**Descrivere la Context and Dependency Injection**

Java EE ha degli Enterprise Java Beans (EJB), gli oggetti CDI (Context and Dependency Injection) sono costruiti sul modello base chiamato Bean Managed.

Dependency Injection è un modello di progettazione che disaccoppia componenti dipendenti, il contenitore inietta quegli oggetti dipendenti per gli sviluppatori e per chi scrive il codice.

Java Enterprise Edition (Java EE) 5 ha introdotto l’injection di risorse per gli sviluppatori. Permetteva agli sviluppatori appunto di iniettare risorse del contenitore come EJB, gestori di entità, origini dati, fabbriche JMS e destinazioni in un insieme di componenti definiti (Servlet, bean di supporto JSF ed EJB). A tale scopo, Java EE 5 ha introdotto un nuovo set di annotazioni (@Resource, @PersistenceContext, @PersistenceUnit, @EJB e @WebServiceRef).

Questo primo passo in Java EE 5 non è stato sufficiente, quindi Java EE 6 ha creato due nuove specifiche per portare la vera DI alla piattaforma: Dependency Injection e Context and Dependency Injection.

**CICLO DI VITA (LIFE-CYCLING MANAGEMENT)**

Se si desidera eseguire un bean CDI all’interno di un contenitore è necessario iniettare il bean ed il contenitore fa il resto, dunque il contenitore è responsabile del ciclo di vita del bean: crea l’istanza e si sbarazza di esso.

Il contenitore risolve le dipendenze e richiama qualsiasi metodo annotato con @PostConstruct prima della prima chiamata al metodo business sul bean. Quindi, la notifica di callback @PreDestroy segnala che l’istanza è in fase di rimozione dal contenitore.

**Interceptor**

Gli interceptor sono usati per intromettersi sulle invocazioni dei metodi di business. Questi potrebbero essere problemi tecnici, o preoccupazioni aziendali.

Queste preoccupazioni possono essere applicate automaticamente attraverso AOP (Aspect Oriented Programming) all’intera applicazione o a un sottoinsieme di esso.

Managed Beans supporta le funzionalità di tipo AOP fornendo la capacità di intercettare l’invocazione del metodo tramite gli intercettori.

Gli intercettori vengono automaticamente attivati dal contenitore quando viene richiamato un metodo Managed Bean.

Gli intercettori ti consentono di aggiungere preoccupazioni traversali ai tuoi bean.

Quando un client richiama un metodo su un bean gestito, il contenitore è in grado di intercettare la chiamata e elaborare la business logic prima che il metodo del bean sia invocato.

Gli intercettori rientrano in quattro tipi:

Constructor-level interceptros (Interceptor associato a un costruttore della classe target, @AroundConstruct)

Method-level interceptors: Interceptor associato a uno specifico metodo di business (@AroundInvoke)

Timeout method interceptors Interceptor che si interpone sui metodi di timeout con @AroundTimeout (utilizzato solo con il servizio timer EJB).

Life-cycle callback interceptors: Interceptor che si interpone sulle callback dell’evento del ciclo di vita dell’istanza di destinazione (@PostConstruct e @PreDestroy)

**Loose Coupling and Strong Typing**

Gli interceptor sono un modo molto efficace per disaccoppiare i problemi tecnici dalla logica aziendale. Con l’injection, un bean non è a conoscenza dell’implementazione concreta di qualsiasi bean con cui interagisce.

I bean possono utilizzare le notifiche degli eventi per disaccoppiare i produttori di eventi dai consumatori di eventi o dai decoratori per disaccoppiare le preoccupazioni aziendali.

E tutte queste strutture sono consegnate in modo tipicamente sicuro. CDI non fa mai affidamento su identificatori basati su String per determinare il modo in cui gli oggetti si adattano. Invece, CDI utilizza annotazioni fortemente tipizzate per collegare insieme i bean.

L’utilizzo dei descrittori XML è ridotto al minimo solo per l’uso d’informazioni veramente specifiche dell’implementazione.

**DECORATORS**

I decorators sono un modello di design comune della “Gang of Four”. L’idea è di prendere una classe e avvolgere un’altra classe intorno ad essa (ad esempio, decorarla).

In questo modo, quando chiami una classe decorata, passi sempre attraverso il decoratore circostante prima di raggiungere la classe target.

I decoratori hanno lo scopo di aggiungere ulteriore logica a un metodo di business. Non sono in grado di risolvere problemi tecnici.

Intercettatori e decoratori, sebbene simili in molti modi, sono complementari.

Iniezione delle dipendenze

Poiché Java EE è un ambiente gestito, non è necessario costruire manualmente le dipendenze, ma è possibile lasciarlo fare al contenitore che inserisce un riferimento per noi.

In breve l’iniezione delle dipendenze CDI è la capacità di iniettare i bean in altri in modo errato, il che significa che non c’è XML ma annotazioni.

L’iniezione già esisteva in Java EE 5 con le annotazioni @Resource, @PersistenceUnit o @EJB, ad esempio.

Con il CDI puoi iniettare praticamente ovunque grazie all’annotazione @Inject.

Si noti che in Java EE 7 è ancora possibile utilizzare gli altri meccanismi di iniezione.

**JPA**

Java Persistence API ci aiuta a rendere persistenti le entità in memoria, grazie a una classe Entity annotata con @Entity, che fa si che venga creata nel database una tabella sul codice scritto in Java.

Caratteristiche di un’entità

La classe dell’entità deve contenere l’annotazione @javax.persistence.Entity

L’annotazione @javax.persistence.Id è denotata per definire una semplice chiave primaria

La classe entità ha un costruttore vuoto che deve essere public() o protected(). Non può contenere la parola chiave final.

Se l’entità deve essere passata come oggetto remoto deve implementare l’interfaccia serializable.

**ORM (Object relation-mapping)**

Il principio di ORM è delegare a strumenti o framework esterni (nel nostro caso, JPA) il compito di creare una corrispondenza tra oggetti e tabelle. Il mondo di classi, oggetti e attributi può quindi essere mappato su database relazionali costituiti da tabelle contenenti righe e colonne. La mappatura offre una vista orientata agli oggetti.

I metadati consentono al provider di persistenza di riconoscere un’entità e interpretare la mappatura.

Il mapping è definito in un file XML esterno che verrà distribuito con le entità.

Questo può essere molto utile quando la configurazione del database cambia a scelta dell’ambiente.

Per accedere agli elementi del database si usano i metodi di accesso(get) sull’oggetto restituito dal database.

**ENTITY MANAGER**

L’entity manager è un elemento centrale di JPA.

Gestisce lo stato e il ciclo di vita delle entità, nonché l’interrogazione delle entità all’interno di un contesto di persistenza.

E’ responsabile della creazione e rimozione di istanze di entità persistenti e ricerca di entità tramite la loro chiave primaria, può utilizzare le query JPQL per recuperare le entità seguendo determinati criteri.

Il punto di forza di JPA è che le entità possono essere utilizzate come oggetti regolari da diversi livelli di un’applicazione e essere gestite dal gestore di entità quando è necessario caricare o inserire dati nel database.

**JPQL**

JPQL è un linguaggio che consente l’interazione tra i database relazionali e JPA.

JPQL viene utilizzato per definire ricerche su entità persistenti indipendenti dal database sottostante.

JPQL è un linguaggio di query che affonda le sue radici nella sintassi di SQL.

Ma la differenza principale è che in SQL i risultati ottenuti sono sottoforma di righe e colonne (tabelle), mentre JPQL utilizza un’entità o una raccolta di entità.

La sintassi JPQL è orientata agli oggetti e quindi più facilmente comprensibile dagli sviluppatori la cui esperienza è limitata ai linguaggi orientati agli oggetti.

Gli sviluppatori gestiscono il loro modello di dominio di entità, non una struttura di tabella, utilizzando la notazione a punti. JPQL utilizza il meccanismo di mappatura per trasformare una query JPQL in un linguaggio comprensibile da un database SQL. La query viene eseguita sul database sottostante con chiamate SQL e JDBC, quindi le istanze delle entità hanno i loro attributi impostati e restituiti all’applicazione.

Lo schema di una query JPQL è il seguente

SELECT, FROM, WHERE, ORDER BY, GROUP BY, HAVING

In JPQL esistono 5 diversi tipi di query e sono i seguenti

Query dinamiche, query con nome, Criteria API, Query native e query stored procedure.

Il tipo query dinamiche è la forma più semplice di query, costituita da nient’altro che una stringa di query JPQL specificata dinamicamente in fase di runtime.

Query con nome (naming): le query denominate sono statiche e non modificabili

Criteria API: JPA 2.0 ha introdotto il concetto di API di query orientata agli oggetti.

Query native: questo tipo di query è utile per eseguire un’istruzione SQL nativa anziché un’istruzione JPQL.

Query store procedure: JPA 2.1 introduce una nuova API per chiamare procedure stored.

DALLE SLIDE DELLA MALANDRINO (DELFINA)

PERSISTENCE UNIT (PU):

Il persistence unit indica all’entity manager il tipo di database da usare, ed i connection parameters definiti in un file XML chiamato persistence.xml

Le informazioni che è possibile inserire per ogni PU

Nome, classe a cui si riferisce, tipo di database, la posizione (URL), modalità per autenticazione

CICLO DI VITA DELLE ENTITA’

Quando si crea un’istanza di una entity con l’operatore new, l’oggetto esiste in memoria ma JPA non sa niente di lui.

Quando diventa “managed” dall’entity manager, la tabella mappa e sincronizza il suo stato.

Chiamare il metodo EntityManager.remove() cancella i dati dal database, ma gli oggetti Java continuano a rimanere in memoria fino all’intervento del garbage collector.

**JMS**

**MOM (Message-oriented middleware)**

MOM è un software che permette lo scambio di messaggi asincroni fra sistemi eterogenei.

Può essere visto come un buffer che produce e consuma messaggi.

E’ intrinsecamente loosely coupled dal momento che i produttori non sanno chi è all’altra estremità del canale di comunicazione ad usare il messaggio.

Il produttore e il consumatore non devono essere disponibili per comunicare.

**Architettura MOM, Scambio messaggi, modelli supportati**

Quando un messaggio viene inviato, il software che memorizza il messaggio e lo invia è detto Provider (broker). Il sender del messaggio è chiamato Producer e la locazione in cui il messaggio è memorizzato è detta destinazione.

La componente che riceve il messaggio è detta Consumer.

Ogni componente interessata ad un messaggio in una particolare destinazione può consumarlo.

**Architettura JMS**

Componenti di un architettura di messaging:

Un provider: componente necessaria per instradare messaggi, gestisce il buffering e il delivery dei messaggi.

Clients: una qualunque applicazione Java o una componente che produce o consuma messaggi per/da un provider. Il termine “Client” si usa genericamente per producer, sender, publisher, consumer, receiver, subscriber.

Messages: oggetti che i client inviano/ricevono dal provider.

Administered objects: oggetti (connection factories e destinazioni) fornite attraverso JNDI lookups o injection.

Il Provider permette comunicazione asincrona fornendo una destinazione dove i messaggi possono essere mantenuti finché non vengono instradati verso un client. Esistono due differenti tipi di destinazioni:

Point-to-point (P2P) model: la destination è chiamata coda. Il client inserisce un messaggio in coda, mentre un altro client riceve il messaggio. Una volta fatto acknowledge, il message provider rimuove il messaggio dalla coda.

Publish-subscribe model: la destination è chiamata topic. Il client pubblica un messaggio con un topic, e tutti i sottoscrittori al topic riceveranno il messaggio.

**Filtraggi**

Si fa in modo che arrivino solo i messaggi a cui si è interessati

Messaggio mandato in broadcast a diversi client, si definisce un selector in modo che venga consumato solo da consumer interessati.

Nessuno spreco di tempo e banda per ricevere cose di non interesse

Si può fare selezione su headers o metadati (JMS Priority < 6) o su proprietà custom (orderAmount < 260)

Il message selector è una stringa che contiene un’espressione

context.createConsumer(queue, “JMSPriority < 6”).receive();

Il messaggio viene creato dal Producer usando metodi per settare proprietà e priorità (nell’header)

Evitare messaggi obsoleti:

Un setting del time-to-live può essere di aiuto per evitare che messaggi obsoleti vengano recapitati ai destinatari.

Si setta il tempo in millisecondi, passato il quale il provider (il broker) rimuove il messaggio

Si utilizza il metodo del producer:

context.createProducer().setTimeToLive(1000).send(queue,message)

**Priorità di messaggi**

La priorità dei messaggi è contenuta nell’header del messaggio e i valori sono da 0 (bassa priorità) a 9 (alta priorità)

Esempio

context.createProducer().setPriority(2).send(queue, message)

**MDB**

I Message-Driven Beans sono message consumer asincroni eseguiti in un EJB container.

L’EJB container si occupa dei servizi, mentre l’MDB si occupa di consumare messaggi. Gli MDBs sono stateless.

In generale gli MDBs sono in ascolto su una destination (queue o topic) e, quando il messaggio arriva, lo consuma e lo processa.

Poiché sono stateless, gli MDBs non mantengono stato attraverso invocazioni separate.

Gli MDBs rispondono a messaggi ricevuti dal container laddove gli stateless session beans rispondo a richieste client attraverso un’interfaccia appropriata: local, remote o no-interface.

Come è fatto un MDB

MDB non è parte del modello EJB Lite: serve un implementazione full EE

Annotazione con @javax.ejb.MessageDriven (o XML equivalente)

Implementare la interfaccia del listener

Definita con public NON final o abstract

Deve esserci un costruttore senza argomenti, per permettere l’istanziazione automatica da parte del container.

La classe non deve avere il metodo finalize()

MDB as a consumer

Per natura gli MDBs sono progettati per funzionare come asynchronous message consumers

Gli MDBs implementano una message listener interface, che viene “risvegliata” dal container quando un messaggio arriva

Può un MDB essere un synchronous consumer?

Si ma non è raccomandato

**Topic**

Topic è come viene chiamata la destination nel publish-subscribe.

Il client pubblica un messaggio con un topic, e tutti i sottoscrittori al topic riceveranno il messaggio.

Il topic conserva i messaggi fino a quando non vengono distribuiti a tutti i subscribers, esiste una dipendenza temporale fra publisher e subscriber.

I subscribers NON ricevono i messaggi inviati PRIMA della loro sottoscrizione e, se il subscriber è inattivo per un periodo di tempo determinato, esso non riceve messaggi vecchi quando diventa nuovamente attivo.

Multipli subscribers possono consumare lo stesso messaggio. E’ utile per broadcast-type applications: singolo messaggio recapitato a diversi consumatori.

**Administraded object**

Oggetti che si configurano amministrativamente, e non programmaticamente. Il provider permette di configurare questi oggetti e li rende disponibili nello spazio dei nomi JNDI. Come JDBC datasources questi oggetti vengono creati solo una volta. I due tipi di oggetti amministrati sono: Connection factory: usato dai client per creare una connessione a una destinazione.

**Durable Consumer**

Nel modello public-subscribe un consumer che non è in esecuzione perde i messaggi che vengono postati sul topic.

Con i durable consumer si può controllare che i messaggi vengono mantenuti dal provider fino a quando tutti i consumer li hanno ricevuti.

Con i durable subscribers, un consumer che si riconnette riceve i messaggi che sono attivati durante la disconnessione.

Creazione attraverso JMSContext con una id specifica “unica”,

A questo punto il client inizia la connessione e riceve messaggi.

Il nome “uniqueID” è usato come identificatore della durable subscription

Ogni durable consumer deve avere un unique ID

che corrisponde alla dichiarazione di un’unica connection factory

per ogni potenziale durable consumer.

**JMS Recapito persistente**

Per gestire la persistenza, JMS supporta 2 modalità di message delivery: persistent e nonpersistent

Persistent delivery: messaggio salvato sul provider

Non-persistent delivery: messaggio non salvato

Persistent delivery è il valore di default… che può essere “degradato” per migliore le prestazioni

esempio: context.createProducer().**setDeliveryMode**(DeliveryMode.NON\_PERSISTENT).send(queue, message)

**DESCRIVERE LE CARATTERISTICHE DI UN’ARCHITTETURA SOA E I PROTOCOLLI ALLA BASE DEL FUNZIONAMENTO DEI WEB SERVICES**

Con Service Oriented Architecture (SOA) si indica solitamente un’architettura software adatta a supportare l’uso di servizi Web per garantire l’interoperabilità tra diversi sistemi così da consentire l’utilizzo delle singole applicazioni come componenti del processo di business e soddisfare le richieste degli utenti in modo integrato e trasparente.

L’architettura è basata anche sui servizi, i servizi hanno l’obiettivo di incapsulare una ben precisa funzionalità di business (logica applicativa), per renderla disponibile e accessibile come servizio software da parte di client software sul web – ciascun servizio può essere usato per costruire diverse applicazioni e processi di business.

In più permette anche l’integrazione dei processi di business all’interno dell’organizzazione.

I servizi hanno tre caratteristiche principalmente:

Tecnologia neutra: devono essere invocati attraverso tecnologie standard a minimo comune denominatore disponibili per quasi tutti gli ambienti IT.

Ciò implica che i meccanismi di chiamata (protocolli, descrizioni e meccanismi di scoperta) dovrebbero essere conformi agli standard ampiamente accettati.

Loosely coupled: non devono richiedere conoscenze o strutture interne o convenzioni (contesto) dal lato del cliente o del servizio.

Supportare la trasparenza della posizione: i servizi devono avere le loro definizioni e le informazioni sulla posizione memorizzate in un repository come UDDI ed essere accessibili da una varietà di client che possono localizzare e invocare i servizi indipendentemente dalla loro posizione.

**SOAP WEB SERVICES**

Si dice che i servizi web SOAP (Simple Object Access Protocol) siano "vagamente accoppiati" perché il client, ovvero il consumatore, di un servizio web non si devono conoscere i dettagli d’implementazione (come il linguaggio usato per svilupparlo, la piattaforma su cui gira).

Il consumatore è in grado di invocare un servizio Web SOAP utilizzando un'interfaccia intuitiva che descrive i metodi commerciali disponibili (parametri e valore di ritorno).

L'implementazione sottostante può essere eseguita in qualsiasi linguaggio (Visual Basic, C #, C, C ++, Java, ecc.). Un consumatore e un fornitore di servizi saranno ancora in grado di scambiare dati in un modo liberamente accoppiato: utilizzando documenti XML.

Un consumatore invia una richiesta a un servizio Web SOAP sotto forma di un documento XML e, facoltativamente, riceve una risposta, anche in XML.

I servizi Web SOAP riguardano anche la distribuzione.

Il software distribuito è in circolazione da molto tempo, ma, a differenza sistemi distribuiti esistenti, i servizi Web SOAP sono adattati al Web.

Il protocollo di rete predefinito è HTTP, un protocollo stateless noto e robusto.

In parole semplici, i servizi Web SOAP costituiscono una sorta di logica aziendale esposta tramite un servizio (ad esempio, il fornitore di servizi) a un cliente (ad esempio, il consumatore del servizio). Tuttavia, a differenza degli oggetti o EJB, i servizi Web SOAP forniscono un'interfaccia

liberamente accoppiata utilizzando XML. Gli standard del servizio Web SOAP specificano che

l'interfaccia a cui viene inviato un messaggio deve definire il formato della richiesta e della risposta

del messaggio e i meccanismi per pubblicare e scoprire le interfacce dei servizi Web (il registro del

servizio).

Il servizio Web SOAP può facoltativamente registrare la propria interfaccia in un registro

(Universal Description Discovery and Integration, o UDDI) in modo che un utente possa scoprirlo.

Una volta che il consumatore conosce l'interfaccia del servizio e il formato del messaggio, può

inviare una richiesta al fornitore di servizi e ricevere una risposta.