RipassoApplicazioni ad Eventi

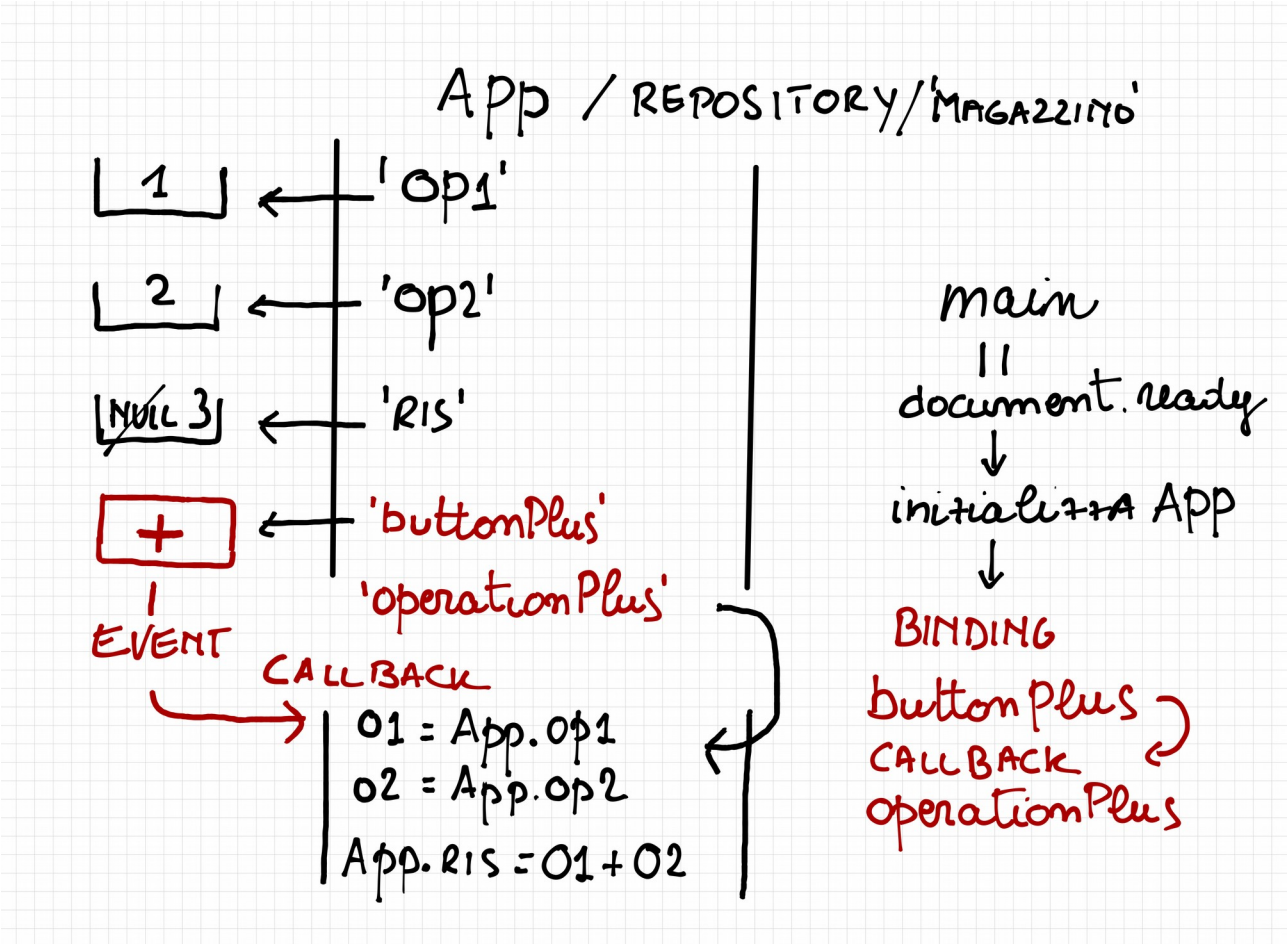
Una Applicazione ad eventi è costituita da

- Un dizionario degli oggetti
- Una serie di funzioni CallBack che accedono al dizionario
- un main()
- Inizializza il dizionario
- Esegue il binding fra Eventi(generatori di eventi) e CallBack

In Javascript

```
var APP = {                                  // Dizionario Applicazione
     "op1" = 0,
     "op2" = 0,
     "ris"  = null,
     "operazionePlus" = function () { …  // CallBack/Listener
       o1=APP.op1
       o2=APP.op2
       APP.ris = o1 + o2
      }
     "InitOPerazionePlus" = function() {
            Binding EventoTastoPiu con Callback operazionePlus
      }
}

document.ready() {                           // main()
      // Crea oggetti della APP
      APP.op1 = 1                  // Valori di default
      APP.op2 = 2
      ….
      // Attiva il Binding di tutti gli eventi
      APP. InitOPerazionePlus()
      ...
}
```



Parole Chiave:

Dizionario (Alias Repository, Application Context … *Magazzino*)
Callback (funzioni che accedono al Dizionario => lo vedono .. scope)
Evento (Generatore di Evento)
Main (Inizializza il Dizionario ed esegue il Binding)
Binding (azione di collegamento Evento e sua CallBack)

Applicazioni Web

Asserzione:
Una applicazione Web è una applicazione ad eventi MCV con Application Context (Repository di tutte le istanze degli oggetti necessari ai listener)

Dimostreremo l'Asserzione per il Framework Spring collegando le cinque parole chiave di una App MCV con i concetti http e le annotazioni specifiche di Spring

Dizionario (Alias Repository, Application Context … *Magazzino*)

Dizionario in Spring si chiama ApplicationContext le istanze degli oggetti contenuti si chiamano Bean
L'accesso ai Bean è semplificato dal meccanismo di @autowired che si chiama Dependency injection

Callback (funzioni che accedono al Dizionario => lo vedono .. scope)

Le Callback vengono definite con una apposita annotazione @MapRequest che contestaulmente esegue anche il binding

Evento (Generatore di Evento)

Gli eventi sono le request Http provenienti da chiamate Ajax o form http e sono individuati da una precisa sottostringa dell'URL quella che indica la risorsa.

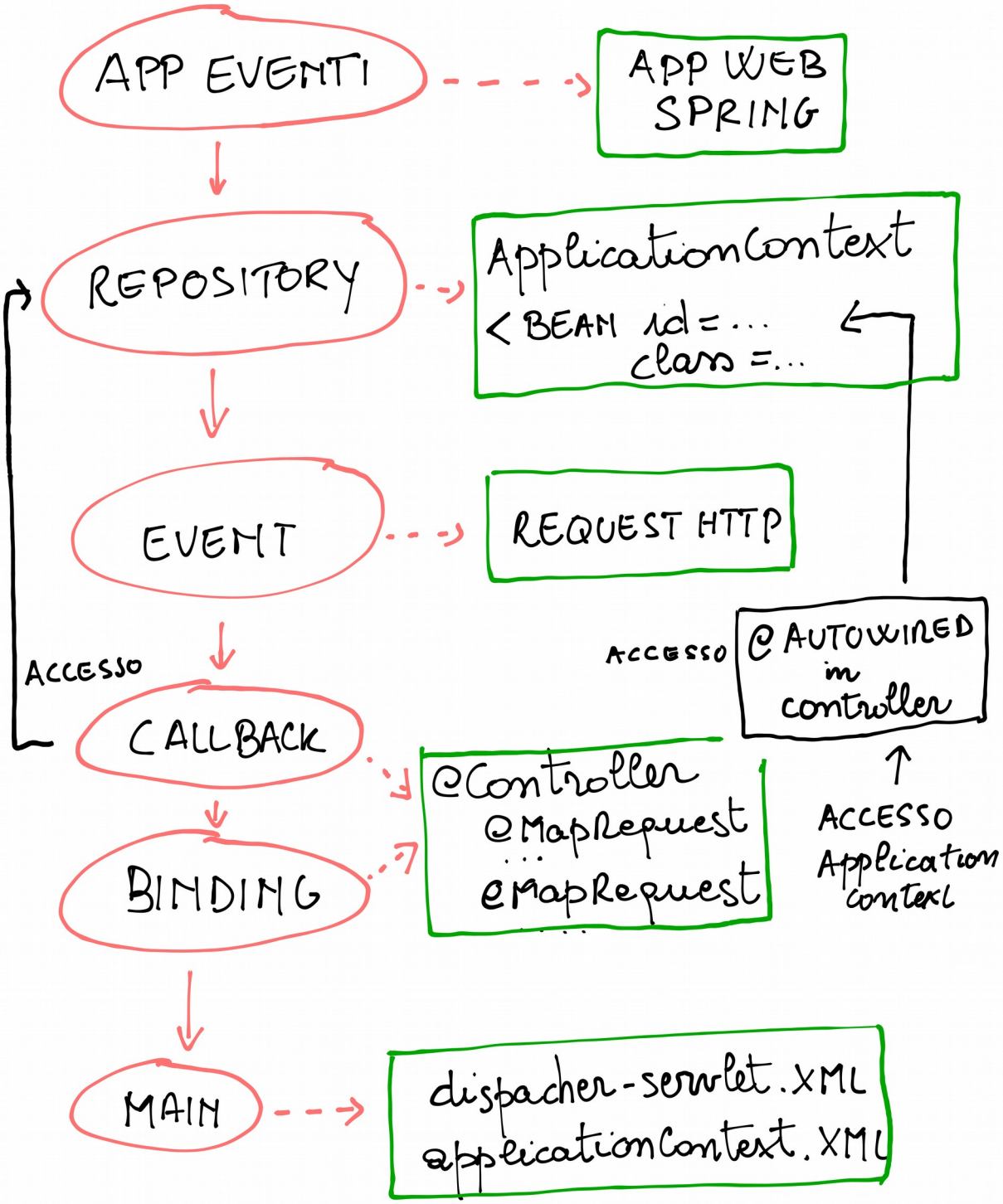
Main (Inizializza il Dizionario ed esegue il Binding)

Il main in Spring è una servlet predifinita HttpDispacherServlet che non deve essere editata perché prende le informazioni di inizializzazione dai due file XML:

applicationContext.xml
dispatcher-servlet.xml

Binding (azione di collegamento Evento e sua CallBack)

Il binding viene svolto contestualmente all'azione di definizione della Callback con @MapRequest


APP EVENTI
APP WEB SPRING
REPOSITORY
ApplicationContext
<BEAN id=...
class=...
EVENT
REQUEST HTTP
ACCESSO
@AUTOWIRED in controller
ACCESSO
CALLBACK
@Controller
@MapRequest
...
@MapRequest
ACCESSO Application context
BINDING
MAIN
dispacher-servlet.XML
applicationContext.XML

Framework Spring è una implementazione del DP MCV in un ambiente web
Generatore di eventi: La connesione TCP/IP di ingresso della servlet

Evento: una request di risorsa http che segue il nome dell'applicazione nel URL della request

Esempio:

GET http://localhost:8080/***webapplication2***/**NOMESERVIZIO.htm**?parametro=idItem

http://localhost:8080 => **Dominio**

webapplication2 **=> *nome applicazione (sul server Tomcat ci sono più applicazioni)***

NOMESERVIZIO.htm => Evento (nome della risorsa richiesta = funzione richiesta)

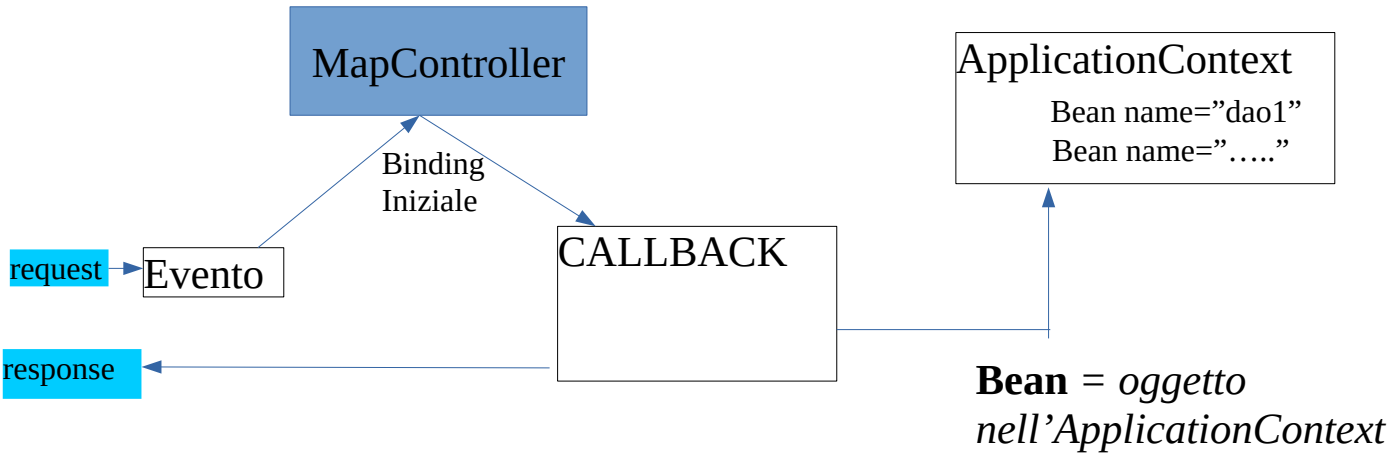
parametro=idItem => coppia nome_parametro=valore_parametro separati da &

Listener : ovvero una CallBack associata all'evento */NOMESERVIZIO.htm*

MapController: associa ad una requestHtml la CallBack che la deve gestire

Repository delle istanze degli oggetti: classe ApplicationContext in cui vengono automaticamente inizializzate le istanze dell'applicazione web (i Bean).

Output: response di una servlet ovvero un oggetto che sarà una pagina (stream) JSON, XML o HTTP secondo la dichiarazione del suo contenuto *content-type*



Content-type =>Json,Xml,Html

Automatismi di Spring

Inserimento Automatico nell'applicationContext

```
@Repository("dao1")
public class Dao {
```

nel nostro caso l'inserimento automatico si ottiene semplicemente annotando la classe con

@Repository("dao1")

il nome **@Repository** è un alias @**Service**("dao1")

Repository ci ricorda che ha che fare con la persistenza dei dati, il nome *Service* va ugualmente bene, ma meno informativo in questo caso.

La stringa **"dao1"** è il nome della chiave associata all'istanza di DAO nel dizionario.

La scrittura @**Repository** senza argomenti genera automaticamente la chiave purché ci sia una sola istanza della classe Dao e nel nostro caso si potrebbe fare anche così.

Automatismi di Spring

Binding automatico fra evento specificato nell'URL e il mapController

2 step

step 1: annotare la classe in cui è contenuto il mapController con l'annotazione generica **@Controller** in cui si specifica che contiene alcuni mapController

```
@Controller
public class  ControllerMultiMap {
```

step 2: annotare i metodi della classe da associare all'evento

```
// Un controller può avere più metodi di risposta a varie request
@RequestMapping(value = "/NOMESERVIZIO_02.htm",method = RequestMethod.GET,
        produces = "application/json;charset=UTF-8")
@ResponseBody
public String dao_PersonaFindByName(HttpServletRequest request ) {

    String valore = request.getParameter("nome");

    List<String> listaN = dao.personaDao.findByName(valore);

    String returnJson = renderJson_dao_PersonaFindByName( listaN);


    return ( returnJson );

}
```

@**RequestMapping**(**value** = "/NOMESERVIZIO_02.htm",
method = RequestMethod.GET,
produces = "application/json;charset=UTF-8")
@**ResponseBody**
public **String** msg00(HttpServletRequest request) { ….. }

*RequestMappin*g vuole l'**evento**, il **method http** usato per accedere e indica che **produces** in uscita nella response uno stream JSON *ResponseBody* indica che la String di return del mapController è il body della response (l' informazione sul formato della stringa è in produces …)

Automatismi di Spring

Accesso automatico alle istanze presenti nell'applicationContext tramite la dependecy injection

I mapController come tutti i listener devono accedere al repository (alias magazzino) delle istanze degli oggetti, ma in realtà usano un numero molto limitato di istanze e tutti i mapController di un Controller useranno più o meno gli stessi oggetti.

Spring mette a disposizione un automatismo che si chiama **Dependency Injection :** le variabili di istanza del Controller sono visibili da tutti i mapController e vengono inizializzate automaticamente con l'annotazione @autowired senza che il programmatore debba aggiungere una riga. L'inizializzazione può avvenire al momento della creazione dell'istanza oppure settando una opportuna opzione "lazy" nei file di configurazione l'inizializzazione può avvenire al primo uso della variabile.

```java
@Controller
public class  ControllerMultiMap {

    // Autowired => cerca un bean di classe Dao e lo assegna qui
    // nell'ipotesi che ce ne sia una solo
    // in presenza di piu' istanze autowired prevede una stringa di selezione
    // @Qualifier
    @Autowired
    @Qualifier("dao1")
    private Dao dao;

    // fine Dependency Injection
```

Autowired => cerca un bean di classe Dao e lo assegna al reference dao nell'ipotesi che ce ne sia una solo di classe Dao.
In presenza di piu' istanze autowired prevede una stringa di selezione in @Qualifier che è il nome della stringa usata in @Repository o in @Service per inserirlo automaticamente nell'applicationContext.

@Autowired
@Qualifier("dao1")
private Dao dao;

Automatismi in Spring

Come fanno a funzionare tutti questi automatismi?

Grazie a due file XML contenuti nella dir web.xml

1) File applicationContext.xml

Serve per inizializzare nell'application context bean che hanno classi di cui non abbiamo il sorgente (e neanche ci interessa …) e che quindi non possiamo annotare come @Repository o @Service

ad esempio se per accedere ad un database si vuole poter agire inizializzando le stringhe di accesso al file con la possibilità di cambiarlo senza toccare il codice

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://www.spri
       http://www.springframework.org/schema/aop http://www.springframework.org/schema
       http://www.springframework.org/schema/tx http://www.springframework.org/schema/
       http://www.springframework.org/schema/context
       http://www.springframework.org/schema/context/spring-context.xsd">


<!--
<bean id="dataSource"
    class="org.springframework.jdbc.datasource.DriverManagerDataSource">
    <property name="driverClassName" value="org.apache.derby.jdbc.EmbeddedDriver"/>
    <property name="url" value="jdbc:derby:DATABASE"/>
    <property name="username" value="admin"/>
    <property name="password" value="admin"/>
</bean>
-->
```

<**bean id**="dao1" **class**="it.marconivr.Dao">

<!-- la classe Dao non ha proprietà da settare e si costruisce correttamente grazie al file Persistence.xml contenuto nel package che contiene la classe Dao -->

</**bean**>

2) File dispatcher-servlet.xml

Serve per indicare a Spring o meglio al main di Spring in quali classi o in quali package cercare le annotazioni per eseguirle

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:context="http://www.springframework.org/schema/context"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:p="http://www.springframework.org/schema/p"
       xmlns:aop="http://www.springframework.org/schema/aop"
       xmlns:tx="http://www.springframework.org/schema/tx"
       xsi:schemaLocation="http://www.springframework.org/schema/beans http://ww
       http://www.springframework.org/schema/aop http://www.springframework.org/
       http://www.springframework.org/schema/tx http://www.springframework.org/s
       http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context.xsd">

    <context:annotation-config />
    <!-- inserito da spring-servlet.xml -->
     <context:component-scan
        base-package="it.marconivr.controller,it.marconivr.DAO" />
```

Appendice A:

Design pattern DAO DatabaseAccessObject

Il design Pattern DAO richiede che tutte le operazioni su una tabella *nomeTabella* di DB rappresentata da una classe persistente (in JPA) siano incapsulate in un classe *nomeTabellaDao.*

Nel linguaggio informatico il suffisso ***Dao*** indica con una convenzione precisa che quella classe si occupa di incapsulare chiamate al DB.

Il DB risulta **totalmente disaccoppiato** dall'applicazione, che usa la classe Dao per accedere alle informazioni come se queste fossero in memoria.

Esempio tabella Persona

```
public class PersonaDao {

    // mockup della tabella

    private List<String> tabellaPersona = null;

    public PersonaDao () {...11 lines }
    // MockUp method FindAll
    public List<String> findAll () {...4 lines }

    // Mockup method FindByName
    // Restituisce qualsiasi riga che contenga name
    public List<String> findByName ( String name) {...13 lines }

}
```

La classe Dao riunisce tutte le cassi Dao di tutte le tabelle delle classi persistenti.

```
@Repository("dao1")
public class Dao {

    // Nota: x la classe Dao si usano solo final static
    // si possono fare molte istanze di Dao, ma tutte vedono le
    // stesse var static  con reference non modificabile

    final static public PersonaDao personaDao = new PersonaDao();
    // final static public prestazioneDao = PrestazioneDao() ;
    public Dao(){      // costruttore senza paranetri

    }


}
```

Per inciso l'annotazione **@Repository("dao1")**

è la comunicazione a Spring che deve fare una istanza di questa classe e deve assegnarla nel dizionario ("*magazzino*") delle istanze alla stringa unica **"dao1".**

Il dizionario delle istanze in Spring si chiama **ApplicationContext** ed è una specifica classe del framework.

Per ripasso vediamo lo stack completo con cui si accede ai Database nelle applicazioni Java.

Ricordiamo i vari livelli di disaccoppiamento:

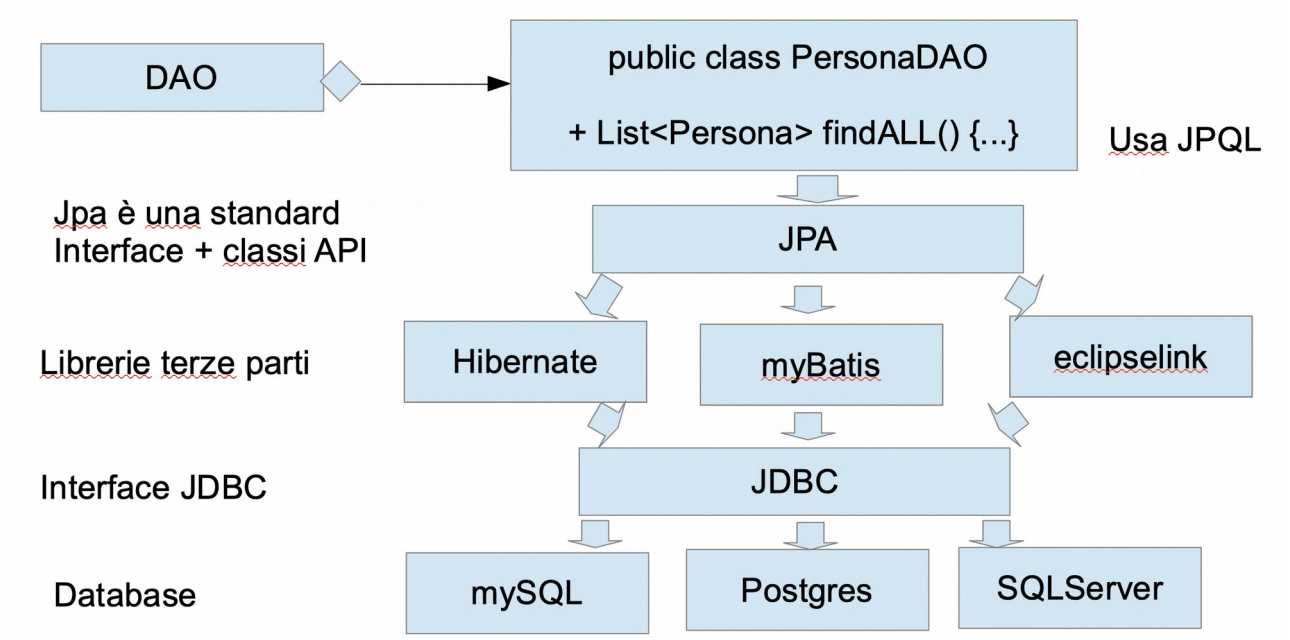
1) **Dao** disaccoppia l'applicazione dalle chiamate al DB

2) JPA disaccoppia l'accesso al DB dalle librerie middleware (tipo Hibernate, eclipslink, myBatis ecc.)

3) le librerie di **middleware** (Hibernate, EclipseLink, myBatis ..) disaccoppiano dalla scelta del DB usando **JDBC** e adattano le loro **query in linguaggio generico SQL al particolare dialetto SQL del DB in uso, fornendo anche le classi Java per tutti i tipi di dato supportati da DB.**

4) **JDBC** è l'Interface unica per tutti che i produttori di DB implementano con uno specifico package per rendere accessibile da Java il proprio DB [Nota: JDBC non disaccoppia completamente perché espone le Query in forma nativa del singolo DB]

Usando DAO e JPA è possibile adattare il codice al cambiamento di librerie oppure di DB oppure di entrambe con poche istruzioni, pochissime se si usano tutte chiamate **JPQL**.


DAO
public class PersonaDAO
+ List<Persona> findALL() {...}
Usa JPQL
Jpa è una standard
Interface + classi API
JPA
Librerie terze parti
Hibernate
myBatis
eclipselink
Interface JDBC
JDBC
Database
mySQL
Postgres
SQLServer