### **Cosa sono le servlet?**

**Le servlet sono componenti lato server scritte in Java che intercettano le richieste HTTP**.

Vengono utilizzate per gestire la logica di business, mentre le **JSP** si occupano della parte visiva. In pratica, le **servlet** sono gli antenati dei moderni controller.

Ad esempio, una servlet è una classe Java che estende HttpServlet, la quale a sua volta estende GenericServlet e infine Object. Generalmente, si effettua un override dei metodi come doGet o doPost per gestire le richieste HTTP. La logica di business viene implementata all’interno della servlet stessa, senza una separazione stratificata come nei moderni controller e servizi.

**Ciclo di vita di una servlet**

Il ciclo di vita di una servlet è gestito dall’application server (ad esempio Tomcat o Wildfly), l’application è un software ed il container che è un suo componente controlla l'esecuzione del codice, non viceversa, quindi si occupa di:

1. **Caricare** la servlet.
2. **Inizializzarla** tramite il metodo init() (eseguito una sola volta).
3. Gestire le richieste HTTP (doGet, doPost, ecc.).
4. **Distruggerla** tramite il metodo destroy() (eseguito una sola volta alla fine).

**Configurazione nel file web.xml**

Per utilizzare una servlet, bisogna configurarla nel file web.xml:

|  |
| --- |
| <web-app>     <servlet>         <servlet-name>MiaServlet</servlet-name>         <servlet-**class**>MiaServlet</servlet-**class**>     </servlet>     <servlet-mapping>         <servlet-name>MiaServlet</servlet-name>         <url-pattern>/ciao</url-pattern>     </servlet-mapping>  </web-app>  ​ |

Le servlet supportano metodi CRUD tramite i metodi HTTP come doGet, doPost, doPut, ecc., che rispondono alle rispettive richieste.

**Differenza tra getParameter e getAttribute**

* **getParameter**: Recupera dati di input provenienti dall'esterno, come i parametri della richiesta HTTP.
* **getAttribute**: Recupera variabili temporanee impostate nel codice.

**Esempio di servlet:**

|  |
| --- |
| **public** **class** UserServlet **extends** HttpServlet {  **protected** void doGet(HttpServletRequest request, HttpServletResponse response) {         PrintWriter writer = null;  **try** {             writer = response.getWriter();             // Uso del writer per generare la risposta        } **finally** {  **if** (writer != null) {                 writer.close();  // Chiusura manuale            }        }    }  }  ​ |

La gestione delle risorse, come nel caso di JDBC, richiede l’apertura e la chiusura manuale, utilizzando costrutti come try-with-resources.  
  
***END TO END, però singola responsabilità.***

**Cos’è l’inversion of control?**

**È la delegazione delle responsabilità di *creazione, inizializzazione e distruzione*** di un oggetto a quello che è un CONTAINER di un application server.  
  
Quindi, la servlet contiene i metodi doGet, doPost ecc., ***ma è il container a decidere quando chiamarli***, ovvero quando una richiesta http viene inviata al server è il container che decide quali metodi inviare e preparare quindi ***httpResponse e httpRequest.***  
  
  
**Cosa sono le JSP?**

Le JSP rappresentano **una tecnologia utilizzata per creare pagine web**, con l’utilizzo di html e java, vengono utilizzare per la creazione di creazione di pagine web con un rendering lato server piuttosto lato client. ***(un tempo c'era più difficoltà a mappare elementi provenienti dal server in linguaggi di creazione pagine web, ecco perché si faceva lato server).  
Inoltre la modifica ad un membro necessitava di un refresh della pagina con il caricamento dell’intero processo.***

**Cos’è l’EAR?**

L’EAR rappresenta un archivio contenente la parte di logica di business JAR (DAO, ecc.), e la parte visiva WAR (File dinamici: JSP, Servlet, File Statici: HTML, CSS) quindi nell’ear abbiamo (Java Archive e Web Archive), l’ear viene distribuita ad un software denominato application server, che contiene un componente chiamato container, che sfrutta il concetto di Inversion Of Control e decide quando chiamare i metodi della servlet di inizializzazione, creazione, CRUD e distruzione della servlet stessa, ***lo sviluppatore non deve chiamare esplicitamente questi metodi - è il container che gestisce quando invocarli.*  
  
Cos’è JDBC?**

**JDBC è un api di basso livello che consente di far comunicare un database a Java,** attraverso degli oggetti, denominati ***Statement e PreparedStament.***  
  
JDBC per funzionare necessità di:

* Un driver jdbc specifico relativo al DB da usare, nel caso di MySQL abbiamo il MYSQL connector J.

**Gli Statement:** effettuano le query direttamente sugli attributi di un oggetto e sono passibili di SQL injection.

|  |
| --- |
| String query = "SELECT \* FROM users WHERE id = " + userInput; |
|  |

**I PreparedStatement:** invece usano una stringa di codice separata dalla query evitando le criticità che si hanno negli Statement, qui gli inserimenti sono separati dalla query, oltre a questo il piano di esecuzione di una query viene compilato una sola volta e riutilizzato, con gli Statement classici questo non è possibile

|  |
| --- |
| PreparedStatement ps = connection.prepareStatement(     "INSERT INTO users (name, email) VALUES (?, ?)"  ); // Aggiungiamo multiple operazioni al batch  ​  **for** (User user : userList) {     ps.setString(1, user.getName());     ps.setString(2, user.getEmail());     ps.addBatch(); // Aggiungiamo al batch  }  ​  // Eseguiamo tutte le operazioni batch in una volta  ps.executeBatch(); |

Inoltre i risultati vengono elaborati e mostrati con il ***resultSet***, è possibile conservare i dati in batch, per poi fare una esecuzione del batch per mostrare tutti i dati, quindi processarli infine tutti insieme.

|  |
| --- |
| ps.setString(1, "value1"); // Prepara operazione 1  ps.addBatch();            // Accumula  ​  ps.setString(1, "value2"); // Prepara operazione 2  ps.addBatch();            // Accumula  ​  ps.executeBatch();        // Esegue tutte le operazioni accumulate  ​ |

**Apertura di una connessione:** L’apertura di una connessione avviene attraverso un singleton, un pattern che garantisce che ci sia una singola istanza accessibile a tutti, tale singleton consente di andare ad aprire una connessione per volta, ed è un metodo static accessibile globalmente, questo perché consente di ottimizzare le risorse e prevenire il sovraccarico del database.  
La chiusura delle risorse è manuale e per facilitarla si usa un try with resources, che ha un finally implicito.

|  |
| --- |
| **public** **static** **synchronized** ConfigurationServlet getInstance() {  **if** (instance == null) {             instance = **new** ConfigurationServlet();        }  **return** instance;    }  ​ |

**Cos’è un Framework?**

**Un framework è una struttura che definisce un’architettura di base di sviluppo e può  
contenere un insieme di libreria**, in sostanza consente di migliorare lo sviluppo di applicazioni, si applica al concetto di ***Inversion Of Control***, tramite il framework mi inserisco in un flusso predeterminato, quindi la responsabilità del flusso di controllo viene sposta dal programmatore al framework stesso (IL PROGRAMMATORE È PARTE DI UN SISTEMA).

**Però la IoC non è solo questo:**  
Non è solo questione di spostare il flusso di controllo dal programmatore al framework, ma è un principio architetturale più profondo che inverte la dipendenza tradizionale del software. Invece di essere il nostro codice a chiamare le librerie, è il framework che chiama il nostro codice.

***"Hollywood Principle": "Don't call us, we'll call you".***  
  
(P.S ***Le librerie* sono un insieme di funzioni riutilizzabili**, è il programmatore a scegliere quando utilizzare quel componente)  
  
**MyBatis è un framework non completamente ORM**, poiché non consente un mappaggio automatico delle query, ma lo fa attraverso ***interfacce mapper*** che contengono dei **metodi astratti** di CRUD e sono corrispettivi alle query SQL, e con un file ***mapper.xml*** (In Jakarta).

**Cosa sono i DAO?**  
In Jakarta EE esistono i DAO, **essenzialmente sono dei pattern design che gestiscono la logica di accesso ai dati, consentono quindi di accedere ai dati presenti nel db comunicando con i mapper.**Sono delle classi che fanno da Wrapper ai mapper nel DB, questo strato include:

* La gestione delle eccezioni specifiche al database.
* L’implementazione di controlli di sicurezza.
* Arricchimento dei dati prima di restituirli.

È possibile far comunicare i mapper con dei DAO per gestire la logica di accesso ai dati, un controller poi potrà comunicare con tale DAO in maniera tale da poter prendere dei dati e poi restituirli al ***frontend*** come risposta alla richiesta.

Quindi abbiamo:

1. **Il frontend:** fa la richiesta
2. **Il controller** ascolta e richiede i dati al dao, il tale chiede questi dati al mapper.
3. **Il mapper** li pesca dal database
4. La risposta torna indietro fino al frontend.

**Cos’è XML?**

**XML è un linguaggio di markup utilizzato per trasportare dati**, a differenza del html che viene utilizzato per visualizzare contenuti su web.

MyBatis per funzionare necessita di un file xml denominato ***mybatis-config***, ovvero, un file xml utilizzato per centralizzare la connessione.

**Cos’è l’EJB?**  
  
**Gli EJB sono delle classi Java, oggetti che consentono a due applicazioni Java di comunicare tra loro**. L’EJB locale è un oggetto utilizzato all’interno della stessa JVM e non è serializzabile, l’EJB remoto è serializzabile in flusso di bye in maniera tale da poter viaggiare in rete:  
  
L'architettura EJB remota funziona così:

1. **JVM Locale (Client)**
   * Riceve una richiesta che richiede un servizio remoto.
   * Crea uno stub (una specie di "rappresentante locale" del servizio remoto).
   * Prepara i dati per il viaggio (serializzazione).
2. **Rete**
   * I dati serializzati viaggiano attraverso la rete.
   * Qui emergono le prime criticità che hai giustamente intuito sulla perdita di pacchetti.
3. **JVM Remota (Server)**
   * Riceve i dati serializzati.
   * Li riconverte in oggetti Java (deserializzazione).
   * Esegue il metodo richiesto.
   * Prepara la risposta (nuova serializzazione).
4. **Ritorno al Client**
   * La risposta fa il viaggio inverso
   * Lo stub riceve la risposta e la presenta al chiamante

(Questo poteva creare delle problematiche, come una perdita di pacchetti se la connessione fosse instabile e sono stati soppiantati con il tempo dai servizi rest).  
  
Una comunicazione più esplicita per gestire gli errori, se una chiamata Rest fallisce vedi chiaramente il codice http di errore, con gli EJB dovevo gestire le eccezioni remote specifiche di Java.  
  
**E L’EJB locale?**   
  
**L’EJB locale** viene utilizzato per gestire richieste all'interno della **stessa JVM** in cui è distribuito il componente. Per esempio, una chiamata per gestire ordini relativi alla sede di Milano potrebbe essere instradata al server di Milano, e in questo caso il metodo usato sarebbe un EJB locale.

D'altra parte, **l’EJB remoto** consente di gestire richieste che devono comunicare tra **JVM diverse**, tipicamente distribuite su server differenti. Ad esempio, una chiamata per ottenere informazioni sulla merce presente in magazzino a Trento verrebbe instradata alla JVM remota del server di Trento, sfruttando un EJB remoto per accedere ai dati.

**Esistono 3 tipi di EJB:**

1. **MDB:** Sono orientati alla programmazione asincrona, (Il client non deve aspettare che tutti i servizi siano pronti, quando è pronto uno lo manda) permettono a più client di sottoscriversi ad un servizio e ricevere i risultati quando pronti, quindi gestiscono questi messaggi in maniera asincrona.

**(sono particolarmente utili in scenari di integrazione enterprise dove è necessaria una comunicazione asincrona affidabile, come nella gestione di code JMS (Java Message Service).**

1. **Entity Beans:** Venivano utilizzati per gestire la persistenza dei dati, ma sono stati superati da nuove tecnologie ORM.
2. **Session beans:** Vengono utilizzati per stabilire una connessione tra client e server, e sono suddivisi in 3 categorie principali:

* ***Stateful*** mantengono lo stato conversazionale, questo è utile per la creazione di carrelli della spesa, uno stato chiaramente effimero che a livello del server vive in una JVM in un client viene in uno storage.
* ***Stateless*** non lo fanno: operazioni di cambio valuta
* ***Singleton***, viene creata un'unica istanza condivisa da tutti i client.

Lo stato di vita di un EJB è gestito da un container dell’Application Server, che si chiama ***EJB Container,*** attraverso quello che è il concetto di ***Inversion Of Control***.

l’EJB viene inizialmente:

1. **Creata,** il container a questo punto invocherà i metodi relativi alla creazione, viene quindi fatta un ***iniezione delle dipendenze se necessaria***, da degli oggetti esterni, poi abbiamo:
2. **Post creazione,** qui vengono invocati i metodi di inizializzazione
3. **Ready state,** il bean è finalmente pronto per essere uso
4. **il passive state,** solamente per gli EJB stateful che rimangono inattivi per un po’ ma si attivano quando richiamati, poiché il loro stato effimero consente un salvataggio nella JVM e la distruzione dell’EJB stesso.

***Perché viene delegato il container?*** Beh per una separazione delle responsabilità e ottimizzazione delle risorse.

**Cos’è JPA?**

**È una specifica definita da Java per la persistenza degli oggetti in database relazionali**, lo fa attraverso le entity, che collegano fondamentalmente una singola entità a due mondi:  
***il database e java.***

***JPA UTILIZZA TRE PILASTRI FONDAMENTALI:***

***l primo pilastro sono le Entità:  
Il secondo pilastro sono le Relazioni tra Entità,  
Il terzo pilastro è l'EntityManager, che rappresenta il cuore operativo di JPA.***

JPA sfrutta dei metadati, quindi dati aggiuntivi che il compilatore riconosce per determinare la natura dei membri, questi metadati prendono il nome di ***@Annotations*** .

Una classe può essere un entity, può essere definita come **table** ed i suoi membri possono rappresentare le colonne del database.  
È necessario avere almeno un id, tale ID rappresenterà la chiave primaria di quella tabella.  
  
La generazione dell’id può essere:

1. **Identitaria:** si affida completamente al database per la generazione.

|  |
| --- |
| @Entity  **public** **class** Studente {     @Id     @GeneratedValue(strategy = GenerationType.IDENTITY)  **private** Long id;  **private** String nome;  } |

1. **Sequenziale:** richiede più configurazione ma offre maggior controllo.

|  |
| --- |
| @Entity  **public** **class** Professore {     @Id     @GeneratedValue(strategy = GenerationType.SEQUENCE,                     generator = "prof\_sequence")     @SequenceGenerator(name = "prof\_sequence",                       sequenceName = "prof\_seq",                       allocationSize = 1)  **private** Long id;  **private** String nome;  } |

Le relazioni tra entity sono assolutamente possibili, esse vengono definite attraverso le annotazioni.  
  
In una relazione JoinTable possiamo creare una tabella di mezzo, e collegarla alle chiavi primarie di queste con:

|  |
| --- |
| @ManyToMany    @JoinTable(        name = "studente\_corso",              // nome tabella di join        joinColumns = @JoinColumn(            // colonna che riferisce questa entità            name = "studente\_id",             // nome colonna nella tabella di join            referencedColumnName = "id"       // colonna riferita in Studente        ),        inverseJoinColumns = @JoinColumn(     // colonna che riferisce l'altra entità            name = "corso\_id",                // nome colonna nella tabella di join            referencedColumnName = "id"       // colonna riferita in Corso        )    )  **private** List<Corso> corsi;  } |

**Il mappedBy** invece va messo sempre nel lato NON proprietario della relazione.  
Per esempio in Vincitore abbiamo un mappedBy di cittadino.

|  |
| --- |
| @Entity  **public** **class** Autore {     @OneToMany(mappedBy = "autore")  **private** List<Libro> libri;  }  ​  @Entity  **public** **class** Libro {     @ManyToOne  **private** Autore autore;  } |

**La classe EntityManager cos’è?**

Questa classe è relativa a JPA, viene utilizzata per gestire il ciclo di vita delle entity.

* **Transient**: l’entità è stata creata ma non esiste nel db, metodo associato persiste.
* **Managed**: a questo punto l’entità è persistente
* **Detached**: oggetto disconnesso dal contesto di persistenza,
* **Removed**: rimozione totale dal db.

I metodi che usa JPA sono: ***persist, find, merge, remove.***

|  |
| --- |
| // 1. Stato TRANSIENT  Studente studente = **new** Studente("Mario");  // L'oggetto esiste in Java ma JPA non lo conosce ancora  ​  // 2. Transizione a MANAGED  entityManager.persist(studente);  // JPA inizia a tracciare l'oggetto e sincronizzarlo col database  ​  // 3. Transizione a DETACHED  entityManager.detach(studente);  // L'oggetto mantiene il suo ID ma non è più sincronizzato  ​  // 4. Ritorno a MANAGED via MERGE  Studente riconnesso = entityManager.merge(studente);  // JPA riprende a tracciare l'oggetto  ​  // 5. Transizione a REMOVED  entityManager.remove(riconnesso);  // L'oggetto viene marcato per l'eliminazione |

Pensa a uno studente universitario:

* All'inizio esiste ma non è registrato (TRANSIENT)
* Si iscrive ufficialmente (PERSIST → MANAGED)
* Va in Erasmus temporaneamente (DETACHED)
* Rientra nella sua università (MERGE)
* Si laurea e viene rimosso dagli studenti attivi (REMOVE)

Inoltre i dati della relazioni possono essere caricati attraverso un fetch.type, questo fetch type può essere:  
  
***lazy o eager***, quando arriva una richiesta http, questa s’interlaccia al server e quel fetch.type gestisce la risposta.

* **Lazy:** i dati vengono caricati quando necessari, quindi richiesti esplicitamente nel codice, quando farò order.getOrderItems(), richiamerò gli oggetti dell’ordine e i dati verranno caricati.
* **Eager:** i dati vengono caricati immediatamente tutti insieme e quando richiamerò il metodo saranno già pronti, attenzione però, utilizzare eager può portarci ad un’eccezione, denominata ***LazyInitializationException***, i dati devono essere caricati prima della chiusura della connessione.

**Cos’è Hibernate?**  
 **Hibernate è un framework ORM utilizzato per la mappatura di oggetti java in un DB relazionale.**  
Hibernate implementa JPA e lo espande, viene proprio considerato un "JPA provider", ovvero un'implementazione concreta delle specifiche JPA.  
Questo è importante perché significa che potremmo teoricamente sostituire Hibernate con un altro provider JPA (come EclipseLink) senza modificare il codice che usa le API JPA.

Ad ogni modo, esso utilizza ***Session*** che è un’interfaccia che ha i medesimi metodi di gestione del ciclo di vita delle entity come in JPA ma ne cambia la semantica, ***save, load, get, evict, flush***, inoltre deve chiamare il session.close manualmente.

Migliora il caching, ovvero avere il caching mette a disposizione una memoria veloce da utilizzare.

**Facciamo un esempio:** il metodo ***get*** per esempio consente di prendere dati dal db, il load li carica e poi questi dati posso riutilizzarli quando necessario, ecco, il caching fa questo, quel dato viene salvato nella memoria ***cache*** e utilizzato quando necessario, evitando un caricamento completo di un record.  
  
La cache infatti è suddivisa in due livelli.

1. **Cache di primo livello**: la prima è locale a quella singola sessione,
2. **Cache di secondo livello:** la seconda è condivisa da tutte le sessioni.

|  |
| --- |
| Session session = sessionFactory.openSession();  **try** {     // Prima richiesta - va sul database     Cliente cliente = session.get(Cliente.**class**, 1L);       // Seconda richiesta - usa la cache di primo livello     Cliente stessoCliente = session.get(Cliente.**class**, 1L);       // In un'altra sessione - potrebbe usare la cache di secondo livello     Session altraSession = sessionFactory.openSession();     Cliente clienteCached = altraSession.get(Cliente.**class**, 1L);  } **catch** (Exception e) {     // gestione errori  } |

**Jpa invece?**  
JPA utilizza solo una cache di primo livello relativa al contesto di persistenza, questa cache di primo livello in JPA si chiama **Persistence context (Locale).**

**Cos’è SpringBoot?**

**SpringBoot è un potente framework utilizzato per la creazione di applicativi aziendali, robusti e scalabili.**  
Nasce per migliorare la vita ai programmatori, migliorando la semantica dei metodi e creando metodi standardizzati evita l'uso di EAR per il deployment dell’applicazione usando un server dedicato quale Tomcat.  
Gestire le configurazioni di Spring Boot è molto più semplice perché il tutto viene gestito all’interno di un file denominato application properties, un centro di controllo, nato poiché centralizza il processo e le connessioni.  
Tutto questo migliora la manutenibilità del codice ed evita di avere diversi file sparsi di configurazione in XML, offre strumenti come Actuator per monitorare e gestire l'applicazione in tempo reale, semplificando il debugging e la risoluzione dei problemi.

**Cos’è REST?  
  
Rest è un’architettura che ci fornisce uno standard per la costruzione di servizi web**.

Si basa su dei principi fondamentali:

In un’applicazione Restful ogni richiesta è Stateless. è a sé,   
questo ci consente di avere

* **Scalabilità**
* **Affidabilità**
* **Suddivisione client server:** Questo ci consente di avere le due parti divise tra di loro e possono evolversi indipendentemente senza un impatto tra loro.

*Il client si preoccupa solo dell'interfaccia (come fare le richieste).  
Il server si preoccupa solo della logica (come processare le richieste)*

Inoltre aggiunge la questione del cacheable, questo consente di evitare richieste ridondanti, poiché è possibile memorizzare grazie a REST richieste, ed esse posso rimandarle attraverso la memoria cache.  
  
**Proprio perché le richieste sono:**

* Idempotenti (GET, PUT, DELETE)
* Ben definite (attraverso URI)
* Standardizzate (metodi HTTP)

è importante ricorda che REST consente di avere un’interfaccia uniforme su cui basarci ed i vantaggi sono ***la standardizzazione, flessibilità***.  
  
**I metodi http sono mappati ai metodi di CRUD.**

***Per concludere:*** la gestione degli errori è una parte fondamentale dell'architettura REST.

REST utilizza i codici di stato HTTP in modo molto strutturato.

* 200 Success
* 300 Redirect
* 400 Errore client
* 500 Errore server

**COSE DA AGGIUNGERE:**  
  
**1) Architettura Stratificata** Noto che hai una buona comprensione dei singoli componenti (Servlet, EJB, JPA), ma sarebbe utile approfondire come questi si integrano in un'architettura a strati completa. Come interagiscono questi componenti in un'applicazione enterprise reale? Quali sono i pattern architetturali comuni?

2)**Gestione delle Transazioni** Nel tuo materiale non vedo molto sulla gestione delle transazioni, che è un aspetto cruciale in Java Enterprise. In particolare:

* Transaction management in JPA/Hibernate
* Gestione delle transazioni distribuite con EJB
* ACID properties e loro importanza

**3)Sicurezza** Mentre menzioni la SQL injection con PreparedStatement, sarebbe utile approfondire:

* Autenticazione e autorizzazione in Java EE
* Security annotations
* Gestione delle sessioni

**4) Aspetti di Deployment** Come vengono deployate le applicazioni Java Enterprise? Quali sono le considerazioni da fare quando si passa da ambiente di sviluppo a produzione?

In JDBC non abbiamo sessioni ma connessioni al database, in quanto jdbc consente di mappare oggetti java in db relazionali, mentre in JPA usiamo il persistence context ed in Hibernate il caching, questo è importante, perché lo si fa?

Perché abbiamo la necessità di salvare in sessione dei dati che sennò verrebbero persi, infatti ***una sessione non è altro che uno stato***

La tua risposta mostra una comprensione profonda del concetto! Hai colto perfettamente il punto chiave: stiamo parlando di "stato" e della sua gestione. Permettimi di espandere il tuo ragionamento:

Hai detto una cosa molto importante: "una sessione non è altro che uno stato". Questo è un concetto fondamentale che collega tutti i pezzi:

**Connessione JDBC vs Persistence Context vs Cache**

* In JDBC: abbiamo una connessione "grezza" - è come avere un telefono sempre acceso con il database
* In JPA: il Persistence Context è come una "memoria di lavoro intelligente" - tiene traccia di cosa è cambiato
* In Hibernate: il caching aggiunge un ulteriore livello di "memoria intelligente" che ottimizza le performance

**Ma perché questa evoluzione?** Pensiamo al problema di base:

1. JDBC ci dà accesso diretto al database, ma dobbiamo gestire tutto manualmente
2. JPA introduce il Persistence Context per:
   * **Ridurre gli accessi al database**
   * **Mantenere la coerenza degli oggetti**
3. Hibernate porta questo concetto oltre con il caching a due livelli

Mi piace molto come hai collegato tutto al concetto di "stato". È esattamente questo: stiamo cercando di bilanciare:

* Performance (non possiamo sempre interrogare il database)
* Consistenza (dobbiamo essere sicuri che i dati siano corretti)
* Risorse (non possiamo tenere tutto in memoria)