**La Portabilità delle Applicazioni Java**

Il problema iniziale riguarda la necessità di rendere le applicazioni portabili, cioè eseguibili su qualsiasi macchina senza modificare il codice. Per risolverlo, Java sfrutta il concetto di "scrivi una volta, esegui ovunque" (Write Once, Run Anywhere), grazie alla Java Virtual Machine (JVM).

Tuttavia, un'ulteriore sfida è stata rendere queste applicazioni servibili tramite client tradizionali come i browser. Qui entrano in gioco le tecnologie basate su **application server**, che consentono di eseguire applicazioni Java all'interno di un ambiente controllato, detto **servlet container**. Un application server non è altro che una JVM avanzata, capace di ospitare e gestire molteplici applicazioni Java.

Questi server intercettano le richieste HTTP dei client e determinano quale classe Java e quali metodi invocare. Ad ogni applicazione dinamica sul server (Dynamic Server Application) corrisponde almeno una **servlet**, ovvero una classe Java conforme a determinate specifiche. L'application server, inoltre, si occupa di gestire l'intero ciclo di vita delle richieste e delle applicazioni.

**Comunicazione tra Applicazioni Java**

Un altro problema da risolvere era far comunicare due applicazioni Java remote, che potevano trovarsi su macchine diverse o su JVM diverse.

1. **Comunicazione su una stessa macchina**  
   Una soluzione semplice potrebbe essere lo scambio di file serializzati. Tuttavia, questo metodo è efficace solo se le applicazioni girano sulla stessa macchina.
2. **Comunicazione tra JVM su macchine diverse**  
   Per questo scopo sono state sviluppate tecnologie come gli **Enterprise JavaBeans (EJB)**.   
     
   Gli EJB **Gli EJB sono delle classi Java, sono degli oggetti e** rappresentano una soluzione standardizzata per consentire a due applicazioni Java di comunicare, anche a distanza, attraverso una chiamata remota. Questa tecnica crea un **segnaposto** nella JVM locale, che punta a un metodo situato in una JVM remota.

**Tipi di EJB**

Gli Enterprise JavaBeans sono suddivisi in tre categorie principali, ognuna progettata per risolvere esigenze specifiche:

1. **Message-Driven Beans (MDB)**  
   Gli MDB sono orientati alla programmazione asincrona e vengono veicolati tramite messaggi. La loro filosofia è semplice: consentono a più client di sottoscriversi a un servizio e ricevere i risultati quando disponibili. Questo approccio è ideale per scenari in cui il client può proseguire la propria esecuzione senza attendere immediatamente il completamento di una richiesta.

Un esempio tipico è un servizio con una **coda** di messaggi, dove i client richiedono lo stesso servizio in modo asincrono. Gli MDB si occupano di notificare i sottoscrittori quando i risultati sono pronti.

1. **Entity Beans**  
   Gli Entity Bean rappresentano un'interfaccia per accedere ai dati e implementano le classiche operazioni di CRUD (Create, Read, Update, Delete). In passato erano molto usati, ma oggi il loro ruolo è stato sostituito da tecnologie più moderne come i DAO (Data Access Object).
2. **Session Beans**  
   I Session Bean sono il tipo di EJB più comune e vengono utilizzati per stabilire una connessione tra client e server con un ciclo di vita definito. Offrono uno **stato conversazionale**, cioè una connessione che può mantenere informazioni specifiche tra una richiesta e l'altra. Si suddividono in tre tipologie:
   * **Stateful**: Mantengono una connessione stabile tra client e server. Quando un client interagisce con un oggetto stateful, il server si "ricorda" di quel client, garantendo che tutte le richieste successive siano gestite dalla stessa istanza. Questo è utile per applicazioni come carrelli di e-commerce, dove è necessario mantenere uno stato specifico durante la sessione.
   * **Stateless**: Non mantengono uno stato conversazionale. Ogni richiesta è indipendente e non c'è garanzia che un client riceva la stessa istanza di un bean per richieste successive. Gli stateless sono ideali per operazioni che non richiedono un contesto persistente.
   * **Singleton**: Viene creata un’unica istanza condivisa da tutti i client. Questa istanza è ideale per scenari in cui si desidera condividere risorse o configurazioni comuni tra tutti i client.

**Ciclo di Vita di un EJB**

Gli EJB seguono un ciclo di vita gestito dal server:

1. **Creazione**: Il bean viene creato nel container del server, che invoca il costruttore della classe.
2. **Iniezione delle dipendenze**: Il server verifica se il bean necessita di oggetti esterni per funzionare e li fornisce.
3. **Post-Creazione**: Vengono eseguiti metodi speciali di inizializzazione.
4. **Ready State**: Il bean è pronto per essere utilizzato. Il server mantiene un pool di bean per ottimizzare le prestazioni.
5. **Passive State**: ***[Solo per gli EJB stateful]*** Il bean viene messo in una modalità inattiva quando non è più utilizzato attivamente, ma può essere ripristinato successivamente.

L'EJB passa in uno stato in cui la sua sessione con il client è messa in pausa, ma la sua istanza viene mantenuta, per poter riprendere l'elaborazione quando il client torna a interagire.

1. **Distruzione**: Quando il bean non è più necessario, il server lo rimuove dalla memoria, eseguendo eventuali metodi di pulizia, come la chiusura delle connessioni al database.

**Inversione del Controllo (IoC)**

Un altro concetto fondamentale per lo sviluppo Java è l’**Inversione del Controllo (IoC)**. Questa tecnica consente di configurare dinamicamente l’istanziazione di un oggetto al di fuori del codice Java stesso. Normalmente, quando un oggetto viene creato nel codice, è necessario conoscere i costruttori e i metodi da chiamare. Ciò crea una forte dipendenza tra l’oggetto chiamante e l’oggetto chiamato.

Con l’IoC, queste informazioni vengono spostate all’esterno del codice, riducendo la rigidità e migliorando la manutenibilità del software. Un framework che implementa IoC è Spring, che utilizza la **Dependency Injection** per fornire automaticamente gli oggetti necessari a un bean. Questo approccio semplifica la gestione delle dipendenze e consente di scrivere codice più modulare ed estensibile.

Quasi tutto corretto, ma ci sono alcuni dettagli che possiamo perfezionare. Ecco una spiegazione più chiara:

**EJB (Enterprise JavaBeans):**

L'EJB è un'**API** che permette di creare e gestire componenti distribuiti e che consente la comunicazione tra applicazioni distribuite. La comunicazione può avvenire tra JVM locali o remote, utilizzando gli EJB locali o remoti.

**EJB Locale:**

* **EJB locale** è un oggetto che viene utilizzato **all'interno della stessa JVM**.
* Non è **serializzabile** perché non c'è bisogno di trasferirlo attraverso la rete (è confinato nella stessa macchina).
* Il client che invoca l'EJB locale lo fa direttamente, senza passare attraverso la rete. **Non sono necessari stub e skeleton** in quanto la comunicazione avviene direttamente tra client e server (entrambe nella stessa JVM).
* L'**EJB locale** è semplicemente un'interfaccia e un'implementazione che risiedono sulla stessa macchina.

**EJB Remoto:**

* **EJB remoto** è un oggetto che viene utilizzato per comunicare tra **JVM diverse**, quindi su macchine diverse.
* Gli oggetti EJB remoti **devono essere serializzabili** perché devono essere trasferiti attraverso la rete (inviati come flusso di byte).
* Quando un client invoca un metodo su un EJB remoto, la chiamata deve passare attraverso la rete e quindi devono essere utilizzati **stub e skeleton**.
  + **Stub**: È un oggetto che si trova **nella JVM client** (locale) e rappresenta l'oggetto remoto, inoltrando la richiesta di metodo attraverso la rete.
  + **Skeleton**: È l'oggetto che si trova nella **JVM remota (server)** e riceve la richiesta dal client, quindi la inoltra al vero EJB remoto che esegue l'operazione richiesta.

L'**EJB remoto** è effettivamente il **metodo vero e proprio** che si trova nella JVM remota, e attraverso il processo di serializzazione, la chiamata passa attraverso il network.

**Per riassumere:**

* L'EJB è un'API che consente di far comunicare due JVM remote.
* **EJB locale**: non è serializzabile, funziona nella stessa JVM (senza stub e skeleton, comunicazione diretta).
* **EJB remoto**: è serializzabile, comunica tra JVM diverse, utilizza **stub** nella JVM client e **skeleton** nella JVM server per inoltrare le chiamate e rispondere.
* Il client (nella JVM locale) invoca il metodo su un EJB remoto, ma attraverso lo **stub**, mentre il server (nella JVM remota) risponde tramite lo **skeleton**.

Quindi, a livello concreto, la differenza tra locale e remoto sta proprio nel fatto che con l'EJB remoto c'è la necessità di fare passare i dati attraverso la rete, e per questo bisogna serializzare gli oggetti, usare stub e skeleton, mentre con l'EJB locale no.

**Domande sugli EJB**

1. **Cos'è un EJB e perché viene utilizzato?**
   * Qual è il ruolo principale degli EJB in un'applicazione Java Enterprise?
   * In che modo gli EJB semplificano lo sviluppo di applicazioni distribuite?

Punto uno: Un EJB è una specifica lato server, quindi una classe utilizzata per far comunicare due JVM distanti.  
  
Punto due: Sono un potente strumento per lo sviluppo di applicazioni distribuite, grazie all’utilizzo del container ejb presente all’interno della application server il ciclo di vita degli EJB è automatico, e non deve essere gestito autonomamente dal programmatore.  
Inoltre l’ejb è una tecnologia sicura, rispetto al passato, ecco perché al giorno di oggi viene utilizzata, rimpiazzando servlet e JSP

**Quali sono i tipi principali di EJB?**

* + Qual è la differenza tra **Stateless Session Bean**, **Stateful Session Bean** e **Singleton Session Bean**?
  + Quando è opportuno utilizzare ciascun tipo?

1. **Cos'è il ciclo di vita di un EJB?**
   * Quali sono gli stati principali di un **Stateless Session Bean**?
   * Come cambia il ciclo di vita per un **Stateful Session Bean**?
2. **Cos'è il container EJB e qual è il suo scopo?**
   * Quali responsabilità delega il container (es. gestione del ciclo di vita, sicurezza, transazioni)?
   * Dove viene eseguito un EJB?
3. **Cos'è un'interfaccia EJB e a cosa serve?**
   * Qual è la differenza tra un'interfaccia locale e remota?
   * Quando si utilizza l'interfaccia locale e quando quella remota?
4. **Come si configura un EJB in un progetto?**
   * Come si definisce un EJB con annotazioni?
   * Qual è il ruolo del file ejb-jar.xml?
5. **Qual è la differenza tra EJB locale e remoto?**
   * In quali situazioni è utile utilizzare un EJB remoto?
   * Come viene gestita la comunicazione tra JVM diverse?
6. **Cos'è il dependency injection negli EJB?**
   * Come si utilizza l'annotazione @EJB per iniettare un EJB in un'altra classe?
   * Qual è la differenza tra il lookup manuale di un EJB e l'iniezione automatica?
7. **Come funzionano le annotazioni più comuni negli EJB?**
   * Cosa fanno annotazioni come @Stateless, @Stateful, @Singleton
8. **Cos'è un Singleton Session Bean?**
   * In quali scenari viene utilizzato?
   * Qual è la differenza tra un Singleton Bean e un Singleton puro in Java?
9. **Quali sono i vantaggi e gli svantaggi degli EJB?**
   * Perché gli EJB sono preferiti in alcune applicazioni aziendali?
   * Quali sono le alternative agli EJB in progetti moderni?
10. **Come si testa un EJB?**
    * Quali strumenti puoi usare per testare EJB in un ambiente controllato?
    * Come si testa un EJB che interagisce con un database o altre risorse esterne?
    * Perché sono state deprecate e sostituite da JPA?
    * Come avviene l'integrazione tra EJB e JPA?
11. **Come si integra un EJB in un'applicazione REST o SOAP?**
    * Come si espone un EJB come web service?
    * Quali annotazioni e configurazioni sono necessarie?