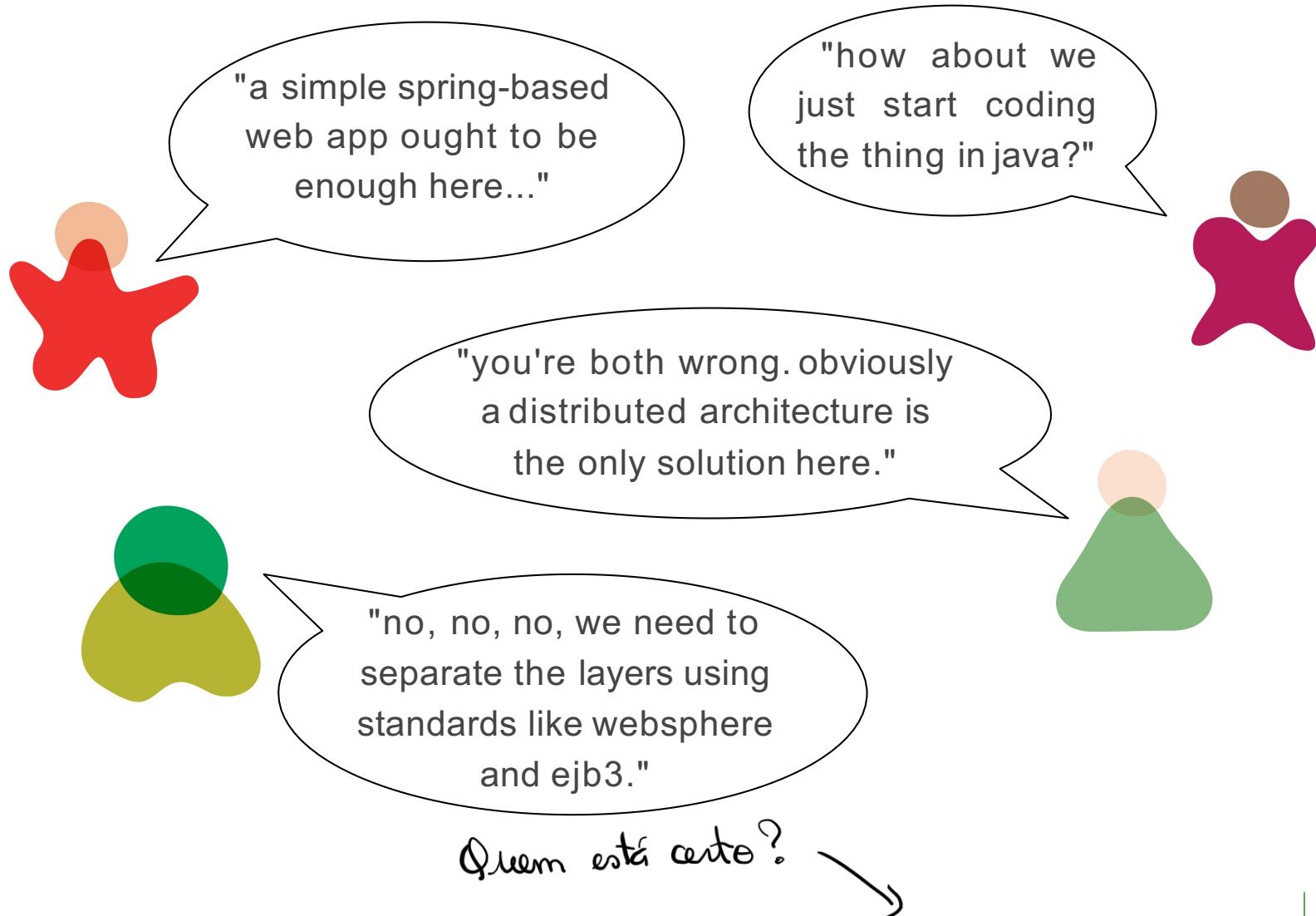
↳ mistakes...

- ❖ Architectures designed by groups resulting in a complex mixture of ideas and no clear vision



# The problem

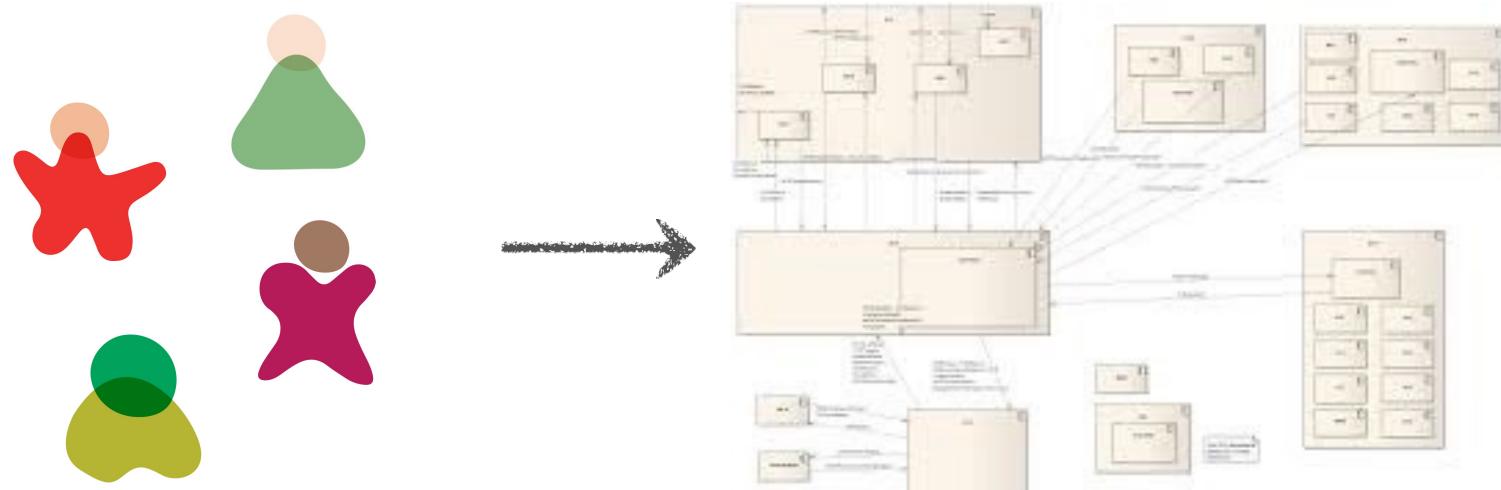
---



# The goal

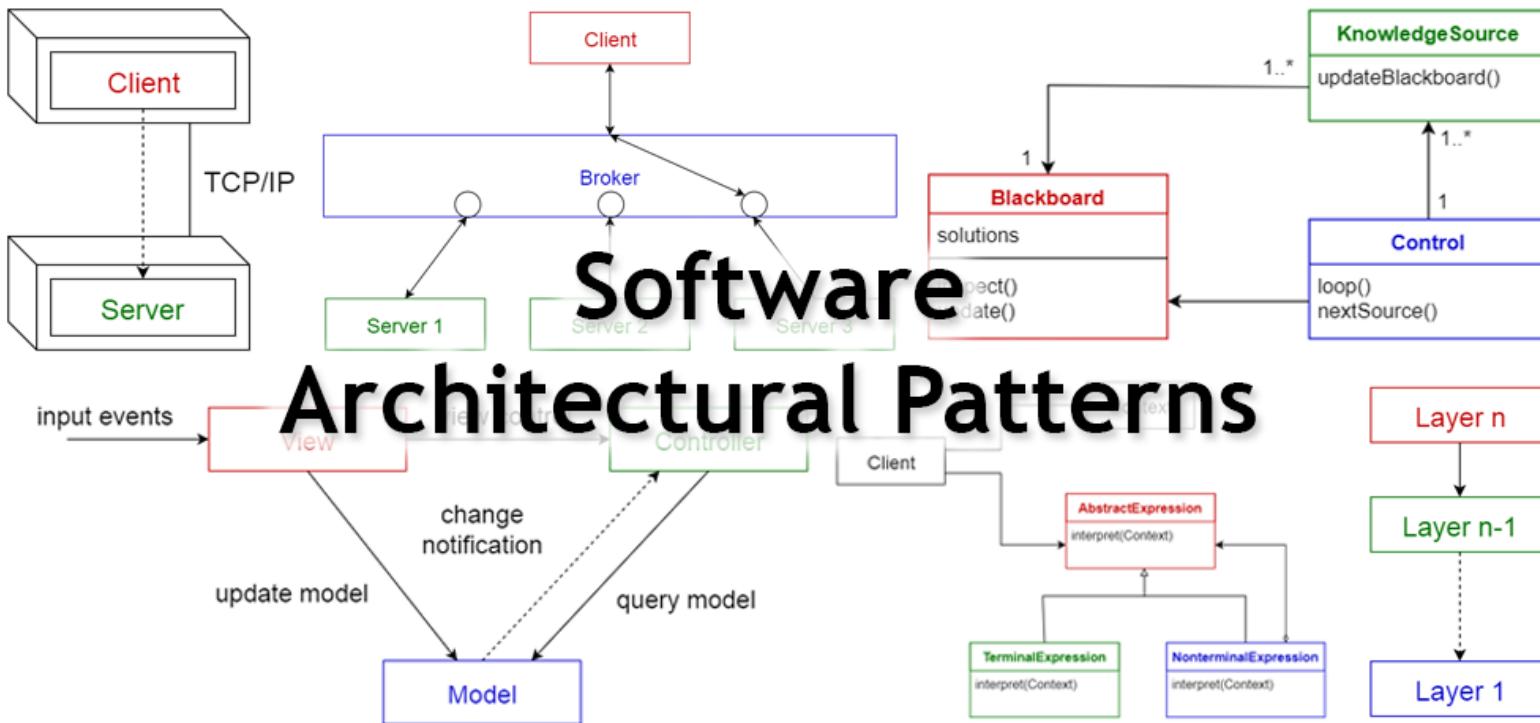
! Necessário chegar a consenso

- ❖ Using collective knowledge and experience to arrive at a unified vision for the architecture



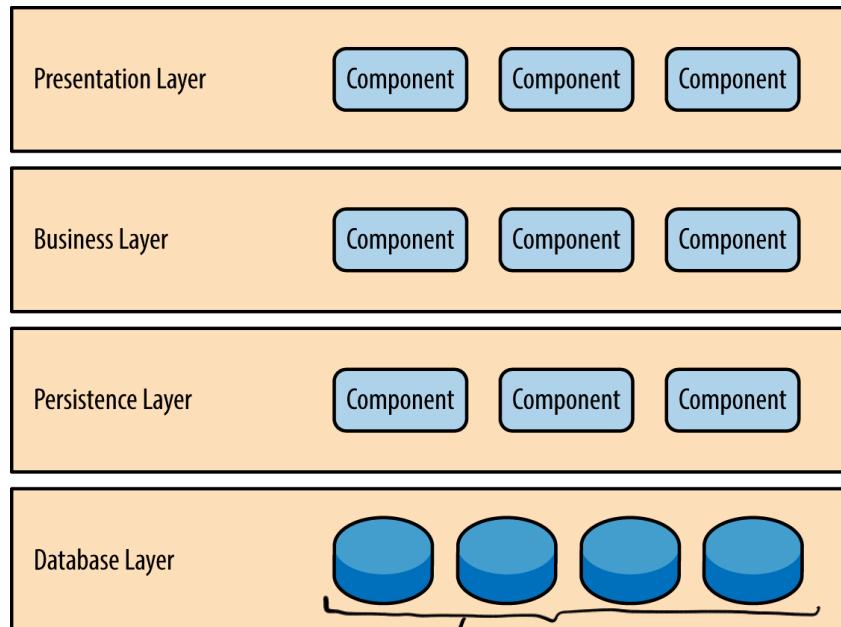
# Architecture Patterns

! Qual é o objetivo?  
Qual a solução?  
Qual o padrão de arquitetura + adequado?



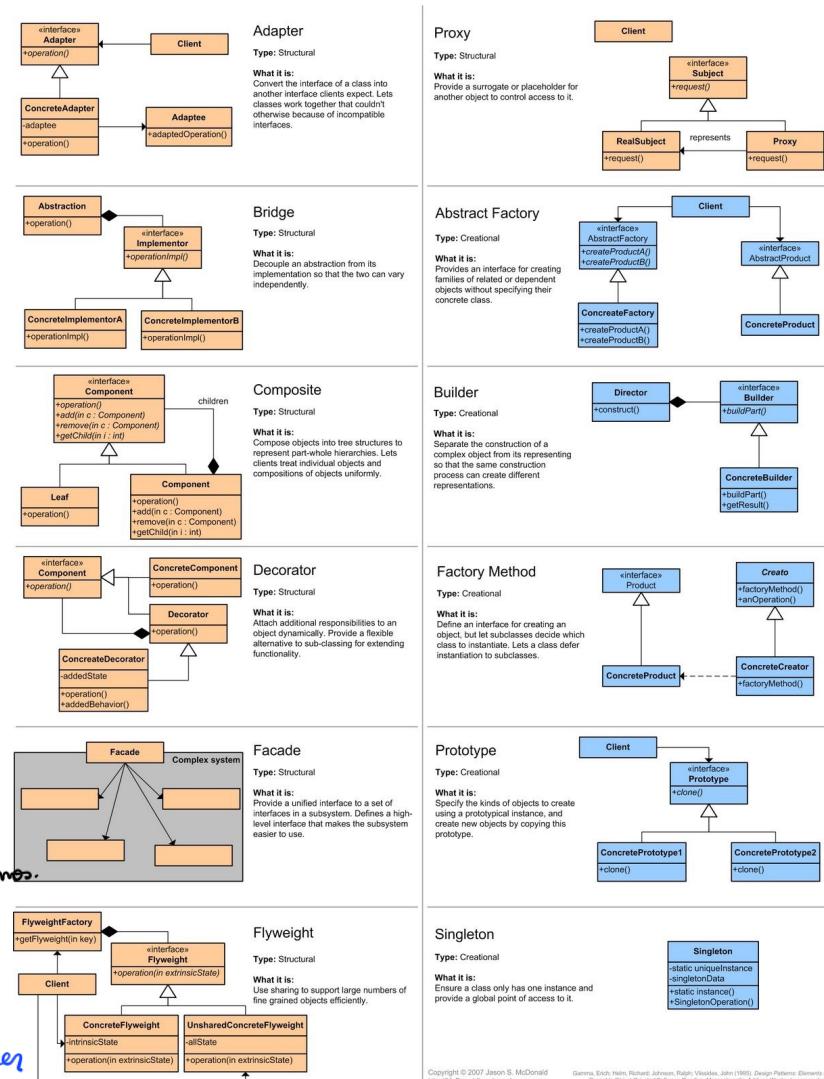
# Architectures, Components, Classes

Em PDS não falamos disso.



No máximo coloca quantas bases de dados temos.  
Mais que isso não interessa.

Presentation Layer / Business Layer / Persistence Layer / DB layer



Copyright © 2007 Jason S. McDonald  
<http://McDonaldLand.wordpress.com>

Gamma, Erich; Helm, Richard; Johnson, Ralph; Vlissides, John (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Reading, Massachusetts: Addison Wesley Longman Inc.

# (Simpler) Architecture Patterns

## ❖ Client-server pattern

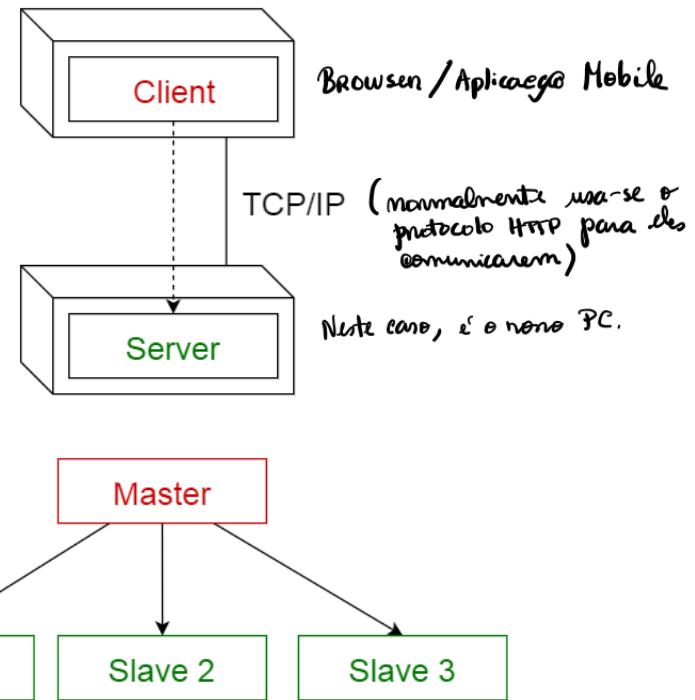
- online apps, email, ..
- ⊕ importante para o m<sup>o</sup> projeto

! Nenhum grupo est<sup>a</sup> a usar

## ❖ Master-slave pattern

- multi-threaded apps, database replication

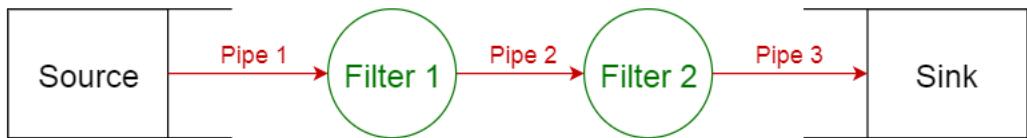
→ útil qd queremos distribuir processamento



## ❖ Pipe-filter pattern

- Compilers, streamed processes, buffering or synch

→ Muito baseado na linha de comandos do linux (conexões de processo)



# Architecture Patterns

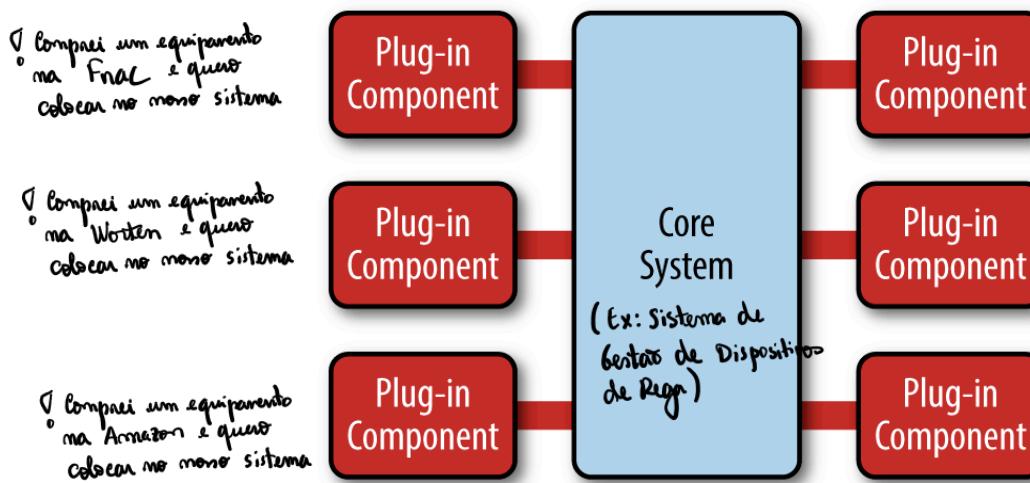
---

- ❖ Microkernel Architecture
- ❖ Layered Architecture
- ❖ Event-Driven Architecture
- ❖ Service-oriented Architecture
- ❖ Microservices Architecture
- ❖ Space-Based Architecture

# Microkernel Architecture

- ❖ **Core** application (stable) + **plugins** (dynamic)

- New application features are added as plug-ins
- Extensibility, feature separation and isolation.



Ex: Kernel do nosso Linux

→ Qd queremos + alguma coisa vamos instalando os pacotes.

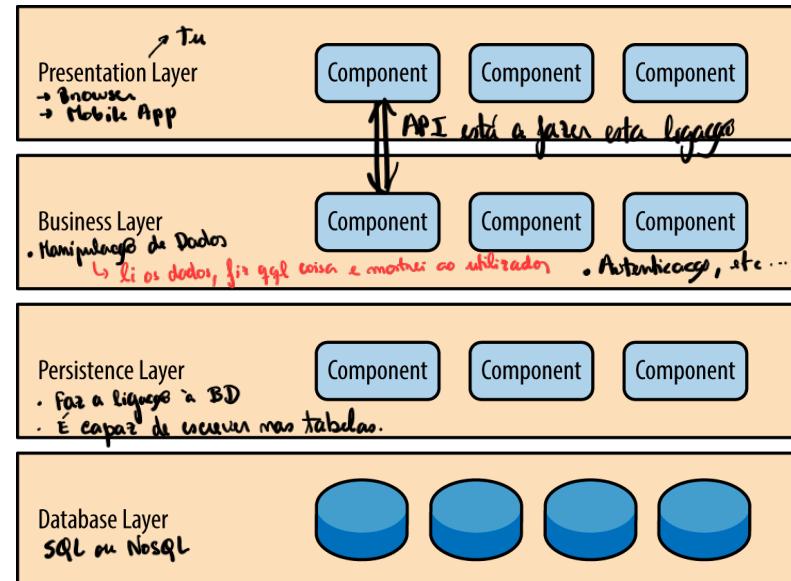
Ex: VsCode

- ❖ It can be embedded or used as part of another architecture pattern.
- ❖ A good first choice for product-based applications.

# Layered Architecture (n-tier)

⇒ Todos temos que ter ( projeto)

- ❖ The most common architecture pattern.
- ❖ A de facto standard for most Java EE applications.
- ❖ Each layer has a specific role and responsibility.
  - Separation of concerns
- ❖ Closely matches the organizational structures found in most companies.
- ❖ Solid general-purpose pattern
  - making it a good starting point for most applications



# Event-driven Architecture

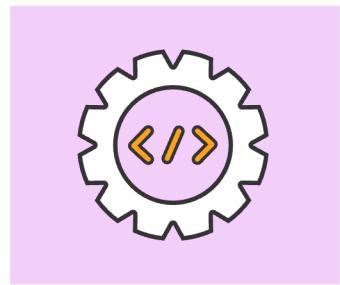
⇒ Todos temos eventos no projeto.

## ❖ Distributed and asynchronous

- used to produce highly scalable applications.
- AKA, message-driven or stream processing

Ex: Consumir os dados dos pilotos e carros na F1.

### EVENT PRODUCER



⇒ Por cada segundo, existem 3 eventos por cada carro.  
(temos 20 carros)

↓  
60 eventos por segundo

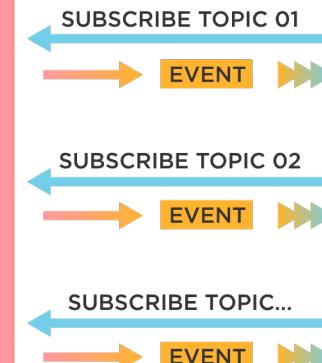


têm que ser  
guardados  
em algum  
lado.

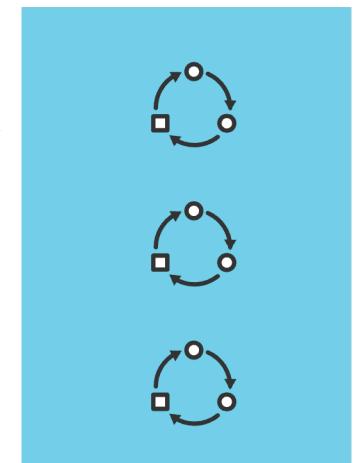
• É financeiramente impossível  
termos um sistema que congega guardar  
todos os eventos numa base de dados.



Alternativa: Atirarmos para  
um Broker.



### EVENT CONSUMER(S)



? Vamos ter uma aula  
a falar só sobre tecnologias

Vafka, RabbitMQ, ...

↓  
Quem quiser lá.  
Está no email durante algum tempo.  
Quem quiser econome.

# Event-driven Architecture

Não garante  
Atomicidade

→ Antes de fazer qq coisa, ela tem que terminar.

→ Os eventos são lançados para o Broker e nenhuma garante que não sejam consumidos.

- ❖ The event-driven architecture pattern is a relatively **complex** pattern to implement
  - Primarily due to its **asynchronous** distributed nature
- ❖ **Lack of atomic transactions** for a single business process
  - Event processor components are highly decoupled and distributed
  - It is very difficult to maintain a transactional unit of work across them
- ❖ A key aspect is the **creation, maintenance, and governance** of the event-processor component contracts.

# Architecture Patterns (recap)

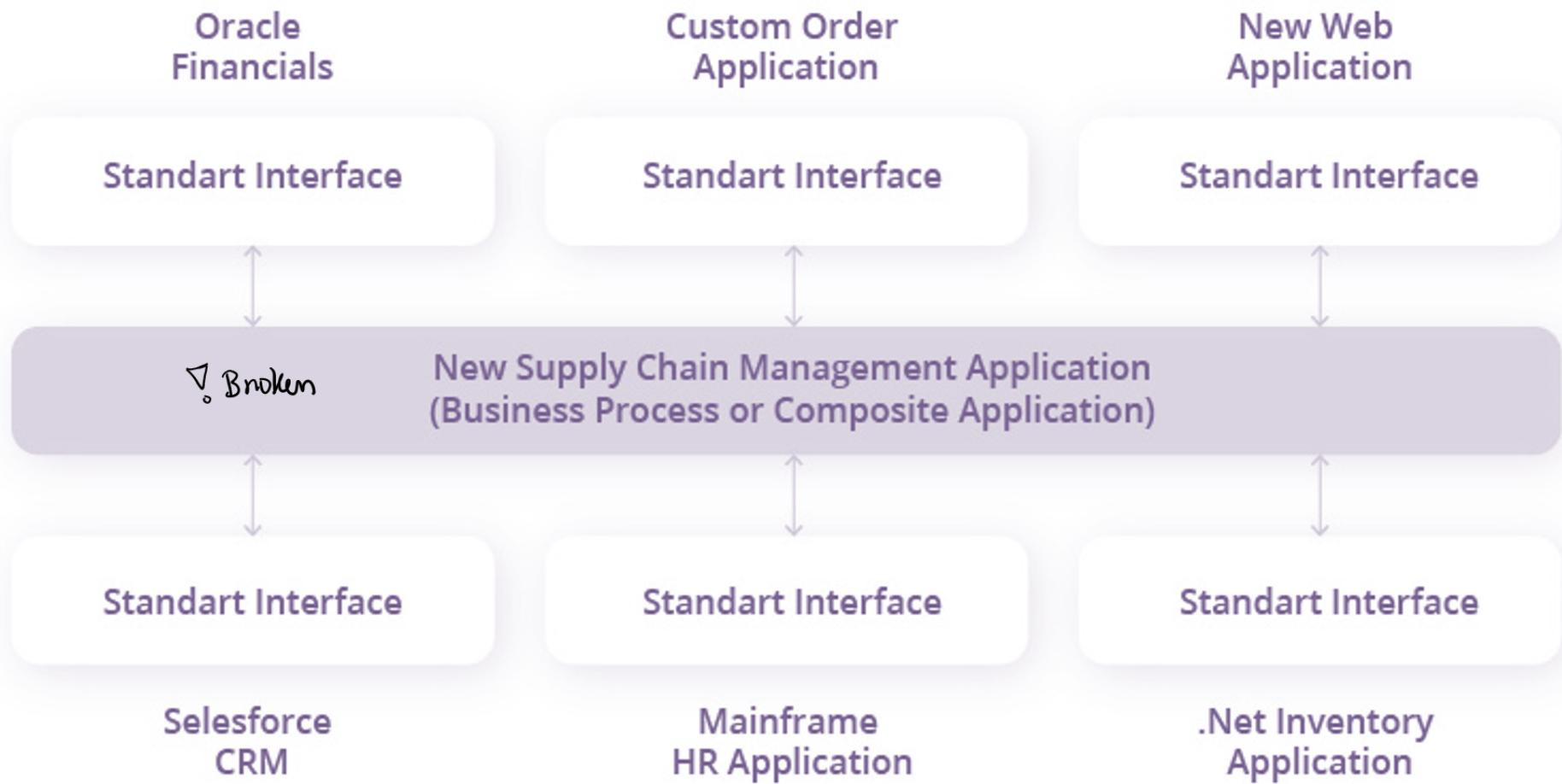
---

- ❖ Microkernel Architecture
- ❖ Layered Architecture
- ❖ Event-Driven Architecture
- ❖ **Service-oriented Architecture**
- ❖ Microservices Architecture
- ❖ Space-Based Architecture

# Service-Oriented Architecture (SOA)

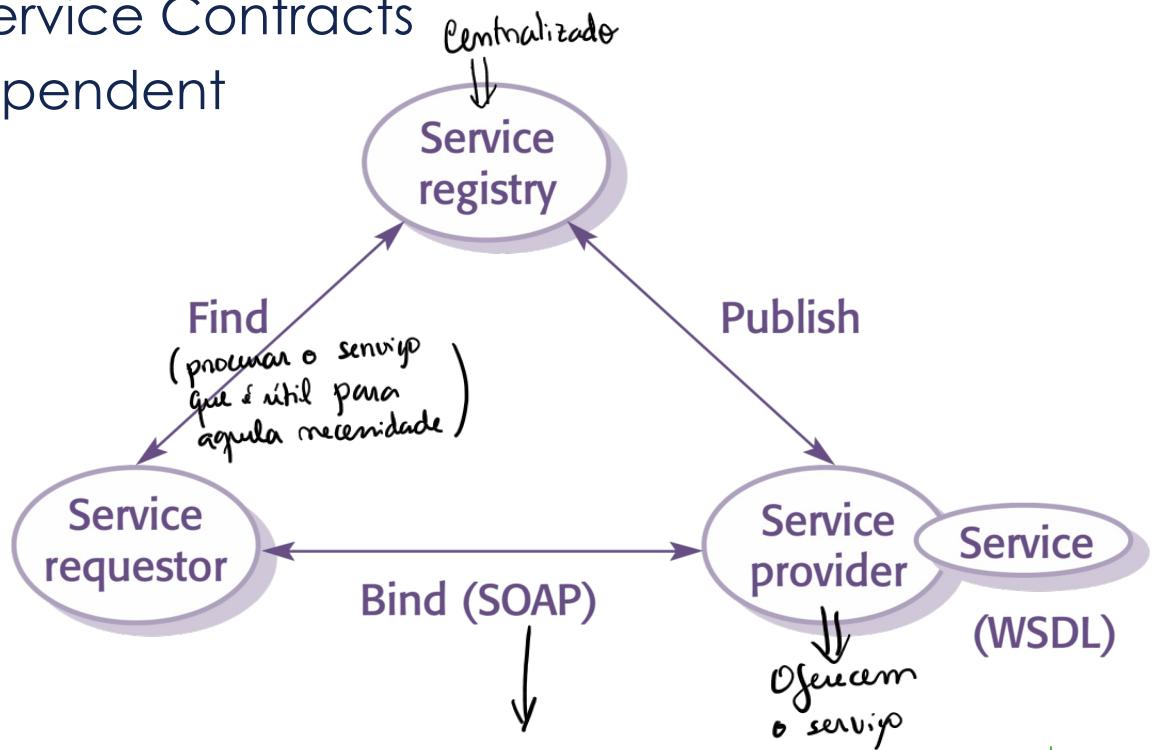
⇒ Provadurte tecnos servicos na mosa arquitectura

Platformas Brancas → sao servicos → consumem ou produzem eventos.



# Service-Oriented Architecture (SOA)

- ❖ A means of developing distributed systems where the components are stand-alone services
- ❖ Key characteristics
  - Standardized Service Contracts
  - Language-independent
  - Loose Coupling
  - Abstraction
  - Reusability
  - Autonomy
  - Statelessness
  - Discoverability
  - Composability



# SOAP (Simple Object Access Protocol)

⇒ Permite que os serviços comunicarem entre eles  
(+ recente evolução Restful web services ⇒ próximo slide)

- ❖ A protocol based on XML language.
  - Platform and language independent communication.
- ❖ The structure of SOAP messages:
  - An Envelope identifying the XML document as a SOAP msg
  - A Header element that contains header attributes
  - A Body containing call and response information
  - A Fault element containing errors and status information

```
<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
    soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">
    <soap:Header>
        ...
    </soap:Header>
    <soap:Body>
        ...
            <soap:Fault>
                ...
            </soap:Fault>
        </soap:Body>
    </soap:Envelope>
```

# RESTful web services

---

- ❖ SOAP/WSDL based web services
  - Not as simple as the acronym would suggest.
  - Criticized as ‘heavyweight’ standards that are over-general and inefficient.
- ❖ An alternative...
- ❖ **REST (REpresentational State Transfer)**



# RESTful web services



- ❖ An architectural style based on **transferring representations of resources** from a server to a client.
- ❖ **It is simpler than SOAP/WSDL** for implementing web services
- ❖ RESTful services involve a lower overhead than so-called 'big web services'
  - are used by many organizations implementing service-based systems

<https://nordicapis.com/rest-better-than-soap-yes-use-cases/>

# REST (REpresentational State Transfer)

→ Permite usar uma linguagem uniforme desde a base de dados até aos clientes.

- ❖ When REST is used, requests can be sent to various endpoints through **GET**, **PUT**, **POST** and **DELETE** HTTP requests.
  - ? Problemas de segurança:  
Atacante podem tirar proveito disto para XSS
  - editar
  - . criar uma nova entidade
  - . Mais prática: usar POST para tudo incluindo editar
- ❖ Provides a stateless, simpler and lightweight way of communicating with a system.
- ❖ It uses JSON format to send and receive data.
  - This format is a simple text containing a series of attribute-value pairs

```
{ "trade": { "cust_id": "12345", "cusip": "037833100", "shares": "1000", } }
```

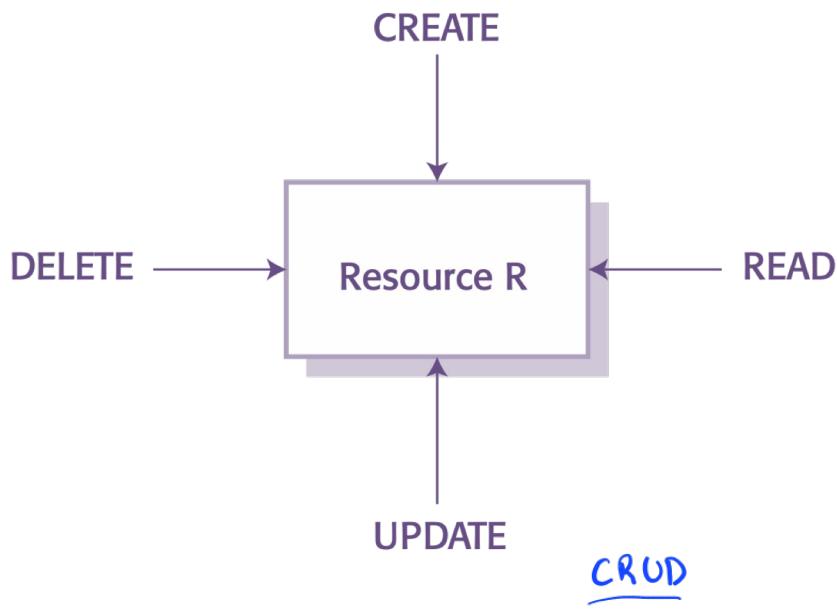
# Resources

---

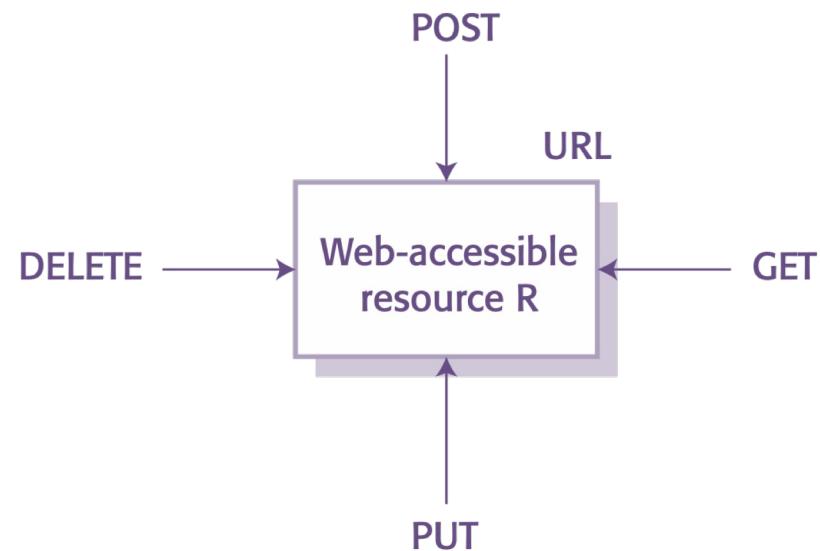
- ❖ The fundamental element in a RESTful architecture is a resource
- ❖ Essentially, a resource is simply a data element such as a catalogue, a medical record, or a document
  - <http://api.ipma.pt/file-example/1110600.json>
  - <http://api.ipma.pt/open-data/distrits-islands.json>
- ❖ In general, resources may have multiple representations, i.e., they can exist in different formats.
  - JSON, TXT, DOC, PDF, HTML, ...

# Resource operations

- ❖ **Create** – bring the resource into existence
- ❖ **Read** – return a representation of the resource
- ❖ **Update** – change the value of the resource
- ❖ **Delete** – make the resource inaccessible



(a) General resource actions

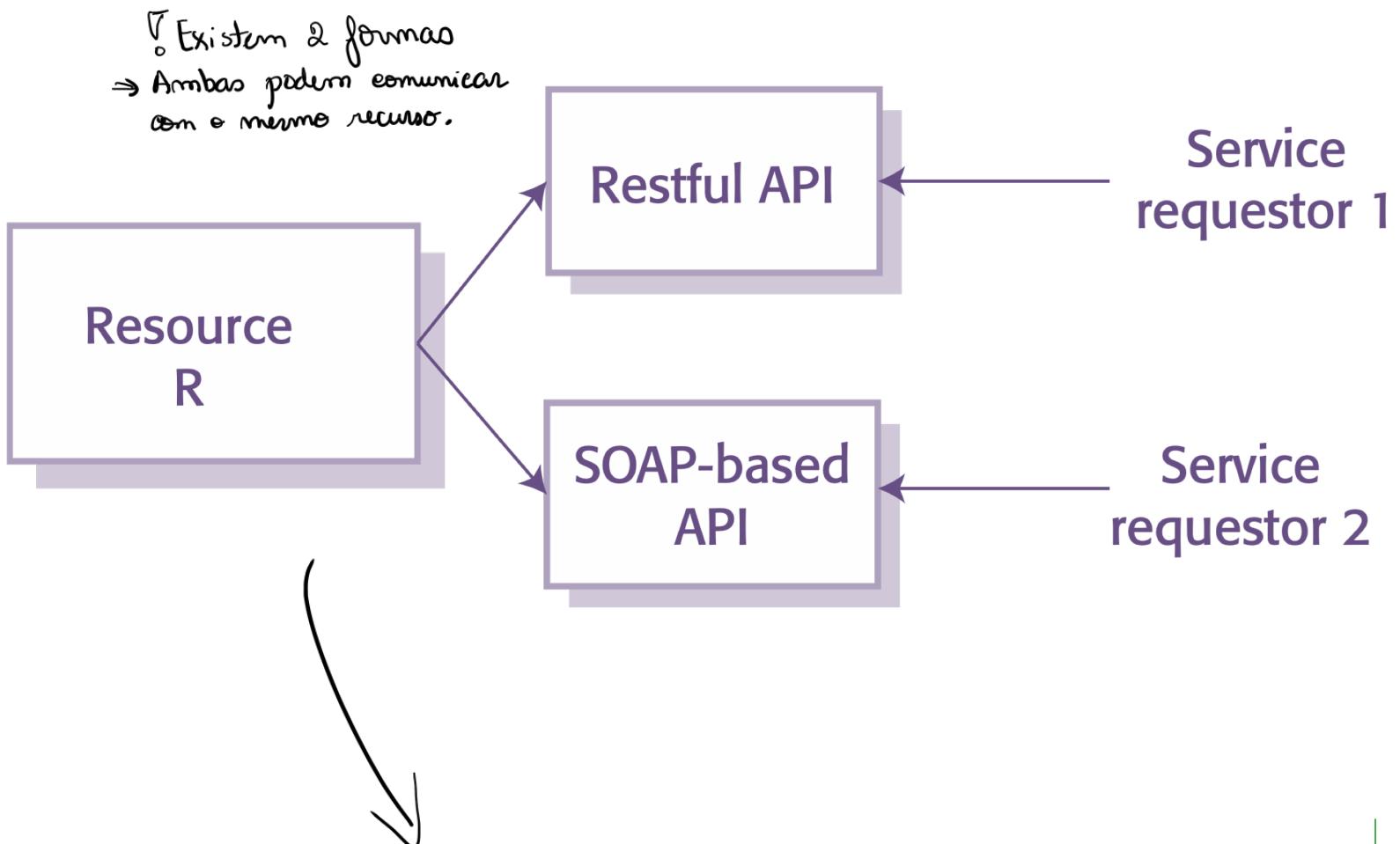


(b) Web resources

# Disadvantages of RESTful approach

- ❖ It can be difficult to design a set of RESTful services
- ❗ to represent complex resources/interfaces ⇒ Se representarmos a UA como um único objeto ficariamos com um JSON gigante.
- ❖ There are no standards for RESTful interface description ⇒ Cada grupo terá uma forma de representar a sua API  
⇒ Existem boas práticas mas nenhuma obrigação rest/api/component  
app/component  
...
- so service users must rely on informal documentation to understand the interface
- ❖ When using RESTful services, we have to implement our own infrastructure for monitoring and managing the quality of service and the service reliability
  - e.g. REST does not impose any sort of security like SOAP
  - SOAP requires less plumbing code than REST services design

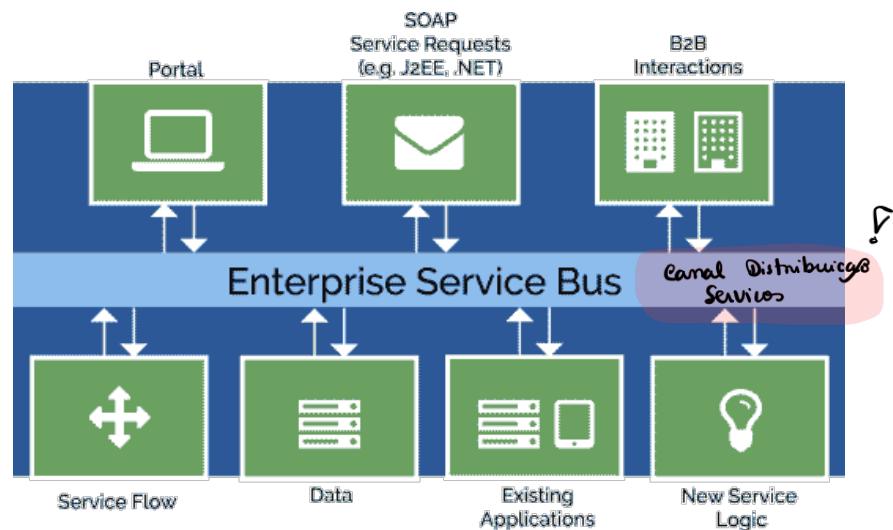
# RESTful and SOAP-based APIs



# SOA topology

⇒ Arquitetura baseada em serviços

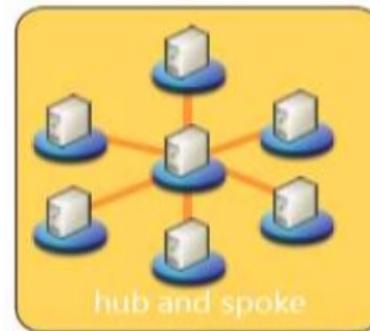
- ❖ SOA was primarily used to develop Monolithic application, though it has evolved from there
- ❖ Many organizations use now SOA to resolve integration complexities.
  - Organizations in this case heavily depend on **Enterprise Service Bus (ESB)** technologies (messaging middleware).
  - All services are accessed through the ESB.



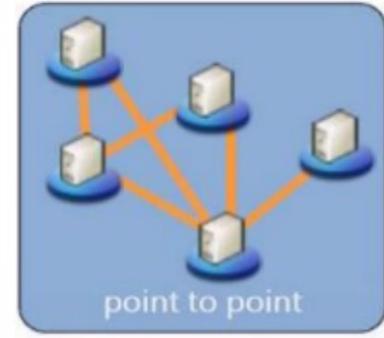
# Key Characteristics of ESB

*Cada serviço é uma caixa negra  
→ Permite modular o trabalho.*

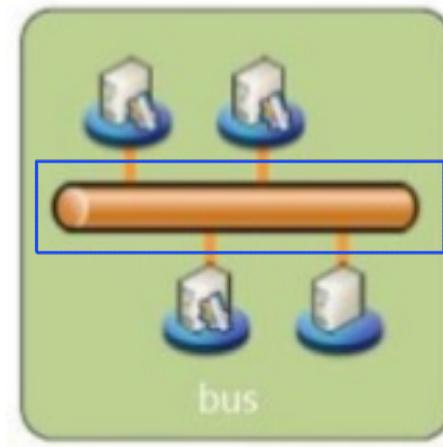
- ❖ Streamlines development
- ❖ Supports multiple binding strategies
- ❖ Performs data transformation
- ❖ Intelligent routing
- ❖ Real time monitoring
- ❖ Exception handling
- ❖ Service security



"too centralized"



"too decentralized"



"just right"

# Architecture Patterns (recap)

---

- ❖ Microkernel Architecture
- ❖ Layered Architecture
- ❖ Event-Driven Architecture
- ❖ Service-oriented Architecture
- ❖ Microservices Architecture
- ❖ Space-Based Architecture

# Microservices Architecture

- ❖ It evolved from **two main sources**:
    - applications developed using the **layered pattern**
    - **distributed applications** developed using **service-oriented architecture**
  - ❖ The first characteristic is the notion of **separately deployed units**.
    - Each service component is deployed as a separate unit, allowing for easier deployment and decoupling
  - ❖ **Distributed architecture**
    - all the components are fully decoupled
    - communication through JMS, AMQP, REST, SOAP, RMI, etc.
- Para o cliente não existe percepção que temos serviços diferentes.*
- 
- ```
graph TD; CR1[Client Requests] <-->UIL[User Interface Layer]; CR2[Client Requests] <-->UIL; CR3[Client Requests] <-->UIL; UIL <--> SC1[Service Component]; UIL <--> SC2[Service Component]; UIL <--> SC3[Service Component]; SC1 <--> M1[module]; SC1 <--> M2[module]; SC2 <--> M3[module]; SC2 <--> M4[module]; SC3 <--> M5[module]; SC3 <--> M6[module];
```
- Exemplos:* ① Neste projeto vamos usar para desacoplar a informação.  
② Microserviço de autenticação com Google.

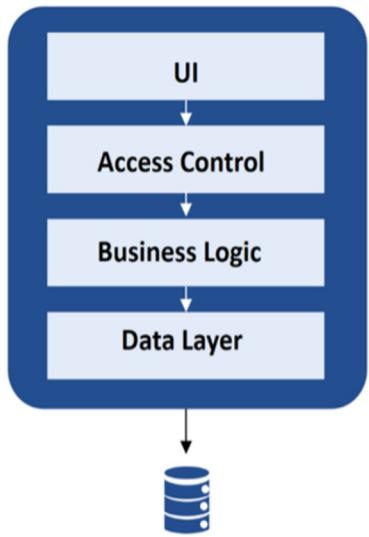
# Microservices Architecture

---

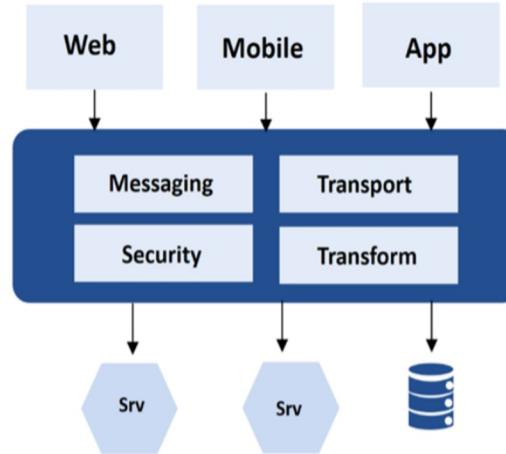
- ❖ Applications are generally more robust, provide better scalability, and can more easily support continuous delivery
  - Capability to do real-time production deployments
- ❖ Only the service components that change need to be deployed
- ❖ But ... distributed architecture
  - it shares some of the same complex issues found in the event-driven architecture pattern, including contract creation, maintenance, and government, remote system availability, and remote access authentication and authorization.

Desvantagens {  
① repositórios para gerir  
② complexo para comunicar

# SOA vs. Microservices

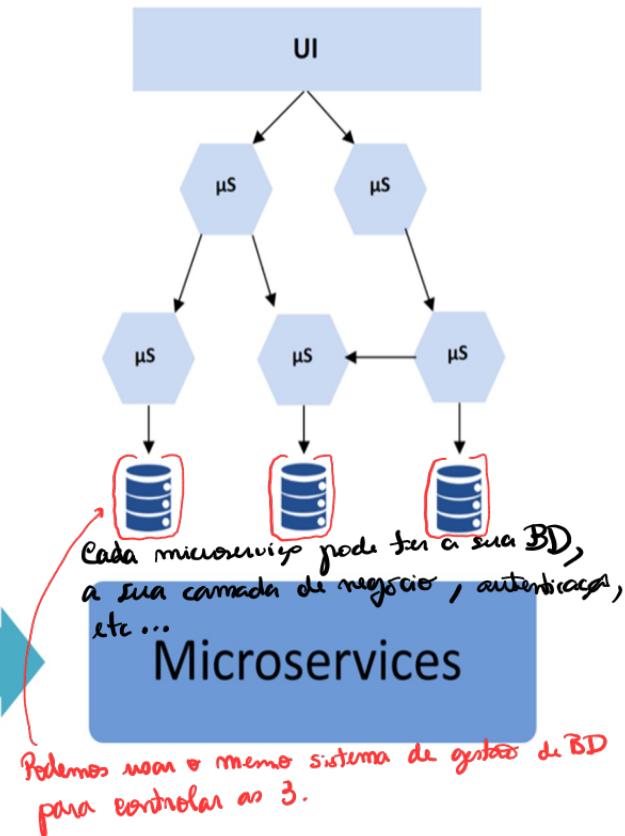


Monolithic



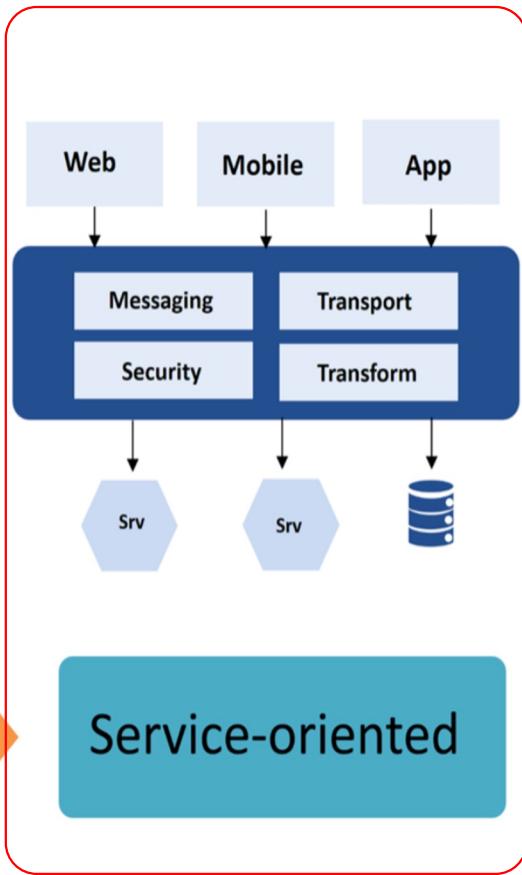
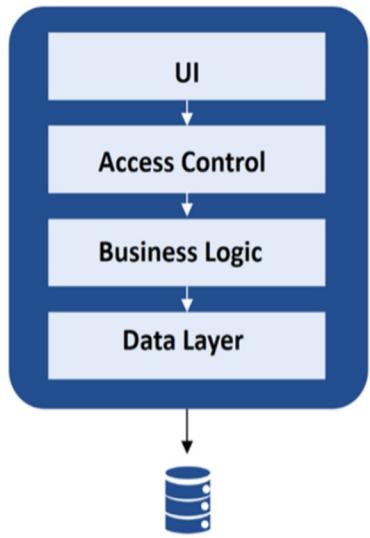
Serviços estarão todos na mesma estrutura.

Service-oriented



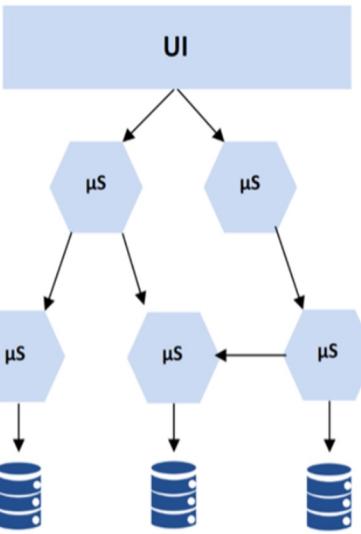
Microservices

# SOA vs. Microservices



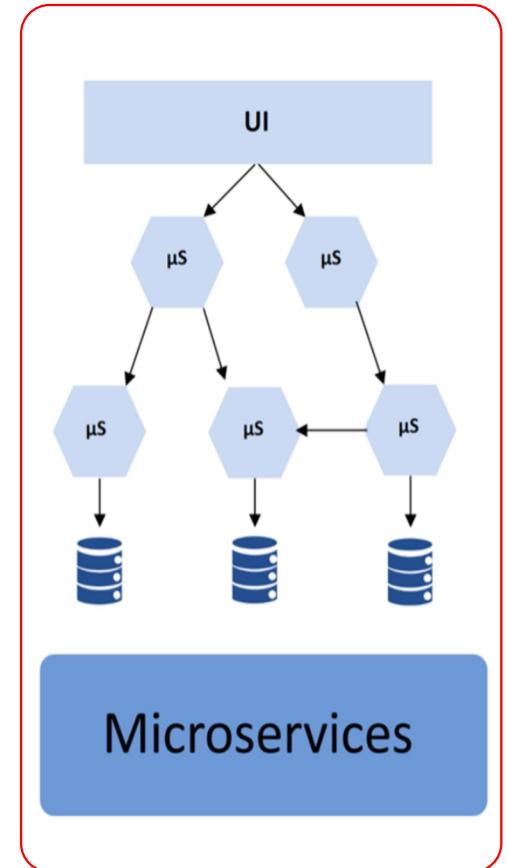
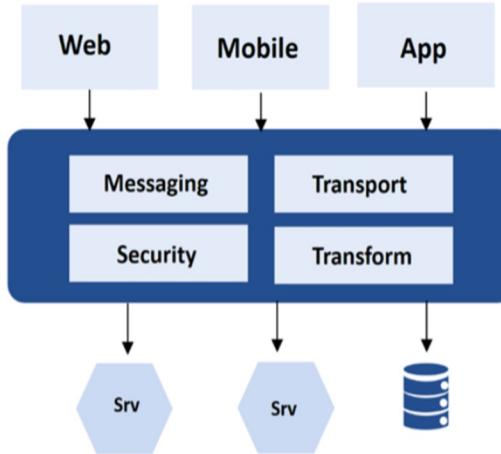
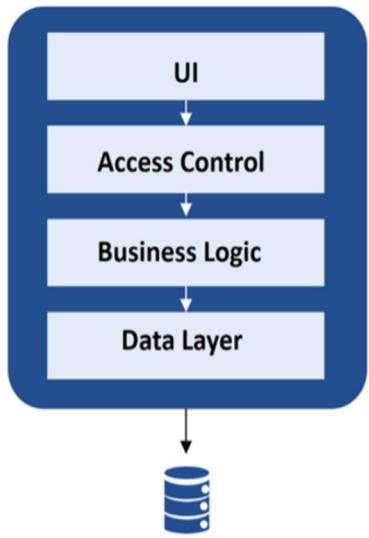
Monolithic

Service-oriented



Microservices

# SOA vs. Microservices



Monolithic



Service-oriented



Microservices

# Space-Based Architecture

- ↑ complexa
- Criada para escalar
- Funciona concurrentemente

• Em princípio não serve para projeto.

## ❖ Web-based applications

- Bottlenecks start appearing as the user load increases
  - first at the web-server layer, then at the application-server layer, and finally at the database-server layer

## ❖ Specifically designed to address **scalability and concurrency** issues

- It is often a better approach than trying to scale out a database into a non-scalable architecture

## ❖ A good choice for applications with variable load

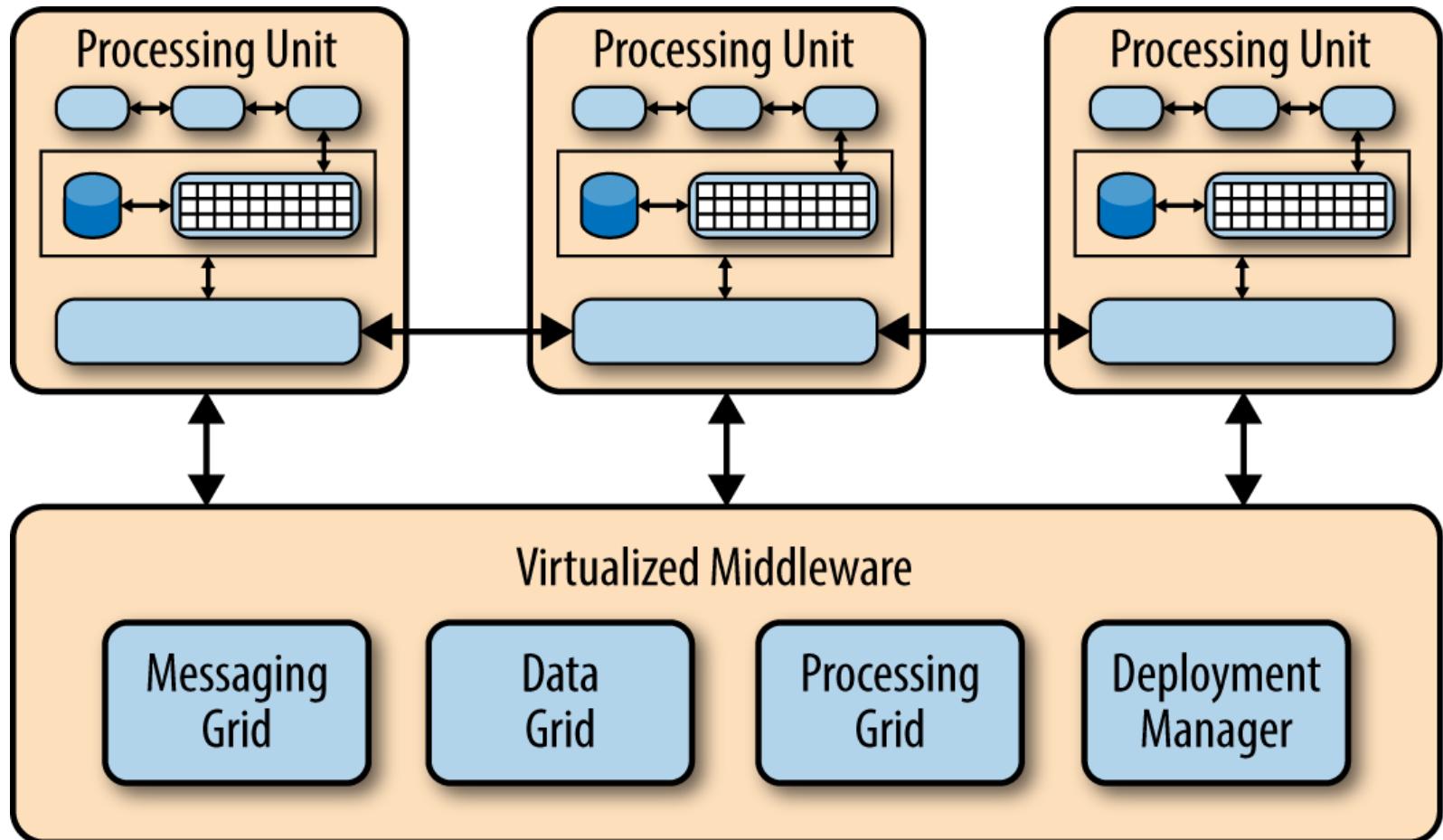
- (e.g., social media sites, bidding and auction sites).

? No Natal a mesma aplicação vai ter muita carga.

criamos clones da mesma app.

A medida que a carga diminui, clones vão desaparecendo.

# Space-Based Architecture



# Space-Based Architecture

Ex: Leilão Online

Coloco 10 carros por cada unidade de processamento

qd os leilões terminam elas vão desaparecendo.

## ❖ Example

- Bidding auction site

1. The application would receive a bid for a particular item
2. Record that bid with a timestamp
3. Update the latest bid information for the item
4. Send the information back to the browser.



# Summary

---

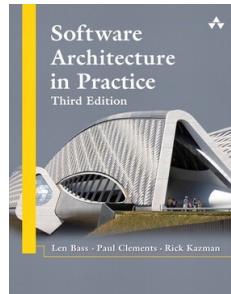
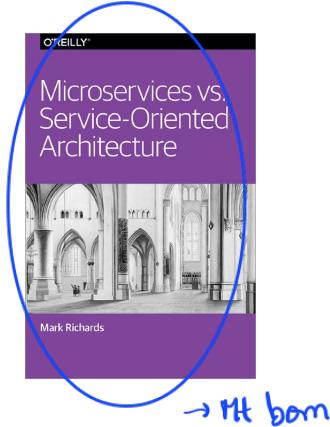
- ❖ A **software architecture** is a description of how a software system is organized
  - Properties of a system such as:
    - Performance
    - Security
    - Availability
- ❖ Architectural design include **decisions** on:
  - the type of application
  - the distribution of the system

# Summary

---

- ❖ Architectures may be **documented** from several different perspectives
  - Possible views include:
    - a conceptual view,
    - a logical view,
    - a process view,
    - a development view,
    - a physical view. *Qts máquinas precêm utilizar?*
- ❖ Architectural **patterns** are a means of reusing knowledge about generic system architectures
  - They describe the architecture,
    - explain when it may be used
    - point out its advantages and disadvantages

# Resources & Credits



- ❖ Ian Sommerville, *Software Engineering*, 10th Edition, Pearson, 2016 (chapters 6, 17, 18)
- ❖ Mark Richards, *Microservices vs. Service-Oriented Architecture*, O'Reilly, 2015
- ❖ Mark Richards,  
*Software Architecture Fundamentals Workshop*  
<https://conferences.oreilly.com/software-architecture>
- ❖ Rick Kazman, Paul Clements, Len Bass,  
*Software Architecture in Practice*, Addison-Wesley, 2012

# Extra resources

---

- ❖ Rick Kazman, Paul Clements, Len Bass,  
Software Architecture in Practice,  
Third Edition, 2012
  
- ❖ Comparing architectures in java ecosystem
  - <https://www.dineshonjava.com/software-architecture-patterns-and-designs/>