Domande teoriche riviste

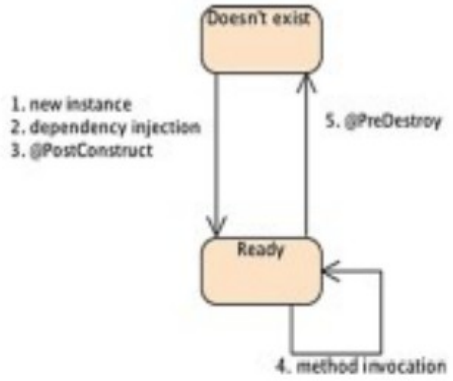
Elenco domande

Descrivere la Context and Dependency Injection

La Context and Dependency Injection, o CDI, fornisce un meccanismo per l'iniezione di componenti come Enterprise Java Beans (EJBs) o Managed Beans in altre componenti quali Java Server Pages (JSPs) o altri EJBs. La Dependency Injection è un design pattern che disaccoppia componenti dipendenti facendo sì che il container esegua una inject degli oggetti dipendenti al posto nostro. È possibile immaginarlo come l'opposto della Java Naming and Directory Interface, o JNDI, poiché questa fornisce un riferimento ad un oggetto su richiesta, mentre DI fa sì che il container inietti la dipendenza nell'oggetto che ne ha bisogno. Alla base di questo meccanismo vi è il concetto del "loose coupling, strong typing"; per descriverlo, analizziamo le due parti separatamente:

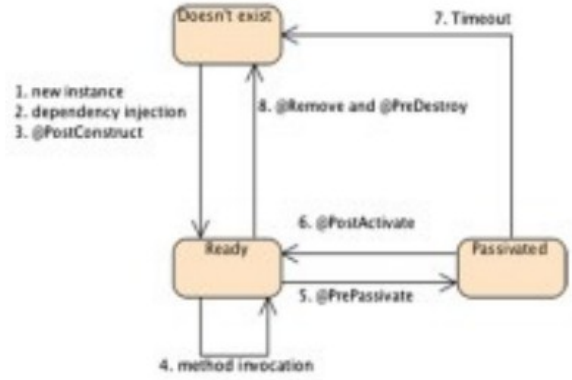
- Con loose coupling si intende il disaccoppiamento tra le componenti, ovvero la loro reciproca indipendenza. In genere si traduce in incapsulamento dei dati ed implica che un oggetto può usarne un altro senza conoscerne i dettagli interni.
- Con strong typing si intende invece un controllo dei tipi rigoroso ed è necessario nel nostro caso per poter legare in modo "safe" i beans.

Descrivere il ciclo di vita di un bean



Stateless e Singleton bean life cycle

1. Il ciclo di vita inizia quando il client richiede un riferimento al bean. Il container crea una nuova istanza
2. Alla creazione, vengono anche iniettate le dipendenze richieste
3. Se l'istanza appena creata ha un metodo annotato con @PostContruct il container lo invoca
4. Invocazioni dei metodi da parte dei client
5. Il container invoca il metodo annotato con @PreDestroy, se esiste, e termina il ciclo di vita della istanza del bean



Stateful bean life cycle

1. Il ciclo di vita inizia quando il client richiede un riferimento al bean. Il container crea una nuova istanza e la memorizza in memoria
2. Alla creazione, vengono anche iniettate le dipendenze richieste
3. Se l'istanza appena creata ha un metodo annotato con @PostContruct il container lo invoca
4. Invocazioni dei metodi da parte dei client
5. Se il client resta idle per un certo periodo di tempo, il container invoca il metodo annotato con @PrePassivate e rende passivo il bean in uno storage permanente
6. Se un client invoca un bean «passivated», il container lo attiva riportandolo in memoria ed invoca il metodo annotato con @PostActivate, se esiste
7. Se un client non invoca il bean all'interno del session timeout period, il container lo elimina
8. Alternativamente al passo 7, se un client chiama un metodo annotato con @Remove, il container invocherà il metodo annotato con @PreDestroy, se esiste, e termina il ciclo di vita del bean

Descrivere un Interceptor

Gli Interceptors sono componenti che si frappongono tra invocazioni di metodi di business per separarli dai cosidetti "cross-cutting concerns". Per il paradigma della Aspect-Oriented Programming, o AOP, questi sono, ad esempio, dei technical concerns, come log di ingresso o uscita da un metodo, o business concerns. Il container si occupa di chiamare gli interceptor prima e/o dopo l'invocazione di un metodo, assicurando i servizi agli EJB tramite una loro catena configurabile. Rispettano il concetto del "loosely

coupled" in quanto non conoscono l'implementazione concreta dei bean CDI con cui interagiscono. Ne esistono quattro tipologie: associati ad un costruttore della classe target, associati ad un metodo della classe target, di timeout e infine di callback del ciclo di vita.

Descrivere un Decorator

I Decorators sono componenti complementari agli interceptors che si occupano di aggiungere logica ai metodi di business, per cui ne conoscono l'implementazione interna. Per fare ciò "avvolgono" la classe da decorare, in modo da dover passare per forza per il decorator se si vuole interagire con tale classe.

Descrivere JPA

La Java Persistence API, o JPA, è un'astrazione di JDBC che lo rende indipendente da SQL. Fornisce object-relational mapping ed ha un linguaggio apposito con cui specifica le query, Java Persistence Query Language. Nel modello JPA, un'entità è un POJO che viene dichiarato, istanziato ed usato come altre classi Java. Viene mappata in una tabella tramite annotazioni, in particolare usiamo *@Entity* sulla classe e *@Id* al suo interno per definire l'ID univoco dell'oggetto. Il principio di ORM è quello di delegare a tools esterni o frameworks, come JPA, il compito di creare una corrispondenza tra oggetti e tabelle e, nel nostro caso, utilizza come metadati le annotazioni ed i descrittori XML.

Descrivere ORM

L'Object-Relational Mapping, o ORM, ha l'obiettivo di creare una corrispondenza tra oggetti e tabelle delegando il compito a tools esterni o frameworks; nel nostro caso si tratta di JPA. Per creare le corrispondenze utilizza come metadati le annotazioni ed i descrittori XML, mentre le informazioni sul database sono nel file persistence.xml.

Descrivere un Entity Manager

L'Entity Manager è un oggetto che si occupa della comunicazione tra entità e database, è responsabile per le operazioni CRUD e permette di eseguire query in formato JPQL. Per

crearne uno c'è bisogno di una EntityManagerFactory, la quale necessita di una Persistence Unit che indichi il tipo di database da usare ed il modo in cui connettersi ad esso. Di ciascuna PU si può specificare anche nome, classe a cui si riferisce, posizione tramite URL e modalità di autenticazione.