

MICROSERVIZI

CORSO INTRODUTTIVO

MICROSERVIZI REST - JAVA, DOCKER, SPRING BOOT

INTRODUZIONE

Giorno 2:

Architettura SOA. Architettura monolitica vs Microservizi. Principi e caratteristiche dei Microservizi. Caso d'uso di applicazioni a Microservizi. Caso di studio: integrazione di un microservizio in Docker.

Giorno 1:

Panoramica Docker e Architettura. Immagini e container. Installazione. Comandi e Navigazione. Costruzione e gestione di un'immagine. Docker aspetti avanzati. Docker Images Repository. Docker Filsystem and Volumes, Docker Networking.

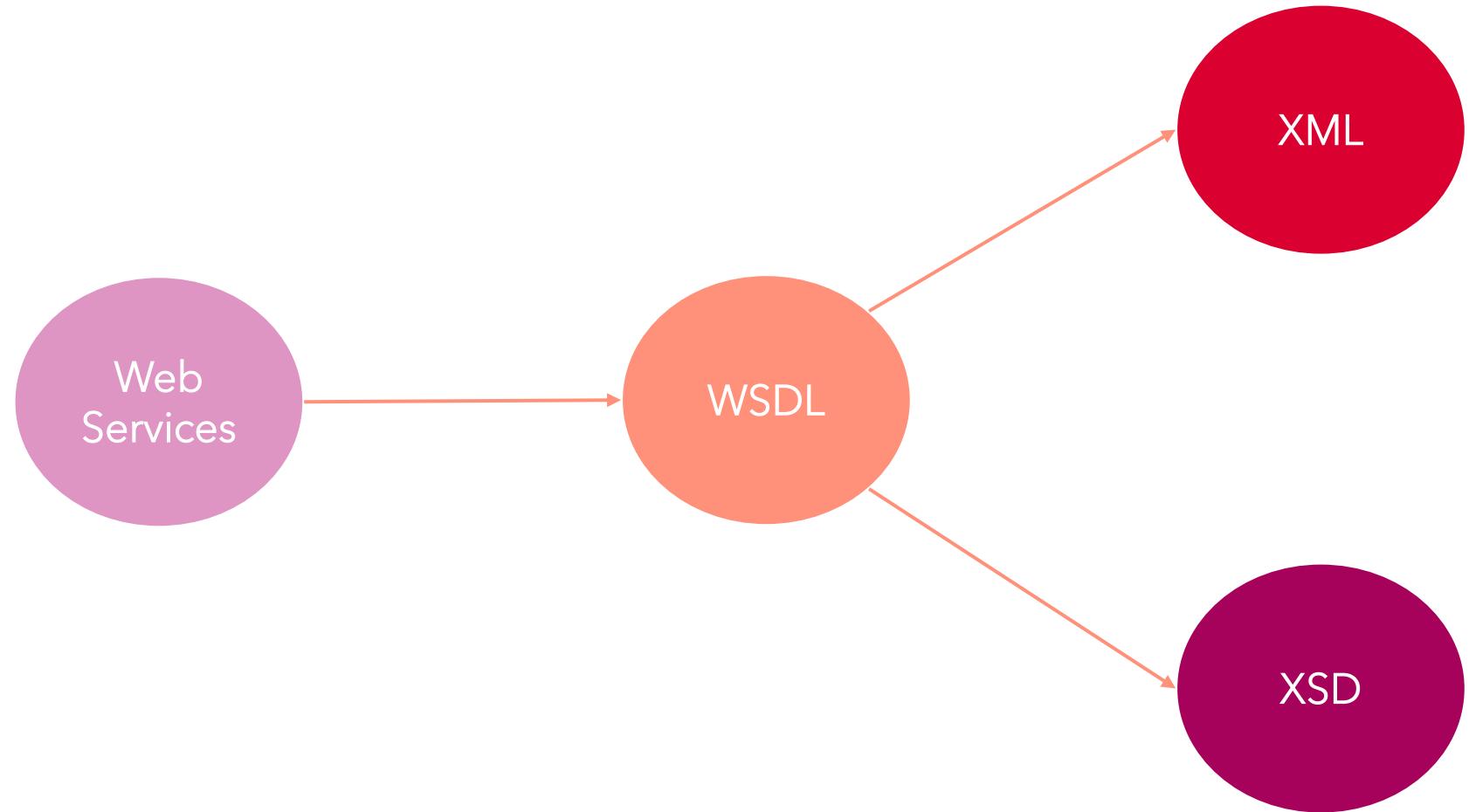
Giorno 3:

Spring Boot: installazione e configurazione dell'ambiente di sviluppo. Introduzione a Spring Boot: Configurazione e servizi REST/SOAP. Esempio di applicazione RESTful con Spring Boot.

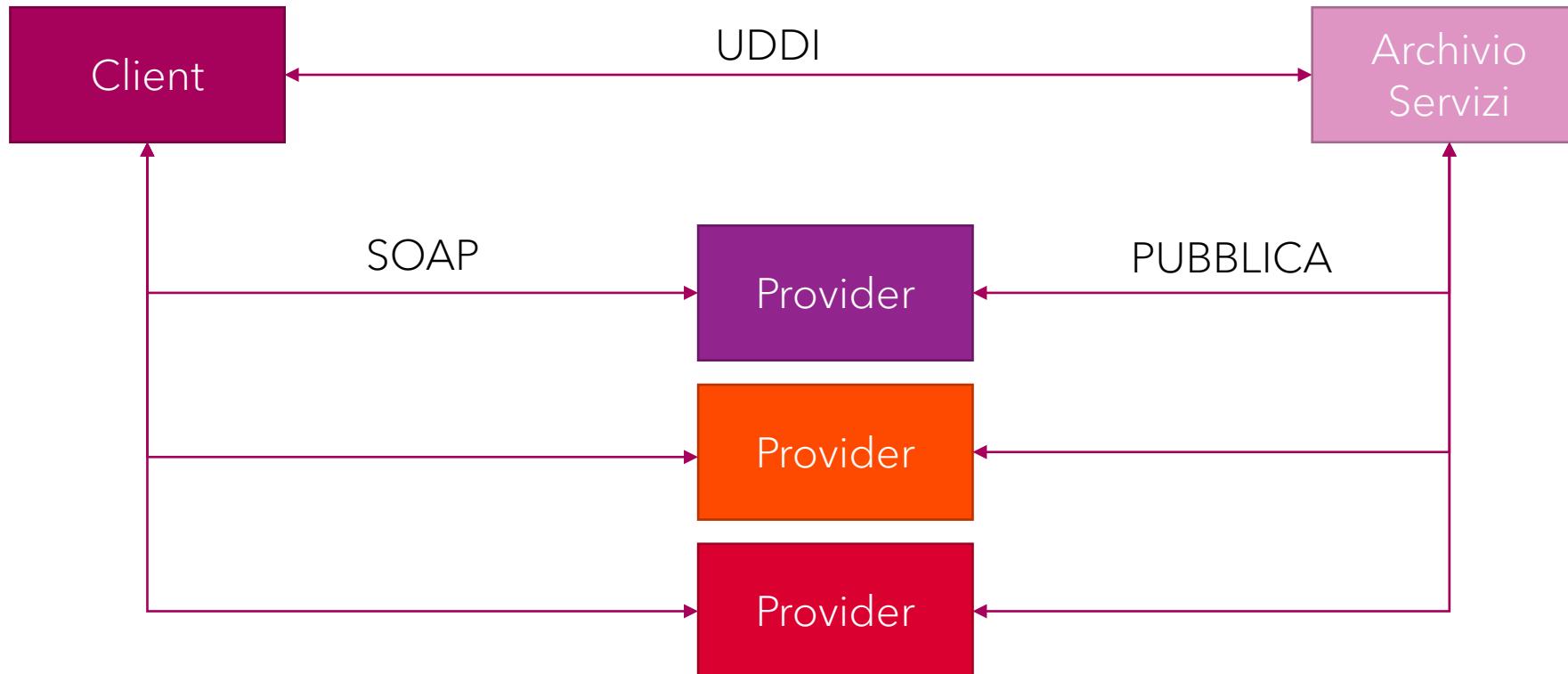
Sicurezza nei Microservizi. Implementazione di sistemi di sicurezza. Abilitare l'accesso cross-origin. Esempio di utilizzo di sistemi di cache. Monitoraggio dei microservizi. Esempi di test automatizzati.

.

ARCHITETTURE SOA: I BLOCCHI FONDAMENTALI



SOA: WEB SERVICE



- Provider: il proprietario del servizio
- Registro: dove il servizio viene pubblicato
- Client: l'utilizzatore del servizio

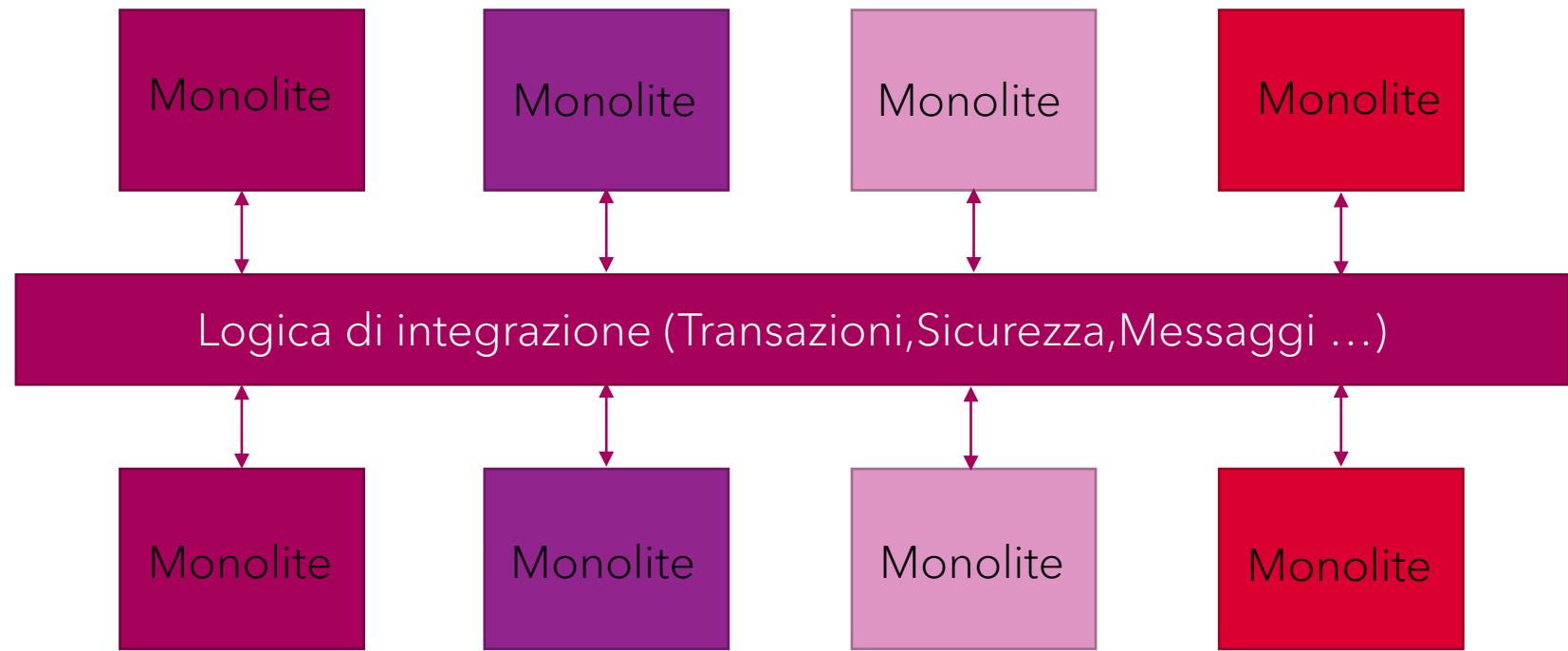
ESB: ENTERPRISE SERVICE BUS

L'ESB è un'architettura, una serie di regole e principi per integrare tra loro numerose applicazioni su un'infrastruttura bus.

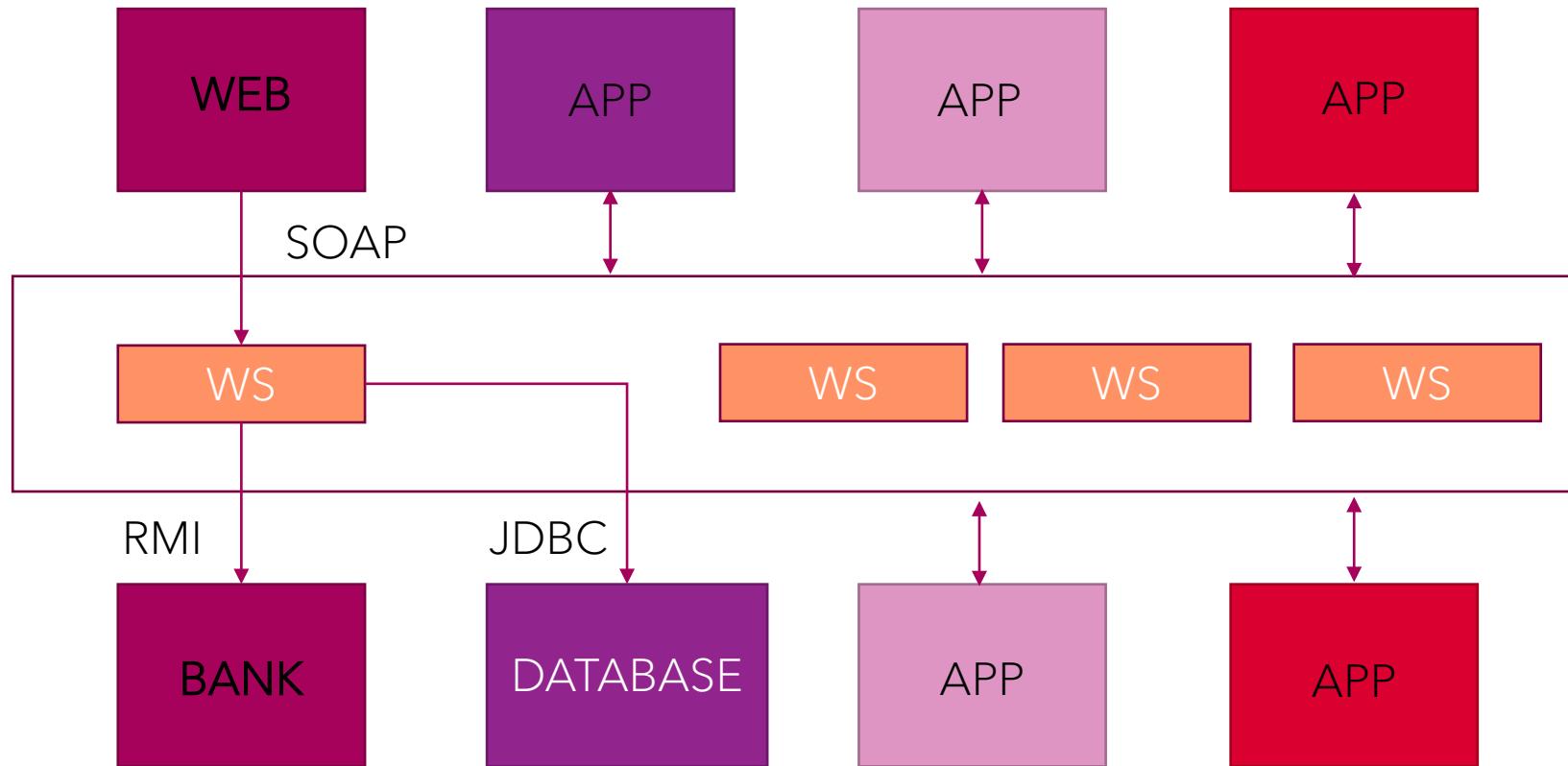
Questo middleware (un software che unifica programmi complessi ed esistenti) ha la responsabilità primaria di **connettere fra loro applicazioni e servizi eterogenei di una Service-Oriented Architecture**.

L'ESB è importante per trasferire, trasformare, inviare messaggi al provider di servizio più opportuno. In questo modo, si semplificano i compiti sia dell'utente che del provider.

Infatti, esso **unifica i diversi metodi usati dai diversi componenti per ricevere e mandare informazioni alle altre applicazioni**.



ESB: ENTERPRISE SERVICE BUS



XML

- Metalinguaggio per la rappresentazione di contenuti testuali

Esempio:

```
<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>  
<documento>  
    <titolo>...</titolo>  
    <data>...</data>  
    <testo>...</testo>  
</documento>
```

XML WELL FORMED

- Gerarchia di tag
- Esistenza del tag root
- Tutti i tag devono essere chiusi
- Ogni tag può avere attributi
- Codifica di caratteri particolari

Entity	Carattere
&amp	&
&alt	<
&gt	>
&quot	"
&apos	'

XSD (XML SCHEMA)

- Definire un documento XML
- Validare un documento XML

```
<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema>
.....
</xsd:schema>
```

XML SCHEMA

```
<?xml version="1.0" encoding="UTF-8"?>
<schema
    xmlns="http://www.w3.org/2001/XMLSchema"
    xmlns:dep="http://www.titan.com/Dependency"
    targetNamespace="http://www.titan.com/Dependency">

    <element name="address" type="dep:Address" >
        <complexType name="AddressType">
            <sequence>
                <element name="street" type="string" />
                <element name="city" type="string" />
                <element name="state" type="string" />
                <element name="zip" type="string" />
            </sequence>
        </complexType>
    </element>
</schema>
```



```
<?xml version="1.0" encoding="UTF-8"?>
<address>
    <street>...</street>
    <city>...</city>
    <state>...</state>
    <zip>...</zip>
</address>
```

NAMESPACE

- Lo schema definito è un linguaggio di markup che descrive un altro linguaggio di markup: il documento effettivo che sarà validato.
- Ci sono dei tipi che possono essere utilizzati in più documenti XSD
- Occorre un meccanismo che eviti ambiguità
- I namespaces sono esattamente la soluzione al problema dell'ambiguità
- Dichiarazione: **xmlns:prefix="..."**
- Definire sempre un Namespace di default e un **targetNamespace**

TIPI DI DEFAULT

XML Schema type	Java type
byte	Byte,byte
boolean	Boolean, boolean
short	Short,short
int	Integer, int
long	Long,long
float	Float,float
double	Double,double
string	String
dateTime	Calendar
integer	BigInteger
decimal	BigDecimal

RESTRIZIONI SUI TIPI

```
<?xml version="1.0"?>  
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">  
  <xsd:element name="valore">  
    <xsd:simpleType>  
      <xsd:restriction base="xsd:integer">  
        <xsd:minInclusive value="0" />  
        <xsd:maxInclusive value="10"/>  
      </xsd:restriction>  
      <xsd:simpleType>  
    </xsd:element>  
</xsd:schema>
```

Restrizione	Significato
eumeration	Lista di valori
fractionDigits	Massimo numero di posizioni decimali dopo la virgola
length	Numero di elementi, caratteri o items di una lista
maxExclusive	Valore massimo non incluso
maxInclusive	Valore massimo incluso
maxLength	Numero massimo di caratteri o items se si utilizza una lista
minExclusive	Valore minimo non incluso
minInclusive	Valore minimo incluso
minLength	Numero minimo di caratteri o items se si utilizza una lista
pattern	Il testo deve rispettare un'espressione regolare
whiteSpace	Gestione di spazi e tabulazioni

ATTRIBUTI SUI TAG

Esempi:

```
<xsd:attribute name="lang" type="xsd:string" default="EN"/>  
<xsd:attribute name="lang" type="xsd:string" use="optional"/>  
<xsd:attribute name="lang" type="xsd:string" use="required"/>
```

PROTOCOLLO SOAP

SOAP è un protocollo leggero che permette di scambiare informazioni in ambiente distribuito.

Aspetti fondamentali:

- SOAP è basato su XML
- SOAP gestisce informazione strutturata
- SOAP gestisce informazione tipata
- SOAP non definisce alcuna semantica per applicazioni o scambio messaggi, ma fornisce un mezzo per definirla
- Generalmente utilizzato con il protocollo HTTP

SOAP: CARATTERISTICHE

Un messaggio SOAP è composto da:

- Un elemento radice, **envelope**, obbligatorio. Il namespace di SOAP viene dichiarato all'interno di questo elemento.
- Un elemento **header** opzionale. Il suo scopo è quello di trasportare informazioni non facenti parte del messaggio, destinate agli "attori", cioè alle varie parti che il messaggio attraverserà per arrivare al suo destinatario finale.
- Un elemento **body** obbligatorio. Questo elemento contiene il messaggio vero e proprio.

SOAP: ESEMPIO

```
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
                     xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
                     xmlns:xsd="http://www.w3.org/1999/XMLSchema">
    <SOAP-ENV:Body>
        <ns1:getTemp xmlns:ns1="urn:xmethods-Temperature"
                      SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
            <zipcode xsi:type="xsd:string">10016</zipcode>
        </ns1:getTemp>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

WSDL: WEB SERVICE DESCRIPTION LANGUAGE

- Descrive un servizio web attraverso un documento XML
- Indipendenza dai protocolli, linguaggi, piattaforme utilizzate

```
<definitions>
    <types></types>
    <message></message>
    <portType></portType>
    <binding></binding>
    <service></service>
</definitions>
```

ESEMPIO DI SERVIZIO WEB

Consentire a diverse applicativi l'accesso al servizio di prenotazione.

Il servizio deve esporre una interfaccia che consenta l'invio dei dati relativi ad un utente e restituisca i dati della prenotazione.

La definizione del contratto WSDL è il primo passo verso la realizzazione.
Attraverso di esso forniamo tutto il necessario per il suo utilizzo.

ESEMPIO DI SERVIZIO WEB: DIPENDENZE

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:dep="http://www.titan.com/Dependency"
  targetNamespace="http://www.titan.com/Dependency">

  <complexType name="AddressType">
    <sequence>
      <element name="street" type="string" />
      <element name="city" type="string" />
      <element name="state" type="string" />
      <element name="zip" type="string" />
    </sequence>
  </complexType>
  <complexType name="CustomerType">
    <sequence>
      <element name="last-name" type="string" />
      <element name="first-name" type="string" />
      <element name="address" type="dep:AddressType" />
    </sequence>
  </complexType>
  ...
</schema>
```

ESEMPIO DI SERVIZIO WEB: DIPENDENZE

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:dep="http://www.titan.com/Dependency"
  targetNamespace="http://www.titan.com/Reservation">

<import namespace="http://www.titan.com/Dependency" schemaLocation="dependency.xsd" />
<element name="Reservation">
  <complexType>
    <sequence>
      <element name="amount" type="float" />
      <element name="room" type="int" />
      <element name="customer" type="dep:CustomerType" />
      <element name="Error" type="dep:Error" />
    </sequence>
  </complexType>
</element>
</schema>
```

ESEMPIO DI SERVIZIO WEB: DIPENDENZE

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:dep="http://www.titan.com/Dependency"
  targetNamespace="http://www.titan.com/Customer">

<import namespace="http://www.titan.com/Dependency" schemaLocation="dependency.xsd" />
<element name="Customer">
  <complexType>
    <sequence>
      <element name="dati" type="dep:CustomerType" />
    </sequence>
  </complexType>
</element>
</schema>
```

DEFINIZIONE WSDL

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<definitions
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:res="http://www.titan.com/Reservation"
    xmlns:cust="http://www.titan.com/Customer"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:titan="http://www.titan.com/titan"
    name="ServizioPrenotazione" targetNamespace="http://www.titan.com/titan">
```

```
<documentation>SOAP Definition</documentation>
```

```
<types>
```

```
  <xsd:schema>
```

```
    <xsd:import namespace="http://www.titan.com/Reservation" schemaLocation="reservation.xsd"/>
```

```
    <xsd:import namespace="http://www.titan.com/Customer" schemaLocation="customer.xsd"/>
```

```
  </xsd:schema>
```

```
</types>
```

```
....
```

DEFINIZIONE WSDL: MESSAGGI E INTERFACCIA

```
<message name="CustomerMessage">
  <part name="parameters" element="cust:Customer"/>
</message>
<message name="ReservationMessage">
  <part name="parameters" element="res:Reservation"/>
</message>

<portType name="ServizioPrenotazionePortType">
  <operation name="prenotazione">
    <input message="titan:CustomerMessage"/>
    <output message="titan:ReservationMessage"/>
  </operation>
</portType>
```

DEFINIZIONE WSDL: BINDING - SERVICE

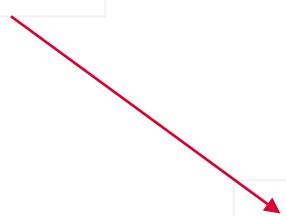
```
<binding name="ServizioPrenotazioneSOAP" type="titan:ServizioPrenotazionePortType">
  <soap:binding style="document" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="prenotazione">
    <soap:operation soapAction="http://www.titan.com/prenotazione"/>
    <input>
      <soap:body use="literal"/>
    </input>
    <output>
      <soap:body use="literal"/>
    </output>
  </operation>
</binding>
<service name="ServizioPrenotazione">
  <port name="ServizioPrenotazioneSOAP" binding="titan:ServizioPrenotazioneSOAP">
    <soap:address location="http://localhost:8080/spring/ws/ServizioPrenotazione"/>
  </port>
</service>
```

STILI DI WEB SERVICES

- 1. RPC
- 2. Document



Il body della chiamata è un documento XML che costruisce la chiamata in stile *remote procedure call*.
La struttura del documento è fissa.



Il body della chiamata è un singolo documento XML. La struttura del documento può essere di qualsiasi tipo (ovviamente definito).

CARATTERISTICHE DI UNA ARCHITETTURA

- Scalabilità e affidabilità sono misure di come l'applicazione può essere servita agli utenti finali.
- Se la nostra applicazione può servire milioni di utenti senza notevoli tempi di inattività, allora possiamo dire che il sistema è altamente scalabile e affidabile.
- **Scalabilità e disponibilità** sono probabilmente i fattori principali per la progettazione di una buona architettura.

Esempio:

Scalabilità = l'applicazione dovrebbe essere in grado di servire milioni di utenti.

Disponibilità = l'applicazione dovrebbe essere disponibile 24 ore su 24, 7 giorni su 7.

Manutenibilità = l'applicazione dovrebbe svilupparsi per diversi anni.

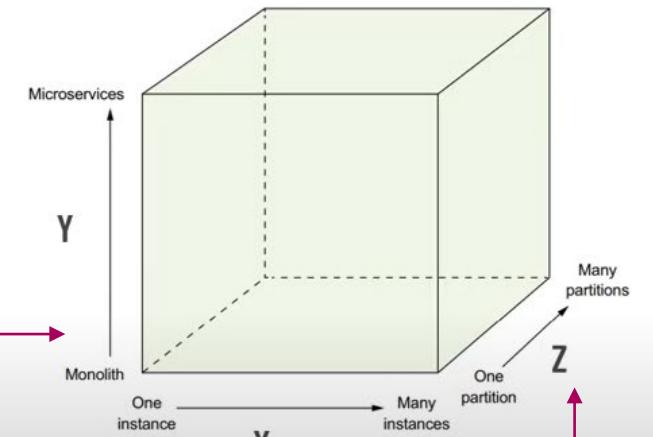
Efficienza = l'applicazione dovrebbe rispondere a una latenza accettabile come $< t$ sec

Definisce tre modi per scalare un'applicazione su tre assi x, y, e z.

Asse X - scalabilità tramite duplicazione dell'applicazione

Asse Y - scalabilità tramite separazione di funzionalità

Asse Z - scalabilità sui dati

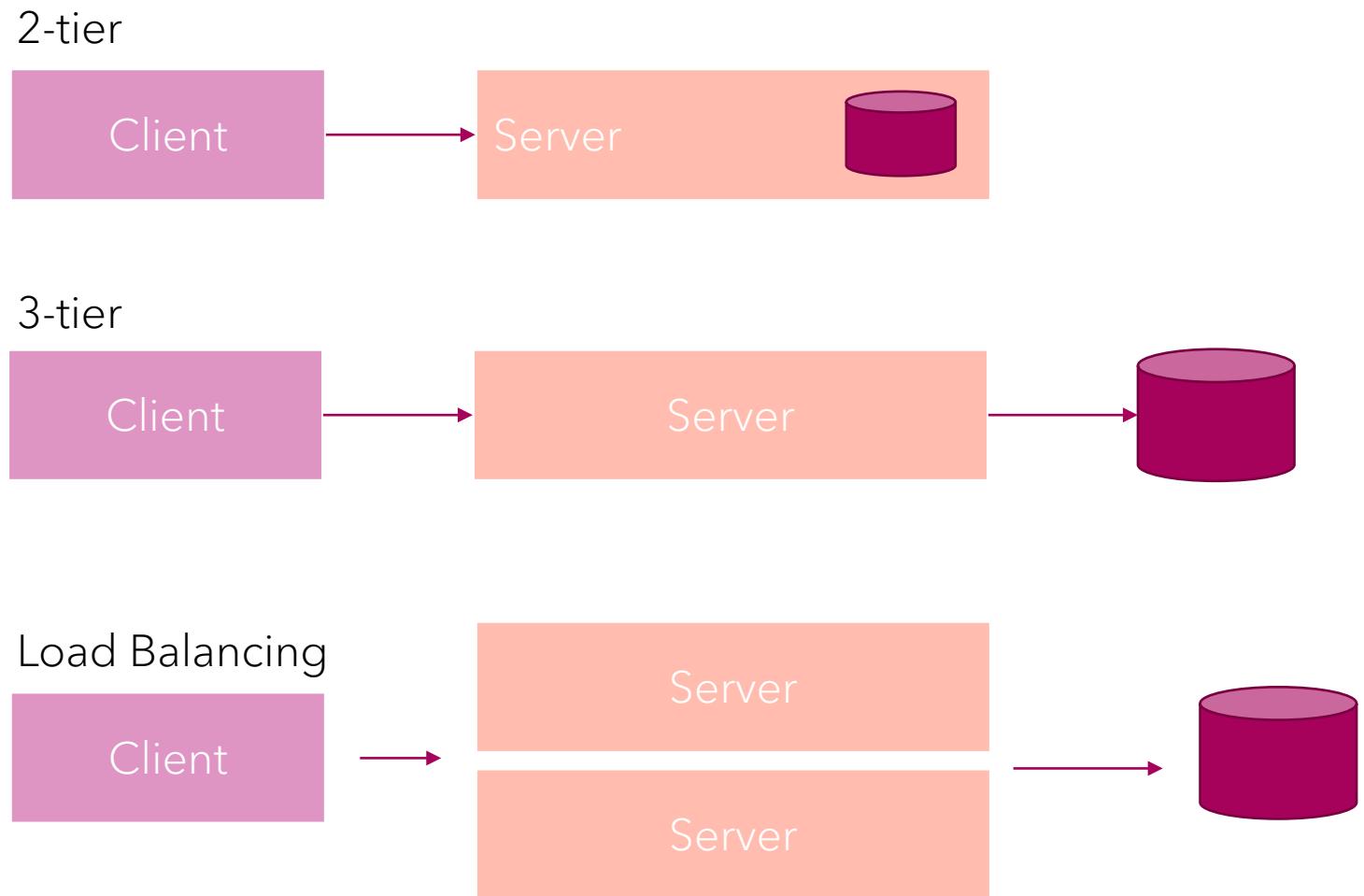


Replicazione app + load balancer

Ogni applicazione si occupa di un sottoinsieme dei dati

Suddivisione applicazioni in web services

ARCHITETTURA MONOLITICA



ARCHITETTURA MONOLITICA

Aggiornamento costoso

L'aggiornamento di un software monolitico comporta necessariamente l'aggiornamento dell'intera applicazione e l'installazione attraverso un nuovo eseguibile, a cui corrisponde una nuova versione del software stesso. Si tratta di un aspetto poco economico quando si avrebbe ad esempio la necessità di risolvere un semplice bug in una funzione. Non sarebbe sufficiente aggiornare il relativo componente, ma andrebbe rilasciata una nuova versione dell'intero applicativo.

Flessibilità ridotta

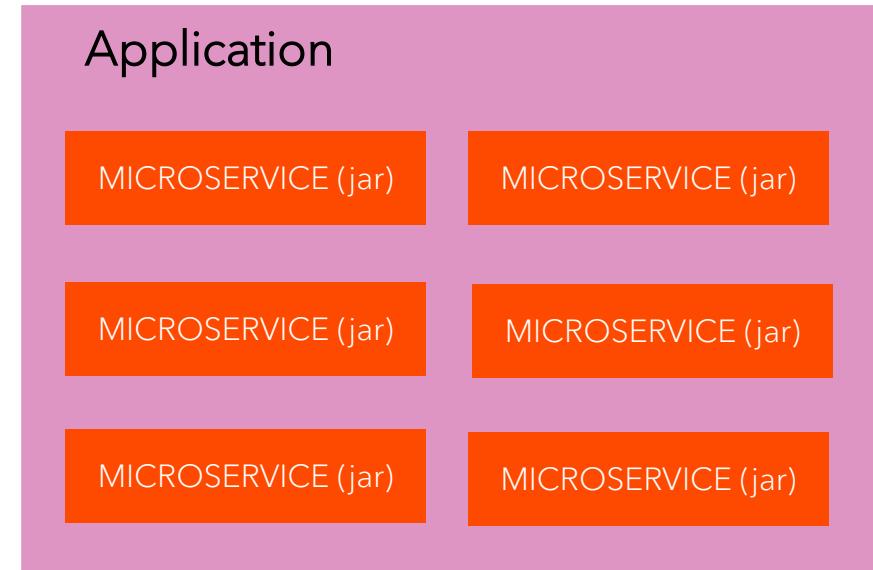
L'architettura monolitica prevede l'impiego di un unico linguaggio di programmazione per lo sviluppo dell'intera applicazione. Tale caratteristica risulta scarsamente strategica nei confronti della grande varietà tecnologica attualmente disponibile, verso cui avrebbe tutto il senso approcciarsi scegliendo la tecnologia più indicata per lo sviluppo di ogni singola funzione presente nel software.

Scalabilità

Un problema ad un singolo modulo costringe a scalare l'intera architettura.

MICROSERVIZI

- Servizio piccolo, distribuito, indipendente e altamente coeso.
- La logica applicativa è decomposta in microservizi.
- I microservizi comunicano in genere attraverso protocollo HTTP e dati JSON. La comunicazione può essere anche asincrona.



MICROSERVIZI: PERCHÉ?

- **Aumento della complessità:** Dobbiamo considerare che le applicazioni comunicano con altre applicazioni, i requisiti cambiano, il software necessita di essere aggiornato ed esteso. Le architetture monolitiche hanno il grosso limite di diventare ingestibili per il raggiungimento di dimensioni e configurazioni ragguardevoli.
- **Rilascio veloce:** Il cliente non vuole attendere tempi lunghi per il rilascio di nuove funzionalità.
- **Performance e scalabilità:** In un'applicazione monolitica occorre scalare l'intera applicazione, non abbiamo la possibilità di scalare in modo diretto una parte dell'applicazione che ne ha bisogno.
- **Capacità di gestione del guasto:** Se una parte dell'applicazione non funziona correttamente l'intera applicazione può bloccarsi.

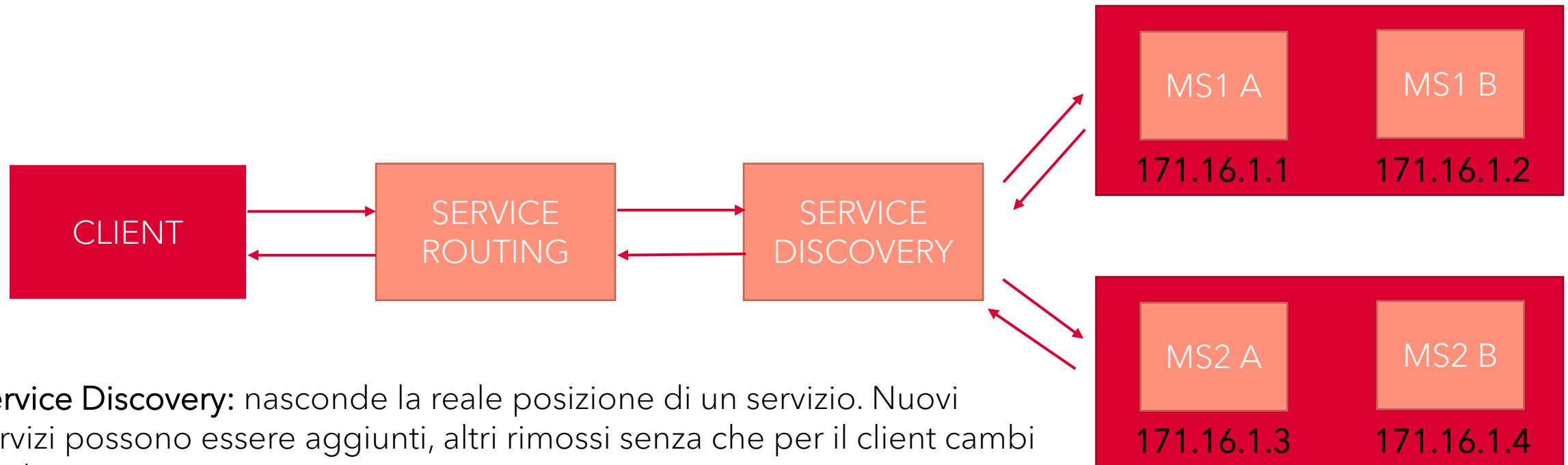
MICROSERVIZI: CARATTERISTICHE

- **Flessibilità:** I servizi possono essere utilizzati per comporre applicazioni. Il loro rilascio è rapido, sono facilmente modificabili e testabili.
- **Gestione del guasto:** Un malfunzionamento si identifica facilmente perché localizzato direttamente sul microservizio. Il malfunzionamento può essere isolato e gestito dinamicamente.
- **Scalabilità:** I microservizi consentono di scalare facilmente le parti dell'applicazione.

MICROSERVIZI: I CONCETTI FONDAMENTALI

- Giusta dimensione: Cosa deve fare un microservizio. La sua granularità.
- Trasparenza di locazione. Un client deve poter raggiungere un servizio usando un URL che non cambia per effetto di avvi o sostituzioni del servizio.
- Resistenza ai malfunzionamenti: Se un servizio ha dei problemi il sistema deve essere in grado di gestire la situazione attraverso tecniche di routing verso un servizio alternativo.
- Ripetibilità: Stessa configurazione per istanze di servizio.
- Scalabilità: Utilizzo di messaggi per minimizzare le comunicazioni dirette tra servizi (latenza di comunicazione).

TRASPARENZA DI LOCAZIONE: ROUTING PATTERN



Service Discovery: nasconde la reale posizione di un servizio. Nuovi servizi possono essere aggiunti, altri rimossi senza che per il client cambi qualcosa.

Service Routing: Fornisce un singolo URL di accesso. Generalmente collabora con il service discovery recuperando preliminarmente l'URL effettivo per contattare il servizio. Punto di accesso per la gestione della sicurezza.

PERFORMANCE: ASPETTI

- Client-side balancing

Mettere in cache la location delle istanze di servizio da parte del client.

- Client/Server-side balancing

Mettere in cache la location delle istanze di servizio da parte del gateway.

- Circuit breakers

Prevenire che un client continui ad invocare un servizio che non sta funzionando correttamente.

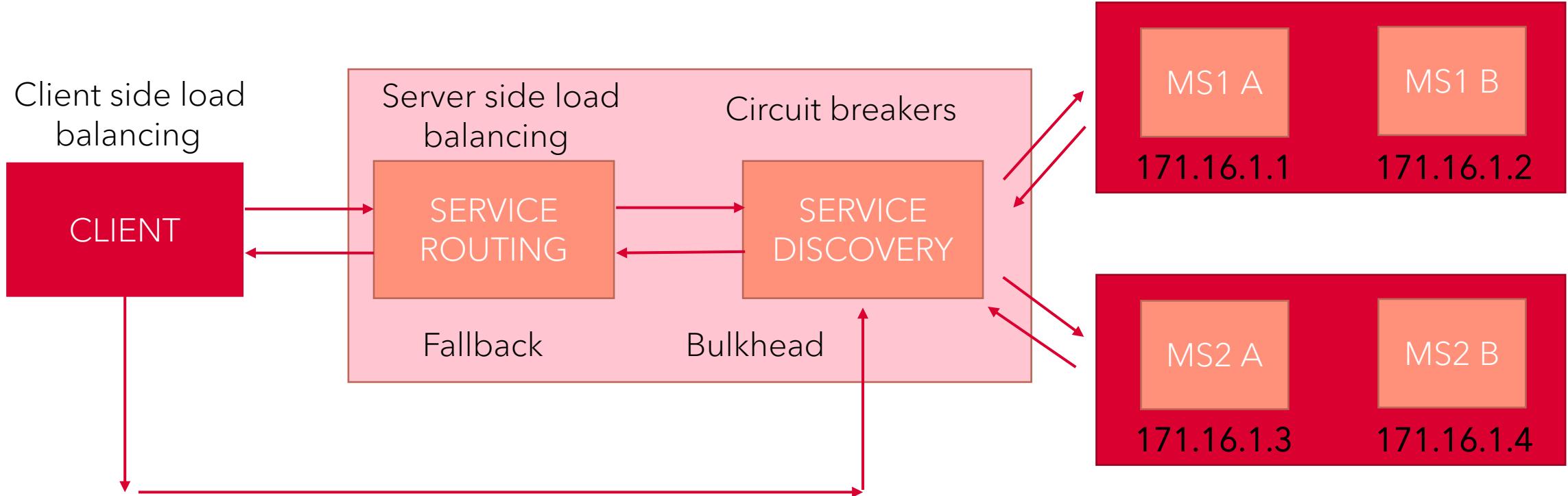
- Fallback

Se un servizio non è raggiungibile per qualsiasi motivo fornire un meccanismo in grado di ridirigere verso un servizio alternativo.

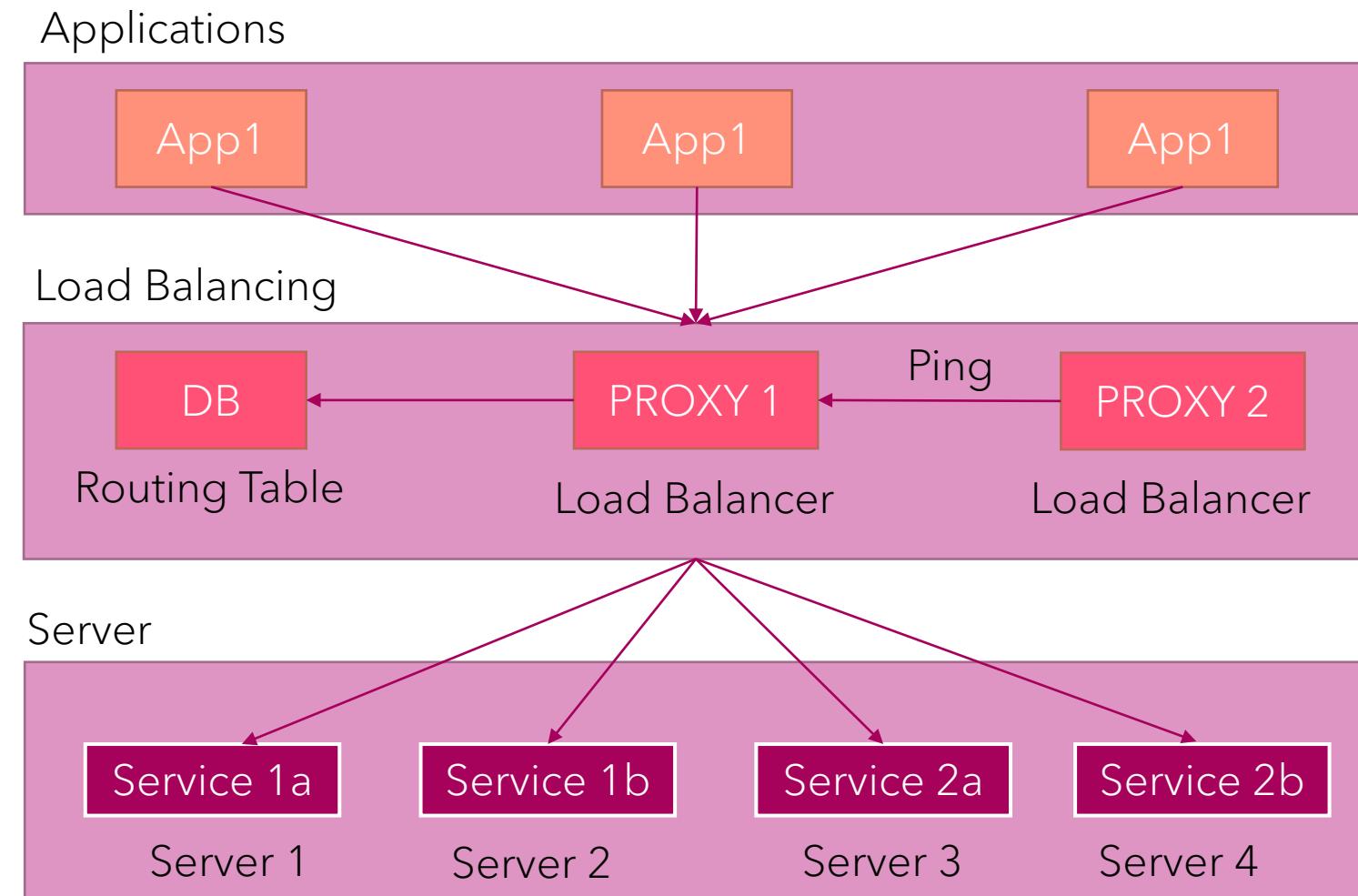
- Bulkhead

Evitare che l'utilizzo di più microservizi da parte di un client non si traduca in un disservizio a causa di un malfunzionamento di uno o più microservizi interessati.

TRASPARENZA DI LOCAZIONE E PATTERNS DI RESISTENZA AL GUASTO



SERVICE LOCATION TRADIZIONALE



SERVICE LOCATION TRADIZIONALE PROBLEMI

- Singolo punto di accesso

Anche se può essere reso altamente disponibile, per sistemi altamente complessi i costi per scalare l'applicazioni diventano enormi.

- Scalabilità orizzontale limitata

Vincoli sulle capacità di ridondanza e sui costi di licenza dei server.

- Gestito staticamente

Non possiamo registrare e rimuovere servizi dinamicamente.

- Complesso

Regole di routing da gestire manualmente.

SERVICE DISCOVERY COME SOLUZIONE NEI MICROSERVIZI

Alta Disponibilità: Molteplicità di nodi di recupero di servizi aggiunti e rimossi dinamicamente.

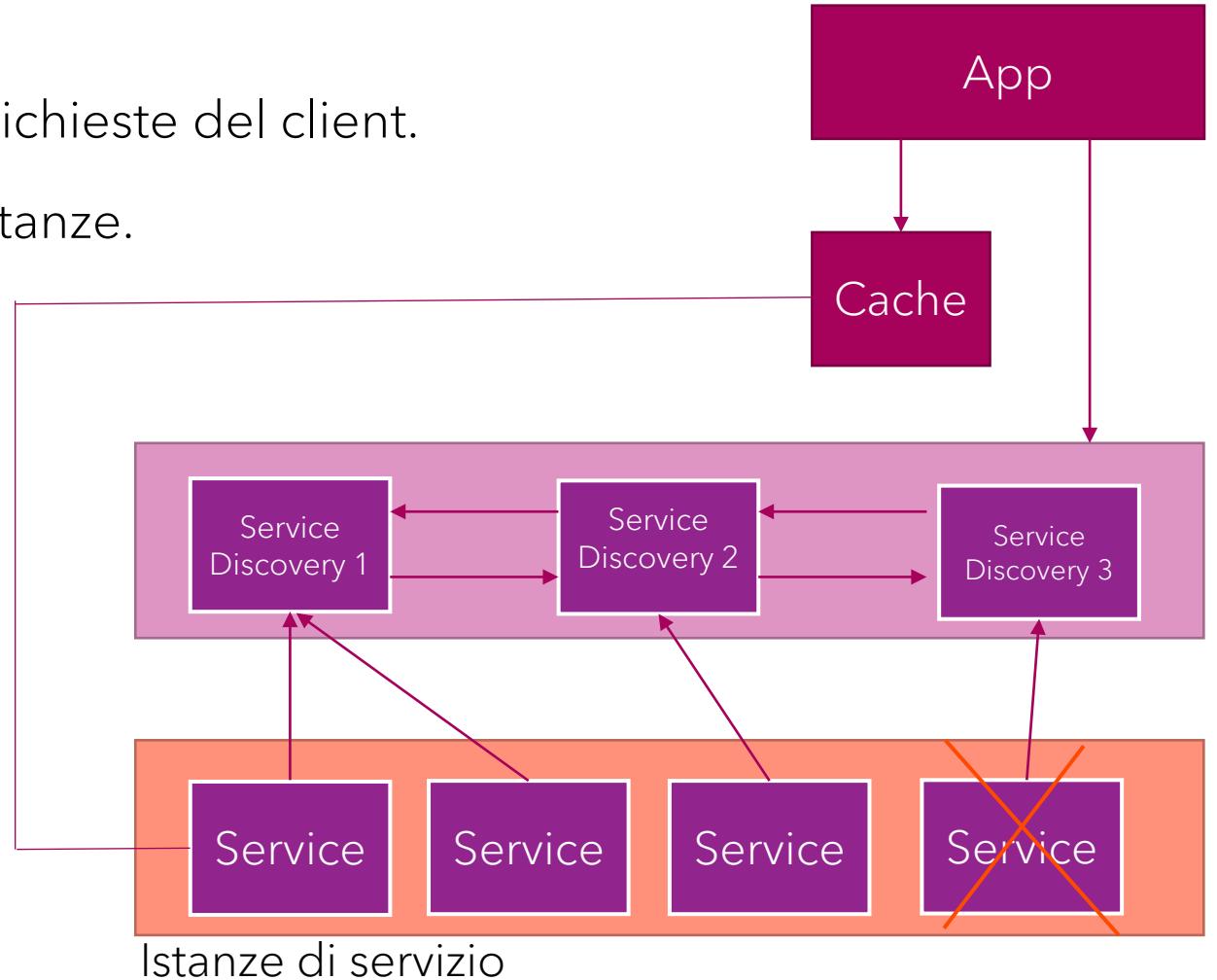
Load Balancing: Rimuove la staticità delle architetture tradizionali. Il suo comportamento è dinamico.

Client cache: Il client può mettere in cache i servizi di cui ha ottenuto la risoluzione.

Tolleranza al guasto: Il discovery servizi rimuove servizi che non sono più funzionanti.

SERVICE DISCOVERY

- Service Registration.
- Individuazione del servizio che deve gestire le richieste del client.
- Condivisione delle informazioni tra le diverse istanze.
- Monitoring dello stato dei servizi.
- Diverse istanze di service discovery, nessun load balancer davanti.
- Un servizio si registra presso un discovery.
- Condivisione dinamica di informazioni peer-to-peer.

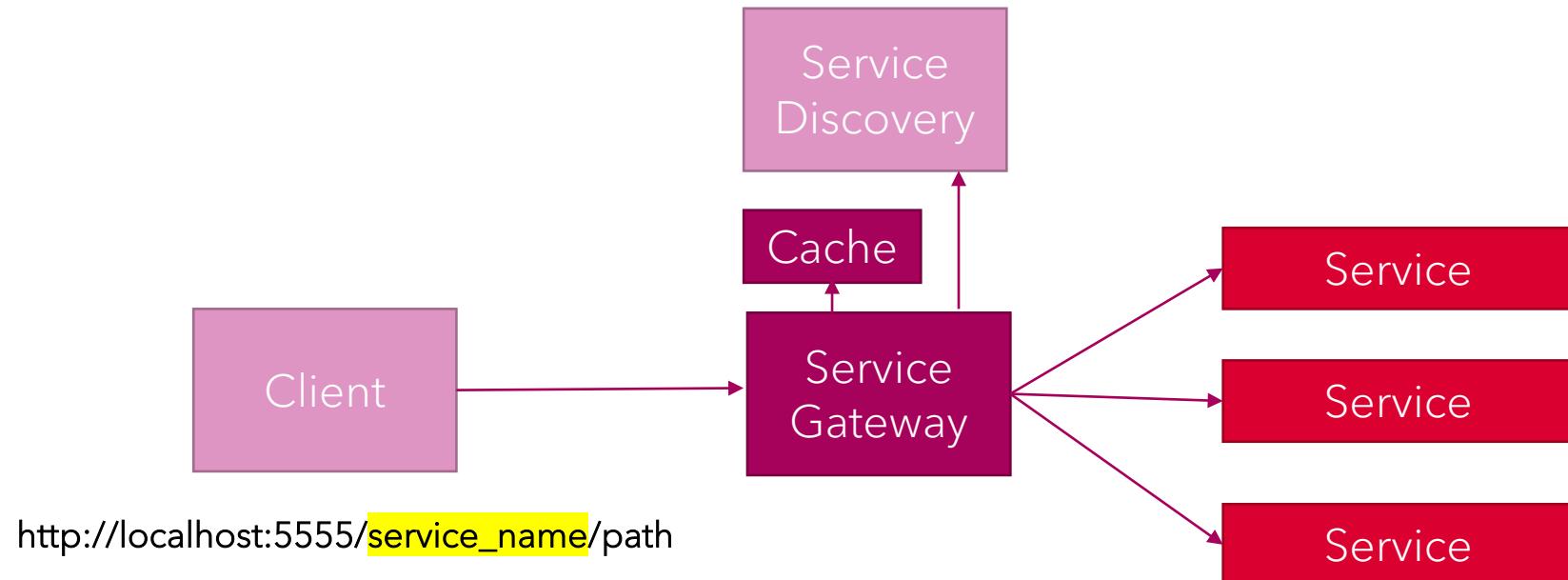


SERVICE ROUTING

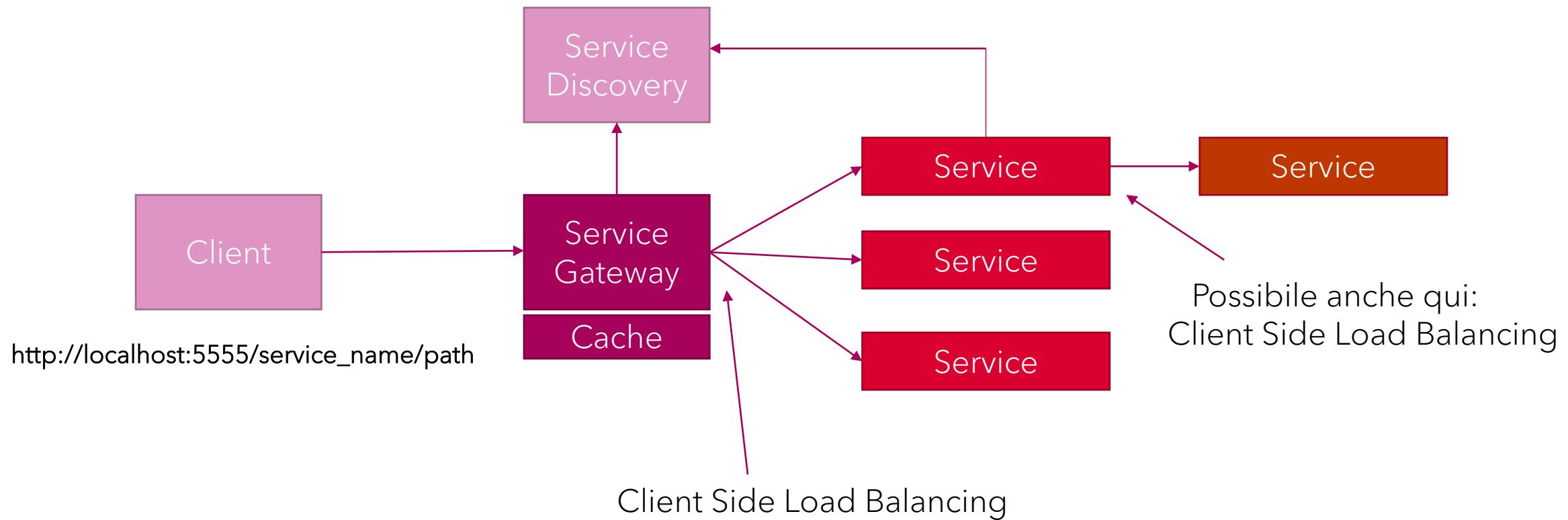
- Gestire la sicurezza, il logging ed il monitoring a livello di singoli servizi è problematico.
- L'idea è quella di avere un punto di accesso che svolga queste funzionalità.
- Questo punto di accesso è il gateway.
- Ogni chiamata di servizio passa attraverso il gateway.

SERVICE ROUTING

- Gestire la sicurezza, il logging ed il monitoring a livello di singoli servizi è problematico.
- L'idea è quella di avere un punto di accesso che svolga queste funzionalità.
- Questo punto di accesso è il gateway.
- Ogni chiamata di servizio passa attraverso il gateway.



SERVICE DISCOVERY: COMUNICAZIONE TRA SERVIZI



SERVICE ROUTING

- Static routing

Un servizio è accessibile solo attraverso gateway. Un client deve solo conoscere un endpoint per richiamare tutti i servizi.

- Dynamic routing

Il gateway elabora l'URL, contatta il service discovery e ridirige verso il servizio.

- Authentication and authorization

Il gateway è il punto nel quale può avvenire l'autenticazione e l'autorizzazione per l'accesso alle risorse,

- Metric collection and logging

Man mano che le chiamate passano nel gateway, il gateway si preoccupa di collezionare informazioni attraverso sistemi di logging.

Perché il gateway non è un singolo punto di fallimento?

Perché è un servizio di cui possiamo avere una molteplicità di istanze. Deve essere mantenuto leggero e non conservare informazioni di stato.

SPRING EUREKA - ZUUL - RIBBON

- Con Spring Boot possiamo realizzare facilmente un:
 - Service Discovery con Eureka
 - Service Gateway con Zuul
 - Client cache con Ribbon

SPRING EUREKA

Eureka fornisce allo stesso tempo un server e client discovery. In produzione generalmente abbiamo un cluster di Eureka servers ed essi si registrano a vicenda (peer registries)

Per default Eureka utilizza un client **heartbeat** per determinare se un client è **UP**. Se non specificato il **Discovery Client** non aggiorna il **current health check status** di un'applicazione utilizzando Spring Boot Actuator.

Come conseguenza, dopo una registrazione ricevuta con successo, Eureka mostra sempre un **UP state**. Questo comportamento può essere modificato abilitando **l'Eureka health checks**. Questa abilitazione attiva la propagazione dello stato verso il server Eureka. Come conseguenza ogni applicazione che non invia informazioni ad Eureka viene cambiata di stato non essendo più UP.

Evitare che Eureka registry se stesso e cerchi altri server:

eureka.client.register-with-eureka=false
eureka.client.fetch-registry=false

Abilitazione health check ed URL di Eureka:

eureka.client.healthcheck.enabled=true
eureka.client.service-url.defaultZone=http://192.168.38.143:\${server.port}/eureka/

SPRING ZUUL

- Service Gateway
- Si può registrare come client con un Eureka Server
- Utilizza Ribbon come client side load balancing

eureka.instance.preferIpAddress=true

eureka.client.registerWithEureka=true

eureka.client.fetchRegistry=true

eureka.client.serviceUrl.defaultZone=http://192.168.38.143:\${eureka.port}/eureka/

management.endpoints.web.exposure.include=routes

hystrix.command.default.execution.isolation.thread.timeoutInMilliseconds=10000

zuul.sensitiveHeaders=

DOCKER

Docker as a Software

Docker è un software che consente una virtualizzazione a livello di Sistema Operativo, conosciuta come "containerization". Rilasciato per la prima volta nel 2013 dalla Docker, Inc.

Docker as a CLI tool

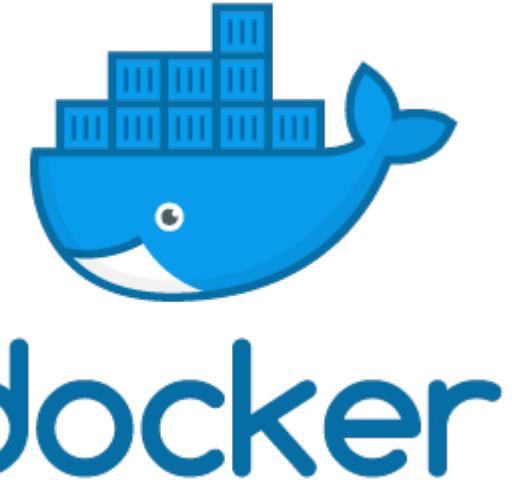
Docker è un programma a riga di comando, con un processo in background, ed un set di servizi per risolvere i più comuni problemi legati allo sviluppo e distribuzione del software.

Docker as a Platform

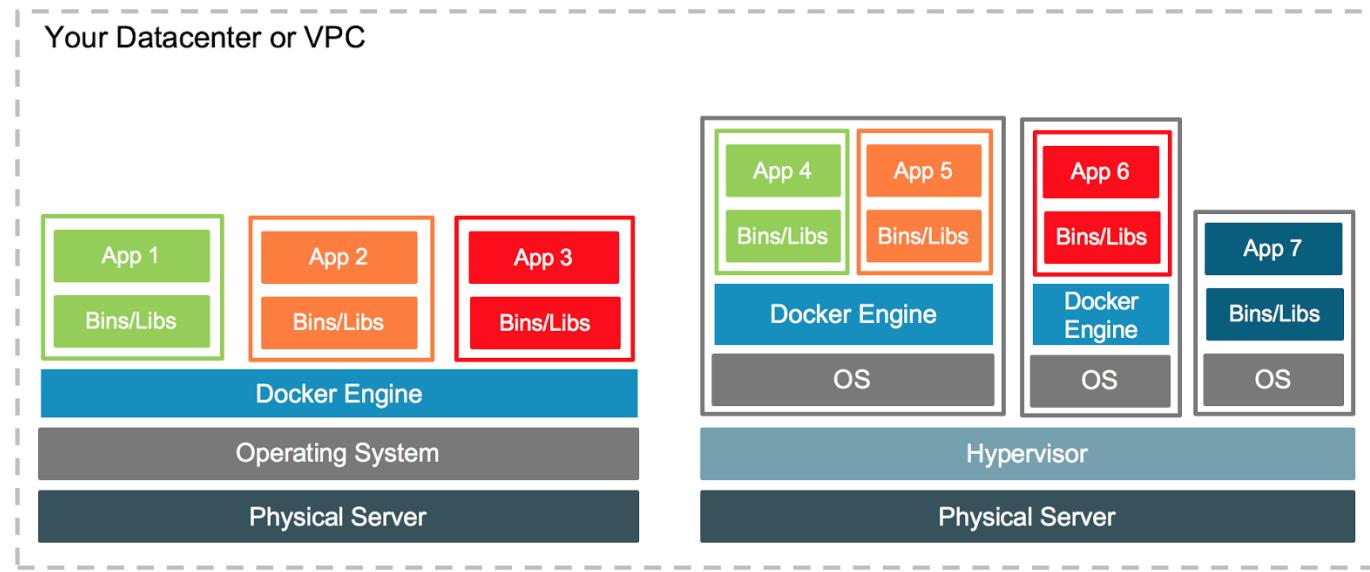
Docker è una piattaforma a container che racchiude un'applicazione e tutte le sue dipendenze in un docker container consentendone l'esecuzione su ogni sistema.

Docker as a Product

Docker è il leader nel containerization market.



DIFFERENZE TRA CONTAINERS E VIRTUAL MACHINES



Con una Virtual Machine sopra l'hardware, abbiamo l'Hypervisor. L'Hypervisor è utilizzato per virtualizzare l'hardware in modo tale che sia condiviso da diversi sistemi operativi.

A differenze delle virtual machines i containers sono normali processi. Docker virtualizza il Sistema Operativo.

CONFRONTO

Virtual Machines	Docker
Each VM runs its own OS	All containers share the same Kernel of the host
Boot up time is in minutes	Containers instantiate in seconds
VMs snapshots are used sparingly	Images are built incrementally on top of another like layers. Lots of images/snapshots
Not effective diffs. Not version controlled	Images can be diffed and can be version controlled. Dockerhub is like GITHUB
Cannot run more than couple of VMs on an average laptop	Can run many Docker containers in a laptop.
Only one VM can be started from one set of VMX and VMDK files	Multiple Docker containers can be started from one Docker image

DOCKER CLIENT VS SERVER

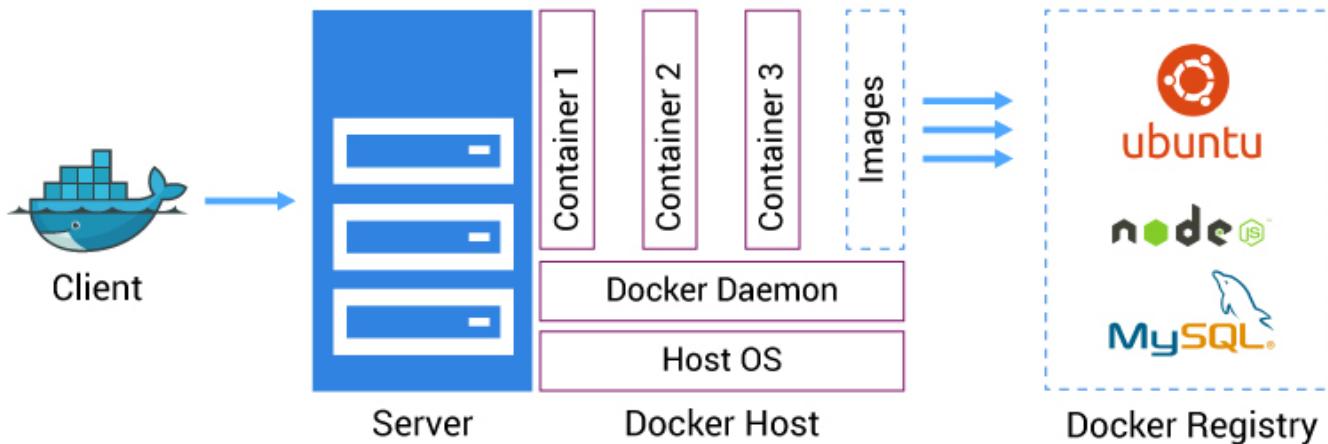


Image courtesy: <https://www.xenonstack.com/blog/docker-application-solutions/>

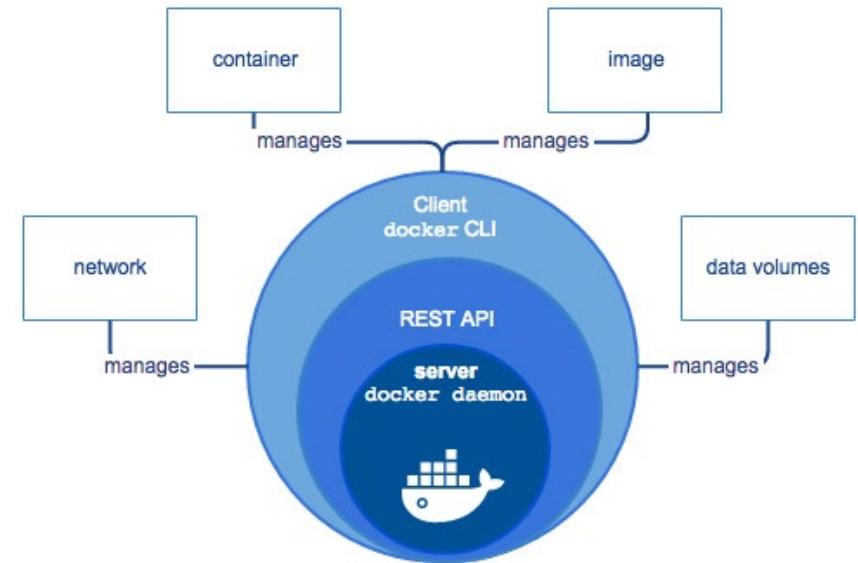


Image courtesy: <https://docs.docker.com/engine/docker-overview/>

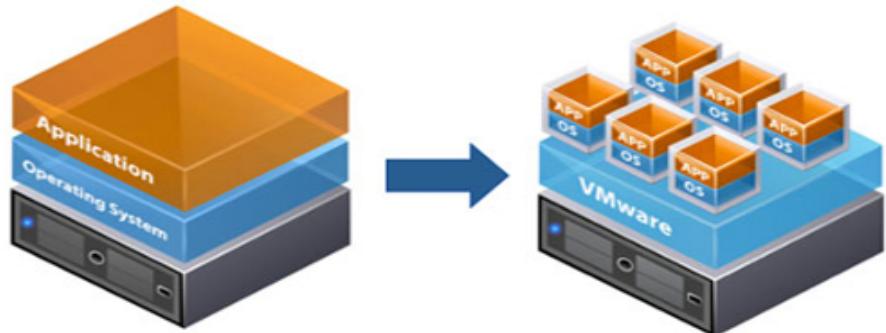
IMMAGINE E CONTAINER DOCKER

Un'immagine è un file immutabile che rappresenta la definizione di un container.

Le immagini sono create da un processo di building memorizzare in un Docker registry (registry.hub.docker.com).

Possiamo dire che, se un'immagine è una classe, allora un container è un'istanza della classe—a runtime object.

RECAP. VIRTUALIZZAZIONE



Un esempio: eseguire un server web sul computer. Vogliamo tenerlo separato dal sistema operativo e dalle applicazioni.

Per fare ciò, è possibile eseguire una **macchina virtuale** contenente il server web.

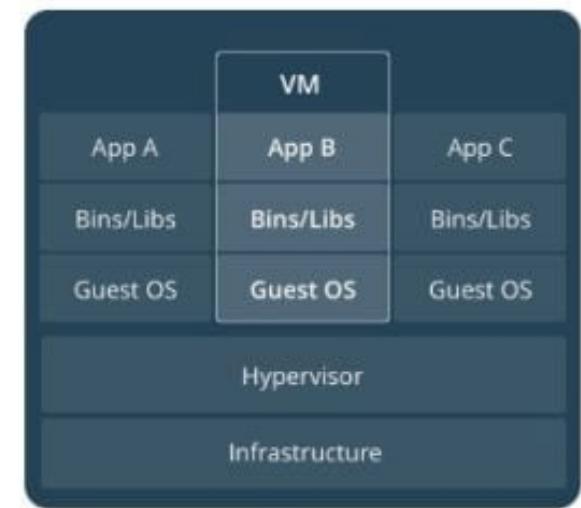
Funziona come un computer separato, ma utilizza il processore e la RAM del computer. Quando si avvia la macchina virtuale, l'intero sistema operativo viene visualizzato in una finestra all'interno del sistema operativo che si sta utilizzando.

RECAP. DOCKER

- Docker è un software opensource per il *deployment* e la gestione dei **container** ovvero dei contenitori all'interno dei quali vengono inserite ed eseguite applicazioni specifiche.



Container



Virtualizzazione

INSTALLAZIONE VMWARE

- Download: <https://www.vmware.com/go/getplayer-win>
- Versione 16.2.1 : <https://customerconnect.vmware.com/downloads/details?downloadGroup=WKST-PLAYER-1621&productId=1039&rPId=85399>
 - Ubuntu 18.04.3 Bionic Beaver
 - Username: osboxes
 - Password: osboxes.org
 - Attenzione! Selezionare tab VMWARE. Scegliere 64 bit o 32 bit a seconda dell'architettura del proprio PC.

<https://www.osboxes.org/ubuntu/#ubuntu-1804-info>

- Guida installazione immagine: <https://www.osboxes.org/guide/>

How to attach/configure image with VMware (VMDK file version)?

- How to install VMware Tools? All'interno della guida VMware

INSTALLAZIONE

Se ci sono errori di lock seguire la seguente procedura:

- 1. sudo killall apt apt-get
- 2. sudo rm /var/lib/apt/lists/lock
- 3. sudo rm /var/lib/dpkg/lock
- 4. sudo rm /var/lib/dpkg/lock-frontend
- 5. sudo dpkg --configure -a
- 6. sudo apt update

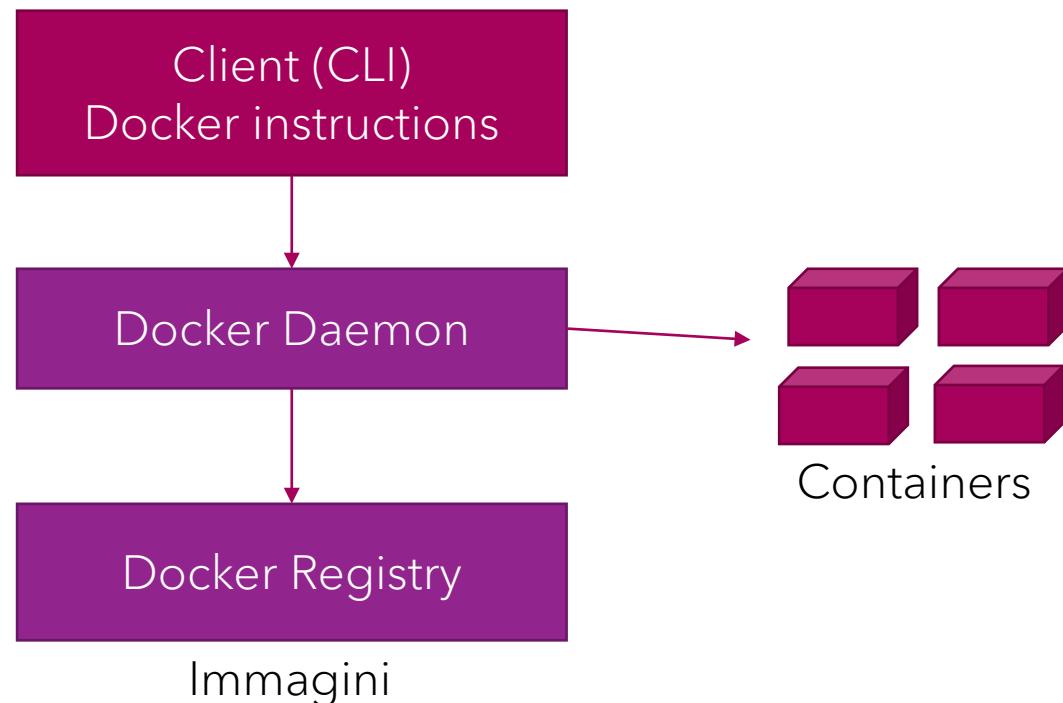
Installazione Desktop Tools:

- 1. sudo apt install open-vm-tools-desktop
- 2. sudo apt install net-tools

INSTALLAZIONE DI DOCKER PER LINUX UBUNTU 18.04

1. sudo apt install apt-transport-https ca-certificates curl software-properties-common
2. curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo apt-key add -
3. sudo add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/ubuntu bionic stable"
4. sudo apt-get update
5. sudo apt-get install docker-ce docker-ce-cli containerd.io
6. sudo service docker start (stop per arrestare il servizio)
7. sudo docker run hello-world

DOCKER ENGINE



- Il Docker Engine è un'applicazione client-server costituita da:
 - Un server che è un processo daemon (dockerd)
 - Un API REST che specifica le interfacce che i programmi possono usare per comunicare con il demone
 - Un'interfaccia a linea di comando (CLI) (docker)
- Poiché viene utilizzata un'architettura client-server, il client istruisce tramite le API REST il daemon process che si occupa di della gestione dei container. Possono funzionare sia sulla stessa macchina o su macchine diverse.

DOCKER CLI

- Il Docker CLI è il modo principale per interagire con Docker. Mediante le API REST il Docker CLI invia una richiesta al demone Docker per eseguire un particolare task. I comandi più comuni sono i seguenti:

docker pull

docker run

docker start <container>

docker stop <container>

IMMAGINE DOCKER, PIÙ IN DETTAGLIO

- Un'immagine è un modello di sola lettura con le istruzioni per la creazione di un contenitore Docker.
- Tutte le immagini Docker si basano a loro volta su un'altra immagine. L'immagine base è tipicamente un sistema operativo linux. Per mantenere la dimensione delle immagini più piccola possibile si usano di solito pacchetti di sistemi operativi come *Alpine* che è una distribuzione Linux di solo 5MB.
- Si possono creare le proprie immagini o si possono utilizzare quelle create da altri e pubblicate in un registro.
- Per costruire invece la propria immagine, si crea un Dockerfile.

CONTAINER DOCKER, PIÙ IN DETTAGLIO

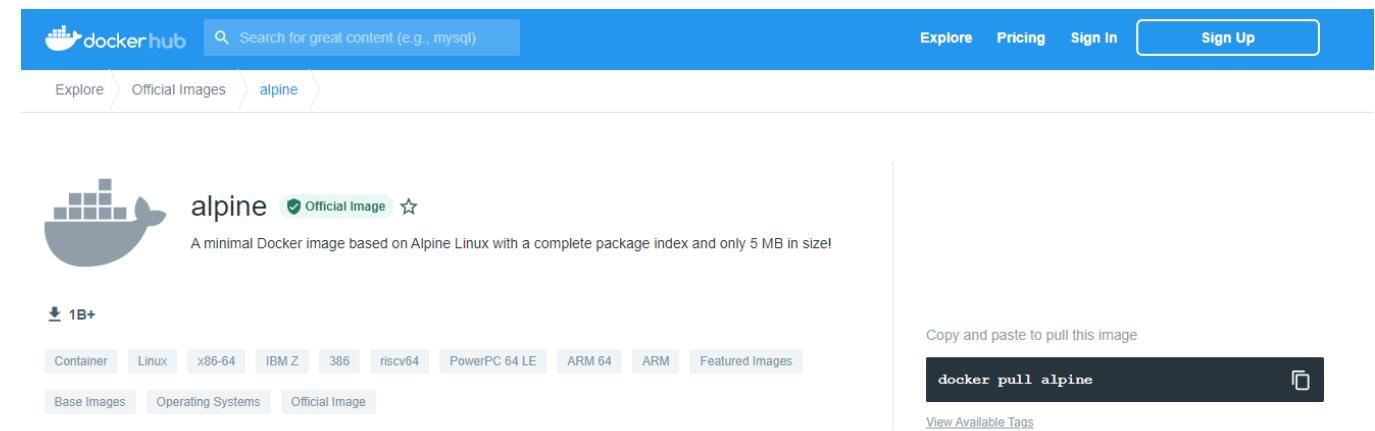
- Quando un'immagine viene eseguita su un host, questa genera un processo chiamato container. Essendo Docker scritto nel linguaggio di programmazione [Go](#), sfrutta diverse caratteristiche del kernel Linux per fornire le sue principali funzionalità.
- Per impostazione predefinita, un container è relativamente ben isolato dagli altri container e dalla macchina host.
- Mediante i comandi del Docker CLI è possibile creare, avviare, fermare, spostare o cancellare un container. Inoltre, un container può essere collegato a una o più reti e/o ad un archivio.

CONTAINER DOCKER, PERSISTENZA

- Un aspetto importante da comprendere è che quando un container è in esecuzione, le modifiche vengono applicate al livello del container.
- Pertanto, quando il container viene rimosso, ogni cambiamento del suo stato non viene salvato. Per ovviare alla perdita dello stato si utilizzano storage persistenti mediante la definizione di **volumi**.

DOCKER: COMANDI FONDAMENTALI

- sudo docker pull alpine
- sudo docker run --name course_postgres -e POSTGRES_PASSWORD=postgres -p 5432:5432 -d alessandrozoia/course_repository
- sudo docker image ls
- sudo docker ps
- sudo docker ps -a
- sudo docker image rm <nome imagine>
- sudo docker rm <nome container>
- sudo docker exec -it <nome container> bash
- sudo docker exec postgres cat /etc/hosts



DOCKERFILE

Un Dockerfile è un documento di testo che contiene in sequenza tutti i comandi che un utente può chiamare sulla linea di comando per assemblare un'immagine. Per costruire un'immagine è necessario lanciare il comando:

```
docker build
```

All'interno della directory che contiene il Dockerfile. Un esempio di Dockerfile:

```
FROM adoptopenjdk:11-jre-hotspot
ARG JAR_FILE=*.jar
COPY ${JAR_FILE} Ms1-1.0.jar
ENTRYPOINT ["java", "-jar", "Ms1-1.0.jar"]
```

Il nome del file è esattamente questo
di default

La modalità di utilizzo prevede la realizzazione di una directory con il Dockerfile e tutto l'occorrente necessario.
**Non utilizzare la root directory del Sistema operativo come path per il contesto di building,
tutta la root verrebbe trasferita.**

DOCKERFILE

Con il comando docker build, la directory corrente è chiamata ***build context***.

Per default, il Dockerfile è assunto posizionato qui. E' possibile specificare una differente locazione.

Ricorsivamente tutti i contenuti della directory corrente sono inviati al processo di building.

DOCKERFILE

Nel caso più semplice con una directory contenente il Dockerfile, ci posizioniamo al suo interno ed eseguiamo

```
sudo docker build .
```

Se vogliamo specificare il Dockerfile proveniente da un altro percorso:

```
docker build . -f /path/to/a/Dockerfile
```

Per associare un nome:

```
docker build . -t my_image
```

DOCKERFILE

FROM	immagine di base da utilizzare per le successive istruzioni
WORKDIR	imposta l'attuale directory di lavoro per le istruzioni RUN, CMD, ENTRYPOINT, COPY e ADD
COPY	copia i file nel contenitore
ADD	simile al comando COPY supporta l'estrazione di archivi e di URL remote
RUN	esegue tutti i comandi in un nuovo livello in cima all'immagine corrente e creerà un nuovo livello disponibile per i prossimi passi nel Dockerfile
CMD	fornisce i valori predefiniti per un contenitore in esecuzione
ENTRYPOINT	indica il comando da eseguire all'avvio del container
ENV	imposta le variabili d'ambiente dell'immagine
VOLUME	crea una directory sull'host che viene montata su un percorso specificato all'interno dell'istruzione
LABEL	aggiunge metadati ad un'immagine come coppia chiave/valore

COS'E' UN REPOSITORY?

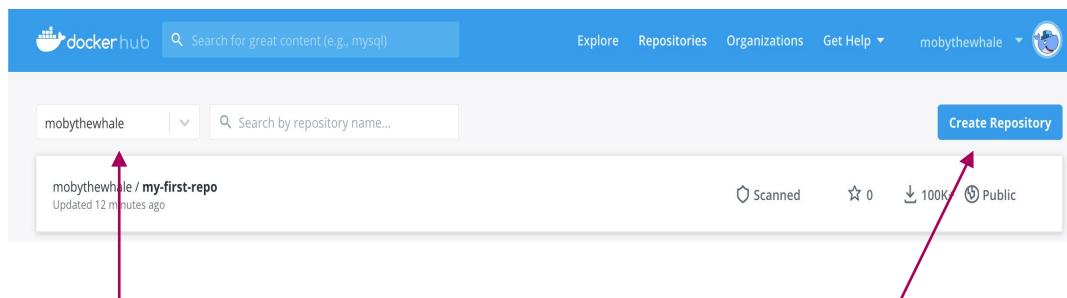
Prima di andare oltre con i Dockerfile vediamo cos'e' un repository. Un repository è un contenitore di immagini.

Docker Hub ne è un esempio. Consente di condividere le immagini con il gruppo di lavoro, o con tutto il mondo.

Un'immagine è pubblicata con il comando:

docker push

CREARE UN REPOSITORY



Utente/Organizzazione

Creare un repository

<https://hub.docker.com/>

docker push <hub-user>/<repo-name>:<tag>

A screenshot of the 'Create Repository' form on Docker Hub. At the top, it says 'Repositories > Create'. On the left, there's a dropdown for 'mobythewhale' and a 'Create Repository' button. On the right, there's a 'Name' input field with 'mobythewhale' selected. Below the name field is a 'Description' input field. Under 'Visibility', there are two options: 'Public' (selected) and 'Private'. The 'Public' option is described as 'Public repositories appear in Docker Hub search results'. The 'Private' option is described as 'Only you can view private repositories'. At the bottom, there's a section for 'Build Settings (optional)' with a note about 'Autobuild triggers a new build with every git push to your source code repository.' There's also a note about re-linking GitHub or Bitbucket accounts. At the very bottom are three buttons: 'Cancel', 'Create' (highlighted in blue), and 'Create & Build'.

DOCKERFILE

- Le istruzioni vengono eseguite in ordine.
- Un Dockerfile deve iniziare con l'istruzione `FROM`, solo gli argomenti possono essere collocati prima.
- L'istruzione `FROM` rappresenta l'immagine padre dalla quale partire per la creazione di una nuova immagine.

`FROM [--platform=<platform>] <repo>[:<tag>]`

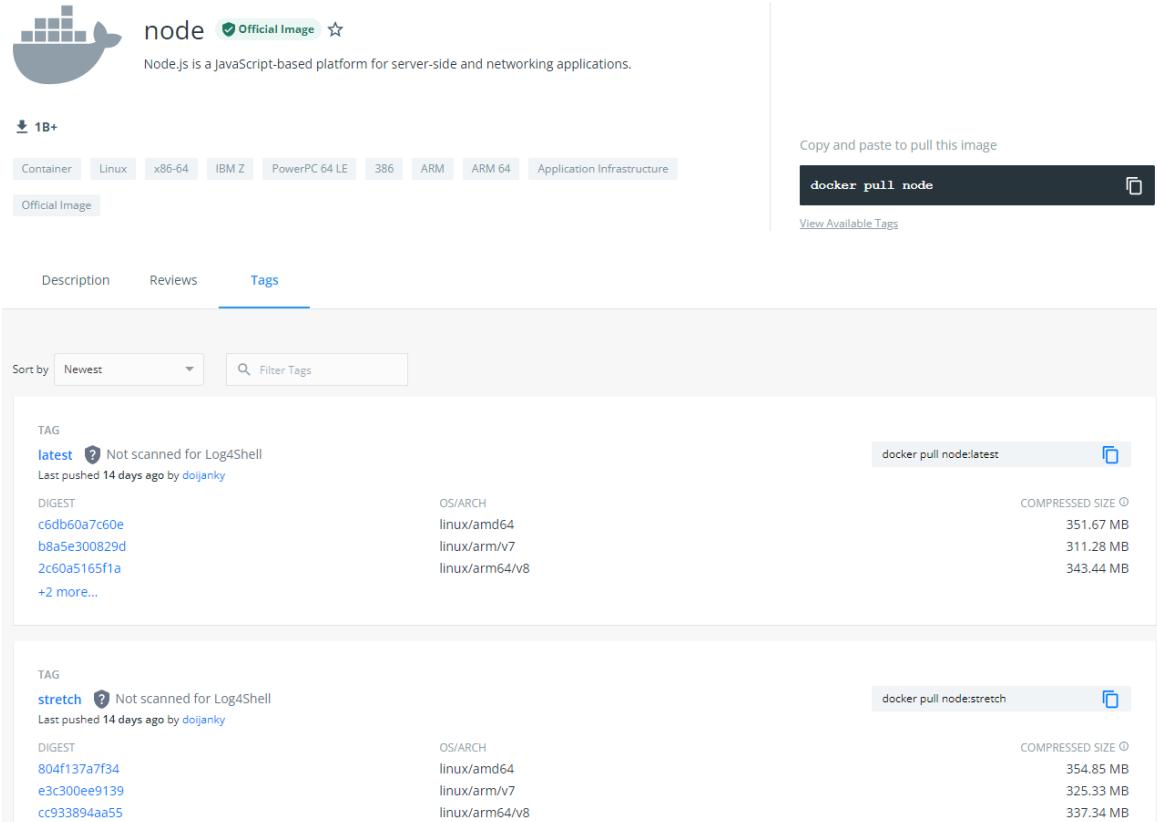
`linux/amd64, linux/arm64, windows/amd64 ...`

Default: la piattaforma dalla quale si effettua il build



Repository

DOCKERFILE



FROM --platform=linux/amd64 node:stretch

Volendo utilizzare gli argomenti:

ARG CODE_VERSION=latest

FROM --platform=linux/amd64 node:\${CODE_VERSION}

.....

DOCKERFILE

- Il comando RUN esegue un'istruzione ed aggiorna l'immagine corrente (un nuovo layer). In pratica sono istruzioni che vengono eseguite alla creazione del container.

RUN <command> (*shell form*, il commando è eseguito in una shell, default `/bin/sh -c` su Linux)
RUN ["executable", "param1", "param2"] (*exec form*)

```
RUN /bin/bash -c 'source $HOME/.bashrc; echo $HOME'
```

```
RUN ["/bin/bash", "-c", "echo hello"]
```

Il comando RUN può essere spezzato su una molteplicità di linee con il carattere \.

1 Layer

```
RUN apt-get -y update \  
 && apt-get install -y libicu-dev
```



Ogni riga del Dockerfile con comandi ADD,COPY E RUN origina un nuovo layer che incrementa le dimensioni di una immagine. La sintassi consente di raggruppare più istruzioni in un solo layer.

Invece di (2 layers):

```
RUN apt-get -y update  
RUN apt-get install -y libicu-dev
```

E SERCIZIO

- Obiettivo: Installare la shell su linux alpine.
- Costruire un Dockerfile che estenda alpine con la bash.

```
FROM alpine:latest
```

```
RUN apk update \
&& apk upgrade \
&& apk add bash \
&& rm -rf /var/cache/*/* \
&& echo "" > /root/.ash_history

RUN sed -i -e "s/bin\\/ash/bin\\/bash/" /etc/passwd

ENV LC_ALL=en_US.UTF-8

WORKDIR /root
```

- Creare l'immagine: ***sudo docker build . -t alpine_ex_repo[:tag]***
 - *Creare il container: ***sudo docker run --name alpine_ex -t -d alpine_ex_repo[:tag]****
 - ***sudo docker exec -it <nome container> /bin/bash***
-
- The diagram consists of two pink arrows originating from the Docker command line in the list above. One arrow points from the tag part of the build command to the label 'Nome container'. Another arrow points from the tag part of the run command to the label 'Nome immagine'.

DOCKERFILE

- L'istruzione CMD fornisce un modalità per l'esecuzione del container. Ci può essere solo un CMD.

CMD ["executable","param1","param2"] (*exec form*)

CMD ["param1","param2"] (*default parameters to ENTRYPPOINT*)

CMD command param1 param2 (*shell form*)

DOCKERFILE

- EXPOSE consente di specificare le porte di ascolto.

EXPOSE 80/udp (tcp...)

Attenzione non pubblica la porta. E' nel run che dobbiamo specificare su quale porta dell'host è mappata la porta che abbiamo esposto.

DOCKERFILE

- Variabili d'ambiente

```
ENV MY_NAME="John"
```

saranno disponibili nel container.

DOCKERFILE

- Directory di lavoro all'interno del container, WORKDIR.
- Utilizzando WORKDIR possiamo specificare la directory nel quale posizionarci nel container quando eseguiamo i comandi RUN, CMD, ENTRYPOINT, COPY, ADD.

WORKDIR /path/to/workdir

DOCKERFILE

L'istruzione ADD copia nuovi file, directory o file remoti dalla `<src>` e li aggiunge al container nel percorso `<dest>`. Il path `<dest>` è assoluto o relativo alla `WORKDIR`.

`ADD <src>... <dest>`

`ADD [<src>,... "<dest>"]`

`ADD test.txt relativeDir/`

Il file è aggiunto alla directory `WORKDIR/relativeDir`.

Se nessuna `WORKDIR` è specificata il path di default è `/`.

DOCKERFILE

Un `ENTRYPOINT` consente di eseguire un container come una applicazione eseguibile.

```
ENTRYPOINT ["executable", "param1", "param2"]
```

Se usato nasconde CMD, possiamo combinare l'uso di CMD ed ENTRYPOINT ma avendo CMD solo come specificatore di parametri.

DOCKERFILE

Container per applicazione Spring Boot:

```
FROM adoptopenjdk:11-jre-hotspot
ARG JAR_FILE=*.jar
COPY ${JAR_FILE} Ms1-1.0.jar
ENTRYPOINT ["java", "-jar", "Ms1-1.0.jar"]
```

DOCKER NETWORKING

Il networking fra container è supportato attraverso i **network drivers**. Per default Docker fornisce due drivers: bridge e overlay. Ogni installazione di Docker include di default 3 networks:

`docker network ls`

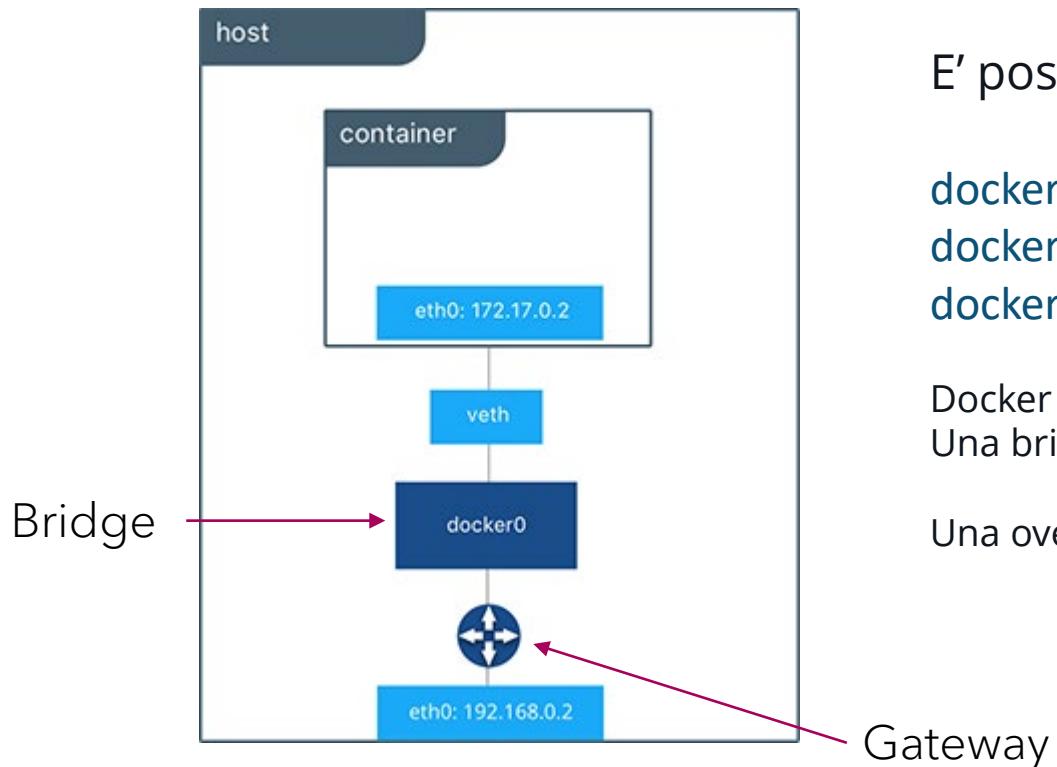
NETWORK ID	NAME	DRIVER
18a2866682b8	none	null
c288470c46f6	host	host
7b369448dccb	bridge	bridge

Disabilita il networking.

Per singoli containers, rimuove l'isolamento tra il container e il docker host, utilizza direttamente il networking dell'host. Per esempio, se lanciamo un container sulla porta 80, l'applicazione del container sarà disponibile sull'ip dell'host.

La rete bridge è particolare. Infatti salvo diverse specifiche Docker lancia sempre un container in questa rete.

DOCKER NETWORK



E' possibile rimuovere un container da una rete disconnettendolo:

`docker network disconnect bridge <container>`
`docker network connect bridge <container>`
`docker network inspect bridge`

Docker Engine supporta sia le bridge networks che le overlay networks.
Una bridge network è limitata ad un singolo host che esegue Docker Engine.

Una overlay network serve per gestire più host.

DOCKER NETWORK PERSONALIZZATA

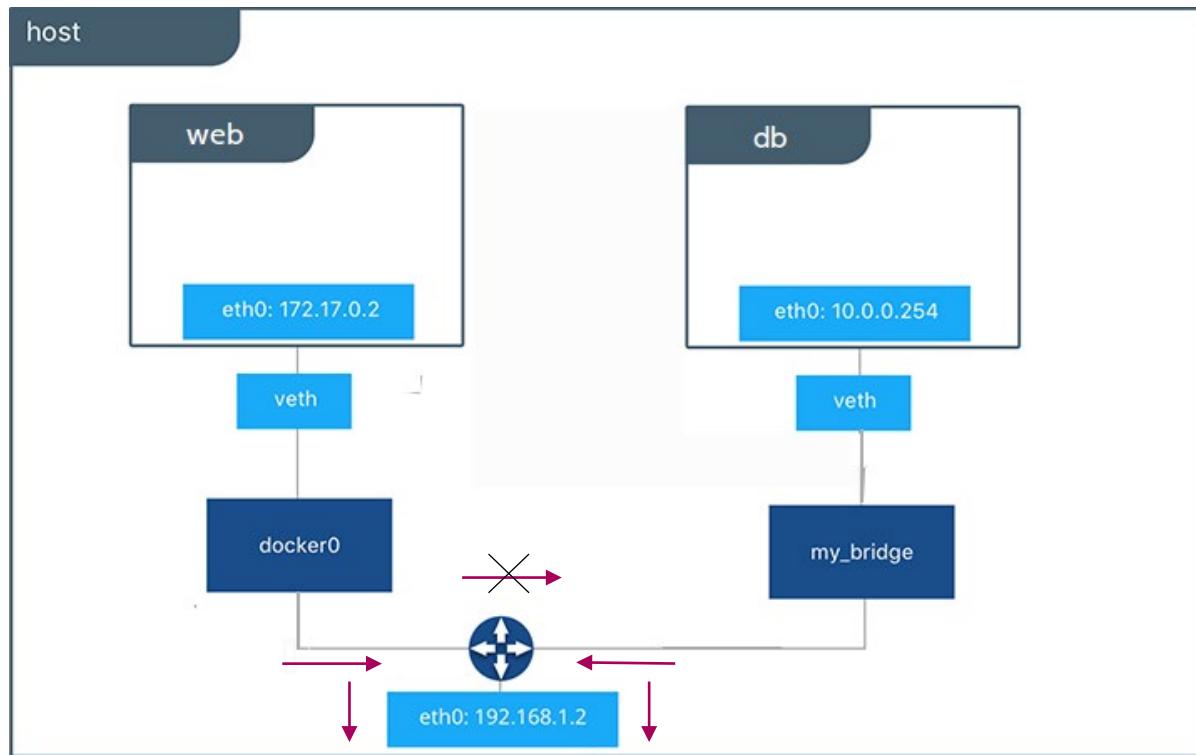
```
docker network create -d bridge my_network
```

```
docker network ls
```

```
docker network inspect my_network
```

```
docker run ... --network=my_network ...
```

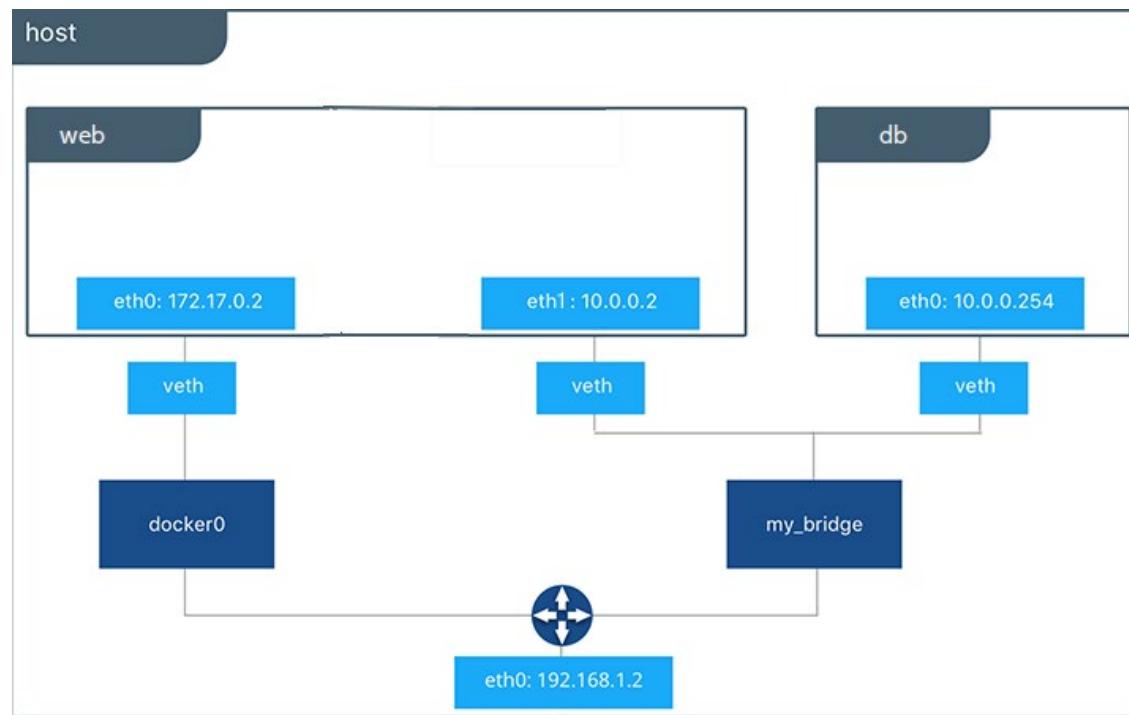
DOCKER NETWORK PERSONALIZZATA



Due container isolati su due reti distinte.
I due container non comunicano.

Ma Docker consente di collegare un container
a più reti.

DOCKER NETWORK PERSONALIZZATA



Il container web è adesso collegato anche alla rete `my_bridge`.

DOCKER NETWORK PERSONALIZZATA

Esercizio:

- Avviare un container sulla rete bridge
- Creare una rete personale
- Aggiungere un altro container alla rete personale e verificare che non comunica con il primo
- Aggiungere il primo anche alla rete personale e verificare che i due container comunicano

`docker network create -d bridge my_network`

`docker network ls / docker network inspect my_network`

`docker run ... --network=my_network ...`

`docker network connect my_network <container>`

DOCKER VOLUME

In Docker esistono due tipologie di dati: quelli **persistenti** e quelli **non persistenti**.

I dati persistenti sono quelli che devono essere preservati nel tempo ed eventualmente condivisi tra più container (es. Database, media, ..).

I dati non persistenti invece sono quelli che possono essere anche persi perché utili solo nel contesto di riferimento, ad esempio i file temporanei.

Docker prevede nativamente delle modalità per la gestione di entrambe le tipologie di dato.

Ogni container è dotato del proprio spazio di memorizzazione dei dati non persistenti che viene poi deallocated nel momento in cui il container viene cancellato.

Per quanto riguarda la gestione dei dati persistenti Docker fornisce due modalità, i **Volumi** e i **Bind Mounts**.

DOCKER VOLUME

Percorsi sul sistema operativo creati e gestiti da Docker. Possiamo crearli con `docker volume create`, o durante la creazione del container.

Quando montiamo un volume in un container, la directory è montata sul container.

Un volume può essere montato su più container. Non sono rimossi automaticamente se nessun container li usa più. Occorre utilizzare `docker volume rm (o prune)` per la rimozione.

Quando si monta un volume, il volume può avere **un nome** oppure anonimo. Agli anonimi è dato un nome randomico.

DOCKER VOLUME

Casi in cui può essere utile un volume:

- Condividere i dati tra più container.
- Persitenza dei dati oltre il tempo di vita del container.
- Memorizzazione di dati su un host remoto oppure in cloud.
- Backup e migrazione di dati tra host, i volumi sono la migliore scelta.
Possiamo fare il back della directory del volume: </var/lib/docker/volumes/<volume-name>>.
- Se l'applicazione richiede un high-performance I/O. I volumi sono memorizzati nella Linux VM piuttosto che nell'host, abbiamo quindi una minore latenza.

DOCKER VOLUME

Esempio:

```
docker volume create my-vol
```

```
docker volume ls
```

```
docker volume inspect my-vol
```

```
docker volume rm my-vol
```

DOCKER VOLUME

- Se avviamo un container con un volume che non esiste, docker lo crea in modo automatico.

```
docker run -d --name devtest --mount source=myvol,target=/app nginx:latest
```

```
docker volume inspect myvol
```

```
[  
 {  
   "CreatedAt": "2022-04-04T01:26:24-04:00",  
   "Driver": "local",  
   "Labels": null,  
   "Mountpoint": "/var/lib/docker/volumes/myvol/_data",  
   "Name": "myvol",  
   "Options": null,  
   "Scope": "local"  
 }]  
]
```

```
docker volume create myvol  
docker volume ls  
docker volume inspect myvol  
docker volume rm myvol
```

Rimozione:

```
docker stop devtest  
docker rm devtest  
docker volume rm myvol
```

DOCKER VOLUME: BINDING

Vogliamo collegare un path assoluto della macchina host. La directory sull'host ha bisogno di essere già esistente quella specificata sul container verrà creata.

```
sudo docker run -d -it --name devtest --mount  
type=bind,source=/home/osboxes/miodisco,target=/hostdisk nginx:latest
```

Quando creiamo un **named volume**, una nuova directory è creata all'interno della Docker storage directory sulla macchina host, e Docker gestisce il contenuto di questa directory.

DOCKER COMPOSE

Compose è un tool per definire ed eseguire delle multi-container Docker applications.

Con Docker Compose, utilizziamo gli YAML per configurare le applicazioni.

Con un singolo comando creiamo le immagini ed i container della nostra applicazione.

L'utilizzo di Docker Compose essenzialmente si articola in 3 step:

1. Definizione di un Docker File*.
2. Definizione dei servizi.
3. Esecuzione di docker-compose up.

* *Un docker file non è obbligatorio*

DOCKER VOLUME IN COMPOSE

- Un volume in docker compose (yml):

```
version: "3.9"
```

```
services:
```

```
frontend:
```

```
image: node:14
```

```
volumes:
```

```
- myapp:/home/node/app
```

```
volumes:
```

```
myapp:
```

- Un volume in docker compose (yml):

```
version: "3.9"
```

```
services:
```

```
frontend:
```

```
image: node:14
```

```
volumes:
```

```
- myapp:/home/node/app
```

```
volumes:
```

```
myapp:
```

```
external: true
```

Volume che verrà creato

Volume esistente

DOCKER BIND VOLUME IN COMPOSE

```
version: "3.2"
```

```
services:
```

```
web:
```

```
image: httpd:latest
```

```
volumes:
```

```
- type: bind
```

```
source: $HOST/location
```

```
target: /container/location
```

```
- type: volume
```

```
source: mydata
```

```
target: /container/location
```

```
volumes:
```

```
mydata:
```

DOCKER COMPOSE

Sezioni all'interno del file:

version '3':

Versione di docker compose

services:

I containers dichiarati

build:

In questa sezione possiamo specificare un Dockerfile, e il . rappresenta la posizione del file **docker-compose.yml**.

ports:

Serve per mappare le porte del container verso l'esterno.

volumes:

Serve per montare i dischi.

links:

Serve per collegare i container.

image:

Se non usiamo un Dockerfile dobbiamo specificare un'immagine.

environment:

Variabili di ambiente.

DOCKER COMPOSE

- Comandi in docker compose:

docker-compose build : costruisce un'immagine

docker-compose images : lista delle immagini

docker-compose stop: stop di tutti i containers

docker-compose start: start di tutti i containers

docker-compose run: creazione dei container a partire dalle immagini

docker-compose up: **docker-compose build** + **docker-compose run**

docker-compose ps : lista di tutti i containers

docker-compose down : stop dei containers e rimozione

INSTALLAZIONE DOCKER COMPOSE LINUX UBUNTU 18.04

1. curl -L

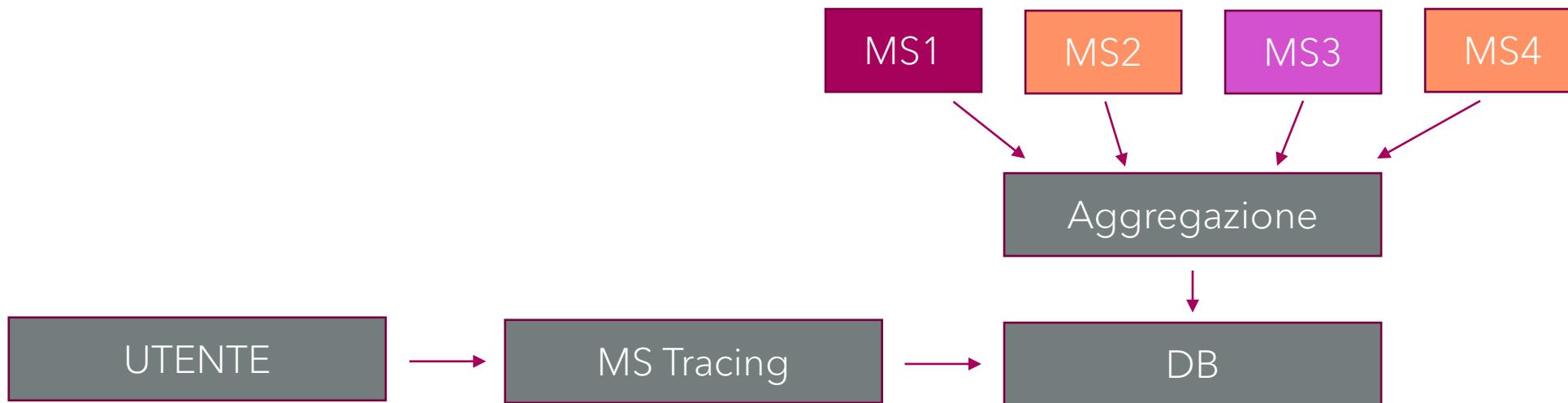
```
"https://github.com/docker/compose/releases/download/1.29.2/docker-compose-$(uname -s)-$(uname -m)" -o /usr/local/bin/docker-compose
```

2. chmod +x /usr/local/bin/docker-compose

3. docker-compose --version

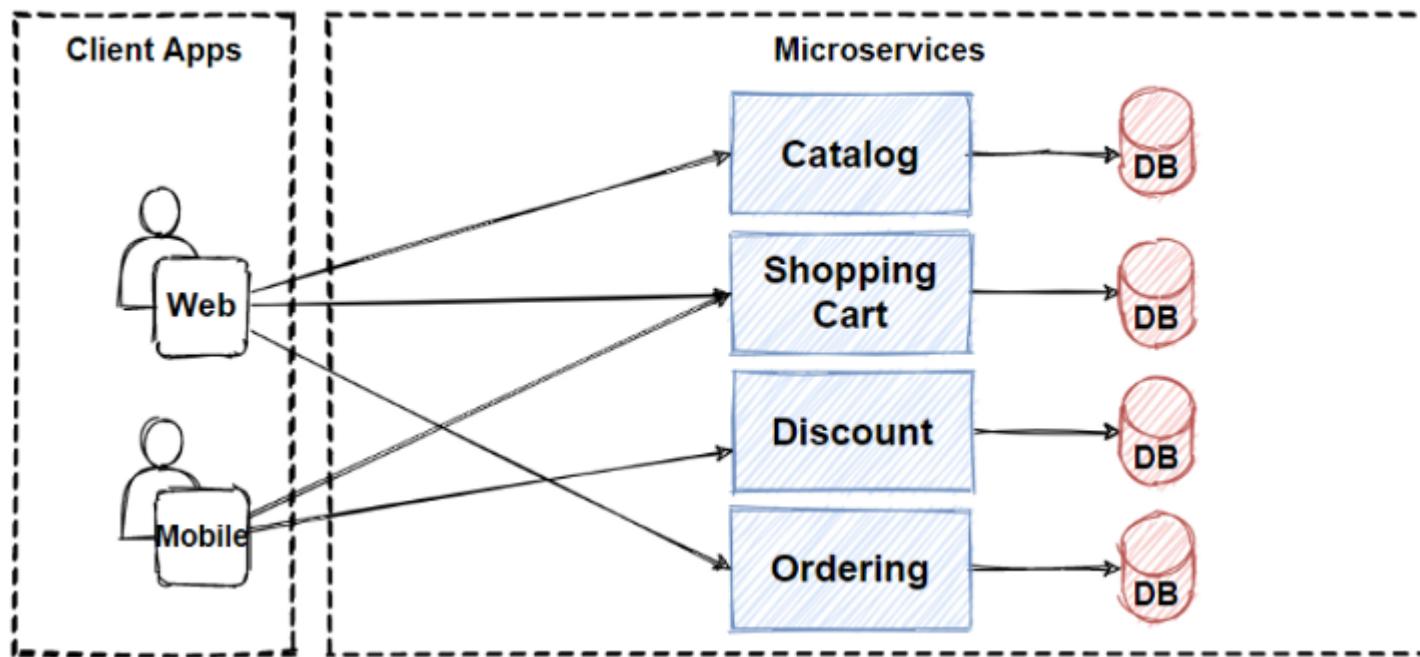
MICROSERVIZI: LOGGING E TRACING

- **Log correlation:** Relazionare i log che coinvolgono l'esecuzione di una funzionalità di business su più microservizi. Utilizzo di un ID di correlazione.
- **Log aggregation:** Inserimento dei log all'interno di un database comune.
- **Tracing:** Monitorare i singoli microservizi.

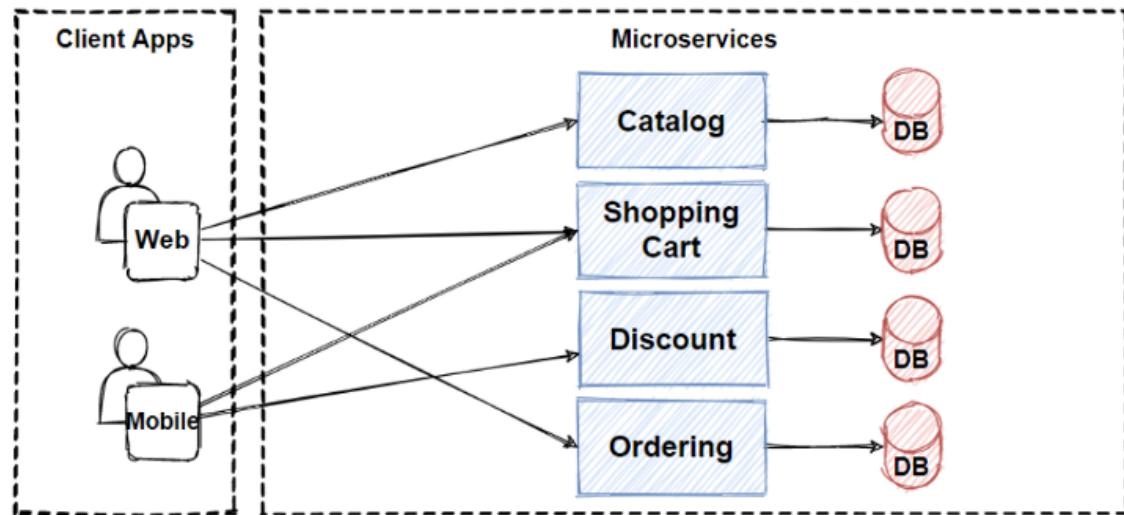


MICROSERVIZI: DESIGN PATTERNS E CASO DI STUDIO

Esempio di applicazione:



MICROSERVIZI: DESIGN PATTERNS E CASO DI STUDIO



La comunicazione dell'interfaccia utente e dei microservizi è diretta e sembra difficile gestire.

I microservizi sono distribuiti e comunicano tra loro tramite la comunicazione tra servizi a livello di rete.

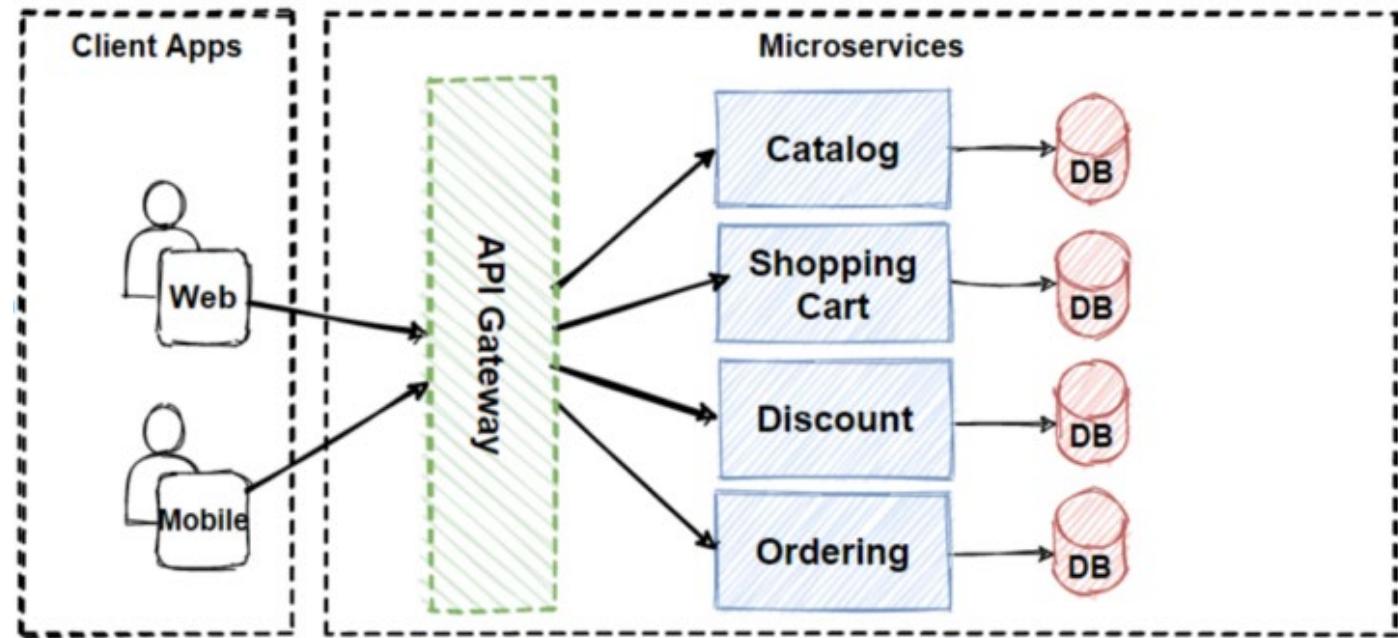
Ogni microservizio ha la propria istanza e processo.

Pertanto, i servizi devono interagire utilizzando protocolli di comunicazione come HTTP, gRPC o il protocollo AMQP dei broker di messaggi.

API GATEWAY

Gestirà le richieste del client e indirizzerà i microservizi interni, aggregherà anche più microservizi interni in un'unica richiesta client e gestirà problemi trasversali come autenticazione e autorizzazione.

Microservices Architecture - Api Gw



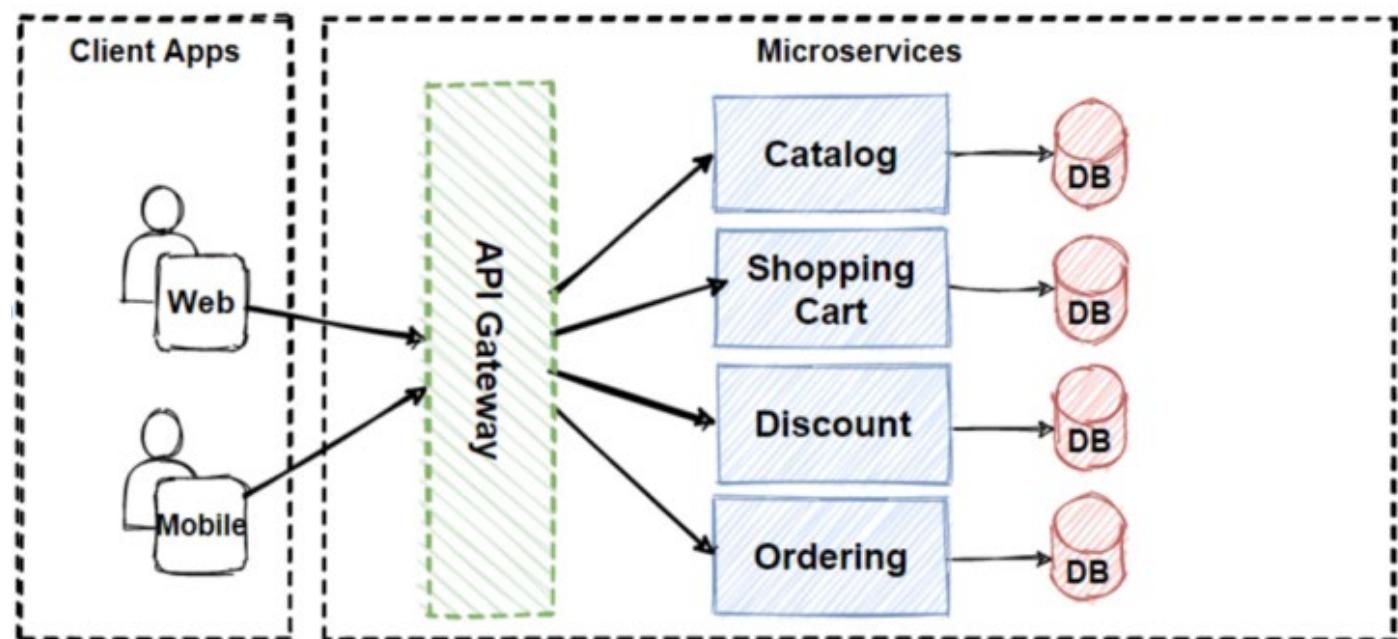
API GATEWAY: ATTENZIONE

Ci sono diverse applicazioni client che si connettono a un singolo gateway API.

Dovremmo fare attenzione a questa situazione, perché se usiamo solo un singolo gateway API, è possibile avere il rischio di un singolo punto di errore.

Se le applicazioni client aumentano o aggiungono più logica alla complessità aziendale il pattern diventa un anti-pattern.

Microservices Architecture - Api Gw



BACKEND PER MODELLO FRONTEND BFF

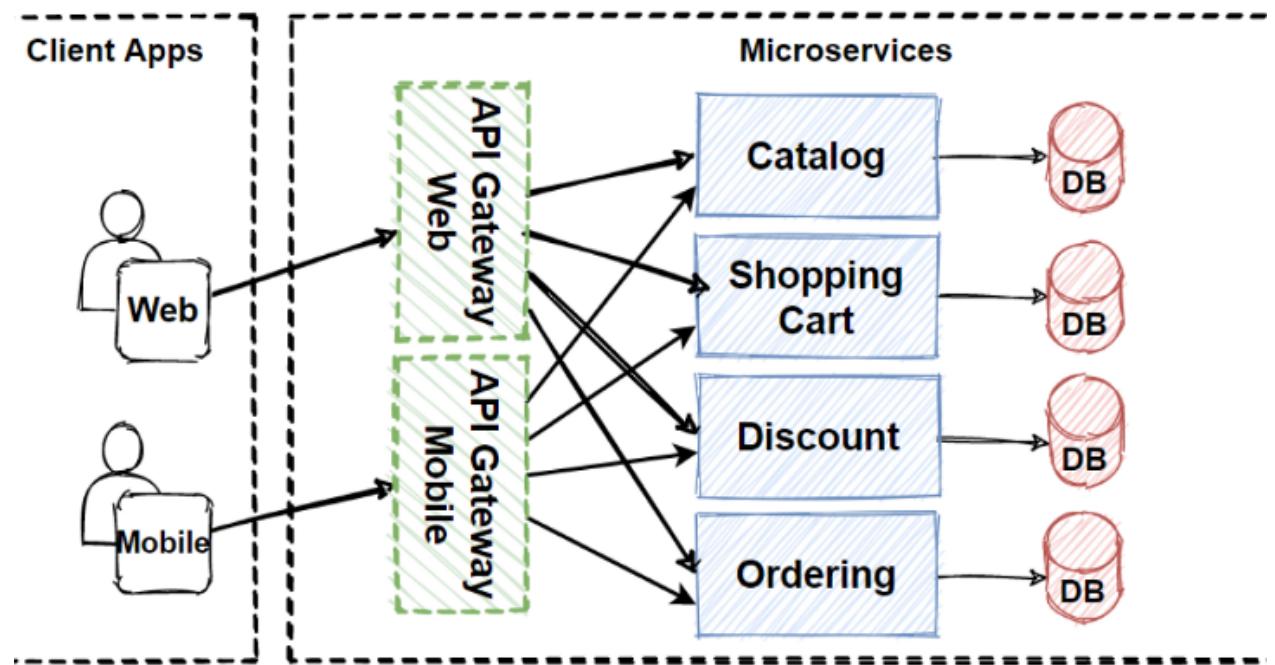
Un gateway API singolo e complesso può essere rischioso e diventare un collo di bottiglia nell'architettura.

I sistemi più grandi spesso espongono più gateway API raggruppando il tipo di client come funzionalità mobile, web e desktop.

Il modello BFF è utile quando si desidera evitare di personalizzare un singolo backend per più interfacce.

Quindi dovremmo creare diversi gateway API secondo le interfacce utente.

Microservices Architecture - Api Gw - BFF



CONSIDERAZIONI

Cosa succede se le richieste del client devono visitare più di un microservizio interno?

Come possiamo gestire le comunicazioni interne dei microservizi?

INTERAZIONI TRA MICROSERVIZI

Quando si progettano applicazioni di microservizi, è necessario prestare attenzione al modo in cui i microservizi interni di backend comunicano tra loro.

La migliore pratica consiste nel ridurre il più possibile la comunicazione tra i servizi.

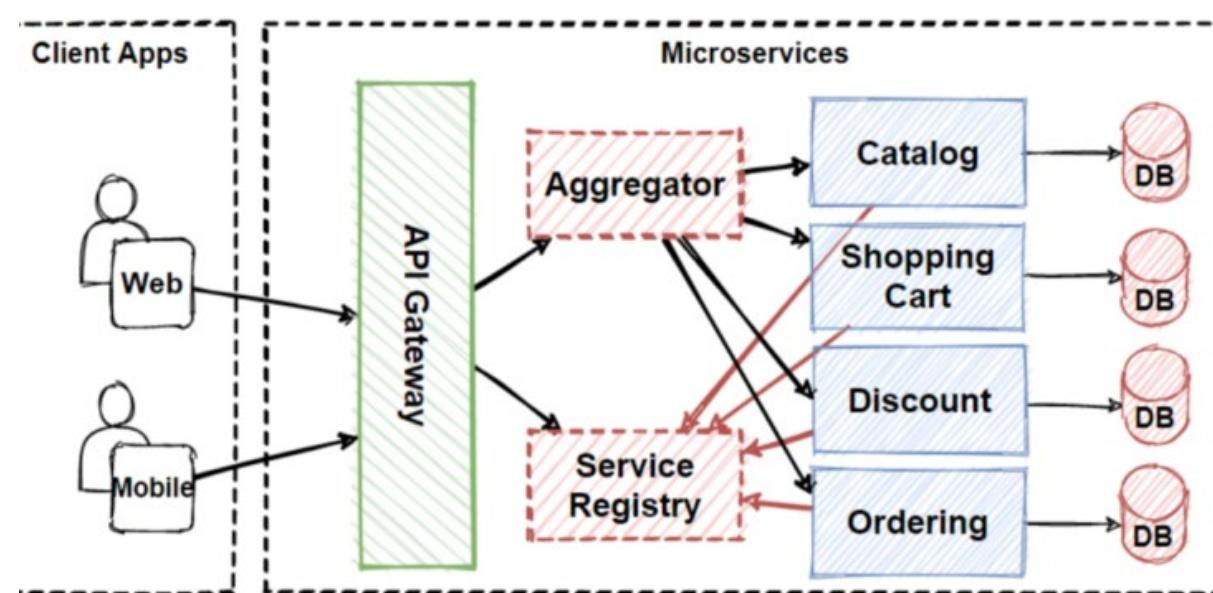
Tuttavia, in alcuni casi, non possiamo ridurre queste comunicazioni interne a causa delle esigenze del cliente o dell'operazione richiesta per visitare diversi servizi interni.

MODELLO DI AGGREGAZIONE DEI SERVIZI

Al fine di ridurre al minimo le comunicazioni da servizio a servizio, possiamo applicare il modello di aggregazione del servizio.

Si riduce la latenza e si gestisce al meglio la situazione di fallimento.

Microservices Architecture - Service Aggregator / Registry Patterns

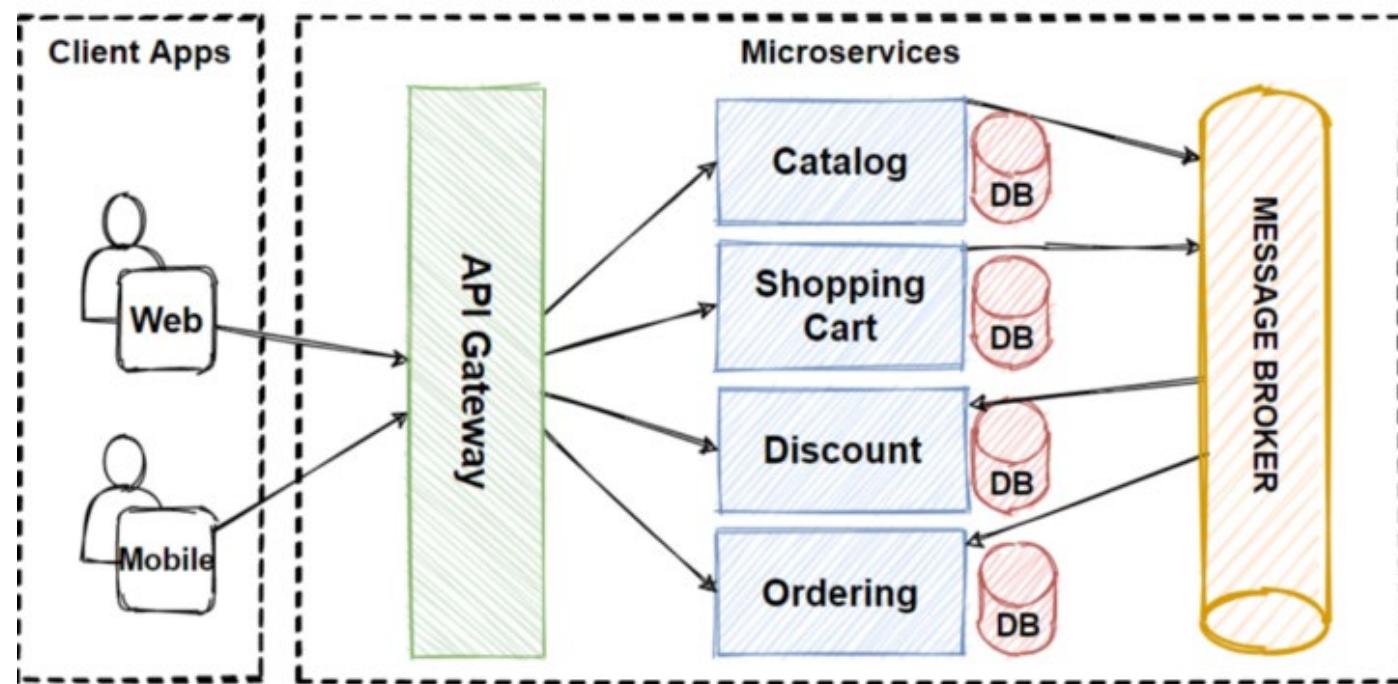


MODELLO DI AGGREGAZIONE DEI SERVIZI

La comunicazione sincrona è buona se avviene solo tra pochi microservizi.

Ma quando si tratta di diversi microservizi che devono interagire tra loro e attendere alcune lunghe operazioni fino al termine, allora dovremmo usare la comunicazione asincrona.

Microservices Architecture - Message Broker



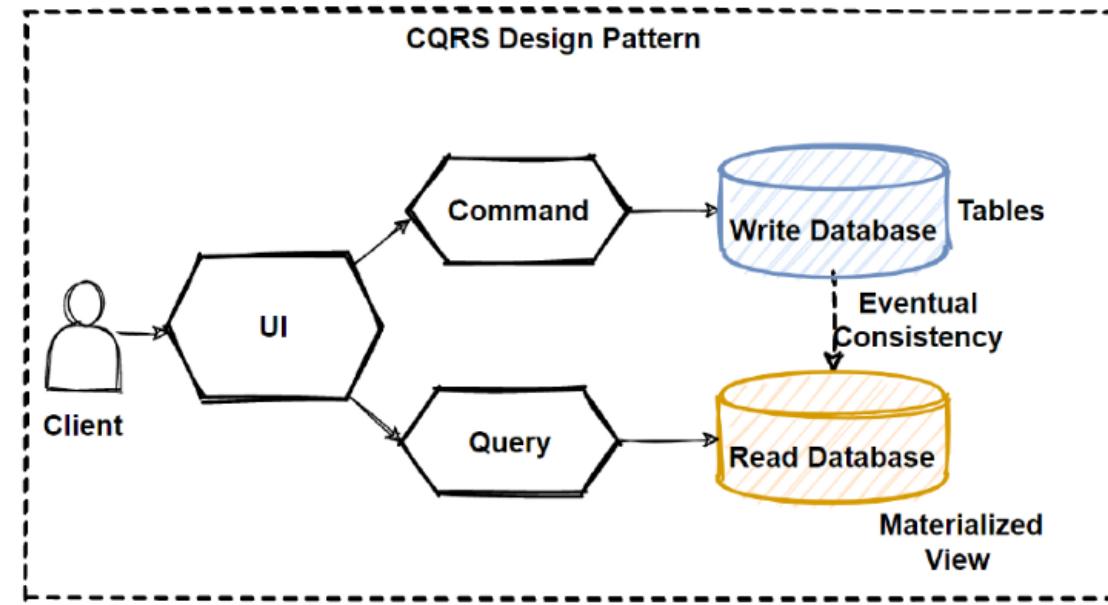
MICROSERVIZI E TRANSAZIONI

- Nelle architetture monolitiche, l'interrogazione dei dati su più tabelle è lo svolgimento delle transazioni è semplice. Eventuali modifiche ai dati vengono aggiornate insieme o vengono ripristinate tutte. I database relazionali con coerenza rigorosa hanno la garanzia delle transazioni ACID.
- Ma nelle architetture di microservizi, quando usiamo la "**persistenza poliglotta**", il che significa che ogni microservizio ha database diversi, database relazionali e no-sql, dovremmo impostare una strategia per gestire questi dati durante l'esecuzione delle interazioni dell'utente.

CQRS + EVENT SOURCING: INTERROGAZIONE SUI DATI

CQRS è uno dei modelli importanti durante l'esecuzione di query tra microservizi. Possiamo utilizzare il modello di progettazione CQRS per evitare query complesse ed eliminare join inefficienti.

CQRS sta per Command and Query Responsibility Segregation. Fondamentalmente questo modello separa le operazioni di lettura e aggiornamento per un database.



Per isolare comandi e query, è consigliabile separare fisicamente il database di lettura e scrittura con 2 database. In questo modo, se la nostra applicazione è ad alta intensità di lettura, il che significa leggere più che scrivere, possiamo definire uno schema di dati personalizzato da ottimizzare per le query.

CQRS + EVENT SOURCING: INTERROGAZIONE SUI DATI

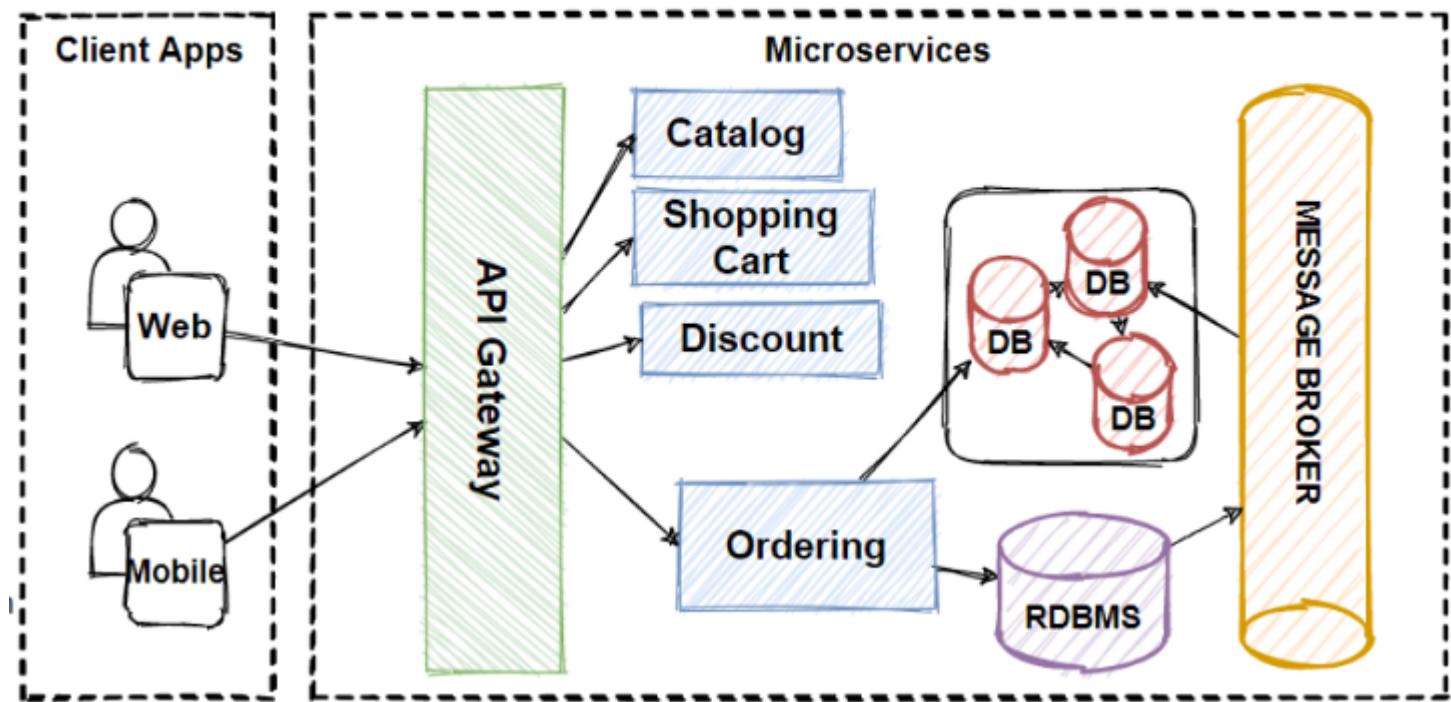
Con l'applicazione del modello Event Sourcing, si passa alle operazioni di salvataggio dei dati nel database. Invece di salvare l'ultimo stato dei dati nel database, il modello Event Sourcing offre il salvataggio di tutti gli eventi nel database con un ordine sequenziale di eventi di dati. Questo database di eventi si chiama archivio eventi.

Invece di aggiornare lo stato di un record di dati, aggiunge ogni modifica a un elenco sequenziale di eventi.

Quindi l'Event Store diventa la fonte di verità per i dati.

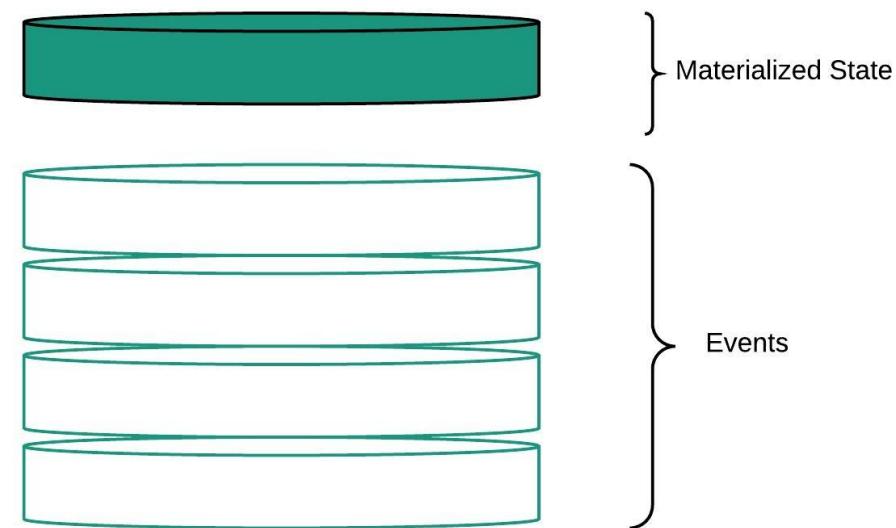
Successivamente, questi archivi eventi vengono convertiti in database di lettura seguendo il modello di viste materializzate. Questa operazione di conversione può essere gestita in base al modello di pubblicazione/sottoscrizione con l'evento di pubblicazione con i sistemi di broker di messaggi.

Microservices Architecture - CQRS, Event Sourcing, Eventual Consistency



EVENT SOURCING

- Event Sourcing è una metodologia per memorizzare lo stato di una applicazione attraverso lo storico degli eventi. Lo stato corrente è ricostruito attraverso la storia degli eventi, dove ogni evento rappresenta qualcosa che è accaduto nella nostra applicazione. Lo stato corrente è chiamato *Materialized state*.



EVENT SOURCING: ESEMPIO

- In un tradizionale Storage system abbiamo soltanto un ordine relativo ad una pizza e una coca cola. In Event Sourcing, vediamo che un utente ha selezionato prima una pizza, poi una coca cola, poi un gelato ed infine ha rimosso il gelato. Con l'Event Sourcing possiamo anche ragionare sul perchè il gelato è stato rimosso...

Online order	Traditional Storage	Event Sourcing
Select pizza	Order 123: Selected pizza	T1: Selected pizza
Select cola	Order 123: Selected pizza, cola	T1: Selected pizza T2: Selected cola
Select ice cream	Order 123: Selected pizza, cola, ice cream	T1: Selected pizza T2: Selected cola T3: Selected ice cream
Deselect ice cream	Order 123: Selected pizza, cola	T1: Selected pizza T2: Selected cola T3: Selected ice cream T4: Deselected ice cream
Confirm order	Order 123: Ordered pizza, cola	T1: Selected pizza T2: Selected cola T3: Selected ice cream T4: Deselected ice cream T5: Confirmed order

EVENT STORE

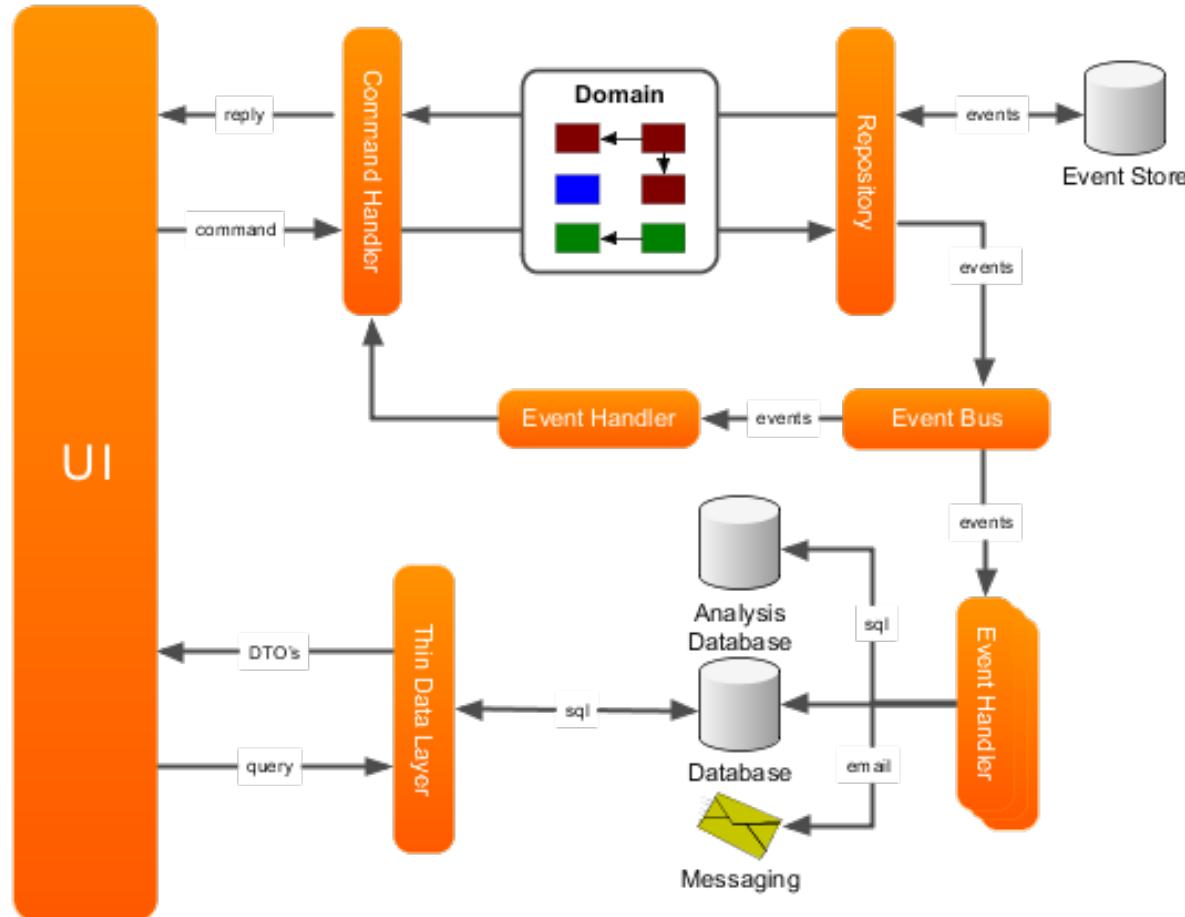
- L'Event Sourcing richiede un Event Store per memorizzare gli eventi. Gli eventi non possono essere modificati. Attraverso gli eventi costruiamo il materialized state.

Axon è un framework che consente di implementare l'Event Sourcing.

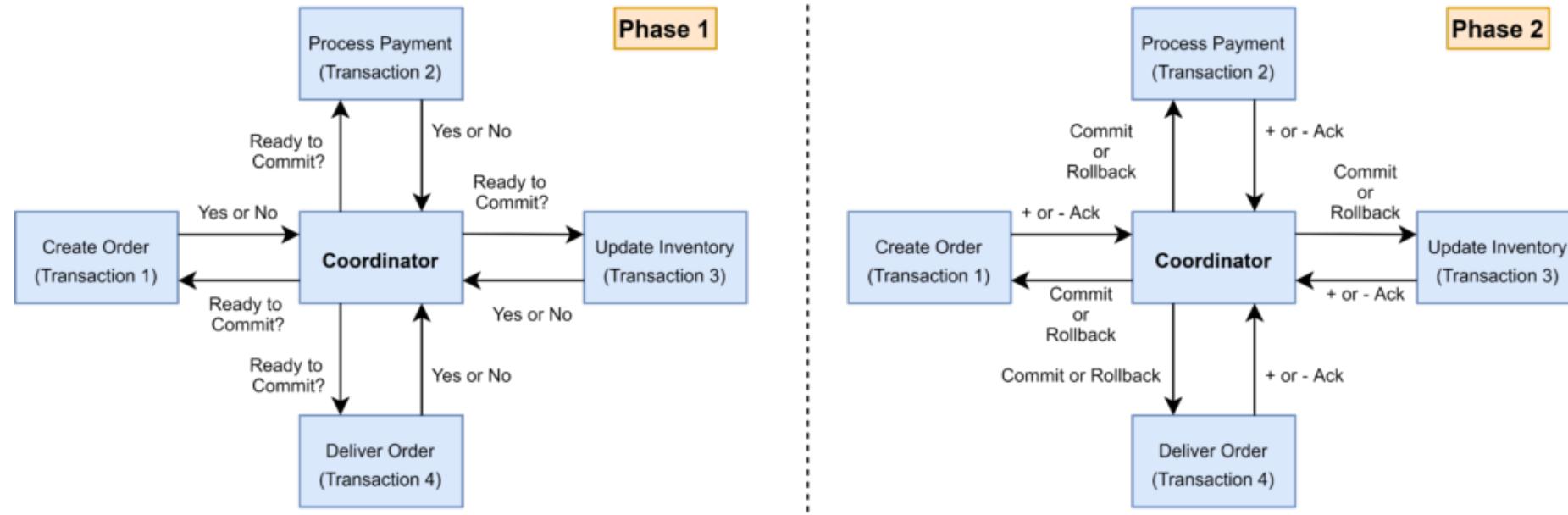
Dal sito:

- *Axon Server* is the default choice within Axon and it offers an *enterprise grade purpose-built event store* which is highly optimized for storing/retrieving events. The Server is available as a Standard Edition or an Enterprise Edition.
- Alternatively, the Axon Framework provides support for an RDBMS or a NoSQL database as an Event Store.

FUNZIONAMENTO CQRS



TRANSAZIONI DISTRIBUITE



1. Prepare Phase: Il coordinatore chiede ai partecipanti se sono pronti al commit. I partecipanti dicono sì o no.

2. Commit Phase: Se tutti rispondono sì, allora il processo di sincronizzazione chiede di effettuare il commit. Se almeno uno risponde negativamente allora viene chiesto a tutti di effettuare il rollback.

PROBLEMA CON IL COMMIT A 2 FASI

Il coordinatore rappresenta un singolo punto di fallimento.

Tutti i servizi devono attendere il più lento.

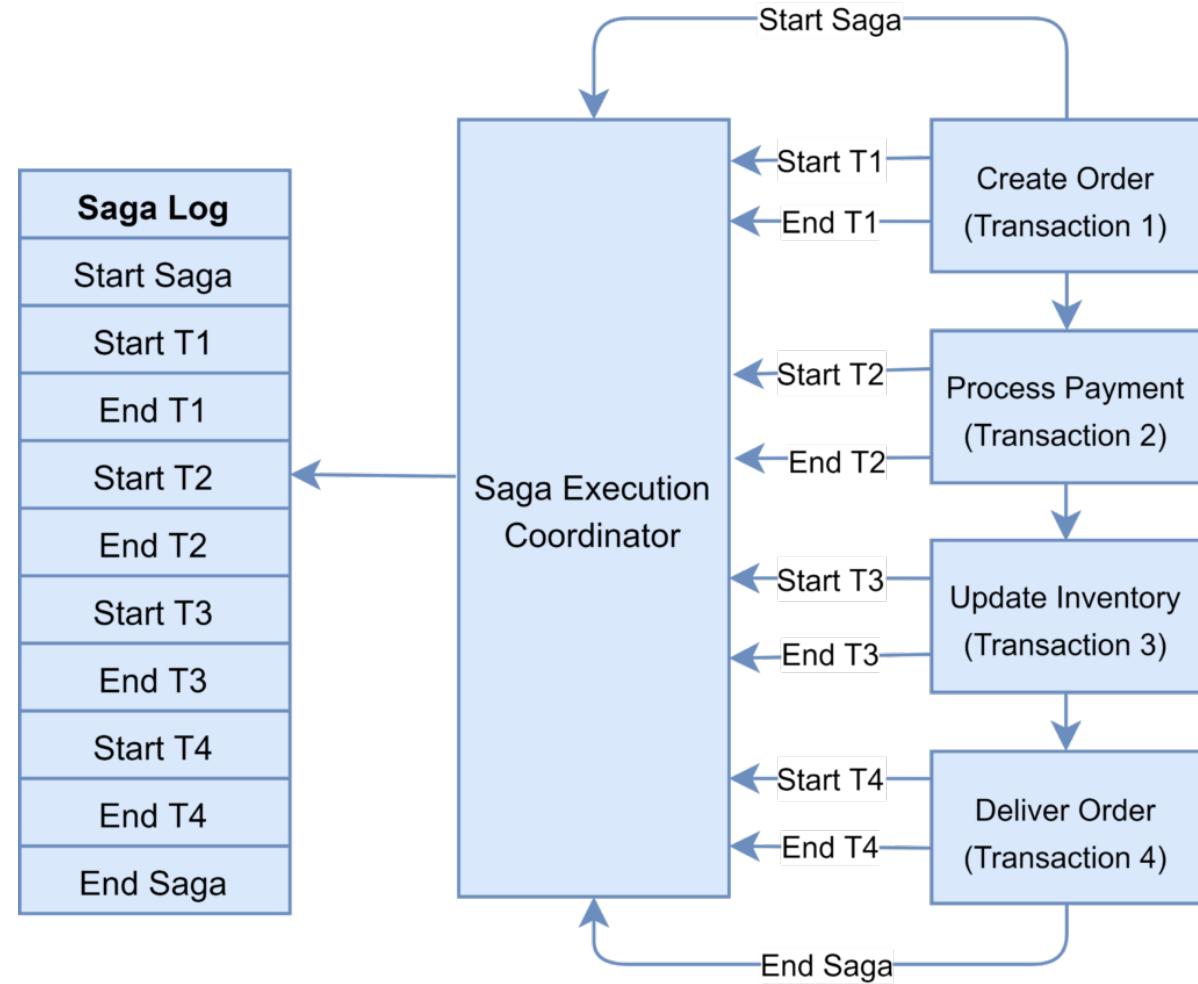
Il protocollo a due fasi è lento, può portare a problemi di scalabilità e di prestazioni in architetture orientate ai microservizi.

Non supportato per database NO SQL.

PATTERN SAGA

Il Saga architecture pattern fornisce una gestione delle transazioni come sequenza di transazioni locali: la saga.

Il pattern Saga garantisce la completa esecuzione di tutte le transazioni oppure l'annullamento completo.



SAGA CHOREOGRAPHY PATTERN

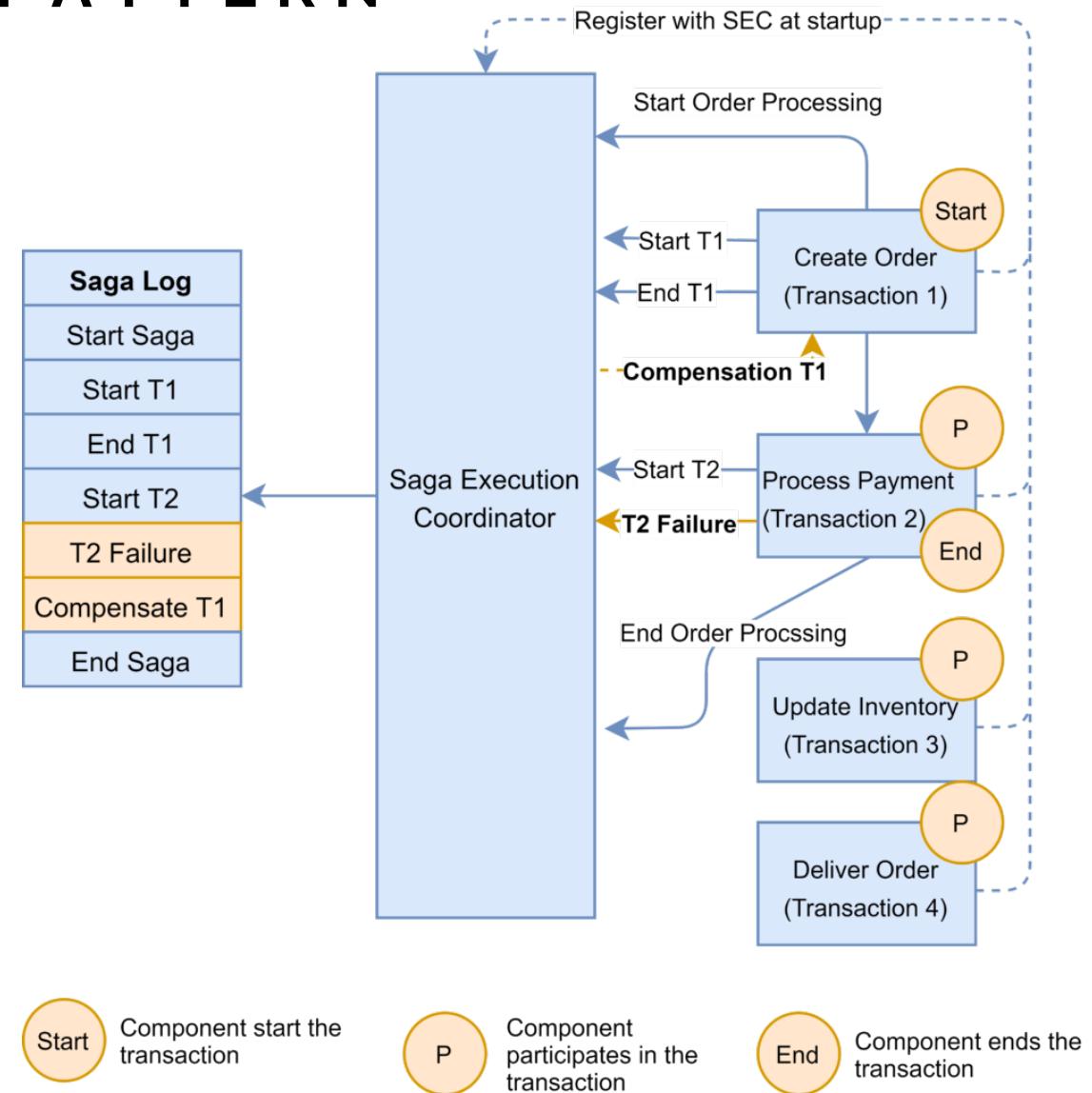
In questo esempio un microservice fallisce, ed il SEC richiama il **compensating transaction**.

Se la chiamata fallisce è responsabilità del SEC ritentare.

Frameworks:

[Axon Saga](#): Utilizzato con Spring Boot.

[Eventuate Tram Saga](#): Utilizzato con Spring Boot.



Component starts the transaction

Component participates in the transaction

Component ends the transaction

COMPENSATING TRANSACTION

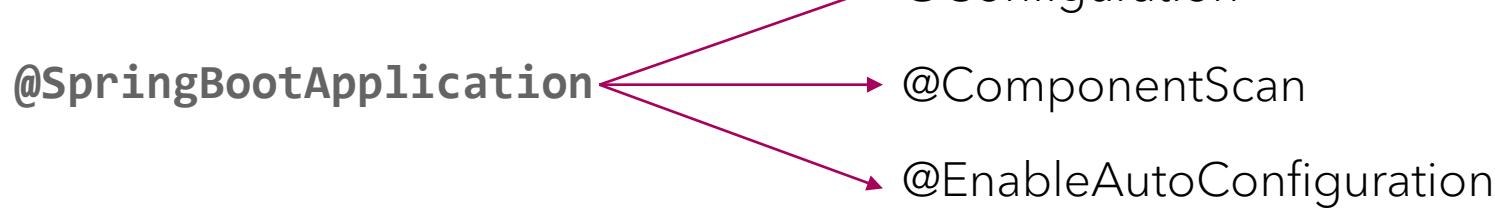
- I passaggi di una transazione di compensazione devono annullare gli effetti dei passaggi nell'operazione originale. Una transazione di compensazione potrebbe non essere in grado di sostituire semplicemente lo stato corrente con lo stato di sistema all'inizio dell'operazione poiché questo approccio potrebbe sovrascrivere le modifiche apportate da altre istanze simultanee di un'applicazione. Deve invece essere un processo intelligente che prenda in considerazione tutte le azioni eseguite.
- Un approccio comune prevede l'uso di un flusso di lavoro per implementare un'operazione coerente che richiede la compensazione. Via via che l'operazione originale procede, il sistema registra informazioni su ogni passaggio e su come può essere annullato il lavoro da esso eseguito. Se l'operazione non riesce in qualsiasi punto, il flusso di lavoro torna indietro nei passaggi completati ed esegue il lavoro che inverte ogni passaggio. Si noti che una transazione di compensazione potrebbe non dover annullare il lavoro seguendo l'esatto ordine inverso dell'operazione originale e alcuni dei passaggi dell'annullamento potrebbero essere eseguiti in parallelo.

SPRING BOOT: INTRODUZIONE

- Progetto che rende più semplice lo sviluppo di applicazioni Spring
- Configurazione automatica (dove possibile)
- Starter dependencies: librerie configurate per l'uso immediato
(<https://www.geeksforgeeks.org/spring-boot-starters/>)

SPRING BOOT: CONFIGURAZIONE MINIMA

```
@SpringBootApplication  
  
public class Application extends SpringBootServletInitializer {  
    public static void main(String[] args) {  
        SpringApplication.run(applicationClass, args);  
    }  
    @Override  
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {  
        return application.sources(applicationClass);  
    }  
    private static Class<Application> applicationClass = Application.class;  
}
```



SPRING BOOT ANNOTATIONS

@Configuration: Annotazione a livello di classe. Una classe nella quale dichiariamo Beans.

@ComponentScan: Utilizzata per la scansione di package che contentono componenti. Utilizzata con l'annotation **@Configuration**.

@Bean: Annotazione a livello di metodo. Dice al metodo di produrre un bean gestito dallo Spring Container.

@Component: Annotazione a livello di classe. Una classe Java annotata con **@Component** è un componente generico. Il framework Spring ne crea un'istanza e la inserisce come **Spring Bean** nell'**application context**.

@Controller: Annotazione a livello di classe. Specializzazione di **@Component**. Marca una classe come gestore di richieste web. Per default ritorna una stringa di navigazione. Utilizzata frequentemente con **@RequestMapping**.

@Service: Annotazione a livello di classe. Dice a Spring che la classe contiene **business logic**.

@Repository: Annotazione a livello di classe. Denota un repository ovvero un **DAO** (Data Access Object).

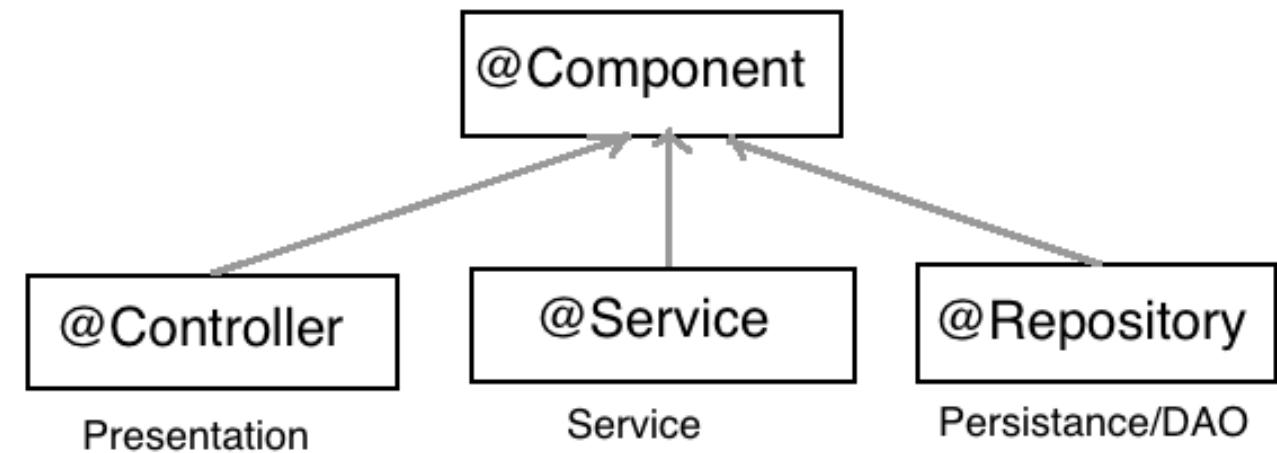
@EnableAutoConfiguration: Auto configura i bean presenti nel classpath.

@SpringBootApplication: Combinazione di tre annotazioni (**@EnableAutoConfiguration**, **@ComponentScan**, and **@Configuration**)

@RestController: Combinazione di **@Controller** e **@ResponseBody**. Elimina la necessità di utilizzare **@ResponseBody**.

SPRING BOOT COMPONENTS

1. **@Component** il componente più generico.
2. **@Repository** specializzata per il persistence layer.
3. **@Service** specializzata per il service layer.
4. **@Controller/RestController** specializzata per il presentation layer (spring-MVC).
5. **@Bean** è utilizzato per dichiarare un singolo bean. Non viene creato da Spring in automatico. Utilizzato nelle classi di configurazione

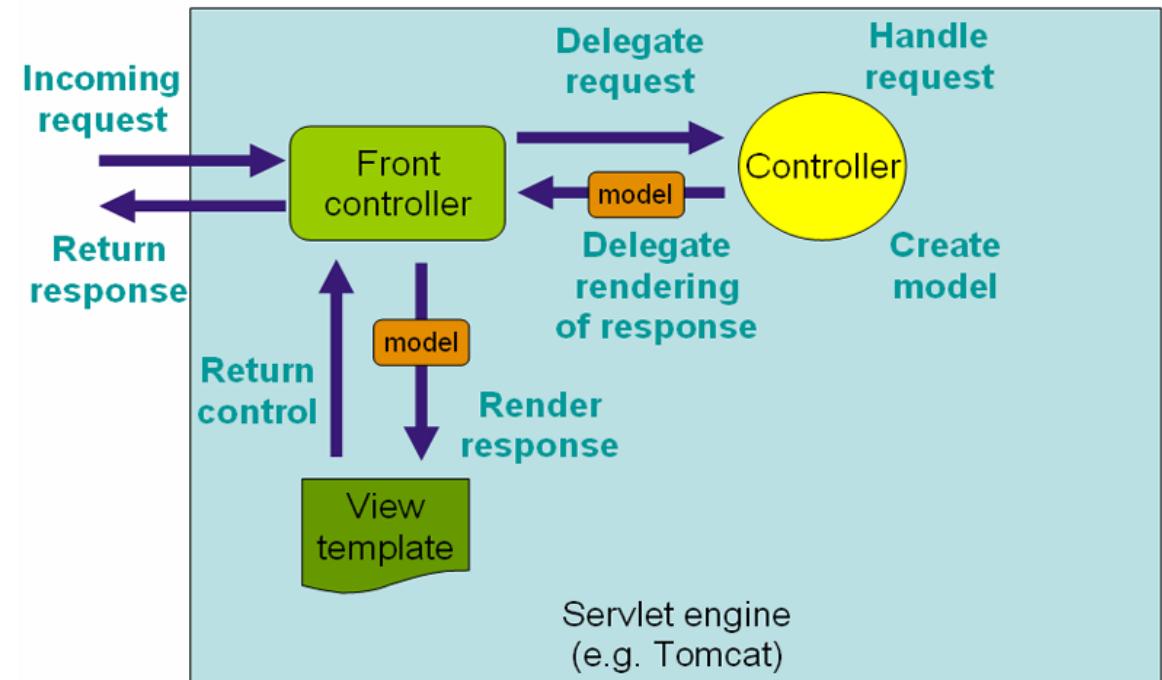


SPRING INITIALIZERS

- Possiamo generare un progetto e scaricarlo come ZIP file. La procedura è la seguente:
 - 1.Visitare il sito <https://start.spring.io>. Qui abbiamo un servizio che consente la selezione di tutte le dipendenze di cui abbiamo bisogno.
 - 2.Possiamo scegliere Gradle or Maven come strumento per la gestione del Progetto Java.
 - 3.Click **Dependencies** e selezionare quelle desiderate.
 - 4.Click **Generate**.
 - 5.Download ZIP file.

SPRING MVC

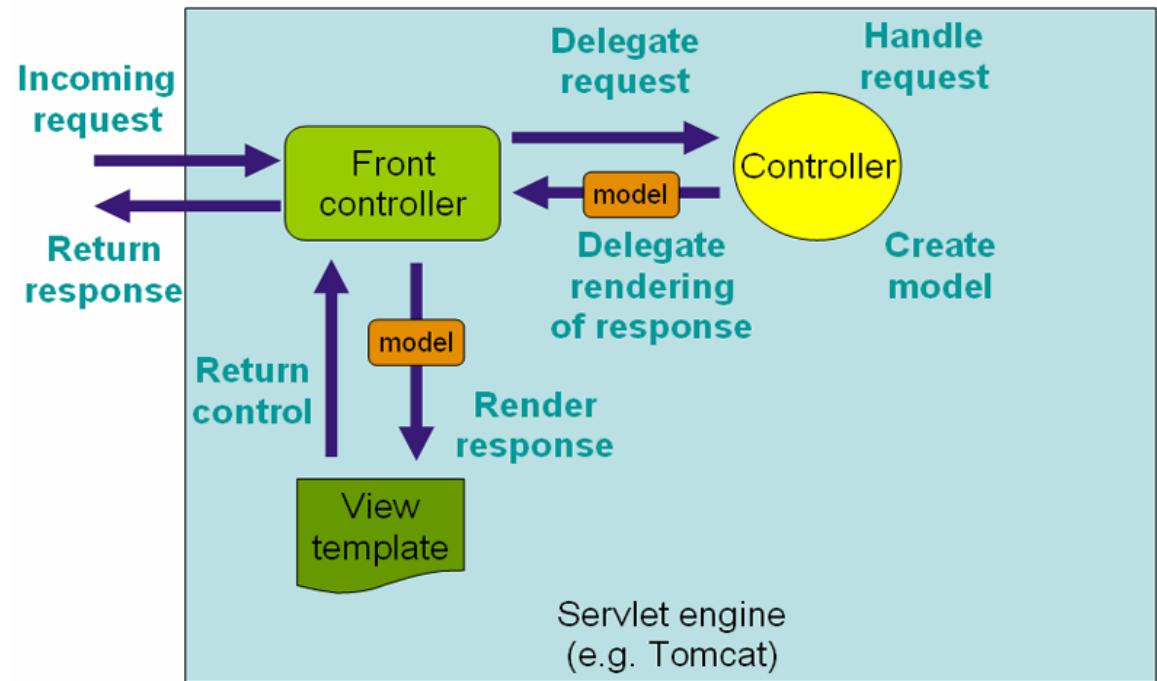
- Spring MVC è il framework Spring per lo sviluppo di applicazione web Java. Consente la creazione di web sites o RESTful services (JSON/XML) ed è chiaramente integrato con l'ecosistema Spring.
- Quando scriviamo web applications in Java, con o senza Spring (MVC/Boot), parliamo di applicazioni che ritornano:
 - **HTML** → Web app che rappresenta la view in HTML.
 - **JSON/XML** → RESTful services, che producono JSON or XML.



SPRING MVC

Il pattern MVC si basa su tre componenti fondamentali:

- **Model**: rappresenta il *dominio dell'applicazione*, è generato dalle logiche di *business* e di accesso ai dati;
- **View**: ha il compito di *presentare i dati* all'utente e consente a quest'ultimo l'interazione con l'applicazione;
- **Controller**: gestisce l'*interazione con l'utente* tramite la *View*, intercetta richieste e fornisce risposte in base allo stato del *Model*.
- L'obiettivo di MVC è quello di separare in modo netto la struttura dei componenti interni dell'applicazione dal modo in cui essi vengono presentati all'utente; di conseguenza, anche il modo in cui l'utente interagisce con questi oggetti è svincolato dalla loro rappresentazione.
- Per essere precisi Spring MVC implementa solamente due componenti del pattern: *View* e *Controller*. L'implementazione del *Model* è infatti delegata alle classi di servizio (*Service*) e ai *Repository* per l'accesso ai dati.



SPRING MVC: DISPATCHER SERVLET

La DispatcherServlet gestisce ogni HTTP request (front controller). Immaginiamo un flusso nel quale inviamo una richiesta HTTP verso un sito nel quale è in esecuzione un'applicazione Spring Boot MVC. La DispatcherServlet svolgerà le seguenti operazioni:

- Dovrà analizzare la tipologia di richiesta ed i parametri inviati.
- Dovrà effettuare eventuali conversioni sui dati (request parameters/body) per ottenere oggetti Java da inviare ai @Controller o @RestController della classe presente sull'applicativo.
- Il metodo del @Controller svolgerà una serie di operazioni.
- Convertire l'output del @Controller nel formato corretto HTML/JSON/XML.

SPRING BOOT @CONTROLLER

Il Controller ha il compito di portare alla visualizzazione dello stato del Model sulla View di destinazione, e di intercettare gli input ricevuti dalla vista e riportarli nel Model. Definire un controller in Spring Boot è abbastanza semplice, si utilizza l'annotation **@Controller**:

```
@Controller  
public class HelloController {  
    @RequestMapping("/hello")  
    public String hello(){  
        return "hello";  
    }  
}
```

L'annotazione **@RequestMapping** viene utilizzata per indicare i metodi addetti alla gestione delle richieste HTTP. Nel nostro caso l'annotazione registra nel contesto dell'applicazione un handler per le chiamate dirette verso (/hello) dell'applicazione.

La stringa "hello" identifica la pagina (View) restituita/visualizzata ogni volta che una chiamata HTTP viene intercettata dal nostro handler method.

L'annotazione **@Controller** è una specializzazione di **@Component**; un Controller in Spring MVC rappresenta infatti un tipo particolare di componente, la cui creazione è gestita dallo Spring IoC container.

@REQUESTMAPPING

- Lo scopo principale dell'annotazione è quello di definire un'associazione tra un handler e un gruppo di url; l'elemento path dell'annotazione (alias dell'elemento implicito **value**) serve proprio a specificare quali url saranno associati al metodo.
- Il mapping definito nell'esempio non fa riferimento ad alcun metodo HTTP, difatti esso supporterà richieste di ogni tipo. Se però volessimo esplicitare quali metodi HTTP devono essere gestiti dall'handler, basterà valorizzare l'attributo **method**:

```
@RequestMapping(value="/hello", method=RequestMethod.GET)
```

- Esistono inoltre alcune specializzazioni di @RequestMapping (introdotte nella versione 4.3 di Spring).
- Vincoli sulla presenza di parametri possono essere imposti con l'attributo **params** inserendo stringhe del tipo **param_name=param_value**:

```
@RequestMapping(value="/hello", method=RequestMethod.GET, params="id=1")
```

- Vincoli sugli header possono essere specificati in modo simile ai vincoli:

```
@RequestMapping(value="/hello", method=RequestMethod.GET, header="id=1")
```

@REQUESTMAPPING

- **Consumes:** Consente di specificare quali sono i *media type* “consumabili”, ovvero quali formati di dato sono supportati dalla richiesta. Si possono indicare più *media type* e *wildcard*, oppure utilizzare operatori di negazione per escludere eventuali tipi non ammessi, ad esempio:

consumes = {"text/plain", "application/*"}

oppure nella forma: **consumes= MediaType.APPLICATION_JSON**

- **Produces:** permette di *mappare* solamente le richieste che “producono” determinati *media type*.

produces= MediaType.APPLICATION_JSON

L'annotazione **@RequestMapping** può essere applicata anche a livello di classe.

L'INTERFACCIA MODEL

- Scopriamo quali meccanismi utilizzare per passare attributi dal Controller alla View e come sfruttare l'integrazione tra Spring MVC e [Thymeleaf](#) per costruire pagine con contenuto dinamico.

```
@RequestMapping(value = "/users/findUsers/{start}/{max}", method = RequestMethod.GET)
public String findUsers(Authentication auth, Model model, @PathVariable Integer start,
@PathVariable Integer max) {
    BusinessUserPageDTO page = userService.findUsers(start, max);
    model.addAttribute("dto", page);
    return "user.html";
}
```

L'interfaccia Model rappresenta un contenitore in cui inserire tutti gli oggetti "condivisi" da Controller e View. Si tratta, a tutti gli effetti, di un wrapper per un oggetto di tipo Map.

VIEW: THYMELEAF

- Spring Boot supporta l'integrazione con [diverse tecnologie e librerie](#) dedicate alla costruzione delle View (comprese le classiche JSP). Al momento, la scelta raccomandata è sicuramente Thymeleaf.
- Thymeleaf è un template engine per lo sviluppo di applicazioni web, che consente di creare in modo molto naturale pagine dinamiche, strutturando il codice HTML in elementi riutilizzabili.
- Includendo lo starter package `spring-boot-starter-thymeleaf` tra le dipendenze del progetto, verrà configurato di default il caricamento dei template HTML dalla directory `/resources/templates`.

PAGINA THYMELEAF: LISTA UTENTI

```
<!DOCTYPE html>
<html lang="en" xmlns:th="http://www.thymeleaf.org">
.....
<a th:href="@{/user/init}" class="btn btn-primary btn-sm mb-3">Create User</a>
<table class="table table-striped table-responsive-md">
<thead>
<tr> <th>Id</th><th>First Name</th><th>Last Name</th><th>Fiscal Code</th></tr>
</thead>
<tbody>
<tr th:each="user: ${dto.page}">
    <td th:text="${user.id}" /> <td th:text="${user.firstName}" /><td th:text="${user.lastName}" />
    <td th:text="${user.fiscalCode}" />
</tr>
</tbody>
</table>
.....
```

THYMELEAF: CREAZIONE UTENTE

```
<!DOCTYPE html>

<html lang="en" xmlns:th="http://www.thymeleaf.org">
    ...
<form th:action="@{/user/create}" th:object="${dto}" method="post">
    <input type="text" th:field="*{firstName}" class="form-control mb-4 col-4"/>
    <input type="text" th:field="*{lastName}" class="form-control mb-4 col-4"/>
    <input type="text" th:field="*{fiscalCode}" class="form-control mb-4 col-4"/>
    <button type="submit" class="btn btn-info col-2">Add</button>
</form>
    ...
```

L'INTERFACCIA MODEL

- Esiste un altro modo di passare oggetti alla View, in Spring MVC. L'annotazione @ModelAttribute permette di "marcare" un parametro come attributo da inserire o recuperare dal Model:

```
@RequestMapping(value="/user/create",
method = RequestMethod.POST,
consumes=MediaType.APPLICATION_FORM_URLENCODED_VALUE)
public String create(Authentication auth, @ModelAttribute("dto") BusinessUserDTO model) {
    .....
}
```

@RESTCONTROLLER

- Qual'e' la differenza tra @Controller e @RestController?

@Controllers di default ritornano HTML attraverso l'utilizzo di una templating library. Si modifica il comportamento aggiungendo l'annotation **@ResponseBody** al metodo, per produrre contenuti differenti. @RestController è effettivamente @Controller + @ResponseBody annotation. Quindi equivale ad aggiungere @Controller con tutti i metodi annotati con @ResponseBody.

@Controller

@ResponseBody

public @interface RestController {...}

IOC: INVERSION OF CONTROL

- Principio dell'ingegneria del software che trasferisce il controllo della creazione di oggetti (object oriented programming) ad un container o framework.
- Vantaggi:
 - Disaccoppiamento dell'esecuzione di un task dalla sua implementazione
 - E' semplice passare da una implementazione ad un'altra
 - Modularità dell'applicazione è capacità di identificare facilmente componenti che utilizzano una comunicazione per **contratto**.
- Realizzabile in diversi modi: Strategy pattern, Service Locator Pattern, Factory Pattern e Dependency Injection.

SPRING IOC CONTAINER

- Nel framework Spring l'interfaccia ApplicationContext rappresenta il container IOC.
- Il container Spring è responsabile per l'istanziazione, gestione, configurazione (beans).
- Il framework è dotato di diverse implementazioni dell'interfaccia ApplicationContext.

A partire dal metodo run, l'application context è costruito ed è eseguita una ricerca delle classi annotate con @Configuration. Tutti i bean in queste classi vengono inizializzati, ed in base allo scope dei bean, vengono inseriti in una parte della JVM denominata IOC container.

Successivamente viene configurata la dispatcher servlet, registrati i controllers e tutto c'e' che necessita di essere configurato.

DEPENDENCY INJECTION

- Implementazione dell'**inversion of control** nel quale le dipendenze sono risolte in automatico da un container. Attraverso XML o annotations siamo in grado di specificare cosa dovrà essere «iniettato» in un particolare riferimento di classe o interfaccia.
- Stiamo parlando di una dipendenza tra un componente verso un altro componente.
- La dependency injection in Spring può essere realizzata attraverso costruttori, metodi set o variabili di istanza.
- **Dependency:** Un oggetto in genere richiede oggetti di altre classi. Chiamiamo questi oggetti **dependencies**.
- **Injection:** Il processo di reperire le **dependencies** di un oggetto.

BEANS

- L'ultima versione del framework Spring definisce sei tipi di scopes:
 - **singleton** (singola istanza per application context)
 - **prototype** (nuova istanza ogni volta che è richiesto il bean)
 - **request** (istanza per richiesta HTTP)
 - **session** (istanza per sessione HTTP)
 - **application** (istanza associate a livello di web application)
 - **websocket**(istanza per sessione socket)

Con l'application scope, il container crea un'istanza per web application runtime. L' application scope è simile al singleton scope. La differenza è che l'*Application scoped* è un singleton per ServletContext mentre il *singleton scoped* is singleton per ApplicationContext. Possiamo avere più application contexts per una singola applicazione.

DEPENDENCY INJECTION (AUTOWIRING): COSTRUTTORE/FIELD

- Il container richiama un costruttore con argomenti. Gli argomenti rappresentano la dipendenza che si desidera iniettare.
- Spring cercherà dei beans che sono compatibili con il tipo dichiarato.
- Con il costruttore siamo sicuri al 100% che la classe non sarà mai istanziata senza le dipendenze.
- Il container IoC farà in modo di recuperare tutte le dipendenze necessarie al costruttore. Nessun **NullPointerException**.
- Con la field-based injection, Spring assegna le dipendenze direttamente ai campi annotate con **@Autowired** annotation.

Esempio nell'IDE...

DEPENDENCY INJECTION: METODO SET

- Nel setter-based injection, le dipendenze vengono fornite attraverso i metodi set. I metodi set devono essere annotati con @Autowired.

Esempio nell'IDE...

DEPENDENCY INJECTION: VANTAGGI NELL'USO DEL COSTRUTTORE

- Constructor injection ci aiuta a capire quante sono le dipendenze della nostra classe. Se il costruttore ha troppi argomenti significa che classe ha troppe responsabilità.
- Constructor injection ci aiuta nella creazione di oggetti immutabili perchè solo attraverso il costruttore possiamo creare oggetti. Non possiamo modificare le dipendenze dopo la creazione.
- Con la setter injection, le dipendenze vengono inniettate dopo la creazione. Quindi abbiamo oggetti mutabili che possono portare a dei problemi.

DEPENDENCY INJECTION: VANTAGGI NEL METODO SET

Con la setter injection, Spring ci consente di specificare dipendenze opzionali:

```
@Autowired(required = false)
```

Non è possibile con la Constructor Injection perchè required=false sarebbe applicato a tutto.

DEPENDENCY INJECTION: SVANTAGGI FIELD INJECTION

- Field injection da evitare perchè:
 1. Non possiamo creare oggetti immutabili (il campo non può essere final).
 2. La classe non può essere istanziata senza reflection.
 3. E' difficile accorgersi delle responsabilità di una classe. Se utilizziamo un costruttore la presenza di troppi argomenti è un segnale di un design non buono.

@CONFIGURATION

- Con questa annotazione impostiamo una classe come classe di configurazione, esporrà quindi dei bean. Possiamo avere più classi di configurazione. Si possono importare queste classi utilizzando:

`@Import({it.backend.conf.ProjectConfig.class})` //dopo l'annotation `@SpringBootApplication`.

La classe è:

```
@Configuration  
public class ProjectConfig {..}
```

@COMPONENT VS @REPOSITORY VS @SERVICE

Nelle applicazioni in generale abbiamo layer distinti: data access, presentation, service per la logica di business.... Per ciascun layer abbiamo diversi bean. Possiamo indicare dove cercare componenti:

`@ComponentScan`

`({"it.backend.web", "it.backend.components", "it.backend.service", "it.backend.repository"})`

`@EnableJpaRepositories({"it.backend.repository"})`

`@EntityScan({"it.backend.entity"})`

`@Service` e `@Repository` sono particolari tipi di componenti (`@Component`). Tecnicamente sono identici a `@Component` ma pensati per altri scopi.

`@Repository`: è in grado di catturare eccezioni specifiche per lo strato di persistenza e rilanciarle come unchecked exceptions proprie di Spring.

`@Service`: indica che un componente esegue logica di business, nient'altro di speciale.

STILE ARCHITETTURALE REST

- REST : Representational State Transfer
- REST non è un framework
- Rest è uno stile di architettura per lo sviluppo di servizi web
- Utilizzo del protocollo HTTP

CARATTERISTICHE REST

- 1) Utilizza le richieste HTTP GET,PUT,POST,DELETE per consentire la gestione delle funzionalità esposte dalle API.
- 2) I dati sono considerati risorse.
- 3) Le risorse sono gestite tipicamente attraverso formati JSON e XML.

VINCOLI REST

Ogni applicazione che segue i seguenti vincoli è un'applicazione che possiamo chiamare REST.

- 1) Separazione responsabilità client-server. In particolare il server non deve intervenire nella logica di rendering e mantenere uno stato client.
- 2) Stateless
- 3) Cacheable
- 4) Layered system

VINCOLI REST

- 5) Uniforme interface: client e server evolvono indipendentemente.
- 6) Risorsa identificabile: HTTP/1.1 get [http://localhost:8080/...](http://localhost:8080/)
- 7) Risorsa rappresentata attraverso un formato: TEXT, HTML, XML, JSON...
- 8) Richieste autodescrittive: devono contenere tutto il necessario affinchè il server sia in grado di generare una risposta.
- 9) HATEOS: Hypermedia as the Engine of Application State. La risposta inviata al client deve essere completa in modo da garantire la navigazione verso la risorsa da parte del client.

PROFILI

- Il processo di sviluppo di un software ha diversi stage: development, testing, e production. Spring Boot **profiles** ci permette di realizzare configurazioni specifiche per l'ambiente nel quale dovrà essere lanciata l'applicazione.
- Un **profile** è un insieme di configurazioni. Spring Boot consente di specificare dei file di properties specifici per profile: **application-{profile}.properties**.
- Per tutti i profili si caricano le proprietà definite nell'application.properties. Le proprietà definite nei profili specifici sovrascrivono quelle dell'application.properties.
- L'annotazione **@Profile** consente di specificare per quale **profile** un componente Spring deve essere attivo. Abbiamo un profilo di default; tutti i bean per i quali non è specificato un profilo appartengono a quello di default.
- Possono essere definiti profili con: Maven settings, JVM system parameters, environment variables, spring.profiles.active property, e SpringApplication methods.

PROFILI: @PROFILE ANNOTATION

- Esempio di uso dell'annotazione `@Profile`. Mappiamo un bean su un particolare profile:

```
@Component @Profile("dev") public class DevDatasourceConfig{}
```

```
@Component @Profile("!dev") public class DevDatasourceConfig{}
```

PROFILI: ATTIVAZIONE

- Web application initializer:

```
@Configuration public class MyWebApplicationInitializer implements  
WebApplicationInitializer {  
  
    @Override  
  
    public void onStartup(ServletContext servletContext) throws  
        ServletException {  
  
        servletContext.setInitParameter("spring.profiles.active", "dev");  
  
    }  
}
```

PROFILI: ATTIVAZIONE

- Utilizzando l'environment:

```
@Autowired
```

```
private
```

```
ConfigurableEnvironment env;
```

```
...
```

```
env.setActiveProfiles("someProfile");
```

PROFILI ATTIVAZIONE

- Context parameter Web App:

```
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/app-config.xml</param-value>
</context-param>
<context-param>
    <param-name>spring.profiles.active</param-name>
    <param-value>dev</param-value>
</context-param>
```

PROFILI ATTIVAZIONE

- Parametro JVM:

```
-Dspring.profiles.active=dev
```

- Variabile di ambiente:

```
export spring_profiles_active=dev
```

PROFILI ATTIVAZIONE

Maven:

```
<profiles>
  <profile>
    <id>dev</id>
    <activation>
      <activeByDefault>true</activeByDefault>
    </activation>
    <properties>
      <spring.profiles.active>dev</spring.profiles.active> ← Valore che verrà inserito nel properties
    </properties>
  </profile>
  <profile>
    <id>prod</id>
    <properties>
      <spring.profiles.active>prod</spring.profiles.active>
    </properties>
  </profile>
</profiles>
```

PROFILI ATTIVAZIONE

- Maven. Nell'application.properties andiamo ad inserire un proprietà che verrà sovrascritta:
spring.profiles.active=@spring.profiles.active@
- Nel pom attiviamo la scrittura delle properties:

```
<build>  
  <resources>  
    <resource>  
      <directory>src/main/resources</directory>  
      <filtering>true</filtering>  
    </resource>  
  </resources>  
  ...  
</build>
```

Eseguiamo: mvn clean package -Pprod

LOGGING

- Spring Boot utilizza Apache Commons logging per le attività di logging. Spring Boot, di default, ha il supporto per Java Util Logging, Log4j2, e Logback.
- Utilizzando gli Spring Boot Starters, Logback fornisce un buon supporto per il logging. Logback fornisce un buon uso di: Common Logging, Util Logging, Log4J, e SLF4J.

LOGGING

- Per default, tutti i log sono stampati sulla console e non su file. Se vogliamo stampare logs su un file, abbiamo bisogno di settare la proprietà **logging.file.name** o **logging.path** nell'application.properties:

```
logging.path = /var/tmp/
```

```
logging.file.name = /var/tmp/mylog.log
```

- Spring Boot supporta tutti i diversi livelli: “TRACE”, “DEBUG”, “INFO”, “WARN”, “ERROR”, “FATAL”, “OFF”. Possiamo ridefinire il Root logger nell'application.properties:

```
logging.level.root = WARN
```

Messaggi possibili nel codice

- Ricordiamo l'ordine dei livelli:

TRACE < DEBUG < INFO < WARN < ERROR < FATAL < OFF

Questo significa che se impostiamo per un logger un determinato livello, tutti i messaggi nel codice che hanno livello uguale o superiore saranno attivi, gli altri disabilitati. Ad esempio stampare a livello info significa stampare tutti i messaggi di info, warning, error, fatal.

LOGGING LOGBACK (DEFAULT)

- Vediamo come configuraere Logback con differenti colori e patterns, e con una *rolling policy* per evitare file di grandi dimensioni.
- Non utilizziamo l'application.properties.
- **Quando nel classpath inseriamo un file con uno dei nomi elencati, Spring Boot automaticamente lo carica come configurazione:**
 - *logback-spring.xml*
 - *logback.xml*
 - *logback-spring.groovy*
 - *logback.groovy*

LOGGING XML CONFIGURATION

```
<?xml version="1.0" encoding="UTF-8"?>
<configuration>
    <property name="LOGS" value="/Users/JDK/mylog.log" />
    <appender name="Console"
        class="ch.qos.logback.core.ConsoleAppender">
        <layout class="ch.qos.logback.classic.PatternLayout">
            <Pattern>
                %black(%d{ISO8601}) %highlight(%-5level) [%blue(%t)] %yellow(%C{1.}): %msg%n%throwable
            </Pattern>
        </layout>
    </appender>
    <appender name="RollingFile"
        class="ch.qos.logback.core.rolling.RollingFileAppender">
        <file>${LOGS}/spring-boot-logger.log</file>
        <encoder
            class="ch.qos.logback.classic.encoder.PatternLayoutEncoder">
            <Pattern>%d %p %C{1.} [%t] %m%n</Pattern>
        </encoder>

        <rollingPolicy
            class="ch.qos.logback.core.rolling.TimeBasedRollingPolicy">
            <!-- rollover daily and when the file reaches 10 MegaBytes -->
            <fileNamePattern>${LOGS}/archived/mobile-logger-%d{yyyy-MM-dd}.%i.log
            </fileNamePattern>
            <timeBasedFileNamingAndTriggeringPolicy
                class="ch.qos.logback.core.rolling.SizeAndTimeBasedFNATP">
                <maxFileSize>10MB</maxFileSize>
            </timeBasedFileNamingAndTriggeringPolicy>
        </rollingPolicy>
    </appender>
    <!-- LOG everything at INFO level -->
    <root level="info">
        <appender-ref ref="RollingFile" />
        <appender-ref ref="Console" />
    </root>
    <!-- LOG at TRACE level -->
    <logger name="it.mobile" level="trace" additivity="false">
        <appender-ref ref="RollingFile" />
        <appender-ref ref="Console" />
    </logger>
</configuration>
```

SPRING DATA (GIORNO 2)

- Repository è l'interfaccia principale dell'astrazione che fornisce Spring Data per il concetto di repository. Riceve una classe di dominio ed il suo identificatore.
- Questa interfaccia agisce come marker interface per catturare il tipo sul quale lavorare ed comprendere le interfacce specifiche che la estendono.
- Ad esempio la CrudRepository interface fornisce metodi CRUD per l'entity gestita.

```
public interface CrudRepository<T, ID> extends Repository<T, ID> {  
    <S extends T> S save(S entity); //Salva l'entità  
    Optional<T> findById(ID primaryKey); // Ritorna l'entity identificata  
    Iterable<T> findAll(); // Ritorna tutte le entity  
    long count(); // Ritorna il numero di entity  
    void delete(T entity); // Cancella un entity  
    boolean existsById(ID primaryKey); // Verifica l'esistenza di un entity  
    // ... more functionality omitted.  
}
```

SPRING DATA

- Spring Data offre un'astrazione anche sulla specifica tecnologia : JpaRepository o MongoRepository ad esempio. Queste interfacce estendono CrudRepository ed espongono funzionalità specifiche della tecnologia di persistenza piuttosto che funzionalità *persistence technology-agnostic* dell'interfaccia CrudRepository.
- PagingAndSortingRepository è un'astrazione che estende CrudRepository per aggiungere metodi per l'ordinamento e la paginazione:

```
public interface PagingAndSortingRepository<T, ID> extends CrudRepository<T, ID> {
```

```
    Iterable<T> findAll(Sort sort);
```

```
    Page<T> findAll(Pageable pageable);
```

```
}
```

```
PagingAndSortingRepository<User, Long> repository = // ... get access to a bean  
Page<User> users = repository.findAll(PageRequest.of(1, 20));
```

Per accedere alla seconda pagina di dimensione 20.

SPRING DATA: QUERY METHODS

- Il proxy per il repository ha due metodi per derivare la query specifica per il sistema di persistenza:
 - Utilizzando la particolare struttura del nome.
 - Attraverso una query specificata direttamente.

Con la Java configuration, si può utilizzare l'attributo `queryLookupStrategy` dell'annotazione `Enable${store}Repositories`.

`CREATE`: cerca di costruire una query analizzando il nome del metodo.

`USE_DECLARED_QUERY`: cerca una query dichiarata per il metodo altrimenti lancia un errore. La query viene dichiarata attraverso annotazione.

`CREATE_IF_NOT_FOUND` (the default): combina `CREATE` e `USE_DECLARED_QUERY`. Cerca prima una query dichiarata, se non la trova analizza il nome del metodo per la creazione della query.

Abilitare uno specifico repository:

```
@Configuration  
@EnableJpaRepositories("com.acme.repositories")  
class ApplicationConfiguration {}
```

SPRING DATA: QUERY METHODS

- Il nome della query deve iniziare con uno dei seguenti prefissi: `find...By`, `read...By`, `query...By`, `count...By`, e `get...By`. Ognuno di essi è un alias per l'altro eccetto `count`. Oppure `exists ... by` che ritorna un booleano.
link: <https://docs.spring.io/spring-data/commons/docs/current/reference/html/#appendix.query.method.subject>

- La prima parte (`find...By`, `exists...By`) definisce il soggetto della query, la seconda parte forma il predicato.

`List<Dog> findByAgeAndHeight(Integer age, double height);`

`List<Dog> findByAgeAndNameAndColor(Integer age, String name, String color);`

`List<Dog> findByNameOrAge(String name, Integer age);`

`List<Dog> findByNomeIgnoreCaseAndColor(String name, String color);`

Le proprietà dell'espressione devono riferirsi a campi di entity. Ci sono situazioni in cui abbiamo riferimenti di classi personalizzate e vogliamo riferci ad adessi:

`List<Person> findByAddress_ZipCode(ZipCode zipCode);`

Campo address con al suo interno il campo zipCode.

SPRING DATA: QUERY METHODS

- Le espressioni sono proprietà combinate con operatori (AND, OR, Between, LessThan, GreaterThan).
- Il parser del metodo supporta anche il flag IgnoreCase per proprietà individuali(`findByLastnameIgnoreCase(...)`) o per tutte le proprietà che supportano un IgnoreCase (`findByLastnameAndFirstnameAllIgnoreCase(...)`).
- Possiamo inserire un'ordinamento con `OrderBy` (Asc o Desc).

Dog `findTopByOrderByBirthDateDesc();` (Individuare il cane più giovane.)

Selezione di un numero di record:

```
Dog findFirstByName(String name);
```

```
Dog findTopByName(String name);
```

```
List<Dog> findTop10ByColor(String color);
```

SPRING DATA: QUERY METHODS

```
interface PersonRepository extends Repository<Person, Long> {  
  
    List<Person> findByEmailAddressAndLastname(EmailAddress emailAddress, String lastname);  
  
    // Enables the distinct flag for the query  
    List<Person> findDistinctPeopleByLastnameOrFirstname(String lastname, String firstname);  
    List<Person> findPeopleDistinctByLastnameOrFirstname(String lastname, String firstname);  
  
    // Enabling ignoring case for an individual property  
    List<Person> findByLastnameIgnoreCase(String lastname);  
    // Enabling ignoring case for all suitable properties  
    List<Person> findByLastnameAndFirstnameAllIgnoreCase(String lastname, String firstname);  
  
    // Enabling static ORDER BY for a query  
    List<Person> findByLastnameOrderByFirstnameAsc(String lastname);  
    List<Person> findByLastnameOrderByFirstnameDesc(String lastname);  
}
```

SPRING DATA: QUERY METHODS

Paginazione:

Page<User> **findByLastname**(String lastname, Pageable pageable);

Slice<User> **findByLastname**(String lastname, Pageable pageable);

List<User> **findByLastname**(String lastname, Sort sort);

List<User> **findByLastname**(String lastname, Pageable pageable);

Il primo metodo prende un'istanza org.springframework.data.domain.Pageable aggiungendo dinamicamente la paginazione al risultato della query. Un oggetto Page conosce il numero totale di elementi e le pagine disponibili. Viene effettuata una query aggiuntiva per il calcolo del numero di elementi. Se ritenuta costosa si può utilizzare uno Slice. *Uno Slice conosce solo se è disponibile un nuovo Slice.*

SPRING DATA: QUERY METHODS

Opzioni di ordinamento sono disponibili anche con l'istanza Pageable. Se abbiamo bisogno solo dell'ordinamento possiamo utilizzare org.springframework.data.domain.Sort.

E' possibile ritornare una lista. In questo caso, i metadati addizionali richiesti da Page non sono creati e quindi la query aggiuntiva non viene eseguita.

Esempio di Sort:

```
Sort sort = Sort.by("firstname").ascending() .and(Sort.by("lastname").descending());
```

SPRING DATA: QUERY METHODS

Query methods che ritornano più risultati possono utilizzare Java Iterable, List, e Set oppure Spring Data's Streamable, un'estensione custom di Iterable.

```
interface PersonRepository extends Repository<Person, Long> {  
    Streamable<Person> findByFirstnameContaining(String firstname);  
    Streamable<Person> findByLastnameContaining(String lastname);  
}
```

Streamable può essere utilizzato come alternativa ad Iterable o collezione Java. Fornisce metodi convenienti per accedere ad uno Stream non parallelo (filter(...),map(...)).

SPRING DATA: QUERY METHODS

- Da Spring Data 2.0, i metodi dei repository CRUD che ritornano una singola istanza possono usare il tipo Optional per indicare la possibile assenza di un valore.
- Alternativamente l'assenza di un valore viene indicata dal valore null.
- Metodi che ritornano collections, wrappers, e streams non ritornano null ma un valore empty.

E' possibile esprimere **vincoli null** per i metodi di un repository utilizzando le annotations:

@NonNullApi: Usato a livello di package per dichiarare che il comportamento di default per parametri e valori di ritorno non sono accettati valori null.

@NonNull: Utilizzato su un parametro o un tipo di ritorno che non deve essere null.

@Nullable: Utilizzato su un parametro o tipo di ritorno che può essere null.

SPRING DATA: QUERY METHODS

- Da Spring Data 2.0, i metodi dei repository CRUD che ritornano una singola istanza possono usare il tipo Optional per indicare la possibile assenza di un valore.
- Alternativamente l'assenza di un valore viene indicata dal valore null.
- Metodi che ritornano collections, wrappers, e streams non ritornano null ma un valore empty.

```
interface UserRepository extends Repository<User, Long> {  
    User getByEmailAddress(Address emailAddress);  
    @Nullable  
    User findByEmailAddress(@Nullable Address emailAdress);  
    Optional<User> findOptionalByEmailAddress(Address emailAddress);  
}
```

SPRING DATA: QUERY METHODS

You can process the results of query methods incrementally by using a Java 8 Stream<T> as the return type. Instead of wrapping the query results in a Stream, data store-specific methods are used to perform the streaming, as shown in the following example:

```
@Query("select u from User u")
```

```
Stream<User> findAllByCustomQueryAndStream();
```

```
Stream<User> readAllByFirstnameNotNull();
```

```
@Query("select u from User u")
```

```
Stream<User> streamAllPaged(Pageable pageable);
```

Uno stream deve essere chiuso dopo il suo utilizzo: metodo close() oppure Java 7 try-with-resources block:

```
try (Stream<User> stream = repository.findAllByCustomQueryAndStream()) { stream.forEach(...); }
```

SPRING DATA: NAMED QUERIES

Alcune volte con i method query non riusciamo ad esprimere una determinata query, oppure il nome diventa troppo lungo perdendo leggibilità. In questi casi possiamo utilizzare l'annotation @Query.

```
@Query(" SELECT MAX(eventId) AS eventId FROM Event ")  
Long lastProcessedEvent();
```

CRITERIA API

- Query costruite attraverso oggetti Java
- **CriteriaBuilder**: permette la creazione di Query
- **CriteriaQuery**: raccoglie la parte Select, From e Where per la costruzione della Query
- **Query Root**: definisce cosa interroghiamo
- Analizziamo query nell'IDE...

@TRANSACTIONAL ANNOTATION (DECLARATIVE TRANSACTION MANAGEMENT)

- In Spring Boot il transaction manager è abilitato di default. Tutto ciò che dobbiamo fare per gestire le transazioni è l'utilizzo dell'annotation `@Transactional`. Questa annotazione supporta:
 - *Propagation Type*
 - *Isolation Level*
 - *Timeout*
 - a *readOnly flag*
 - *Rollback rules for the transaction*

Per default, il rollback viene attivato solo per le runtime unchecked exceptions.

Un checked exception non porta ad un rollback della transazione. Utilizzare `rollbackFor / noRollbackFor` dell'annotation Transactional in questo caso.

@TRANSACTIONAL ANNOTATION (DECLARATIVE TRANSACTION MANAGEMENT)

- **Spring crea proxy per tutte le classi annotate con *@Transactional*.** Il proxy consente al framework di inserire logica prima e dopo l'invocazione del metodo..
- Per default il proxy è un Java Dynamic Proxy. Questo significa che soltanto le chiamate che passano dal proxy sono intercettate. **Ogni self-invocation non attiva transazioni.**
- Soltanto metodi **public dovrebbero essere *@Transactional*.** Metodi con altre visibilità saranno semplicemente ignorati dal contesto transazionale.

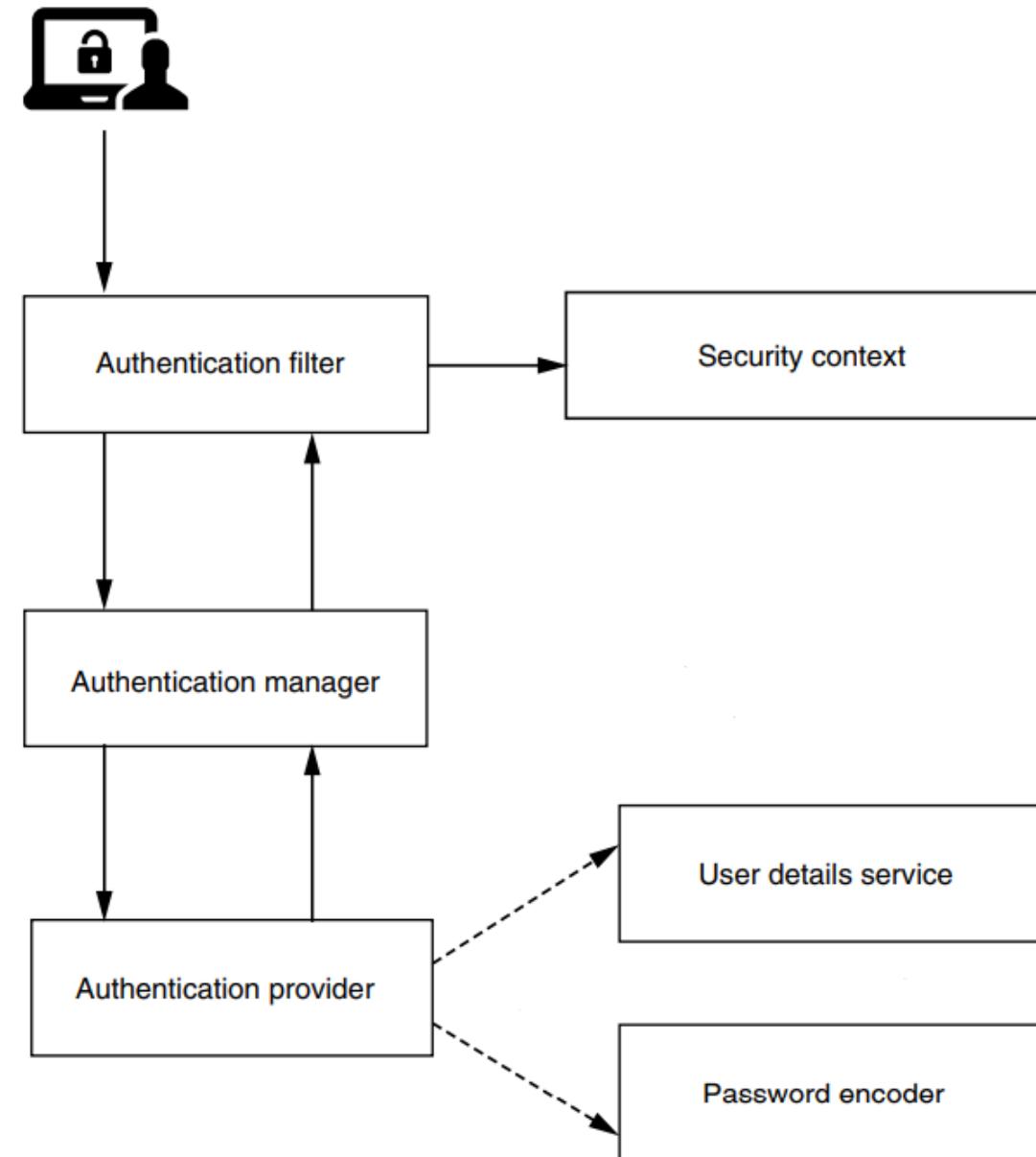
@ TRANSACTIONAL ANNOTATION (DECLARATIVE TRANSACTION MANAGEMENT)

- **Read Only Transaction:** Un hint per il sottosistema transazionale; non causa necessariamente errori se si tentano delle scritture.
- Lo utilizziamo con JPA per evitare il caricamento ed il processamento dell'Hydrated State di un entity. Quindi per transazioni che non eseguono statements di inserimento/aggiornamento o cancellazione.

PROPAGAZIONE DELLE TRANSAZIONI(@TRANSACTIONAL)

- **Required (default)**: Si apre una transazione se non già esistente.
- **Supports**: Esecuzione con o senza transazione attiva.
- **Mandatory**: Deve esistere una transazione attiva.
- **Require_new**: La transazione esistente viene sospesa e se ne apre un'altra.
- **Not_Supported**: Sospende la transazione corrente.
- **Never**: Errore se una transazione è attiva.
- **Nested**: Spring controlla se una transazione esiste, se si crea un save point. Se la logica di business Lancia un'eccezione, abbiamo un rollback al punto di salvataggio. Come Required se non ci sono transazioni attive.

SPRING SECURITY ARCHITECTURE



SPRING SECURITY ARCHITECTURE

- L'Authentication filter delega la richiesta di autenticazione all'Authentication manager e, in base alla risposta, configura il security context.
- L'authentication manager utilizza l'Authentication provider per processare l'autenticazione. L'Authentication provider implementa la logica di autenticazione.
- Il servizio UserDetails implementa la gestione degli utenti, utilizzata dall'Authentication Provider.
- Il Password encoder implementa la gestione delle password, utilizzata dall'Authentication Provider
- Il security context mantiene le informazioni di autenticazione.

USERDETAILSSERVICE

- Utilizzato dall'Authentication Provider per identificare utenti
- Un oggetto che implementa l'interfaccia UserDetailsService (contratto) di Spring Security gestisce i dati utente.
- In precedenza abbiamo utilizzato l'implementazione di default di Spring Boot. Questa implementazione registra delle credenziali di default nella memoria dell'applicazione. Queste credenziali sono **user** con una password rappresentata da uno universally unique identifier (UUID).
- L'implementazione di default serve solo per testare.

PASSWORD ENCODER

- Codifica una password
- Verifica se una password è in match con un certo encoding
- Anche se non così evidente come per lo UserDetailsService object, il PasswordEncoder è obbligatorio per una basic authentication. L'implementazione più semplice gestisce password in chiaro.
- Quando ridefiniamo l'implementazione di default di un UserDetailsService, dobbiamo anche specificare un PasswordEncoder.

HTTP VS HTTPS

- Con Spring Boot è possibile abilitare l'HTTPS a livello di applicazione.
- Si ha bisogno di un certificato firmato da una certification authority (CA). Con questo certificato, il client che invoca gli endpoints sa se la risposta proviene dal server. Nessuno può intercettare la comunicazione.
- Per configurare l'HTTPS soltanto per il test di applicazione, possiamo generare certificati autofirmati utilizzando OpenSSL.

OPENSSL

Certificato autofirmato:

```
openssl req -newkey rsa:2048 -x509 -keyout key.pem -out cert.pem -days 365
```

L'outputs è costituito da due file: **key.pem** (the private key) and **cert.pem** (a public certificate)

Keystore (Public Key Cryptography Standards (pkcs) o Java KeyStore (JKS)):

```
openssl pkcs12 -export -in cert.pem -inkey key.pem -out certificate.p12 -name "certificate"
```

application.properties:

```
server.ssl.key-store-type=PKCS12
```

```
server.ssl.key-store=classpath:certificate.p12
```

```
server.ssl.key-store-password=1234
```

GESTIONE UTENTI

- **UserDetails**: descrive un utente in Spring Security.
- **GrantedAuthority**: le azioni che un utente può eseguire.
- **UserDetailsManager** che estende lo **UserDetailsService** contract. Gestisce anche la creazione di un utente e la cancellazione o modifica di una password.

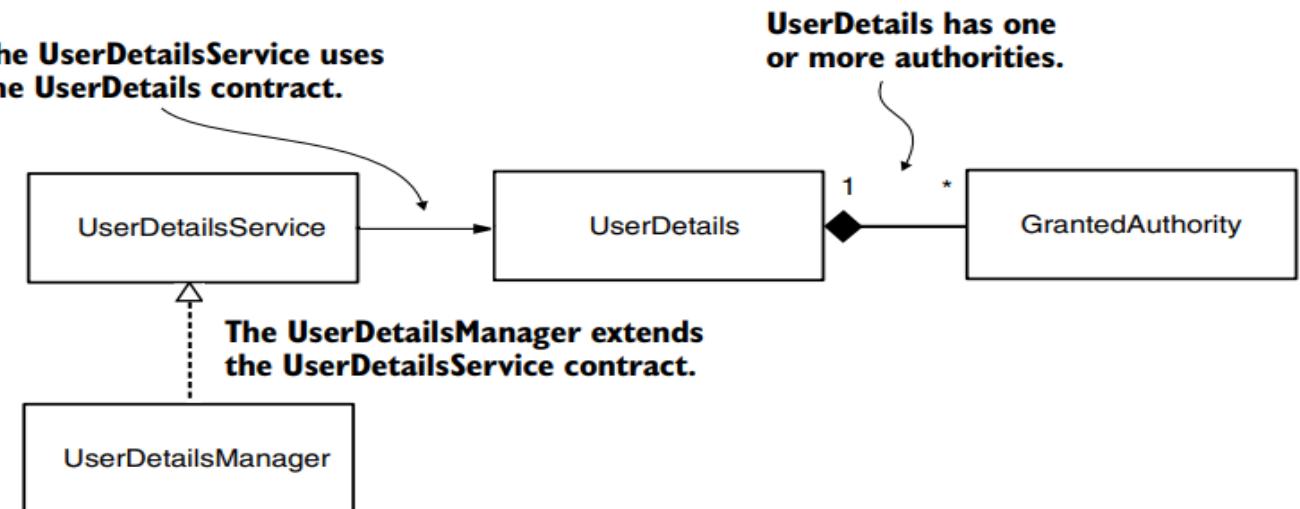
GESTIONE UTENTI

- Cerchiamo di comprendere come Spring Security definisce e gestisce utenti ed autorizzazioni (`UserDetails` e `GrantedAuthority`).
- Andiamo in dettaglio su `UserDetailsService` e come `UserDetailsManager` la estende.
- Esistono diverse implementazioni per `UserDetailsManager` (`InMemoryUserDetailsManager`, `JdbcUserDetailsManager`, `LdapUserDetailsManager`).

INTERFACCE USERDETAILS E GRANTEDAUTHORITY

- UserDetails descrive un utente nel modo compreso da Spring Security.
- GrantedAuthority descrive ciò che un utente può fare(authorities).

INTERFACCE USERDETAILS E GRANTEDAUTHORITY



Dependencies between the components involved in user management. The **UserDetailsService** returns the details of a user, finding the user by its name. The **UserDetails** contract describes the user. A user has one or more authorities, represented by the **GrantedAuthority** interface. To add operations such as create, delete, or change password to the user, the **UserDetailsManager** contract extends **UserDetailsService** to add operations.

USER DETAILS

```
public interface UserDetails extends Serializable {  
    String getUsername();  
    String getPassword();  
    Collection<? extends GrantedAuthority>  
        getAuthorities();  
    boolean isAccountNonExpired();  
    boolean isAccountNonLocked();  
    boolean isCredentialsNonExpired();  
    boolean isEnabled();  
}
```

These methods return the user credentials.

Returns the actions that the app allows the user to do as a collection of GrantedAuthority instances

These four methods enable or disable the account for different reasons.

- Let the account expire
- Lock the account
- Let the credentials expire
- Disable the account

Per un utente possiamo, in base al contratto UserDetails, effettuare le operazioni

GRANTED AUTHORITY

```
public interface GrantedAuthority extends Serializable {  
    String getAuthority();  
}
```

- Le authorities rappresentano ciò che può fare un utente in una applicazione. Senza authorities, tutti gli utenti sono uguali.
- Un'applicazione può avere utenti che leggono soltanto dati ed altri che possono modificarli ecc...

USERDETAILSMANGER

```
public interface UserDetailsManager extends UserDetailsService {  
    void createUser(UserDetails user);  
    void updateUser(UserDetails user);  
    void deleteUser(String username);  
    void changePassword(String oldPassword, String newPassword);  
    boolean userExists(String username);  
}
```

Spring Security utilizza il contratto esposto dalla UserDetailsService per l'autenticazione. Le applicazioni in generale hanno anche bisogno di gestire utenti(aggiunta,aggiornamento, cancellazione).In questo caso implementiamo la UserDetailsManager.

INTERFACCIA PASSWORD ENCODER

- Decide se una password è valida oppure no (matches())
- Può codificare password (encode())
- I metodi encode() e matches() sono fortemente relazionati per questo fanno parte dello stesso contratto offerto da PasswordEncoder.
- La modalità con cui un'applicazione codifica una password è legata alla modalità con cui viene validata.

```
public interface PasswordEncoder {  
  
    String encode(CharSequence rawPassword);  
    boolean matches(CharSequence rawPassword, String encodedPassword);  
  
    default boolean upgradeEncoding(String encodedPassword) {  
        return false;  
    }  
}
```

PASSWORD ENCODER SENZA CODIFICA

```
public class PlainTextPasswordEncoder  
    implements PasswordEncoder {  
  
    @Override  
    public String encode(CharSequence rawPassword) {  
        return rawPassword.toString();           ← We don't change the password,  
    }                                         just return it as is.  
  
    @Override  
    public boolean matches(  
        CharSequence rawPassword, String encodedPassword) {  
        return rawPassword.equals(encodedPassword); ← Checks if the two  
    }                                         strings are equal  
}
```

PASSWORD ENCODER CODIFICA SHA-512

```
public class Sha512PasswordEncoder
    implements PasswordEncoder {

    @Override
    public String encode(CharSequence rawPassword) {
        return hashWithSHA512(rawPassword.toString());
    }

    @Override
    public boolean matches(
        CharSequence rawPassword, String encodedPassword) {
        String hashedPassword = encode(rawPassword);
        return encodedPassword.equals(hashedPassword);
    }
}

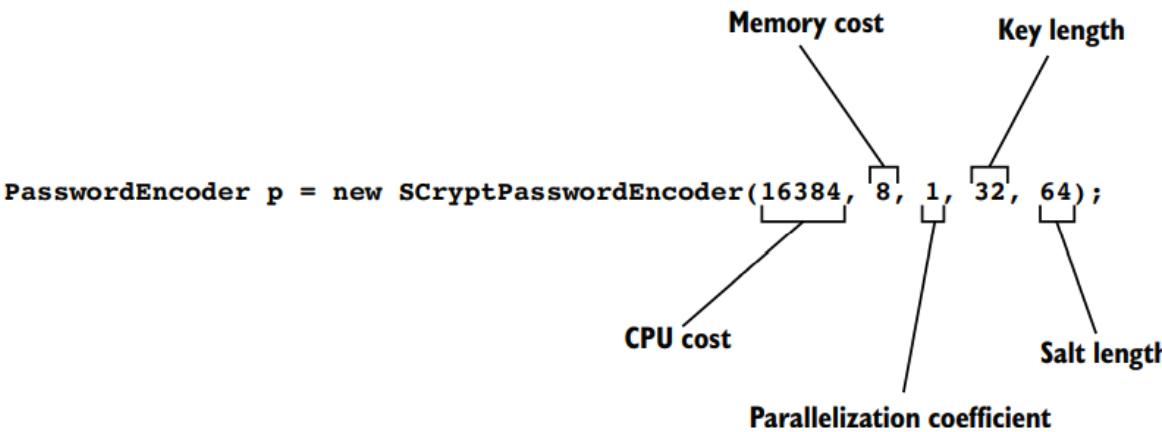
private String hashWithSHA512(String input) {
    StringBuilder result = new StringBuilder();
    try {
        MessageDigest md = MessageDigest.getInstance("SHA-512");
        byte [] digested = md.digest(input.getBytes());
        for (int i = 0; i < digested.length; i++) {
            result.append(Integer.toHexString(0xFF & digested[i]));
        }
    } catch (NoSuchAlgorithmException e) {
        throw new RuntimeException("Bad algorithm");
    }
    return result.toString();
}
```

PROBLEMS WITH CRYPTOGRAPHIC HASH ALGORITHM

- **Brute Force attack:** I valori hash non possono essere invertiti per ottenere la password di partenza. Un attaccante cerca quindi di generare una password per riottenere il valore hash.
- **Hash Collision attack:** Una funzione hash può produrre lo stesso valore per differenti input. MD5, SHA1, SHA2 sono vulnerabili a questo tipo di attacco.

PASSWORD ENCODER IMPLEMENTAZIONI SPRING SECURITY

- NoOpPasswordEncoder : password in chiaro
- StandardPasswordEncoder: SHA-256 per l'hash della password
- Pbkdf2PasswordEncoder: HMAC ripetuto in base ai parametri (password, numero di iterazioni, lunghezza chiave)
- BCryptPasswordEncoder: strong hashing function (log round strength coefficient)
- SCryptPasswordEncoder: script hashing function

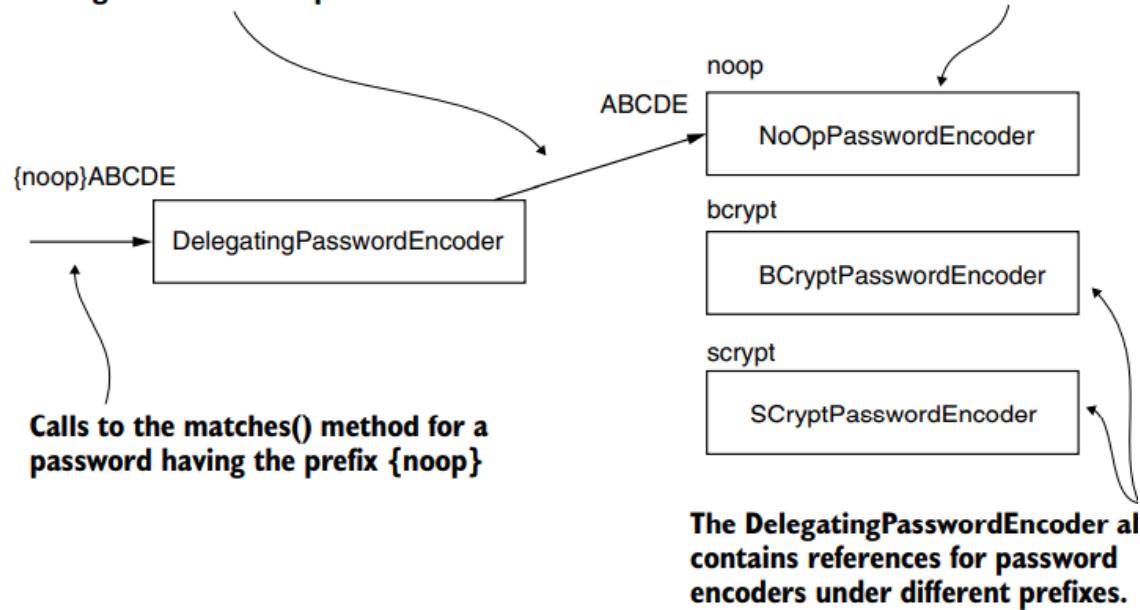


```
PasswordEncoder p = new BCryptPasswordEncoder();  
PasswordEncoder p = new BCryptPasswordEncoder(4);
```

← $2^{\text{log rounds}}$

MULTIPLE ENCODING STRATEGIES: DELEGATINGPASSWORDENCODER

When you call the matches() method with a password and the prefix {noop}, the call is delegated to the NoOpPasswordEncoder.



- In alcune applicazioni è possibile avere la necessità di utilizzare più encoders.
- Uno scenario comune riguarda applicazioni per le quali si cambia in produzione l'algoritmo di encoding, a partire da una particolare versione.
- Immaginiamo che una vulnerabilità è identificata per il corrente algoritmo. Si vuole cambiare encoding per i nuovi utenti lasciando invariato l'algoritmo per quelli esistenti non potendo cambiare la password. Come gestire questa molteplicità di hash?
- Una soluzione è il DelegatingPasswordEncoder

MULTIPLE ENCODING STRATEGIES: DELEGATING PASSWORD ENCODER

- Ogni password codificata sul database deve essere aggiornata in modo tale che abbia come prefisso l'algoritmo utilizzato.
- Ad esempio la password di Paul diventerebbe:

```
{sha512}d44559f602eab6fd62ac7680dacbfaadd13630335e951f97af390e9de176b6  
db28512f2e0b9d4fba5133e8b1c6e8df59db3a8ab9d60be4b97cc9e81db
```

Richiamando l'url:

- curl -k -u Paul2:1234 <https://localhost:8080/hello>,
- Il DelegatingPasswordEncoder, estrae dalla password il prefisso ed utilizza l'algoritmo associato.

RIEPILOGO

The interfaces that represent the main contracts for authentication flow in Spring Security

Contract	Description
UserDetails	Represents the user as seen by Spring Security.
GrantedAuthority	Defines an action within the purpose of the application that is allowable to the user (for example, read, write, delete, etc.).
UserDetailsService	Represents the object used to retrieve user details by username.
UserDetailsManager	A more particular contract for UserDetailsService. Besides retrieving the user by username, it can also be used to mutate a collection of users or a specific user.
PasswordEncoder	Specifies how the password is encrypted or hashed and how to check if a given encoded string matches a plaintext password.

AUTHENTICATION PROVIDER

- E' il componente che implementa la logica di autenticazione
- Il suo output può essere di due tipi:
 1. Il client che effettua la richiesta non viene autenticato. L'utente non viene identificato e la richiesta viene rigettata generalmente con un HTTP 401 Unauthorized.
 2. Il client è autenticato. I dettagli dell'utente vengono memorizzati in un oggetto di una classe che implementa l'interfaccia **SecurityContext**.

AUTHENTICATION PROVIDER

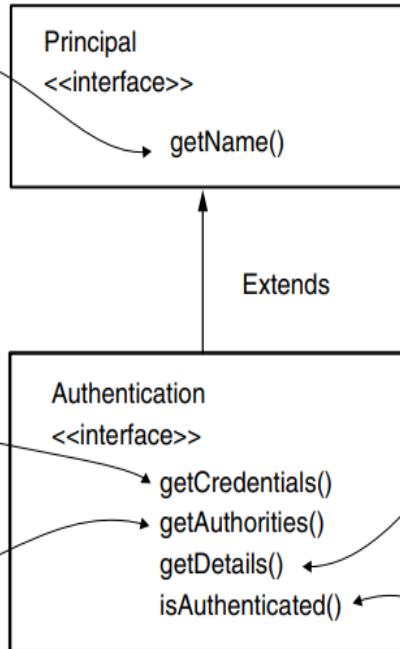
```
public interface AuthenticationProvider {  
  
    Authentication authenticate(Authentication authentication)  
        throws AuthenticationException;  
  
    boolean supports(Class<?> authentication);  
}
```

- Il metodo authenticate() riceve un oggetto Authentication come parametro e ritorna un oggetto Authentication se l'autenticazione ha successo. Altrimenti:
 1. Lancia un'AuthenticationException se l'autenticazione fallisce
 2. Ritorna null se l'oggetto Authentication non è supportato dall'implementazione
- Il metodo supports() consente di definire quale classe di tipo Authentication è supportata

**Someone who wants
to authenticate at least
needs to have a name.**

**For authentication in Spring Security,
the user also needs a secret:
a password, a code, a fingerprint . . .**

**Once authenticated, we also
need to know the user's
privileges. In Spring Security,
these are represented by
the authorities.**



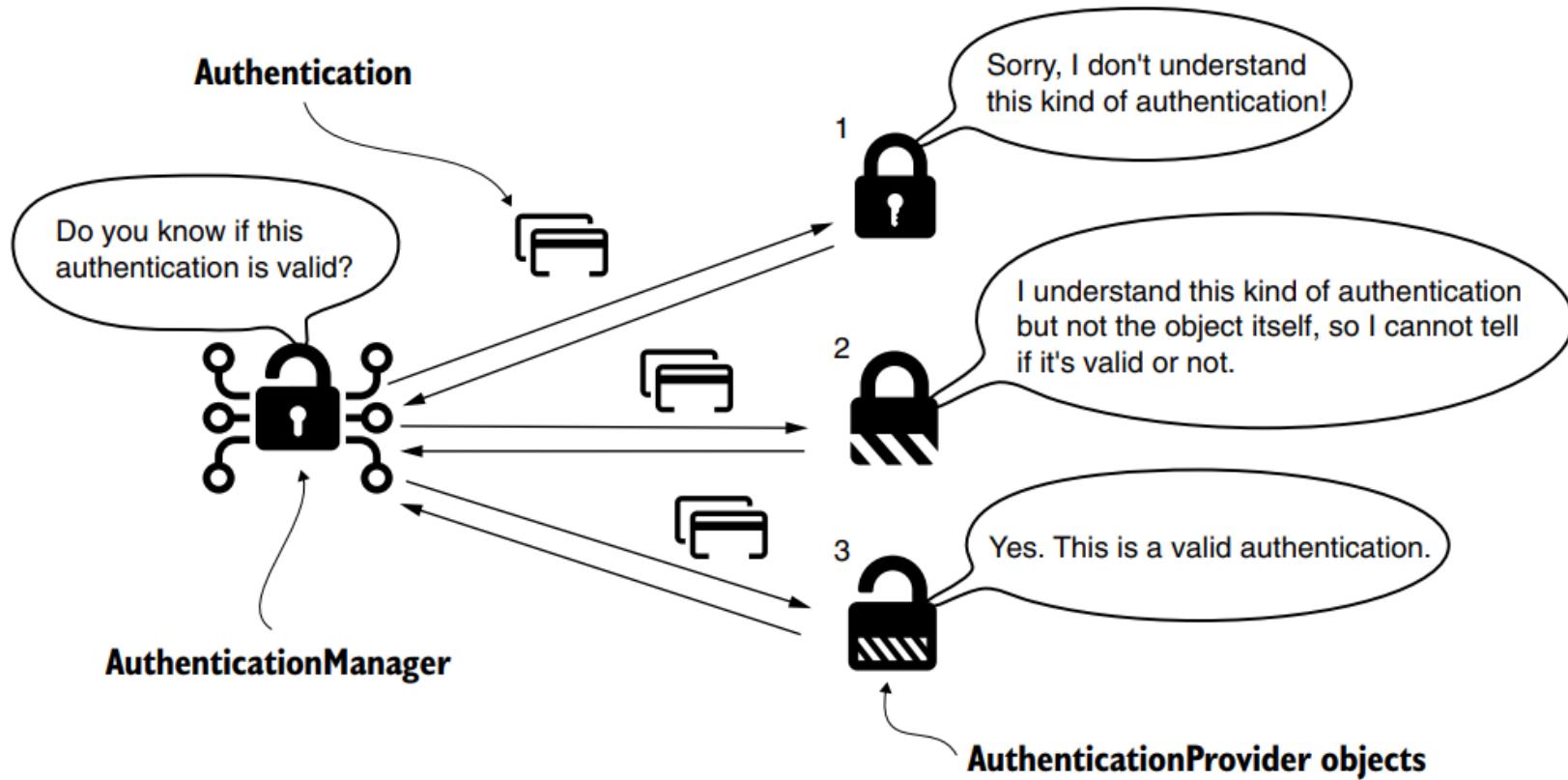
**If the system needs some more
details about the request, you
can provide them by overriding
the getDetails() method.**

**An authentication object
is either authenticated
or in the process of
being authenticated.**

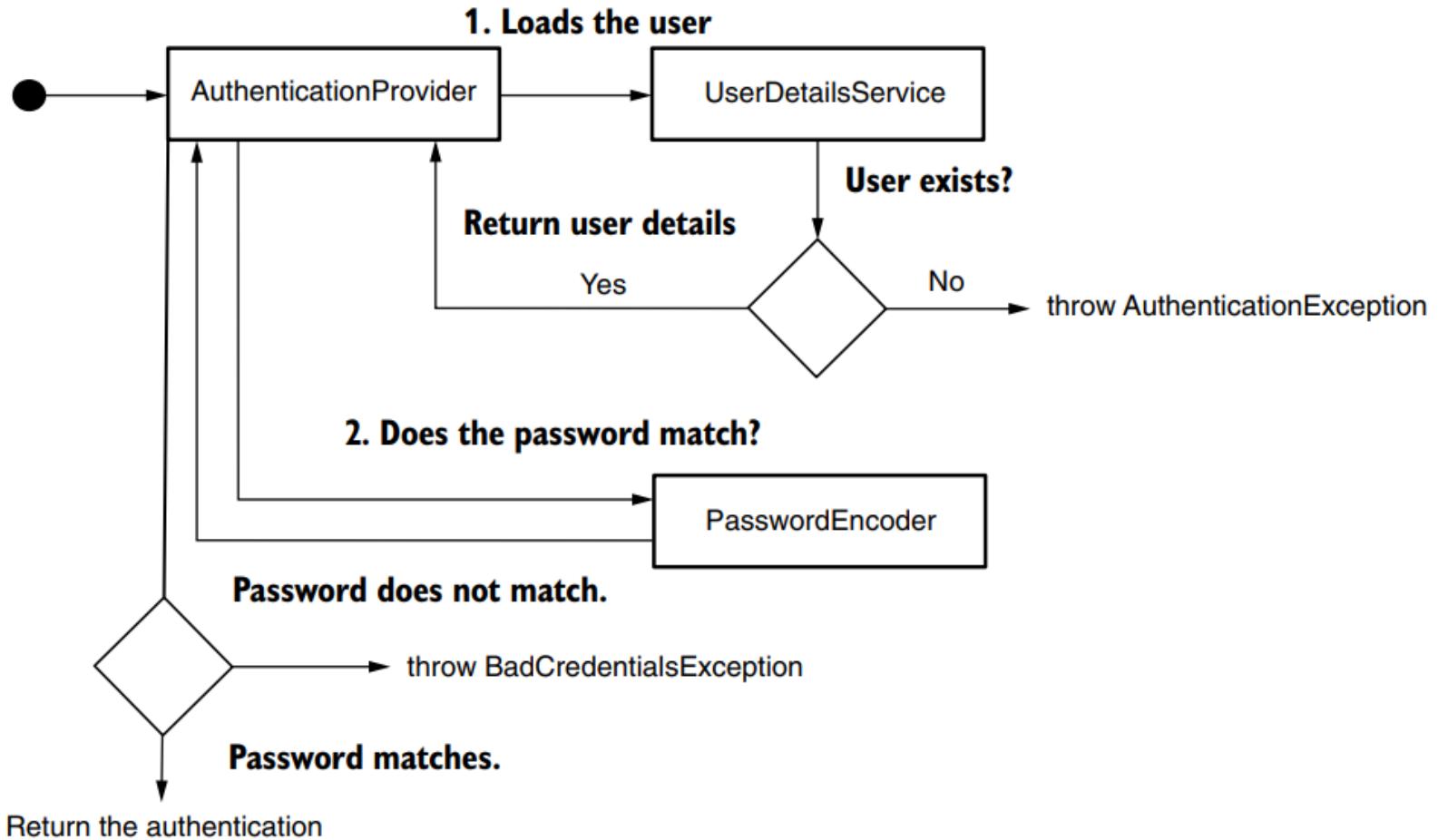
CONTRATTO AUTENTICAZIONE

```
public interface Authentication extends Principal, Serializable {
    Collection<? extends GrantedAuthority> getAuthorities();
    Object getCredentials();
    Object getDetails();
    Object getPrincipal();
    boolean isAuthenticated();
    void setAuthenticated(boolean isAuthenticated)
        throws IllegalArgumentException;
}
```

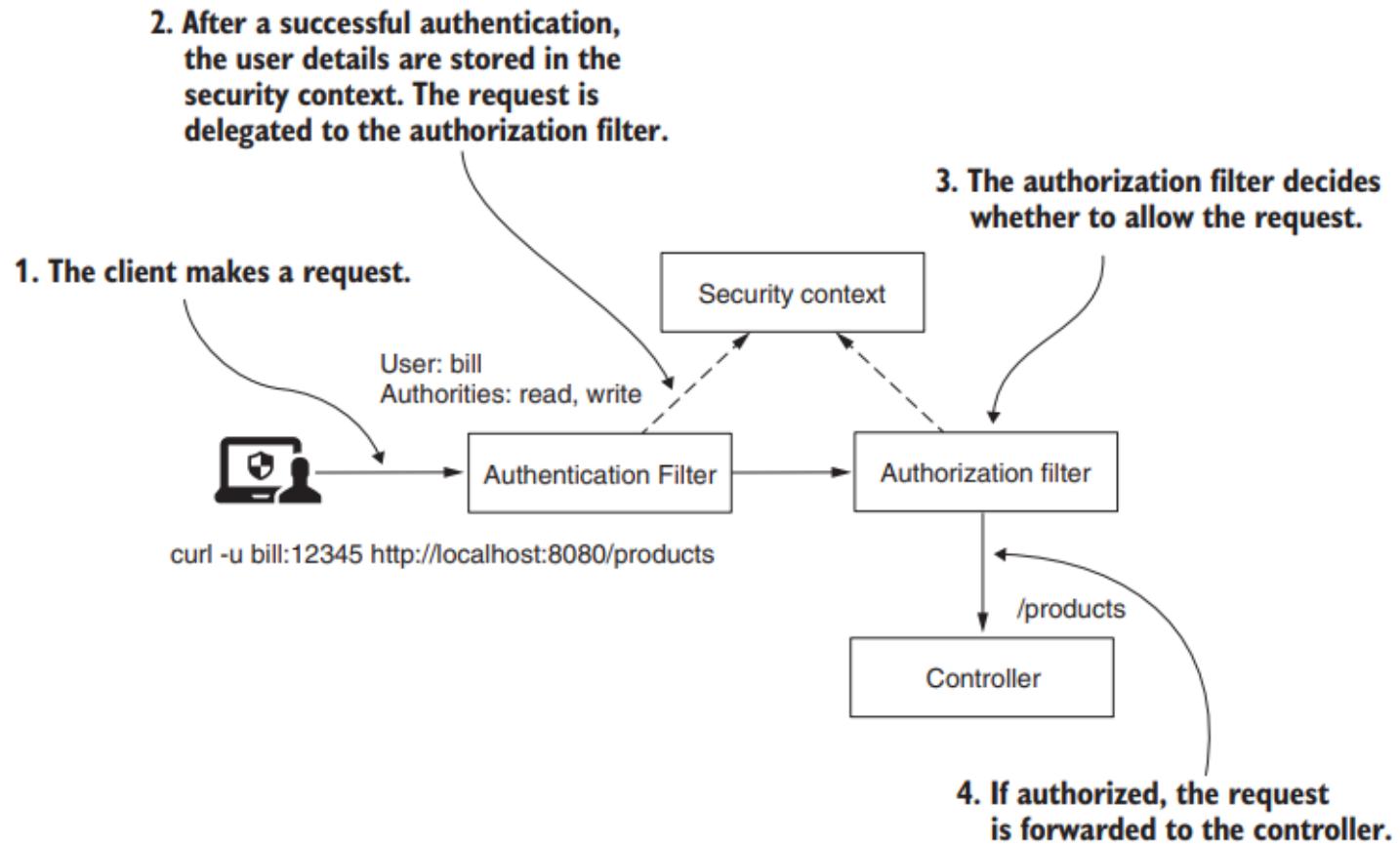
AUTHENTICATION PROVIDER COMPORTAMENTO GENERALE



AUTHENTICATION PROVIDER CUSTOM FLOW



AUTHORIZATION



```
public interface GrantedAuthority extends Serializable {  
    String getAuthority();  
}  
  
public interface UserDetails extends Serializable {  
    Collection<? extends GrantedAuthority> getAuthorities();  
  
    // Omitted code  
}
```

The **UserDetailsService** retrieves the user details during the authentication process.

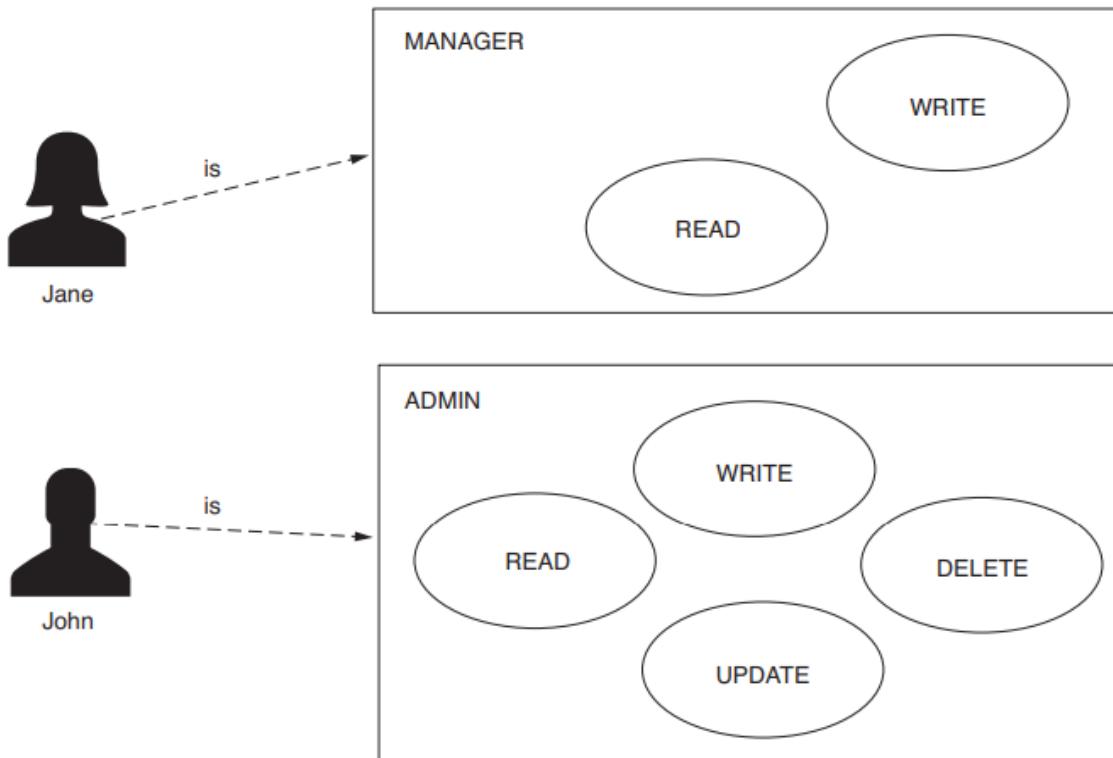


A user has one or more authorities.

A U T H O R I T Y

- Un authority è un azione che un utente può fare su una risorsa.
- Un authority ha un nome ritornato dal metodo `getAuthority()`
- Utilizziamo il nome per definire la regola dell'autorizzazione.
- Spesso una regola di autorizzazione è del tipo: "A Jane è consentito cancellare prodotti," oppure "A John è consentito leggere documenti."

MODALITÀ DI CONFIGURAZIONE PER L'ACCESSO ALLE RISORSE (RUOLI)



- **hasAuthority()**
riceve come parametro una singola stringa di autorizzazione
- **hasAnyAuthority()**
riceve come parametro più stringhe di autorizzazione
- **access()**
accetta una Spring Expression Language (SpEL)

- **hasRole()**
riceve come parametro una singola stringa di ruolo
- **hasAnyRole()**
riceve come parametro più stringhe di ruolo
- **access()**
accetta una Spring Expression Language (SpEL)

SECURITYCONTEXT

```
public interface SecurityContext extends Serializable {  
  
    Authentication getAuthentication();  
    void setAuthentication(Authentication authentication);  
}
```

ACCESSO AL SECURITYCONTEXT

```
@RestController  
  
public class HelloController {  
  
    @GetMapping("/hello")  
    public String hello() {  
  
        SecurityContext context = SecurityContextHolder.getContext();  
        System.out.println("SecurityContext:" + context.toString());  
        return "Hello!";  
    }  
}
```

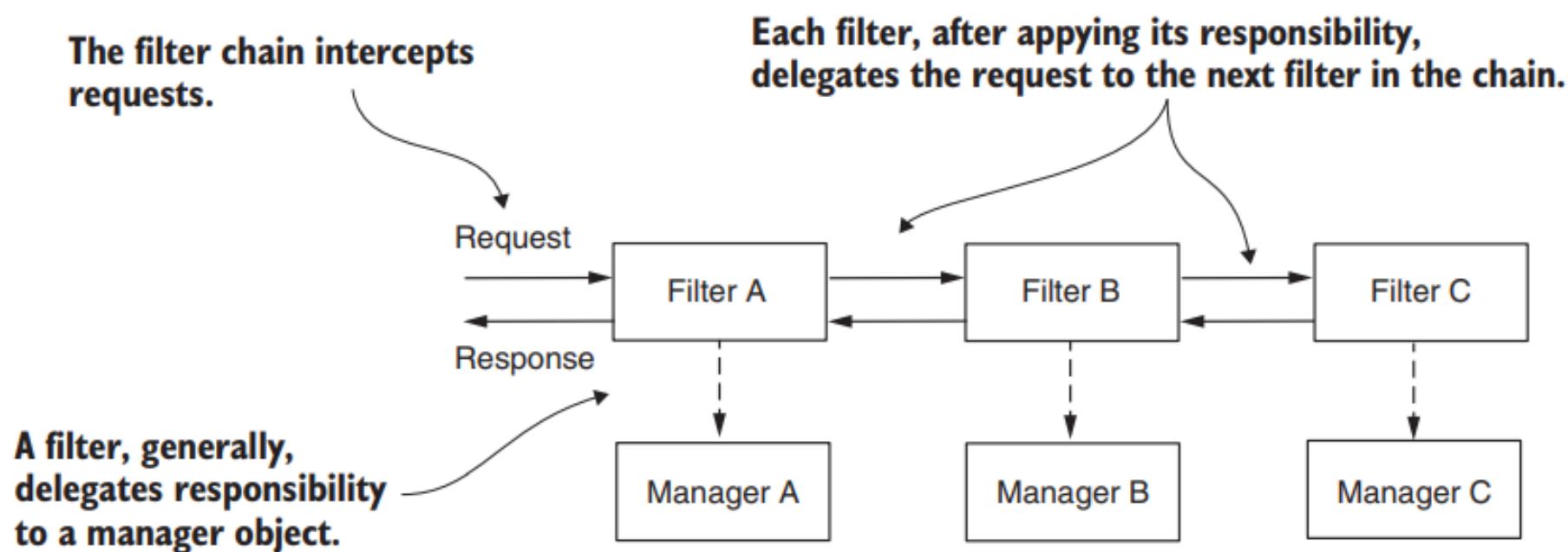
ACCESSO AL SECURITYCONTEXT

```
@RestController  
  
public class HelloController {  
  
    @GetMapping("/hello")  
    public String hello(Authentication context) {  
  
        System.out.println("SecurityContext:" + context.toString());  
        return "Hello!";  
    }  
}
```

FILTRI IN SPRING SECURITY

- Il primo layer dell'architettura di una web application, la quale intercetta richieste HTTP , è una **filter chain (catena di filtra)**.
- E' possibile personalizzare questa catena aggiungo un filtro prima, dopo o nella posizione di un altro.
- Possiamo avere più filtri in una certa posizione. In questo caso l'ordine di esecuzione non è definito.
- Cambiare la catena di filtra permette di personalizzare il comportamento di Spring Security per l'autenticazione ed autorizzazione.

FILTRI IN SPRING SECURITY



FILTRI IN SPRING SECURITY



FILTER 1

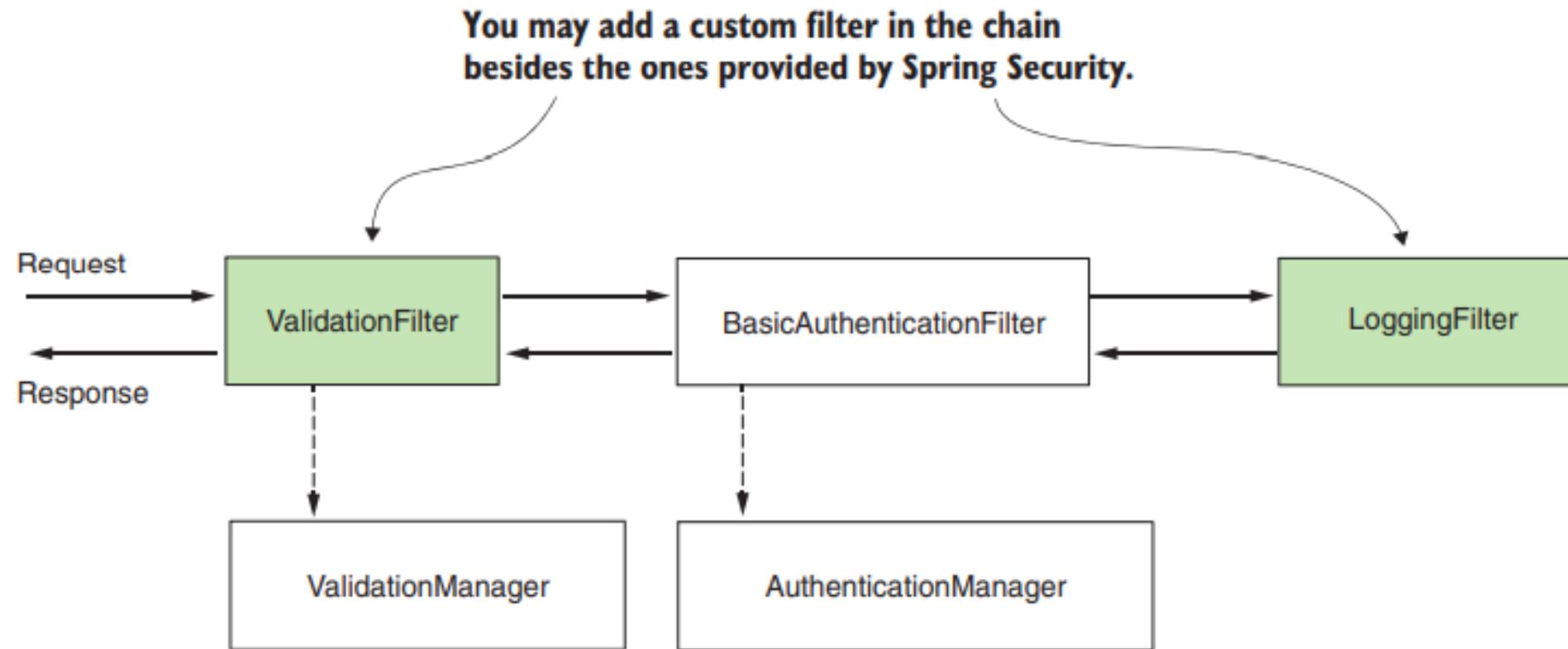


FILTER 2



FILTER 3

FILTRI IN SPRING SECURITY



FILTRI IN SPRING SECURITY

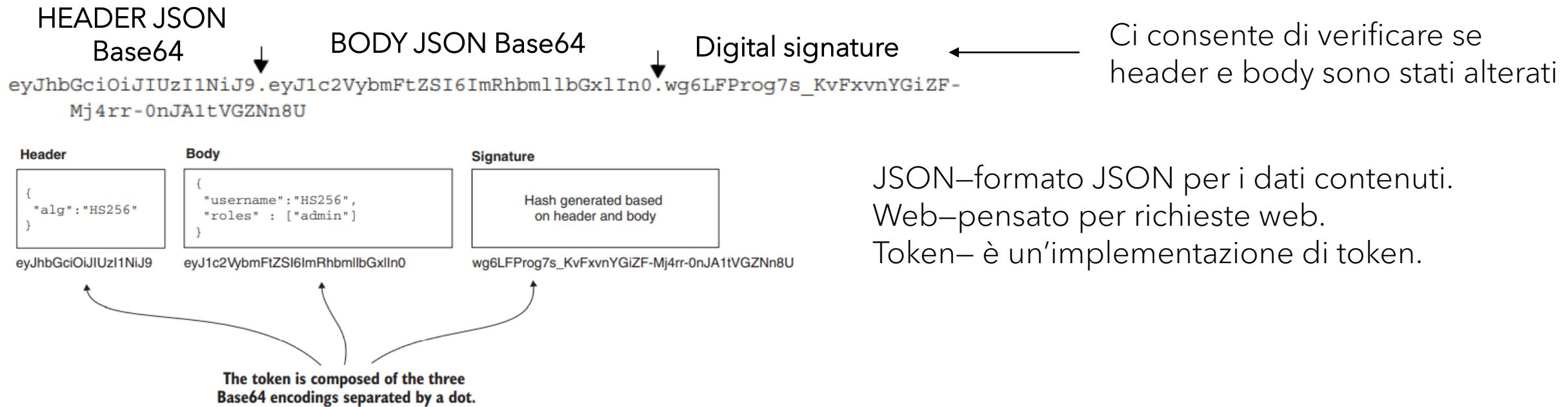
- I filtri in Spring Security sono classici Java Enterprise HTTP filters.
- Possiamo implementare un filtro utilizzando l'interfaccia **Filter** del package `javax.servlet`.
- Così come per ogni HTTP filter, abbiamo bisogno di ridefinire il metodo `doFilter()` per implementare la sua logica.
- Questo metodo riceve in input i parametri `ServletRequest`, `ServletResponse`, e `FilterChain`.

FILTRI IN SPRING SECURITY

- Spring Security fornisce alcune implementazioni di filtri. Tra questi abbiamo:
 - **BasicAuthenticationFilter** che si occupa della basic authentication.
 - **CsrfFilter** per la cross-site request forgery (CSRF) protection.
 - **CorsFilter** per le regole di autorizzazione riguardo la cross-origin resource sharing (CORS).

JSON Web Tokens (JWTs) **Cosa sono I token?**

- Header and the body memorizzano i dettagli del token:



JSON-formato JSON per i dati contenuti.
Web-pensato per richieste web.
Token- è un'implementazione di token.

- I Tokens aiutano ad evitare l'invio di credenziali ad ogni request.
- I Tokens possono essere definiti con uno short lifetime.
- I Tokens possono essere invalidati senza invalidare le credenziali.
- I Tokens possono incapsulare anche autorizzazioni.

JSON WEB TOKENS (JWT)

- Uno scenario molto semplice per l'utilizzo dei JWT prevede:
 1. Generazione di un token da parte di un authorization server
 2. Utilizzo del token verso un resource server

SECURITY VULNERABILITIES IN WEB APPLICATIONS

- Broken authentication
- Session fixation
- XSS
- CSRF
- Injections
- Sensitive data exposure

BROKEN AUTHENTICATION

- Esempio #1: Credential stuffing

Utilizzo di una lista di password note, è un attacco comune.

- Esempio #2: Application session timeouts

Un utente utilizza un computer pubblico. Piuttosto che utilizzare il logout l'utente chiude la tab e va via. Un attaccante usa lo stesso browser un'ora dopo e trova una sessione aperta.

- Esempio #3: Passwords are not properly hashed and salted

Password esposte in chiaro (es. Database)

SESSION FIXATION

Se presente, permette ad un attaccante di impersonare un utente valido utilizzando un session ID generato in precedenza. Questa vulnerabilità si ha se durante il processo di autenticazione l'applicazione web non genera un session ID univoco.

Attacker -> Victim: Go to
<http://innocentsite.org/login?PHPSESSID=0123456789ABCDEF>

Victim -> innocentsite: Logs in with her credentials, using the session ID in the query

Attacker -> innocentsite: My session ID is 0123456789ABCDEF, please transfer all funds to account X

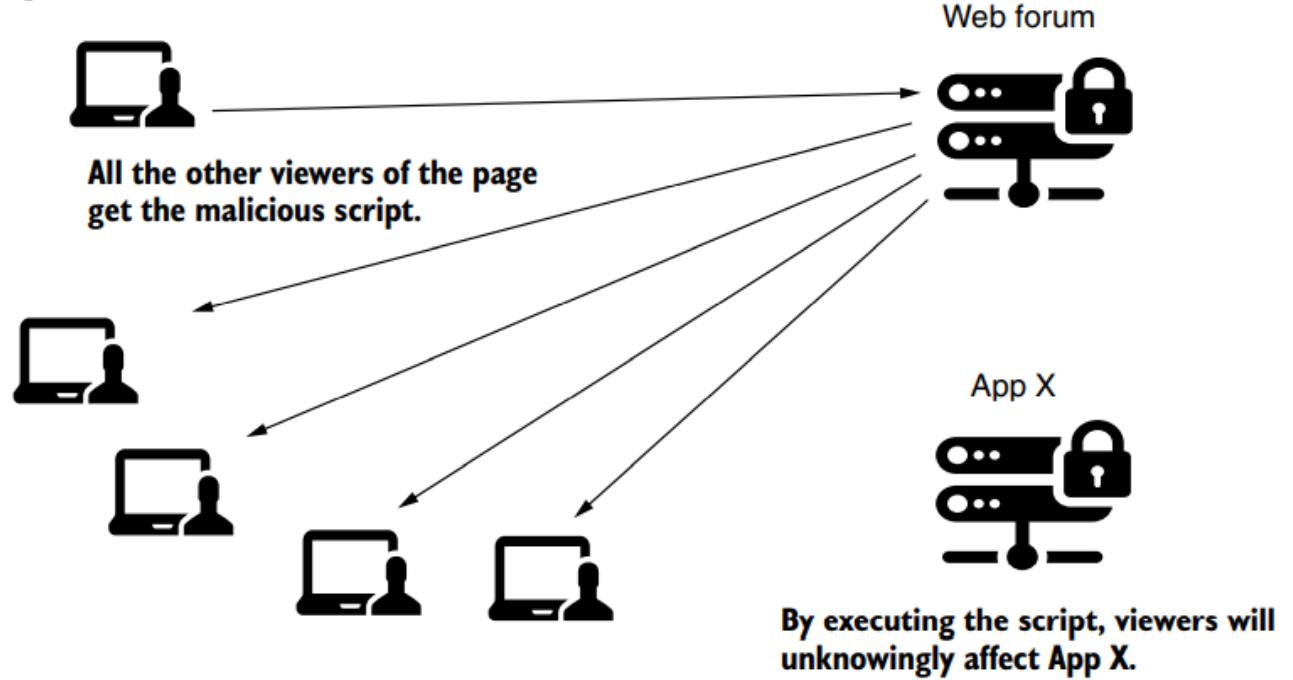
XSS - CROSS-SITE SCRIPTING 1 DI 2

- Iniezione di script client-side all'interno dei servizi esposti da un server

A hacker adds a comment containing a malicious script:

```
<script>  
@#$% Post a lot of data to App X @#$%  
</script>
```

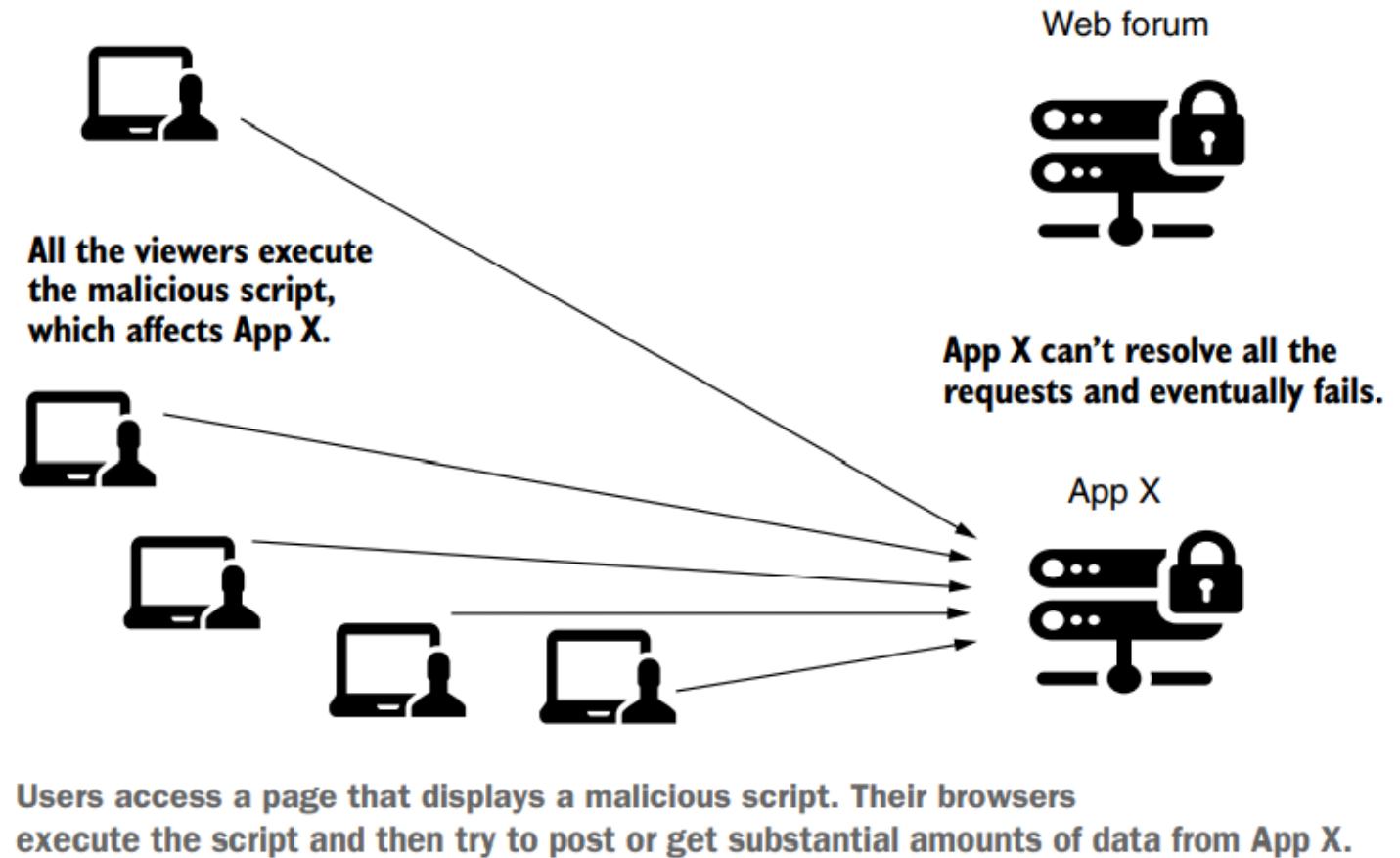
The app does not check the request. It stores it and returns it to be displayed as is.



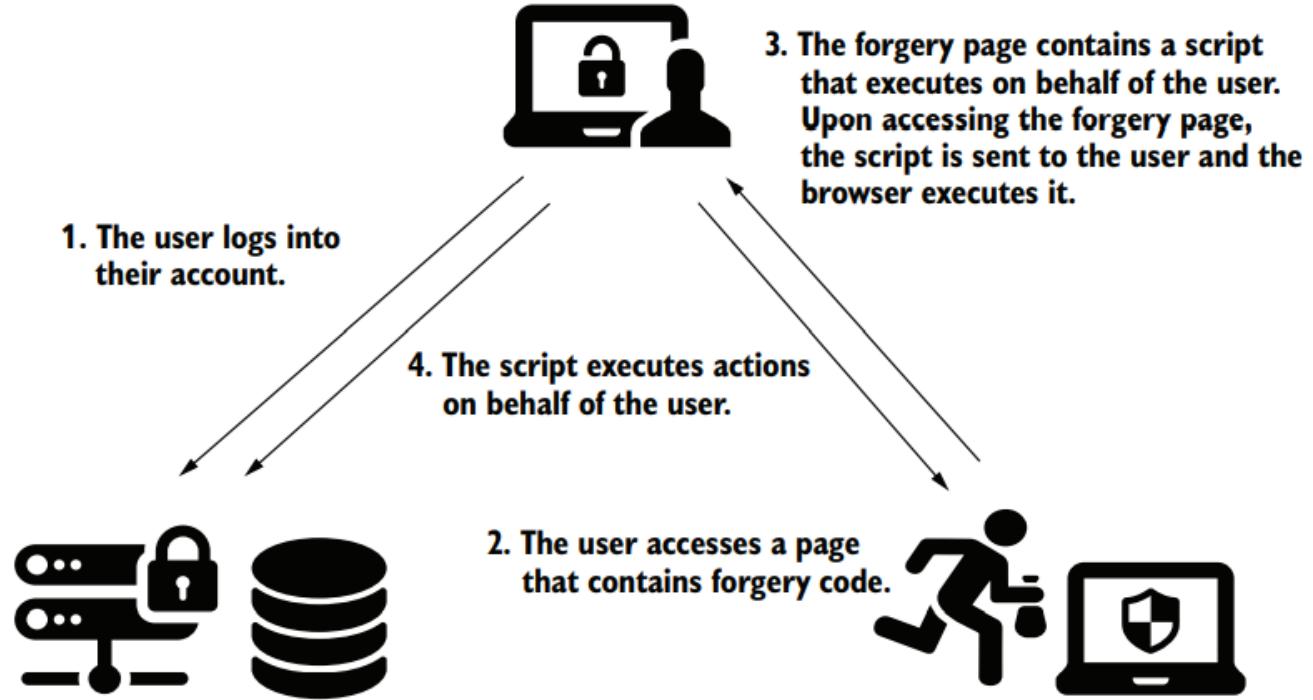
A user posts a comment containing a script, on a web forum. The user defines the script such that it makes requests that try to post or get massive amounts of data from another application (App X), which represents the victim of the attack. If the web forum app allows cross-site scripting (XSS), all the users who display the page with the malicious comment receive it as it is.

XSS - CROSS-SITE SCRIPTING 2 DI 2

- Iniezione di script client-side all'interno dei servizi esposti da un server



CSRF - CROSS- SITE REQUEST FORGERY



Steps of a cross-site request forgery (CSRF). After logging into their account, the user accesses a page that contains forgery code. The malicious code then executes actions on behalf of the unsuspecting user.

Il **Cross-site request forgery**, abbreviato **CSRF** o anche **XSRF**, è una vulnerabilità a cui sono esposti i siti web dinamici quando sono progettati per ricevere richieste da un client senza meccanismi per controllare se la richiesta sia stata inviata intenzionalmente oppure no. Diversamente dal [cross-site scripting](#) (XSS), che sfrutta la fiducia di un utente in un particolare sito, il CSRF sfrutta la fiducia di un sito nel browser di un utente.

INJECTION VULNERABILITIES IN WEB APPLICATIONS

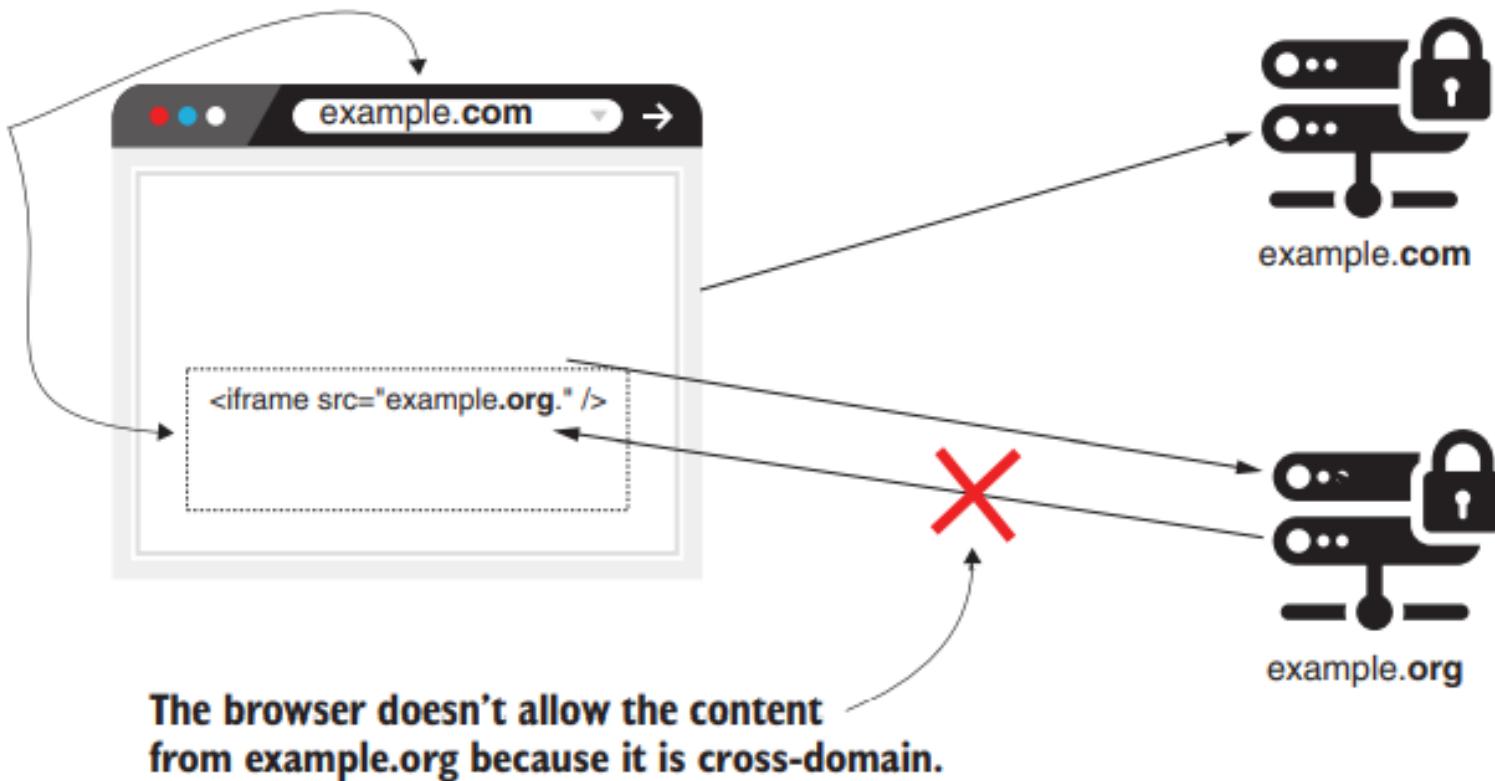
- In un injection attack, l'attaccante sfrutta una vulnerabilità del sistema introducendo dati nel sistema stesso(script malevoli, ad esempio, come visto in un attacco XSS).
- Ne esistono altri: SQL injection, OS command injection, LDAP injection ...

EXPOSURE OF SENSITIVE DATA

- Gli sviluppatori apprendono continuamente da esempi e tutorial.
- Generalmente, gli esempi sono semplificati per spiegare un concetto al lettore.
- Gli sviluppatori possono credere, erroneamente, che ogni cosa letta sia una buona pratica
- Relativamente a Spring Security, nell'application.properties non dobbiamo inserire direttamente password, o chiavi di cifratura. Le password all'interno di un Database devono essere codificate. Anche i log devono essere trattati in modo tale da evitare l'esposizione di dati sensibili.

C O R S

You have opened example.com, but the page uses an <iframe /> to nest content from example.org.



C O R S

Il meccanismo CORS consente di specificare quali domini possono essere acceduti dall'applicazione corrente. I meccanismi più importanti, basati su header HTTP, sono i seguenti:

- **Access-Control-Allow-Origin**—Specifica i domini esterni (origins) ai quali è possibile accedere dal dominio applicativo.
- **Access-Control-Allow-Methods**—Specifica quali richieste HTTP sono consentite verso un dominio esterno.
- **Access-Control-Allow-Headers**—Aggiunge limitazione su quali headers possono essere aggiunti in una richiesta HTTP.

C O R S

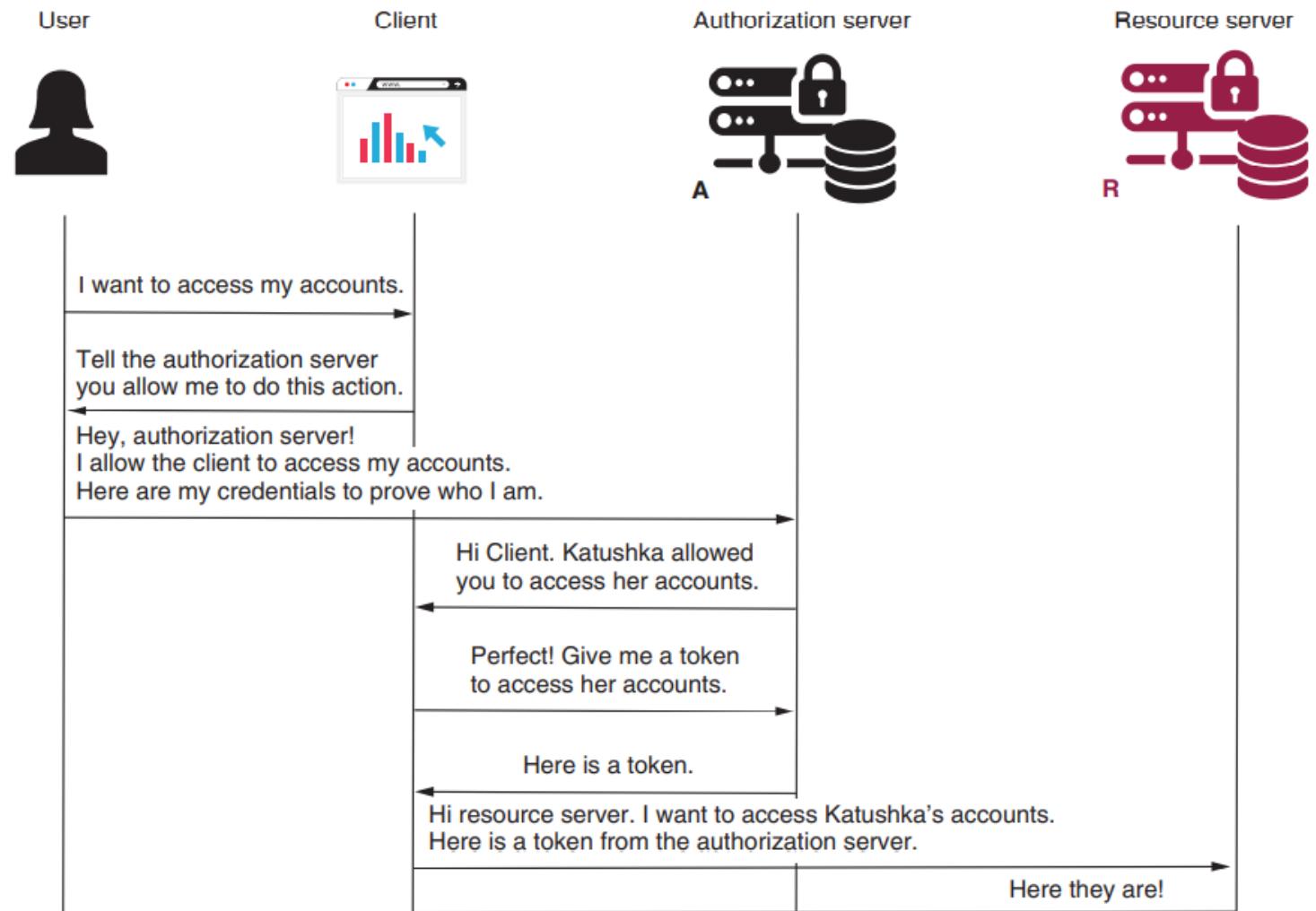
```
@PostMapping("/cors")
@ResponseBody
@CrossOrigin(origins="http://localhost")
public String test() {
    return "http://localhost is now allowed";
}
```

- Attenzione! Non abilitare mai tutte le origini!
- Rischio di esposizione dell'applicazione ad attacchi **DDoS**

CORS, CONFIGURAZIONE

```
http.cors(c -> {  
    CorsConfigurationSource source = request -> {  
        CorsConfiguration config = new CorsConfiguration();  
        config.setAllowedOrigins(List.of("http://localhost"));  
        config.setAllowedMethods(List.of("GET", "POST"));  
        return config;  
    };  
    c.configurationSource(source);  
});
```

OAUTH 2



OAUTH 2 - TIPI DI FLUSSO UTILIZZATI DALL'APPLICAZIONE CLIENT

- AUTHORIZATION CODE GRANT TYPE
- PASSWORD GRANT TYPE o RESOURCE OWNER CREDENTIALS GRANT TYPE
- CLIENT CREDENTIALS GRANT TYPE

OAUTH 2 - STEP 1

AUTHORIZATION CODE GRANT TYPE

- Richiesta di autenticazione dell'utente verso l'applicazione client
- Il client ridirige l'utente verso un endpoint dell'authorization server per l'autenticazione.

NOTA: L'utente interagisce direttamente con l'authorization server. L'utente non invia credenziali all'applicazione client.

OAUTH 2 - STEP 1

AUTHORIZATION CODE GRANT TYPE

Quando il client ridirige l'utente verso l'authorization server, tecnicamente invia una richiesta all'authorization server con i seguenti dettagli:

- **response_type**: con il valore `code`, per dire all'authorization server che il client aspetta un codice.
Il client ha bisogno del codice per ottenere un `access token`.
- **client_id**: un ID che identifica l'applicazione.
- **redirect_uri**: l'url al quale, l'authorization server, ridirige il client dopo un'autenticazione con successo.
- **scope**: simile alle granted authorities.
- **state**: token utilizzato per la CSRF protection.

OAUTH 2 - STEP 1

AUTHORIZATION CODE GRANT TYPE

- L'authorization server genera il primo **codice** come prova che l'utente ha interagito con lui. Il client riceve questo codice e deve usarlo con le sue credenziali per ottenere il token di accesso.
- Il client ottenuto il token accede alle risorse. Il codice prova che l'utente ha autorizzato l'operazione.

OAUTH 2 - STEP 2

AUTHORIZATION CODE GRANT TYPE

Il client adesso effettua una richiesta all'authorization server inviando:

code: codice di autorizzazione ricevuto allo step 1.

client_id and client_secret: credenziali applicazione client.

redirect_uri: lo stesso dello step 1.

grant_type: con il valore **authorization_code**, che identifica il tipo di flusso usato.

Come risposta, il server invia il token per l'accesso.

OAUTH 2 - STEP 3

AUTHORIZATION CODE GRANT TYPE

Dopo aver ottenuto con successo l'access token dall'authorization server, il client può accedere alle risorse protette. Il client utilizza l'access token nell'authorization request header* quando richiama l'endpoint.

*(Un'alternativa è l'invio attraverso Cookie protetto)

OAUTH 2 - STEP 1

PASSWORD GRANT TYPE

L'applicazione client utilizza le credenziali utente per autenticarsi ed ottenere l'access token dall'authorization server. *Si può utilizzare questo flusso solo se client e authorization server sono mantenuti dalla stessa organizzazione.*

Con il **password grant type**, è l'applicazione che presenta il form di login , e acquisisce le credenziali utente.

NOTA:Il password grant type è meno sicuro dell'**authorization code grant type**, principalmente per il fatto che le credenziali utente devono essere condivise con l'applicazione client.

OAUTH 2 - STEP 1

PASSWORD GRANT TYPE

L'applicazione client ottiene le credenziali utente e richiama l'authorization server per ottenere il token di accesso:

grant_type: con il valore **password**.

client_id and **client_secret**: credenziali applicative.

scope: Le granted authorities, username e password dell'utente.

OAUTH 2 - STEP 2

PASSWORD GRANT TYPE

Una volta che il client ha l'access token, lo utilizza per l'accesso agli endpoints sul resource server, esattamente come nell' **authorization code grant type**.

OAUTH 2 - STEP 1

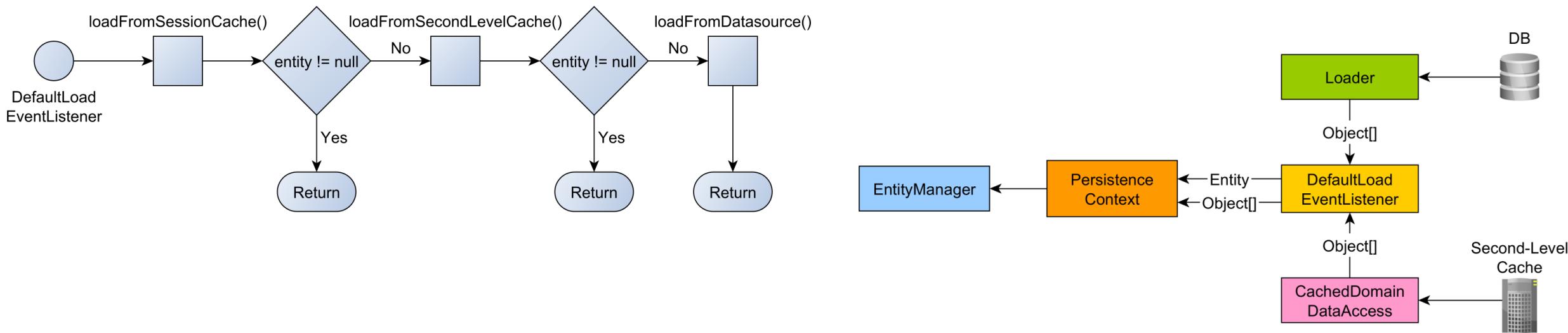
CLIENT CREDENTIALS GRANT TYPE

- Il più semplice tipo di grant descritto in OAuth 2. Si utilizza quando non è coinvolto nessun utente: autenticazione tra due applicazioni.
- Per ottenere l'access token, il client invia una request all'authorization server:
 - `grant_type`: con il valore `client_credentials`.
 - `client_id` and `client_secret`: le credenziali applicative.
 - `scope`: granted authorities.

In risposta, il client riceve l'access token.

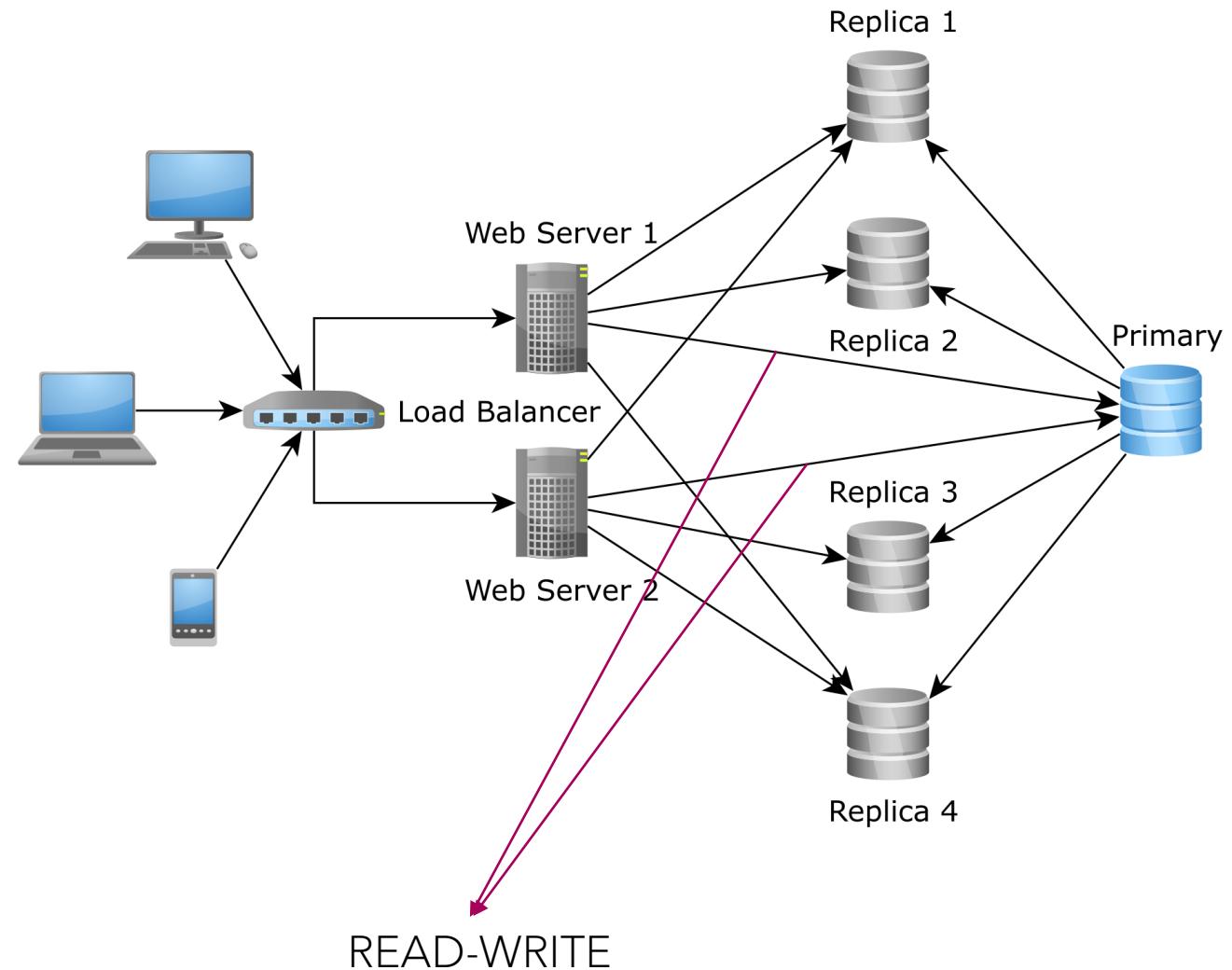
ORM CACHE

- JPA and Hibernate forniscono una cache di primo livello. La first-level cache è legata al corrente Thread di esecuzione, quindi l'entity nella cache di primo livello non è condivisa tra thread.
- Per servire una molteplicità di richieste concorrenti è fornita una second-level cache che incrementa quindi la probabilità di avere un hit nella cache.



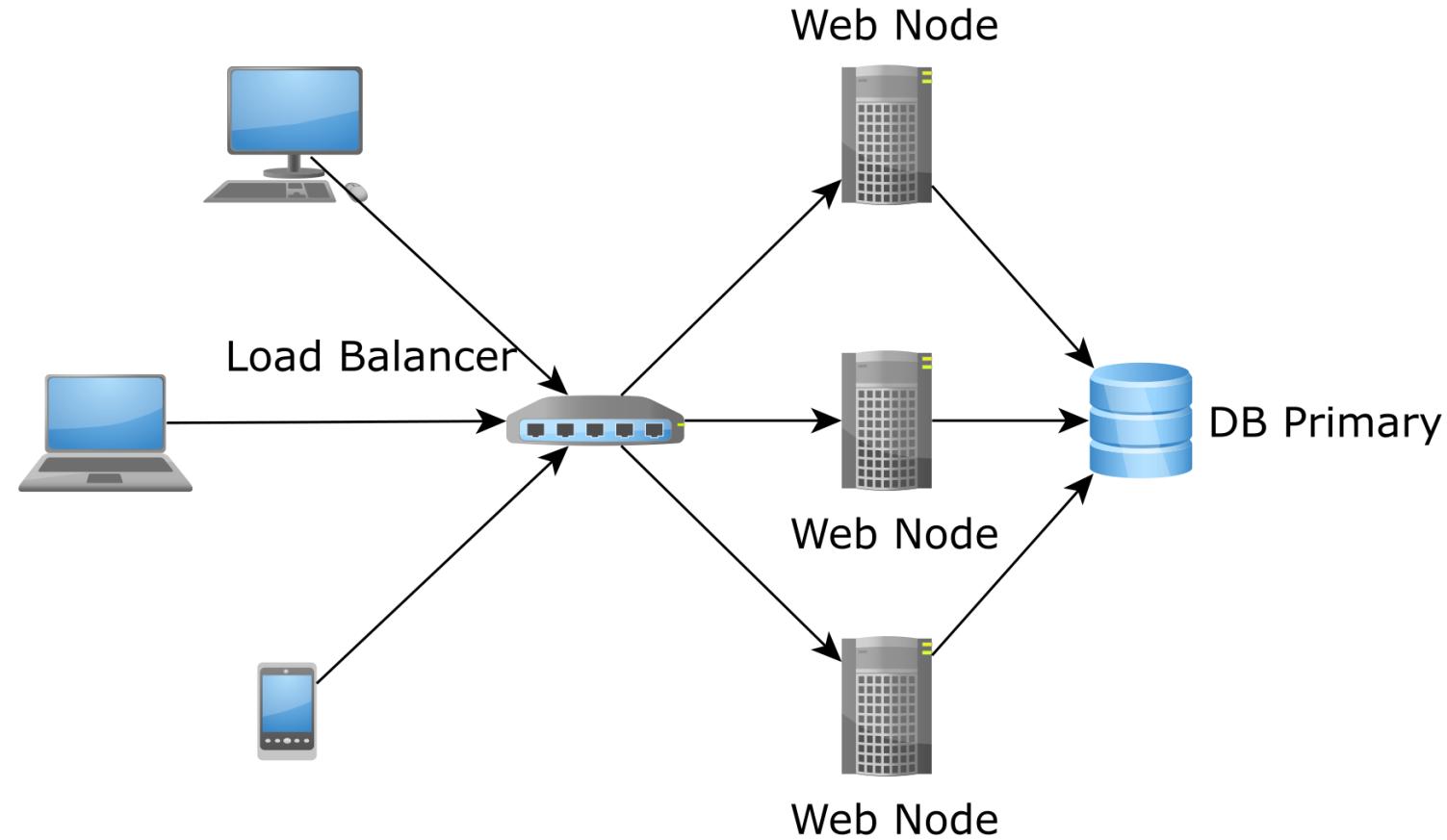
ORM CACHE

- Lo scaling delle read-only transactions può essere realizzato facilmente con i replica nodes. Mentre un nodo primario può essere solo scalato verticalmente.
- La second-level cache è utile proprio per un nodo master. Per le read-write database transactions che devono essere eseguite sul nodo master, la second-level cache può ridurre il carico di query servendole dalla cache di secondo livello di cui viene mantenuta la consistenza.



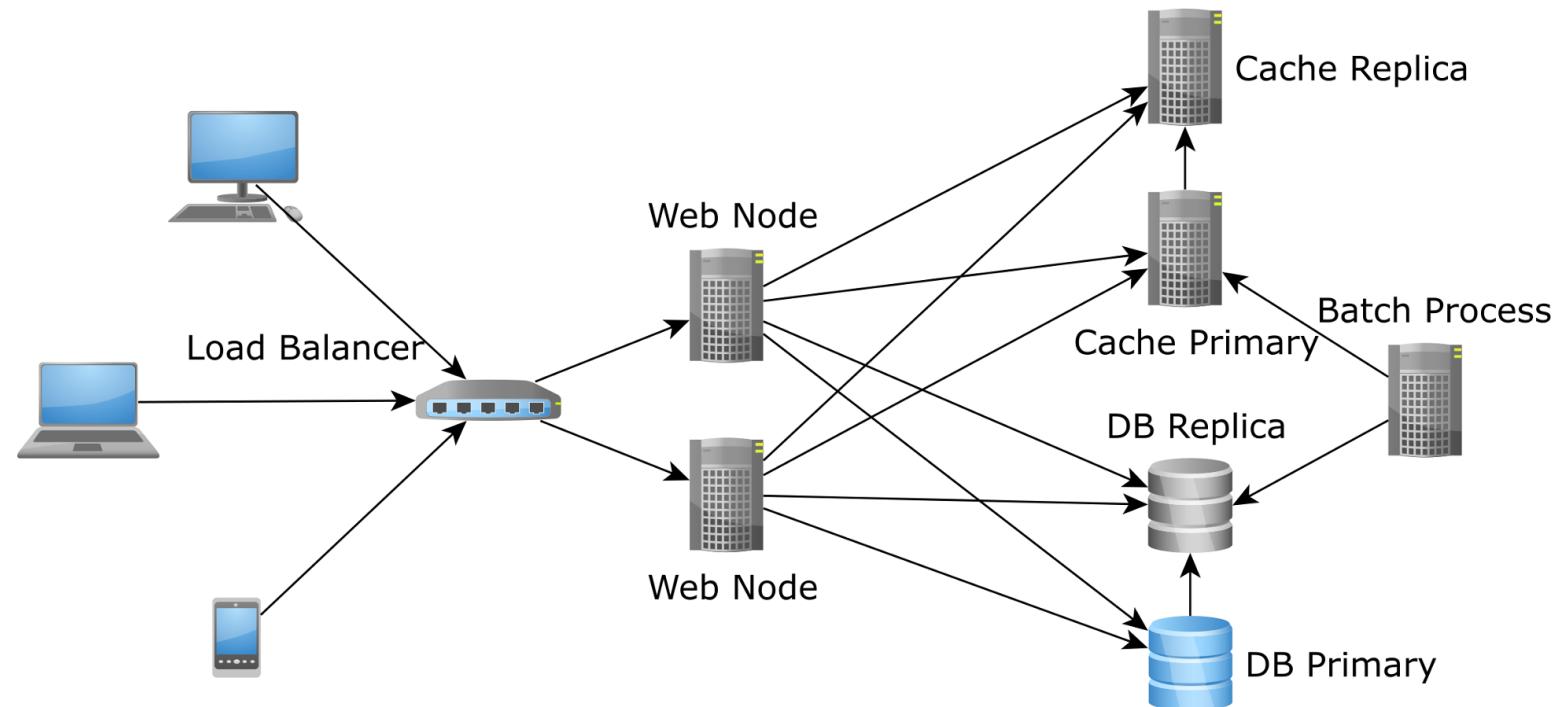
JPA/HIBERNATE CACHE SCALING

- In genere la second-level cache è memorizzata in memoria applicativa e questo è problematico per differenti ragioni.
- La memoria applicativa è limitata, il volume di dati che può essere messo in cache è quindi limitato.
- Quando il traffic aumenta e vogliamo avviare un nuovo nodo, il nuovo nodo inizia con una cache vuota, peggiorando il problema per via del fatto che il database deve essere caricato di query per popolare la cache.

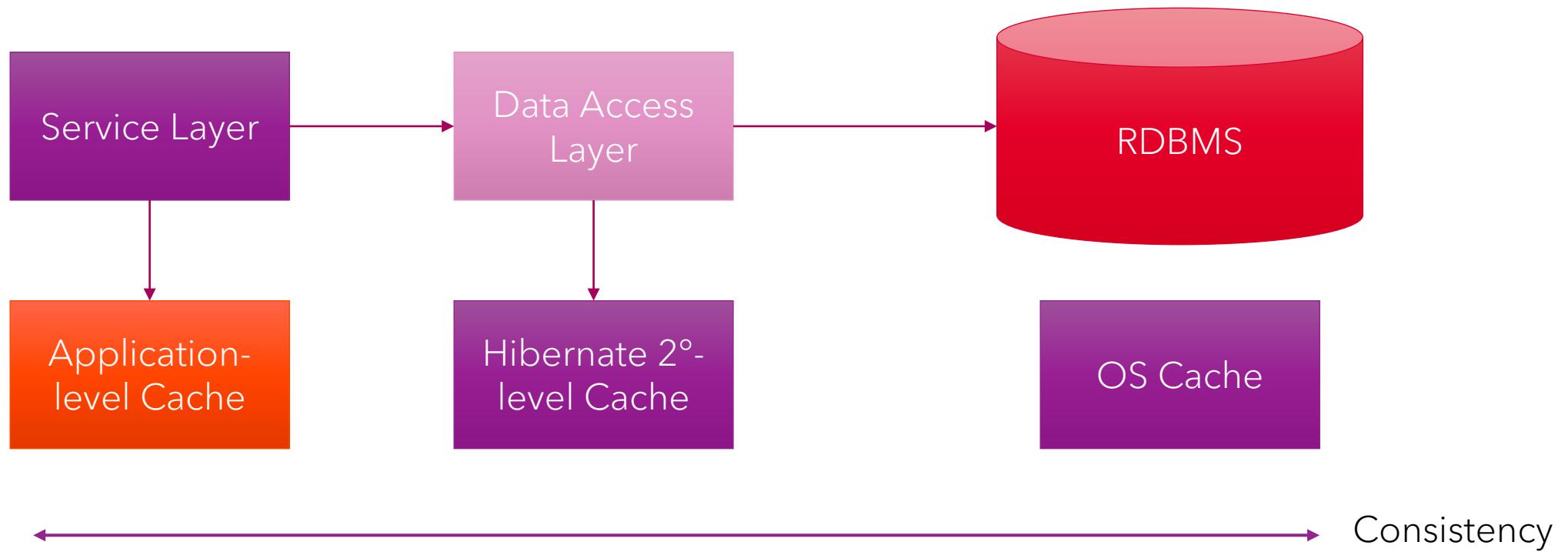


JPA/HIBERNATE CACHE SCALING

- Per gestire questa situazione, è meglio avere una cache distribuita (es. Redis). In questo modo, la quantità di dati in cache non è limitata dalla dimensione della memoria su un nodo singolo dal momento che lo sharding può essere utilizzato per splittare i dati su diversi nodi.
- Quando un nuovo nodo è aggiunto, i dati vengono caricati dalla cache.



ENTERPRISE CACHING LAYERS

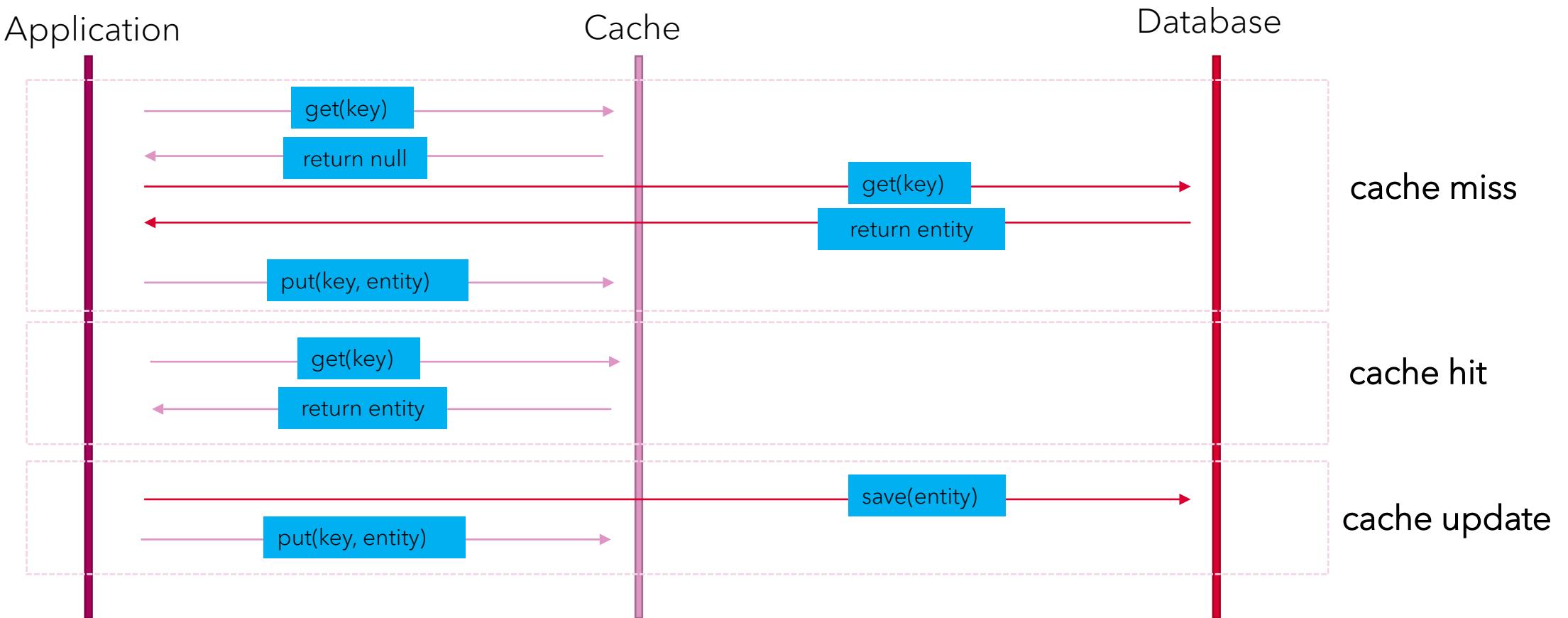


APPLICATION CACHE E DATA ACCESS CACHE

- Un'application cache è molto spesso una cache in cui i dati sono memorizzati in modalità chiave-valore. Una volta che i dati sono aggregati dal database, possono essere memorizzati in cache e servire le successive richieste proprio da questa cache. Utile per:
 - Database down per manutenzione. La cache consente di tenere in esecuzione l'applicazione se una quantità sufficiente di dati è stata caricata dal database.
 - Un'application-level cache è utile per scenari di sola lettura.
- La second-level cache offerta da Hibernate gestisce i problemi di consistenza legati ad una'application cache.
- Hibernate second-level cache è una soluzione data access caching il cui scopo è ridurre il carico sul database legato al fetching.

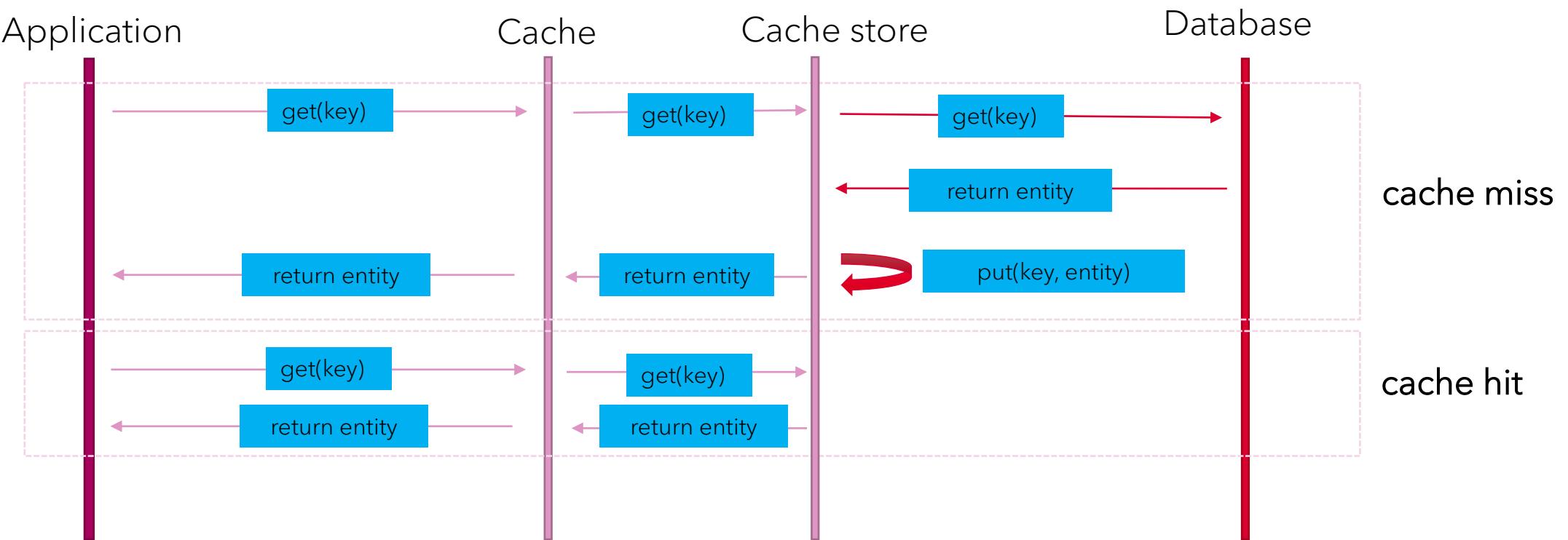
STRATEGIE DI SINCRONIZZAZIONE DELLA CACHE 1 DI 5

- Cache-aside



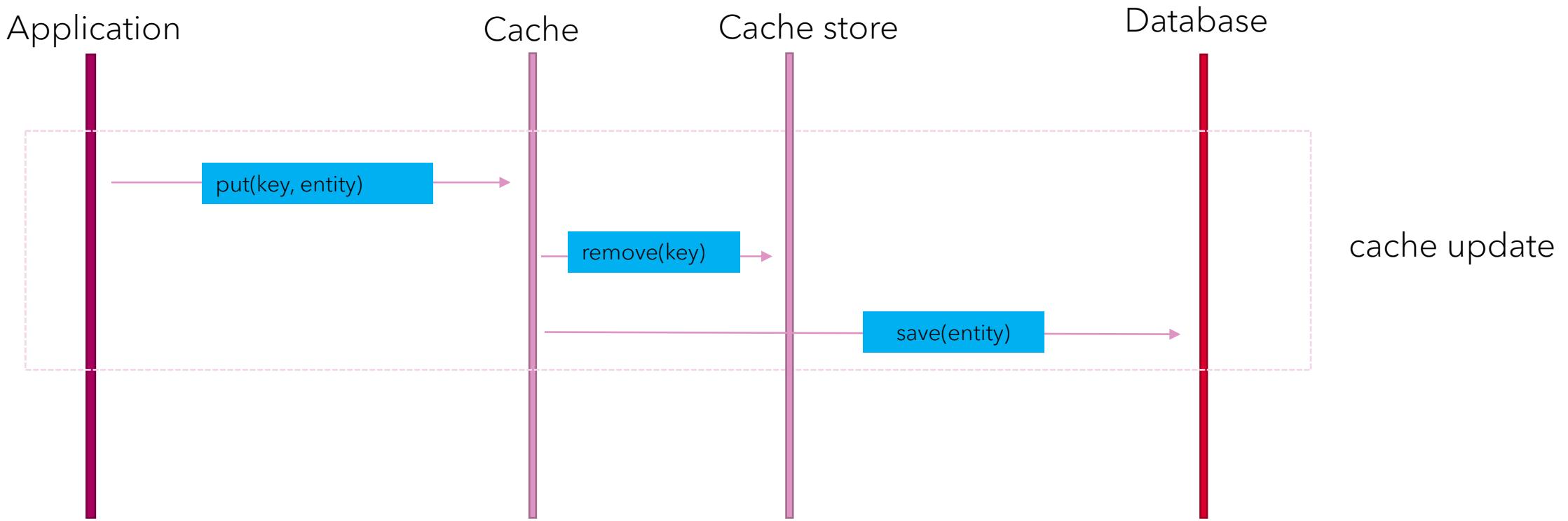
STRATEGIE DI SINCRONIZZAZIONE DELLA CACHE 2 DI 5

- Read-through: l'applicazione interagisce solo con il sistema legato alla cache



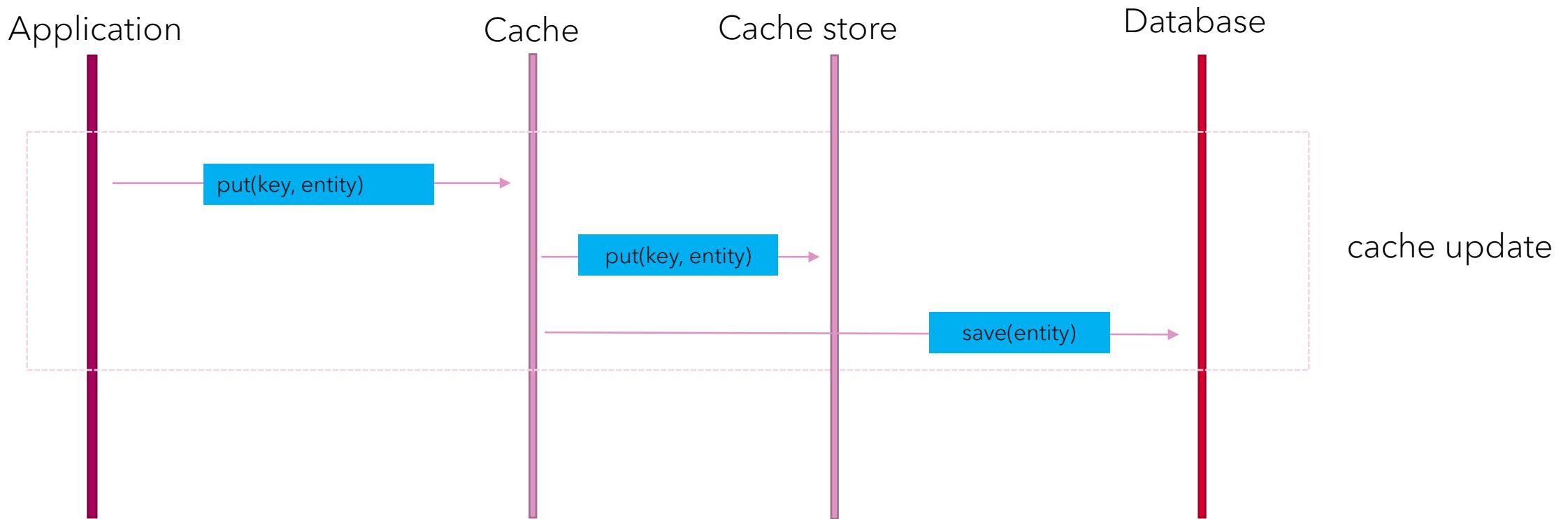
STRATEGIE DI SINCRONIZZAZIONE DELLA CACHE 3 DI 5

- Write-invalidate: se l'entity cambia la cache propaga il cambiamento al database e l'entity viene rimossa dalla cache



STRATEGIE DI SINCRONIZZAZIONE DELLA CACHE 4 DI 5

- Write-through: se l'entity cambia la cache propaga il cambiamento al database e alla cache



CACHE SYNCHRONIZATION PATTERNS

- Mantenere copie di dati (cache e database) è un compito non di facile gestione
- Database e cache devono essere mantenuti sincronizzati
- Non tutti gli applicativi necessitano dello stesso livello di consistenza
 - Un'applicazione real-time potrebbe non voler nessuna cache attiva
 - Un'applicazione i cui dati vengono aggiornati poco frequentemente potrebbe voler beneficiare di una cache
 - Una cache può migliorare il site page rank perché velocizza i tempi di risposta

CACHE SYNCHRONIZATION PATTERNS: SINCRONO, ASINCRONO, CATTURA DEL CAMBIAMENTO

- Se si utilizza il cache-aside la business-logic deve aggiornare il database e tutte le entry in cache (sincrono)
- I cambiamenti lato database possono venire notificati alla cache (asincrono). La business logic non deve gestire la cache.
- Esistono diverse tecniche che agiscono sul cambiamento del dato per aggiornare la cache. La più efficace consiste nell'uso di un framework che parsa il log delle transazioni.

SECOND LEVEL CACHING

- La cache di secondo livello:
 - Non richiede nessun cambiamento al livello di accesso ai dati
 - Offre le strategie read-through e write-through per l'aggiornamento della cache
 - Salva i dati in un formato raw molto vicino al database
 - Non è una sostituzione della cache applicativa
 - Aiuta a ridurre il tempo di risposta di transazioni read-write
 - Hibernate ne fornisce solo la specifica del contratto, l'implementazione è fornita da una terza parte: Infinispan, Ehcache, Hazelcast ...

SPRING CACHE

- Application cache.
- Buona soluzione per transazioni read-only.
- La **Spring Boot Cache Abstraction** richiede delle dipendenze.
- Queste dipendenze si ottengono con lo `spring-boot-starter-cache`.

```
<dependency>
```

```
    <groupId>org.springframework.boot</groupId>
```

```
    <artifactId>spring-boot-starter-cache</artifactId>
```

```
    <version>2.5.0</version>
```

```
</dependency>
```

SPRING CACHE

- Application cache.
- Buona soluzione per transazioni read-only.
- La **Spring Boot Cache Abstraction** richiede delle dipendenze.
- Queste dipendenze si ottengono con lo `spring-boot-starter-cache`.

`<dependency>`

`<groupId>org.springframework.boot</groupId>`

`<artifactId>spring-boot-starter-cache</artifactId>`

`<version>2.5.0</version>`

`</dependency>`

SPRING CACHE

- Per implementare la logica di cache caching logic si utilizzano le **caching-related annotations**.
- Per abilitare la cache, utilizziamo l'annotation `@EnableCaching` su una classe con `@Configuration` oppure sulla classe con `@SpringBootApplication`.

```
@SpringBootApplication  
@EnableCaching  
public class SpringBootCachingApplication {  
    public static void main(String[] args) {  
        SpringApplication.run(SpringBootCachingApplication.class, args);  
    }  
}
```

SPRING CACHE

- `@EnableCaching` automaticamente configura un `CacheManager`.
- In particolare questa annotation ricerca un cache engine da configurare se non lo trova configura una `ConcurrentMapCacheManager`, la quale fornisce un'implementazione di default in memoria basata su un oggetto `ConcurrentHashMap`.

SPRING CACHE: @CACHEABLE

- Ogni volta che un metodo sul quale è applicata l'annotation `@Cacheable` è richiamata, il comportamento di caching è attivato.
- In particolare, Spring Boot controller se il metodo è già stato richiamato con i parametri inseriti. Questo significa ricercare una chiave generata in base ai parametri. Se nessuna entry è individuata in base alla chiave calcolata, il metodo viene eseguito normalmente.
- Altrimenti, il risultato è ritornato dalla cache.

- If no params are given, return 0.
- If only one param is given, return that instance.
- If more than one param is given, return a key computed from the hashes of all parameters

```
@Cacheable("authors")
```

```
public List<Author> getAuthors(List<Int> ids) { ... }
```

SPRING CACHE: @CACHEABLE ALGORITMO DI GENERAZIONE

- Algoritmo di generazione della chiave:
 - If no params are given, return 0.
 - If only one param is given, return that instance.
 - If more than one param is given, return a key computed from the hashes of all parameters
- This approach works well for objects with natural keys as long as the hashCode() reflects that. If that is not the case then for distributed or persistent environments, the strategy needs to be changed as the objects hashCode is not preserved. In fact, depending on the JVM implementation or running conditions, the same hashCode can be reused for different objects, in the same VM instance.
- To provide a different default key generator, one needs to implement the org.springframework.cache.KeyGenerator interface. Once configured, the generator will be used for each declaration that does not specify its own key generation strategy

SPRING CACHE: @CACHEABLE, CHIAVE CUSTOM

```
public class CustomKeyGenerator implements KeyGenerator {  
    public Object generate(Object target, Method method, Object...params) {  
        return target.getClass().getSimpleName() + "_" + method.getName() + "_" +  
            StringUtils.arrayToDelimitedString(params, "_");  
    }  
}  
  
@Configuration  
  
public class ApplicationConfig extends CachingConfigurerSupport {  
    @Bean("customKeyGenerator")  
    public KeyGenerator keyGenerator() {  
        return new CustomKeyGenerator();  
    }  
}  
  
    @Override  
    @Cacheable("products", keyGenerator = "customKeyGenerator")  
    public List < Product > getProducts() {}
```

SPRING CACHE: @CACHEABLE

Possiamo specificare come deve essere generata la chiave:

```
@Cacheable(value="book", key="#isbn")
```

```
public Book findBookByISBN(String isbn) { ... }
```

```
@Cacheable(value="books", key="#author.id")
```

```
public Books findBooksByAuthor(Author author) { ... }
```

SPRING CACHE: @CACHEABLE

E' possibile abilitare il conditional caching. Es. In cache soltanto gli autori il cui nome è inferiore ai 15 caratteri:

```
@Cacheable(value="authors", condition="#fullName.length < 15")
```

SPRING CACHE: @CACHEPUT

- Questa annotation a livello di metodo, va utilizzata se vogliamo aggiornare (put) la cache senza evitare che il metodo venga eseguito.
- Il metodo verrà sempre eseguito – in accordo con le @CachePut options – ed il suo risultato sarà messo in cache.
- La principale differenza tra @Cacheable and @CachePut è che il primo può evitare l'invocazione del metodo mentre il secondo no.
- Il metodo inserisce il risultato in cache, anche se esiste una chiave associata con i parametri.

SPRING CACHE: @CACHEEVICT

- Questa annotation consente di rimuovere(evict) dati dalla cache. Annotando un metodo con @CacheEvict specifichiamo di rimuovere uno o più valori dalla cache. Se vogliamo rimuovere uno valore particolare dobbiamo passare la chiave:

```
@CacheEvict(value="authors", key="#authorId")  
public void evictSingleAuthor(int authorId) { ... }
```

Se vogliamo rimuovere tutti gli elementi da una cache:

```
@CacheEvict(value="authors", allEntries=true)  
public String evictAllAuthorsCached() { ... }
```

SPRING CACHE: @CACHEEVICT

Questa annotation è estremamente importante perchè le dimensioni sono un problema per la cache. Una possibile soluzione alla memoria richiesta dalla cache è la **compressione**.

D'altra parte, il miglior approccio sarebbe quello di evitare di mantenere in cache dati non utilizzati di frequente. La cache cresce rapidamente, occorre aggiornare dati vecchi con @CachePut e rimuovere quelli non utilizzati con @CacheEvict.

SPRING CACHE: COMPRESSIONE

- Ottimizzare l'uso della Ram è importante in generale ed in particolare quando la cache utilizza la stessa memoria applicativa.
- Normalmente, con la cache in Spring Boot, i dati sono serializzati e memorizzati nella cache. Quando necessario i dati sono ripresi dalla cache, deserializzati, e ritornati all'applicazione.
- Aggiungere un layer di compressione significa comprimere i dati prima della serializzazione decomprimerli prima della deserializzazione.

Esempio:

<https://betterprogramming.pub/how-to-add-compression-to-caching-in-spring-boot-d4d21533167c>

SPRING CACHE: @CACHECONFIG

- Consente di configurare una cache in un punto evitando di ripetere la configurazione più volte:

```
@CacheConfig(cacheNames={"authors"})  
public class AuthorDAO {  
    @Cacheable  
    publicList<Author> findAuthorsByFullName(String fullName) { ... }  
    @Cacheable  
    public List<Author> findAuthorsByBook(Book book) { ... }  
}
```



Posteitaliane

Poste Italiane

Si prega di selezionare il proprio dominio di logon

Accedi con

Se l'accesso viene effettuato da **postazione aziendale**, il collegamento alla piattaforma sarà diretto senza utilizzo di credenziali

Se l'accesso viene effettuato da **internet**, dovranno essere inserite le credenziali di accesso alla rete aziendale.



Si accede al titolo del corso a cui si sta partecipando; impostare il filtro «Visualizza per» in «tutte le attività»

CORSI ASSEGNATI

Accedi ai corsi assegnati

Non avviata
Valutazione
Qualità percepita

Valuta

Si potrà compilare il questionario di qualità

Si dovrà cliccare su per visualizzare e scaricare il materiale didattico

Materiali correlati