ChatGPT

Guida Completa alla Certificazione Oracle Java EE 7 Application Developer (1Z0-900)

Architettura Java EE 7 (Understand Java EE Architecture)

Teoria

Java EE 7 (Java Platform, Enterprise Edition 7) è una piattaforma standardizzata per lo sviluppo di applicazioni enterprise in Java. **Standardizzata** significa che definisce un insieme di **specifiche/API** che vari vendor (Oracle, IBM, Red Hat, ecc.) implementano nei loro application server. In altre parole, Java EE fornisce contratti (ad esempio le API Servlet, JPA, EJB, JMS, JSF, JAX-RS, ecc.) e ogni implementazione conforme (come GlassFish, WildFly/JBoss, WebLogic, WebSphere) offre il runtime che rispetta queste specifiche. Questo consente agli sviluppatori di scrivere applicazioni usando le API standard senza preoccuparsi dei dettagli di basso livello, demandando al **container** Java EE (il runtime dell'application server) la gestione di molti servizi di infrastruttura (gestione delle transazioni, sicurezza, pooling di connessioni, ecc.).

Un'applicazione Java EE tipica è **multilivello** (*multi-tiered*), generalmente suddivisa in: **tier client**, **tier server** (che a sua volta comprende un sottolivello web e uno business) e **tier di back-end dati**. - Il *client tier* può essere un browser web, un'app desktop o mobile, ecc., che interagisce con l'applicazione. - Il *server tier* ospita l'applicazione enterprise vera e propria, divisa spesso in **componenti web** (Servlet, JSP, JSF) per la parte di interfaccia utente e gestione delle richieste HTTP, e in **componenti di business** (EJB, CDI bean, servizi REST/SOAP) che incapsulano la logica applicativa e accedono ai dati. - Il *tier di dati* comprende database relazionali, sistemi legacy, servizi esterni, ovvero le **Enterprise Information Systems** da cui l'applicazione recupera o con cui persiste i dati.

Container Java EE: All'interno dell'application server, Java EE definisce diversi *contenitori* logici, ognuno incaricato di eseguire specifici tipi di componenti e di fornire determinati servizi. I principali contenitori in Java EE 7 sono: - **Web Container (contenitore web)** – Esegue componenti web come Servlet, JavaServer Pages (JSP) e JavaServer Faces (JSF). Fornisce servizi come gestione delle richieste HTTP, sessioni utente, sicurezza web, ecc. - **EJB Container (contenitore EJB)** – Esegue Enterprise JavaBeans (Session bean, Message-Driven bean). Offre servizi di transazionalità distribuita, sicurezza a livello metodo, pooling di istanze, gestione di concorrenza, ecc. - **Client Container** – Supporta applicazioni client Java standalone che accedono remotamente alle funzionalità server (meno comune; fornisce librerie client per EJB, JMS, ecc. in ambiente Java SE). - **Applet Container** – (Storicamente) eseguiva applet Java all'interno di un browser; non rilevante nelle moderne applicazioni enterprise.

Ogni container fornisce *servizi di runtime* standard ai componenti che ospita. Ad esempio, il container EJB offre servizi di *persistenza (JPA)*, *messaggistica (JMS)*, *gestione transazioni (JTA)*, *iniezione di dipendenze (CDI)*, *sicurezza*, ecc., senza che lo sviluppatore debba implementare queste funzionalità di supporto manualmente. I componenti Java EE dichiarano requisiti o configuarzioni tramite annotazioni o descrittori di deployment, e il container li soddisfa automaticamente (es: gestione pool di connessioni DB, gestione thread, serializzazione delle sessioni, etc.).

Componenti e tier applicativi: A livello di funzionalità applicative, i vari tipi di componenti Java EE sono pensati per ruoli differenti nei diversi tier. Ad esempio: - Una **Servlet** o una pagina **JSP/JSF** nel contenitore web gestisce richieste HTTP dal client (browser) e genera una risposta (HTML, JSON, etc.). Questo è tipicamente il *presentation layer* o *web layer*. - Un **EJB** (Session bean) nel contenitore EJB incapsula logica di business riutilizzabile, transazionale e sicura. È parte del *business logic layer*. Può essere invocato dalle componenti web o da altri EJB, e può a sua volta interagire con la persistenza (JPA) o altri servizi. - Un **Entity** JPA rappresenta una entità di dati mappata su DB ed è gestita dal *persistence layer* all'interno di un *contesto di persistenza*. Non viene eseguita in un container separato ma è usata tipicamente dai componenti di business (EJB o CDI) per interagire con il database.

Packaging e deployment: Un'applicazione Java EE può essere confezionata come: - **Modulo Web (WAR)**: archivio `.war` contenente Servlet, JSP, JSF, file statici, classi *helper* e opzionalmente EJB *lite* (Java EE 7 permette di includere EJB a livello base nel WAR). Viene eseguito nel web container (e EJB lite nel container EJB integrato). - **Modulo EJB (JAR EJB)**: archivio `.jar` contenente EJB (session bean, MDB) e relative classi. Viene eseguito nel EJB container. - **Applicazione Enterprise (EAR)**: archivio `.ear` che può combinare uno o più WAR, JAR EJB e librerie condivise. Un EAR rappresenta un'applicazione completa con più moduli ed è deployato sull'application server come unità unica. L'EAR può definire risorse condivise e configurazioni comuni ai moduli.

Per il deployment, l'archivio (WAR/EAR) viene caricato nell'application server (es. tramite console di amministrazione o tool a riga di comando). Il server allora: 1. Distribuisce i moduli nei rispettivi container (web, EJB, etc.), 2. Legge annotazioni e descrittori di deployment per configurare risorse, sicurezza, transazioni, 3. Avvia i componenti (ad esempio chiama i metodi di lifecycle come `@PostConstruct`), 4. Comincia a ricevere richieste e smistarle ai componenti appropriati (es. richieste HTTP alle Servlet, ecc.).

EJB vs CDI bean e Managed bean: Java EE 7 introduce diverse tipologie di *bean gestiti*: - **Enterprise JavaBeans (EJB):** sono componenti server-side con funzionalità avanzate out-of-the-box (transazioni CMT, sicurezza declarativa con ruoli, timer di container, pooling). Esistono vari tipi: *Session Bean Stateless, Stateful, Singleton* e *Message-Driven Bean*. Hanno un ciclo di vita governato dal container EJB. - **Managed Bean e CDI Bean:** Il termine *managed bean* in Java EE spesso indica un POJO gestito dal container web o CDI, eventualmente con annotazioni come `@ManagedBean` (JSF) o `@Named` (CDI) che ne consentono l'uso nelle pagine o l'iniezione altrove. I *CDI bean* sono oggetti gestiti dal container CDI (Context and Dependency Injection). CDI fornisce un meccanismo generale di iniezione di dipendenze, eventi e interceptors che può applicarsi a qualsiasi classe annotata opportunamente (es. con `@Named`, `@RequestScoped`, etc.), includendo anche i cosiddetti *bean POJO* di applicazione. - **Life cycle e scopes:** Gli EJB hanno lifecycle predefiniti (es. uno stateless EJB può essere istanziato in pool all'avvio e riutilizzato per varie richieste; uno stateful EJB è creato per un client e mantenuto finché la conversazione è attiva, poi eventualmente passivato su disco se inattivo, ecc.). I CDI bean invece possono essere legati a *scope* espliciti: es. `@RequestScoped` (dura una richiesta HTTP), `@SessionScoped` (dura una sessione utente), `@ApplicationScoped` (unico per l'intera applicazione), ecc., oppure essere *dependent* (senza scope proprio, legato al consumer). Capire gli scope significa comprendere quanto a lungo il bean vive in memoria e se è condiviso tra richieste o utenti. - **Interazione EJB/CDI:** In Java EE 7, EJB e CDI sono integrati. Un EJB può usufruire di injection CDI e viceversa. Ad esempio, è possibile *iniettare* un `@EJB` session bean dentro un CDI bean o una Servlet, così come un CDI managed bean può essere iniettato in un EJB (ad es. per risolvere dipendenze *qualificate* o risorse di configurazione). Tuttavia, ci sono differenze: EJB richiede il container EJB e fornisce servizi come transazioni automaticamente; un CDI bean di per sé non è

transazionale, ma può diventarlo se annotato `@Transactional` (una funzionalità introdotta in Java EE 7 via JTA 1.2), oppure delegando a EJB.

Annotazioni vs JNDI (Iniezione vs lookup): Prima dell'introduzione diffusa delle annotazioni e di CDI, l'accesso a risorse e componenti avveniva tramite **JNDI** (Java Naming and Directory Interface). Ad esempio, per ottenere un riferimento a un EJB o a una DataSource, il codice doveva fare un lookup JNDI con un nome stringa. In Java EE 7, l'uso di JNDI esplicito è in gran parte rimpiazzato dall'**iniezione di risorse e dipendenze** tramite annotazioni: - `@EJB` per iniettare riferimenti ad EJB locali o remoti, - `@Resource` per iniettare risorse come DataSource JDBC, code di messaggi JMS, variabili ambiente, ecc., - `@Inject` (CDI) per iniettare qualunque bean CDI o risorsa, eventualmente qualificato da annotazioni custom (@Qualifier), - `@PersistenceContext` per iniettare l'EntityManager JPA, - `@PersistenceUnit` per iniettare un EntityManagerFactory, ecc.

Durante il deployment, il container risolve queste dipendenze annotando i campi/setter appropriati: in pratica *dietro le quinte* effettua il lookup JNDI corretto e assegna l'istanza. Ciò rende il codice più pulito e tipizzato, evitando errori di stringhe JNDI e semplificando la configurazione.

In sintesi, un'applicazione Java EE 7 è composta da vari tipi di componenti (web, EJB, entità, servizi web, ecc.) ognuno eseguito nel container adatto, comunicanti tra loro tramite *injection* o chiamate dirette, il tutto pacchettizzato in moduli (WAR, EAR) e gestito dall'application server. Comprendere l'architettura significa capire il ruolo di ogni tecnologia nei diversi *tier* e come il container orchestri il tutto fornendo servizi trasversali (transazioni, sicurezza, naming, ecc.).

Esempio pratico di codice

Per illustrare l'architettura multi-tier e l'iniezione di dipendenze, consideriamo un semplice scenario: una Servlet nel web container chiama un EJB nel business layer per ottenere un saluto da mostrare all'utente. Abbiamo quindi due componenti: 1. **Un EJB Stateless** che fornisce la logica (ad esempio un metodo `saluta(nome)` che restituisce "Ciao, nome"). 2. **Una Servlet** web che riceve una richiesta HTTP, preleva un parametro `nome`, chiama l'EJB per ottenere il messaggio di saluto e lo stampa in risposta.

Codice EJB (Session Bean Stateless):

```
package esempio;

import javax.ejb.Stateless;

@Stateless  // Dichiarazione di un Session Bean stateless
public class ServizioSalutoEJB {
    public String saluta(String nome) {
        return "Ciao, " + nome + "!";
    }
}
```

Codice Servlet (componente Web):

```java
package esempio;

import javax.inject.Inject;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;

@WebServlet("/saluto")
public class SalutoServlet extends HttpServlet {
    // Iniettiamo l'EJB localmente
    @Inject
    private ServizioSalutoEJB servizio;  // riferimento al session bean
stateless

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String nome = req.getParameter("nome");
        if (nome == null) nome = "Mondo";
        // Chiamata al metodo dell'EJB
        String messaggio = servizio.saluta(nome);
        resp.setContentType("text/plain");
        resp.getWriter().println(messaggio);
    }
}
```

In questo esempio: - Il **Servlet** (nel modulo web) utilizza `@Inject` (CDI) per ottenere l'istanza del `ServizioSalutoEJB` fornita dal container EJB. Non c'è codice di lookup manuale: il container, all'avvio, cerca un EJB di tipo `ServizioSalutoEJB` disponibile e lo assegna. - Il **Session Bean EJB** è annotato `@Stateless` e fornisce un metodo pubblico. Non ha stato conversazionale (non memorizza dati per singolo client), perciò istanze di questo bean possono essere usate concorrentemente per servire molte richieste. - Quando il client invoca l'URL `/saluto?nome=Pippo`: - Il contenitore web (es. Tomcat integrato nell'AS) instrada la richiesta HTTP alla `SalutoServlet`. - La Servlet ottiene il parametro `nome`, lo passa al metodo EJB `saluta(nome)` tramite il riferimento iniettato. - L'EJB viene eseguito nel contenitore EJB, che può applicare transazioni (non necessarie qui) o sicurezza (se definita). - Il risultato "Ciao, Pippo!" viene ritornato alla Servlet, che lo manda come risposta HTTP al client.

Questo dimostra l'interazione tra **Web Container** (Servlet) e **EJB Container** (Session bean) in Java EE 7, tramite l'infrastruttura di *Dependency Injection* (CDI/EJB). L'applicazione potrebbe essere pacchettizzata in un file WAR contenente sia la Servlet che l'EJB (Java EE 7 permette EJB lite nel WAR), oppure come EAR con un modulo web e un modulo EJB separato. In entrambi i casi, il container gestisce la comunicazione tra i componenti e fornisce i servizi di base senza che lo sviluppatore scriva codice di infrastruttura.

Esercizio

Esercizio proposto: Progetta una semplice applicazione Java EE che segua l'architettura a tre livelli: - **Web tier:** una Servlet che funge da controller per le richieste dell'utente (es: form di login, ricerca, ecc.). - **Business tier:** un EJB che implementa la logica di business (es: verifica delle credenziali, elaborazione di dati, calcoli). - **Data tier:** una semplice logica di persistenza simulata (può essere un repository in memoria o l'accesso a un database tramite JPA, se già conosciuto).

Obiettivo: Implementare, ad esempio, un'applicazione di **gestione di note**. La Servlet riceve richieste HTTP per aggiungere una nota, listare le note esistenti, cancellarne una, ecc. La logica di business è in un EJB `NoteService` che mantiene una lista di note (in memoria o su DB se preferisci). Progetta le classi come segue: - `NoteServiceBean` (`@Stateless`) con metodi come `aggiungiNota(String testo)`, `eliminaNota(int id)`, `elencaNote()`. - `NoteServlet` (`@WebServlet`) che inoltra le operazioni dell'utente all'EJB `NoteServiceBean` (iniettato con `@EJB` o `@Inject`) e prepara una risposta (es. sotto forma di pagina HTML semplice o messaggi di conferma in testo). - (Facoltativo) Un'entità JPA `Nota` con campi id e testo, gestita dal `NoteServiceBean` per persistenza reale su database.

Attività: Implementa queste componenti, configura il deployment (ad esempio un file `web.xml` minimo o usa annotazioni), e avvia l'applicazione su un server compatibile (GlassFish, WildFly, Payara, ecc.). Verifica che: 1. L'iniezione dell'EJB nella Servlet funzioni (cioè le chiamate a `NoteServiceBean` producono l'effetto atteso). 2. Le note aggiunte vengono conservate (in memoria o DB) e restituite correttamente alla richiesta successiva. 3. Il tutto avvenga senza scrivere codice di infrastruttura (niente `new NoteServiceBean()` manuale, niente gestione esplicita di connessioni DB nella Servlet, ecc.: delega tutto ai container).

Questo esercizio aiuta a comprendere come strutturare un'app Java EE a più livelli e come i componenti interagiscono attraverso i servizi della piattaforma.

Domande a scelta multipla

1. Quale affermazione descrive meglio Java EE 7?
2. A. Un insieme di specifiche standard per applicazioni enterprise Java, implementate da vari fornitori di server applicativi.
3. B. Un prodotto software specifico di Oracle per sviluppare applicazioni web Java.
4. C. Un framework leggero per applicazioni monolitiche, alternativo a Spring Boot.
5. D. Un toolkit grafico per applicazioni desktop Java.
6. Quale dei seguenti *container* **non** fa parte della piattaforma Java EE 7?
7. A. Web Container.
8. B. EJB Container.
9. C. Applet Container.
10. D. Swing Container (GUI Container).
11. In un'applicazione Java EE multi-tier, dove risiede tipicamente la logica di business?

12. A. Nel client tier, presso l'applicazione desktop o nel browser.
13. B. Nel server tier, in componenti come EJB o CDI bean che interagiscono con il database.
14. C. Nel database tier, attraverso stored procedure e trigger che rimpiazzano completamente la logica applicativa.
15. D. Nel web tier, all'interno delle JSP o Servlets, senza livelli aggiuntivi.

16. Qual è il vantaggio principale di utilizzare l'iniezione di dipendenze (es. `@Inject`, `@EJB`) invece di JNDI lookup nel codice?

17. A. Nessuno, sono equivalenti in funzionalità e verbosità.
18. B. L'iniezione permette codice più pulito e tipizzato, delegando al container la risoluzione delle risorse, mentre JNDI richiede stringhe di lookup e gestione manuale.
19. C. JNDI funziona solo su Linux mentre l'iniezione è multipiattaforma.
20. D. L'uso di JNDI è deprecato e rimosso in Java EE 7.

21. In un file EAR, quale di questi moduli potremmo **non** trovare?

22. A. Un file WAR contenente Servlets e pagine web.
23. B. Un file JAR con EJB session bean.
24. C. Un file .jar di libreria condivisa.
25. D. Un file .exe eseguibile contenente la logica di business.

26. Cosa accade quando un EJB Stateless viene iniettato in una Servlet Java EE?

27. A. Il container EJB crea un'istanza per ogni chiamata e la distrugge immediatamente dopo.
28. B. Il container web rifiuta l'avvio della Servlet perché le Servlets non possono usare EJB.
29. C. Il container risolve la dipendenza alla deploy-time o startup: la Servlet riceve un riferimento proxy a un'istanza del bean gestita dal container EJB.
30. D. Non è possibile iniettare un EJB in una Servlet; bisogna usare lookup JNDI manuale.

31. In Java EE, qual è il ruolo di un **container**?

32. A. Fornire servizi di runtime (come gestione thread, transazioni, sicurezza) ai componenti applicativi in esso eseguiti.
33. B. Impacchettare l'applicazione in un file deployabile.
34. C. Assicurare che il garbage collector non elimini i bean.
35. D. Tradurre il bytecode Java in codice macchina nativo.

36. Quale tipo di componente **non** è tipicamente eseguito nel Web Container?

37. A. JSP (JavaServer Page).
38. B. Servlet HTTP.
39. C. JSF backing bean (gestito come CDI bean nel web container).

40. D. Message-Driven Bean (MDB).

41. Quale caratteristica è **vera** riguardo agli *Enterprise JavaBeans (EJB)* rispetto ai *CDI bean*?

42. A. Gli EJB supportano transazioni dichiarative out-of-the-box, mentre i CDI bean di default no (richiedono `@Transactional` o gestione esplicita).
43. B. I CDI bean possono essere richiamati remotamente su un altro server, mentre gli EJB no.
44. C. Gli EJB non possono usare l'iniezione CDI, mentre i CDI bean possono iniettare EJB liberamente.

45. D. Un EJB deve sempre essere stateful, mentre i CDI bean sono sempre stateless.

46. In un'applicazione Java EE, come viene tipicamente **gestita la persistenza** degli oggetti nel database?

    - A. Tramite oggetti Entity JPA gestiti da un Persistence Context, con operazioni mediate dall'EntityManager (fornito ad esempio da un EJB o CDI bean).
    - B. Direttamente dalla Servlet mediante codici SQL JDBC in-line per ogni richiesta.
    - C. Attraverso file di testo scritti su disco dal Web Container.
    - D. Mediante l'uso di variabili statiche che mantengono i dati anche dopo il riavvio del server (persistenza in-memory).

Persistenza con JPA e Bean Validation (Manage Persistence using JPA Entities and BeanValidation)

Teoria

JPA (Java Persistence API) è la specifica standard Java EE per la mappatura *object-relational* e l'accesso ai database relazionali. In Java EE 7, JPA è versione 2.1. Tramite JPA, gli sviluppatori possono definire classi **Entity** che rappresentano tabelle del database, e il container fornisce automaticamente funzionalità di persistenza: inserimento, aggiornamento, cancellazione e query sugli oggetti vengono tradotti in operazioni SQL verso il database.

Le principali caratteristiche della persistenza JPA includono:

- **Entity e mapping ORM:** Un'**entità** JPA è una classe POJO annotata con `@Entity` che rappresenta una tabella. I suoi campi tipicamente rappresentano colonne (annotati con `@Column`, o dedotti dal nome se non specificato). Si può specificare una chiave primaria con `@Id` e strategie di generazione chiavi con `@GeneratedValue` (ad esempio `GenerationType.AUTO`, `IDENTITY`, `SEQUENCE`, `TABLE`). Si possono definire relazioni tra entità con annotazioni come `@OneToOne`, `@OneToMany`, `@ManyToOne`, `@ManyToMany`, e specificare il *fetch* (eager/lazy) e le *cascade* (propagazione di operazioni persist, merge, remove).
- **Persistence Unit e EntityManager:** Le entità sono gestite all'interno di un *Persistence Unit*, definito in `persistence.xml`. In un'app Java EE, il container configura la *Persistence Unit* (ad esempio creando un `EntityManagerFactory`) e fornisce un **EntityManager** iniettabile (con `@PersistenceContext`) nelle componenti EJB/CDI. L'`EntityManager` è l'API principale per interagire col datastore: consente di *persistere* un'entità nuova (metodo `persist()`), *rimuovere*

un'entità (metodo `remove()`), *trovare* entità per chiave primaria (`find()`), *unire* uno stato detached nell'unità di persistenza (`merge()`), *eseguire query* (con SQL nativo o JPQL). L'EntityManager opera all'interno di un **contesto di persistenza**, che traccia quali entità sono gestite (attaccate) e sincronizza le modifiche al database quando necessario (tipicamente al *commit* della transazione o manualmente con `flush()`).

- **Ciclo di vita delle entità:** Un'entità JPA può essere in stati: *New/Transient* (oggetto Java non ancora associato al DB), *Managed* (associato a un contesto di persistenza, quindi sincronizzato col DB), *Detached* (disaccoppiato dopo che il contesto è chiuso o l'entità è esplicitamente detach, ovvero non monitorato), *Removed* (marcato per rimozione). Ad esempio, quando chiami `em.persist(obj)` su un nuovo oggetto, esso diventa *managed* e verrà inserito nel DB; `em.remove(obj)` marca l'entità gestita per eliminazione; `em.merge(objDetached)` copia lo stato di un oggetto detached in una nuova entità managed (o la attacca di nuovo se esiste già in contesto). La sincronizzazione col DB avviene su commit transazione o flush esplicito.
- **Transazioni:** L'EntityManager JPA lavora normalmente in un contesto transazionale. In Java EE, se usato dentro un EJB con transazioni gestite dal container (CMT) o in un metodo annotato `@Transactional`, l'`EntityManager` partecipa automaticamente alla transazione JTA corrente. Operazioni come persist, remove, etc. vengono effettivamente applicate al DB al commit. Se la transazione viene fatta rollback, anche le operazioni JPA vengono annullate. In modalità transazioni gestite dall'applicazione (BMT), lo sviluppatore deve esplicitamente demarcare i confini (ad es. usando `UserTransaction`).
- **Locking e concorrenza:** JPA 2.1 supporta meccanismi di locking ottimistico (e pessimistico). Il locking **ottimistico** si ottiene usando una versione (@Version) sulle entità: ogni volta che un record è aggiornato, il valore versione cambia; se due transazioni tentano di aggiornare la stessa entità, la seconda rileverà (al commit) che la versione è cambiata e solleverà una `OptimisticLockException`, evitando inconsistenze. Il locking **pessimistico** può essere invocato con metodi come `em.find(..., LockModeType.PESSIMISTIC_WRITE)`, che tipicamente emette un `SELECT ... FOR UPDATE` sul DB, bloccando la riga. Il supporto di default è ottimistico in JPA (versioning) ed è buona pratica usarlo nelle entità con potenziali concorrenze.
- **Conversioni e tipi personalizzati:** JPA 2.1 introduce gli **Attribute Converter** (`@Converter`), che permettono di convertire automaticamente tipi non nativi in un tipo supportato dal database. Ad esempio, si può creare un converter per memorizzare un `List<String>` come una stringa delimitata nel DB, oppure cifrare/decifrare automaticamente un campo. I converter sono applicati trasparentemente dal provider JPA durante l'ORM.
- **Bean Validation (JSR 303/349):** Java EE 7 integra Bean Validation 1.1 (JSR 349), che consente di dichiarare regole di validazione sui campi delle classi (spesso entità) tramite annotazioni come `@NotNull`, `@Size`, `@Pattern`, ecc. L'integrazione tra JPA e Bean Validation è automatica: se una Persistence Unit vede le librerie di Bean Validation nel classpath, il container valida le entità automaticamente agli eventi di lifecycle predefiniti (prima di un insert, update o delete). Ad esempio, se un'entità ha un campo `@NotNull` e provi a `persist()` un'istanza con quel campo nullo, il provider JPA lancerà una `ConstraintViolationException` e impedirà il salvataggio. Ciò garantisce che i dati persistenti rispettino i vincoli dichiarativi. Bean Validation è usata anche in altri contesti (es. validazione input in JSF), ma in JPA riduce la necessità di scrivere codice di validazione manuale.
- **Generazione di chiavi primarie:** Come accennato, JPA supporta diverse strategie per generare le primary key delle entità. `@GeneratedValue` con `strategy = GenerationType.AUTO` lascia al provider la scelta (di solito sequence su DB che le supportano, oppure tabelle id dedicata, ecc.). `IDENTITY` delega al campo auto-increment del database (inserimento avviene con chiave generata

dal DB). `SEQUENCE` usa una sequenza DB (spesso con `@SequenceGenerator` definito). `TABLE` usa una tabella dedicata a mantenere l'ultimo valore (meno usato per performance). L'uso di sequence è preferibile su DB che la supportano, per efficienza e controllo.

- **JPQL e query:** Oltre al *Criteria API* (costruzione di query in modo typesafe via API Java), JPA include il linguaggio JPQL (Java Persistence Query Language), simile a SQL ma orientato agli oggetti (usa nomi di entità e proprietà invece di tabelle e colonne). Ad esempio: `SELECT e FROM Employee e WHERE e.department.id = :depId` è una JPQL che seleziona oggetti `Employee` il cui dipartimento ha un certo id. Le query JPQL sono eseguite tramite `EntityManager.createQuery(...)` e accettano parametri nominati o posizionali. Possono anche eseguire update/delete in blocco (con `@Modifying` query in JPA 2.1, solitamente usato con `EntityManager.createQuery("UPDATE ...")`). Inoltre JPA supporta query **native SQL** se necessario (con `createNativeQuery`).
- **Entity Graphs (2.1):** JPA 2.1 aggiunge la possibilità di definire *Entity Graph* per controllare il fetch di relazioni (caricamento di grafo di oggetti) in modo dichiarativo, evitando di usare `join fetch` in JPQL o di accedere pigramente alle relazioni (che può causare *LazyInitializationException* fuori dal contesto). Si definisce un EntityGraph (ad es. annotazioni o API) e poi si può usare in find o query per specificare quali associazioni caricare.

In sintesi, JPA fornisce un **sistema ORM** completo integrato con Java EE. Lo sviluppatore definisce entità e relazioni, il container gestisce l'apertura di connessioni al database, la gestione di transazioni, il caching di primo livello (nel contesto di persistenza), il flush dei cambiamenti e l'aderenza ai vincoli dichiarati (grazie a Bean Validation integrata). Saper usare JPA significa saper: - Definire correttamente le entità e i mapping (incluso ereditarietà, relazioni e chiavi). - Utilizzare l'EntityManager per le operazioni CRUD all'interno di transazioni. - Scrivere query JPQL per recuperare i dati necessari. - Applicare Bean Validation per garantire consistenza dei dati (sia lato back-end che eventualmente propagando gli errori al front-end). - Gestire eventuali problemi di concorrenza (usando versioning) e performance (ottimizzare lazy/eager fetch, caching di secondo livello se configurato, etc.).

Esempio pratico di codice

Supponiamo di voler modellare e persistere dati di una semplice applicazione di biblioteca che gestisce *Libri* e *Autori*. Un **Libro** ha un titolo, un autore e un anno di pubblicazione; un **Autore** ha un nome e può aver scritto molti libri. Utilizzeremo JPA per rappresentare questa relazione (Many-to-One da Libro ad Autore) e Bean Validation per assicurare alcuni vincoli, ad esempio che il titolo non sia vuoto.

Definiamo due entità JPA, **Libro** e **Autore**, con relative annotazioni:

```
@Entity
@Table(name="libri")
public class Libro {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)

@NotNull                        // Bean Validation: il titolo non può essere
null
```

```
    @Size(min=1, max=200)           // Bean Validation: lunghezza 1-200
caratteri
    private String titolo;

    @ManyToOne(optional = false)    // Molti Libri per un Autore, non opzionale
(deve avere un autore)
    @JoinColumn(name="autore_id")   // Colonna di join verso la tabella autori
    private Autore autore;

    private int annoPubblicazione;

    // Costruttori, getter, setter...
}
```

```
@Entity
@Table(name="autori")
public class Autore {
    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(length = 100, unique = true)
    @NotNull
    @Size(min=1, max=100)
    private String nome;

    // Esempio di relazione bidirezionale: un autore ha più libri
    @OneToMany(mappedBy = "autore", cascade = CascadeType.ALL)
    private List<Libro> libri = new ArrayList<>();

    // Costruttori, getter, setter...
}
```

Nella classe `Libro`: - `@Table(name="libri")` specifica il nome della tabella (altrimenti di default sarebbe "Libro"). - `@Column(nullable=false)` e `@NotNull` duplicano concettualmente il vincolo di non null; JPA userà `nullable=false` per lo schema DB e Bean Validation userà `@NotNull` per validare l'istanza a runtime (ad esempio prima di un persist). - `@Size(min=1)` sul titolo assicura che non sia stringa vuota (anche questo validato a runtime). - La relazione `@ManyToOne` a `Autore` implica una colonna `autore_id` come FK. `optional=false` significa che non sono permessi libri senza autore (equivale a un NOT NULL sul FK). - In `Autore`, la relazione inversa `@OneToMany(mappedBy="autore")` non ha una colonna propria (usa la colonna in Libro) e Cascade ALL fa sì che operazioni sull'autore si propaghino ai libri (es. se salvo un autore nuovo con una lista di libri dentro, JPA salva anche i libri).

Ora, immaginiamo di voler **salvare** un nuovo libro nel database tramite JPA e verificare la validazione. Lo scenario: abbiamo già un autore esistente, e creiamo un libro associato a quell'autore.

Esempio di utilizzo in un Session Bean EJB (o metodo CDI `@Transactional`):

```
@Stateless
public class BibliotecaService {
    @PersistenceContext(unitName="BibliotecaPU")
    private EntityManager em;

    public Libro creaNuovoLibro(Long idAutore, String titolo, int anno) {
        Autore autore = em.find(Autore.class, idAutore);
        if (autore == null) throw new IllegalArgumentException("Autore non
trovato");
        Libro libro = new Libro();
        libro.setTitolo(titolo);
        libro.setAnnoPubblicazione(anno);
        libro.setAutore(autore);
        // Aggiunge il libro alla lista dell'autore (coerenza lato oggetto)
        autore.getLibri().add(libro);
        em.persist(libro);  // Persist del nuovo libro
        return libro;
    }
}
```

Spiegazione: - Viene iniettato un `EntityManager` associato a una Persistence Unit "BibliotecaPU" definita in persistence.xml. - Il metodo `creaNuovoLibro` recupera l'entità Autore esistente dal DB (`find`). - Crea un oggetto Libro, imposta i campi. **Nota:** se `titolo` fosse null o stringa vuota, al momento di `em.persist(libro)` il container validerebbe l'entità e, trovando una violazione di `@NotNull` o `@Size`, lancerebbe `ConstraintViolationException` interrompendo l'operazione. Ciò impedisce di salvare dati non validi. - Si aggiorna la relazione bidirezionale aggiungendo il libro alla lista dell'autore (coerenza in memoria). - Si chiama `em.persist(libro)`. Essendo dentro un EJB con CMT (Container-Managed Transaction), il container aprirà una transazione, il libro diventa *managed* nel contesto di persistenza. Alla fine del metodo (che di default implica commit se nessuna eccezione), il container effettua il commit: JPA esegue l'`INSERT` nella tabella libri e l'`UPDATE` della tabella autori se necessario (in questo caso potrebbe non servire un update perché la relazione è gestita dal lato libro con la FK). Se il database supporta vincoli e sequenze, `IDENTITY` genererà l'ID. - Il metodo ritorna il Libro persistito (che a questo punto ha un id assegnato).

Query JPQL esempio: per recuperare tutti i libri di un certo autore:

```
List<Libro> libriAutore = em.createQuery(
    "SELECT l FROM Libro l WHERE l.autore.nome = :nomeAutore",
    Libro.class)
  .setParameter("nomeAutore", "Italo Calvino")
  .getResultList();
```

Questa JPQL trova tutti i `Libro` il cui autore ha nome "Italo Calvino". Notare che usa i nomi delle entità (`Libro`, e naviga l'associazione `l.autore.nome`). Il provider tradurrà questo in SQL con join tra libri e autori.

Esercizio

Esercizio proposto: Creare un piccolo modello di dati con JPA e sperimentare le operazioni CRUD e la Bean Validation: - **Modello:** Immagina di dover gestire un registro di **studenti** e i loro **corsi** iscritti. Crea le entità `Studente` e `Corso` con relazione molti-a-molti (molti studenti possono iscriversi a molti corsi). Usa un'entità associativa (es. `Iscrizione`) oppure la relazione many-to-many diretta fornita da JPA con `@ManyToMany` e una tabella di join automatica. - **Vincoli:** Applica Bean Validation, ad esempio: - Lo `Studente` deve avere un nome non vuoto e un'età >= 18. - Il `Corso` deve avere un titolo non vuoto e un codice corso di formattazione specifica (puoi usare `@Pattern` per richiedere ad esempio "COR-XXX"). - **Persistence Unit:** Configura `persistence.xml` (puoi usare un database in memoria come H2 per test rapido, o PostgreSQL/MySQL se disponibili). Assicurati di includere le dipendenze JPA (ad esempio EclipseLink o Hibernate, a seconda del server). - **Operazioni:** Scrivi un EJB o un bean CDI `@Transactional` con metodi per: 1. Creare un nuovo Studente. 2. Creare un nuovo Corso. 3. Iscrivere uno studente a un corso (gestendo la relazione many-to-many). 4. Recuperare tutti i corsi a cui è iscritto uno studente (query). 5. Rimuovere l'iscrizione di uno studente (operazione su relazione). - **Test:** Prova a: - Creare entità valide e salvarle (dovrebbe funzionare). - Provare a salvare entità non valide (es. studente minorenne o nome vuoto) e verificare che JPA/Bean Validation lanci eccezioni impedendo il commit. - Eseguire query JPQL per interrogare le iscrizioni (es. "trova tutti gli studenti iscritti al corso X"). - Rimuovere una entità (ad esempio un corso) e vedere se si propagano cascata le cancellazioni delle iscrizioni (a seconda di come configuri Cascade nell'associazione).

Questo esercizio ti farà toccare con mano: - La definizione di entità e mapping di relazioni. - L'utilizzo di `EntityManager` per persist/find/query. - L'effetto di Bean Validation in JPA (violazioni di vincoli). - La gestione delle associazioni many-to-many e delle cascata. - L'ambiente transazionale di JPA in Java EE.

Domande a scelta multipla

1. In JPA, qual è il ruolo dell'**EntityManager**?
2. A. Gestisce le operazioni CRUD sulle entità (persist, merge, remove, find) e il contesto di persistenza, sincronizzando gli oggetti Java con il database.
3. B. È responsabile dell'esecuzione delle query SQL pure al posto dello sviluppatore.
4. C. Fornisce l'interfaccia grafica per modificare i record del database.

5. D. Si occupa di creare automaticamente schemi e tabelle senza configurazione.

6. Si consideri un'entità `Utente` con campo `@NotNull String email`. Cosa accade se proviamo a `persist()` un oggetto Utente con `email == null` avendo Bean Validation attivo?

7. A. L'operazione riesce, ma il valore null viene convertito in stringa vuota automaticamente.
8. B. Il provider JPA lancia una `ConstraintViolationException` prima di eseguire l'inserimento, prevenendo la violazione del vincolo.
9. C. Il database respinge l'inserimento con un errore di colonna null, ma JPA non può intercettarlo.

10. D. JPA ignora le annotazioni di Bean Validation a meno che non si chiami manualmente `Validator.validate()`.

11. Quale annotazione JPA useresti per definire la relazione "Un Dipartimento ha molti Impiegati" sapendo che in classe `Impiegato` c'è un riferimento a `Dipartimento`?

12. A. Su `Impiegato`: `@ManyToOne` verso `Dipartimento`; su `Dipartimento`: `@OneToMany(mappedBy="dipartimento")`.
13. B. Su `Impiegato`: `@OneToMany`; su `Dipartimento`: `@ManyToOne(mappedBy="impiegati")`.
14. C. Su `Impiegato`: `@ManyToMany`; su `Dipartimento`: `@ManyToMany`.

15. D. Nessuna annotazione, JPA deduce le relazioni automaticamente dai nomi dei campi.

16. In JPA, qual è la differenza tra **Lazy Loading** ed **Eager Loading** in una relazione?

17. A. Lazy (pigro) carica l'entità correlata solo quando è effettivamente accesso il suo getter, mentre Eager (eager) la carica subito insieme all'entità principale (tipicamente via join) al momento del fetch.
18. B. Lazy usa cache in memoria e Eager no.
19. C. Lazy si applica solo alle collezioni, Eager solo alle singole entità.

20. D. Non c'è differenza, sono sinonimi in JPA.

21. Quale dei seguenti *NON* è un vantaggio di usare JPA rispetto al JDBC diretto?

22. A. Astrazione dell'SQL in un modello a oggetti e query JPQL più espressive.
23. B. Gestione automatica del ciclo di vita degli oggetti e caching di primo livello.
24. C. Controllo manuale di ogni singola query SQL eseguita, per avere pieno controllo su performance (JPA invece genera SQL e non permette ottimizzazioni).

25. D. Integrazione con transazioni JTA e Bean Validation, riducendo il codice boilerplate di gestione errori.

26. In JPA, come definiresti una chiave primaria composta (multi-colonna)?

27. A. Non è possibile in JPA, si può avere solo una colonna @Id.
28. B. Usando `@IdClass` o `@EmbeddedId` nella classe entità per rappresentare le più colonne della PK.
29. C. Creando due campi annotati @Id nella stessa entità (JPA permette multipli @Id).

30. D. Definendo un indice sul database che unisce due colonne.

31. Supponiamo di avere entità `Ordine` e `Articolo` in relazione molti-a-molti. Qual è una strategia valida per modellarla in JPA?

32. A. Usare `@ManyToMany` su entrambe le entità con un `mappedBy` da un lato, lasciando che JPA crei automaticamente una tabella di join intermedia.
33. B. Non supportando JPA le relazioni molti-a-molti, è necessario usare JDBC manuale.
34. C. Usare due relazioni uno-a-molti opposte con una terza entità `OrdineArticolo` (chiave composta) che rappresenta la join (strategia many-to-many con entità associativa per aggiungere attributi aggiuntivi alla relazione).

35. D. Avere una lista di ID di Articolo in Ordine e una lista di ID di Ordine in Articolo, senza relazioni JPA annotate.

36. Cosa fa l'annotazione `@Version` su un campo di un'entità JPA?

37. A. Indica che quel campo contiene la versione (numero o timestamp) dell'entità per il controllo di **locking ottimistico**, incrementato ad ogni aggiornamento.
38. B. Serve per versionare lo schema del database tramite JPA.
39. C. Esegue il backup della riga precedente su un'altra tabella prima di ogni update.

40. D. Permette di mantenere più versioni dello stesso oggetto in database.

41. Quale affermazione è vera riguardo alle **Named Queries** in JPA?

42. A. Sono definizioni di query JPQL predefinite, annotate sulle entità (`@NamedQuery`), che possono essere richiamate per nome a runtime.
43. B. Si riferiscono alle query SQL native che JPA non supporta.
44. C. Non supportano parametri, a differenza di createQuery.

45. D. Devono essere definire in un file esterno XML e non tramite annotazioni.

46. Bean Validation in JPA: quando vengono applicate le annotazioni di validazione sulle entità?

    - A. Solo quando si chiama manualmente un `Validator`.
    - B. Automaticamente durante le operazioni di persist, update e remove, se la implementazione BV è presente.
    - C. Al momento della generazione dello schema del database.
    - D. Mai, JPA non interagisce con Bean Validation.

Logica di Business con EJB (Implement Business Logic by Using EJBs)

Teoria

Gli **Enterprise JavaBeans (EJB)** sono componenti server-side dedicati alla logica di business, parte fondamentale di Java EE. In versione 3.x (Java EE 7 adotta EJB 3.2), gli EJB sono semplici POJO annotati che però vengono gestiti da un apposito container EJB, il quale fornisce automaticamente servizi robusti come *gestione transazioni*, *sicurezza declarativa*, *pooling di istanze*, *gestione concorrenza*, *invocazioni remote*, *timer di sistema*, etc. Questo consente allo sviluppatore di concentrarsi sulla logica applicativa, delegando al container compiti complessi.

Tipi di EJB: Java EE 7 definisce tre principali tipi di session bean EJB: - **Stateless Session Bean:** Non mantiene stato conversazionale tra le chiamate dei client. Ogni metodo è eseguito come se l'istanza fosse uguale per tutti (anche se il container può usare istanze diverse in pool). Ideale per servizi riutilizzabili, thread-safe per natura (nessun stato specifico del client). Annotato con `@Stateless`. - **Stateful Session Bean:** Mantiene uno stato legato al client che lo sta usando, attraverso più invocazioni/metodi. Il container garantisce che ogni client abbia la “sua” istanza, permettendo di mantenere informazioni tra chiamate (es: il

contenuto di un carrello di acquisti durante una sessione). Può essere passivato (serializzato su disco) dal container se inattivo, per liberare risorse, e attivato nuovamente su richiesta. Annotato `@Stateful`. - **Singleton Session Bean:** Unico per applicazione, una sola istanza condivisa tra tutti i client (come un service globale). Tipicamente usato per cache applicative, configurazioni condivise, o coordinazione globale. Può essere `@Singleton` (per default accesso concorrente serializzato dal container a livello di metodi, configurabile con `@Lock`). - (Oltre ai session bean, negli EJB ci sono anche i **Message-Driven Bean (MDB)** che sono speciali EJB ascoltatori di messaggi JMS. Li trattiamo nella sezione JMS separatamente.)

Ciclo di vita e callback: Il container EJB gestisce il ciclo di vita dei session bean: - Per **Stateless**: ne crea un pool all'avvio (numero variabile a seconda del carico), li inizializza chiamando eventuale `@PostConstruct`, poi ad ogni richiesta di un metodo EJB da parte di un client assegna un'istanza dal pool. Dopo l'uso, l'istanza può tornare nel pool per essere riutilizzata per un'altra richiesta. Il container può decidere di distruggerla chiamando `@PreDestroy` per gestire risorse (es: chiudere connessioni). - Per **Stateful**: creato su richiesta (es. al primo lookup/injection per un cliente), `@PostConstruct` chiamato, poi rimane associato al client. Può passare in metodo `@PrePassivate` prima di essere passivato (salvato su disco) e `@PostActivate` quando viene riattivato in memoria. Quando il client termina la conversazione o invoca un metodo di fine (`@Remove`), il container chiama `@PreDestroy` e distrugge l'istanza. - Per **Singleton**: creato tipicamente all'avvio applicazione (eager se specificato, altrimenti al primo utilizzo lazy), e distrutto solo allo shutdown. Ha `@PostConstruct` e `@PreDestroy`.

Questi *metodi callback* (`@PostConstruct`, `@PreDestroy`, `@PrePassivate`, `@PostActivate`) permettono di eseguire codice di inizializzazione (es: aprire risorse) o pulizia (chiudere risorse).

Business methods: I metodi pubblici dell'EJB (non static, non final) sono i metodi di business invocabili dal client. Nei Session Bean, per default, tutti i metodi pubblici sono considerati remotamente invocabili (nel caso di interfacce remote) o localmente se iniettati. È possibile comunque controllare quali esporre, ad esempio con `@Remote` e `@Local` su interfacce separate, oppure limitare l'accesso con ruoli (vedi sicurezza).

Transazioni: Una delle funzionalità chiave degli EJB è la gestione automatica delle transazioni (**Container-Managed Transactions, CMT**). Per default, ogni metodo di un EJB `@Stateless` o `@Stateful` viene eseguito dentro una transazione JTA. Il default transaction attribute è *Required* (il metodo si esegue in una transazione: se chiamato da un contesto transazionale esistente, la unisce, altrimenti ne avvia una nuova e la commit/rollback alla fine). Lo sviluppatore può controllare il comportamento con l'annotazione `@TransactionAttribute`, scegliendo tra: - REQUIRED (default): join o crea transazione. - REQUIRES_NEW: sospende quella esistente e ne avvia una nuova. - MANDATORY: richiede già una transazione esistente, se non c'è lancia errore. - SUPPORTS: se c'è una transazione la usa, altrimenti esegue senza. - NOT_SUPPORTED: se c'è una transazione la sospende ed esegue fuori transazione. - NEVER: errore se invocato dentro una transazione.

Alternativamente, i EJB possono usare **Bean-Managed Transactions (BMT)**: l'EJB demarca manualmente transazioni con `UserTransaction` (solo per EJB che lo supportano, tipicamente `@Stateless` e `@Singleton` con certain settings). In BMT, il dev controlla commit/rollback.

Sicurezza declarativa: Con EJB è facile dichiarare controlli di sicurezza a livello metodo usando `@RolesAllowed`, `@PermitAll`, `@DenyAll` sulle classi o metodi. Il container EJB verificherà l'utente

chiamante (principal) e consentirà o negherà l'esecuzione. (La sicurezza in dettaglio è trattata in sezione dedicata.)

Invocazione remota vs locale: EJB originariamente nascono con supporto a invocazioni remote (RMI over IIOP). In EJB 3.x, per esporre un EJB su remoto (ad esempio chiamabile da un client stand-alone), si annota con `@Remote` un'interfaccia che il bean implementa. Se invece è destinato solo a chiamate all'interno della stessa JVM (stessa applicazione server), basta `@Local` o nessuna annotazione (per default, se non specificato, l'interfaccia business è locale). I container generano i proxy adeguati. In Java EE 7, con l'uso intensivo di RESTful web service, l'invocazione remota EJB è meno usata se non in architetture legacy o particolarmente enterprise multi-tier.

Asynchronous EJB: EJB 3.2 permette di eseguire metodi in modo asincrono usando l'annotazione `@Asynchronous` (su classe o metodo). Se un client chiama un metodo `@Asynchronous`, il container restituisce immediatamente al chiamante (se il metodo non ha un valore di ritorno o ritorna `Future`), ed esegue il metodo in un thread separato. Questo è utile per operazioni non critiche da eseguire in background o per migliorare la reattività. Ad esempio un EJB potrebbe avere:

```
@Asynchronous
public Future<String> operazioneLenta() { ... }
```

Il client otterrà subito un `Future` e potrà controllare poi il risultato. Oppure se il metodo è `void` `@Asynchronous`, viene semplicemente lanciato in background.

EJB Timer Service: Il container EJB fornisce un servizio di *scheduler* per operazioni temporizzate. Esistono **timer programmatici** (via `TimerService.createTimer(...)`) e **timer automatici** via annotazione `@Schedule`. Ad esempio, un Singleton bean potrebbe avere:

```
@Schedule(hour="3", minute="0", second="0", persistent=false)
public void manutenzioneNotturna() {
    // codice eseguito ogni giorno alle 3:00
}
```

Ciò permette di eseguire compiti pianificati (cron-like). I timer EJB possono essere persistenti (riavviati dopo restart) o meno. Ci sono anche annotazioni come `@Timeout` per gestire scadenze di timer programmatici. L'EJB Timer Service è il meccanismo standard per *cron job* in Java EE, integrato con transazioni e cluster (persistendo i timer in DB se configurato).

Intercettori (Interceptors): Gli EJB supportano la logica AOP tramite **interceptor**. Si possono definire classi intercettori con `@Interceptor` che contengono un metodo annotato `@AroundInvoke` il quale riceve un `InvocationContext`. Questo metodo può eseguire logica prima/dopo l'invocazione del metodo business e decidere se proseguire (`ctx.proceed()`). Ad esempio:

```java
@Interceptor
@LoggingInterceptorBinding  // un binding custom definito con
@InterceptorBinding
public class LoggingInterceptor {
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception {
        System.out.println("Entering: " + ctx.getMethod().getName());
        Object result = ctx.proceed();
        System.out.println("Exiting: " + ctx.getMethod().getName());
        return result;
    }
}
```

Applicando poi `@LoggingInterceptorBinding` su un EJB (o metodo), il container EJB inserirà l'interceptor nel ciclo di chiamata. Gli intercettori servono per aspetti trasversali come logging, auditing, sicurezza personalizzata, etc., senza contaminare la logica business. Possono essere applicati a livello classe o metodo EJB. Esiste anche `@AroundTimeout` per intercettare metodi di timeout (timer).

EJB e CDI: In Java EE 7, EJB e CDI sono integrati. Un EJB è anche un bean CDI (segue le regole di injection di CDI, può produrre eventi, essere @Injected altrove). Un bean CDI non è di per sé un EJB a meno che non abbia un'annotazione EJB, quindi non avrà transazioni automatiche o security di EJB, ma può utilizzare tali servizi via CDI (es. usando `@Transactional`). Scegliere tra EJB e CDI puro dipende dai bisogni: EJB per servizi heavy-duty con transazioni, sicurezza e remoting; CDI bean per componenti leggeri e granulari. Spesso coesistono: ad esempio, un backing bean JSF può essere un CDI bean che chiama un EJB per operazioni transazionali.

In conclusione, **implementare la logica di business con EJB** significa creare session bean appropriati (stateless per servizi generici e ad alta scalabilità, stateful per scenari conversazionali, singleton per servizi globali) e sfruttare i servizi container: - Controllare il comportamento transazionale (CMT o BMT). - Usare `@Asynchronous` per operazioni in parallelo se necessario. - Pianificare operazioni periodiche con EJB Timer (`@Schedule`). - Applicare intercettori per cross-cutting concerns come logging. - Gestire eventuali eccezioni: EJB distingue tra *checked exceptions* applicative (passano al client così come sono se dichiarate) e *runtime exceptions* non dichiarate che di default causano rollback transazione e vengono rimappate in `EJBException` se propagate al client. - Capire il ciclo di vita per usare correttamente risorse: ad esempio rilasciarle in `@PreDestroy`, inizializzarle in `@PostConstruct`, etc.

Esempio pratico di codice

Consideriamo un caso pratico: un sistema di e-commerce in cui abbiamo un carrello acquisti per ogni utente. Possiamo implementare la gestione del carrello con un **Stateful EJB**, che mantiene nel proprio stato la lista degli articoli selezionati dall'utente. Inoltre, possiamo avere un **Stateless EJB** che calcola il totale dell'ordine (prezzi, tasse, spedizione). Infine, immaginiamo di volere loggare ogni chiamata ai metodi del carrello per monitoraggio: useremo un **Interceptor** per logging.

Stateful EJB - Carrello:

```java
import javax.ejb.Stateful;
import javax.annotation.PreDestroy;
import java.util.ArrayList;
import java.util.List;

@Stateful
@Logging // interceptor custom applicato (definito altrove)
public class CarrelloBean implements Carrello {  // Carrello è un'interfaccia
@Local con i metodi sottostanti
    private List<String> articoli = new ArrayList<>();

    public void aggiungiArticolo(String prodotto) {
        articoli.add(prodotto);
    }

    public void rimuoviArticolo(String prodotto) {
        articoli.remove(prodotto);
    }

    public List<String> getArticoli() {
        return articoli;
    }

    @PreDestroy
    private void svuota() {
        // Metodo invocato prima che il bean sia distrutto (es. fine sessione)
        articoli.clear();
    }
}
```

In questo bean: - È `@Stateful`, quindi un'istanza per ogni utente che inizia una sessione di shopping. - Mantiene lo stato (lista di articoli). - Ha metodi per aggiungere/rimuovere articoli. - Un metodo `getArticoli` che restituisce il contenuto (potrebbe ad esempio essere chiamato al momento del checkout). - Nel metodo `@PreDestroy` svuota la lista (non strettamente necessario, ma illustrativo per rilasciare risorse; in questo caso, serve a mostrare dove si farebbe cleanup, es: salvare stato su DB se necessario).

Stateless EJB - Calcolatore Ordine:

```java
import javax.ejb.Stateless;
import javax.ejb.EJB;
import java.math.BigDecimal;
import java.util.List;

@Stateless
```

```
public class CalcolatoreOrdineBean {
    @EJB
    private CatalogoPrezziBean
catalogo;  // un altro EJB stateless che fornisce i prezzi correnti

    public BigDecimal calcolaTotale(List<String> articoli) {
        BigDecimal totale = BigDecimal.ZERO;
        for(String prod : articoli) {
            BigDecimal prezzo = catalogo.getPrezzo(prod);
            totale = totale.add(prezzo);
        }
        // Aggiungi IVA 22%
        return totale.multiply(new BigDecimal("1.22"));
    }
}
```

Qui: - `@Stateless` perché il calcolo non necessita di stato tra chiamate. Ogni richiesta può essere servita da qualsiasi istanza nel pool. - Iniettiamo un altro EJB `CatalogoPrezziBean` (immagina fornisca prezzi per codice prodotto). - Il metodo `calcolaTotale` scorre la lista di articoli e calcola la somma prezzi più IVA. - Notare che l'iniezione di un EJB dentro un altro (`@EJB private CatalogoPrezziBean catalogo`) è del tutto legale e gestita dal container.

Interceptor per Logging: Supponiamo di avere definito un interceptor binding:

```
@InterceptorBinding
@Target({ METHOD, TYPE })
@Retention(RUNTIME)
public @interface Logging {}
```

E l'interceptor:

```
import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;
import javax.interceptor.Interceptor;

@Logging
@Interceptor
public class LoggingInterceptor {
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception {
        System.out.println(">> Entrando in " +
ctx.getTarget().getClass().getSimpleName() +
                           "." + ctx.getMethod().getName());
        try {
            return ctx.proceed();
```

```
        } finally {
            System.out.println("<< Uscendo da " + ctx.getMethod().getName());
        }
    }
}
```

Abbiamo applicato `@Logging` sul `CarrelloBean` (vedi annotazione su classe CarrelloBean). Ciò significa che ogni chiamata ai metodi di CarrelloBean passerà prima per `LoggingInterceptor`. Nell'output console vedremo messaggi prima e dopo l'esecuzione reale del metodo (utile per debug o audit).

Scenario d'uso: Un utente attraverso l'interfaccia web avvia una sessione e ottiene un riferimento al `CarrelloBean` stateful (injection in un servlet o retrieval via JNDI). Aggiunge alcuni articoli:

```
carrello.aggiungiArticolo("SKU-001");
carrello.aggiungiArticolo("SKU-002");
```

Quando è il momento di calcolare il totale:

```
List<String> articoli = carrello.getArticoli();
BigDecimal totale = calcolatore.calcolaTotale(articoli);
```

Qui `calcolatore` è un riferimento a `CalcolatoreOrdineBean` stateless (iniettato magari nello stesso componente web). Questo chiama il CatalogoPrezzi per ogni SKU e restituisce il totale con IVA.

Infine, il utente conferma l'ordine: possiamo convertire il contenuto del Carrello in un ordine e magari terminare la sessione EJB stateful (invocando un metodo marcato `@Remove` in CarrelloBean, non mostrato sopra, che potrebbe essere definito per indicare al container di distruggere il bean dopo l'uso).

Transazionalità: Nel nostro esempio semplice, non abbiamo parlato di transazioni, ma se queste operazioni coinvolgessero DB (ad esempio ridurre lo stock dei prodotti ordinati, creare una riga ordine nel DB), potremmo voler eseguire tutto in un'unica transazione. Se `CalcolatoreOrdineBean` e `CarrelloBean` fossero chiamati da un metodo esterno in un EJB facciata annotato con `@Transactional` o che ha `REQUIRED`, tutte le operazioni al suo interno parteciperebbero a quella transazione. Esempio:

```
@Stateless
public class OrdineServiceBean {
    @EJB CarrelloBean carrello;
    @EJB CalcolatoreOrdineBean calc;
    @PersistenceContext EntityManager em;
    public void confermaOrdine() {
        List<String> articoli = carrello.getArticoli();
        BigDecimal totale = calc.calcolaTotale(articoli);
        // Persiste l'ordine e i dettagli nel DB
```

```
        Ordine ordine = new Ordine(...);
        em.persist(ordine);
        // svuota o rimuovi il carrello stateful per questo cliente
    }
}
```

Questo metodo, di default `REQUIRED`, si svolgerebbe in una transazione atomica che comprende: calcolo totale (solo letture), scrittura dell'ordine. Se qualcosa va male, la transazione viene rollbaccata e nulla viene confermato a metà.

Esercizio

Esercizio proposto: Realizzare una piccola logica di business usando EJB che simuli un processo reale: - Immagina di implementare un servizio di **prenotazione** (es. prenotazione di posti in un teatro). - Crea un **Stateless EJB** `GestorePrenotazioni` con metodi come `prenotaPosto(String evento, int posto)`, `annullaPrenotazione(int idPrenotazione)`, `getPostiLiberi(String evento)`. Questo bean potrebbe usare JPA per verificare e segnare i posti prenotati in un DB. Gestirà transazionalmente le operazioni: ad esempio `prenotaPosto` dovrà verificare che il posto sia libero, quindi marcarlo come occupato; se due chiamate concorrenti cercano lo stesso posto, uno dei due dovrebbe fallire per via del locking ottimistico/pessimistico su quel record (puoi simulare con JPA + `@Version` o sincronizzando a livello applicativo). - Crea poi un **Stateful EJB** `CarrelloPrenotazioni` che consente a un utente di selezionare più posti (per uno o più eventi) temporaneamente. Mantiene una lista di richieste di prenotazione ma non le finalizza finché l'utente non conferma. - Implementa un metodo `conferma()` in `CarrelloPrenotazioni` che interagisce con `GestorePrenotazioni` per effettuare tutte le prenotazioni accumulate. Questo metodo dovrebbe essere transazionale: o riescono tutte (posti prenotati) o, se anche una sola prenotazione fallisce (posto occupato nel frattempo), fa rollback per evitare conferme parziali. (Tipicamente se una fallisce, potrebbe lanciare un'eccezione che causa rollback e magari comunicare all'utente che un posto non era più disponibile). - Testa il tutto con un componente (può essere un semplice main in un client Java EE embeddable, o una Servlet): 1. Crea il Carrello (stateful EJB). 2. Aggiungi alcune prenotazioni desiderate tramite il carrello. 3. Conferma il carrello: internamente chiamerà il GestorePrenotazioni stateless per ognuna. 4. Verifica che i posti risultino occupati alla fine. Prova anche il caso in cui due carrelli concorrenti tentano lo stesso posto: uno dei due dovrebbe fallire e fare rollback.

- (Facoltativo) Aggiungi un **Timer EJB** nel GestorePrenotazioni che ogni notte reinizializzi le prenotazioni scadute (es. se c'è un concetto di prenotazione provvisoria scaduta).

Questo esercizio ti farà usare: - Stateful EJB per mantenere uno stato conversazionale (il carrello di prenotazioni). - Stateless EJB per operazioni atomiche riutilizzabili (condivise tra più client). - Transazioni container: assicurati di configurare `@TransactionAttribute` appropriati (ad esempio `conferma()` con REQUIRED e i metodi di GestorePrenotazioni con REQUIRED di default). - Gestione errori: se `prenotaPosto` lancia eccezione (es. PostoOccupatoException personalizzata *checked*), come viene gestita dalla transazione? (Spoiler: di default le eccezioni checked **non** causano rollback automatico, occorre annotare il metodo con `@ApplicationException(rollback=true)` per quelle custom, oppure lanciare un RuntimeException). - Interceptor se vuoi aggiungere logging o sicurezza (ad esempio potresti aggiungere un interceptor che logga ogni prenotazione effettuata, simile all'esempio LoggingInterceptor sopra).

Domande a scelta multipla

1. Quale tipo di EJB è più appropriato per mantenere lo stato di una conversazione con un singolo client (ad esempio un carrello acquisti utente)?
2. A. **Stateful** Session Bean, perché conserva dati tra invocazioni per quello specifico client.
3. B. Stateless Session Bean, usando variabili static per tenere traccia dello stato.
4. C. Message-Driven Bean, perché riceve messaggi e può mantenere uno stato per messaggio.
5. D. Singleton Session Bean, uno per tutti gli utenti, così lo stato è condiviso globalmente.
6. Un metodo di un EJB Stateless è annotato con `@TransactionAttribute(REQUIRES_NEW)`. Cosa accade quando viene invocato durante una transazione attiva del chiamante?
7. A. La transazione chiamante viene sospesa, il metodo EJB viene eseguito in una nuova transazione isolata, che viene committata (o rollbaccata) al termine del metodo. Al ritorno, la transazione chiamante riprende.
8. B. Partecipa comunque alla transazione chiamante, perché REQUIRES_NEW non può sovrascrivere una transazione già iniziata.
9. C. L'invocazione fallisce lanciando un'eccezione, poiché non è permesso avviare una nuova transazione dentro un'altra.
10. D. Il container ignora l'annotazione e continua con la transazione esistente (comportamento identico a REQUIRED).
11. Quale delle seguenti è **una caratteristica** dei Session Bean Stateless?
12. A. Mantengono automaticamente i dati dei client tra una chiamata e l'altra.
13. B. Possono eseguire metodi in modo asincrono usando `@Asynchronous`.
14. C. Ogni cliente ne riceve un'istanza dedicata che non viene condivisa.
15. D. Non possono essere iniettati in altri componenti, devono essere cercati solo via JNDI.
16. Un EJB ha un metodo annotato con `@Schedule(dayOfWeek="Mon", hour="1")`. Cosa implica?
17. A. Che il metodo verrà invocato automaticamente dal container ogni lunedì all'1:00 AM.
18. B. Che il metodo può essere chiamato solo il lunedì tra l'1:00 e le 2:00.
19. C. Che il metodo è in grado di calcolare date (funzione di calendario).
20. D. Niente, `@Schedule` è un'annotazione di CDI per pianificare eventi.
21. Quale affermazione sui Message-Driven Bean (MDB) è corretta?
22. A. Sono EJB che ascoltano messaggi JMS in modo asincrono, non hanno interfacce remote/local e vengono attivati dal container alla ricezione di messaggi.
23. B. Sono una variante di Stateful bean con funzionalità di messaggistica.
24. C. Richiedono che il client li invochi via JNDI come gli altri EJB.

25. D. Possono mantenere uno stato conversazionale come i Stateful (ad esempio accumulare messaggi tra invii).

26. Un EJB Stateless lancia una RuntimeException non gestita durante l'esecuzione di un metodo. Cosa succede per default?

27. A. La transazione corrente viene marcata per rollback (se esistente) e l'eccezione viene propagata (incapsulata in EJBException se passa oltre il container).
28. B. Il container ignora l'eccezione e tenta di continuare l'elaborazione.
29. C. La RuntimeException viene trasformata in una checked exception e ritentata l'operazione.

30. D. Il container riavvia l'EJB e riprova automaticamente il metodo una volta.

31. Come si può esporre un Session Bean perché sia invocabile da un'applicazione client esterna (Java SE) in un'altra JVM?

32. A. Annotando l'EJB con `@Remote` e definendo un'interfaccia remota che il bean implementa; il client otterrà il proxy via JNDI RMI.
33. B. Non è possibile: gli EJB funzionano solo all'interno del server applicativo.
34. C. Usando @WebService su un EJB; non serve interfaccia remota in questo caso.

35. D. Convertendo l'EJB in un file .exe eseguibile e facendolo girare lato client.

36. Perché un EJB Stateful potrebbe essere passivato dal container?

37. A. Per liberare risorse di memoria quando è inattivo, serializzandolo su disco (ad esempio se molti stateful bean sono in uso, quelli inutilizzati da tempo vengono passivati).
38. B. Per effettuare il versioning del bean in produzione.
39. C. Per aumentare la sicurezza, nascondendo temporaneamente i dati sensibili.

40. D. I stateful non vengono mai passivati automaticamente dal container.

41. Qual è lo scopo di un Interceptor `@AroundInvoke` in un EJB?

42. A. Permette di eseguire logica prima e/o dopo l'invocazione dei metodi business del bean (ad esempio logging, sicurezza, profiling) senza modificarne il codice.
43. B. Fornisce una alternativa per implementare i metodi business usando AOP invece di Java.
44. C. Serve per intercettare chiamate JNDI verso l'EJB.

45. D. Viene utilizzato per rilevare deadlock tra thread all'interno del container.

46. Hai un EJB stateless con `@RolesAllowed("ADMIN")` su un metodo. Cosa avviene se un utente autenticato con ruolo "USER" invoca quel metodo?

    - A. Il container nega l'accesso e lancia una `EJBAccessException` prima di eseguire il metodo.
    - B. Il metodo viene eseguito ma poi il risultato è scartato se il ruolo non combacia.
    - C. Il controllo dei ruoli non funziona sui metodi EJB, bisogna farlo manualmente.

- D. Il container reindirizza l'utente a una pagina di login ADMIN automaticamente.

API Java Message Service (JMS) e Messaggistica (Use Java Message Service API)

Teoria

Il **Java Message Service (JMS)** è l'API standard Java EE per la **messaggistica asincrona** tra componenti distribuiti. Invece di invocare metodi in tempo reale, un componente può inviare un *messaggio* a un destinatario, e il destinatario lo processerà indipendentemente, permettendo decoupling e tolleranza ai picchi di carico. JMS 2.0 è la versione inclusa in Java EE 7, che semplifica molto l'API rispetto alla 1.1.

Modelli di messaggistica JMS: JMS supporta due modelli principali: - **Point-to-Point (P2P)**: Basato sulle **Queue** (code). Un messaggio inviato a una queue viene ricevuto da **un solo** consumer. È un modello **uno-a-uno**: tipico per task queue, elaborazione concorrente, comunicazioni dirette. Il componente che invia è un *producer*, quello che riceve è un *consumer*; se più consumer sono in ascolto sulla stessa queue, ciascun messaggio della queue va a uno solo (load balancing). - **Publish/Subscribe (Pub/Sub)**: Basato sui **Topic**. Un *producer* pubblica un messaggio su un topic, e **tutti** i consumer sottoscritti a quel topic ne ricevono una copia (modello **uno-a-molti**). È utile per eventi broadcast, notifiche, aggiornamenti in tempo reale a più sistemi. I consumer su topic possono essere: - *Durable* (duraturi): ricevono i messaggi anche se erano offline al momento dell'invio, perché la sottoscrizione persiste (il broker JMS accumula i messaggi finché il consumer non si riconnette). Occorre un **Client ID** univoco e un nome di sottoscrizione per identificare la sottoscrizione durabile. - *Non-Durable*: ricevono solo quando sono attivi; se offline perdono i messaggi.

JMS 2.0 introduce inoltre le **Shared Subscriptions** (sottoscrizioni condivise) ai topic, che permettono a più consumer di condividere una singola sottoscrizione durabile, suddividendosi i messaggi (un mix tra pub/sub e load balancing, per scalare consumatori su un topic).

Componenti JMS in Java EE: - **Connection Factory**: oggetto amministrato (configurato sul server) usato per creare connessioni verso il provider JMS (il broker di messaggi). In Java EE viene tipicamente iniettato con `@Resource(lookup="...")`. - **Destinazione (Destination)**: la Queue o il Topic, anch'essa risorsa amministrata configurata. Iniettabile con `@Resource(lookup="jms/nomeQueue")` o usando l'annotation `@JMSDestinationDefinition` nel codice per dichiararla. - **JMSContext (JMS 2.0)**: nuova interfaccia che combina Connection e Session in un unico oggetto semplificato (auto-closable). Si può ottenere iniettandolo direttamente con `@Inject JMSContext context` (il container Java EE fornisce un JMSContext già connesso, se configurata default connection factory) oppure creando dal ConnectionFactory: `connectionFactory.createContext()`. - **Producer e Consumer API**: JMSContext fornisce metodi per ottenere un *producer* (`context.createProducer()`) e inviare messaggi con vari overload di `send()`, e per creare *consumer* (`context.createConsumer(queue)`), o meglio per ricevere messaggi in modo sincrono (pull) o impostare MessageListener per ricezione asincrona (push).

Produzione di messaggi (sender): Esempio, inviare un semplice messaggio di testo a una coda:

```
@Inject
private JMSContext jmsContext;
```

```
@Resource(lookup = "jms/ConnectionFactory")
private ConnectionFactory connFactory;
@Resource(lookup = "jms/OrdiniQueue")
private Queue ordiniQueue;

public void inviaOrdine(String ordineJson) {
    // Opzione 1: usare JMSContext iniettato direttamente
    jmsContext.createProducer().send(ordiniQueue, ordineJson);
    // Opzione 2: creare un context dal factory (non necessario in Java EE se
inietti)
    try (JMSContext ctx = connFactory.createContext()) {
        ctx.createProducer().setPriority(5).send(ordiniQueue, ordineJson);
    }
}
```

In JMS 2.0, come si vede, è molto semplice: `createProducer().send(destinazione, messaggio)`. Il messaggio può essere direttamente una `String` (sarà inviato come `TextMessage`) oppure un oggetto serializzabile, un flusso di byte, etc., e JMS fa il wrap in un `Message` appropriato.

Si possono impostare proprietà del messaggio con `.setProperty("chiave", valore)`, priorità, tempi di scadenza, persistenza, ecc., prima di `send()`.

Consumo di messaggi (receiver): Ci sono due modalità: - **Sincrona (pull):** Un consumer esplicito chiama `JMSConsumer.receive()` per attendere un messaggio. Adatto per semplici casi ma raramente usato in container (bloccare un thread non è efficiente). - **Asincrona (push):** Usare **Message Listener**. In Java SE si farebbe:

```
JMSConsumer consumer = jmsContext.createConsumer(ordiniQueue);
consumer.setMessageListener(new MioListener());
```

Che registra un callback da eseguire su arrivo di messaggi. In Java EE, però, la modalità tipica è delegare a un **Message-Driven Bean (MDB)**.

Message-Driven Bean (MDB): Un MDB è un EJB speciale che non espone interfacce per chiamate sincrone, ma è registrato per ricevere messaggi da una destinazione JMS in automatico dal container. Si configura tramite annotazioni:

```
@MessageDriven(
    activationConfig = {
      @ActivationConfigProperty(propertyName="destinationLookup",
propertyValue="jms/OrdiniQueue"),
      @ActivationConfigProperty(propertyName="destinationType",
propertyValue="javax.jms.Queue")
    }
)
```

```
public class ElaboraOrdineBean implements MessageListener {
    @Inject private OrderService orderService;
    public void onMessage(Message msg) {
        try {
            String ordineJson = msg.getBody(String.class);
            orderService.processaOrdine(ordineJson);
        } catch (JMSException e) {
            e.printStackTrace();
        }
    }
}
```

Questo MDB ascolterà la coda `OrdiniQueue`. Il container JMS crea pool di sessioni e consegna ogni messaggio a un'istanza (gli MDB sono multiistanza concorrenti come stateless EJB). L'MDB implementa `MessageListener.onMessage`, dove effettua la logica (qui estrae il corpo del messaggio come stringa e chiama un servizio di business). Gli MDB godono di: - Transazionalità supportata: di default l'arrivo di un messaggio avvia una transazione container; se l'MDB lancia un runtime exception, la transazione fa rollback e il messaggio può essere riposizionato in coda (redelivery). - Configurazione concorrenza: in alcuni server si può configurare il pool di thread/istanze per MDB per gestire più messaggi parallelamente. - **Durable subscription per topic:** se l'MDB ascolta un topic, può essere configurato come durable (propertyName="subscriptionDurability", propertyValue="Durable" e un subscriptionName). - Non c'è contesto di sicurezza utente propagato (i messaggi JMS non portano un identity utente a meno di meccanismi custom), ma c'è integrità transazionale e di ordine.

JMS e transazioni: JMS supporta transazioni sia a livello JMS (Session transacted) sia a livello JTA: - Nel contesto Java EE, se si invia un messaggio dentro una transazione JTA e quella transazione fa commit, il messaggio effettivamente viene inviato (commit sul provider JMS). Se la transazione JTA fa rollback, il send non ha effetto (messaggio scartato). Questo avviene automaticamente se si usa un JMSContext iniettato in un EJB transazionale. - Per i consumer/MDB, se la transazione JTA fa rollback (es. eccezione non gestita), il container JMS di solito riproverà a rideliverare il messaggio (secondo politiche di redelivery). - JMS 2 semplifica: con JMSContext in container-managed, `AUTO_ACKNOWLEDGE` mode, se dentro transazione JTA allora ack è differito a commit.

JMS avanzato: JMS 2 aggiunge varie semplificazioni: - Produttori fluenti: `createProducer().setProperty(x,y).setTimeToLive(1000).send(...)`. - Possibilità di inviare oggetti serializzabili direttamente (fa l'ObjectMessage dietro le quinte). - Shared consumer: `context.createSharedConsumer(topic, "nome")` per condividere una sottoscrizione non durabile tra più consumer in esecuzione (split dei messaggi). - JMS competes con altri sistemi di messaging (Kafka, AMQP), ma rimane standard in ambito enterprise con app server integrati (ad esempio ActiveMQ, HornetQ/Artemis su WildFly, WebLogic JMS, etc.).

Utilizzo tipico in Java EE: - Inviare messaggi JMS per *disaccoppiare* componenti: es. un modulo di ordine invia un messaggio "Ordine creato" su un topic; diversi MDB ascoltano quel topic per fare azioni parallele (aggiornare magazzino, inviare email di conferma, registrare statistiche). - Usare code come *work queue*: es. un sistema riceve tasks via web, li mette su una queue "tasks", e li processa con uno o più MDB che consumano dalla coda (pattern producer/consumer). Ciò evita di bloccare richieste utente per lavori lunghi e consente scalabilità (aggiungendo consumer). - Integrare sistemi eterogenei: JMS è spesso usato come

"colla" per integrare sistemi legacy, microservizi, mainframe, etc., dove i messaggi fungono da scambio asincrono affidabile.

Esempio pratico di codice

Scenario d'esempio: un'applicazione di elaborazione ordini online. Quando un cliente finalizza un ordine, invece di processarlo interamente all'interno della richiesta web (magari operazioni lunghe come aggiornare scorte, notificare spedizioniere, ecc.), l'app invia un messaggio JMS con i dettagli dell'ordine a una coda "Ordini". Un componente backend (MDB) elaborerà l'ordine senza bloccare l'utente.

- **Producer (in un EJB o Servlet):** invia il messaggio con i dettagli ordine.
- **Consumer (MDB):** riceve e processa.

Producer code snippet (ad es. in un Session Bean EJB metodo placeOrder):

```
@Resource(lookup = "jms/ConnectionFactory")
private ConnectionFactory jmsFactory;
@Resource(lookup = "jms/OrderQueue")
private Queue orderQueue;

public void inviaOrdine(Order order) {
    try (JMSContext context =
jmsFactory.createContext(JMSContext.AUTO_ACKNOWLEDGE)) {
        // Convertiamo l'ordine in formato testo JSON
        String json = convertOrderToJson(order);
        context.createProducer()
               .setProperty("customerId", order.getCustomerId())
               .setPriority(7)  // priorità alta
               .send(orderQueue, json);
        System.out.println("Ordine inviato nella coda: " + order.getId());
    }
}
```

Note: - Si ottiene un `JMSContext` dal factory. Nella maggior parte dei server Java EE potremmo fare `@Inject JMSContext`, ma qui mostriamo l'uso "manuale" con try-with-resources. - Si crea un Producer, si setta una property custom "customerId" e priorità (0-9, default 4). - `send` della stringa JSON. Il JMS provider incapsula la stringa in un `TextMessage`. - Dopo il try-with-resources, il JMSContext si chiude (che chiude sessione e connessione). - Questo codice sarebbe tipicamente in una transazione. Se fosse dentro un EJB metodo transazionale, e quell'EJB lancia eccezione, il messaggio non verrebbe inviato perché la transazione JTA farebbe rollback.

Consumer code snippet (MDB):

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName="destinationLookup",
```

```
propertyValue="jms/OrderQueue"),
    @ActivationConfigProperty(propertyName="destinationType",
propertyValue="javax.jms.Queue")
})
public class OrderProcessorBean implements MessageListener {

    @EJB
    private MagazzinoBean magazzino; // EJB locale per aggiornare giacenze

    @Override
    public void onMessage(Message message) {
        try {
            String json = message.getBody(String.class);
            Order order = parseJsonToOrder(json);
            // Esempio: chiama un servizio per aggiornare stock
            magazzino.scalaGiacenza(order.getItems());
            System.out.println("Elaborato ordine id=" + order.getId() +
                               " per cliente=" +
message.getStringProperty("customerId"));
        } catch (JMSException e) {
            e.printStackTrace();
            // Se eccezione non gestita, il container farà retry secondo la
policy JMS
        }
    }
}
```

Note: - L'MDB è configurato per ascoltare `jms/OrderQueue`. Il container gli fornirà i messaggi via `onMessage`. - Estrae il body come `String` (per JMS 2, se fosse JMS 1.1 bisognava castare a TextMessage e chiamare getText()). - Converte il JSON in oggetto Order (supponiamo esista parseJsonToOrder). - Chiama un EJB `MagazzinoBean` per scalare le giacenze degli articoli ordinati (questo EJB potrebbe a sua volta usare JPA per aggiornare il database). - Stampa conferma con l'informazione aggiuntiva "customerId" che il producer aveva messo come property. - Se l'elaborazione produce un errore runtime non catturato, quell'ordine potrebbe essere rimesso in coda. JMS provider tipicamente ritenta più volte fino a un max redelivery, poi può inviarlo a una *dead letter queue*.

Transazionalità MDB: Per default, l'MDB in Java EE è transazionato: l'`onMessage` avviene dentro una transazione JTA. Se `MagazzinoBean` (che è EJB) fa operazioni su DB e questo onMessage lancia un runtime exception, sia l'operazione su DB che la ricezione JMS saranno rollbaccate (il DB rollback e il messaggio torna in coda). Se invece tutto va bene, commit DB e ack JMS avvengono insieme al commit JTA finale.

Durable subscriber esempio: se invece di una Queue avessimo un Topic (es. `jms/OrderTopic`) con più MDB che ascoltano, per garantire che ricevano i messaggi anche in caso di downtime:

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(propertyName="destinationLookup",
propertyValue="jms/OrderTopic"),
  @ActivationConfigProperty(propertyName="destinationType",
propertyValue="javax.jms.Topic"),
  @ActivationConfigProperty(propertyName="subscriptionDurability",
propertyValue="Durable"),
  @ActivationConfigProperty(propertyName="clientId",
propertyValue="OrderProc_A"),
  @ActivationConfigProperty(propertyName="subscriptionName",
propertyValue="OrderProcSub")
})
public class OrderProcessorBeanA implements MessageListener { ... }
```

Così `OrderProcessorBeanA` è un subscriber duraturo, con clientId `OrderProc_A` e subscriptionName `OrderProcSub`. Se il server va giù e tornasu, i messaggi inviati al topic nel frattempo saranno consegnati.

Esercizio

Esercizio proposto: Per familiarizzare con JMS, prova a implementare un sistema di **notifiche asincrone**: - Scegli un dominio semplice, ad esempio un'app di blog. Quando viene pubblicato un nuovo post, vuoi notificare via vari canali (email, log, ecc.) gli iscritti. - Definisci un **Topic JMS "nuoviPost"**. Ogni volta che un post è pubblicato (puoi simulare chiamando un metodo EJB `pubblicaPost(titolo, contenuto)`), invia un messaggio sul topic con i dettagli (magari un JSON con titolo e id post). - Crea più **MDB subscriber** su questo topic con compiti diversi, ad esempio: 1. `NotificaEmailBean` – simula l'invio email: all'arrivo di messaggio stampa "Email inviata per nuovo post {titolo}". 2. `IndicizzazioneBean` – simula aggiornamento di un indice di ricerca: all'arrivo stampa "Indicizzato post {id}". 3. `StatisticaBean` – conta il numero di post pubblicati e salva il conteggio (magari in memoria o DB). - Configurali come sottoscrizioni durabili se ritieni (così se spegni e riaccendi l'app non perdi notifiche). - Poi, in un test (ad esempio un main con un InitialContext, o una Servlet di test), richiama il metodo di pubblicazione di post più volte e osserva l'output: dovresti vedere che ogni MDB reagisce al messaggio indipendentemente, in parallelo. - Introduci magari un piccolo ritardo o elaborazione differenziata per vedere concurrency (es. `IndicizzazioneBean` dorme 2 secondi simulando elaborazione pesante, mentre `NotificaEmailBean` è veloce). - Osserva che il client che pubblica il post non attende la fine di queste elaborazioni, procede immediatamente (questo è il vantaggio dell'asincronia). - (Facoltativo) Simula un errore in uno dei MDB (es. lancia un runtime exception in StatisticaBean per un certo post). Configura la *redelivery*: ad esempio aggiungi `@ActivationConfigProperty(propertyName="maxRetryAttempts", value="5")` se supportato, e vedi che il messaggio verrà rideliverato più volte. Oppure senza config default: magari va in dead letter dopo tot tentativi. Assicurati di gestire l'errore (magari segnalandolo).

Questo esercizio aiuta a capire: - L'uso di Topic vs Queue. - Molteplici consumer riceventi lo stesso messaggio (pub/sub). - Configurazione di sottoscrizioni durevoli. - Comportamento in caso di eccezioni (transazionalità JMS). - Come JMS consente un'architettura reattiva e scalabile.

Domande a scelta multipla

1. Nel modello JMS **Point-to-Point**, quale affermazione è corretta?
2. A. Ogni messaggio su una Queue viene consumato da **uno e un solo** consumer.
3. B. Ogni consumer riceve una copia di ogni messaggio pubblicato sulla Queue.
4. C. Non esiste un acknowledgment dei messaggi nel modello P2P.
5. D. È necessario che il producer e il consumer siano attivi contemporaneamente per scambiarsi messaggi.
6. Qual è uno dei vantaggi principali di utilizzare JMS per la comunicazione tra componenti?
7. A. Permette comunicazione **asincrona** e disaccoppiata: il sender non attende il receiver e i componenti possono essere temporaneamente offline.
8. B. Migliora la velocità di esecuzione di una singola transazione rispetto alle chiamate dirette.
9. C. Garantisce che non ci saranno errori nell'elaborazione dei messaggi.
10. D. Sostituisce completamente la necessità di un database condiviso.
11. In JMS 2.0, qual è il modo più semplice per inviare una stringa come messaggio a una destinazione?
12. A. Ottenere un `JMSContext` ed usare `jmsContext.createProducer().send(destinazione, "messaggio")`.
13. B. Creare manualmente un `TextMessage`, aprire una `Connection` e `Session` e usare un `MessageProducer` come in JMS 1.1.
14. C. Inserire il testo in un oggetto `ObjectMessage`.
15. D. Utilizzare `sendTextMessage(destinazione, testo)` sulla classe ConnectionFactory (che però non esiste).
16. Che cos'è un **Message-Driven Bean (MDB)**?
17. A. Un EJB ascoltatore di messaggi JMS, gestito dal container, che implementa `MessageListener` e processa messaggi da una coda o topic.
18. B. Un bean CDI in grado di ricevere eventi JMS.
19. C. Una variante di Stateful EJB con comunicazione asincrona verso il client.
20. D. Un EJB che viene richiamato via HTTP.
21. Come si configura tipicamente un MDB perché ascolti una specifica destinazione JMS?
22. A. Tramite annotazione `@MessageDriven` con `activationConfig` che specifica `destinationLookup` e `destinationType` (Queue o Topic).
23. B. Colocando un file XML nel classpath con la configurazione di ascolto.
24. C. Non c'è bisogno di configurazione; il container lo rileva automaticamente dal nome della classe.
25. D. Annotando la classe con `@JMSConsumer("nomeDestinazione")`.

26. In un contesto Java EE, cosa succede se un MDB lancia una RuntimeException mentre processa un messaggio?

27. A. La transazione JMS viene rollbaccata, quindi il messaggio **non viene riconosciuto come elaborato** e verrà rideliverato (secondo le politiche di redelivery).
28. B. Il messaggio viene considerato elaborato comunque e scartato.
29. C. L'eccezione viene ignorata dal container JMS.

30. D. L'MDB viene disattivato immediatamente.

31. Cosa distingue una **sottoscrizione duratura (durable subscription)** in JMS?

32. A. È relativa ai Topic: un consumer duraturo riceve i messaggi pubblicati anche quando è offline, una volta che si riconnette.
33. B. Vale sia per Queue che Topic e garantisce che i messaggi siano persistiti su disco sempre.
34. C. Significa che il messaggio ha priorità alta.

35. D. È un termine deprecato in JMS 2, non esiste più il concetto di durable.

36. Quale codice è corretto per ricevere asincronamente messaggi JMS in un'applicazione Java SE (non Java EE)?

37. A.

```
JMSContext ctx = connFactory.createContext();
JMSConsumer consumer = ctx.createConsumer(queue);
consumer.setMessageListener(new MioListener());
```

38. B.

```
ctx.subscribe(queue, MioListener.class);
```

39. C.

```
Message msg = consumer.receive();
```

40. D.

```
Message msg = ctx.getMessage();
```

41. Il metodo `JMSProducer.setProperty("key","value")` a cosa serve?

42. A. Aggiunge una proprietà custom nel messaggio inviato, che il consumer potrà leggere (ad esempio con `message.getStringProperty("key")`).
43. B. Imposta un attributo di configurazione sul JMS provider (es: QoS).
44. C. Definisce un filtro: solo i consumer che specificano `key=value` riceveranno il messaggio.
45. D. Imposta l'header JMSCorrelationID.
46. In un'applicazione Java EE, qual è il modo più comune di ottenere una ConnectionFactory o Destination JMS?
    - A. Configurarle come risorse nel server (ad es. tramite file di configurazione o console) e poi usare `@Resource(lookup="jndi/name")` per iniettarle nel codice.
    - B. Instanziarle con `new ActiveMQConnectionFactory()` direttamente nel codice.
    - C. Farle restituire da un metodo di un EJB.
    - D. Non c'è supporto per injection, bisogna fare JNDI lookup manuale ogni volta.

Servizi SOAP con JAX-WS e JAXB (Implement SOAP Services by Using JAX-WS and JAXB)

Teoria

SOAP (Simple Object Access Protocol) è un protocollo standard per web service basato su XML. In Java EE 7, la specifica **JAX-WS (Java API for XML-Web Services)** consente di creare servizi SOAP e client in Java con relativa semplicità, astraendo i dettagli di basso livello (creazione di XML, gestione di messaggi SOAP, ecc.). **JAXB (Java Architecture for XML Binding)** è la tecnologia complementare che effettua il *mapping* tra classi Java e rappresentazione XML, usata da JAX-WS per serializzare/deserializzare oggetti nei messaggi SOAP.

Un servizio SOAP è tipicamente definito da un **WSDL (Web Services Description Language)**, un documento XML che descrive le operazioni offerte, i parametri, i tipi di dati (tramite XSD) e come accedervi (endpoint URL, binding). Con JAX-WS, possiamo **codificare il web service in Java** e lasciare che il runtime generi automaticamente il WSDL (approccio *code-first*), oppure a partire da un WSDL generare scheletro di codice (*wsimport*, approccio *contract-first*).

Creare un Web Service JAX-WS (server side): In Java EE 7, un servizio SOAP può essere implementato come: - Un **POJO annotato con** `@WebService`, eventualmente anche un EJB Stateless per integrare transazioni e pool (spesso è consigliato usare `@Stateless` + `@WebService` insieme). - Ogni metodo pubblico della classe diventa un'operazione SOAP (a meno che annotato con `@WebMethod(exclude=true)` per escluderlo). - Gli eventuali parametri e valori di ritorno vengono automaticamente convertiti in XML usando JAXB (tipi semplici come String, int hanno mapping immediato, oggetti complessi devono rispettare regole JAXB: avere default constructor, campi/proprietà accessibili, ecc.).

Esempio minimale:

```
import javax.jws.WebService;
import javax.jws.WebMethod;
```

```java
import javax.ejb.Stateless;

@Stateless
@WebService(serviceName="CiaoService")
public class CiaoService {
    @WebMethod
    public String saluta(String nome) {
        return "Ciao, " + nome;
    }
}
```

Qui: - `@WebService` indica che la classe espone un servizio SOAP (opzionalmente si può specificare name, targetNamespace, etc., altrimenti vengono derivati dai nomi di classe/pacchetto). - `@Stateless` rende la classe un EJB stateless (facoltativo, ma se volessimo injection di altre risorse e transazionalità automatica, è utile). - Il metodo `saluta` sarà esposto come operazione. `@WebMethod` qui non sarebbe strettamente necessario se vogliamo esporlo (di default i public sono esposti), ma si può usare per controllare il nome dell'operazione o per documentazione.

Quando deployiamo questa classe su un container Java EE (ad esempio GlassFish o WildFly), il container JAX-WS: - Genererà un WSDL per `CiaoService` (di solito accessibile a un URL come `http://<host>:<port>/<app>/CiaoService?wsdl`). - Esporrà un endpoint SOAP ascoltando su un URL (es: `http://<host>:<port>/<app>/CiaoService`). - Gestirà automaticamente le richieste SOAP: quando arriva un SOAP `<saluta>` request, invoca il metodo `saluta` con i parametri estratti, e incapsula la stringa di ritorno in una SOAP response.

Java to WSDL mapping (JAX-WS/JAXB): - I parametri e return vengono convertiti in XML. Per tipi semplici (string, int, boolean, etc.), c'è un mapping definito (xsd:string, xsd:int, etc.). Per classi complesse, JAX-WS utilizza JAXB: ad esempio, se avessimo un tipo `Utente` come parametro, dobbiamo annotare la classe `Utente` con JAXB annotations (`@XmlRootElement`, `@XmlType`, `@XmlElement` su campi se necessario) in modo che possa essere convertita. Se non lo facciamo esplicitamente, JAX-WS tenterà di usare le convenzioni di default di JAXB comunque (JAXB fa binding default anche senza annotazioni per molti casi). - Le *collezioni* generalmente vengono tradotte in elementi ripetuti. Gli *array* vanno. - Possiamo personalizzare il mapping con annotazioni JAXB (es: @XmlElement(name="...") per cambiare nome di tag, @XmlTransient per escludere un campo, etc.). - Se il metodo lancia eccezioni, esse possono essere comunicate come *fault* SOAP. Le eccezioni che estendono `Exception` (non Runtime) e sono dichiarate nel throws, verranno trasformate in fault con un mapping (per default, generano un `faultcode` e un dettaglio con il nome dell'eccezione e messaggio, ma si può controllare con `@WebFault`).

Client JAX-WS (chiamare un servizio SOAP): Ci sono due strade: - **Generare stub client dal WSDL (statico):** usando lo strumento `wsimport` (fornito nel JDK), che crea le classi Java (service endpoint interface, model JAXB) a partire dal WSDL. Poi nel codice Java crei un *Service* e ottieni il port (che implementa l'interfaccia). Esempio: supponi WSDL definisca un servizio `CiaoService` con port `CiaoServicePort` e operazione `saluta`. `wsimport` genera: - `CiaoService` (classe *Service* che estende javax.xml.ws.Service) - `CiaoServicePortType` (interfaccia con il metodo saluta) - eventuali classi per parametri complessi. Allora un client farebbe:

```
URL wsdlURL = new URL("http://host:port/app/CiaoService?wsdl");
QName serviceName = new QName("http://package.ws/", "CiaoService");
CiaoService service = new CiaoService(wsdlURL, serviceName);
CiaoServicePortType port = service.getCiaoServicePort(); // ottenere il proxy
String risposta = port.saluta("Mario");
```

Il port è un proxy che maschera le chiamate SOAP come chiamate a metodi Java normali. - **Client dinamico (Dispatch API o JAX-WS Proxy):** - È possibile evitare la generazione e usare `Service.create()` e poi `service.getPort(EndpointInterface.class, ...)` se si ha già l'interfaccia annotata a disposizione. - Oppure usare `Dispatch<SOAPMessage>` o `Dispatch<Source>` se vuoi costruire manualmente i messaggi XML (più complesso, raramente necessario se hai l'interfaccia). - In Java EE, si può anche usare *Injection* via `@WebServiceRef` per ottenere un port senza manual `wsimport`, ma dietro le quinte il container fa simile:

```
@WebServiceRef(CiaoService.class)
private CiaoService service;
...
CiaoServicePortType port = service.getCiaoServicePort();
```

(Qui si presuppone di avere l'artefatto di servizio già, quindi solitamente si genera comunque).

JAXB personalizzazioni: Con JAXB puoi: - Annotare le classi modello con `@XmlRootElement(name="Persona")`, `@XmlType(propOrder={"nome","eta"})` etc. - Usare adattatori per tipi speciali (es. Date -> String format). - Definire *Schema* XSD a mano e generare classi con `xjc` (approccio contract-first). - In JAX-WS, per mapping complessi, spesso conviene definire il WSDL/XSD e generare piuttosto che cercare di far fare tutto a JAXB, ma per l'esame è importante sapere come funzionano le annotazioni di base e i default.

SOAP vs REST: Vale la pena notare in teoria: - SOAP è protocollo basato su **XML e protocolli standard** (WSDL, SOAP envelope, XSD) e orientato a operazioni (RPC style o document style). - Spesso usato in ambito enterprise per esigenze di formalità, contratti stabili, transazioni distribuite (WS-Transaction), sicurezza avanzata (WS-Security, con firma e crittografia a livello messaggio). - Più verboso di REST/JSON, ma supporta tool di integrazione, e.g. .NET e Java scambiano via SOAP facilmente con WSDL come contratto intermedio. - JAX-WS su Java EE implementa molte specifiche correlate: ad esempio *handlers* (simili a filtri per i messaggi SOAP), *MTOM* (attacchi binari), *WS-Addressing*, *WS-Security* (con librerie aggiuntive), etc., ma l'esame probabilmente non entra in questi dettagli, solo concetti base.

In Java EE, esposizione automatica su server integrato: Ad esempio, su GlassFish, il semplice deployment del `@WebService` class su un WAR o EJB JAR è sufficiente. Su un server come Tomcat (che non è Java EE full), si userebbero librerie come Metro o Apache CXF per avere funzionalità simili.

JAX-WS e EJB integrati: L'esempio sopra mostra che un EJB può essere annotato come WebService. I requisiti (da specifica) includono che l'EJB deve essere stateless (o singleton) e deve avere `@WebService` sull'implementazione. Il container genera l'interfaccia SEI (Service Endpoint Interface) se non fornita e la WSDL automaticamente.

Marshall/Unmarshall con JAXB stand-alone: In contesto SOAP, raramente si usa direttamente, ma sapere che `JAXBContext.newInstance(Classe.class)` e poi `marshal(object, OutputStream)` produce XML e viceversa `unmarshal(xml)` produce oggetti. JAX-WS fa questo dietro le quinte per parametri.

Attachment (MTOM): Se chiedono, SOAP può trasportare allegati binari (file) in modo efficiente con MTOM (Message Transmission Optimization Mechanism). In JAX-WS si abilita con `@MTOM` su servizio e usando `DataHandler` o `byte[]` come param, ma è dettaglio forse avanzato.

Esempio pratico di codice

Illustriamo la creazione di un servizio SOAP e del suo client. Scenario: **Calcolatrice**. Esporremo un servizio che fornisce operazioni aritmetiche base (somma, sottrai, moltiplica, dividi). Dimostreremo: - Il server: un `CalculatorService` JAX-WS con due metodi. - Il client: come chiamare questi metodi con JAX-WS, ipotizzando di aver generato il client con wsimport (per brevità, qui scriveremo come se l'interfaccia fosse nota).

Server-side (implementazione del servizio):

```
import javax.jws.WebService;
import javax.jws.WebMethod;
import javax.ejb.Stateless;

@Stateless
@WebService(serviceName="CalculatorService")
public class CalculatorService {
    @WebMethod
    public int somma(int a, int b) {
        return a + b;
    }
    @WebMethod
    public double dividi(double a, double b) throws DivisionePerZeroException {
        if(b == 0) {
            // Solleva una eccezione personalizzata dichiarata
            throw new DivisionePerZeroException("Divisione per zero non
consentita");
        }
        return a / b;
    }
}
```

Abbiamo: - `serviceName="CalculatorService"` che determinerà il nome del servizio nel WSDL. - Due metodi annotati con `@WebMethod`: - `somma` ritorna int ed è esposto normalmente. - `dividi` lancia un'eccezione checked `DivisionePerZeroException` (che abbiamo definito come estensione di Exception). Perché questo funzioni, l'eccezione dovrebbe essere annotata `@WebFault` con un nome e un fault info (JAX-WS può generare una classe *FaultBean* per trasportare dettagli). Ad esempio:

```
@WebFault(name="DivisionePerZeroFault")
public class DivisionePerZeroException extends Exception {
   public DivisionePerZeroException(String message) {
       super(message);
   }
   // potremmo avere un getter per detail, ma in questo caso semplice no
}
```

Il client riceverà un SOAP Fault con `<faultcode>` etc., e attraverso JAX-WS sarà tradotto di nuovo in `DivisionePerZeroException`.

Supponendo di aver deployato questo su un server locale, avremo un WSDL disponibile, contenente le operazioni "somma" e "dividi", con i tipi xsd appropriati e un fault "DivisionePerZeroFault" per l'operazione dividi.

Client-side (chiamata del servizio):

Metodo 1: usare dynamic proxy con l'interfaccia SEI generata. Poiché noi abbiamo la classe implementazione, possiamo generare un SEI. Comunque, spesso conviene usare wsimport: Se eseguiamo `wsimport http://localhost:8080/myapp/CalculatorService?wsdl`, otteniamo: - `CalculatorService` (classe service) - `CalculatorServicePortType` (interfaccia con metodi somma e dividi) - `DivisionePerZeroException` (riutilizzato se same package, altrimenti genera un wrapper)

Allora:

```
public class TestClient {
    public static void main(String[] args) throws Exception {
        URL wsdlURL = new URL("http://localhost:8080/myapp/CalculatorService?
wsdl");
        QName SERVICE_NAME = new QName("http://mio.package/",
"CalculatorService");
        CalculatorService service = new CalculatorService(wsdlURL,
SERVICE_NAME);
        CalculatorServicePortType port =
service.getCalculatorServicePort();  // ottiene il port (proxy)

        int risultato = port.somma(5, 7);
        System.out.println("Somma(5,7) = " + risultato);

        try {
            double risultato2 = port.dividi(10.0, 0.0);
            System.out.println("Dividi(10,0) = " + risultato2);
        } catch (DivisionePerZeroException e) {
            System.err.println("Errore dal servizio: " + e.getMessage());
        }
```

```
    }
}
```

Spiegazione: - Creiamo un `URL` del WSDL e un `QName` col namespace e nome servizio (questi dettagli li avremmo dal WSDL). - Invochiamo il costruttore di `CalculatorService` generato (questo estende `javax.xml.ws.Service`). - Otteniamo il port (che implementa `CalculatorServicePortType`). - Chiamiamo `somma` e stampiamo il risultato. - Chiamiamo `dividi` con 0, catturiamo l'eccezione `DivisionePerZeroException` lanciata dal servizio (il proxy JAX-WS l'ha convertita dal SOAP Fault in un Exception Java). - Stampiamo il messaggio di errore.

In un ambiente Java EE, potremmo evitare un po' di boilerplate usando injection: Se questo client fosse in un Servlet/EJB del server stesso:

```
@WebServiceRef(wsdlLocation="http://localhost:8080/myapp/CalculatorService?
wsdl")
private CalculatorService service;
...
CalculatorServicePortType port = service.getCalculatorServicePort();
int res = port.somma(2,3);
```

Il container risolverebbe `@WebServiceRef` e gestirebbe creazione del service e injection.

JAXB Example: Anche se JAX-WS maschera JAXB, per chiarezza presentiamo come JAXB mappa un oggetto. Se avessimo un tipo complesso:

```
@XmlRootElement
public class Prodotto {
    @XmlElement
    public String nome;
    @XmlElement
    public double prezzo;
    // + costruttore, getters...
}
```

e un metodo web service `public Prodotto getProdotto(String nome)`. JAX-WS genererebbe nel WSDL un complesso type "Prodotto" con elementi nome (string) e prezzo (double). Il SOAP <getProdottoResponse> includerà un <return> con struttura <nome>..</nome><prezzo>..</prezzo>.

Esercizio

Esercizio proposto: Mettere in pratica un web service SOAP e un client: - Crea e deploya un servizio SOAP che fornisce funzionalità di una **rubrica contatti**. Chiamiamolo `RubricaService`. Deve avere operazioni: - `aggiungiContatto(String nome, String telefono)` -> ritorna un id (int o String) del contatto inserito. - `cercaContatto(String nome)` -> ritorna il telefono del contatto dato il nome, oppure null/

segnalazione se non trovato. - `eliminaContatto(String nome)` -> boolean se ha eliminato. Puoi implementarlo in un EJB stateless con una semplice struttura dati in memoria (es. `Map<String,String>` come repository). - Verifica che il WSDL sia generato correttamente e magari osserva il file (per vedere le operazioni e i tipi). - Implementa un **client stand-alone Java SE** per questo servizio: - Puoi usare wsimport per generare le classi, oppure usare JAX-WS dynamic proxy. - Dal client, invoca `aggiungiContatto`, poi `cercaContatto` per lo stesso nome, verifica che il telefono corrisponda, poi `eliminaContatto` e infine di nuovo `cercaContatto` (dovrebbe non trovarlo). - Gestisci eventuali fault (in questo caso potresti non aver definito fault custom, ma comunque gestisci se cercaContatto restituisce null). - Amplia il servizio aggiungendo un tipo complesso: ad esempio un metodo `getTuttiContatti()` che ritorna un elenco di oggetti `Contatto` (dove Contatto ha nome e telefono). Assicurati di definire la classe `Contatto` con JAXB (`@XmlRootElement` o includila come tipo in un metodo perché JAX-WS la conosca). - Esercitati nel leggere i risultati: la lista di contatti come arriva nel client? (Probabilmente come `List<Contatto>` se usi SEI generata, altrimenti come array). - **Extra:** se possibile, esplora WS-Security a livello base: non implementazione completa, ma prova ad aggiungere nell'header SOAP del client un semplice elemento <auth> e fallo filtrare nel server con un handler. Questo è avanzato, facoltativo.

Questo esercizio toccherà: - Sviluppo e deployment di un servizio JAX-WS su Java EE. - Marshalling/unmarshalling implicito di tipi semplici e collezioni via JAXB. - Uso di wsimport o WebServiceRef per generare e utilizzare un client. - Comprensione di come i tipi Java corrispondono in WSDL/XSD (puoi usare un tool o browser per dare un occhio al WSDL generato). - (Opzionale) Gestione di fault custom se vuoi, ad esempio se `cercaContatto` deve lanciare un Fault se non trovato, definisci un `ContattoNonTrovatoException` con @WebFault e fanne il throw.

Domande a scelta multipla

1. Quale annotazione è utilizzata per contrassegnare una classe Java come servizio web SOAP (JAX-WS)?
2. A. `@WebService` sulla classe implementativa del servizio.
3. B. `@WebServlet`
4. C. `@RestService`
5. D. `@SoapEndpoint`

6. In JAX-WS, come si definisce che un metodo del servizio **NON** venga esposto come operazione nel WSDL?

7. A. Annotandolo con `@WebMethod(exclude=true)`.
8. B. Rendendo il metodo `private`.
9. C. Non è possibile escludere metodi pubblici.
10. D. Rimuovendo l'annotazione @WebService dalla classe prima della compilazione WSDL.

11. Se un metodo di un Web Service JAX-WS lancia un'eccezione controllata (checked exception) non annotata, cosa accade lato SOAP?

12. A. Verrà trasformata automaticamente in un SOAP Fault con un elemento `<faultstring>` contenente il messaggio di eccezione.
13. B. L'eccezione viene ignorata e non segnalata nel SOAP.

14. C. Il servizio si blocca, il client non riceve risposta.

15. D. JAX-WS rilancia un RuntimeException generica al client invece.

16. Quale strumento o tecnica si usa per generare automaticamente classi Java client (stub) a partire da un WSDL?

17. A. Il comando `wsimport` (in JDK) per JAX-WS.
18. B. Il comando `xjc` per i WSDL (dedicato ai WSDL, non esiste).
19. C. Non si può generare, bisogna scriverli manualmente.

20. D. JAX-RS Client API (sbagliato, quella è per REST).

21. In un file WSDL, cosa rappresenta la sezione `<wsdl:portType>`?

22. A. L'interfaccia (collezione di operazioni) offerta dal web service (equivale alle operazioni dell'endpoint Java).
23. B. L'indirizzo (URL) a cui è esposto il servizio.
24. C. Il formato delle particolarità di trasporto (HTTP, JMS).

25. D. Le informazioni di sicurezza.

26. A cosa serve JAXB nel contesto di JAX-WS?

27. A. A effettuare il binding (conversione) tra oggetti Java e XML (elementi nei messaggi SOAP).
28. B. A generare interfacce WSDL a partire dal codice Java.
29. C. A eseguire le query sul database all'interno di un servizio web.

30. D. A gestire la concorrenza tra chiamate contemporanee al servizio.

31. Qual è il modo standard di richiamare un Web Service SOAP in un client Java SE?

32. A. Ottenere un proxy via `Service` e `getPort`, poi invocare i metodi come se fossero locali.
33. B. Effettuare manualmente chiamate HTTP POST costruendo l'XML SOAP con stringhe.
34. C. Utilizzare l'API REST di JAX-RS per chiamare l'URL del WSDL.

35. D. Inviare un JMS con il contenuto del SOAP al server.

36. Se un metodo web service ha firma `public Cliente getCliente(int id)`, cosa deve avere la classe `Cliente` per essere serializzata correttamente in SOAP (usando JAXB)?

37. A. Un costruttore pubblico senza argomenti e campi/proprietà accessibili (pubblici o con getter/setter) per i dati da trasmettere.
38. B. Deve estendere javax.xml.bind.Marshaller.
39. C. Niente, qualsiasi classe funziona perché JAXB usa la reflection libera.

40. D. Deve implementare un'interfaccia speciale WebServiceDTO.

41. Volendo esporre un EJB Stateless come Web Service SOAP, è possibile farlo?

42. A. Sì, basta annotare la classe EJB con `@WebService` e tipicamente mantenere i vincoli (public methods, niente final, ecc.).
43. B. No, EJB e Web Service sono tecnologie incompatibili tra loro.
44. C. Sì, ma bisogna ereditare da una classe base di JAX-WS.

45. D. No, solo i Servlet possono essere Web Service SOAP.

46. Quale è un vantaggio tipico di SOAP rispetto a REST?

    - A. SOAP ha standard per definire formalmente il contratto (WSDL) e supporta estensioni come sicurezza e transazioni (WS-*) a livello di protocollo.
    - B. SOAP è molto più semplice da utilizzare rispetto a REST/JSON.
    - C. SOAP funziona solo con Java, REST no.
    - D. SOAP utilizza meno banda perché XML è più compatto di JSON.

Applicazioni Web con Servlet (Create Java Web Applications using Servlets)

Teoria

Le **Servlet** sono componenti Java lato server che implementano la specifica Java EE Web (Servlet API, versione 3.1 in Java EE 7). Una Servlet riceve richieste HTTP dal client (tipicamente un browser) e produce risposte (HTML, JSON, binario, ecc.). In Java EE, le Servlet costituiscono la base di *tutte* le tecnologie web: sia JSP che JSF alla fine vengono compilati in Servlet. Capire le Servlet significa comprendere come gestire le richieste HTTP in Java.

Ciclo di vita di una Servlet: gestito dal **Web Container**: - All'avvio dell'applicazione (o al primo accesso se la Servlet è lazy), il container crea un'istanza della classe Servlet (deve avere un costruttore pubblico no-arg). - Chiama `init(ServletConfig)` una volta, per inizializzazione (qui la Servlet può leggere parametri di inizializzazione, aprire risorse condivise, etc.). - Per ogni richiesta HTTP destinata alla Servlet: - Il container invoca il metodo `service(HttpServletRequest, HttpServletResponse)` su un thread dal pool. Di default, `service` smista verso `doGet`, `doPost`, `doPut`, etc. in base al metodo HTTP (se la nostra Servlet estende `HttpServlet` e non sovrascrive `service`). - Dentro `doGet`/`doPost` l'applicazione elabora la richiesta: leggere parametri, sessione, etc., e scrive la risposta usando l'oggetto HttpServletResponse (metodo `getWriter()` per testo, o `getOutputStream()` per binario). - Quando l'applicazione viene fermata o la Servlet distrutta (ad esempio per reload), il container chiama `destroy()` sulla Servlet per permetterle di rilasciare risorse (chiudere connessioni, file, thread ecc.). Dopo di che l'istanza è eleggibile a GC.

Mappatura delle Servlet a URL: Ci sono due modi: - Tramite **annotazione** `@WebServlet(urlPatterns={"/path"})` sulla classe (introdotto con Servlet 3.0). Ad esempio `@WebServlet("/saluto")` indica che la servlet risponde all'URL pattern *"/appcontext/saluto". - Tramite dichiarazione in web.xml (deployment descriptor). Lì si definisce* `<servlet>` *e* `<servlet-mapping>` *con un url-pattern (es.* `/saluto`*). I pattern possono essere: - Precisi ("/login") - Con wildcard suffisso ("/api/*"* significa

qualsiasi path che inizia con /api/) - Wildcard prefisso per estensione ("*.jsp" è classico per JSP). Il container sceglie la migliore corrispondenza per la richiesta.

Handling delle richieste HTTP: - `HttpServletRequest` fornisce metodi per ottenere: - **Parametri** (da query string o corpo form) con `getParameter("nome")` restituendo string o `getParameterValues` per array (es. checkbox multiple). - **Header HTTP** con `getHeader("User-Agent")`, `getDateHeader` etc. - **Cookie** con `getCookies()` che dà un array di Cookie presenti. - **Sessione** con `getSession()` che restituisce (e crea se non esiste) un HttpSession, meccanismo per mantenere dati tra richieste del medesimo client (tipicamente con cookie JSESSIONID). - **Stream di input** con `getInputStream()` o `getReader()` se vogliamo leggere direttamente il body (usato per upload di file o JSON in POST). - Informazioni di contesto e percorso: `getRequestURI()`, `getContextPath()`, `getServletPath()`, ecc per capire cosa è stato richiesto. - `HttpServletResponse` permette di: - Impostare **codice di stato** HTTP (200, 404, 500, ecc.) con `setStatus(int)` o metodi convenience come `sendError(404)` o `sendRedirect(url)`. - Impostare **header** con `setHeader("Nome", "Valore")` o specifici come `setContentType("mime/type")`, `setCookie`. - Ottenere **writer** o **output stream** per scrivere il corpo della risposta. Se scriviamo testo (HTML/JSON), tipicamente:

```
resp.setContentType("text/html; charset=UTF-8");
PrintWriter out = resp.getWriter();
out.println("<html><body>Hello</body></html>");
```

Se inviamo binari (es un'immagine PDF) useremo `OutputStream out = resp.getOutputStream();` e poi out.write(byte[]). - Gestire **redirect**: `sendRedirect("altraUrl")` dice al client di fare nuova richiesta (codice 302). - Gestire **forward/include** (in collaborazione con RequestDispatcher). - **Cookies**: class `javax.servlet.http.Cookie`. Per aggiungerne uno nella risposta:

```
Cookie c = new Cookie("nome", "valore");
c.setMaxAge(60*60*24); // dura un giorno
resp.addCookie(c);
```

Il client li rimanderà nelle successive richieste (fintanto che path e dominio corrispondono). - **Sessione**: `HttpSession` ottenuta con `request.getSession()`. Permette `setAttribute` e `getAttribute` per conservare oggetti tra richieste (legati al cookie JSESSIONID). Importante considerare il timeout (di default 30 min configurabile) e la serializzazione se in cluster.

Filtri (Filters): Oltre alle Servlet, la specifica definisce i **Filter** (`javax.servlet.Filter`). Un filter intercetta *prima* che la richiesta arrivi alla Servlet (e può anche post-processare la risposta). Configurati simili a servlets (annotazione `@WebFilter` o web.xml mapping su URL). - Esempi: LoggingFilter (logga ogni richiesta), AuthFilter (blocca accesso se non loggato, reindirizza al login), CompressionFilter (comprime output). - In `doFilter(request, response, chain)`, si può: - Analizzare/modificare request (ad es. aggiungere attributi, wrapping request con una custom wrapper). - Scegliere di **non continuare** la catena (es. abortire rispondendo direttamente, utile per accesso negato). - Oppure **proseguire** con `chain.doFilter(request, response)` che invoca il prossimo elemento (prossimo filtro o la servlet finale). - Dopo `chain.doFilter`, si può eseguire logica *post* (es. misurare tempo di esecuzione, alterare/analizzare la risposta - se si avvolge la response in wrapper). - Filtri funzionano su tutte le risorse matching il

loro pattern, incluse JSP, statici, etc. L'ordine dei filtri è definito da dichiarazione (o con @WebFilter uno può specificare `@Order` se container supporta).

Asynchronous support (Servlet 3.0): Non so se esame copre, ma Servlet 3 ha `request.startAsync()` che permette di gestire richieste asincrone (rilasciando il thread e completando più tardi), e utilizzare la NIO (non-blocking I/O API). Questo permette di scalare meglio per operazioni lunghe senza occupare thread. L'outline del corso menzionava "async servlet e NIO". In breve: - `@WebServlet(asyncSupported=true)` e nel doGet, ad esempio:

```
AsyncContext asyncCtx = request.startAsync();
new Thread(() -> {
     // operazione lunga
     try { Thread.sleep(5000); } catch...
     HttpServletResponse resp = (HttpServletResponse) asyncCtx.getResponse();
     resp.getWriter().write("Operazione completata");
     asyncCtx.complete();
}).start();
```

Il thread di richiesta originale torna libero immediatamente. Quando l'operazione completata, scriviamo risposta e chiamiamo complete. - L'API NIO (Non-Blocking IO) consente di leggere input (request.getInputStream().setReadListener(...)) e scrivere output in modo non bloccante con listener. Argomenti avanzati, probabilmente non in quiz base.

CDI beans in Servlets: Possiamo usare `@Inject` dentro Servlets per ottenere CDI bean o EJB, grazie all'integrazione di Java EE 7 (il web container è un contesto in cui CDI funziona). Quindi una Servlet può, ad esempio, `@Inject PersonService service;` e usarlo, invece di fare lookup JNDI manuale.

Invocare diverse risorse web: A volte la Servlet agisce come controller che chiama JSP (View) per generare HTML: lo fa tramite `RequestDispatcher forward`:

```
RequestDispatcher rd = request.getRequestDispatcher("/risultato.jsp");
rd.forward(request, response);
```

Questo passa il controllo a JSP (sul lato server) e quella JSP vedrà gli attributi request settati dalla servlet. Forward non cambia l'URL visto dal client (è server-side). `rd.include(request, response)` invece include l'output di un'altra risorsa (header e status non vengono commutati, utile per componenti come header/footer includibili).

Error handling: - Tramite web.xml si può mappare codici di errore o eccezioni a pagine di errore (elemento `<error-page>`). Es: `<error-page><error-code>404</error-code><location>/error404.jsp</location></error-page>`. - All'interno di Servlet/JSP, si può usare `sendError` per inviare un codice e lasciare che il container visualizzi la pagina d'errore designata. - Oppure gestire tramite try/catch e dispatch a pagine dedicate.

Contesto (ServletContext): Rappresenta l'applicazione web. Con `request.getServletContext()` si ottengono parametri globali, si può loggare (context.log), leggere risorse (getResourceAsStream per file in classpath/webcontent), ecc. Il ServletContext è anche disponibile in `init()` e come injection `@Resource`.

Esempio pratico di codice

Creiamo una semplice Servlet che illustra vari concetti: **LoginServlet** che gestisce una form di login. - In GET mostra un form HTML di login. - In POST riceve username e password, verifica le credenziali (diciamo in modo banale in codice o usando un EJB injectato). - Se valide, imposta l'utente in sessione e reindirizza a una pagina di benvenuto. - Se non valide, reindirizza di nuovo al form con messaggio di errore (magari usando parametri query o sessione).

E inoltre, definiremo un Filter che controlla l'accesso alle pagine protette (es. la pagina di benvenuto) assicurandosi che il login sia avvenuto.

LoginServlet.java:

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
    // Simuliamo un servizio di autenticazione, in scenario reale sarebbe EJB/
CDI
    private Map<String, String> utenti = new HashMap<>();
    @Override
    public void init() throws ServletException {
        // Inizializza qualche utente fittizio
        utenti.put("alice", "1234");
        utenti.put("bob", "abcd");
    }

    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        resp.setContentType("text/html; charset=UTF-8");
        PrintWriter out = resp.getWriter();
        // Messaggio di errore se presente come param (es. ?error=1)
        String error = req.getParameter("error");
        out.println("<!DOCTYPE html><html><body>");
        if ("1".equals(error)) {
            out.println("<p style='color:red;'>Credenziali non valide,
riprova.</p>");
        }
        out.println("<form method='POST' action='login'>");
        out.println("Utente: <input name='username'/><br/>");
        out.println("Password: <input type='password' name='password'/><br/>");
        out.println("<button type='submit'>Login</button>");
```

```
        out.println("</form></body></html>");
    }

    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String user = req.getParameter("username");
        String pass = req.getParameter("password");
        if (user != null && pass != null && pass.equals(utenti.get(user))) {
            // Credenziali valide
            req.getSession().setAttribute("user", user);
            resp.sendRedirect("benvenuto");  // pagina protetta
        } else {
            // Credenziali non valide, rimanda a login con parametro di errore
            resp.sendRedirect("login?error=1");
        }
    }
}
```

Note: - In `init`, la Servlet crea una mini mappa di utenti->password per test. In un'app reale, questo sarebbe DB o un EJB `UserService` injected. - **doGet**: produce la pagina HTML del form login. Se l'URL contiene `?error=1`, mostra un messaggio di errore. - **doPost**: legge parametri `username` e `password` dal corpo della request (essendo form POST). - Se la password corrisponde a quella attesa (autenticazione banale), allora: - Usa la HttpSession (`req.getSession()`) per salvare un attributo "user" con il nome utente, segnando che l'utente è loggato. - Fa un redirect alla pagina di benvenuto (la cui URL mapping supponiamo sia "/benvenuto"). - Se login fallisce, redirect di nuovo a "/login?error=1". - Usiamo `sendRedirect`, quindi il browser farà una nuova richiesta GET verso l'URL specificato. Notare che potevamo fare un forward lato server verso una JSP di errore, ma qui preferiamo redirect per far vedere la URL cambiata e prevenire repost del form.

BenvenutoServlet.java: (la pagina protetta)

```
@WebServlet("/benvenuto")
public class BenvenutoServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        HttpSession session = req.getSession(false);
        String user = (session != null) ? (String)
session.getAttribute("user") : null;
        resp.setContentType("text/html; charset=UTF-8");
        PrintWriter out = resp.getWriter();
        out.println("<html><body>");
        if (user != null) {
            out.println("<h1>Benvenuto, " + user + "!</h1>");
            out.println("<a href='logout'>Logout</a>");
```

```
        } else {
            out.println("<h1>Utente non loggato</h1>");
            out.println("<a href='login'>Vai al login</a>");
        }
        out.println("</body></html>");
    }
}
```

Questa Servlet semplicemente mostra un messaggio di benvenuto se l'utente è in sessione, altrimenti comunica che non è loggato. (Abbiamo aggiunto un link al logout per completezza.)

LogoutServlet.java: (per terminare la sessione)

```
@WebServlet("/logout")
public class LogoutServlet extends HttpServlet {
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        HttpSession session = req.getSession(false);
        if (session != null) {
            session.invalidate(); // invalida la sessione esistente
        }
        resp.sendRedirect("login");
    }
}
```

Invalida la sessione e reindirizza al login.

Ora, per assicurare che solo utenti loggati accedano a /benvenuto (e magari /logout): **AuthFilter.java:**

```
@WebFilter({"/benvenuto", "/logout"})
public class AuthFilter implements Filter {
    @Override
    public void doFilter(ServletRequest request, ServletResponse response,
FilterChain chain)
            throws IOException, ServletException {
        HttpServletRequest req = (HttpServletRequest) request;
        HttpServletResponse resp = (HttpServletResponse) response;
        HttpSession session = req.getSession(false);
        boolean loggedIn = (session != null && session.getAttribute("user") !=
null);
        if (!loggedIn) {
            resp.sendRedirect(req.getContextPath() + "/login");
        } else {
            chain.doFilter(request, response);
```

```
        }
    }
    // init e destroy possono essere lasciati vuoti se non usati
}
```

Questo filtro è mappato sulle URL di benvenuto e logout. - Controlla se esiste una sessione con attributo "user". - Se no, manda un redirect al login (aggiungendo magari anche contextPath per sicurezza). - Se sì, `chain.doFilter` lascia passare la richiesta alla Servlet target.

In questo modo, anche se qualcuno cercasse manualmente di accedere a /benvenuto senza aver fatto login, verrà rimandato al login.

Abbiamo così illustrato: - Lettura parametri (user, pass). - Uso di sessione e cookie JSESSIONID implicito. - sendRedirect vs forward. - Un filtro di autenticazione. - Semplice gestione di errori (messaggio parametro).

Cookie Example: Aggiungiamo un piccolo dettaglio: potremmo usare cookie per ricordare l'utente. Ad esempio, se volessimo implementare "Ricordami", potremmo in doPost, se login ok e un checkbox "remember" è spuntato:

```
Cookie ck = new Cookie("utente", user);
ck.setMaxAge(60*60*24*30); // 30 giorni
resp.addCookie(ck);
```

E nel doGet del login, precompilare il campo utente se trova il cookie:

```
Cookie[] cookies = req.getCookies();
if(cookies != null){
  for(Cookie c: cookies){
    if(c.getName().equals("utente")){
       out.println("<input name='username' value='"+c.getValue()+"'/>");
       ...
    }
  }
}
```

Ciò come esempio di uso cookie.

Esercizio

Esercizio proposto: Realizzare una piccola applicazione web usando Servlets: - **Scenario:** una mini-app di sondaggio. All'utente viene presentata una domanda con più opzioni (es. "Qual è il tuo linguaggio preferito? Java, Python, C++, etc."). - Implementare una Servlet `SondaggioServlet` che in GET mostra la domanda e un form con radio button delle opzioni, e in POST raccoglie la risposta: - In GET genera HTML con `<form method="POST">` e le opzioni come `<input type="radio" name="linguaggio" value="Java">Java...` ecc. - In POST legge il parametro selezionato, incrementa un contatore per

quella risposta e magari salva in sessione che l'utente ha votato per evitare rivoti. - Dopo il voto, può mostrare i risultati aggregati (percentuali). - Usa la **sessione** per: - Tenere traccia se l'utente ha già votato (es. con un attributo booleano "votato"). - Tenere traccia temporanea dei risultati (per semplicità, puoi memorizzare un Map<String,Integer> come attributo di ServletContext o come static in Servlet, in un contesto reale sarebbe in DB). - Aggiungi un **Filter** `NoCacheFilter` che impedisca il caching della pagina dei risultati, impostando header `Cache-Control: no-cache, no-store` e `Pragma: no-cache` (così che l'utente veda sempre risultati aggiornati e non possa tornare indietro al form se già votato). - Prova a gestire gli output: setta encoding UTF-8, content type, etc. Mostra eventuali accentate nel testo per verificare encoding. - Gestisci se un utente prova ad accedere direttamente ai risultati senza votare: magari se sessione "votato" non c'è, redirect al form.

Questo esercizio copre: - Generazione di contenuto HTML con Servlet (sarebbe più comodo con JSP, ma serve pratica con out.println). - Gestione di parametri (ottenere il valore selezionato). - Uso di sessione per stato utente. - Uso di ServletContext per stato applicativo (i totali voti). - Uso di Filter per controllare header (no cache) o accesso. - (Se si vuole, usage di RequestDispatcher include: ad esempio, potresti fare header e footer HTML come JSP e includerle nella Servlet output).

Domande a scelta multipla

1. Quale metodo della classe HttpServlet viene invocato dal container per gestire una richiesta HTTP GET?
2. A. `doGet(HttpServletRequest req, HttpServletResponse resp)` nella nostra Servlet.
3. B. `service(HttpRequest, HttpResponse)` e basta, senza doGet.
4. C. `doPost(...)`.
5. D. `onGet(Request,Response)` (inesistente).

6. Come si recupera un parametro di query o di form all'interno di una Servlet?

7. A. Usando `request.getParameter("nomeParametro")`, che restituisce il valore come String.
8. B. Leggendo manualmente `request.getInputStream()` e parsando la stringa query.
9. C. Con `System.getenv("nomeParametro")`.
10. D. Tramite `ServletConfig.getParameter()`.

11. A cosa serve il metodo `HttpServletResponse.sendRedirect(String location)`?

12. A. Invia al client una risposta di **redirect** (codice 302/303) indicando di fare una nuova richiesta verso l'URL specificato.
13. B. Inoltra la richiesta corrente internamente ad un'altra Servlet.
14. C. Cambia il metodo della richiesta da GET a POST.
15. D. Riavvia il server su un'altra porta.

16. Qual è un modo corretto per far sì che una Servlet chiami (forward) una JSP per generare la risposta?

17. A. Usando `RequestDispatcher.forward(request, response)` ottenuto da `request.getRequestDispatcher("pagina.jsp")`.
18. B. Usando `response.sendRedirect("pagina.jsp")`.
19. C. Chiamando direttamente il metodo main della JSP.
20. D. Non è possibile chiamare JSP da Servlet.

21. Dove viene memorizzato l'oggetto HttpSession tipicamente?

22. A. Nel server (RAM o persistenza) identificato da un cookie (JSESSIONID) che il client conserva.
23. B. Nel browser dell'utente, come cookie contenente tutti i dati.
24. C. In un campo nascosto su ogni form della pagina.
25. D. In un database per ogni applicazione.

26. Un filtro (`Filter`) con quale meccanismo può bloccare l'esecuzione della Servlet di destinazione?

27. A. Semplicemente non chiamando `chain.doFilter()` e magari producendo una risposta alternativa (es. errore o redirect).
28. B. Lanciando una eccezione che termina la catena.
29. C. Non c'è modo, i filtri non possono mai bloccare, eseguono solo logica aggiuntiva.
30. D. Chiamando `return` nel doFilter dopo il chain.doFilter (questo però blocca post-elaborazione, non la chiamata a monte).

31. Se in una Servlet chiami `request.getSession(false)`, cosa succede se **non** esiste già una sessione per quella richiesta?

32. A. Restituisce `null` invece di crearne una nuova (a differenza di getSession(true) o getSession() che la creerebbero).
33. B. Crea comunque una nuova sessione.
34. C. Lancia IllegalStateException.
35. D. Restituisce un HttpSession vuoto ma con ID casuale.

36. Quali dichiarazioni su ServletContext sono vere?

37. A. ServletContext è un oggetto unico per l'intera applicazione web, usato per parametri di inizializzazione globali e per condividere dati tra Servlet (attributi di contesto).
38. B. Corrisponde alla sessione utente.
39. C. Può essere usato per ottenere un InputStream di risorse statiche presenti nel WAR.
40. D. Si ottiene in una Servlet con `getServletContext()`.

(Seleziona tutte le vere, ad es. A, C, D)

1. Quali intestazioni (header) HTTP vanno impostate per impedire il caching di una pagina dinamica da parte del browser?

2. A. `Cache-Control: no-cache, no-store, must-revalidate` e `Pragma: no-cache` e spesso `Expires: 0`.
3. B. `Cache: false`.
4. C. `Refresh: 0`.
5. D. `Content-Type: no-cache`.

6. In una applicazione web, cosa rappresenta l'URL pattern `*.jsp` nel web.xml?

    - A. Un mapping che applica quella regola (tipicamente a JSP) per qualsiasi richiesta il cui path termina in ".jsp". Il container di default lo mappa al compilatore JSP.
    - B. Un carattere jolly non valido (non si possono usare * nei pattern).
    - C. Indica tutte le Servlet.
    - D. È usato per filtrare parametri di query.

Pagine JSP (JavaServer Pages) (Create Java Web Applications using JSPs)

Teoria

JavaServer Pages (JSP) è una tecnologia Java EE per semplificare la generazione di contenuti HTML (o altri formati testuali) rispetto all'uso di sole Servlet. Una JSP è sostanzialmente una pagina di markup (HTML/XML) con frammenti di codice Java ed espressioni, che viene **tradotta e compilata** in una Servlet dal container al primo utilizzo (o al deploy se precompilata). Da Java EE 7, JSP 2.3 e JSTL sono le versioni rilevanti.

Ciclo di vita JSP: - Il file `.jsp` viene tradotto in una classe Servlet (il container genera codice Java: trasforma HTML in out.write() etc., e include i nostri snippet Java). - Questa servlet generata viene compilata in .class. - Dal punto di vista runtime, poi, segue il ciclo di vita delle Servlet: init, service (che esegue il contenuto generato), destroy. - Gli sviluppatori di solito non vedono la servlet generata a meno di guardare la cartella work del server: però è utile sapere che dietro le quinte c'è quello.

Elementi sintattici JSP: - **Direttive JSP:** Istruzioni per il compilatore JSP. Sintassi: `<%@ directive attribute="..." %>`. Le principali: - `page`: es. `<%@ page import="java.util.*, com.mio.Model" contentType="text/html;charset=UTF-8" %>`. Serve a importare classi, impostare content type, buffering, gestione errori (`errorPage` e `isErrorPage`), e altro. Ad esempio, `session="false"` per disabilitare HttpSession implicita, `isThreadSafe="false"` per indicare al container di serializzare le richieste (o generare SingleThreadModel, ma è deprecato). - `include`: per includere staticamente un file JSP al momento della traduzione. `<%@ include file="header.jsp" %>` inserisce il contenuto di header.jsp come se fosse parte della pagina (utile per header/footer comuni). - `taglib`: per dichiarare una libreria di tag custom (JSTL o altre). Ad esempio: `<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>` per JSTL core. - **Scriptlet:** Blocchi di codice Java arbitrario dentro la pagina. Sintassi: `<% ... codice Java ... %>`. Esempio:

```
<%
    String user = (String) session.getAttribute("user");
    int count = 5;
%>
```

Questo codice verrà inserito tal quale nel metodo _jspService() generato. Va usato con cautela: mescolare troppo Java e HTML può creare confusione (difficile manutenzione). - **Expression (Espressione JSP):** Sintassi `<%= ... %>`. Valuta un'espressione Java e ne inserisce il risultato (toString) nell'output. Esempio: `<%= new Date() %>` stampa la data corrente. Internamente viene convertito in out.print(...). Comodo per inserire variabili. - **Declaration:** Sintassi `<%! ... %>`. Definisce variabili o metodi *a livello di classe* della servlet generata. Ad esempio `<%! private int accessCount = 0; %>` crea un campo statico (o membro istanza) usato magari su più richieste. Va usato con estrema attenzione, perché JSP di default è multithread (un'unica istanza servlet per tutte richieste), quindi le dichiarazioni sono condivise! Un metodo dichiarato qui diventa un metodo della classe, utile per fattorizzare codice all'interno della JSP. - **Commenti JSP:** - Commento JSP (non inviato al client) `<%-- Questo è un commento JSP --%>` - Commento HTML standard `<!-- ... -->` rimane nel sorgente inviato. - **Output di contenuto statico:** HTML puro incluso viene inviato come tale.

Oggetti predefiniti JSP (implicit objects): JSP mette a disposizione alcune variabili predefinite, senza doverle dichiarare: - `request`, `response` (HttpServletRequest/Response correnti). - `session` (HttpSession, a meno che disabilitato con session="false"). - `application` (ServletContext). - `out` (JspWriter, wrapper di PrintWriter per output bufferizzato). - `config` (ServletConfig della servlet JSP). - `pageContext` (PageContext JSP, utility che dà accesso a request, session, etc, e scopi). - `page` (equivalente a `this` per la servlet JSP). - In JSP "error page" c'è anche `exception`. Questi sono disponibili direttamente in scriptlet e expression.

Java code in JSP - perché evitarlo: Storicamente JSP permetteva mettere logica in pagina (scriptlet), ma questo mescola presentazione e logica. Best practice è spostare la logica lato server in Servlet o bean, e mantenere JSP solo per vista. Per questo sono nate JSTL e Expression Language, per ridurre la necessità di scriptlet.

Expression Language (EL): JSP 2.0 ha introdotto l'EL (oggi unificata con JSF). Sintassi: `${expression}` dentro la pagina JSP può accedere a variabili e proprietà in modo semplice, senza Java esplicito. - Può accedere agli attributi in vari scope: page, request, session, application. Ad esempio se in request abbiamo `request.setAttribute("utente", u)`, in JSP possiamo fare `${utente.nome}` per ottenere la proprietà `getNome()` di quell'oggetto. - L'EL supporta semplici operazioni: arithmetic `${count + 1}`, logiche `${param.username == 'admin'}`, chiamare metodi get (sintassi `${obj.metodo()}` se proprietà). - Implicit objects in EL: `${param.nome}` per getParameter, `${sessionScope.xyz}` per attributo di session, `${cookie.nomeCookie.value}` per cookie, `${header.User-Agent}`, `${initParam.paramName}` per context init params. - L'EL in JSP è di default abilitata (in JSTL e taglib). Permette di evitare di scrivere `<%=` e cast ecc. - Non può chiamare metodi arbitrari (a meno di liberare restrizioni, ma di base no, per sicurezza). - Supporta operatori ternari, empty, etc.

Tag Libraries e JSTL: JSTL (JSP Standard Tag Library) è una libreria di tag predefiniti per compiti comuni: - Core `<c:out>`, `<c:if>`, `<c:forEach>`, `<c:choose>` etc, per condizionali e loop. - Formatting

`<fmt:formatDate>`, `<fmt:message>` etc, per formattare numeri, date, localizzazione. - SQL (non molto usata, permette query direttamente da JSP, considerata cattiva pratica in app reali). - XML processing (se devi manipolare XML). - To use JSTL, devi includere la taglib con prefix e uri. Ad esempio:

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
```

- Esempio:

```
<c:if test="${utente != null}">
  <h1>Ciao ${utente.nome}!</h1>
</c:if>
<c:forEach var="item" items="${listaProdotti}">
  <p>Prodotto: ${item.nome} - ${item.prezzo}</p>
</c:forEach>
```

- JSTL e EL lavorano insieme, come si vede.

Includere contenuto e template: - `jsp:include page="header.jsp"` (azione JSP standard) include dinamicamente un'altra JSP al momento della richiesta, permettendo anche di parametri `<jsp:param>`. Differisce da include direttiva: l'include direttiva è statico (compile time), jsp:include è runtime (può cambiare output se la risorsa inclusa dipende da runtime). - `jsp:forward page="altra.jsp"` per fare forward server-side. - Ci sono anche `jsp:useBean`, `jsp:setProperty` ecc. storici per gestire JavaBean come modello di dati a cui JSP lega i form: ormai poco usati (sostituiti da Expression Language e tag JSTL).

Error handling in JSP: - Se in web.xml definisci `<error-page>` per eccezioni, il container può inviare la richiesta a una JSP d'errore. - In JSP d'errore (definita con `<%@ page isErrorPage="true" %>`), l'oggetto implicito `exception` sarà presente e puoi mostrare dettagli. - Oppure manualmente try-catch scriptlet (non consigliato) dentro JSP.

Best Practices: - Evitare scriptlet Java grezzo. Usare JSTL/EL per la logica di presentazione e delegare logica complessa a controllori e modelli (Servlet, backing beans). - Strutturare l'app con pattern MVC: Servlet (Controller) prepara i dati come request attributes, JSP (View) li visualizza usando EL e tag. - Organizzare JSP in WEB-INF (così non accessibili direttamente via URL se devono passare da controller). - Mantenere separati i frammenti ripetuti (header/footer) e includerli. - Ad esempio, con JSTL: avere file JSP di template con `<jsp:include>` e maybe use `<c:import>` for cross context includes if needed.

Esempio pratico di codice

Per dimostrare l'uso di JSP, rifacciamo la parte di vista dell'esercizio del login precedente, ma usando JSP per il form di login e la pagina di benvenuto invece di scrivere HTML nella Servlet.

Avremo: - `login.jsp` – mostra il modulo di login. - `benvenuto.jsp` – mostra messaggio di benvenuto. - Una Servlet `LoginServlet` modificata che in POST controlla credenziali e poi fa forward a benvenuto.jsp invece di produrre direttamente HTML.

login.jsp:

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core" %>
<!DOCTYPE html>
<html><head><title>Login</title></head><body>
    <c:if test="${not empty param.error}">
        <p style="color:red;">Credenziali non valide, riprova.</p>
    </c:if>
    <form action="${pageContext.request.contextPath}/login" method="post">
        Utente: <input type="text" name="username" value="${param.user}"/><br/>
        Password: <input type="password" name="password"/><br/>
        <button type="submit">Login</button>
    </form>
</body></html>
```

Spiegazioni: - Uso JSTL core `<c:if>` per mostrare messaggio errore se `error` param presente. - Form invia in POST a URL /login relativo al context. (Uso `${pageContext.request.contextPath}` per essere sicuro di includere il contesto applicativo se non è root). - Riempio il campo utente con eventuale `param.user` passato (magari se volessi far sì che dopo errore, l'utente immesso rimanga visibile; infatti in sendRedirect potrei aggiungere `&user=...` al query string). - Sto usando param come implicit object param: `${param.error}` prende `request.getParameter("error")`.

benvenuto.jsp:

```
<%@ page contentType="text/html; charset=UTF-8" %>
<!DOCTYPE html>
<html><body>
<%-- Qui assumiamo che l'utente sia in sessione come attributo "user" --%>
    <h1>Benvenuto, <%= session.getAttribute("user") %>!</h1>
    <a href="${pageContext.request.contextPath}/logout">Logout</a>
</body></html>
```

Qui, per mostrare l'utente, ho usato uno scriptlet `<%= session.getAttribute("user") %>` convertendolo in stringa. Avrei potuto passare l'username come attributo di request nella Servlet e usare EL `${username}`. Diciamo che manteniamo come semplice scriptlet per esempio (anche se fare out con scriptlet è sconsigliato se posso farlo in EL; il motivo potrei dire: la user è in sessione, avrei potuto fare `${sessionScope.user}` se quell'attributo fosse una semplice string, il che era). - Il link logout va alla servlet logout.

Servlet LoginServlet (modificata per JSP):

```
@WebServlet("/login")
public class LoginServlet extends HttpServlet {
```

```
    private Map<String,String> utenti = new HashMap<>();
    @Override
    public void init() {
        utenti.put("alice","1234"); utenti.put("bob","abcd");
    }
    @Override
    protected void doGet(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        // gira direttamente la richiesta alla JSP del form
        RequestDispatcher rd = req.getRequestDispatcher("/login.jsp");
        rd.forward(req, resp);
    }
    @Override
    protected void doPost(HttpServletRequest req, HttpServletResponse resp)
            throws ServletException, IOException {
        String user = req.getParameter("username");
        String pass = req.getParameter("password");
        if (user != null && pass != null && pass.equals(utenti.get(user))) {
            req.getSession().setAttribute("user", user);
            // forward a benvenuto.jsp (utente loggato con successo)
            RequestDispatcher rd = req.getRequestDispatcher("/benvenuto.jsp");
            rd.forward(req, resp);
        } else {
            // redirige al form con indicazione errore e conserva magari user
come parametro
            resp.sendRedirect(req.getContextPath() + "/login?error=1&user=" +
                              (user != null ? URLEncoder.encode(user,
"UTF-8") : ""));
        }
    }
}
```

Cosa cambia: - doGet: invece di scrivere HTML, facciamo forward alla JSP. Notare che se la JSP è messa nella root (in contextPath)/login.jsp e la mapping /login colpisce la servlet, l'url pattern è la stessa. Con forward, la servlet chiama la JSP interna. Alternativamente, avremmo potuto mappare la servlet su /doLogin e lasciare /login.jsp per accesso diretto. Dipende da design (si può proteggere JSP in WEB-INF). - doPost: se successo, invece di sendRedirect, facciamo forward interno direttamente alla JSP di benvenuto. Questo significa che l'URL rimane /login, ma mostra contenuto di benvenuto.jsp. A volte meglio fare redirect per evitare il refresh duplicate submission problem. Ma forward consente di mostrare subito la pagina senza altra roundtrip. Up to design. Qui per brevità ho forward. - se fallisce, do sendRedirect come prima, includendo il nome utente nella query string per riempire il form. Uso URLEncoder per sicurezza.

Nota sulla gestione error forward vs redirect: - Con forward su esito positivo, se l'utente fa refresh, re-invia il form (perché la URL è ancora /login POST). Non ideale. Un pattern comune è "POST-Redirect-GET": dopo elaborazione, fare sendRedirect a una pagina di risultato (es. /benvenuto), così refresh non reinvia il post. Quindi avrei potuto fare `resp.sendRedirect("benvenuto.jsp")`. Ma /benvenuto.jsp non è dietro servlet in questo scenario, potrei esporla direttamente. - Miglior design: `/login` servlet fa redirect su `/`

`benvenuto` *servlet* e quella servlet poi forward a JSP. Così le JSP non sono chiamate direttamente. Questo è pattern MVC tipico: Servlet come controller, JSP come view, separate. - Comunque, l'esempio illustra forward.

Uso di JSTL nella benvenuto: Potrei invece di scriptlet usare:

```
<p>Benvenuto, ${sessionScope.user}!</p>
```

Che funziona perché sessionScope dell'EL consente accesso attributi session, e user fu messo li.

Localizzazione con JSTL fmt (opzionale): Ad esempio, potrei usare `<fmt:message>` con risorse. Se volessero nell'esame: JSTL fmt + resource bundles. E.g.:

```
<%@ taglib prefix="fmt" uri="http://java.sun.com/jsp/jstl/fmt" %>
<fmt:setBundle basename="messages" />
<fmt:message key="welcome.message">
  <fmt:param value="${sessionScope.user}" />
</fmt:message>
```

Assumendo un file messages.properties con `welcome.message=Benvenuto, {0}!`.

Errori comuni JSP: - Dimenticare di impostare contentType/charset (risolvibile con direttiva page). - UseBean: meglio injection via servlet, preferire JSTL/EL. - Concorrenza: evitate dichiarazioni con stato mutabile senza sincronizzazione (es. `<%! int count %>` per contare accessi: quell'istanza è condivisa su thread multipli, va sincronizzato o definire page directive `isThreadSafe="false"` per forzare SingleThreadModel semantic). - Null pointer in EL: se qualcosa null, EL restituisce "" (stringa vuota) invece di errore, e `empty` può controllarlo.

Esercizio

Esercizio proposto: Creare un insieme di JSP con JSTL per mostrare una lista di elementi e filtrarla: - Immagina di avere in applicazione una lista di prodotti (nome, categoria, prezzo). Puoi creare questa lista come attributo di application (ServletContext) o derivarla da un bean. - Crea una JSP `prodotti.jsp` che visualizza tutti i prodotti in una tabella HTML. Usa `<c:forEach>` per iterare su ${applicationScope.listaProdotti} (o come la passi) e genera righe tabella. - Aggiungi un form in cima con un campo di testo per filtrare per nome (o categoria). - Quando l'utente invia il filtro (GET o POST a stessa pagina), la JSP applica un filtro in JSTL: ad esempio, `<c:forEach ...><c:if test="${empty param.filter or prodotto.nome contains param.filter}"> ... visualizza ... </c:if></c:forEach>`. Oppure per semplificare, fai che la form chiami una Servlet la quale filtra e mette risultati in request, e la JSP li mostra. - Fornisci la possibilità di cambiare lingua (internazionalizzazione): ad esempio due link "ITA" e "ENG" che aggiungono param locale=it o locale=en. Usa `<fmt:setLocale value="${param.locale}" scope="session"/>` e `<fmt:bundle>` con due file di risorse (prodotti_it.properties, prodotti_en.properties) per i titoli colonne e testi. Usa `<fmt:message>` per visualizzarli. - Così c'è pratica con JSTL core e fmt.

Questo esercizio tocca: - JSTL core tag usage (forEach, if). - Expression language for bean properties (prodotto.nome etc). - Possibly format numbers with `<fmt:formatNumber>` for price (like currency style). - Internationalization via JSTL fmt. - Passing parameters in link and retrieving via param in EL.

Domande a scelta multipla

1. Cosa avviene quando un client richiede una pagina `.jsp` al server per la prima volta (in un container JSP)?
2. A. Il container **traduce** la JSP in una classe Servlet Java, la **compila** in bytecode, poi la esegue (e per successive richieste usa la classe compilata).
3. B. Il container la invia tal quale al client interpretando il codice Java sul client.
4. C. Il container la interpreta riga per riga senza compilare niente.
5. D. La JSP deve essere già compilata manualmente dall'utente prima del deploy.
6. In JSP, a cosa serve l'oggetto implicito `out`?
7. A. È un `JspWriter` usato per scrivere il contenuto di risposta (simile a PrintWriter).
8. B. Rappresenta l'output standard del server (console).
9. C. Serve per leggere l'input dell'utente.
10. D. È deprecato in JSP 2.x, si usa System.out al suo posto.
11. Come si può accedere a un attributo impostato nel request (ad esempio da una Servlet) all'interno di una JSP usando Expression Language (EL)?
12. A. Se la Servlet fa `request.setAttribute("user", u)`, in JSP si può usare `${user}` (il requestScope è ricercato di default per le variabili).
13. B. Tramite `<%= request.getAttribute("user") %>`.
14. C. Non è possibile, bisogna passarlo come parametro query.
15. D. Con `${request.user}`.
16. Qual è la differenza tra un `include` statico (`<%@ include file="header.jsp" %>`) e `<jsp:include page="header.jsp" />` in JSP?
17. A. Il primo include il contenuto di header.jsp al momento della compilazione della JSP (come se fosse un unico file), il secondo invece include la risposta di header.jsp *dinamicamente* ad ogni richiesta (runtime).
18. B. Nessuna differenza, sono sinonimi.
19. C. Il static include funziona solo per HTML, non JSP.
20. D. jsp:include non permette passaggio di parametri, <%@ include sì>.
21. Nell'Expression Language `${utente.nome}`, cosa avviene se `utente` è null (non presente in alcuno scope)?
22. A. L'espressione EL restituisce stringa vuota `""` e non lancia errore.

23. B. Lancia un'eccezione NullPointerException.
24. C. Mostra `${utente.nome}` letteralmente nella pagina perché non viene risolta.
25. D. Stampa "null" come stringa.

26. Per quale motivo è sconsigliato inserire troppa logica Java all'interno di scriptlet JSP?

27. A. Perché rende la pagina difficile da mantenere e mescola logica di business con presentazione; è preferibile spostare la logica in Servlets o usare JSTL/EL per mantenere la JSP pulita.
28. B. Perché il codice Java in JSP è interpretato più lentamente.
29. C. Perché i scriptlet sono deprecati e rimossi nelle versioni moderne (non vero, sono sconsigliati ma ancora supportati).
30. D. Perché non è possibile effettuare debug sul codice scriptlet.

31. Come si itera su una collezione di elementi in una JSP senza usare scriptlet `for` in Java?

32. A. Utilizzando il tag JSTL `<c:forEach>` specificando items e var (e.g., `<c:forEach var="elem" items="${lista}"> ... </c:forEach>`).
33. B. Non si può, serve codice Java.
34. C. Usando un ciclo in EL (EL da solo non ha costrutto loop).
35. D. Con un tag HTML speciale `<loop>`.

36. Quale direttiva JSP va usata per dichiarare l'utilizzo della libreria JSTL Core nella pagina JSP?

37. A. `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`.
38. B. `<%@ include lib="jstl" %>`.
39. C. `<jsp:useLibrary name="JSTL" %>`.
40. D. `<%@ page import="org.jstl.*" %>` (import normale non basta, serve taglib directive).

41. Se volessimo internazionalizzare le stringhe in JSP, quale tag JSTL potremmo usare?

42. A. `<fmt:message key="chiaveMessaggio" />` insieme a `<fmt:setBundle basename="mio.bundle" />` e possibilmente `<fmt:setLocale>` per impostare la lingua.
43. B. `<c:translate>` nel core JSTL (non esiste tale tag).
44. C. Usare scriptlet che chiama ResourceBundle.getString().
45. D. `<jsp:locale code="it_IT"/>` (non esiste così).

46. Che scopo ha l'attributo `isErrorPage="true"` nella direttiva page di una JSP?

    - A. Indica che quella JSP può fungere da pagina di errore (cioè ricevere l'oggetto implicito `exception`), normalmente in seguito a un forward automatico quando un'altra pagina lancia un'eccezione.
    - B. Abilita la generazione di stack trace in output per debug.
    - C. Serve a far sì che la pagina sollevi sempre un errore 500.

- D. Nessuno, è un attributo obsoleto non usato.

Servizi REST con JAX-RS (Implement REST Services using JAX-RS API)

Teoria

REST (Representational State Transfer) è uno stile architetturale per servizi web leggeri basato su HTTP e risorse. Invece di un protocollo rigido come SOAP, con REST si usano le semantiche base di HTTP: metodi (GET, POST, PUT, DELETE, ...), URL significativi che rappresentano risorse, formati di dati leggeri (tipicamente JSON o XML). JAX-RS è la specifica Java EE per costruire facilmente servizi RESTful.

Caratteristiche principali di un servizio REST: - **Rappresenta risorse con URL**: ad es., `/clienti/123` può identificare il cliente con ID 123. - Usa i **metodi HTTP** appropriatamente: - GET per leggere (non modifica, idempotente). - POST per creare (o operazioni non idempotenti). - PUT per creare o aggiornare completamente una risorsa specificata dall'URL (idempotente in teoria). - DELETE per rimuovere. - Altri: PATCH (aggiornamento parziale), HEAD (metadati), OPTIONS (info sulle capabilities). - **Stateless**: il server non conserva lo stato della conversazione HTTP lato server tra richieste (ogni richiesta dovrebbe essere completa di info necessarie, eventualmente tramite token se sessione necessaria). - **Codici di stato HTTP**: si sfruttano per indicare esito (200 OK, 201 Created, 404 Not Found, 400 Bad Request, 500 Server Error, ecc.). - **Formati**: JSON è il de facto standard odierno (in passato XML). Un servizio può spesso *content-negotiate* in base all'header Accept del client se offrire JSON, XML, etc. (JAX-RS fa questo tramite `@Produces` e `@Consumes`). - **HATEOAS** (Hypermedia as the Engine of App State): concetto avanzato dove le risposte includono link navigabili. Non obbligatorio per dire RESTful, ma parte dei principi Roy Fielding.

JAX-RS 2.0 (Java EE 7) fornisce: - **Annotazioni per risorse**: la principale è `@Path` per assegnare un URL path a una classe o metodo. - **Annotazioni HTTP**: `@GET`, `@POST`, `@PUT`, `@DELETE`, `@PATCH` su metodi per legarli ai corrispondenti metodi HTTP. - `@Produces` **e** `@Consumes`: per specificare i MIME types prodotti (es: `application/json`) o consumati dal metodo (es: JSON input). - **Param injection**: - `@PathParam` per estrarre parti dell'URL, - `@QueryParam` per parametri query string, - `@HeaderParam` per header, - `@FormParam` per form data, - `@CookieParam`, etc. - Oppure `@Context` per ottenere oggetti contestuali (UriInfo, HttpHeaders, SecurityContext, Request, ecc.). - **Automatic marshaling**: JAX-RS integra provider per convertire automaticamente oggetti Java in JSON/XML (usando in EE 7 la specifica **JAXB** per XML e tipicamente la libreria **Jackson** o equivalente per JSON via provider come MOXy se presente). - Per JSON: in Java EE 7 c'era JAX-RS integration with e.g. MOXy JSON, ma è un dettaglio implementativo. Comunque, se il client chiede JSON (Accept: application/json) e `@Produces` include JSON, JAX-RS converte l'entity (es. un POJO) in JSON. - I POJO devono avere, per default, annotazioni JAXB o segui convenzioni: JAX-RS in EE7 usare MOXy implicito se c'è, altrimenti JSON-B non c'era fino EE8). - **Response object**: se si vuole controllo fine, c'è classe `Response` builder per impostare codice, header manualmente. Oppure far restituire direttamente un oggetto e lasciare a JAX-RS gestire (che di default manda 200 OK se non diversamente specificato). - **Exception handling**: - Si possono lanciare eccezioni specifiche (es. `WebApplicationException` o sue sottoclassi come `NotFoundException`) per far JAX-RS inviare specifici codici (es. NotFoundException -> 404). - O definire `ExceptionMapper<ExceptionType>` per controllare come convertirle in Response. - **Sub-resource**: possibili path gerarchici con `@Path` su metodi restituenti una classe risorsa, per organizzare meglio endpoints nidificati. - **Filters/Interceptors**: JAX-RS 2 offre Client/Server filter mechanism per logging, auth, etc., e interceptors per entità (rare usage). - **Client API**: JAX-RS 2.0 fornisce API fluente per chiamare servizi REST (ClientBuilder, WebTarget, etc., vedremo).

Esempio minimal:

```
@Path("/citta")
public class CittaResource {
    @GET
    @Produces(MediaType.APPLICATION_JSON)
    public List<Citta> getAll() {
        return servizioListaCitta.tutteLeCitta();
    }

    @GET @Path("/{id}")
    @Produces(MediaType.APPLICATION_JSON)
    public Citta getById(@PathParam("id") int id) {
        Citta c = servizio.find(id);
        if (c == null) {
            throw new NotFoundException();
        }
        return c;
    }

    @POST
    @Consumes(MediaType.APPLICATION_JSON)
    public Response addCitta(Citta c) {
        servizio.save(c);
        URI uriNuova = UriBuilder.fromResource(CittaResource.class)
                    .path("{id}").resolveTemplate("id", c.getId()).build();
        return Response.created(uriNuova).build();
    }
}
```

- La classe ha @Path("/citta") quindi risponde sotto quell'URI base. - `getAll`: `@GET` senza subpath => GET /citta. Restituisce `List<Citta>`. JAX-RS convertirà la lista in JSON array (grazie a provider). - `getById`: `@Path("/{id}")` cattura il segmento successivo come `id`. Restituisce singola Citta. Se non trovata, lancia `NotFoundException()` che JAX-RS intercetta e produce 404. - `addCitta`: `@POST` su /citta. @Consumes indica che aspetta JSON nel body che verrà convertito a un oggetto Citta in input. Il metodo non restituisce l'oggetto, ma crea un Response con status 201 Created e Location header (usando UriBuilder per costruire URI /citta/{nuovoId}).

Configurazione: Bisogna registrare le risorse JAX-RS con l'app: - In Java EE 7 con servlet 3, la presenza di classe con `@Path` in un WAR dovrebbe essere rilevata se definisci anche un `Application` subclass:

```
@ApplicationPath("/api")
public class RestApp extends Application { }
```

Questo definisce il contesto root per JAX-RS (/api) e poi tutte le risorse @Path sono relative a quello. Alternativamente, in web.xml si può aggiungere la servlet JAX-RS (jersey, resteasy etc.) mapping manualmente. Ma con ApplicationPath, no xml needed. - Necessarie le librerie JAX-RS fornite dal server (e.g. Jersey su GlassFish/Payara, Resteasy su WildFly, etc.)

Client JAX-RS: - JAX-RS 2 ha un API per chiamare servizi: Esempio:

```
Client client = ClientBuilder.newClient();
WebTarget target = client.target("http://host/app/api/citta");
List<Citta> lista = target.request(MediaType.APPLICATION_JSON).get(new
GenericType<List<Citta>>(){});
Citta nuova = new Citta("Milano");
Response resp = target.request().post(Entity.json(nuova));
```

- `ClientBuilder.newClient()` crea un client. - `target(...)` definisce l'URL base. - `.request(MediaType.X)` prepara la richiesta e .get() esegue GET. Bisogna specificare la class attesa. Per generics, usano GenericType wrapper. - Per POST, usare `Entity.json(obj)` per corpo JSON. - Il client ha supporto automatico (provider) per convertire JSON in oggetti se i classes same on client or at least structure matches and provider like Jackson available. - The client should be closed when done (client.close()) or use try-with-resources if possible.

Sicurezza REST: - Nessuna built-in sicurezza robusta in JAX-RS, ma integrabile con Java EE Security: e.g. `@RolesAllowed` on methods works if security context configured (like using container-managed security or JWT in EE8). Or implement token logic manually by reading headers. - Often stateless, so authentication via HTTP Basic, API keys, OAuth tokens or JWT is common.

Confronto con SOAP: - REST is simpler for many cases, no WSDL needed, uses JSON (lighter). - But lacks formal contract and advanced QoS (but there are solutions like OpenAPI for documentation, etc). - JAX-RS is quite intuitive with annotations vs JAX-WS.

Errors: - If a param fails to convert (e.g. `id` param not an int), JAX-RS likely returns 400 Bad Request by default. - If method throws generic exception not handled, likely 500.

Esempio pratico di codice

Costruiamo un piccolo servizio REST JAX-RS: gestione di una collezione di **libri** in biblioteca. - Base path: `/api/libri` - Funzioni: ottenere tutti i libri (GET /libri), ottenerne uno per id (GET /libri/{id}), aggiungere libro (POST /libri), aggiornare libro (PUT /libri/{id}), cancellare libro (DELETE /libri/{id}).

Definiamo la classe risorsa e il modello Libro:

```
// Modello Libro
public class Libro {
    private int id;
    private String titolo;
```

```
    private String autore;
    // + costruttore, getters, setters (necessari per JAXB/JSON-B)
}
```

```
@Path("/libri")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class LibroResource {

    private static Map<Integer, Libro> archivio = new ConcurrentHashMap<>();
    private static AtomicInteger sequenzaId = new AtomicInteger(0);

    @GET
    public Collection<Libro> tuttiLibri() {
        return archivio.values();
    }

    @GET
    @Path("/{id}")
    public Libro trovaLibro(@PathParam("id") int id) {
        Libro libro = archivio.get(id);
        if (libro == null) {
            throw new NotFoundException(); // 404
        }
        return libro;
    }

    @POST
    public Response creaLibro(Libro nuovo) {
        int id = sequenzaId.incrementAndGet();
        nuovo.setId(id);
        archivio.put(id, nuovo);
        URI uri = UriBuilder.fromResource(LibroResource.class)
                            .path(String.valueOf(id)).build();
        return Response.created(uri).entity(nuovo).build();
    }

    @PUT
    @Path("/{id}")
    public Response aggiornaLibro(@PathParam("id") int id, Libro aggiornato) {
        Libro esistente = archivio.get(id);
        if (esistente == null) {
            throw new NotFoundException();
        }
        // Aggiorna i campi
        esistente.setTitolo(aggiornato.getTitolo());
```

```
        esistente.setAutore(aggiornato.getAutore());
        return Response.ok(esistente).build();
    }

    @DELETE
    @Path("/{id}")
    public Response eliminaLibro(@PathParam("id") int id) {
        Libro rimosso = archivio.remove(id);
        if (rimosso == null) {
            throw new NotFoundException();
        }
        return Response.noContent().build();
    }
}
```

Spiegazione: - `@Path("/libri")` definisce endpoint base. - `@Produces` e `@Consumes` su classe definiscono default per tutti i metodi (possono essere override su metodo se differente). - Uso una struttura in-memory static (Map) per archiviare libri e un AtomicInteger per ID (in real scenario, DB). - `tuttiLibri()`: restituisce `Collection<Libro>` (JAX-RS trasformerà in JSON array). - `trovaLibro(id)`: se non trova, lancia NotFoundException (JAX-RS mappa a 404). - `creaLibro`: aggiunge a map con nuovo id. Ritorna Response 201 Created e mette location header (Uri builder con fromResource(LibroResource) e .path id). .entity(nuovo) incolla nel body l'oggetto creato (spesso per convenzione si restituisce la rappresentazione creata). - `aggiornaLibro`: se non c'è libro, 404. Se c'è, aggiorna campi (qui semplicemente titolo e autore, id non cambiato). Ritorna 200 OK con libro aggiornato (in body). - `eliminaLibro`: se esiste, rimuove. Se no, 404. Ritorna 204 No Content (no body). - Il Resource class è annotato per JSON, quindi se un client chiede XML, forse risponderebbe comunque JSON perché l'unico produce definito. Si potrebbe aggiungere `MediaType.APPLICATION_XML` e aggiungere annotazioni JAXB a Libro per supportare XML.

Registrazione JAX-RS: Add an application class:

```
@ApplicationPath("/api")
public class RestApp extends Application {
    // empty
}
```

Quindi final endpoints become e.g. GET /appcontext/api/libri etc.

Testing con un client: Possiamo scrivere un main con JAX-RS client:

```
Client client = ClientBuilder.newClient();
WebTarget base = client.target("http://localhost:8080/miaapp/api/libri");

// GET all
List<Libro> libri = base.request().get(new GenericType<List<Libro>>(){});
```

```
System.out.println("Libri iniziali: " + libri.size());

// POST new
Libro l = new Libro();
l.setTitolo("Java EE 7 Guide"); l.setAutore("Oracle");
Response postResp = base.request().post(Entity.json(l));
System.out.println("POST status: " + postResp.getStatus()); // should be 201
Libro creato = postResp.readEntity(Libro.class);
System.out.println("Nuovo libro ID: " + creato.getId());

// GET specific
Libro fetch = base.path(String.valueOf(creato.getId()))
                  .request().get(Libro.class);
System.out.println("Titolo libro creato: " + fetch.getTitolo());

// PUT update
creato.setTitolo("Java EE 7 Complete Guide");
Response putResp = base.path(String.valueOf(creato.getId()))
                  .request().put(Entity.json(creato));
System.out.println("PUT status: " + putResp.getStatus()); // 200 expected

// DELETE
Response delResp = base.path(String.valueOf(creato.getId()))
                  .request().delete();
System.out.println("DELETE status: " + delResp.getStatus()); // 204 expected

client.close();
```

Questo simula CRUD.

Esercizio

Esercizio proposto: Creare un servizio REST JAX-RS per gestione di **todo list**: - Risorsa "tasks", endpoint base `/tasks`. - Un *Task* ha campi: id (int), descrizione (string), completed (boolean). - Implementare: - GET /tasks -> restituisce lista di task (non completati? o tutti). - GET /tasks/{id} -> singolo task (404 se non esiste). - POST /tasks -> aggiunge nuovo task (JSON inviato senza id, server assegna id). - PUT /tasks/{id} -> aggiorna un task esistente (possibile marcare completato). - DELETE /tasks/{id} -> rimuove task. - Provare: - Aggiungere qualche task via POST usando un client (può essere Postman, o scrivere un JAX-RS client o un semplice cURL command). - Recuperare la lista via GET e assicurarsi che il nuovo appare. - Provare un GET di id inesistente (es. /tasks/999) e vedere che ricevi 404. - Aggiornare un task (metti completed=true) e controllare col GET. - Eliminare un task e verificare con list GET che è sparito. - Opzionale: aggiungi query param per filtrare tasks completati vs non: es GET /tasks?completed=true -> filtra solo completati. Usa @QueryParam in JAX-RS for that logic. - Implementa in memoria come sopra (Map). - Usa Response appropriatamente (201 for create, 204 for delete). - Documenta con qualche esempio di JSON: - GET /tasks -> `[{ "id":1, "descrizione":"...","completed":false}, {...}]`. - Indica i content-type da usare: application/json.

Questo esercizio rafforza: - JAX-RS resource coding. - Param injection (path param, query param). - Checking state (404). - restful principles.

Domande a scelta multipla

1. In JAX-RS, quale annotazione si usa per mappare una classe o metodo a un percorso URL specifico?
2. A. `@Path("/resourcePath")`.
3. B. `@WebServlet("/resourcePath")`.
4. C. `@Endpoint("/resourcePath")`.
5. D. `@WebService("/resourcePath")`.

6. Come si indica che un metodo JAX-RS deve rispondere alle richieste HTTP GET?

7. A. Con l'annotazione `@GET` sul metodo.
8. B. Nel nome del metodo (deve iniziare per "get").
9. C. Con `@HttpMethod("GET")` generico oppure `@Path("GET")`.
10. D. Non c'è modo, c'è solo POST di default.

11. Se un metodo JAX-RS restituisce un oggetto POJO e ha `@Produces("application/json")`, cosa fa il runtime?

12. A. Converte automaticamente l'oggetto in JSON (ad esempio tramite una libreria JSON-binding disponibile) e lo invia come response con content-type application/json.
13. B. Solleva un errore perché JAX-RS non sa serializzare un POJO a JSON.
14. C. Cerca un toString() sull'oggetto e lo manda.
15. D. Ignora il @Produces e invia XML.

16. Quale annotazione usi per estrarre una parte del percorso dall'URL e passarla come parametro al metodo?

17. A. `@PathParam("nome")`, dove il nome corrisponde al segnaposto definito in @Path, es. @Path("/utenti/{nome}").
18. B. `@QueryParam`.
19. C. `@UrlParam`.
20. D. `@PathVariable` (questa è di Spring MVC, non di JAX-RS standard).

21. Se vuoi che il metodo JAX-RS accetti input JSON nel corpo della richiesta (ad esempio per un POST che crea risorsa), quale annotazione devi usare?

22. A. `@Consumes(MediaType.APPLICATION_JSON)` sul metodo (o classe).
23. B. `@Accepts("application/json")`.
24. C. `@Input(json=true)`.
25. D. Nessuna, è implicito se usi @POST.

26. Come configuri il base URI di tutte le risorse JAX-RS nella tua applicazione?

27. A. Usando `@ApplicationPath("/api")` su una sottoclasse di `javax.ws.rs.core.Application`.
28. B. Impostando nel web.xml un servlet-mapping per la servlet JAX-RS su /api/*.
29. C. Entrambe le soluzioni sopra sono valide a seconda se preferisci configurazione con annotazione o con descriptor.

30. D. Cambiando il context root dell'intera applicazione sul server.

31. Quale codice di stato HTTP viene tipicamente usato per indicare "risorsa non trovata"?

32. A. **404 Not Found**.
33. B. 204 No Content.
34. C. 500 Internal Server Error.

35. D. 302 Found.

36. In JAX-RS, come restituisci un codice di stato diverso dal 200 predefinito, ad esempio 201 Created dopo un POST?

37. A. Restituendo un oggetto `Response` costruito con `Response.status(201).entity(obj).build()` o usando `Response.created(uri).build()`.
38. B. Lanciando un'eccezione con codice di stato.
39. C. Non è possibile, JAX-RS mette sempre 200 a meno che non sia errore.

40. D. Impostando una variabile globale ResponseCode prima di ritornare.

41. JAX-RS Client API: quale classe o metodo usi per inviare una richiesta GET a un endpoint REST?

42. A. `WebTarget.request().get()` dove WebTarget è ottenuto da `Client.target(url)`.
43. B. `HttpURLConnection` manualmente (non è JAX-RS API).
44. C. `Client.get("url")` statico.

45. D. Non c'è API client in JAX-RS, bisogna usare altri strumenti.

46. In JAX-RS, se un metodo lancia una `NotFoundException`, cosa succede?

    - A. JAX-RS cattura l'eccezione e genera una risposta HTTP 404 Not Found automaticamente al client.
    - B. L'applicazione web viene terminata.
    - C. Il container la ignora e restituisce comunque 200 con corpo vuoto.
    - D. Viene mappata a un 500 Internal Server Error se non gestita.

Applicazioni WebSocket (Create Java Applications using WebSockets)

Teoria

WebSocket (specifica JSR 356 per Java EE 7) è una tecnologia che abilita comunicazione bidirezionale full-duplex tra client e server su una singola connessione persistente, tipicamente per applicazioni realtime (chat, notifiche, IoT, giochi, ecc.). A differenza di HTTP (request/response stateless), WebSocket dopo una handshake iniziale HTTP, stabilisce un canale su cui sia client che server possono inviarsi messaggi in qualsiasi momento, con bassa latenza.

Stile di comunicazione WebSocket: - Basato su frames: testo o binario. Non c'è più concetto di richiesta/risposta ma scambio asincrono di messaggi. - Il protocollo inizia con un handshake HTTP: il client invia un GET con `Upgrade: websocket` e se server supporta, risponde con 101 Switching Protocols. Da lì si passa a WS. - Ha concetto di sessione WebSocket distinta da sessione HTTP.

Java API for WebSocket (JSR 356): - **Endpoint lato server**: - Può essere una classe POJO annotata con `@ServerEndpoint` con un certo URI. - Oppure implementare l'interfaccia Endpoint base (più flessibile). - Più comune: `@ServerEndpoint("/chat")` su classe e definire metodi con annotazioni: - `@OnOpen` – metodo chiamato quando una nuova connessione si apre. - `@OnMessage` – metodo per quando arriva un messaggio dal client (può esserci un metodo per testo e uno per binario separati). - `@OnClose` – quando la connessione si chiude. - `@OnError` – se c'è un errore. - In questi metodi, parametri possono includere `Session` (sessione WS), il messaggio (String per testo, ByteBuffer per binario, o decodificato in un oggetto se usi Decoder), e optional path parameters se endpoint path ha placeholders. - **Endpoint lato client**: - In Java EE, si può anche avere un client endpoint in applicazione Java SE (annotazione `@ClientEndpoint` e simile usage). - Ma spesso il client è JavaScript nel browser usando la WebSocket JS API: `let ws = new WebSocket("ws://host/app/endpointPath"); ws.onmessage = ...; ws.send("hello");`.

Sessione WebSocket: - L'oggetto `Session` rappresenta la connessione tra client e server. - Permette di inviare messaggi dal server al client tramite `session.getBasicRemote().sendText("msg")` o `getAsyncRemote().sendText(...)` per non bloccare. - Può tenere attributi (like user info). - Ha identificatore, info handshake (like query string, user principal if authenticated). - *Attenzione:* WebSockets non hanno integrato di default meccanismi di autorizzazione come sessioni HTTP. Spesso si integrano con cookie sessione o token.

Message handling: - Messaggi testuali: param method `String message` in OnMessage or use a `Reader` if large stream. - Messaggi binari: param `ByteBuffer` or `byte[]` or InputStream. - Puoi definire custom Encoder/Decoder classes to automatically convert between complex objects and text/json or binary format. - Annotate endpoint with `encoders = {MyEncoder.class}, decoders = {MyDecoder.class}`. - For example, a decoder from JSON string to a `ChatMessage` object. - `@OnMessage` can have boolean last part if processing partial messages (fragmented frames, advanced usage).

Lifecycle e Concurrency: - Di default, container creates one instance of endpoint class per client connection (though this is not strictly mandated? Actually default is per-connection if using POJO endpoint).

So each connected session has its own endpoint object, simplifying concurrency (stateful per session). - Thus you can maintain connection-specific state in fields (like username for that session). - If you want to broadcast to all, need a static or application-scoped list of sessions. - If using `@OnMessage` in method, by default it's single-threaded per connection, but multiple different sessions can be handled concurrently with separate instances.

Example usage scenario: Chat room: - Clients connect to `ws://server/app/chat` endpoint. - OnOpen: maybe add session to a list of connected sessions. - OnMessage: when a message comes from one client, server broadcasts it to all (iterate sessions and send). - OnClose: remove session from list.

Error handling: - OnError called if an exception thrown in methods or connection breaks abnormally. Good for logging.

Integration: - You can use dependency injection in endpoint (JEE7 says you can inject EJB, CDI beans in server endpoints). - E.g. `@EJB ChatService chatService;` inside endpoint to use business logic.

WebSocket vs HTTP: - WebSocket is not good for request/response style ephemeral tasks, but great for continuous updates and event-driven comms. - It requires client support (modern browsers do). - If many short-lived connections or infrequent events, HTTP maybe simpler or SSE (Server-Sent Events) can be alternative (unidirectional server push).

Security: - Typically runs either ws:// or wss:// (over TLS). - Authentication often done via initial HTTP handshake - e.g. requiring cookies from logged HTTP session or a token param in URL. - Java server can check `Session.getUserPrincipal()` if integrated with Java EE security (if handshake was done with an authenticated HTTP session, container might propagate user principal). - Or an `EndpointConfig` to validate an API key.

Esempio pratico di codice

Implementiamo un semplice server WebSocket di chat: - Endpoint URI: "/chatroom". - Quando un client si connette, salviamo la sessione in una lista. - Quando un messaggio arriva, lo ribaltiamo a tutti i client (broadcast). - Quando un client disconnette, lo rimuoviamo.

ServerEndpoint class:

```
import javax.websocket.*;
import javax.websocket.server.ServerEndpoint;
import java.util.Set;
import java.util.concurrent.CopyOnWriteArraySet;

@ServerEndpoint("/chatroom")
public class ChatEndpoint {
    // insieme thread-safe delle sessioni connesse
    private static Set<Session> sessions = new CopyOnWriteArraySet<>();

    @OnOpen
```

```
    public void onOpen(Session session, EndpointConfig config) {
        sessions.add(session);
        System.out.println("Nuovo client connesso: " + session.getId());
    }

    @OnMessage
    public void onMessage(String message, Session sender) {
        System.out.println("Ricevuto: " + message + " da " + sender.getId());
        // broadcast a tutti
        for (Session s : sessions) {
            if (s.isOpen()) {
                if (s == sender) {
                    // al mittente posso anche mandare ack differente, qui re-
invio per ora
                }
                try {
                    s.getBasicRemote().sendText(message);
                } catch (Exception e) {
                    e.printStackTrace();
                }
            }
        }
    }

    @OnClose
    public void onClose(Session session, CloseReason reason) {
        sessions.remove(session);
        System.out.println("Sessione " + session.getId() + " disconnessa: " +
reason);
    }

    @OnError
    public void onError(Session session, Throwable error) {
        System.err.println("Errore nella sessione " + session.getId() + ": " +
error);
    }
}
```

Note: - Use `CopyOnWriteArraySet` for thread-safe set (multiple threads may call onMessage concurrently on different sessions). - onMessage simply loops and send to all. If many, could do asynchronously with getAsyncRemote for performance, here fine. - This code echoes the message to all including sender. Real chat might exclude sender (or include with different styling).

Client (JavaScript): In an HTML page:

```
<script>
  let ws = new WebSocket("ws://localhost:8080/myapp/chatroom");
  ws.onopen = () => console.log("Connesso al server WebSocket");
  ws.onmessage = evt => {
    let msg = evt.data;
    let area = document.getElementById("chatArea");
    area.value += "Ricevuto: " + msg + "\n";
  };
  ws.onclose = () => console.log("Connessione chiusa");

  function sendMsg() {
    let input = document.getElementById("msgInput");
    ws.send(input.value);
    input.value = "";
  }
</script>
<textarea id="chatArea" cols="50" rows="10" readonly></textarea><br/>
<input type="text" id="msgInput"/><button onclick="sendMsg()">Invia</button>
```

This creates a WebSocket and attaches event handlers. On pressing button, send the content.

If multiple browser windows open that page, they all join the same "room".

Running: - Deploy server endpoint on server. - Host the HTML page (maybe just static page or served by a Servlet). - When open page, it connects to ws://... (for wss if server uses TLS). - You can open multiple browser windows to simulate multiple clients. - Type a message, see it broadcast.

Encoder/Decoder example: If we had a complex JSON e.g. message object with user and text: We could define:

```
public class ChatMessage {
   private String user;
   private String text;
   // getters/setters
}
```

and have onMessage accept ChatMessage:

```
@OnMessage
public void onMessage(ChatMessage msg, Session session) { ... }
```

Then need a Decoder:

```
public class ChatMessageDecoder implements Decoder.Text<ChatMessage> {
   private Gson gson = new Gson(); // or use JSON-P / JSON-B
   public ChatMessage decode(String s) { return gson.fromJson(s,
ChatMessage.class); }
   public boolean willDecode(String s) { return true; }
   public void init(EndpointConfig config) {}
   public void destroy() {}
}
```

and similarly an Encoder if you want to send as JSON:

```
public class ChatMessageEncoder implements Encoder.Text<ChatMessage> {
   private Gson gson = new Gson();
   public String encode(ChatMessage msg) { return gson.toJson(msg); }
   ...
}
```

Then annotate endpoint:

```
@ServerEndpoint(value="/chatroom", encoders={ChatMessageEncoder.class},
decoders={ChatMessageDecoder.class})
```

So now, onMessage can directly take ChatMessage param (the runtime uses decoder to convert incoming text to ChatMessage).

But for brevity, not included in above code.

Esercizio

Esercizio proposto: Implementare un semplice **gioco in tempo reale** con WebSocket: - Esempio: un "counter condiviso". - Server endpoint: quando un client invia "increment", il server incrementa un contatore globale e manda il nuovo valore a tutti i client. - Ciò significa: - `@ServerEndpoint("/counter")` con OnOpen che manda il valore iniziale al nuovo cliente (così è allineato). - OnMessage: se message == "inc", incrementa contatore e broadcast valore. - OnClose: eventualmente non molto da fare se contatore è globale statico. - UI: pagina HTML con un bottone "Incrementa" che manda il comando, e un elemento che mostra il valore corrente, aggiornato quando server manda messaggio. - Questo prova: - Stato condiviso sul server (contatore static in endpoint). - Broadcast via WebSocket di un valore numerico. - Concorrenza: se più client cliccano simili, il contatore deve incrementare correttamente (so use atomic). - Testare aprendo 2-3 pagine, premendo bottone in uno, e vedere aggiornarsi su tutti.

Estensioni possibili: - Creare un piccolo "canvas drawing" app: broadcast coordinate disegnate. - O un chat come fatto.

Focus su: - Understand how to get Session and sending messages. - Possibly look at Session.getId or RemoteEndpoint.

Domande a scelta multipla

1. Quale annotazione viene utilizzata per definire un endpoint WebSocket lato server in Java?
2. A. `@ServerEndpoint("/path")` sulla classe che gestisce il WebSocket.
3. B. `@WebSocketService`.
4. C. `@Endpoint` (generica non esiste, c'è classe base Endpoint ma annotazione è ServerEndpoint).
5. D. `@WebServlet("/path")` con websocket=true.
6. Con riferimento a WebSocket JSR356, cosa fa un metodo annotato con `@OnMessage`?
7. A. Viene richiamato dal container ogni volta che arriva un messaggio dal client su quella connessione.
8. B. Viene eseguito quando il server invia un messaggio al client.
9. C. Si occupa di gestire la fase di handshake iniziale.
10. D. Viene invocato quando la connessione viene chiusa.
11. Come si invia un messaggio testuale a un client specifico dal lato server?
12. A. Ottenendo l'oggetto Session del client e chiamando `session.getBasicRemote().sendText("messaggio")` all'interno, ad esempio, di @OnMessage o altrove.
13. B. Usando `System.out.println` con il testo (non invia al client).
14. C. Inviando un pacchetto tramite Socket manuale (non necessario con WebSocket API).
15. D. Non è possibile, il server può solo ricevere.
16. Se volessi inviare un oggetto Java come messaggio WebSocket (non solo String o byte[]), come potresti fare?
17. A. Implementando un Encoder/Decoder personalizzato e dichiarandolo nell'annotazione @ServerEndpoint (così l'oggetto può essere trasformato, ad es. in JSON, per l'invio).
18. B. Serializzandolo con ObjectOutputStream e inviando i byte direttamente (possibile se controlli i client, ma non standard per interoperabilità).
19. C. Convertendolo manualmente in stringa JSON dentro OnMessage e chiamando sendText.
20. D. Tutte le opzioni sono valide tecnicamente (A è la via standard prevista dalla spec, C è un workaround manuale comune, B è fattibile se client Java deserializza ma non cross-language).
21. Cosa succede se più client sono connessi al tuo ServerEndpoint e vuoi inviare un messaggio a tutti?
22. A. Devi iterare su tutte le Session attive (magari conservate in una lista thread-safe) e chiamare sendText/sendBinary su ciascuna.
23. B. Il container fornisce un metodo broadcast automatico (non direttamente, devi farlo tu).

24. C. Puoi usare `session.getOpenSessions()` se dentro un OnMessage di una sessione, per ottenere tutte le sessioni connesse a quell'endpoint, e iterare.

25. D. Utilizzare JMS internamente (non necessario per semplici broadcast, fatto in-memory va bene).

26. In un'applicazione web, come può un client JavaScript aprire una connessione WebSocket verso il server?

27. A. Usando `let socket = new WebSocket("ws://servername:port/app/path")` nella pagina web, e poi assegnando socket.onopen, socket.onmessage, etc.
28. B. Tramite AJAX (XMLHttpRequest) persistente.
29. C. Chiamando un metodo WebSocket sul DOM di HTML (no, l'API è WebSocket JS).

30. D. Non si può fare da JavaScript, solo Java può essere client.

31. Quando un client WebSocket chiude la connessione, quale metodo annotato potrebbe essere invocato sul server?

32. A. `@OnClose` con parametri Session e CloseReason.
33. B. `@OnDisconnect`.
34. C. `sessionClosed()` nel Session object.

35. D. `@OnError` (questo è per errori, non per chiusura normale).

36. Quale protocollo viene utilizzato per la comunicazione WebSocket sicura cifrata?

37. A. **wss://** (WebSocket Secure, di solito porta 443 come HTTPS) che è WebSocket sopra TLS.
38. B. https:// (No, https è per HTTP, wss è l'equivalente per WebSocket).
39. C. websocket://

40. D. ssl-ws:// (non standard, giusta è wss).

41. In che modo un endpoint WebSocket in Java EE può accedere a un bean EJB o CDI?

42. A. Può usare l'injection come in altre parti del Java EE (es: `@EJB MyBean bean;` all'interno della classe annotata @ServerEndpoint, supportato dal container se il endpoint è gestito in contesto).
43. B. Non può accedere ad altri componenti Java EE.
44. C. Deve fare lookup JNDI manuale.

45. D. Solo tramite messaggi JMS.

46. Qual è un caso d'uso tipico in cui WebSocket è preferibile rispetto ad HTTP tradizionale?

    - A. Applicazioni in tempo reale con aggiornamenti frequenti dal server al client (es. chat, notifica istantanea, trading, giochi online) grazie alla comunicazione full-duplex persistente.
    - B. Download di file di grandi dimensioni (meglio HTTP per semplici download).
    - C. Form submission occasionali (HTTP sufficiente).
    - D. Pagine web statiche (no).

Applicazioni Web con JSF (JavaServer Faces) (Develop Web Applications using JSF)

Teoria

JavaServer Faces (JSF) è il framework MVC component-based standard di Java EE per costruire interfacce utente web. Versione in Java EE 7 è JSF 2.2. JSF fornisce: - Un modello a componenti UI (con tag xhtml che rappresentano componenti, legati a bean server). - Gestione dello stato (stateful server side or partially client side). - Ciclo di vita definito per elaborare richieste (conversione, validazione, aggiornamento modello, invocazione logica, render view). - Navigazione configurabile (via outcome string o config). - Integrazione con CDI per backing beans (@Named beans). - Supporto per AJAX mediante f:ajax tag.

Architettura e componenti: - Pagine scritte in **Facelets (XHTML)** con tag libraries: - `h:` (HTML Basic components, e.g. h:form, h:inputText, h:dataTable, h:commandButton, etc). - `f:` (core JSF actions: f:actionListener, f:convertDateTime, f:ajax). - `ui:` (templating, ui:composition, ui:insert, etc). - **Managed Beans** come controller/model backing: - In JSF 2.x si può usare `@ManagedBean` (JSF proprietary) o meglio CDI: `@Named @RequestScoped` etc. - These beans have properties with getters/setters that UI components bind to (via EL). - **Expression Language (EL)**: used in page to bind values and invoke actions: - Value binding: `value="#{utenteBean.nome}"` binds input field to bean property. - Method binding: `action="#{loginBean.login}"` calls login() method on bean when button clicked. - **Lifecycle JSF** (per ogni richiesta): 1. **Restore View**: JSF builds (or retrieves) component tree for page. 2. **Apply Request Values**: populates components with request parameters (setting local values). 3. **Process Validations**: each component with validator (or required fields) is validated. If any validation fails, JSF goes to render response (display error messages), skipping update and invocation. 4. **Update Model Values**: if all validations passed, the local values are moved to model (i.e., bean properties are set). 5. **Invoke Application**: if a form submission triggered an action (like a button with action method), call that method on bean. 6. **Render Response**: produce the output (if coming from a fresh GET or after processing an action). If validation failed earlier, we also come here but do not update model or call action. - There's the concept of **navigation**: - The action method can return a string outcome which maps to a certain page. - Or use `<navigation-rule>` in faces-config.xml or implicit nav where outcome "success" goes to "success.xhtml". - **FacesContext** holds info about the context (similar to request context for JSF). - **Messages**: - Use h:message or h:messages to display conversion/validation errors or messages added by app. - If a field fails validation, a FacesMessage attached to component is shown by h:message for that component. - **Validation and Conversion**: - Built-in converters (dates, numbers). - `f:convertDateTime` and `f:convertNumber` to specify formatting. - Custom converters implement javax.faces.convert.Converter. - Validators: `f:validateLength`, `f:validateRegex` etc built-in; or custom by implementing javax.faces.validator.Validator. - Bean Validation (JSR 303) integrates too: if bean property has @NotNull etc, JSF will use that automatically to validate after convert but before update model? Actually, with <f:validateBean> or default in newer JSF 2.3 automatically. - **Scopes** for beans: - RequestScoped (for each HTTP request - good for ephemeral actions). - ViewScoped (lasts for the JSF view across requests e.g. if partial refresh or navigating via GET stays same page). - SessionScoped (persist across session). - ApplicationScoped (global). - FlowScoped (JSF 2.2 has concept of flows, wizards). - ConversationScoped (if using CDI conversation). - In JSF, if using @ManagedBean, there's @ViewScoped etc. If using CDI, they introduced @ViewScoped, etc. Could also use CDI's @ConversationScoped or others. - **Templating**: - Facelets allow template pages: define <ui:composition template="base.xhtml"> with <ui:define name="content"> etc. In base template use

<ui:insert name="content"/>. - This avoids repeating header, menu, footer etc. - **AJAX in JSF**: - Very easy: include `<f:ajax>` inside components to partial submit and partial update. - Example:

```
<h:inputText value="#{bean.name}">
  <f:ajax event="keyup" render="nameOutput"/>
</h:inputText>
<h:outputText id="nameOutput" value="#{bean.name}" />
```

This would update the outputText with id nameOutput on each key press via AJAX. - `render` attribute lists IDs (or @form, @all etc) to update. - `execute` attribute can narrow what fields to submit (default @this, meaning only triggering component). - So JSF has built-in partial processing and updating via f:ajax. - There are also JSF component libraries (PrimeFaces, etc) that provide richer Ajax components. - **Localization**: - Use <f:loadBundle basename="messages" var="msg"/> to load resource bundle, then use e.g. #{msg['welcome.title']} in output. - Or use h:outputFormat or f:convertNumber with locale.

JSF vs JSP: - JSF is an abstraction above JSP. Historically JSF 1.x used JSP, but JSF 2 uses Facelets as default. - Facelets is more powerful for templating and composites. - JSP still can embed JSF tags, but Facelets is recommended.

JSF integrated with CDI: - In Java EE 7, a CDI bean with `@Named` is recognized in EL out-of-the-box (the unification of EL context). - So you can just use CDI beans (with proper scope annotations) instead of using @ManagedBean. (But if using @ViewScoped with CDI, needed CDI extension in EE7; in EE8 they standardized that). - Still, exam likely expecting understanding of general concept.

JSF UI component model: - Each UI component (like h:inputText) has value-binding, validators, converters, attributes like required etc. - They maintain local value states. - There are events too: valueChangeEvent, actionEvent etc, and possibility to attach listeners or use immediate attribute to influence when executed (immediate = true means process during apply request phase, skipping validation of others).

Navigation: - If action returns a string "page2", by default it looks for page2.xhtml in same folder (or defined in faces-config). - Or can use faces-config to map logical outcome to physical page.

Example brief: Let's imagine a small form:

```
<h:form>
  <h:outputLabel for="name" value="Nome:" />
  <h:inputText id="name" value="#{userBean.name}" required="true" />
  <h:message for="name" />

  <h:commandButton value="Salva" action="#{userBean.save}" />
  <h:messages globalOnly="true"/>
</h:form>
```

- There's a form, outputLabel (associates with input by id). - inputText bound to userBean.name, required true means if empty triggers validation error with default message. - h:message for name shows specific error if any. - commandButton triggers userBean.save() in bean (assuming userBean is e.g. @SessionScoped or @RequestScoped). - h:messages globalOnly shows any global messages (like success). - userBean.save could add a FacesMessage global or return a navigation outcome.

Back-end bean:

```
@Named
@RequestScoped
public class UserBean {
   private String name;
   // + getter, setter
   public String save() {
      if (name != null) {
         // perhaps do some saving logic
         FacesContext.getCurrentInstance().addMessage(null,
             new FacesMessage("Dati salvati per " + name));
         return "confermato"; // navigate to confermato.xhtml maybe
      }
      return null; // stay on same page
   }
}
```

- It uses FacesContext to add a global message (clientId null means global). - Then navigation might go to "confermato.xhtml".

How error flows: - If name empty and is required, validation fails at Process Validations, sets a message for component "name" and skip to Render Response. The page reloads showing message via h:message.

JSF specific terms: - Backing bean = bean associated with UI form to handle input and actions. - Managed property (to inject one bean into another via faces-config or @ManagedProperty). - Facelet composition / Composite components (custom re-usable components, e.g. <ez:myComp ...> implemented as a composition with interface definition). - Partial state saving: by default JSF does save component tree state between requests for postbacks so it can restore if user navigates away/back or for the view.

Esempio pratico di codice

Creiamo una semplice applicazione JSF: Scenario: **Registrazione utente**. - Una pagina "register.xhtml" con un form chiedendo: username, email, password (due volte per conferma). - Un bean `RegisterBean` che gestisce i campi e l'azione di registrazione. - Validazioni: - Username required, min lunghezza 3. - Email required, pattern email. - Password required, e match con conferma. - Usare valiatori JSF e uno custom for confirm password.

register.xhtml:

```
<!DOCTYPE html xmlns="http://www.w3.org/1999/xhtml"
  xmlns:h="http://xmlns.jcp.org/jsf/html"
  xmlns:f="http://xmlns.jcp.org/jsf/core">
<h:head>
   <title>Registrazione</title>
</h:head>
<h:body>
<h:form>
  <h3>Form di Registrazione</h3>

  <h:panelGrid columns="2">
    <h:outputLabel for="username" value="Username: *" />
    <h:inputText id="username" value="#{registerBean.username}" required="true"
label="Username">
       <f:validateLength minimum="3" />
    </h:inputText>
    <h:message for="username" style="color:red"/>

    <h:outputLabel for="email" value="Email: *" />
    <h:inputText id="email" value="#{registerBean.email}" required="true"
label="Email">
       <f:validateRegex pattern="[^@]+@[^\\.]+\\..+" />
    </h:inputText>
    <h:message for="email" style="color:red"/>

    <h:outputLabel for="password" value="Password: *" />
    <h:inputSecret id="password" value="#{registerBean.password}"
required="true" label="Password" />
    <h:message for="password" style="color:red"/>

    <h:outputLabel for="confPassword" value="Conferma Password: *" />
    <h:inputSecret id="confPassword" value="#{registerBean.confirmPassword}"
required="true" label="Conferma Password" />
    <h:message for="confPassword" style="color:red"/>
  </h:panelGrid>

  <h:commandButton value="Registrati" action="#{registerBean.register}" />
  <h:messages globalOnly="true" style="color:green"/>
</h:form>
</h:body>
</html>
```

Spiegazione: - Abbiamo un form con 4 campi: - username: `required="true"` (so not empty) e f:validateLength min 3. - email: required true e regex basic pattern. - password e confirmPassword: required true. (no additional validation here yet, but we'll handle matching in bean). - Each has an associated h:message for field (displays error message if validation fails, style red). - Label attribute on input helps to

show in message like "Username: must be at least 3" because JSF default messages often include label. - A panelGrid just arranges label and field in two columns. - CommandButton triggers registerBean.register method. - Global messages (like success after registration) displayed with h:messages globalOnly in green.

RegisterBean.java:

```
@Named
@RequestScoped
public class RegisterBean {
    private String username;
    private String email;
    private String password;
    private String confirmPassword;

    // getters and setters for all

    public String register() {
        // Custom validation: check if password and confirmPassword match
        if (password != null && !password.equals(confirmPassword)) {
            FacesContext ctx = FacesContext.getCurrentInstance();
            // Add error message associated with confirmPassword field:
            ctx.addMessage("confPassword",
                new FacesMessage(FacesMessage.SEVERITY_ERROR,
                "Le password non coincidono", null));
            // by adding message, JSF will redisplay page and show this error,
            // and skip navigation because we return null:
            return null;
        }

        // If validations passed:
        // Perhaps call a service to save user to DB (omitted).

        // Add a global success message
        FacesContext.getCurrentInstance().addMessage(null,
            new FacesMessage(FacesMessage.SEVERITY_INFO,
            "Registrazione completata con successo!", null));

        // Stay on the same page to show the success message, or navigate to
success page:
        return "register";
    }
}
```

Note: - We verify if password equals confirm. If not, we add a message specifically for confPassword field (with clientId "confPassword" matching the inputText id). This will show up via h:message for confPassword. Also note we not doing a navigation, just return null to remain on same page (so user can correct). - If okay,

maybe we store user (skip actual DB call here). - Add a global message "successfully registered". - Then return "register" as outcome or null. If we have a success page, we could navigate there. But for demonstration, staying on same with message.

Alternatively, we could have created a custom Validator class for password confirmation and attach with f:validator on confPassword field. But doing in action method is fine for demonstration.

Flow: - On initial GET /register.xhtml: JSF displays form (Restore view etc). - User fills data and submits: - JSF Apply Request: sets fields local values. - Process validations: required and regex, length are done. If any fail, error message is set and skip rest. - If built-ins pass, Update Model: sets RegisterBean fields (bean is request scope so new each request? Actually, request scope means the bean will be constructed at start of this post request since it's referenced by form. Values get set now). - Invoke Application: call registerBean.register(). The bean method does additional validation for passwords. - If mismatch, it adds error message and returns null. When a message is added and outcome null, JSF will jump to Render Response of same page, showing the message. Because we didn't call FacesContext.responseComplete, so it re-renders with messages. - If success, it adds global message and returns "register". If configured for navigation, it could go to some page. But since outcome equals current page name, it just reloads form page (recreating bean because new request scope). But global message is attached and will display because we added it. Actually careful: It's added to this request's context, it will show on the result page if it's the same lifecycle. But after navigation, if it was new request, might need to preserve message (there is flash scope for that). In our case, returning "register" likely triggers new GET of register.xhtml (unless faces-config says not do redirect maybe). - Could use faces-config navigation-case from "register" to "register.xhtml" with redirect false so that message still there, or use flash context. - Simpler approach: `return null;` to stay on same request and show message. But then if refresh page after, form re-submission? Could then incorporate flash.

This intricacy aside, concept is shown.

If more advanced: We might separate success page.

Validation messages: - Provided by JSF API in default locale if needed like "Username: Validation Error: Value is required." etc. The label attribute and messages are default for required and length etc.

We could also show usage of templates if needed: But let's skip due complexity.

Esercizio

Esercizio proposto: Creare una piccola applicazione JSF per gestione di **articoli di blog**: - Pagine: - listaArticoli.xhtml: mostra tabella di articoli (titolo, autore, snippet), con link "vedi" per dettaglio, e link "nuovo articolo". - dettaglioArticolo.xhtml: mostra contenuto completo di un articolo selezionato. - nuovoArticolo.xhtml: form per inserire nuovo articolo (titolo, testo, autore). - Beans: - ArticleService (app scope or app-managed) that stores articles list (simulate DB). - ArticleListBean (request/view scope): to fetch list from service. - ArticleDetailBean: to load selected article (maybe by id param). - ArticleFormBean: for new article form, with action to save via service. - Flow: - On list page, h:dataTable bound to list in ArticleListBean. CommandLink or h:link to detail page with article id param. - On detail page, get param id (f:viewParam or use viewAction) to load article in bean. - On new page, form with fields, with validation (title required, content required min length, etc). On submit, add to service and navigate back to list with success message.

- Use facelets templates: - template.xhtml with common header (maybe a nav menu or title). - each page uses <ui:composition template="template.xhtml"> with defines for content and title.

This covers: - h:dataTable usage. - Navigation via h:link or faces-config. - f:viewParam usage to pass an id from list to detail backing bean. - Basic validation (like required). - Possibly fileupload if advanced (skip maybe). - Flash scope: when new article saved, add message then redirect to list, ensure message persists (FacesContext.getExternalContext().getFlash().setKeepMessages(true)).

Focus on demonstrating JSF tags and bean integration: - core tags (h:form, h:inputText, h:commandButton, h:messages). - data table usage for list. - navigation outcomes or explicit pages.

Domande a scelta multipla

1. In JSF, cosa rappresenta un "backing bean"?
2. A. Un JavaBean (gestito da JSF o CDI) che contiene lo stato e la logica per interagire con una pagina JSF, tipicamente con proprietà a cui i componenti UI sono legati e metodi di azione per eventi (es. pulsanti).
3. B. Un file di configurazione XML per la navigazione.
4. C. Il file .xhtml della vista.
5. D. Un EJB session bean usato internamente da JSF (non necessariamente, può essere POJO CDI).
6. Quale tag JSF useresti per rendere un campo di input obbligatorio e mostrare un messaggio di errore se lasciato vuoto?
7. A. `h:inputText` con attributo `required="true"` e un `h:message for="campoId"` associato per visualizzare l'errore.
8. B. `f:validateRequired` dentro l'input (c'è anche questo in JSF2, optional, but required attr is simpler).
9. C. `h:outputText required="true"`.
10. D. `f:validateNotNull`.
11. In JSF, come colleghi il valore di un componente UI a una proprietà di un bean?
12. A. Tramite l'attributo `value="#{nomeBean.nomeProprieta}"` sul componente (per esempio h:inputText).
13. B. Mettendo lo stesso nome al componente e alla proprietà.
14. C. Con un tag `<f:bind>` (non esiste).
15. D. Non si può, i valori vanno gestiti via request.getParameter come in Servlet (no, JSF fa binding automatico).
16. A cosa serve `<h:commandButton action="#{bean.metodo}" />`?

17. A. Renderizza un pulsante che quando premuto invoca il metodo `metodo()` del bean specificato, dopo aver passato con successo il ciclo di validazione e aggiornamento modello. Il ritorno del metodo può determinare la navigazione.
18. B. Esegue subito il metodo via AJAX senza validare input (no, esegue dopo process validations normally unless immediate true).
19. C. È solo un pulsante statico senza azione.
20. D. Richiede un attributo onClick per funzionare.

21. Se un metodo di azione JSF nel backing bean ritorna la stringa "successo", cosa succede?

22. A. JSF cercherà una pagina di navigazione corrispondente a "successo" (ad esempio successo.xhtml) secondo le regole di navigazione (implicite o configurate).
23. B. Ignora il valore e rimane sulla stessa pagina.
24. C. Interpreta "successo" come un messaggio di esito da mostrare.
25. D. Genera un'eccezione se non trova una pagina chiamata "successo.xhtml".

26. Durante il ciclo di vita JSF, in quale fase vengono eseguite le conversioni e validazioni dei campi input?

27. A. Nella fase **Process Validations** (dopo aver applicato i valori ai componenti e prima di aggiornare il modello).
28. B. Immediatamente al caricamento pagina.
29. C. Dopo l'invocazione del metodo di azione.
30. D. Non c'è una fase specifica, avvengono casualmente.

31. In JSF, qual è la differenza tra `h:message` e `h:messages`?

32. A. `h:message` mostra il messaggio di errore relativo a un singolo componente (identificato dall'attributo for), mentre `h:messages` può mostrare tutti i messaggi (globali e di vari componenti) eventualmente accumulati.
33. B. Nessuna, sono alias.
34. C. h:message serve per output di debug.
35. D. h:messages può essere usato una sola volta per pagina.

36. Se volessi definire un layout comune per più pagine JSF (ad esempio header e footer riutilizzabili), quale funzionalità JSF useresti?

37. A. **Facelets Templating**: creare una pagina template.xhtml e far sì che le altre pagine la utilizzino tramite `<ui:composition>` e `<ui:define>`.
38. B. JSP include directive.
39. C. Duplicare il codice header/footer in ogni pagina.
40. D. Frameset HTML (superato, non usato in JSF).

41. Quale annotazione CDI/JSF potrebbe essere usata per rendere un bean JSF legato alla vita di una specifica pagina vista (quindi sopravvive a multiple request Ajax/postback sulla stessa pagina ma non oltre)?

42. A. `@ViewScoped` (dal pacchetto javax.faces.bean per JSF managed bean, oppure javax.faces.view.ViewScoped per CDI in EE7 con estensione).
43. B. `@RequestScoped`.
44. C. `@SessionScoped`.
45. D. `@ApplicationScoped`.

46. In JSF, come si può integrare la validazione Bean Validation (JSR 303) con i campi di input?

    - A. Se il bean di backing ha annotazioni come `@NotNull`, `@Size` sulle proprietà, JSF per default (soprattutto in JSF 2.2/2.3) esegue la validazione Bean Validation dopo le proprie (oppure usando <f:validateBean>). I messaggi di violazione vengono aggiunti come FacesMessages automaticamente.
    - B. Non si integra, bisogna fare manualmente nel metodo di azione.
    - C. Solo tramite un Validator custom che chiama ValidatorFactory.
    - D. JSF non supporta Bean Validation.
